Deep Learning Applied to Computational Mechanics:
A Comprehensive Review, State of the Art, and the Classics

Loc Vu-Quoc,[1,✉] **Alexander Humer**[2]

[1,✉] Aerospace Engineering, University of Illinois at Urbana-Champaign, IL 61801, USA ● ✉ vql@illinois.edu

[2] Institute of Technical Mechanics, Johannes Kepler University, A-4040 Linz, Austria ● alexander.humer@jku.at

A classic never dies.

*"We must welcome the future, for it soon will be the past,
and we must respect the past, for it was once all that was humanly possible."*

George Santayana

ABSTRACT

Three recent breakthroughs due to AI in arts and science serve as motivation: An award winning digital image, protein folding, fast matrix multiplication. Many recent developments in artificial neural networks, particularly deep learning (DL), applied and relevant to computational mechanics (solid, fluids, finite-element technology) are reviewed in detail. Both hybrid and pure machine learning (ML) methods are discussed. Hybrid methods combine traditional PDE discretizations with ML methods either (1) to help model complex nonlinear constitutive relations, (2) to nonlinearly reduce the model order for efficient simulation (turbulence), or (3) to accelerate the simulation by predicting certain components in the traditional integration methods. Here, methods (1) and (2) relied on Long-Short-Term Memory (LSTM) architecture, with method (3) relying on convolutional neural networks. Pure ML methods to solve (nonlinear) PDEs are represented by Physics-Informed Neural network (PINN) methods, which could be combined with attention mechanism to address discontinuous solutions. Both LSTM and attention architectures, together with modern and generalized classic optimizers to include stochasticity for DL networks, are extensively reviewed. Kernel machines, including Gaussian processes, are provided to sufficient depth for more advanced works such as shallow networks with infinite width. Not only addressing experts, readers are assumed familiar with computational mechanics, but not with DL, whose concepts and applications are built up from the basics, aiming at bringing first-time learners quickly to the forefront of research. History and limitations of AI are recounted and discussed, with particular attention at pointing out misstatements or misconceptions of the classics, even in well-known references. Positioning and pointing control of a large-deformable beam is given as an example.

KEYWORDS

Deep learning, breakthroughs, network architectures, backpropagation, stochastic optimization methods from classic to modern, recurrent neural networks, long short-term memory, gated recurrent unit, attention, transformer, kernel machines, Gaussian processes, libraries, Physics-Informed Neural Networks, state-of-the-art, history, limitations, challenges; *Applications to computational mechanics*; Finite-element matrix integration, improved Gauss quadrature; Multiscale geomechanics, fluid-filled porous media; Fluid mechanics, turbulence, proper orthogonal decomposition; *Nonlinear-manifold model-order reduction*, autoencoder, hyper-reduction using gappy data; control of large deformable beam.

SUMMARY

Three representative applications of deep learning in computational mechanics—involving numerical integration for finite element, complex constitutive model in solid mechanics, and proper orthogonal decomposition in fluid mechanics—are reviewed in detail, and used as motivation for a further in-depth review of some key technologies of deep learning, building up from the basics to the state of the art, focusing on, to the extent possible, the most recent papers that had an important impact in the field.

Both static and dynamic time-dependent problems are discussed. Discrete time-dependent problems, as a sequence of data, can be modeled with recurrent neural networks, using the 1997 classic architecture such as Long Short-Term Memory (LSTM), but also the recent 2017-18 architectures such as transformer, based on the concept of attention, all of which are discussed in detail. Continuous recurrent neural networks originally developed in neuroscience to model the brain and the connection to their discrete counterparts in deep learning are also discussed in detail.

For training networks—i.e., finding optimal parameters that yield low training error and lowest validation error—both classic deterministic optimization methods (using full batch) and stochastic optimization methods (using minibatches) are reviewed in detail, and at times even derived. Deterministic gradient descent with classical line search methods, such as Armijo's rule, were generalized to add stochasticity. Detailed pseudocodes for these methods are provided. The classic stochastic gradient descent (SGD), with add-on tricks such as momentum, step-length decay, cyclic annealing, weight decay are presented, often with detailed derivations.

Step-length decay is shown to be equivalent to simulated annealing using stochastic differential equation equivalent to the discrete parameter update. A consequence is to increase the minibatch size, instead of decaying the step length. In particular, we obtain a new result for minibatch-size increase.

Highly popular adaptive step-length (learning-rate) methods are discussed in a unified manner, which covers AdaGrad, RMSProp, the "immensely successful" Adam and its variants, through to the recent AdamW.

Overlooked in (or unknown to) other review papers and even well-known books on deep learning, exponential smoothing of time series, the key technique of adaptive methods, and originating from the field of forecasting dated since the 1950s, is carefully explained.

Particular attention is given to a recent criticism of adaptive methods, revealing their marginal value for generalization, compared to good old SGD with effective initial step-length tuning and decay. The results were confirmed in three recent independent papers.

Kernel machines, including Gaussian processes, a most important class of non-parametric modeling with accurate uncertainty estimates, are introduced to sufficient details to prepare for more advanced works on networks with infinite width that constitute the 2021 breakthrough in computer science.

Applications of deep learning in computational mechanics often aim at reducing computational cost, which naturally connects to the field of (nonlinear) model-order reduction (MOR). We review how LSTM networks were trained to predict the rate-dependent constitutive response in multi-scale problem of porous media and how they were used as time-integrators for reduced order models (ROMs) inferred from highly-resolved direct numerical simulations of turbulent flows. Autoencoders based on shallow networks provide effective means in nonlinear manifold-based MOR and hyper-reduction method built on top.

A rare feature of the present paper is in a detailed review of some important classics to connect to the relevant concepts in modern literature, sometimes revealing misunderstanding in recent works, which was likely due to a lack of verification of the assertions made with the corresponding classics. For example, the first artificial neural network, conceived by Rosenblatt (1957) [1], (1962) [2] had 1000 neurons, but was reported as having a single neuron. Going beyond probabilistic analysis, Rosenblatt even built the Mark I computer to implement his 1000-neuron network. Another example is the "heavy ball" method, for which everyone referred to [3], but who more precisely called the "small heavy sphere" method. Others were quick to dismiss classical deterministic line-search methods that have been generalized to add stochasticity for network training. Unintended misrepresentation of the classics would mislead first-time learners, and unfortunately even seasoned researchers who used second-hand information from others, without checking the original classics themselves.

The experiments in the 1950s that discovered the rectified linear behavior in neuronal axon, modeled as

a circuit with a diode, together with the use of the rectified linear activation function in neural networks in neuroscience years before being adopted for use in deep-learning networks, are reviewed.

The use of Volterra series to model the nonlinear behavior of neuron in term of input and output firing rates, leading to continuous recurrent neural networks is examined in detail. The linear term of the Volterra series is a convolution integral that provides a theoretical foundation for the use of linear combination of inputs to a neuron, with weights and biases.

A goal of this in-depth review is not only to provide the state of the art for computational-mechanics readers with some familiarity of deep-learning networks, but also with first-time learners in mind, by developing relevant fundamental concepts from the basics. Moreover, for the convenience of the readers, detailed references are provided, e.g., page numbers in thick books, links to online references and open reviews where available.

TABLE OF CONTENTS

1 Opening remarks and organization

Breakthroughs due to AI in arts and science. On 2022.08.29, Figure 1, an image generated by the AI software Midjourney, became one of the first of its kind to win first place in an art contest.[1] The image author signed his entry to the contest as "Jason M. Allen via Midjourney," indicating that the submitted digital art was not created by him in the traditional way, but under his text commands to an AI software. Artists not using AI software—such as Midjourney, DALL.E 2, Stable Diffusion—were not happy [4].

In 2021, an AI software achieved a feat that human researchers were not able to do in the last 50 years in predicting protein structures quickly and in a large scale. This feat was named the scientific breakthough of the year; Figure 2, left. In 2016, another AI software beat the world grandmaster in the Go game, which is described as the most complex game that human ever created; Figure 2, right.

On 2022.10.05, DeepMind published a paper on breaking a 50-year record of fast matrix multiplication by reducing the number of multiplications in multiplying two 4×4 matrices from 49 to 47 (with the traditional method requiring 64 multiplications), owing to an algorithm discovered with the help of their AI software AlphaTensor [8].[2] Just barely a week later, two mathematicians announced an algorithm that required only 46 multiplications.

Since the preprint of this paper was posted on the arXiv in Dec 2022 [9], there have been considerable excitements and concerns about ChatGPT—a large language-model chatbot that can interact with humans in a conversational way—which would be incorporated into Microsoft Bing to make web "search interesting again, after years of stagnation and stasis" [10], whose author wrote "I'm going to do something I thought I'd never do: I'm switching my desktop computer's default search engine to Bing. And Google, my default source of information for my entire adult life, is going to have to fight to get me back." Google would release its own answer to ChatGPT called "Bard" [11]. The race is on.

---

[1] See also the Midjourney Showcase, Internet archived on 2022.09.07, the video Guide to MidJourney AI Art - How to get started FREE! and several other Midjourney tutorial videos on Youtube.
[2] Their goal was of course to discover fast multiplication algorithms for matrices of arbitrarily large size. See also "Discovering novel algorithms with AlphaTensor," DeepMind, 2022.10.05, Internet archive.

Figure 1: *AI-generated image won contest* in the category of Digital Arts, Emerging Artists, on 2022.08.29 (Section 1). "Théâtre D'opéra Spatial" (Space Opera Theater) by "Jason M. Allen via Midjourney", which is "an artificial intelligence program that turns lines of text into hyper-realistic graphics" [4]. Colorado State Fair, 2022 Fine Arts First, Second & Third. (Permission of Jason M. Allen, CEO, Incarnate Games)

Audience. This review paper is written by mechanics practitioners to mechanics practitioners, who may or may not be familiar with neural networks and deep learning. We thus assume that the readers are familiar with continuum mechanics and numerical methods such as the finite element method. Thus, unlike typical computer-science papers on deep learning, notation and convention of tensor analysis familiar to practitioners of mechanics are used here whenever possible.[3]

For readers not familiar with deep learning, unlike many other review papers, this review paper is not just a summary of papers in the literature for people who already have some familiarity with this topic,[4] particularly papers on deep-learning neural networks, but contains also a tutorial on this topic aiming at bringing first-time learners (including students) quickly up-to-date with modern issues and applications of deep learning, especially to computational mechanics.[5] As a result, this review paper is a convenient

---

[3] Tensors are not matrices; other concepts are summation convention on repeated indices, chain rule, and matrix index convention for natural conversion from component form to matrix (and then tensor) form. See Section 4.2 on Matrix notation.

[4] See the review papers on deep learning, e.g., [12] [13] [14] [15] [16] [17] [18], many of which did not provide extensive discussion on applications, particularly on computational mechanics, such as in the present review paper.

[5] An example of a confusing point for *first-time learners* with knowledge of electrical circuits, hydraulics, or (biological) computational neuroscience [19] would be the interpretation of the arrows in an artificial neural network such as those in Figure 7 and Figure 8: Would these arrows represent real physical flows (electron flow, fluid flow, etc.)? No, they represent function mapping (or information passing); see Section 4.3.1 on Graphical representation. Even a tutorial such as [20] would follow the same format as many other papers, and while alluding to the human brain in
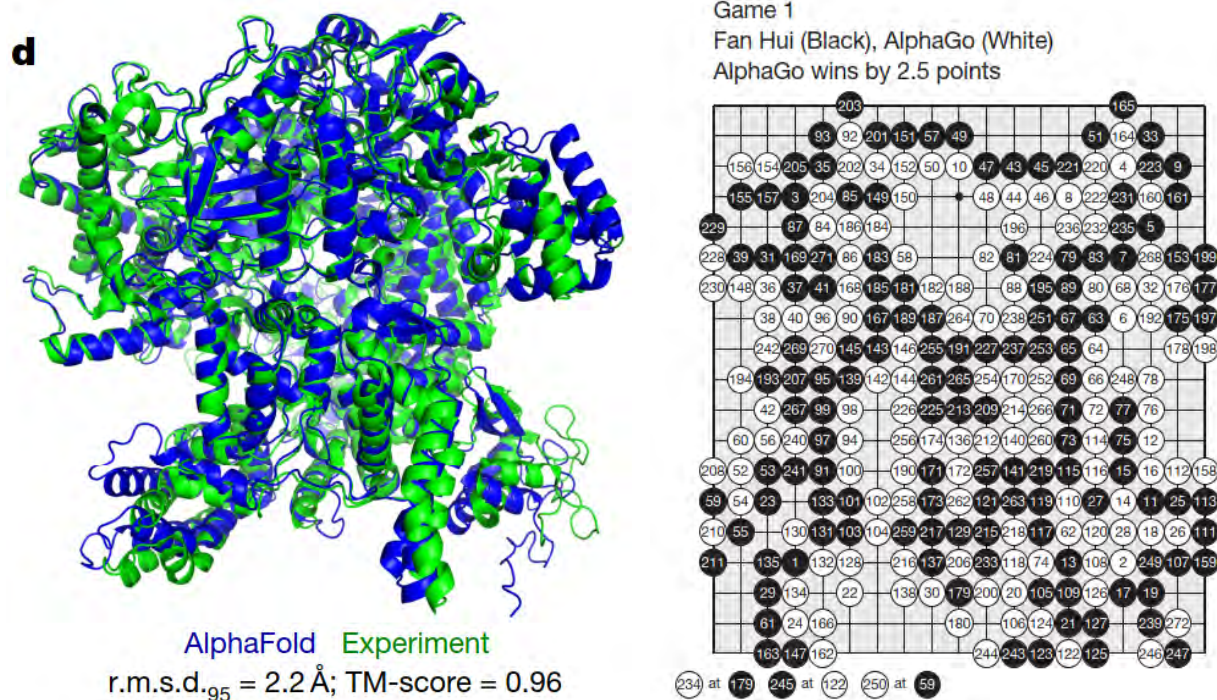
Figure 2: *Breakthroughs in AI* (Section 2). *Left:* The journal *Science* 2021 Breakthough of the Year. Protein folded 3-D shape produced by the AI software AlphaFold compared to experiment with high accuracy [5]. The AlphaFold Protein Structure Database contains more than 200 million protein structure predictions, a holy grail sought after in the last 50 years. *Right:* The AI solfware AlphaGo, a runner-up in the journal *Science* 2016 Breakthough of the Year, beat the European Go champion Fan Hui five games to zero in 2015 [6], and then went on to defeat the world Go grandmaster Lee Sedol in 2016 [7]. (Permission by Nature.)

"one-stop shopping" that provides the necessary fundamental information, with clarification of potentially confusing points, for first-time learners to quickly acquire a general understanding of the field that would facilitate deeper study and application to computational mechanics.

Deep-learning software libraries. Just as there is a large number of available software in different subfields of computational mechanics, there are many excellent deep-learning libraries ready for use in applications; see Section 9, in which some examples of the use of these libraries in engineering applications are provided with the associated computer code. Similar to learning finite-element formulations versus learning how to run finite-element codes, our focus here is to discuss various algorithmic aspects of deep-learning and their applications in computational mechanics, rather than how to use deep-learning libraries in applications. We agree with the view that "a solid understanding of the core principles of neural networks and deep learning" would provide "insights that will still be relevant years from now" [21], and that would not be obtained from just learning to run some hot libraries.

Readers already familiar with neural networks may find the presentation refreshing,[6] and even find new information on neural networks, depending how they used deep learning, or when they stopped working in this area due to the waning wave of connectionism and the new wave of deep learning.[7] If not, readers

---

their Figure 2 (which is the equivalent of Figure 8 below), did not explain the meaning of the arrows.

[6] Particularly the top-down approach for both feedforward network (Section 4) and back propagation (Section 5).

[7] It took five years from the publication of Rumelhart *et al.* 1986 [22] to the paper by Ghaboussi *et al.* 1991 [23], in which backpropagation (Section 5) was applied. It took more than twenty years from the publication of Long Short-

can skip these sections to go directly to the sections on applications of deep learning to computational mechanics.

Applications of deep learning in computational mechanics. We select some recent papers on application of deep learning to computational mechanics to review in details in a way that readers would also understand the computational mechanics contents well enough without having to read through the original papers:

- Fully-connected feedforward neural networks were employed to make element-matrix integration more efficient, while retaining the accuracy of the traditional Gauss-Legendre quadrature [38];[8]

- Recurrent neural network (RNN) with Long Short-Term Memory (LSTM) units[9] was applied to multiple-scale, multi-physics problems in solid mechanics [25];

- RNNs with LSTM units were employed to obtain reduced-order model for turbulence in fluids based on the proper orthogonal decomposition (POD), a classic linear project method also known as principal components analysis (PCA) [26]. More recent nonlinear-manifold model-order reduction methods, incorporating encoder / decoder and hyper-reduction of dimentionality using gappy (incomplete) data, were introduced, e.g., [47] [48].

Organization of contents. Our review of each of the above papers is divided into two parts. The first part is to summarize the main results and to identify the concepts of deep learning used in the paper, expected to be new for first-time learners, for subsequent elaboration. The second part is to explain in details how these deep-learning concepts were used to produce the results.

The results of deep-learning numerical integration [38] are presented in Section 2.3.1, where the deep-learning concepts employed are identified and listed, whereas the details of the formulation in [38] are discussed in Section 10. Similarly, the results and additional deep-learning concepts used in a multi-scale, multi-physics problem of geomechanics [25] are presented in Section 2.3.2, whereas the details of this formulation are discussed in Section 11. Finally, the results and additional deep-learning concepts used

---

Term Memory (LSTM) units in [24] to the two recent papers [25] and [26], which are reviewed in detail here, and where recurrent neural networks (RNNs, Section 7) with LSTM units (Section 7.2) were applied, even though there were some early works on application of RNNs (without LSTM units) in civil / mechanical engineering such as [27] [28] [29] [30]. But already, "fully attentional Transformer" was proposed to render "intricately constructed LSTM" unnecessary [31]. Most modern networks use the default rectified linear function (ReLU)—which was introduced in computational neuroscience since at least before [32] and [19], and then adopted in computer science beginning with [33] and [34]—instead of the traditional sigmoid function dated since the mid 1970s with [35], but yet many newer activation functions continue to appear regularly, aiming at improving accuracy and efficiency over previous activation functions, e.g., [36], [37]. In computational mechanics, by the beginning of 2019, there has not yet widespread use of ReLU activation function, even though ReLU was mentioned in [38], where the sigmoid function was actually employed to obtain the results (Section 10). See also Section 13 on Historical perspective.

[8] It would be interesting to investigate on how the adjusted integration weights using the method in [38] would affect the stability of an element stiffness matrix with reduced integration (even in the absence of locking) and the superconvergence of the strains / stresses at the Barlow sampling points. See, e.g., [39], p. 499. The optimal locations of these strain / stress sampling points do not depend on the integration weights, but only on the degree of the interpolation polynomials; see [40] [41]. "The Gauss points corresponding to reduced integration are the Barlow points (Barlow, 1976) at which the strains are most accurately predicted if the elements are well-shaped" [42].

[9] It is only a coincidence that (1) Hochreiter (1997), the first author in [24], which was the original paper on the widely used and highly successful Long Short-Term Memory (LSTM) unit, is on the faculty at Johannes Kepler University (home institution of this paper's author A.H.), and that (2) Ghaboussi (1991), the first author in [23], who was among the first researchers to apply fully-connected feedforward neural network to constitutive behavior in solid mechanics, was on the faculty at the University of Illinois at Urbana-Champaign (home institution of author L.V.Q.). See also [43], and for early applications of neural networks in other areas of mechanics, see e.g., [44], [45], [46].

in turbulent fluid simulation with proper orthogonal decomposition [26] are presented in Section 2.3.3, whereas the details of this formulation, together with the nonlinear-manifold model-order reduction [47] [48], are discussed in Section 12.

All of the deep-learning concepts identified from the above selected papers for in-depth are subsequently explained in detail in Sections 3 to 7, and then more in Section 13 on "Historical perspective".

The parallelism between computational mechanics, neuroscience, and deep learning is summarized in Section 3, which would put computational-mechanics first-time learners at ease, before delving into the details of deep-learning concepts.

Both time-independent (static) and time-dependent (dynamic) problems are discussed. The architecture of (static, time-independent) feedforward multilayer neural networks in Section 4 is expounded in detail, with first-time learners in mind, without assuming prior knowledge, and where experts may find a refreshing presentation and even new information.

Backpropagation, explained in Section 5, is an important method to compute the gradient of the cost function relative to the network parameters for use as a descent direction to decrease the cost function for network training.

For training networks—i.e., finding optimal parameters that yield low training error and lowest validation error—both classic deterministic optimization methods (using full batch) and stochastic optimization methods (using minibatches) are reviewed in detail, and at times even derived, in Section 6, which would be useful for both first-time learners and experts alike.

The examples used in training a network form the training set, which is complemented by the validation set (to determine when to stop the optimization iterations) and the test set (to see whether the resulting network could work on examples never seen before); see Section 6.1.

Deterministic gradient descent with classical line search methods, such as Armijo's rule (Section 6.2), were generalized to add stochasticity. Detailed pseudocodes for these methods are provided. The classic stochastic gradient descent (SGD) by Robbins & Monro (1951) [49] (Section 6.3, Section 6.3.1), with add-on tricks such as momentum Polyak (1964) [3] and fast (accelerated) gradient by Nesterov (1983 [50], 2018 [51]) (Section 6.3.2), step-length decay (Section 6.3.4), cyclic annealing (Section 6.3.4), minibatch-size increase (Section 6.3.5), weight decay (Section 6.3.6) are presented, often with detailed derivations.

Step-length decay is shown to be equivalent to simulated annealing using stochastic differential equation equivalent to the discrete parameter update. A consequence is to increase the minibatch size, instead of decaying the step length (Section 6.3.5). In particular, we obtain a new result for minibatch-size increase.

In Section 6.5, highly popular adaptive step-length (learning-rate) methods are discussed in a unified manner in Section 6.5.1, followed by the first paper on AdaGrad [52] (Section 6.5.2).

Overlooked in (or unknown to) other review papers and even well-known books on deep learning, exponential smoothing of time series originating from the field of forecasting dated since the 1950s, the key technique of adaptive methods, is carefully explained in Section 6.5.3.

The first adaptive methods that employed exponential smoothing were RMSProp [53] (Section 6.5.4) and AdaDelta [54] (Section 6.5.5), both introduced at about the same time, followed by the "immensely successful" Adam (Section 6.5.6) and its variants (Sections 6.5.7 and 6.5.8).

Particular attention is then given to a recent criticism of adaptive methods in [55], revealing their marginal value for generalization, compared to the good old SGD with effective initial step-length tuning and step-length decay (Section 6.5.9). The results were confirmed in three recent independent papers, among which is the recent AdamW adaptive method in [56] (Section 6.5.10).

Dynamics, sequential data, and sequence modeling are the subjects of Section 7. Discrete time-dependent problems, as a sequence of data, can be modeled with recurrent neural networks discussed in

Section 7.1, using the 1997 classic architecture such as Long Short-Term Memory (LSTM) in Section 7.2, but also the recent 2017-18 architectures such as transformer introduced in [31] (Section 7.4.3), based on the concept of attention [57]. Continuous recurrent neural networks originally developed in neuroscience to model the brain and the connection to their discrete counterparts in deep learning are also discussed in detail, [19] and Section 13.2.2 on "Dynamic, time dependence, Volterra series".

The features of several popular, open-source deep-learning frameworks and libraries—such as Tensor-Flow, Keras, PyTorch, etc.—are summarized in Section 9.

As mentioned above, detailed formulations of deep learning applied to computational mechanics in [38] [25] [26] [47] [48] are reviewed in Sections 10, 11, 12.

History of AI, limitations, danger, and the classics. Finally, a broader historical perspective of deep learning, machine learning, and artificial intelligence is discussed in Section 13, ending with comments on the geopolitics, limitations, and (identified-and-proven, not just speculated) danger of artificial intelligence in Section 14.

A rare feature is in a detailed review of some important classics to connect to the relevant concepts in modern literature, sometimes revealing misunderstanding in recent works, likely due to a lack of verification of the assertions made with the corresponding classics. For example, the first artificial neural network, conceived by Rosenblatt (1957) [1], (1962) [2], had 1000 neurons, but was reported as having a single neuron (Figure 42). Going beyond probabilistic analysis, Rosenblatt even built the Mark I computer to implement his 1000-neuron network (Figure 133, Sections 13.2 and 13.2.1). Another example is the "heavy ball" method, for which everyone referred to Polyak (1964) [3], but who more precisely called the "small heavy sphere" method (Remark 6.6). Others were quick to dismiss classical deterministic line-search methods that have been generalized to add stochasticity for network training (Remark 6.4). Unintended misrepresentation of the classics would mislead first-time learners, and unfortunately even seasoned researchers who used second-hand information from others, without checking the original classics themselves.

The use of Volterra series to model the nonlinear behavior of neuron in term of input and output firing rates, leading to continuous recurrent neural networks is examined in detail. The linear term of the Volterra series is a convolution integral that provides a theoretical foundation for the use of linear combination of inputs to a neuron, with weights and biases [19]; see Section 13.2.2.

The experiments in the 1950s by Furshpan et al. [58] [59] that revealed the rectified linear behavior in neuronal axon, modeled as a circuit with a diode, together with the use of the rectified linear activation function in neural networks in neuroscience years before being adopted for use in deep learning network, are reviewed in Section 13.3.2.

Reference hypertext links and Internet archive. For the convenience of the readers, whenever we refer to an online article, we provide both the link to original website, and if possible, also the link to its archived version in the Internet Archive. For example, we included in the bibliography entry of Ref. [60] the links to both the Original website and the Internet archive.[10]

2 Deep Learning, resurgence of Artificial Intelligence

In Dec 2021, the journal *Science* named, as its "2021 Breakthrough of the Year," the development of the AI software AlphaFold and its amazing feat of predicting a large number of protein structures [64]. "For nearly 50 years, scientists have struggled to solve one of nature's most perplexing challenges—predicting the complex 3D shape a string of amino acids will twist and fold into as it becomes a fully functional protein. This year, scientists have shown that artificial intelligence (AI)-driven software can achieve this

---

[10] While in the long run an original website may be moved or even deleted, the same website captured on the Internet Archive (also known as Web Archive or Wayback Machine) remains there permanently.
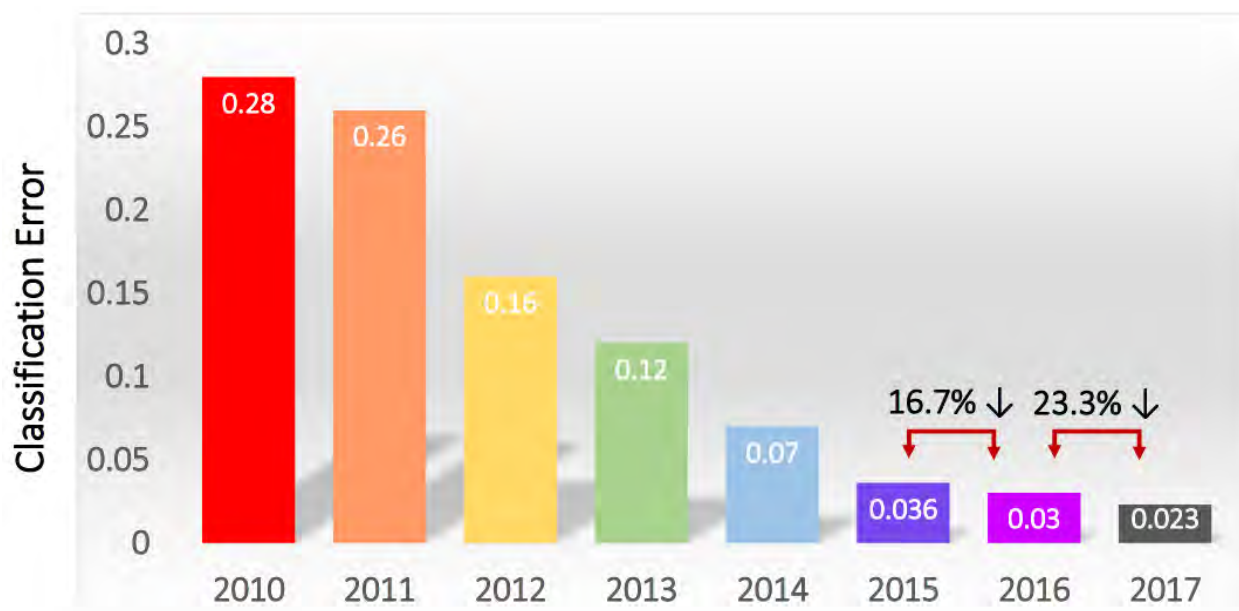
Figure 3: *ImageNet competitions* (Section 2). Top (smallest) classification error rate versus competition year. A sharp decrease in error rate in 2012 sparked a resurgence in AI interest and research [13]. By 2015, the top classification error rate surpassed human classification error rate of 5.1% with Parametric Rectified Linear Unit [61]; see Section 5.3.3 and also [62]. Figure from [63]. (Figure reproduced with permission of the authors.)

long-standing goal and predict accurate protein structures by the thousands and at a fraction of the time and cost involved with previous methods" [64].

The 3-D shape of a protein, obtained by folding a linear chain of amino acid, determines how this protein would interact with other molecules, and thus establishes its biological functions [64]. There are some 200 million proteins, the building blocks of life, in all living creatures, and 400,000 in the human body [64]. The AlphaFold Protein Structure Database already contained "over 200 million protein structure predictions."[11] For comparison, there were only about 190 thousand protein structures obtained through experiments as of 2022.07.28 [65]. "Some of AlphaFold's predictions were on par with very good experimental models [Figure 2, left], and potentially precise enough to detail atomic features useful for drug design, such as the active site of an enzyme" [66]. The influence of this software and its developers "would be epochal."

On the 2019 new-year day, *The Guardian* [67] reported the most recent breakthrough in AI, published less than a month before on 2018 Dec 07 in the journal *Science* in [68] on the development of the software AlphaZero, based on deep reinforcement learning (a combination of deep learning and reinforcement learning), that can teach itself through self-play, and then "convincingly defeated a world champion program in the games of chess, shogi (Japanese chess), as well as Go"; see Figure 2, right.

Go is the most complex game that mankind ever created, with more combinations of possible moves than chess, and thus the number of atoms in the observable universe.[12] It is "the most challenging of classic

---

[11] See also AlphaFold Protein Structure Database Internet archived as of 2022.09.02.

[12] The number of atoms in the observable universe is estimated at 10^{80}. For a board game such as chess and Go, the number of possible sequences of moves is $m = b^d$, with b being the game breadth (or "branching factor", which is the "number of legal moves per position" or average number of moves at each turn), and d the game depth (or length, also known as number of "plies"). For chess, $b \approx 35$, $d \approx 80$, and $m = 35^{80} \approx 10^{123}$, whereas For Go, $b \approx 250$, $d \approx 150$, and $m = 250^{150} \approx 10^{360}$. See, e.g., "Go and mathematics", Wikipedia, version 03:40, 13 June 2018; "Game-tree complexity", Wikipedia, version 07:04, 9 October 2018; [6].

games for artificial intelligence [AI] owing to its enormous search space and the difficulty of evaluating board positions and moves" [6].

This breakthrough is the crowning achievement in a string of astounding successes of deep learning (and reinforcent learning) in taking on this difficult challenge for AI.[13] The success of this recent breakthrough prompted an AI expert to declare close the multidecade long, arduous chapter of AI research to conquer immensely-complex games such as chess, shogi, and Go, and to suggest AI researchers to consider a new generation of games to provide the next set of challenges [73].

In its long history, AI research went through several cycles of ups and downs, in and out of fashion, as described in [74], 'Why artificial intelligence is enjoying a renaissance' (see also Section 13 on historical perspective):

> "THE TERM "artificial intelligence" has been associated with hubris and disappointment since its earliest days. It was coined in a research proposal from 1956, which imagined that significant progress could be made in getting machines to "solve kinds of problems now reserved for humans if a carefully selected group of scientists work on it together for a summer". That proved to be rather optimistic, to say the least, and despite occasional bursts of progress and enthusiasm in the decades that followed, AI research became notorious for promising much more than it could deliver. Researchers mostly ended up avoiding the term altogether, preferring to talk instead about "expert systems" or "neural networks". But in the past couple of years there has been a dramatic turnaround. Suddenly AI systems are achieving impressive results in a range of tasks, and people are once again using the term without embarrassment."

The recent resurgence of enthusiasm for AI research and applications dated only since 2012 with a spectacular success of almost halving the error rate in image classification in the ImageNet competition,[14] Going from 26% down to 16%; Figure 3 [63]. In 2015, deep-learning error rate of 3.6% was smaller than human-level error rate of 5.1%,[15] and then decreased by more than half to 2.3% by 2017.

The 2012 success[16] of a deep-learning application, which brought renewed interest in AI research out of its recurrent doldrums known as "AI winters",[17] is due to the following reasons:

- Availability of much larger datasets for training deep neural networks (find optimized parameters). It is possible to say that without ImageNet, there would be no spectacular success in 2012, and thus no resurgence of AI. Once the importance of having large datasets to develop versatile, working deep networks was realized, many more large datasets have been developed. See, e.g., [60].

- Emergence of more powerful computers than in the 1990s, e.g., the graphical processing unit (or GPU), "which packs thousands of relatively simple processing cores on a single chip" for use to process and display complex imagery, and to provide fast actions in today's video games" [77].

---

[13] See [69] [6] [70] [71]. See also the film *AlphaGo* (2017), "an excellent and surprisingly touching documentary about one of the great recent triumphs of artificial intelligence, Google DeepMind's victory over the champion Go player Lee Sedol" [72], and "AlphaGo versus Lee Sedol," Wikipedia version 14:59, 3 September 2022.

[14] "ImageNet is an online database of millions of images, all labelled by hand. For any given word, such as "balloon" or "strawberry", ImageNet contains several hundred images. The annual ImageNet contest encourages those in the field To compete and measure their progress in getting computers to recognise and label images automatically" [75]. See also [62] and [60], for a history of the development of ImageNet, which played a critical role in the resurgence of interest and research in AI by paving the way for the mentioned 2012 spectacular success in reducing the error rate in image recognition.

[15] For a report on the human image classification error rate of 5.1%, see [76] and [62], Table 10.

[16] Actually, the first success of deep learning occurred three years earlier in 2009 in speech recognition; see Section 2 regarding the historical perspective on the resurgence of AI.

[17] See [74].

- Advanced software infrastructure (libraries) that facilitates faster development of deep-learning applications, e.g., TensorFlow, PyTorch, Keras, MXNet, etc. [78], p. 25. See Section 9 on some reviews and rankings of deep-learning libraries.

- Larger neural networks and better training techniques (i.e., optimizing network parameters) that were not available in the 1980s. Today's much larger networks, which can solve once intractatable / difficult problems, are "one of the most important trends in the history of deep learning", but are still much smaller than the nervous system of a frog [78], p. 21; see also Section 4.6. A 2006 breakthrough, ushering in the dawn of a new wave of AI research and interest, has allowed for efficient training of deeper neural networks [78], p. 18.[18] The training of large-scale deep neural networks, which frequently involve highly nonlinear and non-convex optimization problems with many local minima, owes its success to the use of *stochastic-gradient* descent method first introduced in the 1950s [80].

- Successful applications to difficult, complex problems that help people in their every-day lives, e.g., image recognition, speech translation, etc.

  ⋆ In medicine, AI "is beginning to meet (and sometimes exceed) assessments by doctors in various clinical situations. A.I. can now diagnose skin cancer like dermatologists, seizures like neurologists, and diabetic retinopathy like ophthalmologists. Algorithms are being developed to predict which patients will get diarrhea or end up in the ICU,[19] and the FDA[20] recently approved the first machine learning algorithm to measure how much blood flows through the heart—a tedious, time-consuming calculation traditionally done by cardiologists." Doctors lamented that they spent "a decade in medical training learning the art of diagnosis and treatment," and were now easily surpassed by computers [81]. "The use of artificial intelligence is proliferating in American health care—outpacing the development of government regulation. From diagnosing patients to policing drug theft in hospitals, AI has crept into nearly every facet of the health-care system, eclipsing the use of machine intelligence in other industries" [82].

  ⋆ In micro-lending, AI has helped the Chinese company SmartFinance reduce the default rates of more than 2 millions loans per month to low single digits, a track record that makes traditional brick-and-mortar banks extremely jealous" [83].

  ⋆ In the popular TED talk "How AI can save humanity" [84], the speaker alluded to the above-mentioned 2006 breakthrough ([78], p. 18) that marked the beginning of the "deep learning" wave of AI research when he said:[21] "About 10 years ago, the grand AI discovery was made by three North American scientists,[22] and it's known as deep learning".

Section 13 provides a historical perspective on the development of AI, with additional details on current and future applications.

---

[18] The authors of [13] cited this 2006 breakthrough paper by Hinton, Osindero & Teh in their reference no.32 with the mention "This paper introduced a novel and effective way of training very deep neural networks by pre-training one hidden layer at a time using the unsupervised learning procedure for restricted Boltzmann machines (RBMs)." A few years later, it was found out that RBMs were not necessary to train deep networks, as it was sufficient to use rectified linear units (ReLUs) as active functions ([79], interview with Y. Bengio); see also Section 4.4.2 on active functions. For this reason, we are not reviewing RBMs here.

[19] Intensive Care Unit.

[20] Food and Drug Administration.
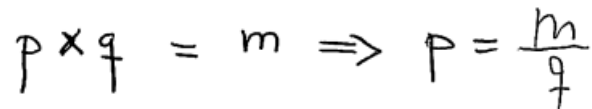
[21] At video time 1:51. In less than a year, this 2018 April TED talk had more than two million views as of 2019 March.

[22] See Footnote 18 for the names of these three scientists.

It was, however, disappointing that despite the above-mentioned exciting outcomes of AI, during the Covid-19 pandemic beginning in 2020,[23] none of the hundreds of AI systems developed for Covid-19 diagnosis were usable for clinical applications; see Section 13.5.1. As of June 2022, the Tesla electric vehicle autopilot system is under increased scrutiny by the National Highway Traffic Safety Administration as there were "16 crashes into emergency vehicles and trucks with warning signs, causing 15 injuries and one death."[24] In addition, there are many limitations and danger in the current state-of-the-art of AI; see Section 14.

2.1 Handwritten equation to LaTeX code, image recognition

An image-recognition software useful for computational mechanicists is Mathpix Snip,[25] which recognizes hand-written math equations, and transforms them into LaTex codes. For example, Mathpix Snip transforms the hand-written equation below by an 11-year old pupil:
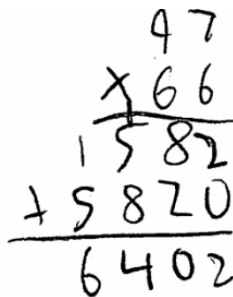
Figure 4: *Handwritten equation 1* (Section 2.1)

into this LaTeX code "`p \times q = m \Rightarrow p = \frac { m } { q }`" to yield the equation image:

$$p \times q = m \Rightarrow p = \frac{m}{q} \tag{1}$$

Another example is the hand-written multiplication work below by the same pupil:

Figure 5: *Handwritten equation 2* (Section 2.1). Hand-written multiplication work of an eleven-year old pupil.

---

[23] "The World Health Organization declares COVID-19 a pandemic" on 2020 Mar 11, CDC Museum COVID-19 Timeline, Internet archive 2022.06.02.
[24] Krisher T., Teslas with Autopilot a step closer to recall after wrecks, Associated Press, 2022.06.10.
[25] We thank Kerem Uguz for informing the senior author LVQ about Mathpix.

that Mathpix Snip transformed into the equation image below:[26]

$$\begin{array}{r} 97 \\ \times 66 \\ \hline 1582 \\ +5820 \\ \hline 6402 \end{array} \tag{2}$$

2.2 Artificial intelligence, machine learning, deep learning

We want to immediately clarify the meaning of the terminologies "Artificial Intelligence" (AI), "Machine Learning" (ML), and "Deep Learning" (DL), since their casual use could be confusing for first-time learners.

For example, it was stated in a review of primarily two computer-science topics called "Neural Networks" (NNs) and "Support Vector Machines" (SVMs) and a physics topic that [85]:[27]

"The respective underlying fields of basic research—quantum information versus machine learning (ML) and artificial intelligence (AI)—have their own specific questions and challenges, which have hitherto been investigated largely independently."

Questions would immediately arise in the mind of first-time learners: Are ML and AI two different fields, or the same fields with different names? If one field is a subset of the other, then would it be more general to just refer to the larger set? On the other hand, would it be more specific to just refer to the subset?

In fact, Deep Learning is a subset of methods inside a larger set of methods known as Machine Learning, which in itself is a subset of methods generally known as Artificial Intelligence. In other words, Deep Learning is Machine Learning, which is Artificial Intelligence; [78], p. 9.[28] On the other hand, Artificial Intelligence is not necessarily Machine Learning, which in itself is not necessarily Deep Learning.

The review in [85] was restricted to Neural Networks (which could be deep or shallow)[29] and Support Vector Machine (which is Machine Learning, but not Deep Learning); see Figure 6. Deep Learning can be thought of as multiple levels of composition, going from simpler (less abstract) concepts (or representations) to more complex (abstract) concepts (or representations).[30]

Based on the above relationship between AI, ML, and DL, it would be much clearer if the phrase "machine learning (ML) and artificial intelligence (AI)" in both the title of [85] and the original sentence quoted above is replaced by the phrase "machine learning (ML)" to be more specific, since the authors mainly reviewed Multi-Layer Neural (MLN) networks (deep learning, and thus machine learning) and Support Vector Machine (machine learning).[31] MultiLayer Neural (MLN) network is also known as MultiLayer Perceptron

---

[26] Mathpix Snip "misunderstood" that the top horizontal line was part of a fraction, and upon correction of this "misunderstanding" and font-size adjustment yielded the equation image shown in Eq. (2).

[27] We are only concerned with NNs, not SVMs, in the present paper.

[28] References to books are accompanied with page numbers for specific information cited here so readers don't waste time to wade through an 800-page book to look for such information.

[29] Network depth and size are discussed in Section 4.6.1. An example of a shallow network with one hidden layer can be found in Section 12.4 on nonlinear-manifold model-order reduction applied to fluid mechanics.

[30] See, e.g., [78], p. 5, p. 8, p. 14.

[31] For more on Support Vector Machine (SVM), see [78], p. 137. In the early 1990s, SVM displaced neural networks with backpropagation as a better method for the machine-learning community ([79], interview with G. Hinton). The resurgence of AI due to advances in deep learning started with the seminal paper [86], in which the authors demonstrated via numerical experiments that MLN network was better than SVM in terms of error in the handwriting-recognition benchmark test using the MNIST handwritten digit database, which contains "a training set of 60,000
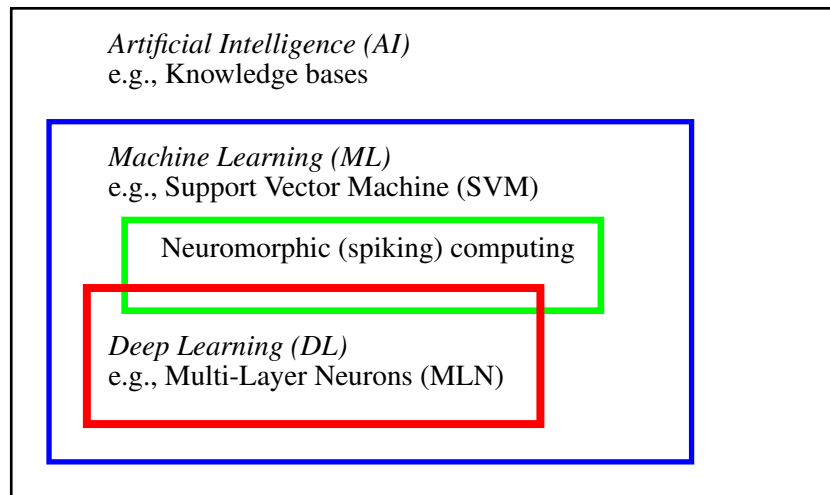
Figure 6: *Artificial intelligence and subfields* (Section 2.2). Three classes of methods—*Artificial Intelligence* (AI), *Machine Learning* (ML), and *Deep Learning* (DL)—and their relationship, with an example of method in each class. A knowledge-base method is an AI method, but is neither a ML method, nor a DL method. Support Vector Machine and spiking computing are ML methods, and thus AI methods, but not a DL method. Multi-Layer Neural (MLN) network is a DL method, and is thus both an ML method and an AI method. See also Figure 158 in Appendix 4 on *Cybernetics*, which encompassed all of the above three classes.

(MLP).[32] both MLN networks and SVMs are considered as artificial intelligence, which in itself is too broad and thus not specific enough.

Another reason for simplifying the title in [85] is that the authors did not consider using any other AI methods, except for two specific ML methods, even though they discussed AI in the general historical context.

The engine of neuromorphic computing, also known as spiking computing, is a hardware network built into the IBM TrueNorth chip, which contains "1 million programmable spiking neurons and 256 million configurable synapses",[33] and consumes "extremely low power" [87]. Despite the apparent difference with the software approach of deep computing, neuromorphic chip could implement deep-learning networks, and thus the difference was not fundamental [88]. There is thus an overlap between neuromorphic computing and deep learning, as shown in Figure 6, instead of two disconnected subfields of machine learning as reported in [20].[34]

2.3 Motivation, applications to mechanics

As motivation, we present in this section the results in three recent papers in computational mechanics, mentioned in the Opening Remarks in Section 1, and identify some deep-learning fundamental concepts (in

---

examples, and a test set of 10,000 examples." But kernel methods studied for the development of SVM have now been used in connection with networks with infinite width to understand how deep learning works; see Section 8 on "Kernel machines" and Section 14.2 on "Lack of understanding."

[32] See [78], p. 5.

[33] The neurons are the computing units, and the synapses the memory instead of grouping the computing units into a central processing unit (CPU), separated from the memory, and connect the CPU and the memory via a bus, which creates a communication bottleneck, like the brain, each neuron in the TrueNorth chip has its own synapses (local memory).

[34] In [20], there was only a reference to [87], but not to [88]. It is likely that the authors of [20] were not aware of [88], and thus an intersection between neuromorphic computing and deep learning.
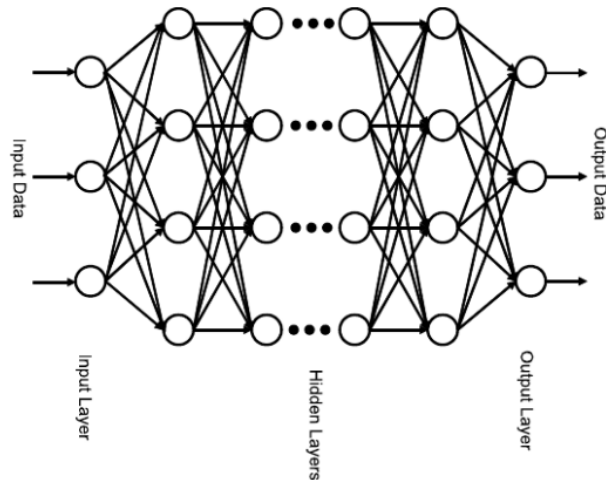
Figure 7: *Feedforward neural network* (Section 2.3.1). A feedforward neural network in [38], rotated clockwise by 90 degrees to compare to its equivalent in Figure 23 and Figure 35 further below. All terminologies and fundamental concepts will be explained in detail in subsequent sections as listed. See Section 4.1.1 for a top-down explanation and Section 4.1.2 for bottom-up explanation. This figure of a network could be confusing to first-time learners, as already indicated in Footnote 5. (Figure reproduced with permission of the authors.)

italics) employed in these papers, together with the corresponding sections in the present paper where these concepts are explained in detail. First-time learners of deep learning likely find these fundamental concepts described by obscure technical jargon, whose meaning will be explained in details in the identified subsequent sections. Experts of deep learning would understand how deep learning is applied to computational mechanics.

2.3.1 Enhanced numerical quadrature for finite elements

To integrate efficiently and accurately the element matrices in a general finite element mesh of 3-D hexahedral elements (including distorted elements), the power of Deep Learning was harnessed in two applications of *feedforward MultiLayer Neural networks* (MLN,[35] Figures 7-8, Section 4) [38]:

(1) Application 1.1: For each element (particularly distorted elements), find the number of integration points that provides accurate integration within a given error tolerance. Section 10.2 contains the details.

(2) Application 1.2: Uniformly use $2 \times 2 \times 2$ integration points for all elements, distorted or not, and find the appropriate quadrature weights[36] (*different* from the traditional quadrature weights of the Gauss-Legendre method) at these integration points. Section 10.3 contains the details.

   To *train*[37] the networks—i.e., to optimize the network parameters (weights and biases, Figure 8) to minimize some *loss (cost, error) function* (Sections 5.1, 6)—up to 20,000 randomly distorted hexahedrals were generated by displacing nodes from a regularly shaped element [38], see Figure 9. For each distorted shape, the following are determined: (1) the minimum number of integration points required to reach a

---

[35] MLN is also called MultiLayer Perceptron (MLP); see Footnote 32.
[36] The *quadrature* weights at integration points are not to be confused with the *network* weights in a MLN network.
[37] See Section 6 on "Network training, optimization methods".
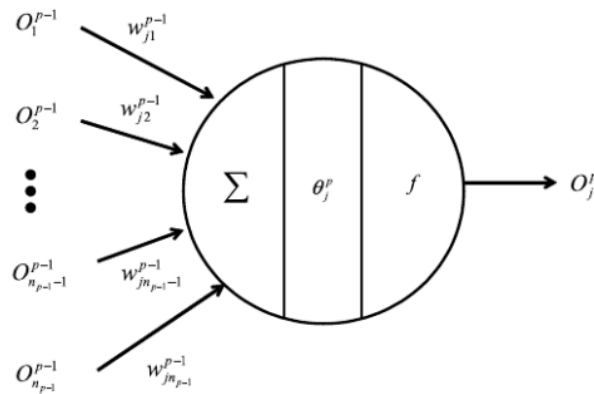
Figure 8: *Artificial neuron* (Section 2.3.1). A neuron with its multiple inputs O_i^{p-1} (which are outputs from the previous layer $(p-1)$, and thus the variable name "O"), processing operations (multiply inputs with network weights w_{ji}^{p-1}, sum weighted inputs, add bias θ_j^p, activation function f), and single output O_j^p [38]. See the equivalent Figure 36, Section 4.4.3 further below. All terminologies and fundamental concepts will be explained in detail in subsequent sections as listed. See Section 4 on feedforward networks, Section 4.1.1 on top-down explanation and Section 4.1.2 on bottom-up explanation. This figure of a neuron could be confusing to first-time learners, as already indicated in Footnote 5. (Figure reproduced with permission of the authors.)
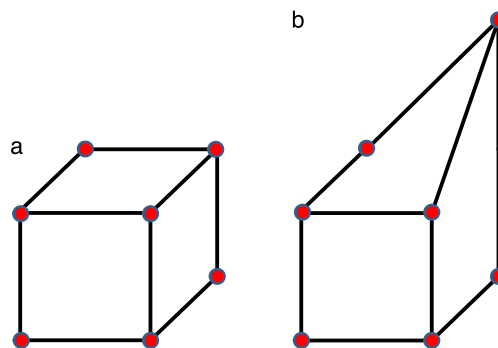
Figure 9: *Cube and distorted cube elements* (Section 2.3.1). Regular and distorted linear hexahedral elements [38]. (Figure reproduced with permission of the authors.)

prescribed accuracy, and (2) corrections to the quadrature weights by trying one million randomly generated sets of correction factors, among which the best one was retained.

While Application 1.1 used one *fully-connected* (Section 4.6.1) feedforward neural network (Section 4), Application 1.2 relied on two neural networks: The first neural network was a classifier that took the element shape (18 normalized nodal coordinates) as input and estimated whether or not the numerical integration (quadrature) could be improved by adjusting the quadrature weights for the given element (one output), i.e., the network classifier only produced two outcomes, yes or no. If an error reduction was possible, a second neural network performed regression to predict the corrected quadrature weights (eight outputs for $2 \times 2 \times 2$ quadrature) from the input element shape (usually distorted).

To train the classifier network, 10,000 element shapes were selected from the prepared dataset of 20,000 hexahedrals, which were divided into a *training set* and a *validation set* (Section 6.1) of 5000 elements each.[38]

---

[38] For the definition of training set and test set, see Section 6.1. Briefly, the training set is used to optimize the network

To train the second regression network, 10,000 element shapes were selected for which quadrature could be improved by adjusting the quadrature weights [38].

Again, the training set and the test set comprised 5000 elements each. The parameters of the neural networks (*weights, biases*, Figure 8, Section 4.4) were optimized (trained) using a *gradient descent method* (Section 6) that minimizes a *loss function* (Section 5.1), whose gradients with respect to the parameters are computed using *backpropagation* (Section 5).

(a) Training patterns.   (b) Test patterns.
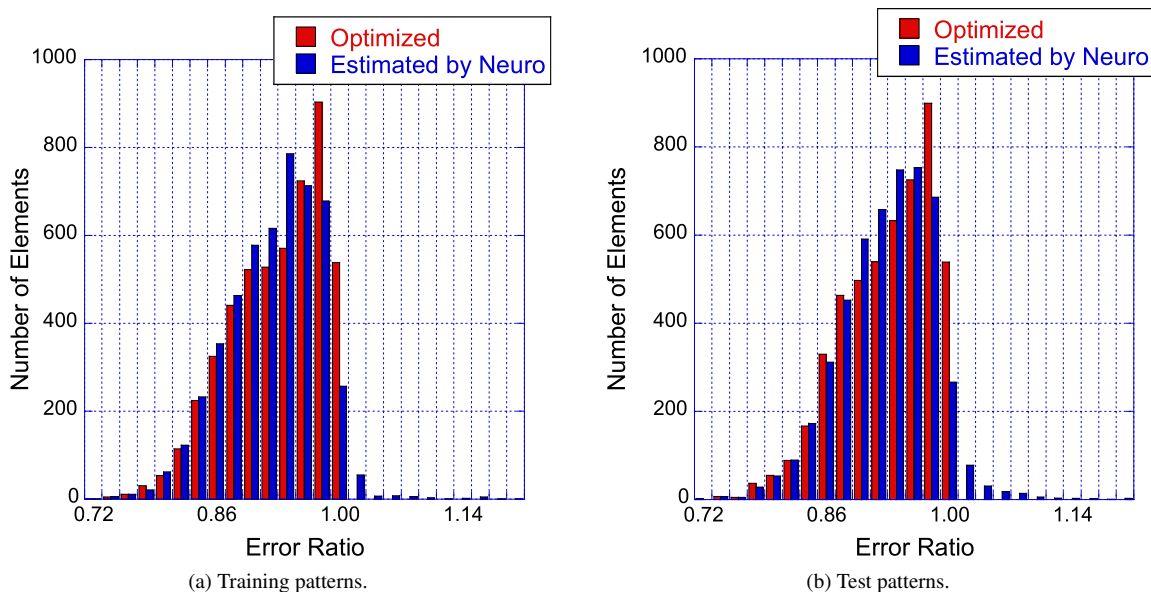
Figure 10: *Effectiveness of quadrature weight prediction* (Section 2.3.1). Subfigure (a): Distribution of error-reduction ratio R_{error} for the 5,000 elements in the training set. The red bars ("Optimized") are the error ratios obtained from the optimal weights (found by a large number of trials-and-errors) that were used to train the network. The blue bars ("Estimated by Neuro") are the error ratios obtained from the trained neural network. $R_{\text{error}} < 1$ indicates improved quadrature accuracy. As a result of using the optimal weights, there were no red bars with $R_{\text{error}} > 1$. That there were very few blue bars with $R_{\text{error}} > 1$ showed that the proposed method worked in reducing the integration error in more than 97% of the elements. Subfigure (b): Error ratios for the test set with 5000 elements [38]. More detailed explanation is provided in Section 10.3.3. (Figure reproduced with permission of the authors.)

The best results were obtained from a classifier with four *hidden layers* (Figure 7, Section 4.3, Remark 4.2) with 30 *neurons* (Figure 8, Figure 36, Section 4.4.3) each and a regression network that had a depth of five hidden layers, where each layer was 50 neurons wide, Figure 7. The results were obtained using the *logistic sigmoid function* (Figure 30) as *activation function* (Section 4.4.2) due to existing software, even though the *rectified linear function* (Figure 24) were more efficient, but yielded comparable accuracy on a few test cases.[39]

To quantify the effectiveness of the approach in [38], an error-reduction ratio was introduced, i.e., the quotient of the quadrature error with quadrature weights predicted by the neural network and the error obtained with the standard quadrature weights of Gauss-Legendre quadrature with $2 \times 2 \times 2$ integration points; see Eq. (402) in Section 10.3 with $q = 2$ and "*opt*" stands for "optimized" (or "predicted"). When the

---

parameters, while the test set is used to see how good the network with these optimized parameters can predict the targets of never-seen-before inputs.

[39] Information provided by author A. Oishi of [38] through a private communication to the authors on 2018 Nov 16.

error-reduction ratio is less than 1, the integration using the predicted quadrature weights is more accurate than that using the standard quadrature weights. To compute the two quadrature errors mentioned above (one for the predicted quadrature weights and one for the standard quadrature weights, both for the same $2 \times 2 \times 2$ integration points), the reference values considered as most accurate were obtained using $30 \times 30 \times 30$ integration points with the standard quadrature quadrature weights; see Eq. (401) in Section 10.2 with $q_{max} = 30$.

For most element shapes of both the training set (a) and the test set (b), each of which comprised 5000 elements, the blue bars in Figure 10 indicate an error ratio below one, i.e., the quadrature weight correction effectively improved the accuracy of numerical quadrature.

Readers familiar with Deep Learning and neural networks can go directly to Section 10, where the details of the formulations in [38] are presented. Other sections are also of interest such as classic and state-of-the-art optimization methods in Section 6, attention and transformer unit in Section 7, historical perspective in Section 13, limitations and danger of AI in Section 14.

Readers not familiar with Deep Learning and neural networks will find below a list of the concepts that will be explained in subsequent sections. To facilitate the reading, we also provide the section number (and the link to jump to) for each concept.

Deep-learning concepts to explain and explore:

(1) Feedforward neural network (Figure 7): Figure 23 and Figure 35, Section 4

(2) Neuron (Figure 8): Figure 36 in Section 4.4.4 (artificial neuron), and Figure 131 in Section 13.1 (biological neuron)

(3) Inputs, output, hidden layers, Section 4.3

(4) Network depth and width: Section 4.3

(5) Parameters, weights, biases 4.4.1

(6) Activation functions: Section 4.4.2

(7) What is "deep" in "deep networks" ? Size, architecture, Section 4.6.1, Section 4.6.2

(8) Backpropagation, computation of gradient: Section 5

(9) Loss (cost, error) function, Section 5.1

(10) Training, optimization, stochastic gradient descent: Section 6

(11) Training error, validation error, test (or generalization) error: Section 6.1

This list is continued further below in Section 2.3.2. Details of the formulation in [38] are discussed in Section 10.
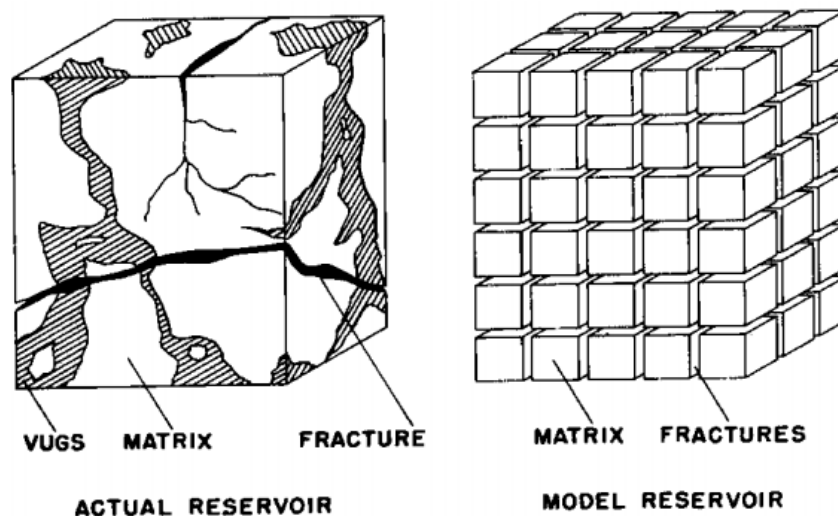
Figure 11: *Dual-porosity single-permeability medium* (Section 2.3.2). *Left*: Actual reservoir. Dual (or double) porosity indicates the presence of two types of porosity in naturally-fractured reservoirs (e.g., of oil): (1) Primary porosity in the matrix (e.g., voids in sands) with low permeability, within which fluid does not flow, (2) Secondary porosity due to fractures and vugs (cavities in rocks) with high (anisotropic) permeability, within which fluid flows. Fluid exchange is permitted between the matrix and the fractures, but not between the matrix blocks (sugar cubes), of which the permeability is much smaller than in the fractures. *Right*: Model reservoir, idealization. The primary porosity is an array of cubes of homogeneous, isotropic material. The secondary porosity is an "orthogonal system of continuous, uniform fractures", oriented along the principal axes of anisotropic permeability [89]. (Figure reproduced with permission of the publisher SPE.)

2.3.2 Solid mechanics, multiscale modeling

One way that deep learning can be used in solid mechanics is to model complex, nonlinear constitutive behavior of materials. In single physics, balance of linear momentum and strain-displacement relation are considered as definitions or "universal principles", leaving the constitutive law, or stress-strain relation, to a large number of models that have limitations, no matter how advanced [91]. Deep learning can help model complex constitutive behaviors in ways that traditional phenomenological models could not; see Figure 105.

Deep *recurrent neural networks* (RNNs) (Section 7.1) was used as a scale-bridging method to efficiently simulate multiscale problems in hydromechanics, specifically plasticity in porous media with dual porosity *and* dual permeability [25].[40]

The *dual-porosity single-permeability* (DPSP) model was first introduced for use in oil-reservoir simulation [89], Figure 11, where the fracture system was the main flow path for the fluid (e.g., two phase

---

[40] Porosity is the ratio of void volume over total volume. Permeability is a scaling factor, which when multiplied by the negative of the pressure gradient, and divided by the fluid dynamic viscosity, gives the fluid velocity in Darcy's law, Eq. (409). The expression "dual-porosity dual-permeability poromechanics problem" used in [25], p. 340, could confuse first-time readers—especially those who are familiar with traditional reservoir simulation, e.g., in [92]—since dual porosity (also called "double porosity" in [89]) and dual permeability are two different models of naturally-fractured porous media; these two models for radionuclide transport around nuclear waste repository were studied in [93]. Further added to the confusion is that the dual-porosity model is more precisely called *dual-porosity-single permeability* model, whereas the dual-permeability model is called *dual-porosity dual-permeability* model [94], which has a different meaning than the one used in [25].
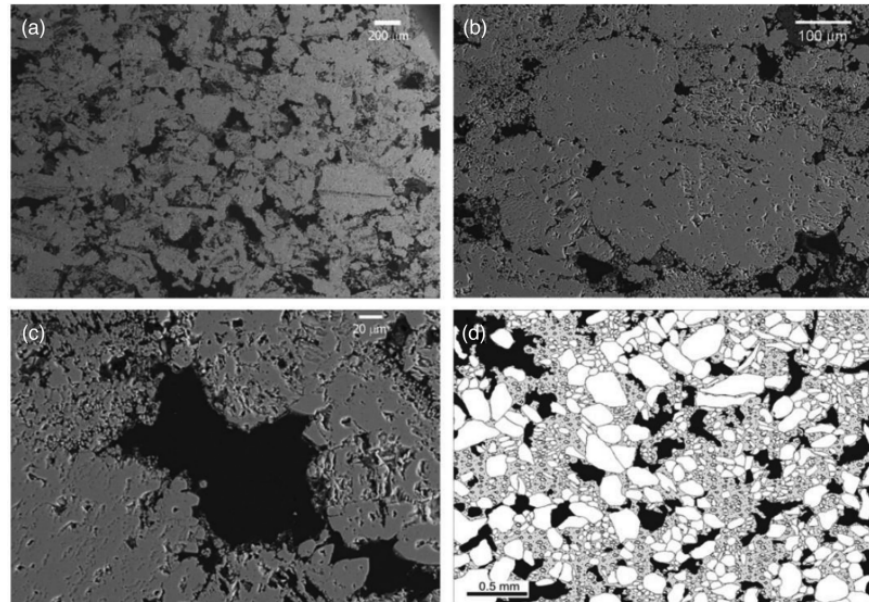
Figure 12: *Pore structure of Majella limestone, dual porosity* (Section 2.3.2), a carbonate rock with high total porisity at 30%. Backscattered SEM images of Majella limestone: (a)-(c) sequence of zoomed-ins; (d) zoomed-out. (a) The larger macropores (dark areas) have dimensions comparable to the grains (allochems), having an average diameter of 54 μm, with macroporosity at 11.4%. (b) Micropores embedded in the grains and cemented regions, with microporosity at 19.6%, which is equal to the total porosity at 30% minus the macroporosity. (c) Numerous micropores in the periphery of a macropore. (d) Map performed manually under optical microscope showing the partitioning of grains, matrix (mostly cement) and porosity [90]. See Section 11 and Remark 11.8. (Figure reproduced with permission of the authors.)

oil-water mixture, one-phase oil-solvent mixture). Fluid exchange is permitted between the rock matrix and the fracture system, but not between the matrix blocks. In the DPSP model, the fracture system and the rock matrix, each has its own porosity, with values not differing from each other by a large factor. On the contrary, the permeability of the fracture system is much larger than that in the rock matrix, and thus the system is considered as having only a single permeability. When the permeability of the fracture system and that of the rock matrix do not differ by a large factor, then both permeabilities are included in the more general *dual-porosity dual-permeability* (DPDP) model [94].

Since 60% of the world's oil reserve and 40% of the world's gas reserve are held in carbonate rocks, there has been a clear interest in developing an understanding of the mechanical behavior of carbonate rocks such as limestones, having from lowest porosity (Solenhofen at 3%) to high porosity (e.g., Majella at 30%). Chalk (Lixhe) is a carbonate rock with highest porosity at 42.8%. Carbonate rock reservoirs are also considered to store carbon dioxide and nuclear waste [95] [93].

In oil-reservoir simulations in which the primary interest is the flow of oil, water, and solvent, the porosity (and pore size) within each domain (rock matrix or fracture system) is treated as constant and homogeneous [94] [96].[41] On the other hand, under mechanical stress, the pore size would change, cracks and other defects would close, leading to a change in the porosity in carbonate rocks. Indeed, "at small stresses, experimental mechanical deformation of carbonate rock is usually characterized by a non-linear stress-strain relationship, interpreted to be related to the closure of cracks, pores, and other defects. The

---

[41] See, e.g., [94], p. 295, Chap. 9 on "Advanced Topics: Fluid Flow in Fractured Reservoirs and Compositional Simulation".
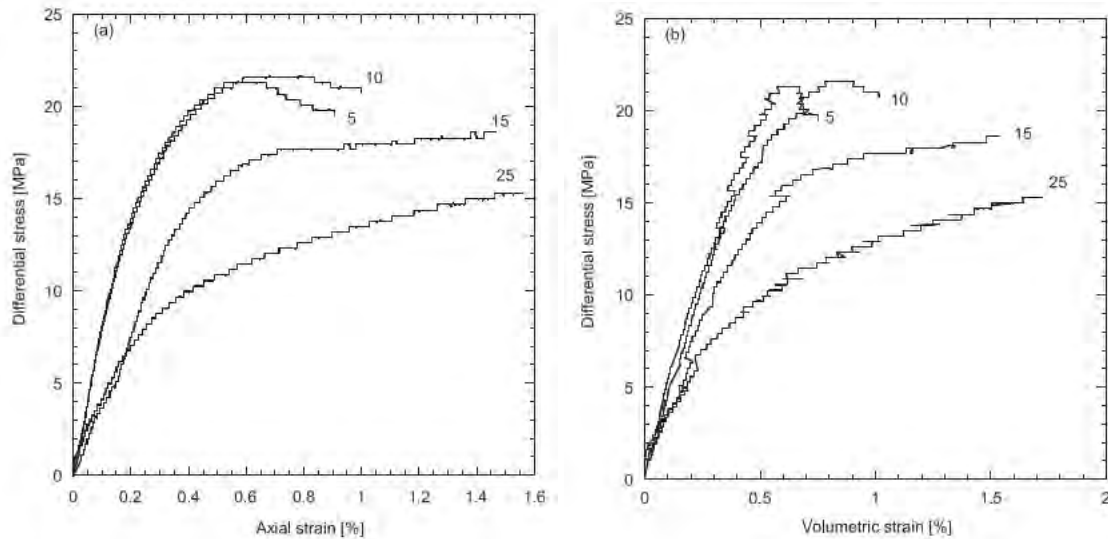
Figure 13: *Majella limestone, nonlinear stress-strain relations* (Section 2.3.2). Differential stress (i.e., the difference between the largest principal stress and the smallest one) vs axial strain (left) and vs volumetric strain (right) [90]. See Remark 11.7, Section 11.3.4, and Remark 11.10, Section 11.3.5. (Figure reproduced with permission of the authors.)

non-linear stress-strain relationship can be related to the amount of cracks and various type of pores" [95], p. 202. Once the pores and cracks are closed, the stress-strain relation becomes linear, at different stress stages, depending on the initial porosity and the geometry of the pore space [95].

Moreover, pores have different sizes, and can be classified into different pore sub-systems. For the Majella limestone in Figure 12 with total porosity at 30%, its pore space can be partitioned into two subsystems (and thus dual porosity), the macropores with macroporosity at 11.4% and the micropores with microporosity at 19.6%. Thus the meaning of dual-porosity as used in [25] is different from that in oil-reservoir simulation. Also characteristic of porous rocks such as the Majella limestone is the non-linear stress-strain relation observed in experiments, Figure 13, due the changing size, and collapse, of the pores.

Likewise, the meaning of "dual permeability" is different in [25] in the sense that "one does not seek to obtain a single effective permeability for the entire pore space". Even though it was not explicitly spelled out,[42] it appears that each of the two pore sub-systems would have its own permeability, and that fluid is allowed to exchange between the two pore sub-systems, similar to the fluid exchange between the rock matrix and the fracture system in the DPSP and DPDP models in oil-reservoir simulation [94].

In the problem investigated in [25], the presence of localized discontinuities demands three scales—microscale (μ), mesoscale (cm), macroscale (km)—to be considered in the modeling, see Figure 14. Classical approaches to consistently integrate microstructural properties into macroscopic constitutive laws relied on hierarchical simulation models and homogenization methods (e.g., discrete element method (DEM)–FEM coupling, FEM^2). If more than two scales were to be considered, the computational complexity would become prohibitively, if not intractably, large.

Instead of coupling multiple simulation models online, two (adjacent) scales were linked by a neural network that was trained offline using data generated by simulations on the smaller scale [25]. The trained

---

[42] At least at the beginning of Section 2 in [25].

[43] The LSTM variant with peephole connections is not the original LSTM cell (Section 7.2); see, e.g, [97]. The equations describing the LSTM unit in [25], whose authors never mentioned the word "peephole", correspond to the original LSTM without peepholes. It was likely a mistake to use this figure in [25].
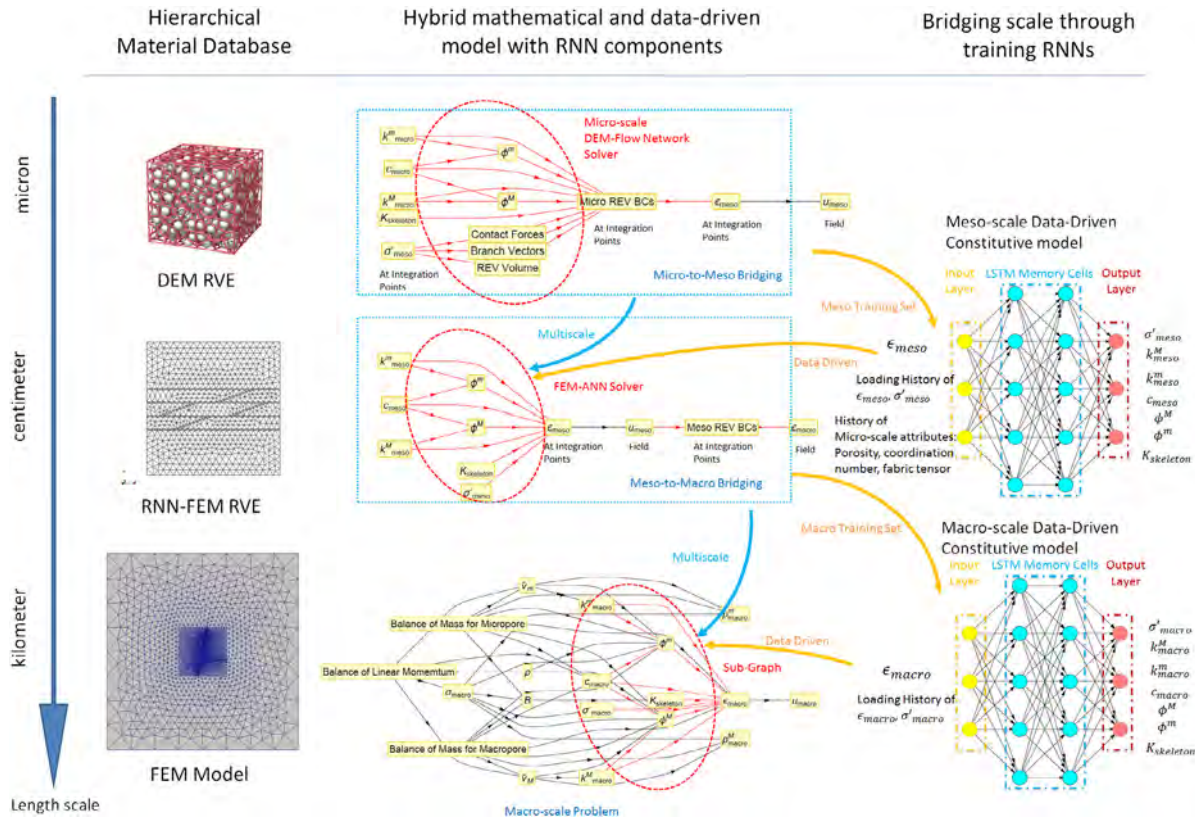
Figure 14: *Hierarchy of a multi-scale multi-physics poromechanics problem for fluid-infiltrating media* [25] (Sections 2.3.2, 11.1, 11.3.1, 11.3.3, 11.3.5). Microscale (μ), mesoscale (cm), macroscale (km). DEM RVE = Discrete Element Method Representative Volume Element. RNN-FEM = Recurrent Neural Network (Section 7.1) - Finite Element Method. LSTM = Long Short-Term Memory (Section 7.2). The mesoscale has embedded strong discontinuities equivalent to the fracture system in Figure 11. See Figure 104, where the orientations of the RVEs are shown, Figure 106 for the microscale RVE (Remark 11.1) and Figure 113 for the mesoscale RVE (Remark 11.9). (Figure reproduced with permission of the authors.)

network subsequently served as a surrogate model in online simulations on the larger scale. With three scales being considered, two recurrent neural networks (RNNs) with Long Short-Term Memory (LSTM) units were employed consecutively:

(1) *Mesoscale RNN with LSTM units:* On the microscopic scale, a representative volume element (RVE) was an assembly of discrete-element particles, subjected to large variety of representative loading paths to generate training data for the supervised learning of the mesoscale RNN with LSTM units, a neural network that was referred to as "Mesoscale data-driven constitutive model" [25] (Figure 14). Homogenizing the results of DEM-flow model provided constitutive equations for the traction-separation law and the evolution of anisotropic permeabilities in damaged regions.

(2) *Macroscale RNN with LSTM units:* The mesoscale RVE (middle row in Figure 14), in turn, was a finite-element model of a porous material with embedded strong discontinuities equivalent to the fracture system in oil-reservoir simulation in Figure 11. The host matrix of the RVE was represented by an isotropic linearly elastic solid. In localized fracture zones within, the traction-separation law and the hydraulic response were provided by the mesoscale RNN with LSTM units developed above. Training data for the macroscale RNN with LSTM units—a network referred to as "Macroscale data-driven
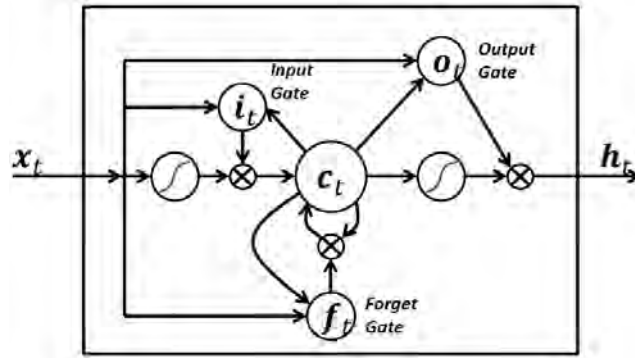
Figure 15: *LSTM variant with "peephole" connections*, block diagram (Sections 2.3.2, 7.2).[43] Unlike the original LSTM unit (see Section 7.2), both the input gate and the forget gate in an LSTM unit with peephole connections receive the cell state as input. The above figure from Wikipedia, version 22:56, 4 October 2015, is identical to Figure 10 in [25], whose authors erroneously used this figure without mentioning the source, but where the original LSTM unit *without* "peepholes" was actually used, and with the detailed block diagram in Figure 81, Section 7.2. See also Figure 82 and Figure 117 for the original LSTM unit applied to fluid mechanics. (CC-BY-SA 4.0)

constitutive model" [25]—is generated by computing the (homogenized) response of the mesoscale RVE to various loadings. In macroscopic simulations, the mesoscale RNN with LSTM units provided the constitutive response at a sealing fault that represented a strong discontinuity.

Path-dependence is a common characteristic feature of the constitutive models that are often realized as neural networks; see, e.g., [23]. For this reason, it was decided to employ RNN with LSTM units, which could mimic internal variables and corresponding evolution equations that were intrinsic to path-dependent material behavior [25]. These authors chose to use a neural network that had a depth of two hidden layers with 80 LSTM units per layer, and that had proved to be a good compromise of performance and training efforts. After each hidden layer, a dropout layer with a dropout rate 0.2 were introduced to reduce overfitting on noisy data, but yielded minor effects, as reported in [25]. The output layer was a fully-connected layer with a logistic sigmoid as activation function.

An important observation is that including micro-structural data—the porosity ϕ, the coordination number CN (number of contact points, Figure 16), the fabric tensor (defined based on the normals at the contact points, Eq. (404) in Section 11; Figure 16 provides a visualization)—as network inputs significantly improved the prediction capability of the neural network. Such improvement is not surprising since soil fabric—described by scalars (porosity, coordination number, particle size) and vectors (fabric tensors, particle orientation, branch vectors)—exerts great influence on soil behavior [99]. Coordination number[44] has been used to predict soil particle breakage [100], morphology and crushability [99], and in a study of internally-unstable soil involving a mixture of coarse and fine particles [101]. Fabric tensors, with theoretical foundation developed in [102], provide a mean to represent directional data such as normals at contact points, even though other types of directional data have been proposed to develop fabric tensors [103]. To model anisotropic behavior of granular materials, contact-normal fabric tensor was incorporated in an isotropic constitutive law.

---

[44] The coordination number (Wikipedia version 20:43, 28 July 2020) is a concept originated from chemistry, signifying the number of bonds from the surrounding atoms to a central atom. In Figure 16 (a), the uranium borohydride $U(BH_4)_4$ complex (Wikipedia version 08:38, 12 March 2019) has 12 hydrogen atoms bonded to the central uranium atom.
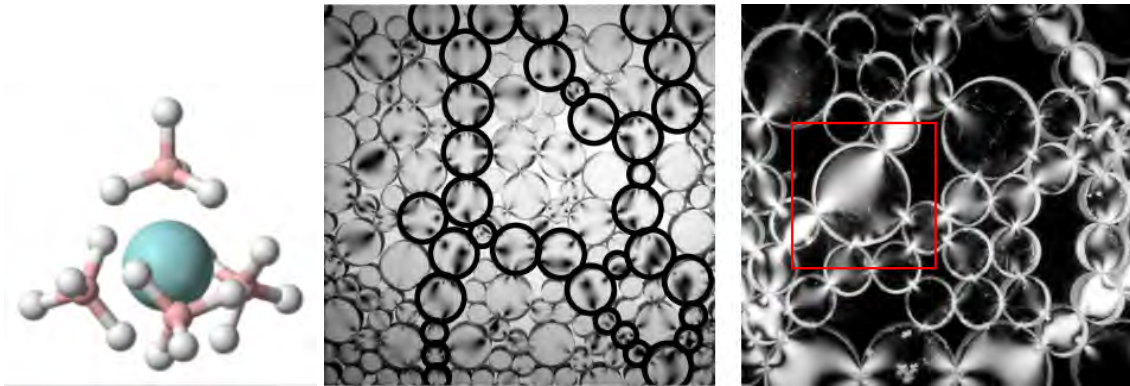
Figure 16: *Coordination number CN* (Section 2.3.2, 11.3.2). (a) Chemistry. Number of bonds to the central atom. Uranium borohydride $U(BH_4)_4$ has $CN = 12$ hydrogen bonds to uranium. (b, c) Photoelastic discs showing number of contact points (coordination number) on a particle. (b) Random packing and force chains, different force directions along principal chains and in secondary particles. (c) Arches around large pores, precarious stability around pores. The coordination number for the large disc (particle) in red square is 5, but only 4 of those had non-zero contact forces based on the bright areas showing stress action. Figures (b, c) also provide a visualization of the "flow" of the contact normals, and thus the fabric tensor [98]. See also Figure 17. (Figure reproduced with permission of the author.)

Figure 17 illustrates the importance of incorporating microstructure data, particularly the fabric tensor, in network training to improve prediction accuracy.

Deep-learning concepts to explain and explore: (continued from above in Section 2.3.1)

(12) Recurrent neural network (RNN), Section 7.1

(13) Long Short-Term Memory (LSTM), Section 7.2

(14) Attention and Transformer, Section 7.4.3

(15) Dropout layer and dropout rate,[45] which had minor effects in the particular work repoorted in [25], and thus will not be covered here. See [78], p. 251, Section 7.12.

Details of the formulation in [25] are discussed in Section 11.

2.3.3 Fluid mechanics, reduced-order model for turbulence

The accurate simulation of turbulence in fluid flows ranks among the most demanding tasks in computational mechanics. Owing to both the spatial and the temporal resolution, transient analysis of turbulence by means of high-fidelity methods such as Large Eddy Simulation (LES) or direct numerical simulation (DNS) involves millions of unknowns even for simple domains.

To simulate complex geometries over larger time periods, reduced-order models (ROMs) that can capture the key features of turbulent flows within a low-dimensional approximation space need to be resorted to. Proper Orthogonal Decomposition (POD) is a common data-driven approach to construct an orthogonal

---

[45] Briefly, dropout means to drop or to remove non-output units (neurons) from a base network, thus creating an ensemble of sub-networks (or models) to be trained for each example, and can also be considered as a way to add noise to inputs, particularly of hidden layers, to train the base network, thus making it more robust, since neural networks were known to be not robust to noise. Adding noise is also equivalent to increasing the size of the dataset for training, [78], p. 233, Section 7.4 on "Dataset augmentation".
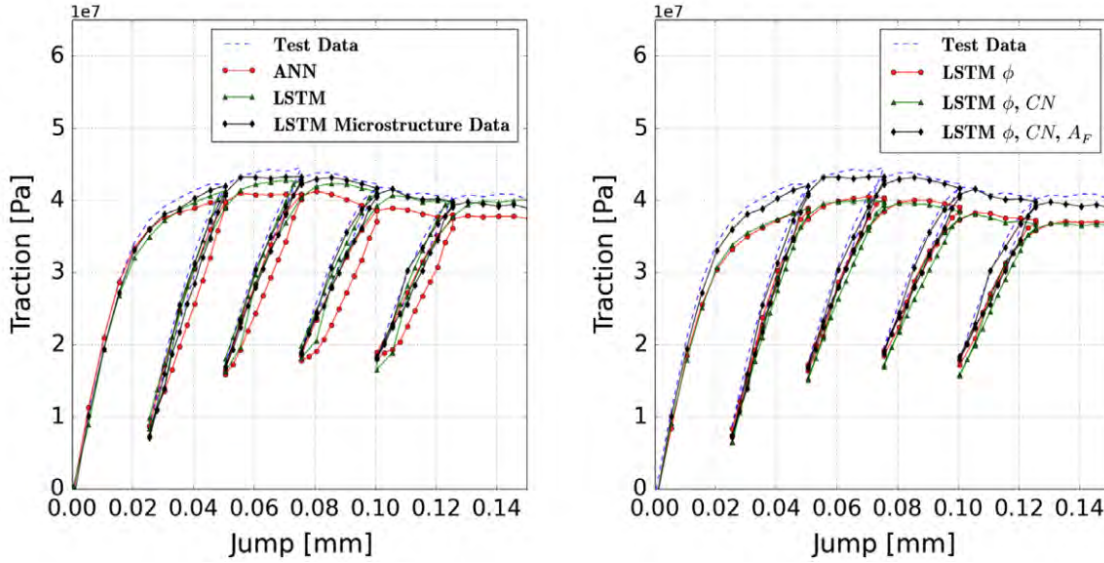
Figure 17: *Network with LSTM and microstructure data* (porosity ϕ, coordination number $CN = N_c$, Figure 16, fabric tensor $\boldsymbol{F} = \boldsymbol{A}_F \cdot \boldsymbol{A}_F$, Eq. (404)) (Section 2.3.2, 11.3.2). Simple shear test using Discrete Element Method to provide network training data under loading-unloading conditions. ANN = Artificial Neural Network with no LSTM units. While network with LSTM units and (ϕ, CN, $\boldsymbol{A}_F$) improved the predicted traction, compared to network with LSTM units and only (ϕ) or (ϕ, CN); the latter two networks produced predicted traction that was worse compared to network with LSTM alone, indicating the important role of the fabric tensor, which contained directional data that were absent in scalar fields like (ϕ, CN) [25]. (Figure reproduced with permission of the author.)

basis $\{\phi_1(\boldsymbol{x}), \phi_2(\boldsymbol{x}), \ldots \phi_\infty(\boldsymbol{x})\}$ from high-resolution data obtained from high-fidelity models or measurements, in which $\boldsymbol{x}$ is a point in a 3-D fluid domain $\mathcal{B}$; see Section 12.1. A flow dynamic quantity $u(\boldsymbol{x}, t)$, such as a component of the flow velocity field, can be projected on the POD basis by separation of variables as (Figure 18)

$$u(\boldsymbol{x}, t) = \sum_{i=1}^{i=\infty} \phi_i(\boldsymbol{x})\alpha_i(t) \approx \sum_{i=1}^{i=k} \phi_i(\boldsymbol{x})\alpha_i(t) , \text{ with } k < \infty , \tag{3}$$

where k is a finite number, which could be large, and $\alpha_i(t)$ a time-dependent coefficient for $\phi_i(\boldsymbol{x})$. The computation would be more efficient if a much smaller subset with, say, $m \ll k$, POD basis functions,

$$u(\boldsymbol{x}, t) \approx \sum_{j=1}^{j=m} \phi_{i_j}(\boldsymbol{x})\alpha_{i_j}(t) , \text{ with } m \ll k \text{ and } i_j \in \{1, \ldots, k\} , \tag{4}$$

where $\{i_j, j = 1, \ldots, m\}$ is a subset of indices in the set $\{1, \ldots, k\}$, and such that the approximation in Eq. (4) is with minimum error compared to the approximation in Eq. (3)$_2$ using the much larger set of k POD basis functions.

In a Galerkin-Project (GP) approach to reduced-order model, a small subset of dominant modes form a basis onto which high-dimensional differential equations are projected to obtain a set of lower-dimensional differential equations for cost-efficient computational analysis.

Instead of using GP, RNNs (Recurrent Neural Networks) were used in [26] to predict the evolution of fluid flows, specifically the coefficients of the dominant POD modes, rather than solving differential equations. For this purpose, their LSTM-ROM (Long Short-Term Memory - Reduced Order Model) approach combined concepts of ROM based on POD with deep-learning neural networks using either the original
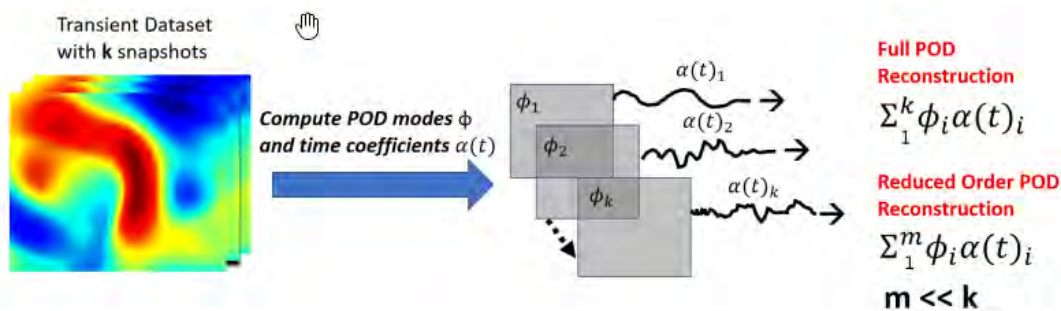
Figure 18: *Reduced-order POD basis* (Sections 2.3.3, 12.1). For each dataset (also Figure 116), which contained k snapshots, the full POD reconstruction of the flow-field dynamical quantity $u(\boldsymbol{x}, t)$, where $\boldsymbol{x}$ is a point in the 3-D flow field, consists of all k basis functions $\phi_i(\boldsymbol{x})$, with $i = 1, \ldots, k$, using Eq. (3); see also Eq. (439). Typically, k is large; a reduced-order POD basis consists of selecting $m \ll k$ basis functions for the reconstruction, with the smallest error possible. See Figure 19 for the use of deep-learning networks to predict $\alpha_i(t + t')$, with $t' > 0$, given $\alpha_i(t)$ [26]. (Figure reproduced with permission of the author.)

LSTM units, Figure 117 (left) [24], or the bidirectional LSTM (BiLSTM), Figure 117 (right) [104], the internal states of which were well-suited for the modeling of dynamical systems.

To obtain training/testing data, which were crucial to train/test neural networks, the data from transient 3-D Direct Navier-Stokes (DNS) simulations of two physical problems, as provided by the Johns Hopkins turbulence database [105] were used [26]: (1) The *Forced Isotropic Turbulence* (ISO) and (2) The *Magnetohydrodynamic Turbulence* (MHD).

To generate training data for LSTM/BiLSTM networks, the 3-D turbulent fluid flow domain of each physical problem was decomposed into five equidistant 2-D planes (slices), with one additional equidistant 2-D plane served to generate testing data (Section 12, Figure 116, Remark 12.1). For the same subregion in each of those 2-D planes, POD was applied on the k snapshots of the velocity field ($k = 5,023$ for ISO, $k = 1,024$ for MHD, Section 12.1), and out of k POD modes $\{\phi_i(\boldsymbol{x}), \ i = 1, \ldots, k\}$, the five ($m = 5 \ll k$) most dominant POD modes $\{\phi_i(\boldsymbol{x}), \ i = 1, \ldots, m\}$ representative of the flow dynamics (Figure 18) were retained to form a reduced-order basis onto which the velocity field was projected. The coefficient $\alpha_i(t)$ of the POD mode $\phi_i(\boldsymbol{x})$ represented the evolution of the participation of that mode in the velocity field, and was decomposed into thousands of small samples using a moving window. The first half of each sample was used as input signal to an LSTM network, whereas the second half of the sample was used as output signal for supervised training of the network. Two different methods were proposed [26]:

(1) *Multiple-network method:* Use a RNN for each coefficient of the dominant POD modes

(2) *Single-network method:* Use a single RNN for all coefficients of the dominant POD modes

For both methods, variants with the original LSTM units or the BiLSTM units were implemented. Each of the employed RNN had a single hidden layer.

Demonstrative results for the prediction capabilities of both the original LSTM and the BiLSTM networks are illustrated in Figure 20. Contrary to the authors' expectation, networks with the original LSTM units performed better than those using BiLSTM units in both physical problems of isotropic turbulence (ISO) (Figure 20a) and magnetohydrodyanmics (MHD) (Figure 20b) [26].

Details of the formulation in [26] are discussed in Section 12.
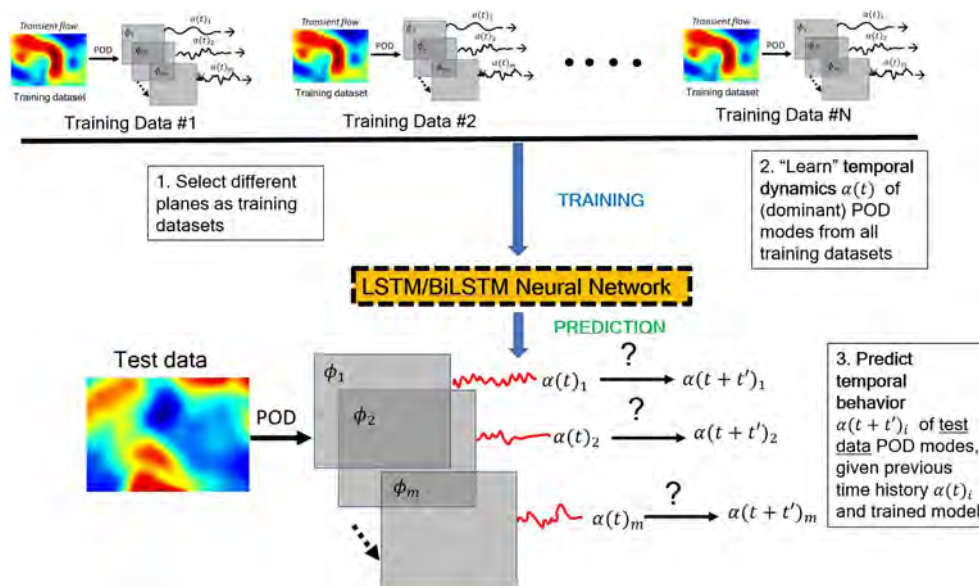
Figure 19: *Deep-learning LSTM/BiLSTM Reduced Order Model* (Sections 2.3.3, 12.2). See Figure 18 for POD reduced-order basis. The time-dependent coefficients $\alpha_i(t)$ of the dominant POD modes ϕ_i, with $i = 1, \ldots, m$, from each training dataset were used to train LSTM (or BiLSTM, Figure 117) neural networks to predict $\alpha_i(t + t')$, with $t' > 0$, given $\alpha_i(t)$ of the test datasets. [26]. (Figure reproduced with permission of the author.)

3 Computational mechanics, neuroscience, deep learning

Table 1 below presents a rough comparison that shows the parallelism in the modeling steps in three fields: Computational mechanics, neuroscience, and deep learning, which heavily indebted to neuroscience until it reached more mature state, and then took on its own development.

We assume that readers are familiar with the concepts listed in the second column on "Computational mechanics", and briefly explain some key concepts in the third column on "Neuroscience" to connect to the fourth column "Deep learning", which is explained in detail in subsequent sections.

See Section 13.2 for more details on the theoretical foundation based on Volterra series for the spatial and temporal combinations of inputs, weights, and biases, widely used in artificial neural networks or multilayer perceptrons.

Neuron spiking response such as shown in Figure 21 can be modelled accurately using a model such as "Integrate-and-Fire". The firing-rate response $r(\cdot)$ of a biological neuron to a stimulus $s(\cdot)$ is described by a convolution integral[46]

$$r(t) = r_0 + \int_{\tau=-\infty}^{\tau=t} \mathcal{K}(t-\tau)\, s(\tau)\, d\tau \,, \tag{5}$$

where r_0 is the background firing rate at zero stimulus, $\mathcal{K}(\cdot)$ is the synaptic kernel, and $s(\cdot)$ the stimulus; see, e.g., [19], p. 46, Eq. (2.1).[47]. The stimulus $s(\cdot)$ in Eq. (5) is a train (sequence in time) of spikes described

---

[46] From here on, if Eq. (5) is found a bit abstract at first reading, first-time learners could skip the remaining of this short Section 3 to begin reading Section 4, and come back later after reading through subsequent sections, particularly Section 13.2, to have an overview of the connection among seemingly separate topics.

[47] Eq. (5) is a reformulated version of Eq. (2.1) in [19], p. 46, and is similar to Eqs. (7.1)-(7.2) in [19], p. 233, Chapter "7 Network Models".

(a) Isotropic turbulence (ISO)   **(b)** Magnetohydrodynamics (MHD)
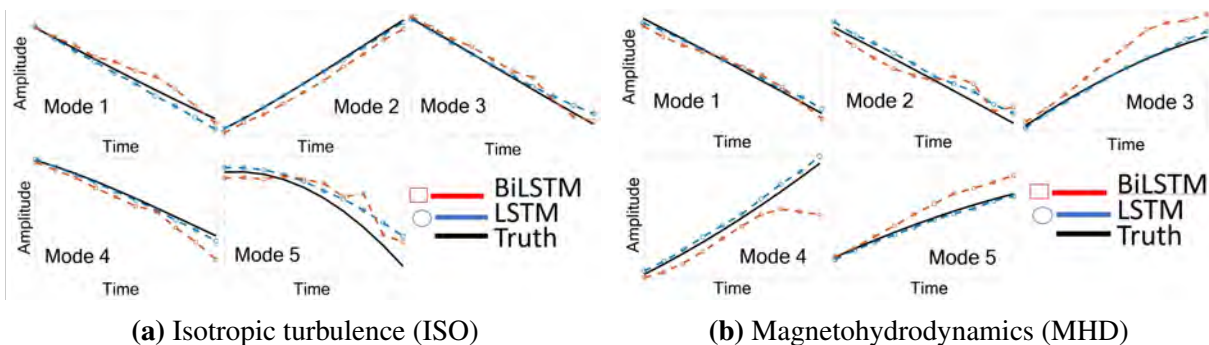
Figure 20: *Prediction results and errors for LSTM/BiLSTM networks* (Sections 2.3.3, 12.3). Coefficients $\alpha_i(t)$, for $i = 1, \ldots, 5$, of dominant POD modes. LSTM had smaller errors compared to BiLSTM for both physical problems (ISO and MHD).
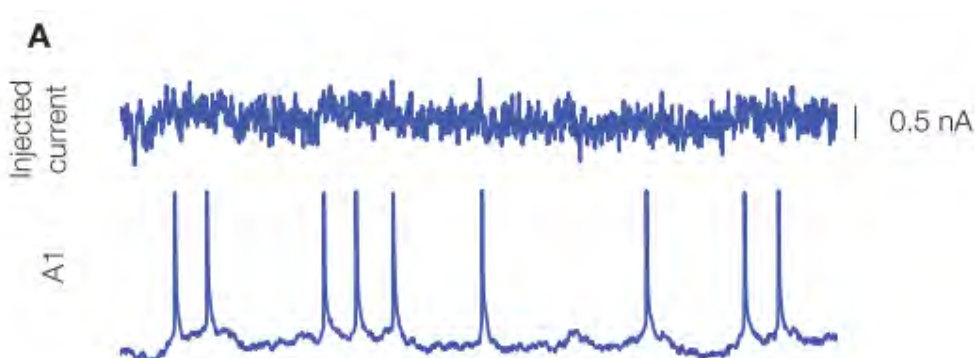
Figure 21: *Biological neuron response to stimulus, experimental result* (Section 3). An oscillating current was injected into the neuron (top), and neuron spiking response was recorded (below) [106], with permission from the authors and Frontiers Media SA. A spike, also called an action potential, is an electrical potential pulse across the cell membrane that lasts about 100 milivolts over 1 miliseconds. "Neurons represent and transmit information by firing sequences of spikes in various temporal patterns" [19], pp. 3-4.

by

$$s(\tau) = \sum_i \delta(\tau - t_i) \tag{6}$$

where $\delta(\cdot)$ is the Dirac delta. Eq. (5) then describes the firing rate $r(t)$ at time t as the collective memory effect of all spikes, going from the current time $\tau = t$ back far in the past with $\tau = -\infty$, with the weight for the spike $s(\tau)$ at time τ provided by the value of the synaptic kernel $\mathcal{K}(t - \tau)$ at the same time τ.

It will be seen in Section 13.2.2 on "Dynamics, time dependence, Volterra series" that the convolution integral in Eq. (5) corresponds to the linear part of the Volterra series of nonlinear response of a biological neuron in terms of the stimulus, Eq. (497), which in turn provides the theoretical foundation for taking the linear combination of inputs, weights, and biases for an artificial neuron in a multilayer neural networks, as represented by Eq. (26).

The Integrate-and-Fire model for biological neuron provides a motivation for the use of the rectified linear units (ReLU) as activation function in multilayer neural networks (or perceptrons); see Figure 28.

Eq. (5) is also related to the exponential smoothing technique used in forecasting and applied to stochastic optimization methods to train multilayer neural networks; see Section 6.5.3 on "Forecasting time series,

exponential smoothing".

4 Statics, feedforward networks

We examine in detail the forward propagation in feedforward networks, in which the function mappings flow[48] only one forward direction, from input to output.

4.1 Two concept presentations

There are two ways to present the concept of deep-learning neural networks: The top-down approach versus the bottom-up approach.

4.1.1 Top-down approach

The *top-down* approach starts by giving up-front the mathematical big picture of what a neural network is, with the big-picture (high level) graphical representation, then gradually goes down to the detailed specifics of a processing unit (often referred to as an *artificial neuron*) and its low-level graphical representation. A definite advantage of this top-down approach is that readers new to the field immediately have the big picture in mind, before going down to the nitty-gritty details, and thus tend not to get lost. An excellent reference for the top-down approach is [78], and there are not many such references.

Specifically, for a multilayer feedforward network, by top-down, we mean starting from a general description in Eq. (18) and going down to the detailed construct of a neuron through a weighted sum with bias in Eq. (26) and then a nonlinear activation function in Eq. (35).

In terms of block diagrams, we begin our *top-down* descent from the big picture of the overall multilayer neural network with L layers in Figure 23, through Figure 34 for a typical layer (ℓ) and Figure 35 for the lower-level details of layer (ℓ), then down to the most basic level, a neuron in Figure 36 as one row in layer (ℓ) in Figure 35, the equivalent figure of Figure 8, which in turn was the starting point in [38].

4.1.2 Bottom-up approach

The *bottom-up* approach typically starts with a biological neuron (see Figure 131 in Section 13.1 below), then introduces an *artificial neuron* that looks similar to the biological neuron (compare Figure 8 to Figure 131), with multiple inputs and a single output, which becomes an input to each of a multitude of other artificial neurons; see, e.g., [23] [21] [38] [20].[49] Even though Figure 7, which preceded Figure 8 in [38], showed a network, but the information content is not the same as Figure 23.

Unfamiliar readers when looking at the graphical representation of an artificial neural network (see, e.g., Figure 7) could be misled in thinking in terms of electrical (or fluid-flow) networks, in which Kirchhoff's law applies at the junction where the output is split into different directions to go toward other artificial neurons. The big picture is not clear at the outset, and could be confusing to readers new to the field, who would take some time to understand; see also Footnote 5. By contrast, Figure 23 clearly shows a multilevel function composition, assuming that first-time learners are familiar with this basic mathematical concept.

4.2 Matrix notation

In mechanics and physics, tensors are intrinsic geometrical objects, which can be represented by infinitely many matrices of components, depending on the coordinate systems.[50] vectors are tensors of order 1.

---

[48] There is no physical flow here, only function mappings.
[49] Figure2 in[20] is essentially the same as Figure 8.
[50] See, e.g., [107], [108].

Table 1: *Top-down (rough) comparison of modeling steps in three fields* (Section 3): Computational mechanics, neuroscience, and deep learning

Study object	Engineering continuum	The Brain	Image recognition
Field	Computational mechanics	Computational neuroscience	Deep learning
Modeling	Partial Differential Equations	Biological neural networks	Artificial neural networks
Inputs	Forces (solids), velocities (fluids)	Firing rate as stimulus	An image to classify
1	Weak form, finite-element mesh, order of interpolation	Network architectures, two layers (input, output), several neurons per layer	Network architectures, many layers (input, hidden, output), very high number of neurons and parameters
2	Elements	Neurons, dendrites, synapses, axons	Processing units (neurons, perceptrons)
3	Nonlinear force-displacement and stress-strain (σ-ϵ) relations	Firing model, spiking model, firing rate vs input current (FI) relation, continuous stimulus and response, Volterra series, kernels of increasing orders	—
4	Linearized force-displacement and stress-strain relations (Hooke's law)	Linear term in Volterra series, synaptic kernel $\mathcal{K}_1(\tau)$ of order 1, continuous temporal weight	Many hidden layers (discrete weights and biases)
5	—	Linear combination of inputs, with input weights $\boldsymbol{w}$	Linear combination of inputs plus biases, input weights $\boldsymbol{w}$
6	—	Static nonlinearity	Activation function
Outputs	Displacements (solids), velocities (fluids)	Firing rate as response	Image classified (car, frog, human)

For this reason, we do *not* use neither the name "vector" for a column matrix, nor the name "tensor" for an array with more than two indices.[51] All arrays are matrices.

The matrix notation used here can follow either (1) the Matlab / Octave code syntax, or (2) the more compact component convention for tensors in mechanics.

Using Matlab / Octave code syntax, the inputs to a network (to be defined soon) are gathered in an $n \times 1$ column matrix $\boldsymbol{x}$ of real numbers, and the (*target* or labeled) outputs[52] in an $m \times 1$ column matrix $\boldsymbol{y}$

$$\boldsymbol{x} = [x_1, \cdots , x_n]^T = [x_1; \cdots ; x_n] = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n \times 1} \, , \quad \boldsymbol{y} = [y_1, \cdots , y_m]^T \in \mathbb{R}^{m \times 1} \tag{7}$$

where the commas are separators for matrix elements in a row, and the semicolons are separators for rows. For matrix transpose, we stick to the standard notation using the superscript "T" for written documents, instead of the prime "$\prime$" as used in matlab / octave code. In addition, the prime "$\prime$" is more customarily used to denote derivative in handwritten and in typeset equations.

Using the component convention for tensors in mechanics,[53] The coefficients of a $n \times m$ matrix shown below

$$[A_{ij}] = [A^i_j] \in \mathbb{R}^{n \times m} \tag{8}$$

are arranged according to the following convention for the free indices i and j, which are automatically expanded to their respective full range, i.e., $i = 1, \ldots , n$ and $j = 1, \ldots m$ when the variable $A_{ij} = A^i_j$ are enclosed in square brackets:

(1) In case both indices are subscripts, then the left subscript (index i of A_{ij} in Eq. (8)) denotes the row index, whereas the right subscript (index j of A_{ij} in Eq. (8)) denotes the column index.

(2) In case one index is a superscript, and the other index is a subscript, then the superscript (upper index i of A^i_j in Eq. (8)) is the row index, and the subscript (lower index j of A^i_j in Eq. (8)) is the column index.[54]

With this convention (lower index designates column index, while upper index designates row index), the coefficients of array $\boldsymbol{x}$ in Eq. (7) can be presented either in row form (with lower index) or in column form (with upper index) as follows:

$$[x_i] = [x_1, x_2, \cdots , x_n] \in \mathbb{R}^{1 \times n} \, , \quad [x^i] = \begin{bmatrix} x^1 \\ x^2 \\ \vdots \\ x^n \end{bmatrix} \in \mathbb{R}^{n \times 1} \, , \quad \text{with } x_i = x^i \, , \forall i = 1, \cdots , n \tag{9}$$

Instead of automatically associating any matrix variable such as $\boldsymbol{x}$ to the column matrix of its components, the matrix dimensions are clearly indicated as in Eq. (7) and Eq. (9), i.e., by specifying the values m (number of rows) and n (number of columns) of its containing space $\mathbb{R}^{m \times n}$.

---

[51] See, e.g., [78], p. 31, where a "vector" is a column matrix, and a "tensor" is an array with coefficients (elements) having more than two indices, e.g., A_{ijk}. It is important to know the terminologies used in computer-science literature.

[52] The inputs $\boldsymbol{x}$ and the *target* (or labeled) outputs $\boldsymbol{y}$ are the data used to train the network, which produces the predicted (or approximated) output denoted by $\widetilde{\boldsymbol{y}}$ with an overhead tilde reminiscent of the approximation symbol $\approx$. See also Footnote 87.

[53] See, e.g., [107] [109] [110].

[54] See, e.g., [111], Footnote 11. For example, $A_{32} = A^3_2$ is the coefficient in row 3 and column 2.

Consider the Jacobian matrix

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \left[\frac{\partial y_i}{\partial x_j} \right] \in \mathbb{R}^{m \times n} \text{ and let } A^i_j := \frac{\partial y_i}{\partial x_j} \tag{10}$$

where $\boldsymbol{y}$ and $\boldsymbol{x}$ are column matrices shown in Eq. (7). Then the coefficients of this Jacobian matrix are arranged with the upper index i being the row index, and the lower index j being the column index.[55] This convention is natural when converting a chain rule in component form into matrix form, i.e., consider the composition of matrix functions

$$\boldsymbol{z}(\boldsymbol{y}(\boldsymbol{x}(\boldsymbol{\theta}))) \, , \text{ with } \boldsymbol{\theta} \in \mathbb{R}^p \, , \quad \boldsymbol{x} : \mathbb{R}^p \to \mathbb{R}^n \, , \quad \boldsymbol{y} : \mathbb{R}^n \to \mathbb{R}^m \, , \quad \boldsymbol{z} : \mathbb{R}^m \to \mathbb{R}^l \tag{11}$$

where implicitly

$$\mathbb{R}^p \equiv \mathbb{R}^{p \times 1} \, , \quad \mathbb{R}^n \equiv \mathbb{R}^{n \times 1} \, , \quad \mathbb{R}^m \equiv \mathbb{R}^{m \times 1} \, , \quad \mathbb{R}^l \equiv \mathbb{R}^{l \times 1} \tag{12}$$

are the spaces of column matrices, Then using the chain rule

$$\frac{\partial z_i}{\partial \theta_j} = \frac{\partial z_i}{\partial y_r} \frac{\partial y_r}{\partial x_s} \frac{\partial x_s}{\partial \theta_j} \tag{13}$$

where the summation convention on the repeated indices r and s is implied. Then the Jacobian matrix $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{\theta}}$ can be obtained directly as a product of Jacobian matrices from the chain rule just by putting square brackets around each factor:

$$\frac{\partial \boldsymbol{z}(\boldsymbol{y}(\boldsymbol{x}(\boldsymbol{\theta})))}{\partial \boldsymbol{\theta}} = \left[\frac{\partial z_i}{\partial \theta_j} \right] = \left[\frac{\partial z_i}{\partial y_r} \right] \left[\frac{\partial y_r}{\partial x_s} \right] \left[\frac{\partial x_s}{\partial \theta_j} \right] \tag{14}$$

Consider the scalar function $E : \mathbb{R}^m \to \mathbb{R}$ that maps the column matrix $\boldsymbol{y} \in \mathbb{R}^m$ into a scalar, then the components of the gradient of E with respect to $\boldsymbol{y}$ are arranged in a row matrix defined as follows:

$$\left[\frac{\partial E}{\partial y_j} \right]_{1 \times m} = \left[\frac{\partial E}{\partial y_1}, \cdots, \frac{\partial E}{\partial y_m} \right] = \nabla^T_{\boldsymbol{y}} E \in \mathbb{R}^{1 \times m} \tag{15}$$

with $\nabla^T_{\boldsymbol{y}} E$ being the transpose of the $m \times 1$ column matrix $\nabla_{\boldsymbol{y}} E$ containing these same components.[56]

Now consider this particular scalar function below:[57]

$$Z = \boldsymbol{w}\boldsymbol{y} \text{ with } \boldsymbol{w} \in \mathbb{R}^{1 \times n} \text{ and } \boldsymbol{y} \in \mathbb{R}^{n \times 1} \tag{16}$$

Then the gradients of z are[58]

$$\frac{\partial z}{\partial w_j} = y_j \implies \left[\frac{\partial z}{\partial w_j} \right] = [y_j] = \boldsymbol{y}^T \in \mathbb{R}^{1 \times n} \text{ and } \left[\frac{\partial z}{\partial y_j} \right] = [w_j] = \boldsymbol{w} \in \mathbb{R}^{1 \times n} \tag{17}$$

---

[55] For example, the coefficient $A^3_2 = \frac{\partial y_3}{\partial x_2}$ is in row 3 and column 2. The Jacobian matrix in this convention is the transpose of that used in [39], p. 175.

[56] In [78], the column matrix (which is called "vector") $\nabla_{\boldsymbol{y}} E$ is referred to as the gradient of E. Later on in the present paper, E will be called the error or "loss" function, $\boldsymbol{y}$ the outputs of a neural network, and the gradient of E with respect to $\boldsymbol{y}$ is the first step in the "backpropagation" algorithm in Section 5 to find the gradient of E with respect to the network parameters collected in the matrix $\boldsymbol{\theta}$ for an optimization descent direction to minimize E.

[57] Soon, it will be seen in Eq. (26) that the function z is a linear combination of the network inputs $\boldsymbol{y}$, which are outputs coming from the previous network layer, with $\boldsymbol{w}$ being the weights. An advantage of defining $\boldsymbol{w}$ as a row matrix, instead of a column matrix like $\boldsymbol{y}$, is to de-clutter the equations in dispensing of (1) the superscript T designating the transpose as in $\boldsymbol{w}^T \boldsymbol{y}$, or (2) the dot product symbol as in $\boldsymbol{w} \boldsymbol{\cdot} \boldsymbol{y}$, leaving space for other indices, such as in Eq. (26).

[58] The gradients of z will be used in the backpropagation algorithm in Section 5 to obtain the gradient of the error (or loss) function E to find the optimal weights that minimize E.

4.3 Big picture, composition of concepts

A fully-connected feedforward network is a chain of successive applications of functions $f^{(\ell)}$ with $\ell = 1, \ldots, L$, one after another—with L being the number of "layers" or the *depth* of the network—on the input $\boldsymbol{x}$ to produce the predicted output $\widetilde{\boldsymbol{y}}$ for the target output $\boldsymbol{y}$:

$$\widetilde{\boldsymbol{y}} = f(\boldsymbol{x}) = (f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(\ell)} \circ \cdots \circ f^{(2)} \circ f^{(1)})(\boldsymbol{x})$$

$$= f^{(L)}(f^{(L-1)}(\cdots (f^{(\ell)}(\cdots (f^{(2)}(f^{(1)}(\boldsymbol{x}))) \cdots)) \cdots)) \tag{18}$$

or breaking Eq. (18) down, step by step, from inputs to outputs:[59]

$$\boldsymbol{y}^{(0)} = \boldsymbol{x}^{(1)} = \boldsymbol{x} \text{ (inputs)}$$

$$\boldsymbol{y}^{(\ell)} = f^{(\ell)}(\boldsymbol{x}^{(\ell)}) = \boldsymbol{x}^{(\ell+1)} \text{ with } \ell = 1, \cdots, L-1$$

$$\boldsymbol{y}^{(L)} = f^{(L)}(\boldsymbol{x}^{(L)}) = \widetilde{\boldsymbol{y}} \text{ (predicted outputs)} \tag{19}$$

Remark 4.1. The notation $\boldsymbol{y}^{(\ell)}$, for $\ell = 0, \cdots, L$ in Eq. (19) will be useful to develop a concise formulation for the computation of the gradient of the cost function relative to the parameters by backpropagation for use in training (finding optimal parameters); see Eqs. (91)-(92) in Section 5 on Backpropagation. ∎

The quantities associated with layer (ℓ) in a network are indicated with the superscript (ℓ), so that the inputs to layer (ℓ) as gathered in the $m_{(\ell-1)} \times 1$ column matrix

$$\boldsymbol{x}^{(\ell)} = [x_1^{(\ell)}, \cdots, x_{m_{(\ell-1)}}^{(\ell)}]^T = \boldsymbol{y}^{(\ell-1)} \in \mathbb{R}^{m_{(\ell-1)} \times 1} \tag{20}$$

are the *predicted* outputs from the previous layer $(\ell-1)$, gathered in the matrix $\boldsymbol{y}^{(\ell-1)}$, With $m_{(\ell-1)}$ being the *width* of layer $(\ell-1)$. Similarly, the outputs of layer (ℓ) as gathered in the $m_{(\ell)} \times 1$ matrix

$$\boldsymbol{y}^{(\ell)} = [y_1^{(\ell)}, \cdots, y_{m_\ell}^{(\ell)}]^T = \boldsymbol{x}^{(\ell+1)} \in \mathbb{R}^{m_{(\ell)} \times 1} \tag{21}$$

are the inputs to the subsequent layer $(\ell+1)$, gathered in the matrix $\boldsymbol{x}^{(\ell+1)}$, with $m_{(\ell)}$ being the *width* of layer (ℓ).

Remark 4.2. The output for layer (ℓ), denoted by $\boldsymbol{y}^{(\ell)}$, can also be written as $\boldsymbol{h}^{(\ell)}$, where "h" is mnemonic for "hidden", since the inner layers between the input layer (1) and the output layer (L) are considered as being "hidden". Both notations $\boldsymbol{y}^{(\ell)}$ and $\boldsymbol{h}^{(\ell)}$ are equivalent

$$\boldsymbol{y}^{(\ell)} \equiv \boldsymbol{h}^{(\ell)} \tag{22}$$

and can be used interchangeably. In the current Section 4 on "Static, feedforward networks", the notation $\boldsymbol{y}^{(\ell)}$ is used, whereas in Section 7 on "Dynamics, sequential data, sequence modeling", the notation $\boldsymbol{h}^{[n]}$ is used to designate the output of the "hidden cell" at state $[n]$ in a recurrent neural network, keeping in mind the equivalence in Eq. (276) in Remark 7.1. Whenever necessary, readers are reminded of the equivalence in Eq. (22) to avoid possible confusion when reading deep-learning literature. ∎

The above chain in Eq. (18)—see also Eq. (23) and Figure 23—is referred to as "multiple levels of composition" that characterizes modern deep learning, which no longer attempts to mimic the working of the brain from the neuroscientific perspective.[60] Besides, a complete understanding of how the brain functions is still far remote.[61]

---

[59] To alleviate the notation, the predicted output $\boldsymbol{y}^{(\ell)}$ from layer (ℓ) is indicated by the superscript (ℓ), without the tilde. The output $\boldsymbol{y}^{(L)}$ from the last layer (L) is the network predicted output $\widetilde{\boldsymbol{y}}$.

[60] See [78], p. 14, p.163.

[61] In the review paper [12] addressing to computer-science experts, and dense with acronyms and jargon "foreign" to

4.3.1 Graphical representation, block diagrams

A function can be graphically represented as in Figure 22.

$$y = f(x) \quad \text{or} \quad x \xrightarrow{f} y \quad \text{or} \quad x \rightarrow \boxed{f} \rightarrow y$$

Figure 22: *Function mapping, graphical representation* (Section 4.3.1): n inputs in $x \in \mathbb{R}^{n \times 1}$ ($n \times 1$ column matrix of real numbers) are fed into function f to produce m outputs in $y \in \mathbb{R}^{m \times 1}$.

The multiple levels of compositions in Eq. (18) can then be represented by

$$\underbrace{x = y^{(0)}}_{\text{Input}} \underbrace{\xrightarrow{f^{(1)}} y^{(1)} \xrightarrow{f^{(2)}} \cdots y^{(\ell-1)} \xrightarrow{f^{(\ell)}} y^{(\ell)} \cdots \xrightarrow{f^{(L-1)}} y^{(L-1)} \xrightarrow{f^{(L)}}}_{\text{Network as multilevel composition of functions}} \underbrace{y^{(L)} = \widetilde{y}}_{\text{Output}} \tag{23}$$

revealing the structure of the *feedforward network* as a multilevel composition of functions (or chain-based network) in which the output $y^{(\ell-1)}$ of the previous layer $(\ell-1)$ serves as the input for the current layer (ℓ), to be processed by the function $f^{(\ell)}$ to produce the output $y^{(\ell)}$. the input $x = y^{(0)}$ for the input layer (1) is the input for the entire network. The output $y^{(L)} = \widetilde{y}$ of the (last) layer (L) is the predicted output for the entire network.

$$x = y^{(0)} \rightarrow \boxed{f^{(1)}} \rightarrow \quad \cdots \quad y^{(\ell-1)} \rightarrow \boxed{f^{(\ell)}} \rightarrow y^{(\ell)} \cdots \rightarrow \boxed{f^{(L)}} \rightarrow y^{(L)} = \widetilde{y}$$
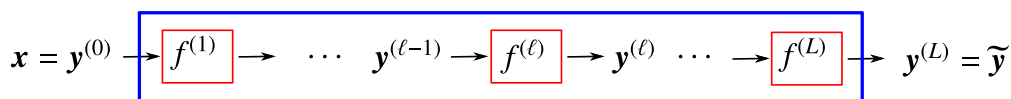
Figure 23: *Feedforward network* (Sections 4.3.1, 4.4.4): Multilevel composition in feedforward network with L layers represented as a sequential application of functions $f^{(\ell)}$, with $\ell = 1, \cdots, L$, to n inputs gathered in $x = y^{(0)} \in \mathbb{R}^{n \times 1}$ ($n \times 1$ column matrix of real numbers) to produce m outputs gathered in $y^{(L)} = \widetilde{y} \in \mathbb{R}^{m \times 1}$. This figure is a higher-level block diagram that corresponds to the lower-level neural network in Figure 7 or in Figure 35.

Remark 4.3. *Layer definitions, action layers, state layers.* In Eq. (23) and in Figure 23, an *action layer* is defined by the action, i.e., the function $f^{(\ell)}$, on the inputs $y^{(\ell-1)}$ to produce the outputs $y^{(\ell)}$. There are L action layers. A *state layer* is a collection of inputs or outputs, i.e., $y^{(\ell)}, \ell = 0, \ldots, L$, each describes a state of the system, thence the number of state layers is $L + 1$, and the number of hidden (state) layers (excluding the input layer $y^{(0)}$ and the output layer $y^{(L)}$) is $(L-1)$. For an illustration of state layers, see [78], p. 6, Figure1.2. See also Remark 11.3. From here on, "hidden layer" means "hidden *state* layer", agreeing with the terminology in [78]. See also Remark 4.5 on depth definitions in Section 4.6.1 on "Depth, size". ∎

4.4 Network layer, detailed construct

4.4.1 Linear combination of inputs and biases

First, an affine transformation on the inputs (see Eq. (26)) is carried out, in which the coefficients of the inputs are called the weights, and the constants are called the biases. The output $y^{(\ell-1)}$ of layer $(\ell-1)$ is the input to layer (ℓ)

$$y^{(\ell-1)} = [y_1^{(\ell-1)}, \cdots, y_{m_{(\ell-1)}}^{(\ell-1)}]^T \in \mathbb{R}^{m_{(\ell-1)} \times 1} . \tag{24}$$

first-time learners, the authors mentioned "It is ironic that artificial NNs [neural networks] (ANNs) can help to better understand biological NNs (BNNs)", and cited a 2012 paper that won an "image segmentation" contest in helping to construct a 3-D model of the "brain's neurons and dendrites" from "electron microscopy images of stacks of thin slices of animal brains".

The column matrix $\boldsymbol{z}^{(\ell)}$

$$\boldsymbol{z}^{(\ell)} = [z_1^{(\ell)}, \cdots, z_{m_{(\ell)}}^{(\ell)}]^T \in \mathbb{R}^{m_{(\ell)} \times 1} \tag{25}$$

is a linear combination of the inputs in $\boldsymbol{y}^{(\ell-1)}$ plus the biases (i.e., an affine transformation)[62]

$$\boxed{\boldsymbol{z}^{(\ell)} = \boldsymbol{W}^{(\ell)}\boldsymbol{y}^{(\ell-1)} + \boldsymbol{b}^{(\ell)} \text{ such that } z_i^{(\ell)} = \boldsymbol{w}_i^{(\ell)}\boldsymbol{y}^{(\ell-1)} + b_i^{(\ell)} \text{, for } i = 1, \ldots, m_{(\ell)} \text{,}} \tag{26}$$

where the $m_{(\ell)} \times m_{(\ell-1)}$ matrix $\boldsymbol{W}$ contains the weights[63]

$$\boldsymbol{w}_i^{(\ell)} = [w_{i1}^{(\ell)}, \cdots, w_{im_{(\ell-1)}}^{(\ell)}] \in \mathbb{R}^{m_{(\ell-1)} \times 1} \,, \quad \boldsymbol{W}^{(\ell)} = [\boldsymbol{w}_1^{(\ell)}; \cdots; \boldsymbol{w}_{m_{(\ell)}}^{(\ell)}] = \begin{bmatrix} \boldsymbol{w}_1^{(\ell)} \\ \vdots \\ \boldsymbol{w}_{m_{(\ell)}}^{(\ell)} \end{bmatrix} \in \mathbb{R}^{m_{(\ell)} \times m_{(\ell-1)}} \,, \tag{27}$$

and the $m_{(\ell)} \times 1$ column matrix $\boldsymbol{b}^{(\ell)}$ the biases:[64]

$$\boldsymbol{z}^{(\ell)} = [z_1^{(\ell)}, \cdots, z_{m_{(\ell)}}^{(\ell)}]^T \in \mathbb{R}^{m_{(\ell)} \times 1} \,, \quad \boldsymbol{b}^{(\ell)} = [b_1^{(\ell)}, \cdots, b_{m_{(\ell)}}^{(\ell)}]^T \in \mathbb{R}^{m_{(\ell)} \times 1} \,. \tag{28}$$

Both the weights and the biases are collectively known as the network parameters, defined in the following matrices for layer (ℓ):

$$\boldsymbol{\theta}_i^{(\ell)} = \left[\theta_{ij}^{(\ell)}\right] = \left[\theta_{i1}^{(\ell)}, \ldots, \theta_{im_{(\ell-1)}}^{(\ell)} \mid \theta_{i(m_{(\ell-1)}+1)}^{(\ell)}\right] = \left[w_{i1}^{(\ell)}, \ldots, w_{im_{(\ell-1)}}^{(\ell)} \mid b_i^{(\ell)}\right]$$

$$= \left[\boldsymbol{w}_i^{(\ell)} \mid b_i^{(\ell)}\right] \in \mathbb{R}^{1 \times [m_{(\ell-1)}+1]} \tag{29}$$

$$\boldsymbol{\Theta}^{(\ell)} = \left[\theta_{ij}^{(\ell)}\right] = \begin{bmatrix} \boldsymbol{\theta}_1^{(\ell)} \\ \vdots \\ \boldsymbol{\theta}_{m_{(\ell)}}^{(\ell)} \end{bmatrix} = \left[\boldsymbol{W}^{(\ell)} \mid \boldsymbol{b}^{(\ell)}\right] \in \mathbb{R}^{m_{(\ell)} \times [m_{(\ell-1)}+1]} \tag{30}$$

For simplicity and convenience, the set of all parameters in the network is denoted by $\boldsymbol{\theta}$, and the set of all parameters in layer (ℓ) by $\boldsymbol{\theta}^{(\ell)}$:[65]

$$\boldsymbol{\theta} = \{\boldsymbol{\theta}^{(1)}, \cdots, \boldsymbol{\theta}^{(\ell)}, \cdots, \boldsymbol{\theta}^{(L)}\} = \{\boldsymbol{\Theta}^{(1)}, \cdots, \boldsymbol{\Theta}^{(\ell)}, \cdots, \boldsymbol{\Theta}^{(L)}\} \text{, such that } \boldsymbol{\theta}^{(\ell)} \equiv \boldsymbol{\Theta}^{(\ell)} \tag{31}$$

Note that the set $\boldsymbol{\theta}$ in Eq. (31) is not a matrix, but a set of matrices, since the number of rows $m_{(\ell)}$ for a layer (ℓ) may vary for different values of ℓ, even though in practice, the widths of the layers in a fully connected feed-forward network may generally be chosen to be the same.

---

[62] See Eq. (497) for the continuous temporal summation, counterpart of the discrete spatial summation in Eq. (26).

[63] It should be noted that the use of both $\boldsymbol{W}$ and $\boldsymbol{W}^T$ in [78] in equations equivalent to Eq. (26) is confusing. For example, on p. 205, in Section 6.5.4 on backpropagation for fully-connected feedforward network, Algorithm 6.3, an equation that uses $\boldsymbol{W}$ in the same manner as Eq. (26) is $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)}\boldsymbol{h}^{(k)}$, whereas on p. 191, in Section 6.4, Architecture Design, Eq. (6.40) uses $\boldsymbol{W}^T$ and reads as $\boldsymbol{h}^{(1)} = G^{(1)}(\boldsymbol{W}^{(1)T}\boldsymbol{x} + \boldsymbol{b}^{(1)})$, which is similar to Eq. (6.36) on p. 187. On the other hand, both $\boldsymbol{W}$ and $\boldsymbol{W}^T$ appear on the same p. 190 in the expressions $\cos(\boldsymbol{Wx} + \boldsymbol{b})$ and $\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b})$. Here, we stick to a single definition of $\boldsymbol{W}^{(\ell)}$ as defined in Eq. (27) and used in Eq. (26).

[64] Eq. (26) is a linear (additive) combination of inputs with possibly non-zero biases. An additive combination of inputs with zero bias, and a "multiplicative" combination of inputs of the form $z_i^{(\ell)} = \prod_{k=1}^{k=m_{(\ell)}} w_{ik}^{(\ell)} y_k^{(\ell)}$ with zero bias, were mentioned in [12]. In [112], the author went even further to propose the general case in which $\boldsymbol{y}^{(\ell)} = \mathcal{F}^{(\ell)}(\boldsymbol{y}^{(k)}, \text{ with } k < \ell)$, where $\mathcal{F}_i^{(\ell)}$ is *any* differentiable function. But it is not clear whether any of these more complex functions of the inputs were used in practice, as we have not seen any such use, e.g., in [21] [78], and many other articles, including review articles such as [13] [20]. On the other hand, the additive combination has a clear theoretical foundation as the linear-order approximation to the Volterra series Eq. (496); see Eq. (497) and also [19].

[65] For the convenience in further reading, wherever possible, we use the same notation as in [78], p.xix.

Similar to the definition of the parameter matrix $\boldsymbol{\theta}^{(\ell)}$ in Eq. (30), which includes the biases $\boldsymbol{b}^{(\ell)}$, it is convenient for use later in elucidating the backpropagation method in Section 5 (and Section 5.2 in particular) to expand the matrix $\boldsymbol{y}^{(\ell-1)}$ in Eq. (26) into the matrix $\overline{\boldsymbol{y}}^{(\ell-1)}$ (with an overbar) as follows:

$$z^{(\ell)} = W^{(\ell)}y^{(\ell-1)} + b^{(\ell)} \equiv \left[W^{(\ell)} \,\middle|\, b^{(\ell)} \right] \begin{bmatrix} y^{(\ell-1)} \\ 1 \end{bmatrix} =: \theta^{(\ell)} \, \overline{y}^{(\ell-1)} \,, \tag{32}$$

with

$$\boldsymbol{\theta}^{(\ell)} := \left[\boldsymbol{W}^{(\ell)} \,\middle|\, \boldsymbol{b}^{(\ell)} \right] \in \mathbb{R}^{m_{(\ell)} \times [m_{(\ell-1)}+1]} \,, \text{ and } \overline{\boldsymbol{y}}^{(\ell-1)} := \begin{bmatrix} \boldsymbol{y}^{(\ell-1)} \\ 1 \end{bmatrix} \in \mathbb{R}^{[m_{(\ell-1)}+1] \times 1} \,. \tag{33}$$

The total number of parameters of a fully-connected feedforward network is then

$$P_T := \sum_{\ell=1}^{\ell=L} m_{(\ell)} \times [m_{(\ell-1)} + 1] \,. \tag{34}$$

But why using a linear (additive) combination (or superposition) of inputs with weights, plus biases, as expressed in Eq. (26) ? See Section 13.2.

4.4.2 Activation functions

An activation function $a : \mathbb{R} \to \mathbb{R}$, which is a nonlinear real-valued function, is used to decide when the information in its argument is relevant for a neuron to activate. In other words, an activation function filters out information deemed insignificant, and is applied *element-wise* to the matrix $\boldsymbol{z}^{(\ell)}$ in Eq. (26), obtained as a linear combination of the inputs plus the biases:

$$\boxed{\boldsymbol{y}^{(\ell)} = a(\boldsymbol{z}^{(\ell)}) \text{ such that } y_i^{(\ell)} = a(z_i^{(\ell)})} \tag{35}$$

Without the activation function, the neural network is simply a linear regression, and cannot learn and perform complex tasks, such as image classification, language translation, guiding a driver-less car, etc. See Figure 32 for the block diagram of a one-layer network.

An example is a linear one-layer network, without activation function, being unable to represent the seemingly simple XOR (exclusive-or) function, which brought down the first wave of AI (cybernetics), and that is described in Section 4.5.

Rectified linear units (ReLU). Nowadays, for the choice of activation function $a(\cdot)$, Most modern *large* deep-learning networks use the default,[66] well-proven *rectified linear function* (more often known as the "positive part" function) defined as[67]

$$a(z) = z^+ = [z]_+ = \max(0, z) = \begin{cases} 0 & \text{for } z \leq 0 \\ z & \text{for } 0 < z \end{cases} \tag{36}$$

and depicted in Figure 24, for which the processing unit is called the *rectified linear unit* (ReLU),[68] which was demonstrated to be superior to other activation functions in many problems.[69] Therefore, in this section, we discuss in detail the rectified linear function, with careful explanation and motivation. It is important to

---

[66] "In modern neural networks, The default recommendation is to use the rectified linear unit, or ReLU," [78], p. 168.
[67] The notation z^+ for positive part function is used in the mathematics literature, e.g., "Positive and negative parts", Wikipedia, version 12:11, 13 March 2018, and less frequently in the computer-science literature, e.g., [33]. The notation $[z]_+$ is found in the neuroscience literature, e.g., [32] [19], p. 63. The notation $\max(0, z)$ is more widely used in the computer-science literature, e.g., [34] [113], [78].
[68] A similar relation can be applied to define the Leaky ReLU in Eq. (40).
[69] In [78], p. 15, the authors cited the original papers [33] and [34], where ReLU was introduced in the context of image / object recognition, and [113], where the superiority of ReLU over hyperbolic-tangent units and sigmoidal units was demonstrated.

note that ReLU is superior for *large* network size, and may have about the same, or less, accuracy than the older logistic sigmoid function for "very small" networks, while requiring less computational efforts.[70]
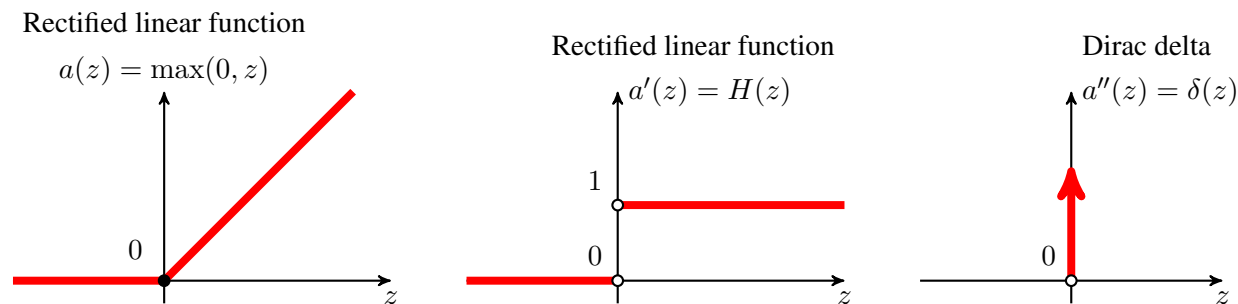
Figure 24: *Activation function* (Section 4.4.2): Rectified linear function and its derivatives. See also Section 5.3.3 and Figure 54 for Parametric ReLU that helped surpass human level performance in ImageNet competition for the first time in 2015, Figure 3 [61]. See also Figure 26 for a halfwave rectifier.

To transform an alternative current into a direct current, the first step is to rectify the alternative current by eliminating its negative parts, and thus The meaning of the adjective "rectified" in *rectified linear unit* (ReLU). Figure 25 shows the current-voltage relation for an ideal diode, for a resistance, which is in series with the diode, and for the resulting ReLU function that rectifies an alternative current as input into the halfwave rectifier circuit in Figure 26, resulting in a halfwave current as output.
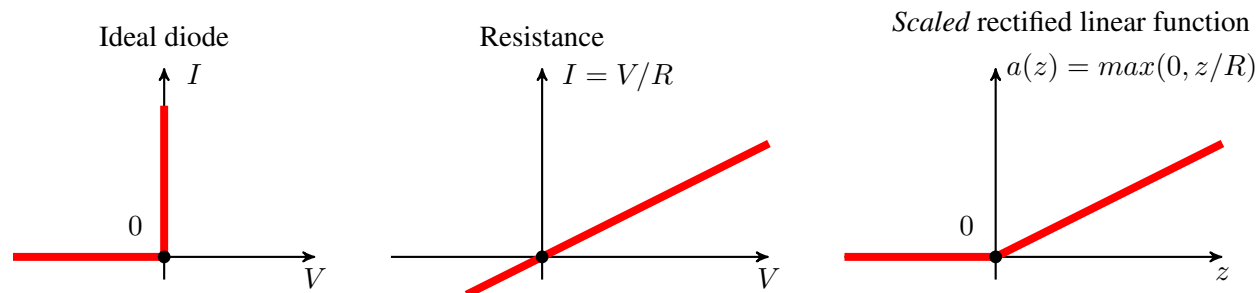
Figure 25: *Current I versus voltage V* (Section 4.4.2): Ideal diode, resistance, *scaled* rectified linear function as activation (transfer) function for the ideal diode and resistance in series. (Figure plotted with $R = 2$.) See also Figure 26 for a halfwave rectifier.

Mathematically, a periodic function remains periodic after passing through a (nonlinear) rectifier (active function):

$$z(x + T) = z(x) \implies y(x + T) = a(z(x + T)) = a(z(x)) = y(x) \tag{37}$$

where T in Eq. (37) is the period of the input current z.

Biological neurons encode and transmit information over long distance by generating (firing) electrical pulses called action potentials or spikes with a wide range of frequencies [19], p. 1; see Figure 27. "To reliably encode a wide range of signals, neurons need to achieve a broad range of firing frequencies and to move smoothly between low and high firing rates" [114]. From the neuroscientific standpoint, the rectified

---

[70] See, e.g., [78], p. 219, and Section 13.3 on the history of active functions. See also Section 4.6 for a discussion of network size. The reason for less computational effort with ReLU is due to (1) it being an identity map for positive argument, (2) zero for negative argument, and (3) its first derivative being the Step (Heaviside) function as shown in Figure 24, and explained in Section 5 on Backpropagation.
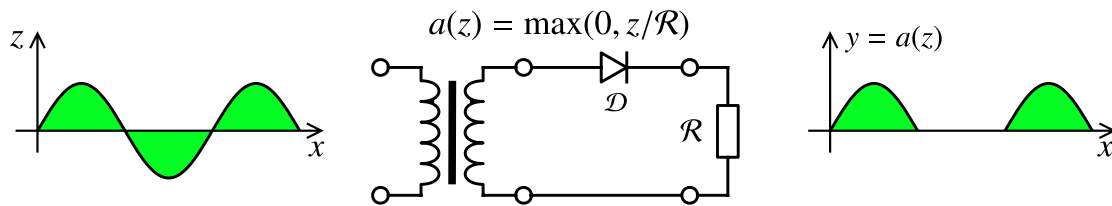
Figure 26: *Halfwave rectifier circuit* (Section 4.4.2), with a primary alternative current z going in as input (left), passing through a transformer to lower the voltage amplitude, with the secondary alternative current out of the transformer being put through a closed circuit with an ideal diode $\mathcal{D}$ and a resistor $\mathcal{R}$ in series, resulting in a halfwave output current, which can be grossly approximated by the *scaled* rectified linear function $y \approx a(z) = \max(0, z/\mathcal{R})$ (right) as shown in Figure 25, with scaling factor $1/\mathcal{R}$. The rectified linear unit in Figure 24 corresponds to the case with $\mathcal{R} = 1$. For a more accurate Shockley diode model, The relation between current I and voltage V for this circuit is given in Figure 29. Figure based on source in Wikipedia, version 01:49, 7 January 2015.

linear function could be motivated as an idealization of the "Type I" relation between the firing rate (F) of a biological neuron and the input current (I), called the FI curve. Figure 27 describes three types of FI curves, with Type I in the middle subfigure, where there is a continuous increase in the firing rate with increase in input current.

The Shockley equation for a current I going through a diode $\mathcal{D}$, in terms of the voltage V_D across the diode, is given in mathematical form as:

$$I = p \left[e^{qV_D} - 1 \right] \implies V_D = \frac{1}{q} \log \left(\frac{I}{p} + 1 \right) . \tag{38}$$

With the voltage across the resistance being $V_R = RI$, the voltage across the diode and the resistance in series is then

$$-V = V_D + V_R = V_D = \frac{1}{q} \log \left(\frac{I}{p} + 1 \right) + RI , \tag{39}$$

which is plotted in Figure 29. The rectified linear function could be seen from Figure 29 as a very rough approximation of the current-voltage relation in a halfwave rectifier circuit in Figure 26, in which a diode and a resistance are in series. In the Shockley model, the diode is leaky in the sense that there is a small amount of current flow when the polarity is reversed, unlike the case of an ideal diode or ReLU (Figure 24), and is better modeled by the Leaky ReLU activation function, in which there is a small positive (instead of just flat zero) slope for negative z:

$$a(z) = \max(0.01z, z) = \begin{cases} 0.01\, z & \text{for } z \leq 0 \\ z & \text{for } 0 < z \end{cases} \tag{40}$$

Prior to the introduction of ReLU, which had been long widely used in neuroscience as activation function prior to 2011,[71] the state-of-the-art for deep-learning activation function was the hyperbolic tangent (Figure 31), which performed better than the widely used, and much older, sigmoid function[72] (Figure 30); see [113], in which it was reported that

---

[71] See, e.g., [19], p. 14, where ReLU was called the "half-wave rectification operation", the meaning of which is explained above in Figure 26. The logistic sigmoid function (Figure 30) was also used in neuroscience since the 1950s.

[72] See Section 13.3.1 for a history of the sigmoid function, which dated back at least to 1974 in neuroscience.
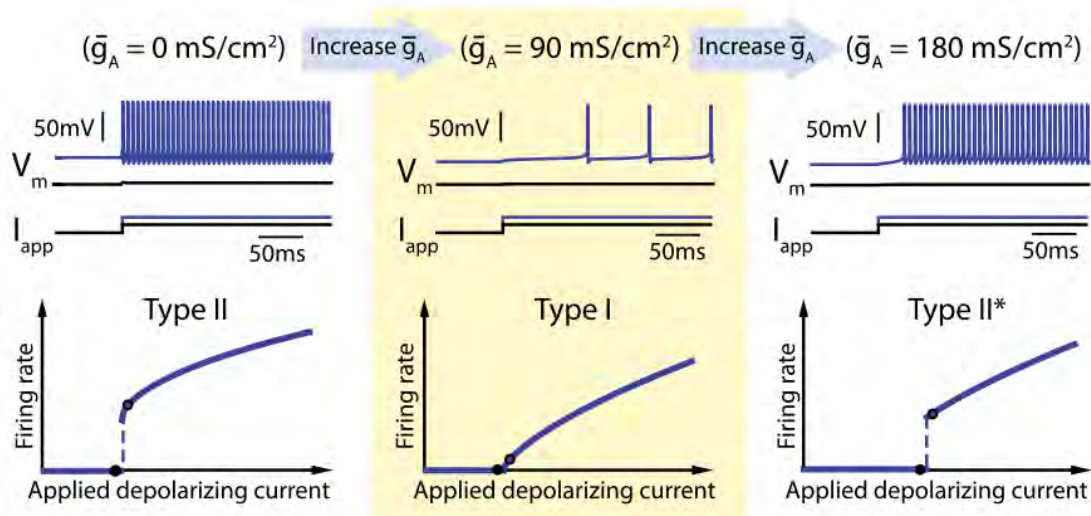
Figure 27: *FI curves* (Sections 4.4.2, 13.2.2). Firing rate frequency (F) versus applied depolarizing current (I), thus FI curves. Three types of FI curves. The time histories of voltage V_m provide a visualization of the spikes, current threshold, and spike firing rates. The applied (input) current I_{app} in increased gradually until it passes a current threshold, then the neuron begins to fire. Two input current levels (two black dots on FI curves at the bottom) near the current threshold are shown, with one just below the threshold (black-line time history for I_{app}) and and one just above the threshold (blue line). two corresponding histories of voltage V_m (flat black line, and blue line with spikes) are also shown. Type I displays a continuous increase in firing frequency from zero to higher values when the current continues to increase past the current threshold. Type II displays a discontinuity in firing frequency, with a sudden jump from zero to a finite frequency, when the current passes the threshold. At low concentration $\bar{g}_A$ of potassium, the neuron exhibits Type-II FI curve, then transitions to Type-I FI curve as $\bar{g}_A$ is increased, and returns to Type-II* for higher concentration $\bar{g}_A$. see [114]. the *scaled* rectified linear unit (scaled ReLU, Figure 25 and Figure 26) can be viewed as approximating Type-I FI curve, see also Figure 28 and Eq. (505) where the FI curve is used in biological neuron firing-rate models. Permission of NAS.

"While logistic sigmoid neurons are more biologically plausible than hyperbolic tangent neurons, the latter work better for training multilayer neural networks. Rectifying neurons are an even better model of biological neurons and yield equal or better performance than hyperbolic tangent networks in spite of the hard non-linearity and non-differentiability at zero."

The hard non-linearity of ReLU is localized at zero, but otherwise ReLU is a very simple function—identity map for positive argument, zero for negative argument—making it highly efficient for computation.

Also, due to errors in numerical computation, it is rare to hit exactly zero, where there is a hard non-linearity in ReLU:

"In the case of $g(z) = \max(0, z)$, the left derivative at $z = 0$ is 0, and the right derivative is 1. Software implementations of neural network training usually return one of the one-sided derivatives rather than reporting that the derivative is undefined or raising an error. This may be heuristically justified by observing that gradient-based optimization on a digital computer is subject to numerical error anyway. When a function is asked to evaluate $g(0)$, it is very unlikely that the underlying value truly was 0. Instead, it was likely to be some small value
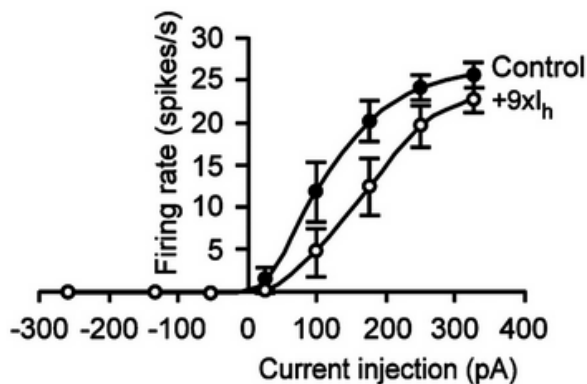
(a) Experiment. Neuron firing rate (F) vs input current (I), FI curve. Hyperpolarization-activated cation current I_h shifts FI curve to the right [115]. Copyright 2004 NAS, with permission of NAS.
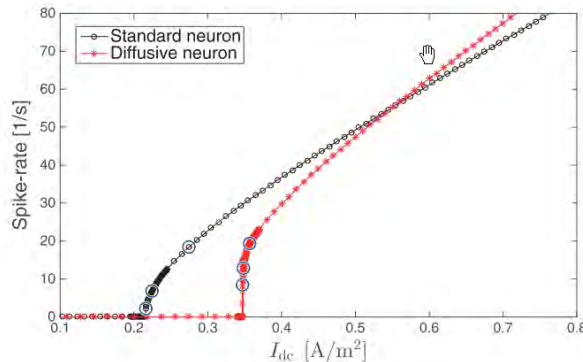
(b) Modeling. FI curve for a Type-I neuron (black, Wilson equations Eqs. (508)-(509)), and FI curve resulting from an aggregation of a large population of 10^4 to 10^5 of these Type-I neurons giving rise to a Type-II response (red, diffusive Wilson equations) [116]. CC-BY-3.0 license.
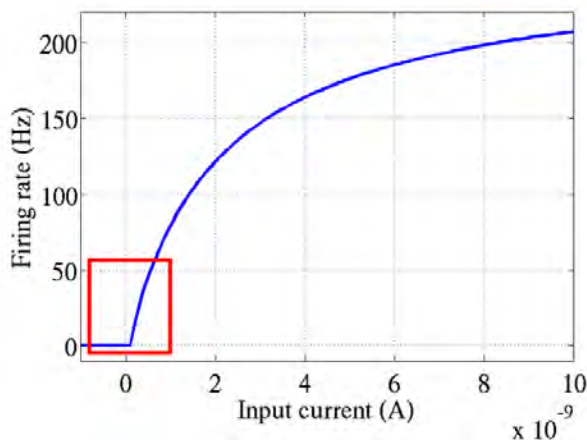
(c) Modeling. FI curve using the Integrate-and-Fire model; see [19], p. 164, Eq. (5.11). The figure is from [113].
(Figure reproduced with permission of the authors.)
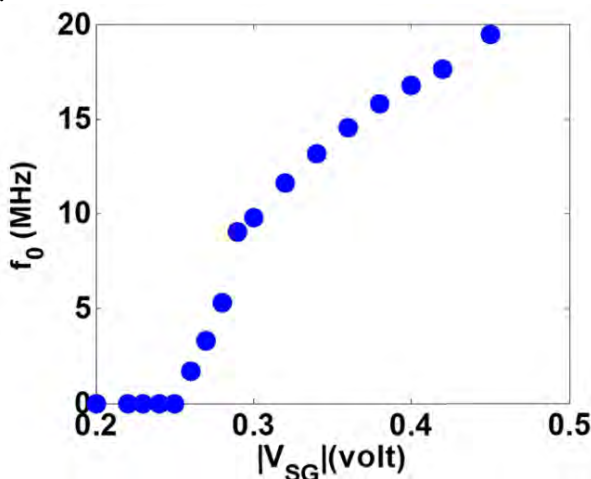
(d) Hardware. Electronic-neuron firing rate (F) vs. Input voltage (V). Very high firing rates up to 20 MHz, roughly 10^5 to 10^6 times faster than firing rate of biological neurons [117], CC-BY-4.0 license.

Figure 28: *FI or FV curves* (Sections 3, 4.4.2, 13.2.2). Neuron firing rate (F) versus input current (I) (FI curves, a,b,c) or voltage (V). The Integrate-and-Fire model in SubFigure (c) can be used to replace the sigmoid function to fit the experimental data points in SubFigure (a). The ReLU function in Figure 24 can be used to approximate the region of the FI curve just beyond the current or voltage thresholds, as indicated in the red rectangle in SubFigure (c). Despite the advanced mathematics employed to produce the Type-II FI curve of a large number of Type-I neurons, as shown in SubFigure (b), it is not clear whether a similar result would be obtained if the single neuron displays a behavior as in Figure 27, with transition from Type II to Type I to Type II* in a single neuron. See Section 13.2.2 on "Dynamic, time dependence, Volterra series" for more discussion on Wilson's equations Eqs. (508)-(509) [118]. On the other hand for deep-learning networks, the above results are more than sufficient to motivate the use of ReLU, which has deep roots in neuroscience.
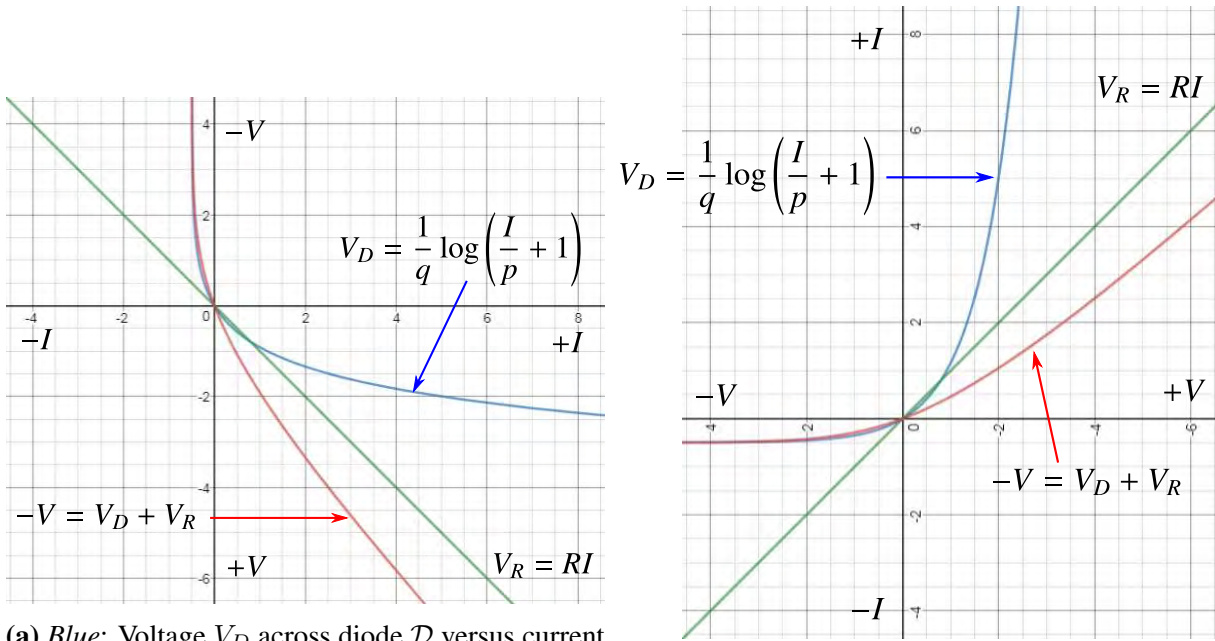
(a) *Blue*: Voltage V_D across diode $\mathcal{D}$ versus current I. *Green*: Voltage V_R across resistance $\mathcal{R}$ versus current I. *Red*: Voltage $-V$ across $\mathcal{D}$ and $\mathcal{R}$ in series versus I.

(b) The same SubFigure (a) rotated by +90 deg to show current $+I$ versus voltage $+V$, as a graphical solution for the nonlinear Eq. (39).

Figure 29: *Halfwave rectifier* (Sections 4.4.2, 5.3.2). Current I versus voltage V [red line in SubFigure (b)] in the halfwave rectifier circuit of Figure 26, for which the ReLU function in Figure 24 is a gross approximation. SubFigure (a) was plotted with $p = 0.5$, $q = -1.2$, $R = 1$. See also Figure 138 for the synaptic response of crayfish similar to the red line in SubFigure (b).

that was rounded to 0." [78], p. 186.

Thus, in addition to the ability to train deep networks, another advantage of using ReLU is the high efficiency in computing both the layer outputs and the gradients for use in optimizing the parameters (weights and biases) to lower cost or loss, i.e., training; see Section 6 on Training, and in particular Section 6.3 on Stochastic Gradient Descent.

The activation function ReLU approximates closer to how biological neurons work than other activation functions (e.g., logistic sigmoid, tanh, etc.), as it was established through experiments some sixty years ago, and have been used in neuroscience long (at least ten years) before being adopted in deep learning in 2011. Its use in deep learning is a clear influence from neuroscience; see Section 13.3 on the history of activation functions, and Section 13.3.2 on the history of the rectified linear function.

Deep-learning networks using ReLU mimic biological neural networks in the brain through a trade-off between two competing properties [113]:

(1) *Sparsity*. Only 1% to 4% of brain neurons are active at any one point in time. Sparsity saves brain energy. In deep networks, "rectifying non-linearity gives rise to real zeros of activations and thus truly sparse representations." Sparsity provides representation robustness in that the non-zero features[73] would have small changes for small changes of the data.

---

[73] See the definition of image "predicate" or image "feature" in Section 13.2.1, and in particular Footnote 302.
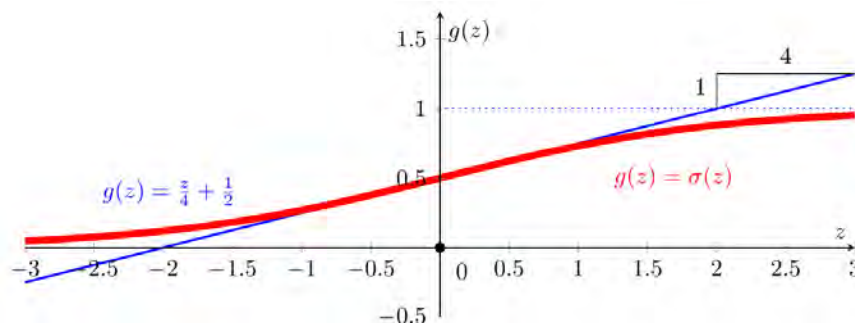
Figure 30: *Logistic sigmoid function* (Sections 4.4.2, 5.1.3, 5.3.1, 13.3.3): $\mathfrak{s}(z) = [1 + \exp(-z)]^{-1} = [\tanh(z/2) + 1]/2$ (red), with the tangent at the origin $z = 0$ (blue). See also Remark 5.3 and Figure 46 on the softmax function.
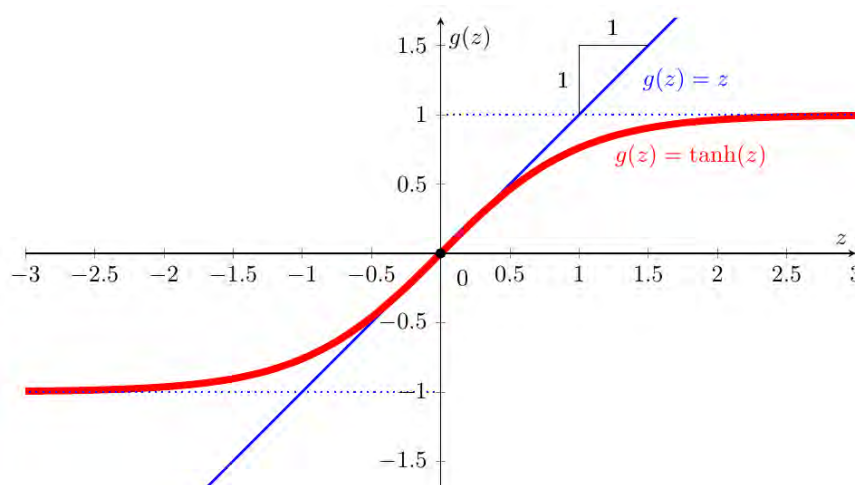
Figure 31: *Hyperbolic tangent function* (Section 4.4.2): $g(z) = \tanh(z) = 2\mathfrak{s}(2z) - 1$ (red) and its tangent $g(z) = z$ at the coordinate origin (blue), showing that this activation function is *identity* for small signals.

(2) *Distributivity.* Each feature of the data is represented distributively by many inputs, and each input is involved in distributively representing many features. Distributed representation is a key concept dated since the revival of connectionism with [119] [120] and others; see Section 13.2.1.

4.4.3 Graphical representation, block diagrams

The block diagram for a one-layer network is given in Figure 32, with more details in terms of the number of inputs and of outputs given in Figure 33.
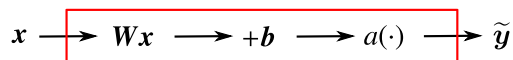
Figure 32: *One-layer network* (Section 4.4.3) representing the relation between the predicted output $\widetilde{y}$ and the input x, i.e., $\widetilde{y} = f(x) = a(Wx + b) = a(z)$, with the weighted sum $z := Wx + b$; see Eq. (26) and Eq. (35) with $\ell = 1$. For a lower-level details of this one layer, see Figure 33.

For a multilayer neural network with L layers, with input-output relation shown in Figure 34, the detailed components are given in Figure 35, which generalizes Figure 33 to layer (ℓ).
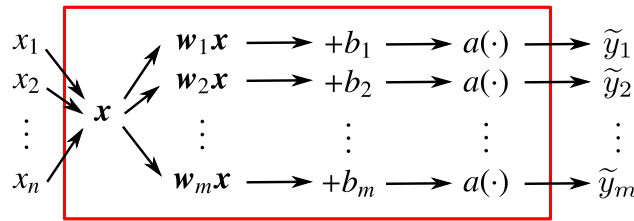
Figure 33: *One-layer network* (Section 4.4.3) in Figure 32: Lower level details, with m processing units (rows or neurons), inputs $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]^T$ and predicted outputs $\widetilde{\boldsymbol{y}} = [\widetilde{y}_1, \widetilde{y}_2, \ldots, \widetilde{y}_m]^T$.

$$\boldsymbol{y}^{(\ell)} = f^{(\ell)}(\boldsymbol{x}^{(\ell)}) = f^{(\ell)}(\boldsymbol{y}^{(\ell-1)}) \quad \text{or} \quad \boldsymbol{y}^{(\ell-1)} = \boldsymbol{x}^{(\ell)} \rightarrow \boxed{f^{(\ell)}} \rightarrow \boldsymbol{y}^{(\ell)}$$

Figure 34: *Input-to-output mapping* (Sections 4.4.3, 4.4.4): Layer (ℓ) in network with L layers in Figure 23, input-to-output mapping $f^{(\ell)}$ for layer (ℓ).
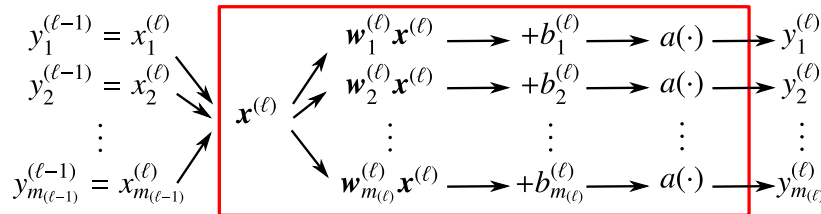
Figure 35: *Low-level details of layer* (ℓ) (Sections 4.4.3, 4.4.4) of the multilayer neural network in Figure 23, with m_ℓ as the number of processing units (rows or neurons), and thus the width of this layer, representing the layer processing (input-to-output) function $f^{(\ell)}$ in Figure 34.

4.4.4 Artificial neuron

And finally, we now complete our *top-down* descent from the big picture of the overall multilayer neural network with L layers in Figure 23, through Figure 34 for a typical layer (ℓ) and Figure 35 for the lower-level details of layer (ℓ), then down to the most basic level, a neuron in Figure 36 as one row in layer (ℓ) in Figure 35.
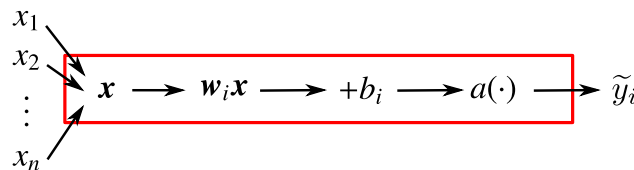
Figure 36: *Artificial neuron* (Sections 2.3.1, 4.4.4, 13.1), row i in layer (ℓ) in Figure 35, representing the multiple-inputs-to-single-output relation $\widetilde{y}_i = a(\boldsymbol{w}_i\boldsymbol{x}+b_i) = a(\sum_{i=1}^{n} w_{ij}x_j + b_i)$ with $\boldsymbol{x} = [x_1, x_2, \cdots, x_n]^T$ and $\boldsymbol{w}_i = [w_{i1}, w_{i2}, \cdots, w_{in}]$. This block diagram is the exact equivalent of Figure 8, Section 2.3.1, and in [38]. See Figure 131 for the corresponding biological neuron in Section 13.1 on "Early inspiration from biological neurons".

4.5 Representing XOR function with two-layer network

The XOR (exclusive-or) function played an important role in bringing down the first wave of AI, known as the cybernetics wave ([78], p. 14) since it was shown in [121] that Rosenblatt's perceptron (1958 [119], 1962 [120] [2]) could not represent the XOR function, defined in Table 2:

Table 2: *Exclusive-or (XOR) function* (Section 4.5) produces the *True* value only if two arguments are different. The symbol ⊕ ("Oh-plus") denotes the XOR operator. A concrete example of the XOR function would be that there is one and only one of two poker player would be the winner, and there is no tie possible, i.e., both players cannot win, and both cannot lose.

j	$\boldsymbol{x}_j = [x_{1j}, x_{2j}]^T$	$y_i = f(\boldsymbol{x}_j) = x_{1j} \oplus x_{2j}$ (XOR)
1	$[0,0]^T$	0
2	$[1,0]^T$	1
3	$[0,1]^T$	1
4	$[1,1]^T$	0

The dataset or design matrix[74] $\boldsymbol{X}$ is the collection of the coordinates of all four points in Table 2:

$$\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_4] = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \in \mathbb{R}^{2 \times 4} \, , \text{ with } \boldsymbol{x}_i \in \mathbb{R}^{2 \times 1} \tag{41}$$

An approximation (or prediction) for the XOR function $y = f(\boldsymbol{x})$ with $\boldsymbol{\theta}$ parameters is denoted by $\widetilde{y} = \widetilde{f}(\boldsymbol{x}, \boldsymbol{\theta})$, with mean squared error (MSE) being:

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{i=1}^{4} \left[\widetilde{f}(\boldsymbol{x}_i, \boldsymbol{\theta}) - f(\boldsymbol{x}_i) \right]^2 = \frac{1}{4} \sum_{i=1}^{4} \left[\widetilde{y}_i - y_i \right]^2 \tag{42}$$

We begin with a one-layer network to show that it cannot represent the XOR function,[75] then move on to a two-layer network, which can.

4.5.1 One-layer network

Consider the following one-layer network,[76] in which the output $\widetilde{y}$ is a linear combination of the coordinates (x_1, x_2) as inputs:

$$\widetilde{y} = f(\boldsymbol{x}, \boldsymbol{\theta}) = f^{(1)}(\boldsymbol{x}) = \boldsymbol{w}_1^{(1)} \boldsymbol{x} + b_1^{(1)} = \boldsymbol{w}\boldsymbol{x} + b = w_1 x_1 + w_2 x_2 + b \, , \tag{43}$$

with the following matrices

$$\boldsymbol{w}_1^{(1)} = \boldsymbol{w} = [w_1, w_2] \in \mathbb{R}^{1 \times 2} \, , \quad \boldsymbol{x} = [x_1, x_2]^T \in \mathbb{R}^{2 \times 1} \, , \quad b_1^{(1)} = b \in \mathbb{R}^{1 \times 1} \tag{44}$$

$$\boldsymbol{\theta} = [\theta_1, \theta_2, \theta_3]^T = [w_1, w_2, b]^T \tag{45}$$

since it is written in [78], p. 14:

> "Model based on the $f(\boldsymbol{x}, \boldsymbol{w}) = \sum_i w_i x_i$ used by the perceptron and ADALINE are called linear models. Linear models have many limitations... Most famously, they cannot learn the XOR function... Critics who observed these flaws in linear models caused a backlash against biologically inspired learning in general (Minsky and Papert, 1969). This was the first major dip in the popularity of neural networks."

---

[74] See [78], p. 103.

[75] This one-layer network is not the Rosenblatt perceptron in Figure 132 due to the absence of the Heaviside function as activation function, and thus Section 4.5.1 is not the proof that the Rosenblatt perceptron cannot represent the XOR function. For such proof, see [121].

[76] See [78], p. 167.

First-time learners, who have not seen the definition of Rosenblatt's (1958) perceptron [119], could confuse Eq. (43) as the perceptron—which was not a linear model, but more importantly the Rosenblatt perceptron was a network with many neurons[77]—because Eq. (43) is only a linear unit (a single neuron), and does not have an (nonlinear) activation function. A neuron in the Rosenblatt perceptron is Eq. (489) in Section 13.2, with the Heaviside (nonlinear step) function as activation function; see Figure 132.

$$\boldsymbol{X} \longrightarrow \boxed{z^{(1)}} \longrightarrow \widetilde{\boldsymbol{y}} \longrightarrow \boxed{J(\boldsymbol{\theta})}$$
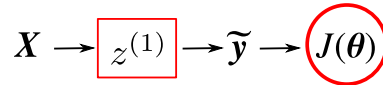
Figure 37: *Representing XOR function* (Sections 4.5, 13.2). This one-layer network (which is not the Rosenblatt perceptron in Figure 132) cannot perform this task. For each input matrix $\boldsymbol{x}_i$ in the design matrix $\boldsymbol{X} = [\boldsymbol{x}_1, \dots, \boldsymbol{x}_4] \in \mathbb{R}^{2 \times 4}$, with $i = 1, \dots, 4$ (see Table 2), the linear unit (neuron) $z^{(1)}(\boldsymbol{x}_i) = \boldsymbol{w}\boldsymbol{x}_i + b \in \mathbb{R}$ in Eq. (43) predict a value $\widetilde{y}_i = z^{(1)}(\boldsymbol{x}_i)$ as output, which is collected in the output matrix $\widetilde{\boldsymbol{y}} = [\widetilde{y}_1, \dots, \widetilde{y}_4] \in \mathbb{R}^{1 \times 4}$. The MSE cost function $J(\boldsymbol{\theta})$ in Eq. (42) is used in a gradient descent to find the parameters $\boldsymbol{\theta} = [\boldsymbol{w}, b]$. The result is a constant function, $\widetilde{y}_i = \frac{1}{2}$, for $i = 1, \dots, 4$, which cannot represent the XOR function.

The MSE cost function in Eq. (42) becomes

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{i=1}^{4} \left[b^2 + (1 - w_1 - b)^2 + (1 - w_2 - b)^2 + (w_1 + w_2 + b)^2 \right]^2 \tag{46}$$

Setting the gradient of the cost function in Eq. (46) to zero and solving the resulting equations, we obtain the weights and the bias:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \left[\frac{\partial J}{\partial \theta_1}, \frac{\partial J}{\partial \theta_2}, \frac{\partial J}{\partial \theta_3} \right]^T = \left[\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \frac{\partial J}{\partial b} \right]^T \tag{47}$$

$$\left. \begin{aligned} \frac{\partial J}{\partial w_1} = 0 &\Longrightarrow 2w_1 + w_2 + 2b = 1 \\ \frac{\partial J}{\partial w_2} = 0 &\Longrightarrow w_1 + 2w_2 + 2b = 1 \\ \frac{\partial J}{\partial b} = 0 &\Longrightarrow w_1 + w_2 + 2b = 1 \end{aligned} \right\} \Longrightarrow w_1 = w_2 = 0, \quad b = \frac{1}{2}, \tag{48}$$

from which the predicted output $\widetilde{y}$ in Eq. (43) is a constant for any points in the dataset (or design matrix) $\boldsymbol{X} = [\boldsymbol{x}_1, \dots, \boldsymbol{x}_4]$:

$$\widetilde{y}_i = f(\boldsymbol{x}_i, \boldsymbol{\theta}) = \frac{1}{2} \text{, for } i = 1, \dots, 4 \tag{49}$$

and thus this one-layer network cannot represent the XOR function. Eqs. (48) are called the "normal" equations.[78]

---

[77] See Section 13.2 on the history of the linear combination (weighted sum) of inputs with biases.

[78] In least-square linear regression, the normal equations are often presented in matrix form, starting from the errors (or residuals) at the data points, gathered in the matrix $\boldsymbol{e} = \boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}$. To minimize the squared of the errors represented by $\| \boldsymbol{e} \|^2$, consider a perturbation $\boldsymbol{\theta}_\epsilon = \boldsymbol{\theta} + \epsilon\gamma$ and $\boldsymbol{e}_\epsilon = \boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}_\epsilon$, then set the directional derivative of $\| \boldsymbol{e} \|^2$ to zero, i.e., $\frac{d}{d\epsilon} \| \boldsymbol{e}_\epsilon \|^2 \big|_{\epsilon=0} = \boldsymbol{X}^T(\boldsymbol{d} - \boldsymbol{X}\boldsymbol{\theta}) = 0$, which is the "normal equation" in matrix form, since the error matrix $\boldsymbol{e}$ is required to be "normal" (orthogonal) to the span of $\boldsymbol{X}$. For the above XOR function with four data points, the relevant matrices are (using the Matlab / Octave notation) $\boldsymbol{e} = [e_1, e_2, e_3, e_4]^T$, $\boldsymbol{y} = [0, 1, 1, 0]^T$, and $\boldsymbol{X} = [[0, 0, 1]; [1, 0, 1]; [0, 1, 1]; [1, 1, 1]]$, which also lead to Eq. (48). See, e.g., [122] [123] and [78], p. 106.

4.5.2 Two-layer network

The four points in Table 2 are not linearly separable, i.e., there is no straight line that separates these four points such that the value of the XOR function is zero for two points on one side of the line, and one for the two points on the other side of the line. One layer could not represent the XOR function, as shown above. Rosenblatt (1958) [119] wrote:

> "It has, in fact, been widely conceded by psychologists that there is little point in trying to 'disprove' any of the major learning theories in use today, since by extension, or a change in parameters, they have all proved capable of adapting to any specific empirical data. In considering this approach, one is reminded of a remark attributed to Kistiakowsky, that '*given seven parameters, I could fit an elephant.*' "

So we now add a second layer, and thus more parameters in the hope to be able to represent the XOR function, as shown in Figure 38.[79]
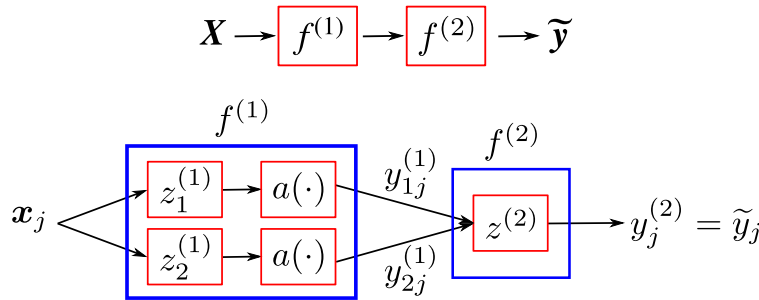
Figure 38: *Representing XOR function* (Sections 4.5). This two-layer network can perform this task. The four points in the design matrix $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_4] \in \mathbb{R}^{2 \times 4}$ (see Table 2) are converted into three points that are linearly separable by the two nonlinear units (neurons or rows) of Layer (1), i.e., $\boldsymbol{Y}^{(1)} = [\boldsymbol{y}_1^{(1)}, \ldots, \boldsymbol{y}_1^{(4)}] = f^{(1)}(\boldsymbol{X}^{(1)}) = a(\boldsymbol{Z}^{(1)}) = \boldsymbol{X}^{(2)} \in \mathbb{R}^{2 \times 4}$, with $\boldsymbol{Z}^{(1)} = \boldsymbol{w}^{(1)} \boldsymbol{X}^{(1)} + \boldsymbol{B}^{(1)} \in \mathbb{R}^{2 \times 4}$, as in Eq. (58), and $a(\cdot)$ a nonlinear activation function. Layer (2) consists of a single linear unit (neuron or row) with three parameters, i.e., $\widetilde{\boldsymbol{y}} = [\widetilde{y}_1, \ldots, \widetilde{y}_4] = f^{(2)}(\boldsymbol{X}^{(2)}) = \boldsymbol{w}_1^{(2)} \boldsymbol{X}^{(2)} + b_1^{(2)} \in \mathbb{R}^{1 \times 4}$. The three non-aligned points in $\boldsymbol{X}^{(2)}$ offer three equations to solve for the three parameters $\boldsymbol{\theta}^{(2)} = [\boldsymbol{w}_1^{(2)}, b_1^{(2)}] \in \mathbb{R}^{1 \times 3}$; see Eq. (61).

Layer (1): six parameters (4 weights, 2 biases), plus a (nonlinear) activation function. The purpose is to change coordinates to move the four input points of the XOR function into three points, such that the two points with XOR value equal 1 are coalesced into a single point, and such that these three points are aligned on a straight line. Since these three points remain not linearly separable, the activation function then moves these three points out of alignment, and thus linearly separable.

$$\boldsymbol{z}_i^{(1)} = \boldsymbol{w}_i^{(1)} \boldsymbol{x}_i + \boldsymbol{b}_i^{(1)} = \boldsymbol{w}^{(1)} \boldsymbol{x}_i + \boldsymbol{b}^{(1)} \text{ , for } i = 1, \ldots, 4 \tag{50}$$

$$\boldsymbol{w}_i^{(1)} = \boldsymbol{w}^{(1)} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \text{ , } \quad \boldsymbol{b}_i^{(1)} = \boldsymbol{b}^{(1)} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \text{ , for } i = 1, \ldots, 4 \tag{51}$$

$$\boldsymbol{Z}^{(1)} = \left[\boldsymbol{z}_1^{(1)}, \ldots, \boldsymbol{z}_4^{(1)} \right] \in \mathbb{R}^{2 \times 4} \text{ , } \quad \boldsymbol{X}^{(1)} = \left[\boldsymbol{x}_1^{(1)}, \ldots, \boldsymbol{x}_4^{(1)} \right] = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \in \mathbb{R}^{2 \times 4} \tag{52}$$

---

[79] Our presentation is more detailed and more general than in [78], pp. 167-171, where there was no intuitive explanation of how the numbers were obtained, and where only the activation function ReLU was used.

$$\boldsymbol{B}^{(1)} = \left[\boldsymbol{b}_1^{(1)}, \ldots, \boldsymbol{b}_4^{(1)} \right] = \begin{bmatrix} 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 \end{bmatrix} \in \mathbb{R}^{2 \times 4} \tag{53}$$

$$\boldsymbol{Z}^{(1)} = \boldsymbol{w}^{(1)} \boldsymbol{X}^{(1)} + \boldsymbol{B}^{(1)} \in \mathbb{R}^{2 \times 4} . \tag{54}$$

To map the two points $\boldsymbol{x}_2 = [1, 0]^T$ and $\boldsymbol{x}_3 = [0, 1]^T$, at which the XOR value is 1, into a single point, the two rows of $\boldsymbol{w}^{(1)}$ are selected to be identically $[1, 1]$ as shown in Eq. (51). The first term in Eq. (54) yields three points aligned along the bisector in the first quadrant (i.e., the line $z_2 = z_1$ in the z-plane), with all positive coordinates, Figure 39:

$$\boldsymbol{w}^{(1)} \boldsymbol{X}^{(1)} = \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 1 & 1 & 2 \end{bmatrix} \in \mathbb{R}^{2 \times 4} . \tag{55}$$
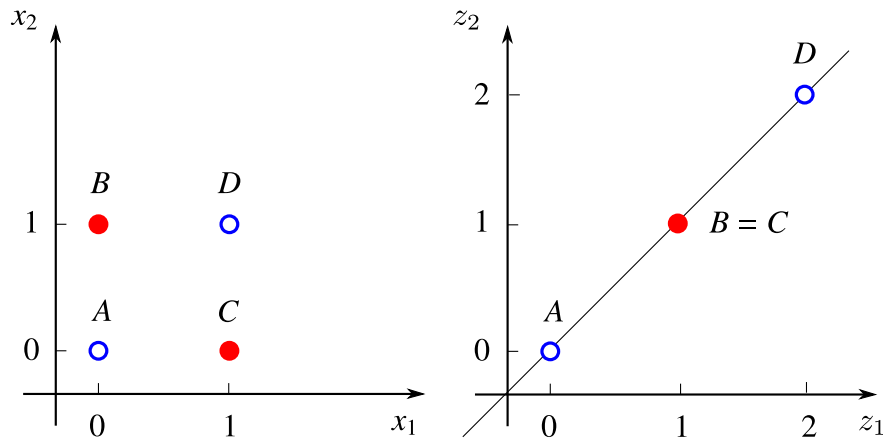
Figure 39: *Two-layer network for XOR representation* (Sections 4.5). *Left*: XOR function, with $A = \boldsymbol{x}_1^{(1)} = [0, 0]^T$, $B = \boldsymbol{x}_2^{(1)} = [0, 1]^T$, $C = \boldsymbol{x}_3^{(1)} = [1, 0]^T$, $D = \boldsymbol{x}_4^{(1)} = [1, 1]^T$; see Eq. (52). The XOR value for the solid red dots is 1, and for the open blue dots 0. *Right*: Images of points A, B, C, D in the z-plane due *only* to the first term of Eq. (54), i.e., $\boldsymbol{w}^{(1)} \boldsymbol{X}^{(1)}$, which is shown in Eq. (55). See also Figure 40.

For activation functions such as ReLu or Heaviside[80] to have any effect, the above three points are next translated in the negative z_2 direction using the biases in Eq. (53), so that Eq. (54) yields:

$$\boldsymbol{Z}^{(1)} = \begin{bmatrix} 0 & 1 & 1 & 2 \\ -1 & 0 & 0 & 1 \end{bmatrix} \in \mathbb{R}^{2 \times 4} , \tag{56}$$

and thus

$$\boldsymbol{Y}^{(1)} = \left[\boldsymbol{y}_1^{(1)}, \ldots, \boldsymbol{y}_4^{(1)} \right] = a(\boldsymbol{Z}^{(1)}) = \begin{bmatrix} 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \boldsymbol{X}^{(2)} = \left[\boldsymbol{x}_1^{(2)}, \ldots, \boldsymbol{x}_4^{(2)} \right] \in \mathbb{R}^{2 \times 4} , \tag{57}$$

For general activation function $a(\cdot)$, the outputs of Layer (1) are:

$$\boldsymbol{Y}^{(1)} = \boldsymbol{X}^{(2)} = a(\boldsymbol{Z}^{(1)}) = \begin{bmatrix} a(0) & a(1) & a(1) & a(2) \\ a(-1) & a(0) & a(0) & a(1) \end{bmatrix} \in \mathbb{R}^{2 \times 4} . \tag{58}$$

---

[80] In general, the Heaviside function is not used as activation function since its gradient is zero, and thus would not work for gradient descent. But for this XOR problem *without* using gradient descent, the Heaviside function offers a workable solution as the rectified linear function.

Layer (2): three parameters (2 weights, 1 bias), no activation function. Eq. (59) for this layer is identical to Eq. (43) for the one-layer network above, with the output $\boldsymbol{y}^{(1)}$ of Layer (1) as input $\boldsymbol{x}^{(2)} = \boldsymbol{y}^{(1)}$, as shown in Eq. (57):

$$\widetilde{y}_j = f(\boldsymbol{x}_j^{(2)}, \boldsymbol{\theta}^{(2)}) = f^{(2)}(\boldsymbol{x}_j^{(2)}) = \boldsymbol{w}_1^{(2)}\boldsymbol{x}_j^{(2)} + b_1^{(2)} = \boldsymbol{w}\boldsymbol{x}_j^{(2)} + b = w_1 x_{1j}^{(2)} + w_2 x_{2j}^{(2)} + b \,, \tag{59}$$

with three distinct points in Eq. (57), because $\boldsymbol{x}_2^{(2)} = \boldsymbol{x}_3^{(2)} = [1, 0]^T$, to solve for these three parameters:

$$\boldsymbol{\theta}^{(2)} = [\boldsymbol{w}_1^{(2)}, b_1^{(2)}] = [w_1, w_2, b] \tag{60}$$

We have three equations:

$$\begin{bmatrix} \widetilde{y}_1 \\ \widetilde{y}_2 \\ \widetilde{y}_4 \end{bmatrix} = \begin{bmatrix} a(0) & a(-1) & 1 \\ a(1) & a(0) & 1 \\ a(2) & a(1) & 1 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \,, \tag{61}$$

for which the exact analytical solution for the parameters $\boldsymbol{\theta}^{(2)}$ is easy to obtain, but the expressions are rather lengthy. Hence, here we only give the numerical solution for $\boldsymbol{\theta}^{(2)}$ in the case of the logistic sigmoid function in Table 3.

Table 3: *Two-layer network for XOR representation* (Section 4.5). Values of parameters $\boldsymbol{\theta}^{(2)}$ in Eq. (60). The results are exact for ReLU and Heaviside, but rounded for sigmoid due to the irrational Euler's number e. See Figure 40.

Activation function	Parameters $\boldsymbol{\theta}^{(2)}$
ReLU	$[1, -2, \quad 0]^T$
Heaviside	$[1, -1, \quad 0]^T$
Sigmoid	$[24.5942, -20.2663, -6.8466]^T$
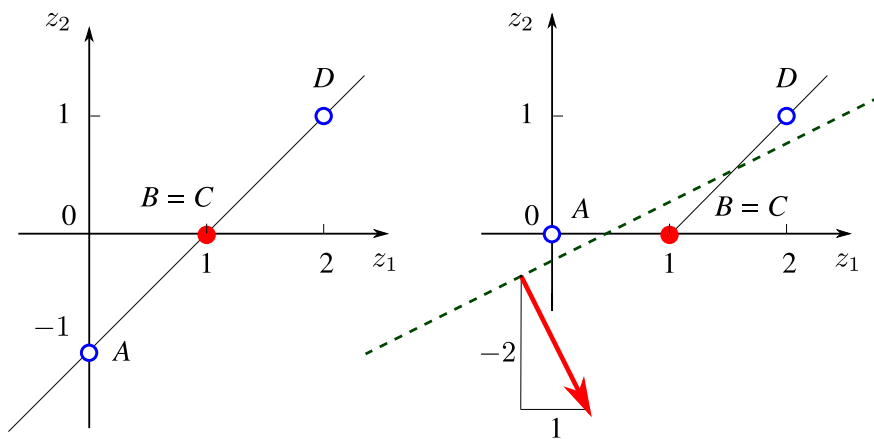
Figure 40: *Two-layer network for XOR representation* (Sections 4.5). *Left*: Images of points A, B, C, D of $\boldsymbol{Z}^{(1)}$ in Eq. (56), obtained after a translation by adding the bias $\boldsymbol{b}^{(1)} = [0, -1]^T$ in Eq. (51) to the same points A, B, C, D in the right subfigure of Figure 39. The XOR value for the solid red dots is 1, and for the open blue dots 0. *Right*: Images of points A, B, C, D after applying the ReLU activation function, which moves point A to the origin; see Eq. (57). the points A, B (= C), D are no longer aligned, and thus linearly separable by the green dotted line, whose normal vector has the components $[1, -2]^T$, which are the weights shown in Table 3.

We conjecture that any (nonlinear) function $a(\cdot)$ in the zoo of activation functions listed, e.g., in "Activation function", Wikipedia, version 21:00, 18 May 2019 or in [36] (see Figure 139), would move the three points in $\boldsymbol{Z}^{(1)}$ in Eq. (56) out of alignment, and thus provide the corresponding unique solution $\boldsymbol{\theta}^{(2)}$ for Eq. (61).

Remark 4.4. *Number of parameters.* In 1953, Physicist Freeman Dyson (Princeton Institute of Advanced Study) once consulted with Nobel Laureate Enrico Fermi about a new mathematical model for a difficult physics problem that Dyson and his students had just developed. Fermi asked Dyson how many parameters they had. "Four", Dyson replied. Fermi then gave his now famous comment "I remember my friend Johnny von Neumann used to say, with four parameters I can fit an elephant, and with five I can make him wiggle his trunk" [124].

But it was only more than sixty years later that physicists were able to plot an elephant in 2-D using a model with four complex numbers as parameters [125].

With nine parameters, the elephant can be made to walk (representing the XOR function), and with a billion parameters, it may even perform some acrobatic maneuver in 3-D; see Section 4.6 on depth of multilayer networks.   ■

4.6 What is "deep" in "deep networks" ? Size, architecture

4.6.1 Depth, size

The concept of network depth turns out to be more complex than initially thought. While for a *fully-connected* feedforward neural network (in which all outputs of a layer are connected to a neuron in the following layer), depth could be considered as the number of layers, there is in general no consensus on the accepted definition of depth. It was stated in [78], p. 8, that:[81]

> "There is no single correct value for the depth of an architecture,[82] just as there is no single correct value for the length of a computer Program. Nor is there a consensus about how much depth a model requires to Qualify as "deep." "

For example, keeping the number of layers the same, then the "depth" of a *sparsely-connected* feedforward network (in which not all outputs of a layer are connected to a neuron in the following layer) should be smaller than the "depth" of a *fully-connected* feedforward network.

The lack of consensus on the boundary between "shallow" and "deep" networks is echoed in [12]:

> "At which problem depth does *Shallow Learning* end, and *Deep Learning* begin? Discussions with DL experts have not yet yielded a conclusive response to this question. Instead of committing myself to a precise answer, let me just define for the purposes of this overview: problems of depth > 10 require *Very Deep Learning*."

---

[81] There are two viewpoints on the definition of depth, one based on the computational graph, and one based on the conceptual graph. From the computational-graph viewpoint, depth is the number of sequential instructions that must be executed in an architecture. From the conceptual-graph viewpoint, depth is the number of concept levels, going from simple concepts to more complex concepts. See also [78], p. 163, for the depth of fully-connected feedforward networks as the "length of the chain" in Eq. (18) and Eq. (23), which is the number of layers.

[82] There are several different network architectures. *Convolutional neural networks* (CNN) use sparse connections, have achieved great success in image recognition, and contributed to the burst of interest in deep learning since winning the ImageNet competion in 2012 by almost halving the image classification error rate; see [13, 12, 75]. *Recurrent neural networks* (RNN) are used to process a sequence of inputs to a system with changing states as in a dynamical system, to be discussed in Section 7. there are other networks with skip connections, in which information flows from layer (ℓ) to layer $(\ell + 2)$, skipping layer $(\ell + 1)$; see [78], p. 196.

Remark 4.5. *Action depth, state depth.* In view of Remark 4.3, which type of layer (action or state) were they talking about in the above quotation? We define here *action depth* as the number of action layers, and *state depth* as the number of state layers. The abstract network in Figure 23 has action depth L and state depth $(L + 1)$, with $(L - 1)$ as the number of hidden (state) layers. ∎

The review paper [13] was attributed in [38] for stating that "training neural networks with more than three hidden layers is called deep learning", implying that a network is considered "deep" if its number of hidden (state) layers $(L - 1) > 3$. In the work reported in [38], the authors used networks with number of hidden (state) layers $(L - 1)$ varying from one to five, and with a constant hidden (state) layer width of $m_{(\ell)} = 50$, for all hidden (state) layers $\ell = 1, \ldots, 5$; see Table 1 in [38], reproduced in Figure 99 in Section 10.2.2.

An example of recognizing multidigit numbers in photographs of addresses, in which the test accuracy increased (or test error decreased) with increasing depth, is provided in [78], p. 196; see Figure 41.
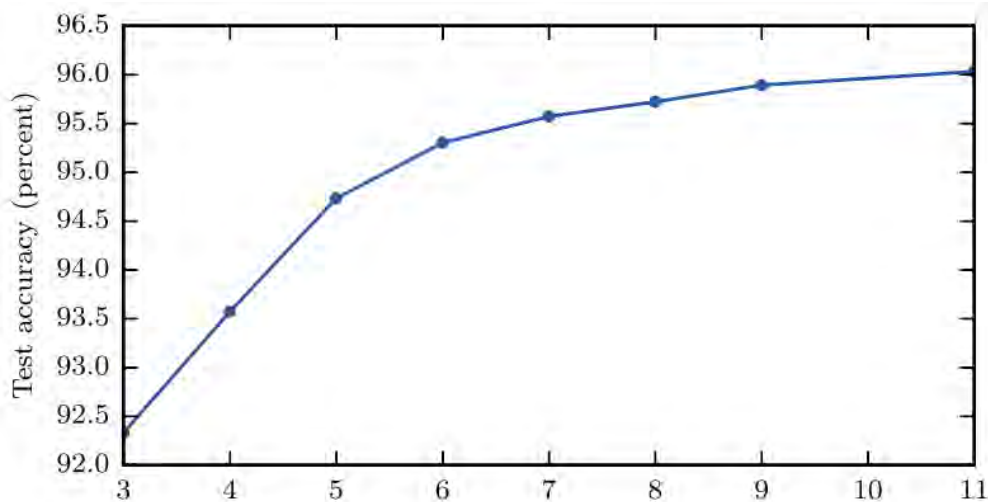
Figure 41: *Test accuracy versus network depth* (Section 4.6.1), showing that test accuracy for this example increases monotonically with the network depth (number of layers). [78], p. 196. (Figure reproduced with permission of the authors.)

But it is not clear where in [13] that it was actually said that a network is "deep" if the number of hidden (state) layers is greater than three. An example in image recognition having more than three layers was, however, given in [13] (emphases are ours):

> "An image, for example, comes in the form of an array of pixel values, and the learned features in the *first layer* of representation typically represent the presence or absence of edges at particular orientations and locations in the image. The *second layer* typically detects motifs by spotting particular arrangements of edges, regardless of small variations in the edge positions. The *third layer* may assemble motifs into larger combinations that correspond to parts of familiar objects, and *subsequent layers* would detect objects as combinations of these parts."

But the above was not a criterion for a network to be considered as "deep". It was further noted on the number of the model parameters (weights and biases) and the size of the training dataset for a "typical deep-learning system" as follows [13] (emphases are ours):

> " In a typical *deep-learning* system, there may be *hundreds of millions* of these adjustable *weights*, and *hundreds of millions* of labelled examples with which to train the machine."

See Remark 7.2 on recurrent neural networks (RNNs) as equivalent to "very deep feedforward networks". Another example was also provided in [13]:

> "Recent ConvNet [convolutional neural network, or CNN][83] architectures have 10 to 20 layers of ReLUs [rectified linear units], hundreds of millions of weights, and billions of connections between Units."[84]

A neural network with 160 billion parameters was perhaps the largest in 2015 [126]:

> "Digital Reasoning, a cognitive computing company based in Franklin, Tenn., recently announced that it has trained a neural network consisting of 160 billion parameters—more than 10 times larger than previous neural networks.
>
> The Digital Reasoning neural network easily surpassed previous records held by Google's 11.2-billion parameter system and Lawrence Livermore National Laboratory's 15-billion parameter system."

As mentioned above, for general network architectures (other than feedforward networks), not only that there is no consensus on the definition of depth, there is also no consensus on how much depth a network must have to qualify as being "deep"; see [78], p. 8, who offered the following intentionally vague definition:

> "Deep learning can be safely regarded as the study of models that involve a greater amount of composition of either learned functions or learned concepts than traditional machine learning does."

Figure 42 depicts the increase in the number of neurons in neural networks over time, from 1958 (Network 1 by Rosenblatt (1958) [119] in Figure 42 with one neuron, which was an error in [78], as discussed in Section 13.2) to 2014 (Network 20 GoogleNet with more than one million neurons), which was still far below the more than ten million biological neurons in a frog.

4.6.2 Architecture

The architecture of a network is the number of layers (depth), the layer width (number of neurons per layer), and the connection among the neurons.[85] We have seen the architecture of fully-connected feedforward neural networks above; see Figure 23 and Figure 35.

One example of an architecture different from that fully-connected feedforward networks is convolutional neural networks, which are based on the convolutional integral (see Eq. (497) in Section 13.2.2 on "Dynamic, time dependence, Volterra series"), and which had proven to be successful long before deep-learning networks:

> "Convolutional networks were also some of the first neural networks to solve important commercial applications and remain at the forefront of commercial applications of deep learning today. By the end of the 1990s, this system deployed by NEC was reading over 10 percent of all the checks in the United States. Later, several OCR and handwriting recognition systems based on convolutional nets were deployed by Microsoft." [78], p. 360.

---

[83] A special type of deep network that went out of favor, then now back in favor, among the computer-vision and machine-learning communities after the spectacular success that ConvNet garnered at the 2012 ImageNet competition; see [13] [75] [74]. Since we are reviewing in detail some specific applications of deep networks to computational mechanics, we will not review ConvNet here, but focus on MultiLayer Neural (MLN)—also known as MultiLayer Perceptron (MLP)—networks.

[84] A network processing "unit" is also called a "neuron".

[85] See [78], p. 166.
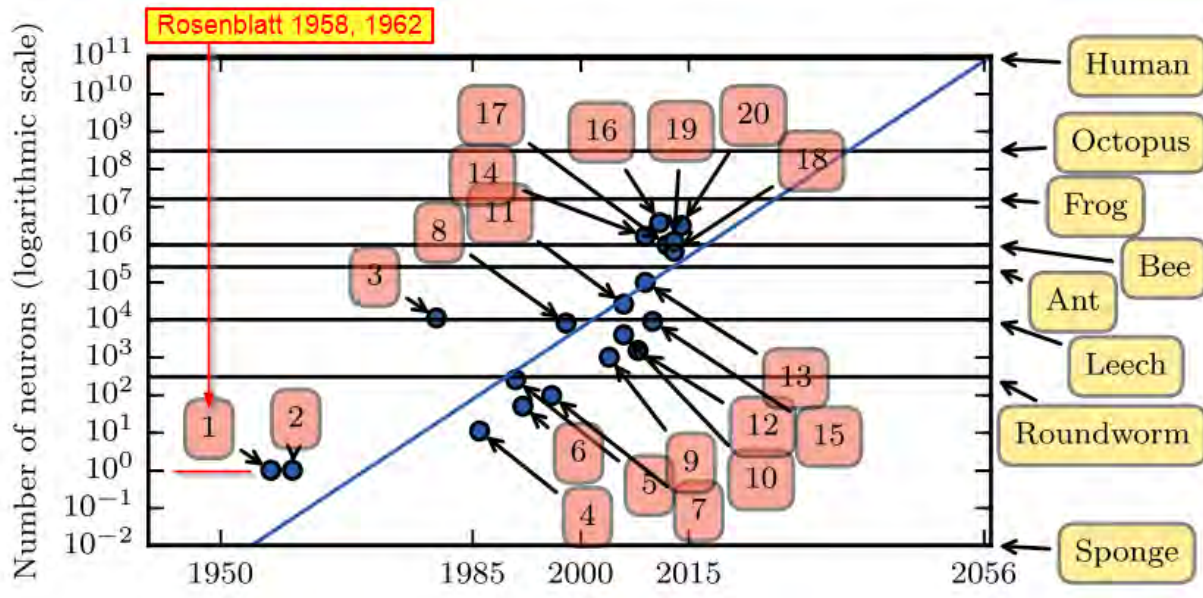
Figure 42: *Increasing network size over time* (Section 4.6.1, 13.2). All networks before 2015 had their number of neurons smaller than that of a frog at 1.6×10^7, and still far below that in a human brain at 8.6×10^{10}; see "List of animals by number of neurons", Wikipedia, version 02:46, 9 May 2019. In [78], p. 23, it was estimated that neural network size would double every 2.4 years (a clear parallel to Moore's law, which stated that the number of transistors on integrated circuits doubled every 2 years). It was mentioned in [78], p. 23, that Network 1 by Rosenblatt (1958 [119], 1962 [2]) as having one neuron (see figure above), which was incorrect, since Rosenblatt (1957) [1] conceived a network with 1000 neurons, and even built the Mark I computer to run this network; see Section 13.2 and Figure 133. (Figure reproduced with permission of the authors.)

"Fully-connected networks were believed not to work well. It may be that the primary barriers to the success of neural networks were psychological (practitioners did not expect neural networks to work, so they did not make a serious effort to use neural networks). Whatever the case, it is fortunate that convolutional networks performed well decades ago. In many ways, they carried the torch for the rest of deep learning and paved the way to the acceptance of neural networks in general." [78], p. 361.

Here, we present a more recent and successful network architecture different from the fully-connected feedforward network. Residual network was introduced in [127] to address the problem of vanishing gradient that plagued "very deep" networks with as few as 16 layers during training (see Section 5 on Backpropagation) and the problem of increased training error and test error with increased network depth as shown in Figure 43.

Remark 4.6. *Training error, test (generalization) error.* Using a set of data, called training data, to find the parameters that minimize the loss function (i.e., doing the training) provides the training error, which is the least square error between the predicted outputs and the training data. Then running the optimally trained model on a different set of data, which was not been used for the training, called test data, provides the test error, also known as generalization error. More details can be found in Section 6, and in [78], p. 107. ■

The basic building block of residual network is shown in Figure 44, and a full residual network in Figure 45. The rationale for residual networks was that, if the identity map were optimal, it would be easier
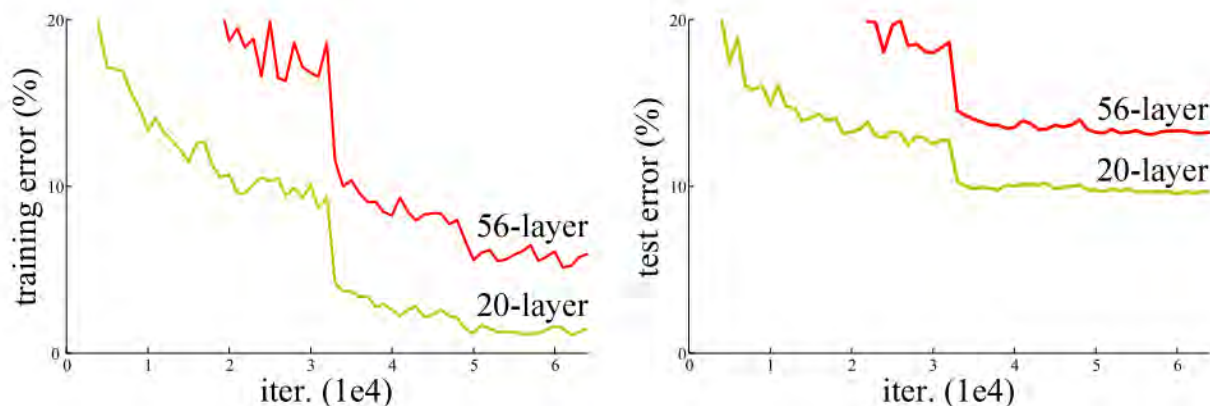
Figure 43: *Training/test error vs. iterations, depth* (Sections 4.6.2, 6). The training error and test error of deep fully-connected networks increased when the number of layers (depth) increased [127]. (Figure reproduced with permission of the authors.)
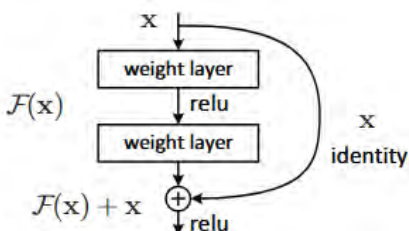
Figure 44: *Residual network* (Sections 4.6.2, 6), basic building block having two layers with the rectified linear activation function (ReLU), for which the input is x, the output is $\mathcal{H}(x) = \mathcal{F}(x) + x$, where the internal mapping function $\mathcal{F}(x) = \mathcal{H}(x) - x$ is called the residual. Chaining this building block one after another forms a deep residual network; see Figure 45 [127].

(Figure reproduced with permission of the authors.)

for the optimization (training) process to drive the residual $\mathcal{F}(x)$ down to zero than to fit the identity map with a bunch of nonlinear layers; see [127], where it was mentioned that deep residual networks won 1st places in several image recognition competitions.

Remark 4.7. The identity map that jumps over a number of layers in the residual network building block in Figure 44 and in the full residual network in Figure 45 is based on a concept close to that for the path of the cell state $c^{[k]}$ in the Long Short-Term Memory (LSTM) unit for recurrent neural networks (RNN), as described in Figure 81 in Section 7.2. ∎

A deep residual network with more than 1,200 layers was proposed in [128]. A wide residual-network architecture that outperformed deep and thin networks was proposed in [129]: "For instance, [their] wide 16-layer deep network has the same accuracy as a 1000-layer thin deep network and a comparable number of parameters, although being several times faster to train."

It is still not clear why some architecture worked well, while others did not:

> "The design of hidden units is an extremely active area of research and does not yet have many definitive guiding theoretical principles." [78], p. 186.

Figure 45: *Full residual network* (Sections [4.6.2], [6]) with 34 layers, made up from 16 building blocks with two layers each (Figure [44]), together with an input and an output layer. This residual network has a total of 3.6 billion floating-point operations (FLOPs with fused multiply-add operations), which could be considered as the network "computational depth" [127]. (Figure reproduced with permission of the authors.)

5 Backpropagation

Backpropagation, sometimes abbreviated as "backprop", was a child of whom many could claim to be the father, and is used to compute the gradient of the cost function with respect to the parameters (weights and biases); see Section [13.4.1] for a history of backpropagation. This gradient is then subsequently used in an optimization process, usually the Stochastic Gradient Descent method, to find the parameters that minimize the cost or loss function.

5.1 Cost (loss, error) function

Two types of cost function are discussed here: (1) the mean squared error (MSE), and (2) the maximum likelihood (probability cost).[86]

5.1.1 Mean squared error

For a given input x (a single example) and target output $y \in \mathbb{R}^m$, the squared error (SE) of the predicted output $\widetilde{y} \in \mathbb{R}^m$ for use in least squared error problem is defined as half the squared error:

$$J(\boldsymbol{\theta}) = \frac{1}{2}\,\mathrm{SE} = \frac{1}{2}\parallel \boldsymbol{y} - \widetilde{\boldsymbol{y}} \parallel^2 = \frac{1}{2}\sum_{k=1}^{k=m}\left(y_k - \widetilde{y}_k\right)^2 \ . \tag{62}$$

The factor $\frac{1}{2}$ is for the convenience of avoiding to carry the factor 2 when taking the gradient of the cost (or loss) function J.[87]

While the components y_k on the output matrix $\boldsymbol{y}$ cannot be *independent and identically distributed* (*i.i.d.*), since $\boldsymbol{y}$ must represent a recognizable pattern (e.g., an image), in the case of training with m examples as inputs:[88]

$$\mathbb{X} = \{\boldsymbol{x}^{|1|}, \cdots, \boldsymbol{x}^{|\mathsf{m}|}\} \text{ and } \mathbb{Y} = \{\boldsymbol{y}^{|1|}, \cdots, \boldsymbol{y}^{|\mathsf{m}|}\} \tag{63}$$

---

[86] For other types of loss function, see, e.g., (1) Section "Loss functions" in "torch.nn — PyTorch Master Documentation" (Original website, Internet archive), and (2) Jah 2019, A Brief Overview of Loss Functions in Pytorch (Original website, Internet archive).

[87] There is an inconsistent use of notation in [78] that could cause confusion, e.g., in [78], Chap. 5, p. 104, Eq. (5.4), the notation $\widehat{y}$ (with the hat) was defined as the network outputs, i.e., predicted values, with y (without the hat) as target values, whereas later in Chap. 6, p. 163, the notation y (without the hat) was used for the network outputs. Also, in [78], p. 105, the cost function was defined as the mean squared error, without the factor $\frac{1}{2}$. See also Footnote [52].

[88] In our notation, m is the dimension of the output array $\boldsymbol{y}$, whereas m (in a different font) is here the number of examples in Eq. ([63]), and later represents the minibatch size in Eqs. ([136])-([138]). The size of the whole training set, called the "full batch" (Footnote [117]), is denoted by M (Footnote [144]).

where $\mathbb{X}$ is the set of m examples, and $\mathbb{Y}$ the set of the corresponding outputs, the examples $\boldsymbol{x}^{|k|}$ can be *i.i.d.*, and the half MSE cost function for these outputs is half the expectation of the SE:

$$J(\boldsymbol{\theta}) = \frac{1}{2}\,\text{MSE} = \frac{1}{2}\,\mathbb{E}(\{SE^{|k|}, k = 1, \cdots, \mathsf{m}\}) = \frac{1}{2\mathsf{m}}\sum_{k=1}^{k=\mathsf{m}} \parallel \boldsymbol{y}^{|k|} - \widetilde{\boldsymbol{y}}^{|k|} \parallel^2 \ . \tag{64}$$

5.1.2 Maximum likelihood (probability cost)

Many (if not most) modern networks employed a probability cost function based in the principle of maximum likelihood, which has the form of negative log-likelihood, describing the cross-entropy between the training data with probability distribution $\hat{p}_{data}$ and the model with probability distribution p_{model} ([78], p. 173):

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{data}} \log p_{model}(\boldsymbol{y} \,|\, \boldsymbol{x}; \boldsymbol{\theta}) \ , \tag{65}$$

where $\mathbb{E}$ is the expectation; $\mathbf{x}$ and $\mathbf{y}$ are random variables for training data with distribution $\hat{p}_{data}$; the inputs $\boldsymbol{x}$ and the target outputs $\boldsymbol{y}$ are values of $\mathbf{x}$ and $\mathbf{y}$, respectively, and p_{model} is the conditional probability of the distribution of the target outputs $\boldsymbol{y}$ given the inputs $\boldsymbol{x}$ and the parameters $\boldsymbol{\theta}$, with the predicted outputs $\widetilde{\boldsymbol{y}}$ given by the model f (neural network), having as arguments the inputs $\boldsymbol{x}$ and the parameters $\boldsymbol{\theta}$:

$$\widetilde{\boldsymbol{y}} = f(\boldsymbol{x}, \boldsymbol{\theta}) \Leftrightarrow \widetilde{y}_k = f_k(\boldsymbol{x}, \boldsymbol{\theta}) \tag{66}$$

The expectations of a function $g(\mathbf{x} = x)$ of a random variable $\mathbf{x}$, having a probability distribution $P(\mathbf{x} = x)$ for the discrete case, and the probability distribution density $p(\mathbf{x} = x)$ for the continuous case, are respectively[89]

$$\mathbb{E}_{\mathbf{x}\sim P}\, g(x) = \sum_x g(x)P(x) = \langle g(x) \rangle \ , \quad \mathbb{E}_{\mathbf{x}\sim p}\, g(x) = \int g(x)p(x)dx = \langle g(x) \rangle \ . \tag{67}$$

Remark 5.1. *Information content, Shannon entropy, maximum likelihood.* The expression in Eq. (65)—with the minus sign and the log function—can be abstract to readers not familiar with the probability concept of maximum likelihood, which is related to the concepts of information content and Shannon entropy. First, an event x with low probability (e.g., an asteroid will hit the Earth tomorrow) would have higher information content than an event with high probability (e.g., the sun will rise tomorrow morning). since the probability of x, i.e., $P(x)$, is between 0 and 1, the negative of the logarithm of $P(x)$, i.e.,

$$I(x) = -\log P(x) \ , \tag{68}$$

called the information content of x, would have large values near zero, and small values near 1. In addition, the probability of two independent events to occur is the product of the probabilities of these events, e.g., the probability of having two heads in two coin tosses is

$$P(\mathbf{x} = \text{head}, \mathbf{y} = \text{head}) = P(\text{head}) \times P(\text{head}) = \frac{1}{2} \times \frac{1}{2} = \frac{1}{4} \tag{69}$$

The product (chain) rule of conditional probabilities consists of expressing a joint probability of several random variables $\{\mathbf{x}^{|1|}, \mathbf{x}^{|2|}, \cdots, \mathbf{x}^{|n|}\}$ as the product[90]

$$P(\mathbf{x}^{|1|}, \cdots, \mathbf{x}^{|n|}) = P(\mathbf{x}^{|1|}) \prod_{k=2}^{k=n} P(\mathbf{x}^{|k|}|\mathbf{x}^{|1|}, \cdots, \mathbf{x}^{|k-1|}) \tag{70}$$

---

[89] The simplified notation $\langle \cdot \rangle$ for expectation $\mathbb{E}(\cdot)$, with implied probability distribution, is used in Section 6.3.4 on step-length decay and simulated annealing (Remark 6.9, Section 6.3.5) as an add-on improvement to the stochastic gradient descent algorithm.

[90] See, e.g., [78], p. 57. The notation $\mathbf{x}^{|k|}$ (with vertical bars enclosing the superscript k) is used to designate example k in the set $\mathbb{X}$ of examples in Eq. (73), instead of the notation $\boldsymbol{x}^{(k)}$ (with parentheses), since the parentheses were already used to surround the layer number k, as in Figure 35.

The logarithm of the products in Eq. (69) and Eq. (70) is the sum of the factor probabilities, and provides another reason to use the logarithm in the expression for information content in Eq. (68): Independent events have additive information. Concretely, the information content of two asteroids independently hitting the Earth should double that of one asteroid hitting the Earth.

The parameters $\widetilde{\boldsymbol{\theta}}$ that minimize the probability cost $J(\boldsymbol{\theta})$ in Eq. (65) can be expressed as[91]

$$\widetilde{\boldsymbol{\theta}} = -\arg\min_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{data}} \log p_{model}(\boldsymbol{y} \,|\, \boldsymbol{x}; \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \frac{1}{\mathsf{m}} \sum_{k=1}^{k=\mathsf{m}} \log p_{model}(\boldsymbol{y}^{|k|} \,|\, \boldsymbol{x}^{|k|}; \boldsymbol{\theta}) \tag{71}$$

$$= \frac{1}{\mathsf{m}} \arg\max_{\boldsymbol{\theta}} \log \prod_{k=1}^{k=\mathsf{m}} P_{model}(\boldsymbol{y}^{|k|} \,|\, \boldsymbol{x}^{|k|}; \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} p_{model}(\mathbb{Y} \,|\, \mathbb{X}; \boldsymbol{\theta}) \tag{72}$$

with $\mathbb{X} = \{\boldsymbol{x}^{|1|}, \cdots, \boldsymbol{x}^{|\mathsf{m}|}\}$ and $\mathbb{Y} = \{\boldsymbol{y}^{|1|}, \cdots, \boldsymbol{y}^{|\mathsf{m}|}\}$, $\tag{73}$

where $\mathbb{X}$ is the set of m examples that are *independent and identically distributed* (*i.i.d.*), and $\mathbb{Y}$ the set of the corresponding outputs. The final form in Eq. (72), i.e.,

$$\widetilde{\boldsymbol{\theta}} = \arg\max_{\boldsymbol{\theta}} p_{model}(\mathbb{Y} \,|\, \mathbb{X}; \boldsymbol{\theta}) \tag{74}$$

is called the *Principle of Maximum Likelihood*, in which the model parameters are optimized to maximize the likelihood to reproduce the empirical data.[92]   ∎

Remark 5.2. *Relation between Mean Squared Error and Maximum Likelihood.* The MSE is a particular case of the Maximum Likelihood. Consider having m examples $\mathbb{X} = \{\boldsymbol{x}^{|1|}, \cdots, \boldsymbol{x}^{|\mathsf{m}|}\}$ that are independent and identically distributed (*i.i.d.*), as in Eq. (63). If the model probability $p_{model}(\boldsymbol{y}^{|k|} \,|\, \boldsymbol{x}^{|k|}; \boldsymbol{\theta})$ has a normal distribution, with the predicted output

$$\widetilde{\boldsymbol{y}}^{|k|} = f(\boldsymbol{x}^{|k|}, \boldsymbol{\theta}) \tag{75}$$

as in Eq. (66), predicting the mean of this normal distribution,[93] then

$$p_{model}(\boldsymbol{y}^{|k|} \,|\, \boldsymbol{x}^{|k|}; \boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{y}^{|k|}; f(\boldsymbol{x}^{|k|}, \boldsymbol{\theta}), \sigma^2) = \mathcal{N}(\boldsymbol{y}^{|k|}; \widetilde{\boldsymbol{y}}^{|k|}, \sigma^2) = \frac{1}{[2\pi\sigma^2]^{\frac{1}{2}}} \exp\left[-\frac{\|\, \boldsymbol{y}^{|k|} - \widetilde{\boldsymbol{y}}^{|k|} \,\|^2}{2\sigma^2} \right], \tag{76}$$

with σ designating the standard deviation, i.e., the error between the target output $\boldsymbol{y}^{|k|}$ and the predicted output $\widetilde{\boldsymbol{y}}^{|k|}$ is normally distributed. By taking the negative of the logarithm of $p_{model}(\boldsymbol{y}^{|k|} \,|\, \boldsymbol{x}^{|k|}; \boldsymbol{\theta})$, we have

$$\log p_{model}(\boldsymbol{y}^{|k|} \,|\, \boldsymbol{x}^{|k|}; \boldsymbol{\theta}) = \frac{1}{2} \log(2\pi\sigma^2) - \frac{\|\, \boldsymbol{y}^{|k|} - \widetilde{\boldsymbol{y}}^{|k|} \,\|^2}{2\sigma^2} . \tag{77}$$

Then summing Eq. (77) over all examples $k = 1, \cdots, \mathsf{m}$ as in the last expression in Eq. (71) yields

$$\sum_{k=1}^{k=\mathsf{m}} \log p_{model}(\boldsymbol{y}^{|k|} \,|\, \boldsymbol{x}^{|k|}; \boldsymbol{\theta}) = \frac{\mathsf{m}}{2} \log(2\pi\sigma^2) - \sum_{k=1}^{k=\mathsf{m}} \frac{\|\, \boldsymbol{y}^{|k|} - \widetilde{\boldsymbol{y}}^{|k|} \,\|^2}{2\sigma^2} , \tag{78}$$

---

[91] A tilde is put on top of $\boldsymbol{\theta}$ to indicate that the matrix $\widetilde{\boldsymbol{\theta}}$ contains the estimated values of the parameters (weights and biases), called the estimates, not the true parameters. Recall from Footnote 87 that [78] used an overhead "hat" (˙) to indicate predicted value; see [78], p. 120, where $\boldsymbol{\theta}$ is defined as the true parameters, and $\hat{\boldsymbol{\theta}}$ the predicted (or estimated) parameters.

[92] See, e.g., [78], p. 128.

[93] The normal (Gaussian) distribution of scalar random variable x, mean μ, and variance σ^2 is written as $\mathcal{N}(x; \mu, \sigma^2) = (2\pi\sigma^2)^{-1/2} \exp[-(x - \mu)^2/(2\sigma^2)]$; see, e.g., [130], p. 24.

and thus the minimizer $\widetilde{\boldsymbol{\theta}}$ in Eq. (71) can be written as

$$\widetilde{\boldsymbol{\theta}} = -\arg\max_{\boldsymbol{\theta}} \sum_{k=1}^{k=\mathsf{m}} \frac{\|\,\boldsymbol{y}^{|k|} - \widetilde{\boldsymbol{y}}^{|k|}\,\|^2}{2\sigma^2} = \arg\min_{\boldsymbol{\theta}} \sum_{k=1}^{k=\mathsf{m}} \frac{\|\,\boldsymbol{y}^{|k|} - \widetilde{\boldsymbol{y}}^{|k|}\,\|^2}{2\sigma^2} = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})\,, \tag{79}$$

where the MSE cost function $J(\boldsymbol{\theta})$ was defined in Eq. (64), noting that constants such as m or $2m$ do not affect the value of the minimizer $\widetilde{\boldsymbol{\theta}}$.

Thus finding the minimizer of the maximum likelihood cost function in Eq. (65) is the same as finding the minimizer of the MSE in Eq. (62); see also [78], p. 130. ∎

Remark 5.2 justifies the use of Mean Squared Error as *a* Maximum Likelihood estimator.[94] For the purpose of this review paper, it is sufficient to use the MSE cost function in Eq. (42) to develop the back-propagation procedure.

5.1.3 Classification loss function

In classification tasks—such as used in [38], Section 10.2, and Footnote 265—a neural network is trained to predict which of k different classes (categories) an input $\boldsymbol{x}$ belongs to. The most simple classification problem only has two classes ($k = 2$), which can be represented by the values $\{0, 1\}$ of a single binary variable y. The probability distribution of such single boolean-valued variable is called *Bernoulli distribution*.[95] The Bernoulli distribution is characterized by a single parameter $p(y = 1\,|\,\boldsymbol{x})$, i.e., the conditional probability of $\boldsymbol{x}$ belonging to the class $y = 1$. To perform binary classification, a neural network is therefore trained to estimate the conditional probability distribution $\widetilde{y} = p(y = 1\,|\,\boldsymbol{x})$ using the principle of maximum likelihood (see Section 5.1.2, Eq. (65)):

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{data}} \log p_{model}(y\,|\,\boldsymbol{x};\boldsymbol{\theta})$$

$$= -\frac{1}{\mathsf{m}} \sum_{k=1}^{k=\mathsf{m}} \left\{ y^{|k|} \log p_{model}(y^{|k|} = 1\,|\,\boldsymbol{x}^{|k|};\boldsymbol{\theta}) + (1 - y^{|k|}) \log(1 - p_{model}(y^{|k|} = 1\,|\,\boldsymbol{x}^{|k|};\boldsymbol{\theta})) \right\}. \tag{80}$$

The output of the neural network is supposed to represent the probability $p_{model}(y = 1\,|\,\boldsymbol{x};\boldsymbol{\theta})$, i.e., a real-valued number in the interval $[0, 1]$. A linear output layer $\widetilde{y} = y^{(L)} = z^{(L)} = \boldsymbol{W}^{(L)}\boldsymbol{y}^{(L-1)} + b^{(L)}$ does not meet this constraint in general. To squash the output of the linear layer into the range of $[0, 1]$, the logistic sigmoid function $\mathfrak{s}$ (see Figure 30) can be added to the linear output unit to render $z^{(L)}$ a probability

$$\widetilde{y} = y^{(L)} = a(z^{(L)}), \qquad a(z^{(L)}) = \mathfrak{s}(z^{(L)}). \tag{81}$$

In case more than two categories occur in a classification problem, a neural network is trained to estimate the probability distribution over the discrete number ($k > 2$) of classes. Such distribution is referred to as *multinoulli* or *categorial* distribution, which is parameterized by the conditional probabilities $p_i = p(y = i\,|\,\boldsymbol{x}) \in [0, 1], i = 1, \ldots, k$ of an input $\boldsymbol{x}$ belonging to the i-th category. The output of the neural network accordingly is a k-dimensional vector $\widetilde{\boldsymbol{y}} \in \mathbb{R}^{k \times 1}$, where $\widetilde{y}_i = p(y = i\,|\,\boldsymbol{x})$. In addition to the requirement of each component $\widetilde{y}_i$ being in the range $[0, 1]$, we must also guarantee that all components sum up to 1 to satisfy the definition of a probability distribution.

---

[94] See, e.g., [78], p. 130.
[95] See, e.g., [130], p. 68.

For this purpose, the idea of *exponentiation and normalization*, which can be expressed as a change of variable in the logistic sigmoid function $\mathfrak{s}$ (Figure 30, Section 5.3.1), as in the following example [78], p. 177:

$$p(y) = \mathfrak{s}[(2y-1)z] = \frac{1}{1 + \exp[-(2y-1)z]} \ , \ \text{with } y \in \{0,1\} \text{ and } z = \text{constant} , \tag{82}$$

$$p(0) + p(1) = \mathfrak{s}(-z) + \mathfrak{s}(z) = \frac{1}{1 + \exp(z)} + \frac{1}{1 + \exp(-z)} = 1 , \tag{83}$$

and is generalized then to vector-valued outputs; see also Figure 46.

The *softmax function* converts the vector formed by a linear unit $\boldsymbol{z}^{(L)} = \boldsymbol{W}^{(L)}\boldsymbol{y}^{(L-1)} + \boldsymbol{b}^{(L)} \in \mathbb{R}^k$ into the vector of probabilities $\widetilde{\boldsymbol{y}}$ by means of

$$\widetilde{y}_i = (\text{softmax } \boldsymbol{z}^{(L)})_i = \frac{\exp z_i^{(L)}}{\sum_{j=1}^{j=k} \exp z_j^{(L)}} \ , \tag{84}$$

and is a smoothed version of the max function [130], p. 198.[96]
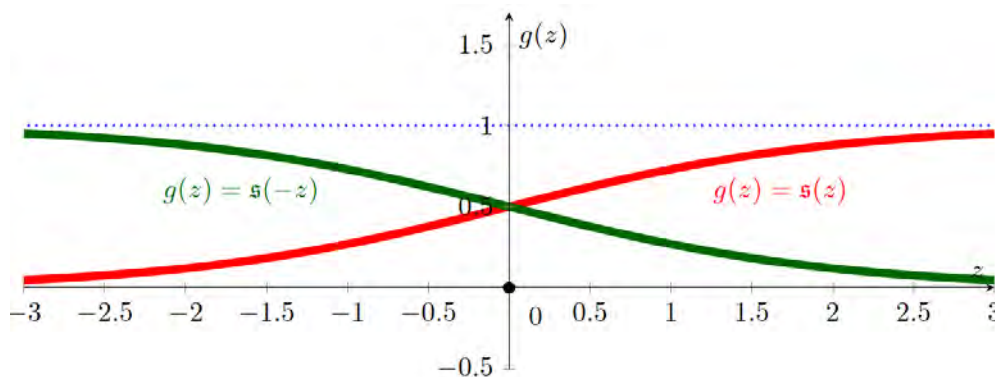
Figure 46: *Softmax function for two classes, logistic sigmoid* (Section 5.1.3, 5.3.1): $\mathfrak{s}(z) = [1 + \exp(-z)]^{-1}$ and $\mathfrak{s}(-z) = [1 + \exp(z)]^{-1}$, such that $\mathfrak{s}(z) + \mathfrak{s}(-z) = 17$. See also Figure 30.

Remark 5.3. *Softmax function from Bayes' theorem.* For a classification with multiple classes $\{\mathcal{C}_k, k = 1, \ldots, K\}$, particularized to the case of two classes with $K = 2$, the probability for class $\mathcal{C}_1$, given the input column matrix $\boldsymbol{x}$, is obtained from Bayes' theorem[97] as follows ([130], p. 197):

$$p(\mathcal{C}_1|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\boldsymbol{x})} = \frac{p(\boldsymbol{x}, \mathcal{C}_1)}{p(\boldsymbol{x}, \mathcal{C}_1) + p(\boldsymbol{x}, \mathcal{C}_2)} = \frac{\exp(z)}{\exp(z) + 1} = \frac{1}{1 + \exp(-z)} = \mathfrak{s}(z) , \tag{85}$$

$$\text{with } z := \ln \frac{p(\boldsymbol{x}, \mathcal{C}_1)}{p(\boldsymbol{x}, \mathcal{C}_2)} \Rightarrow \frac{p(\boldsymbol{x}, \mathcal{C}_1)}{p(\boldsymbol{x}, \mathcal{C}_2)} = \exp(z) , \tag{86}$$

---

[96] See also [78], p. 179 and p. 78, where the softmax function is used to stabilize against the underflow and overflow problem in numerical computation.

[97] Since the probability of x and y is $p(x, y) = p(y, x)$, and since $p(x, y) = p(x|y)p(y)$ (which is the product rule), where $p(x|y)$ is the probability of x given y, we have $p(x|y)p(y) = p(y|x)p(x)$, and thus $p(y|x) = p(x|y)p(y)/p(x)$. The sum rule is $p(x) = \sum_y p(x, y)$. See, e.g., [130], p. 15. The right-hand side of the second equation in Eq. (85)$_2$ makes common sense in terms of the predator-prey problem, in which $p(\boldsymbol{x}, \mathcal{C}_1)$ would be the percentage of predator in the total predator-prey population, and $p(\boldsymbol{x}, \mathcal{C}_2)$ the percentage of prey, as the self-proclaimed "best mathematician of France" Laplace said "probability theory is nothing but common sense reduced to calculation" [130], p. 24.

where the product rule was applied to the numerator of Eq. $(85)_2$, the sum rule to the denominator, and $\mathfrak{s}$ the logistic sigmoid. Likewise,

$$p(\mathcal{C}_2|\boldsymbol{x}) = \frac{p(\boldsymbol{x}, \mathcal{C}_2)}{p(\boldsymbol{x}, \mathcal{C}_1) + p(\boldsymbol{x}, \mathcal{C}_2)} = \frac{1}{\exp(z) + 1} = \mathfrak{s}(-z) \tag{87}$$

$$p(\mathcal{C}_1|\boldsymbol{x}) + p(\mathcal{C}_2|\boldsymbol{x}) = \mathfrak{s}(z) + \mathfrak{s}(-z) = 1 \ , \tag{88}$$

as in Eq. (83), and $\mathfrak{s}$ is also called a normalized exponential or softmax function for $K = 2$. Using the same procedure, for $K > 2$, the softmax function (version 1) can be written as[98]

$$p(\mathcal{C}_i|\boldsymbol{x}) = [1 + \sum_{j=1, j\neq i}^{K} \exp(-z_{ij})]^{-1} \ , \text{ with } z_{ij} := \ln \frac{p(\boldsymbol{x}, \mathcal{C}_i)}{p(\boldsymbol{x}, \mathcal{C}_j)} \ . \tag{89}$$

Using a different definition, the softmax function (version 2) can be written as

$$p(\mathcal{C}_i|\boldsymbol{x}) = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)} \ , \text{ with } z_i := \ln p(\boldsymbol{x}, \mathcal{C}_i) \ , \tag{90}$$

which is the same as Eq. (84). ∎

5.2 Gradient of cost function by backpropagation

The gradient of a cost function $J(\boldsymbol{\theta})$ with respect to the parameters $\boldsymbol{\theta}$ is obtained using the chain rule of differentiation, and backpropagation is an efficient way to compute the chain rule. In the forward propagation, the computation (or function composition) moves from the first layer (1) to the last layer (L); in the backpropagation, the computation moves in reverse order, from the last layer (L) to the first layer (1).

Remark 5.4. We focus our attention on developing backpropagation for fully-connected networks, for which an explicit derivation was not provided in [78], but would help clarify the pseudocode.[99] The approach in [78] was based on computational graph, which would not be familiar to first-time learners from computational mechanics, albeit more general in that it was applicable to networks with more general architecture, such as those with skipped connections, which require keeping track of parent and child processing units for constructing the path of backpropagation. See also Appendix 1 where the backprop Algorithm 1 below is rewritten in a different form to explain the equivalent Algorithm 6.4 in [78], p. 206. ∎

It is convenient to recall here some equations developed earlier (keeping the same equation numbers) for the computation of the gradient $\partial J/\partial \boldsymbol{\theta}^{(\ell)}$ of the cost function $J(\boldsymbol{\theta})$ with respect to the parameters $\boldsymbol{\theta}^{(\ell)}$ in layer (ℓ), going backward from the last layer $\ell = L, \cdots, 1$.

• Cost function $J(\theta)$:

$$J(\boldsymbol{\theta}) = \frac{1}{2} \text{MSE} = \frac{1}{2\mathsf{m}} \parallel \boldsymbol{y} - \widetilde{\boldsymbol{y}} \parallel^2 = \frac{1}{2\mathsf{m}} \sum_{k=1}^{k=\mathsf{m}} \left(y_k - \widetilde{y}_k \right)^2 \ , \tag{62}$$

• Inputs $\boldsymbol{x} = \boldsymbol{y}^{(0)} \in \mathbb{R}^{m_{(0)} \times 1}$ with $m_{(0)} = n$ and predicted outputs $\boldsymbol{y}^{(L)} = \widetilde{\boldsymbol{y}} \in \mathbb{R}^{m_{(L)} \times 1}$ with $m_{(0)} = m$:

$$\boldsymbol{y}^{(0)} = \boldsymbol{x}^{(1)} = \boldsymbol{x} \text{ (inputs)}$$

$$\boldsymbol{y}^{(\ell)} = f^{(\ell)}(\boldsymbol{x}^{(\ell)}) = \boldsymbol{x}^{(\ell+1)} \text{ with } \ell = 1, \ldots, L-1$$

$$\boldsymbol{y}^{(L)} = f^{(L)}(\boldsymbol{x}^{(L)}) = \widetilde{\boldsymbol{y}} \text{ (predicted outputs)} \tag{19}$$

---

[98] See also [130], p. 115, version 1 of softmax function, i.e., $\mu_k = \sum \exp(\eta_k)/[1 + \sum_j \exp \eta_j]$ and $\sum_k \mu_k \leq 1$, had "1" as a summand in the denominator, similar to Eq. (89) while version 2 did not, similar to Eq. (90) [130], p. 198, and was the same as Eq. (84).

[99] See [78], Section 6.5.4, p. 206, Algorithm 6.4.
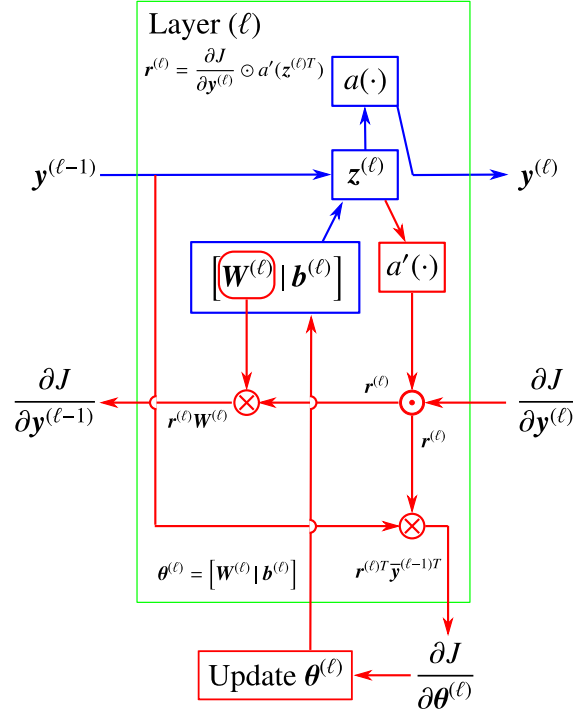
Figure 47: *Backpropagation building block, typical layer* (ℓ) (Section 5.2, Algorithm 1, Appendix 1). The forward propagation path is shown in blue, with the backpropagation path in red. The update of the parameters $\boldsymbol{\theta}^{(\ell)}$ in layer (ℓ) is done as soon as the gradient $\partial J/\partial \boldsymbol{\theta}^{(\ell)}$ is available using a gradient descent algorithm. The row matrix $\boldsymbol{r}^{(\ell)} = \partial J/\partial \boldsymbol{z}^{(\ell)}$ in Eq. (104) can be computed once for use to evaluate both the gradient $\partial J/\partial \boldsymbol{\theta}^{(\ell)}$ in Eq. (105) and the gradient $\partial J/\partial \boldsymbol{y}^{(\ell-1)}$ in Eq. (106), then discarded to free up memory. See pseudocode in Algorithm 1.

.

• Weighted sum of inputs and biases $\boldsymbol{z}^{(\ell)} \in \mathbb{R}^{m_{(\ell)} \times 1}$:

$$\boldsymbol{z}^{(\ell)} = \boldsymbol{W}^{(\ell)} \boldsymbol{y}^{(\ell-1)} + \boldsymbol{b}^{(\ell)} \text{ such that } z_i^{(\ell)} = \boldsymbol{w}_i^{(\ell)} \boldsymbol{y}^{(\ell-1)} + b_i^{(\ell)} \text{ , for } i = 1, \dots, m_{(\ell)} \text{ ,} \tag{26}$$

• Network parameters $\boldsymbol{\theta}$ and layer parameters $\boldsymbol{\theta}^{(\ell)} \in \mathbb{R}^{m_{(\ell)} \times [m_{(\ell-1)}+1]}$:

$$\boldsymbol{\Theta}^{(\ell)} = \left[\theta_{ij}^{(\ell)}\right] = \begin{bmatrix} \boldsymbol{\theta}_1^{(\ell)} \\ \vdots \\ \boldsymbol{\theta}_{m_{(\ell)}}^{(\ell)} \end{bmatrix} = \left[\boldsymbol{W}^{(\ell)} \mid \boldsymbol{b}^{(\ell)} \right] \in \mathbb{R}^{m_{(\ell)} \times [m_{(\ell-1)}+1]} \tag{30}$$

$$\boldsymbol{\theta} = \{\boldsymbol{\theta}^{(1)}, \cdots, \boldsymbol{\theta}^{(\ell)}, \cdots, \boldsymbol{\theta}^{(L)}\} = \{\boldsymbol{\Theta}^{(1)}, \cdots, \boldsymbol{\Theta}^{(\ell)}, \cdots, \boldsymbol{\Theta}^{(L)}\}, \text{ such that } \boldsymbol{\theta}^{(\ell)} \equiv \boldsymbol{\Theta}^{(\ell)} \tag{31}$$

• Expanded layer outputs $\overline{\boldsymbol{y}}^{(\ell-1)} \in \mathbb{R}^{[m_{(\ell-1)}+1] \times 1}$:

$$\boldsymbol{z}^{(\ell)} = \boldsymbol{W}^{(\ell)} \boldsymbol{y}^{(\ell-1)} + \boldsymbol{b}^{(\ell)} \equiv \left[\boldsymbol{W}^{(\ell)} \mid \boldsymbol{b}^{(\ell)} \right] \begin{bmatrix} \boldsymbol{y}^{(\ell-1)} \\ 1 \end{bmatrix} =: \boldsymbol{\theta}^{(\ell)} \overline{\boldsymbol{y}}^{(\ell-1)} \text{ ,} \tag{32}$$

• Activation function $a(\cdot)$:

$$\boldsymbol{y}^{(\ell)} = a(\boldsymbol{z}^{(\ell)}) \text{ such that } y_i^{(\ell)} = a(z_i^{(\ell)}) \tag{35}$$

The gradient of the cost function $J(\boldsymbol{\theta})$ with respect to the parameters $\boldsymbol{\theta}^{(\ell)}$ in layer (ℓ), for $\ell = L, \cdots, 1$,
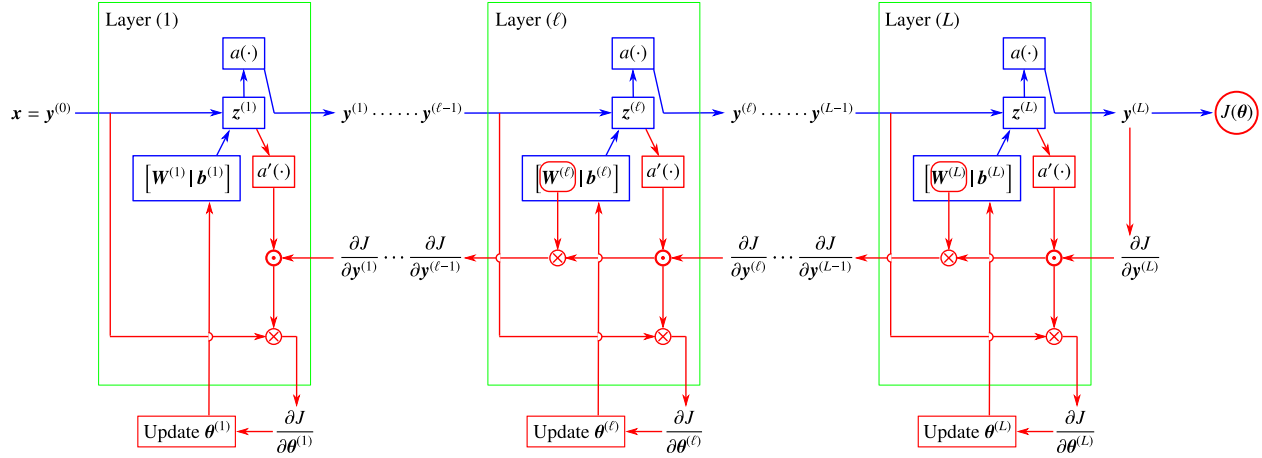
Figure 48: *Backpropagation in fully-connected network* (Section 5.2, 5.3, Algorithm 1, Appendix 1). Starting from the predicted output $\widetilde{y} = y^{(L)}$ In the last layer (L) at the end of any forward propagation (blue arrows), and going backward (red arrows) to the first layer with $\ell = L, \cdots, 1$, and along the way at layer (ℓ), compute the gradient of the cost function J relative the the parameters $\boldsymbol{\theta}^{(\ell)}$ to update those parameters in a gradient descent, then compute the gradient of J relative to the outputs $y^{(\ell-1)}$ of the lower-level layer $(\ell - 1)$ to continue the backpropagation. See pseudocode in Algorithm 1. For a particular example of the above general case, see Figure 51.

is simply:

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^{(\ell)}} = \frac{\partial J(\boldsymbol{\theta})}{\partial y^{(\ell)}} \frac{\partial y^{(\ell)}}{\partial \boldsymbol{\theta}^{(\ell)}} \Leftrightarrow \frac{\partial J}{\partial \theta_{ij}^{(\ell)}} = \sum_{k=1}^{m_{(\ell)}} \frac{\partial J}{\partial y_k^{(\ell)}} \frac{\partial y_k^{(\ell)}}{\partial \theta_{ij}^{(\ell)}} \tag{91}$$

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^{(\ell)}} = \left[\frac{\partial J}{\partial \theta_{ij}^{(\ell)}} \right] \in \mathbb{R}^{m_{(\ell)} \times [m_{(\ell-1)}+1]}, \quad \frac{\partial J(\boldsymbol{\theta})}{\partial y^{(\ell)}} = \left[\frac{\partial J}{\partial y_k^{(\ell)}} \right] \in \mathbb{R}^{1 \times m_{(\ell)}} \text{ (row)} \tag{92}$$

The above equations are valid for the last layer $\ell = L$, since since the predicted output $\widetilde{y}$ is the same as the output of the last layer (L), i.e., $\widetilde{y} \equiv y^{(L)}$ by Eq. (19). Similarly, these equations are also valid for the first layer (1) since the input for layer (1) is $x^{(1)} = x = y^{(0)}$. using Eq. (35), we obtain (no sum on k)

$$\frac{\partial y_k^{(\ell)}}{\partial \theta_{ij}^{(\ell)}} = a'(z_k^{(\ell)}) \frac{\partial z_k^{(\ell)}}{\partial \theta_{ij}^{(\ell)}} = \sum_p a'(z_k^{(\ell)}) \frac{\partial \theta_{kp}^{(\ell)}}{\partial \theta_{ij}^{(\ell)}} \overline{y}_p^{(\ell-1)} = \sum_p a'(z_k^{(\ell)}) \delta_{ki} \delta_{pj} \overline{y}_p^{(\ell-1)} = a'(z_k^{(\ell)}) \delta_{ki} \overline{y}_j^{(\ell-1)} \tag{93}$$

Using Eq. (93) in Eq. (91) leads to the expressions for the gradient, both in component form (left) and in matrix form (right):

$$\frac{\partial J}{\partial \theta_{ij}^{(\ell)}} = \frac{\partial J}{\partial y_i^{(\ell)}} a'(z_i^{(\ell)}) \overline{y}_j^{(\ell-1)} \text{ (no sum on } i) \Leftrightarrow \left[\frac{\partial J}{\partial \theta_{ij}^{(\ell)}} \right] = \left[\left[\frac{\partial J}{\partial y_i^{(\ell)}} \right]^T \odot \left[a'(z_i^{(\ell)}) \right] \right] \left[\overline{y}_j^{(\ell-1)} \right]^T, \tag{94}$$

where $\odot$ is the elementwise multiplication, known as the Hadamard operator, defined as follows:

$$[p_i], [q_i] \in \mathbb{R}^{m \times 1} \Rightarrow [p_i] \odot [q_i] = [(p_i q_i)] = [p_1 q_1, \cdots, p_m q_m]^T \in \mathbb{R}^{m \times 1} \text{ (no sum on } i), \tag{95}$$

1 Backpropagation pseudocode
Data:
- Input into network $\boldsymbol{x} = \boldsymbol{y}^{(0)} \in \mathbb{R}^{m_{(0)} \times 1}$
- Learning rate ϵ; see Section 6.2 on deterministic optimization
- Results from *any* forward propagation:
  - ⋆ Network parameters $\boldsymbol{\theta} = \{\boldsymbol{\theta}^{(1)}, \cdots, \boldsymbol{\theta}^{(L)}\}$ (all layers)
  - ⋆ Layer weighted inputs and biases $\boldsymbol{z}^{(\ell)}$, for $\ell = 1, \cdots, L$
  - ⋆ Layer outputs $\boldsymbol{y}^{(\ell)}$, for $\ell = 1, \cdots, L$

Result: Updated network parameters $\boldsymbol{\theta}$ to reduce cost function J.

2 Initialize:
3 • Gradient $\partial J/\partial \boldsymbol{y}^{(L)}$ relative to predicted output $\widetilde{\boldsymbol{y}} = \boldsymbol{y}^{(L)}$ of last layer L using Eq. (99)
4 • Set layer counter ℓ to last layer L, i.e., $\ell \leftarrow L$;

5 **while** $\ell > 0$, for current layer (ℓ), **do**
6 $\quad$ Compute gradient $\boldsymbol{r}^{(\ell)} = \partial J/\partial \boldsymbol{z}^{(\ell)}$ of cost J relative to weighted sum $\boldsymbol{z}^{(\ell)}$ using Eq. (104) ;
7 $\quad$ Compute gradient $\boldsymbol{g}^{(\ell)} = \partial J/\partial \boldsymbol{\theta}^{(\ell)}$ of cost J relative to layer parameters $\boldsymbol{\theta}^{(\ell)}$ using Eq. (105) ;
8 $\quad$ Update layer parameter $\boldsymbol{\theta}^{(\ell)}$ to decrease cost J using gradient descent Eq. (120), Section 6.2 ;
9 $\quad$ Compute gradient $\partial J/\partial \boldsymbol{y}^{(\ell-1)}$ of cost J relative to outputs $\boldsymbol{y}^{(\ell-1)}$ using Eq. (106) ;
10 $\quad$ Decrement layer counter by one: $\ell \leftarrow \ell - 1$;
11 $\quad$ Propagate to lower-level layer $(\ell - 1)$;
12 **end**

Algorithm 1: *Backpropagation pseudocode* (Section 5.2). Compute the gradient of cost function J relative to the parameters $\boldsymbol{\theta}^{(\ell)}$, with $\ell = L, \cdots, 1$, by backpropagation in one step of a gradient-descent algorithm to find parameters that decrease the cost function. The focus here is backpropagation, not the overall optimization. So the algorithm starts at the end of *any* forward propagation to begin backpropagation, from the last layer (L) back to the first layer (1), to update the parameters to decrease the cost function. See also the block diagrams in Figure 47 and Figure 48, and Appendix 1 where an alternative backprop Algorithm 9 is used to explain the equivalent Algorithm 6.4 in [78], p. 206.

and

$$\left[\frac{\partial J}{\partial y_i^{(\ell)}} \right] \in \mathbb{R}^{1 \times m_{(\ell)}} \text{ (row)} , \ \boldsymbol{z}^{(\ell)} = \left[z_k^{(\ell)} \right] \in \mathbb{R}^{m_{(\ell)} \times 1} \text{ (column)} \Rightarrow a'(z_i^{(\ell)}) \in \mathbb{R}^{m_{(\ell)} \times 1} \text{ (column)} \tag{96}$$

$$\left[\left[\frac{\partial J}{\partial y_i^{(\ell)}} \right]^T \odot \left[a'(z_i^{(\ell)}) \right] \right] \in \mathbb{R}^{m_{(\ell)} \times 1} \text{ (column, no sum on } i) , \text{ and } \left[\overline{y}_j^{(\ell-1)} \right] \in \mathbb{R}^{[m_{(\ell-1)}+1] \times 1} \text{ (column)}$$
$$\tag{97}$$

$$\Rightarrow \left[\frac{\partial J}{\partial \theta_{ij}^{(\ell)}} \right] = \left[\left[\frac{\partial J}{\partial y_i^{(\ell)}} \right]^T \odot \left[a'(z_i^{(\ell)}) \right] \right] \left[\overline{y}_j^{(\ell-1)} \right]^T \in \mathbb{R}^{m_{(\ell)} \times [m_{(\ell-1)}+1]} , \tag{98}$$

which then agrees with the matrix dimension in the first expression for $\partial J/\partial \boldsymbol{\theta}^{(\ell)}$ in Eq. (92). For the last layer $\ell = L$, all terms on the right-hand side of Eq. (98) are available for the computation of the gradient

$\partial J/\partial \theta_{ij}^{(L)}$ since

$$\boldsymbol{y}^{(L)} = \widetilde{\boldsymbol{y}} \Rightarrow \frac{\partial J}{\partial \boldsymbol{y}^{(L)}} = \frac{\partial J}{\partial \widetilde{\boldsymbol{y}}} = \frac{1}{\mathsf{m}} \sum_{k=1}^{k=\mathsf{m}} \left(\widetilde{\boldsymbol{y}}^{|k|} - \boldsymbol{y}^{|k|} \right) \in \mathbb{R}^{1 \times m_{(L)}} \text{ (row)} \tag{99}$$

with m being the number of examples and $m_{(L)}$ the width of layer (L), is the mean error from the expression of the cost function in Eq. (64), with $z_i^{(L)}$ and $\overline{y}_j^{(L-1)}$ already computed in the forward propagation. To compute the gradient of the cost function with respect to the parameters $\boldsymbol{\theta}^{(L-1)}$ in layer $(L-1)$, we need the derivative $\partial J/\partial y_i^{(L-1)}$, per Eq. (98). Thus, in general, the derivative of cost function J with respect to the output matrix $\boldsymbol{y}^{(\ell-1)}$ of layer $(\ell-1)$, i.e., $\partial J/\partial y_i^{(\ell-1)}$, can be expressed in terms of the previously computed derivative $\partial J/\partial y_i^{(\ell)}$ and other quantities for layer (ℓ) as follows:

$$\frac{\partial J}{\partial y_i^{(\ell-1)}} = \sum_k \frac{\partial J}{\partial y_k^{(\ell)}} \frac{\partial y_k^{(\ell)}}{\partial y_i^{(\ell-1)}} = \sum_k \frac{\partial J}{\partial y_k^{(\ell)}} a'(z_k^{(\ell)}) w_{ki}^{(\ell)} = \sum_k \frac{\partial J}{\partial z_k^{(\ell)}} \frac{\partial z_k^{(\ell)}}{\partial y_i^{(\ell-1)}} \tag{100}$$

$$\left[\frac{\partial J}{\partial y_i^{(\ell-1)}} \right] = \left[\left[\frac{\partial J}{\partial y_k^{(\ell)}} \right] \odot \left[a'(z_k^{(\ell)}) \right]^T \right] \left[w_{ki}^{(\ell)} \right] \in \mathbb{R}^{1 \times m_{(\ell-1)}} \text{ (no sum on } k) \tag{101}$$

$$\left[\frac{\partial J}{\partial y_i^{(\ell-1)}} \right] \in \mathbb{R}^{1 \times m_{(\ell-1)}} , \quad \left[\frac{\partial J}{\partial y_k^{(\ell)}} \right] \in \mathbb{R}^{1 \times m_{(\ell)}} \text{ (row) }, \quad \left[a'(z_k^{(\ell)}) \right] \in \mathbb{R}^{m_{(\ell)} \times 1} \text{ (column)} \tag{102}$$

$$\left[\frac{\partial J}{\partial y_k^{(\ell)}} \right] \odot \left[a'(z_k^{(\ell)}) \right]^T = \frac{\partial J}{\partial z_k^{(\ell)}} \in \mathbb{R}^{1 \times m_{(\ell)}} \text{ (no sum on } k) , \quad \left[w_{ki}^{(\ell)} \right] = \frac{\partial z_k^{(\ell)}}{\partial y_i^{(\ell-1)}} \in \mathbb{R}^{m_{(\ell)} \times m_{(\ell-1)}} . \tag{103}$$

Comparing Eq. (101) and Eq. (98), when backpropagation reaches layer (ℓ), the same row matrix

$$\boldsymbol{r}^{(\ell)} := \frac{\partial J}{\partial \boldsymbol{z}^{(\ell)}} = \left[\left[\frac{\partial J}{\partial y_i^{(\ell)}} \right] \odot \left[a'(\boldsymbol{z}_i^{(\ell)}) \right]^T \right] = \frac{\partial J}{\partial \boldsymbol{y}^{(\ell)}} \odot a'(\boldsymbol{z}^{(\ell)T}) \in \mathbb{R}^{1 \times m_{(\ell)}} \text{ (row)} \tag{104}$$

is only needed to be computed once for use to compute both the gradient of the cost J relative to the parameters $\boldsymbol{\theta}^{(\ell)}$ [see Eq. (98) and Figure 47]

$$\frac{\partial J}{\partial \boldsymbol{\theta}^{(\ell)}} = \boldsymbol{r}^{(\ell)T} \overline{\boldsymbol{y}}^{(\ell-1)T} \in \mathbb{R}^{m_{(\ell)} \times [m_{(\ell-1)}+1]} \tag{105}$$

and the gradient of the cost J relative to the outputs $\boldsymbol{y}^{(\ell-1)}$ of layer $(\ell-1)$ [see Eq. (101) and Figure 47]

$$\frac{\partial J}{\partial \boldsymbol{y}^{(\ell-1)}} = \boldsymbol{r}^{\ell} \boldsymbol{W}^{(\ell)} \in \mathbb{R}^{1 \times m_{(\ell-1)}} . \tag{106}$$

The block diagram for backpropagation at layer (ℓ)—as described in Eq. (104), Eq. (105), Eq. (106)—is given in Figure 47, and for a fully-connected network in Figure 48, with pseudocode given in Algorithm 1.

5.3 Vanishing and exploding gradients

To demonstrate the vanishing gradient problem, a network is used in [21], having an input layer containing 784 neurons, corresponding to the $28 \times 28 = 784$ pixels in the input image, four hidden layers, with each hidden layer containing 30 neurons, and an output layer containing 10 neurons, corresponding to the 10 possible classifications for the MNIST digits ('0', '1', '2', ... , '9'). A key ingredient is the use of the sigmoid function as active function; see Figure 30.

We note immediately that the vanishing / exploding gradient problem can be resolved using the rectified linear function (ReLu, Figure 24) as active function in combination with "normalized initialization"[100] and "intermediate normalization layers", which are mentioned in [127], and which we will not discuss here.

The speed of learning of a hidden layer (ℓ) in Figure 49 is defined as the norm of the gradient $\boldsymbol{g}^{(\ell)}$ of the cost function $J(\boldsymbol{\theta})$ with respect to the parameters $\boldsymbol{\theta}^{(\ell)}$ in the hidden layer (ℓ):

$$\parallel \boldsymbol{g}^{(\ell)} \parallel = \left\Vert \frac{\partial J}{\partial \boldsymbol{\theta}^{(\ell)}} \right\Vert \tag{107}$$

The speed of learning in each of the four layers as a function of the number of epochs[101] of training drops down quickly after less than 50 training epochs, then plateaued out, as depicted in Figure 49, where the speed of learning of layer (1) was 100 times less than that of layer (4) after 400 training epochs.
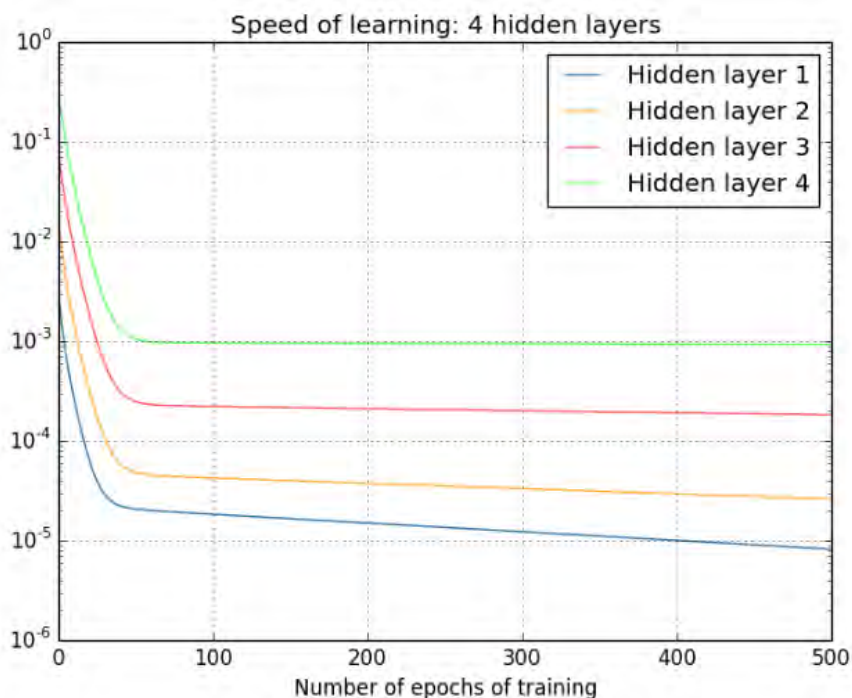
Figure 49: *Vanishing gradient problem* (Section 5.3). Speed of learning of earlier layers is much slower than that of later layers. Here, after 400 epochs of training, the speed of learning of Layer (1) at 10^{-5} (blue line) is 100 times slower than that of Layer (4) at 10^{-3} (green line); [21], Chapter 5, 'Why are deep neural networks hard to train ?' (CC BY-NC 3.0).

To understand the reason for the quick and significant decrease in the speed of learning, consider a network with four layers, having one scalar input x with target scalar output y, and predicted scalar output $\widetilde{y}$, as shown in Figure 50, where each layer has one neuron.[102] The cost function and its derivative are

$$J(\boldsymbol{\theta}) = \frac{1}{2}(y - \widetilde{y})^2 \,, \quad \frac{\partial J}{\partial y} = y - \widetilde{y} \tag{108}$$

---

[100] See [78], p. 295.

[101] An epoch is when all examples in the dataset had been used in a training session of the optimization process. For a formal definition of "epoch", see Section 6.3.1 on stochastic gradient descent (SGD) and Footnote 145.

[102] See also [21].
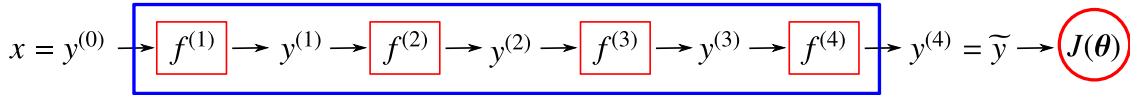
Figure 50: *Neural network with four layers* (Section 5.3), one neuron per layer, scalar input x, scalar output y, cost function $J(\boldsymbol{\theta}) = \frac{1}{2}(y - \widetilde{y})^2$, with $\widetilde{y} = y^{(4)}$ being the target output and also the output of layer (4), such that $f^{(\ell)}(y^{(\ell-1)}) = a(z^{(\ell)})$, with $a(\cdot)$ being the active function, $z^{(\ell)} = w^{(\ell)}y^{(\ell-1)} + b^{(\ell)}$, for $\ell = 1, \ldots, 4$, and the network parameters are $\boldsymbol{\theta} = [w_1, \ldots, w_4, b_1, \ldots, b_4]$. The detailed block diagram is in Figure 51.

The neuron in layer (ℓ) accepts the scalar input $y^{(\ell-1)}$ to produce the scalar output $y^{(\ell)}$ according to

$$f^{(\ell)}(y^{(\ell-1)}) = a(z^{(\ell)}), \text{ with } z^{(\ell)} = w^{(\ell)}y^{(\ell-1)} + b^{(\ell)} . \tag{109}$$

As an example of computing the gradient, the derivative of the cost function J with respect to the bias $b^{(1)}$ of layer (1) is given by

$$\frac{\partial J}{\partial b^{(1)}} = (y - \widetilde{y})[a'(z^{(4)})w^{(4)}][a'(z^{(3)})w^{(3)}][a'(z^{(2)})w^{(2)}][a'(z^{(1)})w^{(1)}] \tag{110}$$

The back propagation procedure to compute the gradient $\partial J/\partial b^{(1)}$ in Eq. (110) is depicted in Figure 51, which is a particular case of the more general Figure 48.
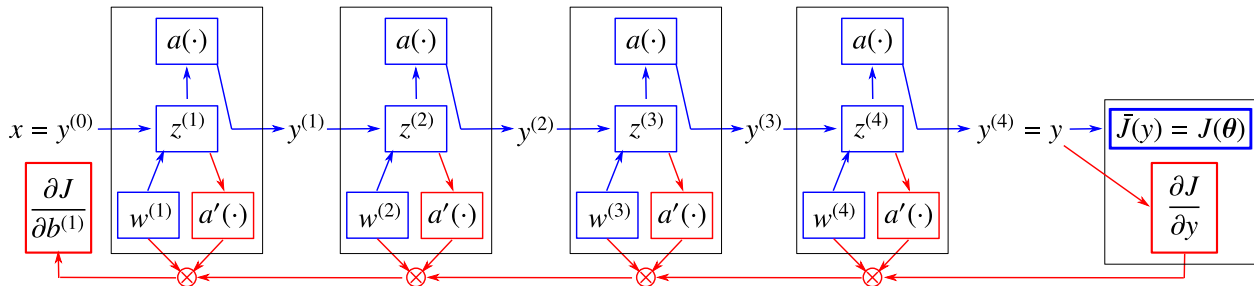
Figure 51: *Neural network with four layers* in Figure 50 (Section 5.3). Detailed block diagram. Forward propagation (blue arrows) and backpropagation (red arrows). In the forward propagation wave, at each layer (ℓ), the product $a'w^{(\ell)}$ is computed and stored, awaiting for the chain-rule derivative to arrive at this layer to multiply. The cost function $J(\theta)$ is computed together with its derivative $\partial J/\partial y$, which is the backprop starting point, from which, when following the backpropagation red arrow, the order of the factors are as in Eq. (110), until the derivative $\partial J/\partial b^{(1)}$ is reached at the head of the backprop red arrow. (Only the weights are shown, not the biases, which are not needed in the back propagation, to save space.) The speed of learning is slowed down significantly in early layers due to vanishing gradient, as shown in Figure 49. See also the more general case in Figure 48.

Whether the gradient $\frac{\partial J}{\partial b^{(1)}}$ in Eq. (110) vanishes or explodes depends on the magnitude of its factors

$$|a'(z^{(\ell)})w^{(\ell)}| < 1, \; \forall \ell \Rightarrow \text{ Vanishing gradient} \tag{111}$$

$$|a'(z^{(\ell)})w^{(\ell)}| > 1, \; \forall \ell \Rightarrow \text{ Exploding gradient} \tag{112}$$

In other mixed cases, the problem of vanishing or exploding gradient could be alleviated by the changing of the magnitude $|a'(z^{(\ell)})w^{(\ell)}|$, above 1 and below 1, from layer to layer.

Remark 5.5. While the vanishing gradient problem for multilayer networks (static case) may be alleviated by weights that vary from layer to layer (the mixed cases mentioned above), this problem is especially

critical in the case of Recurrent Neural Networks, since the weights stay constant for all state numbers (or "time") in a sequence of data. See Remark 7.3 on "short-term memory" in Section 7.2 on Long Short-Term Memory. In back-propagation through the states in a sequence of data, from the last state back to the first state, the same weight keeps being multiplied by itself. Hence, when a weight is less than 1, successive powers of its magnitude eventually decrease to zero when progressing back the first state.   ∎

5.3.1 Logistic sigmoid and hyperbolic tangent

The first derivatives of the sigmoid function and hyperbolic tangent function depicted in Figure 30 (also in Remark 5.3 on the softmax function and Figure 46) are given below:

$$a(z) = \mathfrak{s}(z) = \frac{1}{1 + \exp(-z)} \implies a'(z) = \mathfrak{s}'(z) = \mathfrak{s}(z)[1 - \mathfrak{s}(z)] \in (0, 1) \tag{113}$$

$$a(z) = \tanh(z) \implies a'(z) = \tanh'(z) = \frac{1}{1 + z^2} \in (0, 1] \tag{114}$$

and are less than 1 in magnitude (everywhere for the sigmoid function, and almost everywhere for the hyperbolic tangent tanh function), except at $z = 0$, where the derivative of the tanh function is equal to 1; Figure 52. Successive multiplications of these derivatives will result in smaller and smaller values along the back propagation path. If the weights $w^{(\ell)}$ in Eq. (110) are also smaller than 1, then the gradient $\partial J / \partial b^{(1)}$ will tend toward 0, i.e., vanish. The problem is further exacerbated in deeper networks with increasing number of layers, and thus increasing number of factors less than 1 (i.e., $|a'(z^{(\ell)})w^{(\ell)})| < 1$). We have encountered the vanishing-gradient problem.
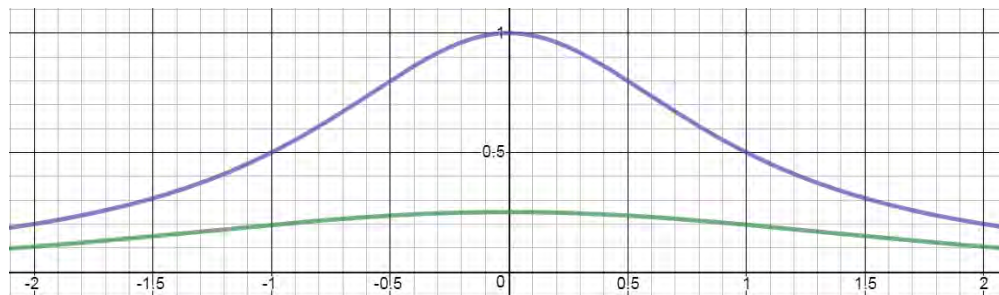
Figure 52: *Sigmoid and hyperbolic tangent functions, derivative* (Section 5.3.1). The derivative of sigmoid function ($\mathfrak{s}'(z) = \mathfrak{s}(z)[1 - \mathfrak{s}(z)]$, green line) is less than 1 everywhere, whereas the derivative of the hyperbolic tangent ($\tanh'(z) = (1 + z^2)^{-1}$, purple line) is less than 1 everywhere, except at the abscissa $z = 0$, where it is equal to 1.

The exploding gradient problem is opposite to the vanishing gradient problem, and occurs when the gradient has its magnitude increases in subsequent multiplications, particularly at a "cliff", which is a sharp drop in the cost function in the parameter space.[103] The gradient at the brink of a cliff (Figure 53) leads to large-magnitude gradients, which when multiplied with each other several times along the back propagation path would result in an exploding gradient problem.

5.3.2 Rectified linear function (ReLU)

The rectified linear function depicted in Figure 24 with its derivative (Heaviside function) equal to 1 for any input greater than zero, would resolve the vanishing-gradient problem, as it is written in [113]:

"For a given input only a subset of neurons are active. Computation is linear on this subset ... Because of this linearity, gradients flow well on the active paths of neurons (there is no gradient

---

[103] See [78], p. 281
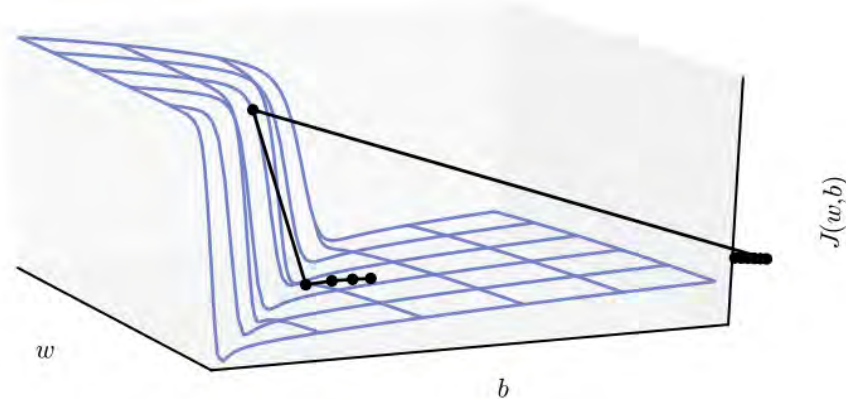
Figure 53: *Cost-function cliff* (Section 5.3.1). A cliff, or a sharp drop in the cost function. The parameter space is represented by a weight w and a bias b. The slope at the brink of the cliff leads to large-magnitude gradients, which when multiplied with each other several times along the back propagation path would result in an exploding gradient problem. [78], p. 281. (Figure reproduced with permission of the authors.)

vanishing effect due to activation non-linearities of sigmoid or tanh units), and mathematical investigation is easier. Computations are also cheaper: there is no need for computing the exponential function in activations, and sparsity can be exploited."

A problem with ReLU was that some neurons were never activated, and called "dying" or "dead", as described in [131]:

"However, ReLU units are at a potential disadvantage during optimization because the gradient is 0 whenever the unit is not active. This could lead to cases where a unit never activates as a gradient-based optimization algorithm will not adjust the weights of a unit that never activates initially. Further, like the vanishing gradients problem, we might expect learning to be slow when training ReL networks with constant 0 gradients."

To remedy this "dying" or "dead" neuron problem, the Leaky ReLU, proposed in [131],[104] had the expression already given previously in Eq. (40), and can be viewed as an approximation to the leaky diode in Figure 29. Both ReLU and Leaky ReLU have been known and used in neuroscience for years before being imported into artificial neural network; see Section 13 for a historical review.

5.3.3 Parametric rectified linear unit (PReLU)

Instead of arbitrarily fixing the slope s of the Leaky ReLU at 0.01 for negative z as in Eq. (40), it is proposed to leave this slope s as a free parameter to optimize along with the weights and biases [61]; see Figure 54:

$$a(z) = \max(s\,z, z) = \begin{cases} s\,z & \text{for } z \leq 0 \\ z & \text{for } 0 < z \end{cases} \tag{115}$$

and thus the network adaptively learned the parameters to control the leaky part of the activation function. Using the Parametric ReLU in Eq.(115), a deep convolutional neural network (CNN) in [61] was able to surpass the level of human performance in image recognition for the first time in 2015; see Figure 3 on ImageNet competition results over the years.

---

[104] According to Google Scholar, [113] (2011) received 3,656 citations on 2019.10.13 and 8,815 citations on 2022.06.23, whereas [131] (2013) received 2,154 and 6,380 citations on these two respective dates.
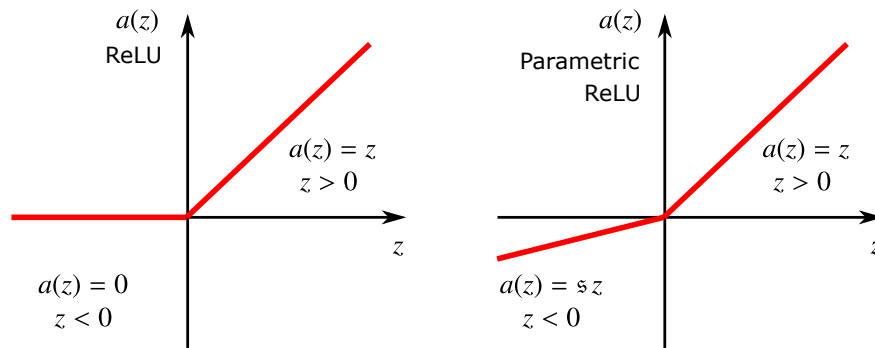
Figure 54: *Rectified Linear Unit* (ReLU, left) and *Parametric ReLU* (right) (Section 5.3.2), in which the slope s is a parameter to optimize; see Section 5.3.3. See also Figure 24 on ReLU.
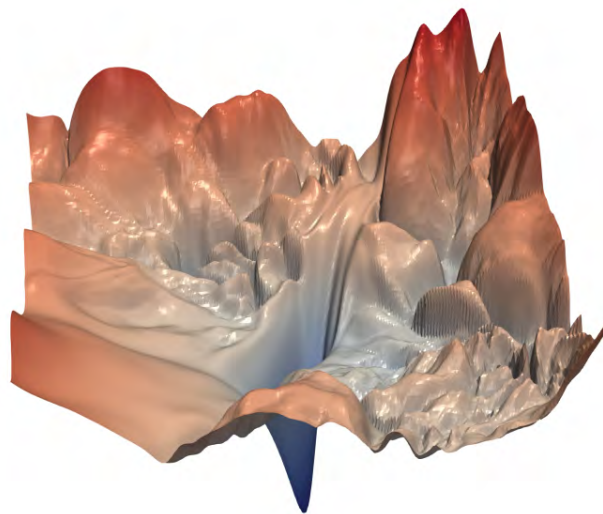
Figure 55: *Cost-function landscape* (Section 6). Residual network with 56 layers (ResNet-56) on the CIFAR-10 training set. Highly non-convex, with many local minima, and deep, narrow valleys [132]. The training error and test error for fully-connected network increased when the number of layers was increased from 20 to 56, Figure 43, motivating the introduction of residual network, Figure 44 and Figure 45, Section 4.6.2. (Figure reproduced with permission of the authors.)

6 Network training, optimization methods

For network training, i.e., to find the optimal network parameters $\boldsymbol{\theta}$ that minimize the cost function $J(\boldsymbol{\theta})$, we describe here both deterministic optimization methods used in full-batch mode,[105] and stochastic optimization methods used in minibatch[106] mode.

Figure 55 shows the highly non-convex landscape of the cost function of a residual network with 56 layers trained using the CIFAR-10 dataset (Canadian Institute For Advanced Research), a collection of images commonly used to train machine learning and computer vision algorithms, containing 60,000 32x32 color images in 10 different classes, representing airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. Each class has 6,000 images.[107]

---

[105] A "full batch" is a complete training set of examples; see Footnote 117.
[106] A minibatch is a random subset of the training set, which is called here the "full batch"; see Footnote 117.
[107] See "CIFAR-10", Wikipedia, version 16:44, 14 October 2019.

Deterministic optimization methods (Section 6.2) include first-order gradient method (Algorithm 2) and second-order quasi-Newton method (Algorithm 3), with line searches based on different rules, introduced by Goldstein, Armijo, and Wolfe.

Stochastic optimization methods (Section 6.3) include

- First-order stochastic gradient descent (SGD) methods (Algorithm 4), with add-on tricks such as momentum and accelerated gradient

- Adaptive learning-rate algorithms (Algorithm 5): Adam and variants such as AMSGrad, AdamW, etc. that are popular in the machine-learning community

- Criticism of adaptive methods and SGD resurgence with add-on tricks such as effective tuning and step-length decay (or annealing)

- Classical line search with stochasticity: SGD with Armijo line search (Algorithm 6), second-order Newton method with Armijo-like line search (Algorigthm 7)
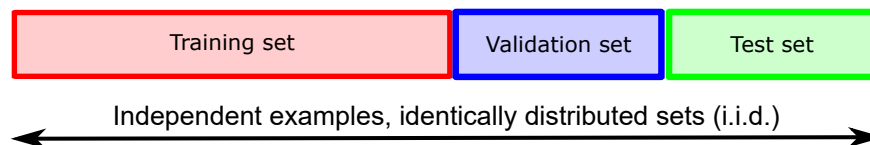
Figure 56: *Training set, validation set, test set* (Section 6.1). Partition of whole dataset. The examples are independent. The three subsets are identically distributed.

6.1 Training set, validation set, test set, stopping criteria

The classical (old) thinking—starting in 1992 with [133] and exemplified by Figures 57, 58, 59, 60 (a, left)—would surprise first-time learners that *minimizing the training error is not optimal* in machine learning. A reason is that training a neural network is *different* from using "pure optimization" since it is desired to decrease not only the error during training (called training error, and that's pure optimization), but also the error committed by a trained network on inputs never seen before.[108] Such error is called generalization error or test error. This classical thinking, known as the *bias-variance trade-off*, has been included in books since 2001 [134] (p. 194) and even repeated in 2016 [78] (p. 268). Models with lower number of parameters have higher bias and lower variance, whereas models with higher number of parameters have lower bias and higher variance; Figure 59.[109]

The modern thinking is exemplified by Figure 60 (b, right) and Figure 61, and does not contradict the intuitive notion that *decreasing the training error to zero is indeed desirable*, as overparameterizing networks beyond the interpolation threshold (zero training error) in modern practice generalizes well (small test error). In Figure 61, the test error continued to decrease significantly with increasing number of parameters N beyond the interpolation threshold $N^\star = 825$, whereas the classical regime ($N < N^\star$) with the bias-variance trade-off (blue) in Figure 59 was restrictive, and did not generalize as well (larger test error).

---

[108] See also [78], p. 268, Section 8.1, "How learning differs from pure optimization"; that's the classical thinking.

[109] See [134], p. 11, for a classification example using two methods: (1) linear models and least squares and (2) k-nearest neighbors. "The linear model makes huge assumptions about structure [high bias] and yields stable [low variance] but possibly inaccurate predictions [high training error]. The method of k-nearest neighbors makes very mild structural assumptions [low bias]: its predictions are often accurate [low training error] but can be unstable [high variance]." See also [130], p. 151, Figure 3.6.

Beyond the interpolation threshold $N^\star$, variance can be decreased by using ensemble average, as shown by the orange line in Figure 61.

Such modern practice was the motivation for research into shallow networks with *infinite* width as a first step to understand how overparameterized networks worked so well; see Figure 148 and Section 14.2 "Lack of understanding on why deep learning worked."
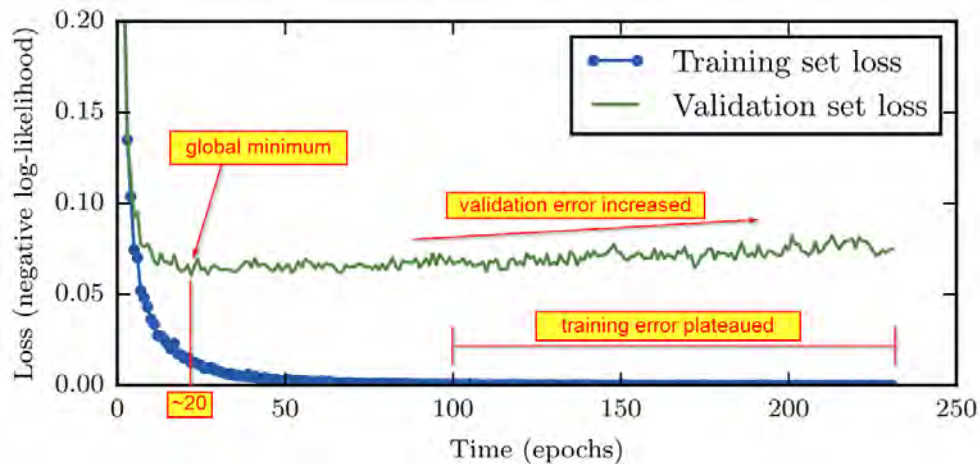
Figure 57: *Training and validation learning curves—Classical viewpoint* (Section 6.1), i.e., plots of training error and validation errors versus epoch number (time). While the training cost decreased continuously, the validation cost reaches a minimum around epoch 20, then started to gradually increase, forming an "asymmetric U-shaped curve." Between epoch 100 and epoch 240, the training error was essentially flat, indicating convergence. Adapted from [78], p. 239. See Figure 60 (a, left), where the classical risk curve is the classical viewpoint, whereas the modern interpolation viewpoint is on the right subfigure (b). (Figure reproduced with permission of the authors.)

To develop a neural-network model, a dataset governed by the same probability distribution, such as the CIFAR-10 dataset mentioned above, can be typically divided into three non-overlapping subsets called *training set, validation set*, and *test set*. The validation set is also called the *development set*, a terminology used in [55], in which an effective method of step-length decay was proposed; see Section 6.3.4.

It was suggested in [135], p. 61, to use 50% of the dataset as training set, 25% as validation set, and 25% as test set. On the other hand, while a validation set with size about $1/4$ of the training set was suggested in [78], p. 118, there was no suggestion for the relative size of the test set.[110] See Figure 56 for a conceptual partition of the dataset.

Examples in the training set are fed into an optimizer to find the network parameter estimate $\widetilde{\theta}$ that minimizes the cost function estimate $\widetilde{J}(\widetilde{\theta})$.[111] As the optimization on the training set progresses from epoch

---

[110] Andrew Ng suggested the following partitions. For small datasets having less than 10^4 examples, the training/validation/test ratio of $60\%/20\%/20\%$ could be used. For large datasets with order of 10^6 examples, use ratio $98\%/1\%/1\%$. For datasets with much more than 10^6 examples, use ratio $99.5\%/0.25\%/0.25\%$. See Coursera course "Improving deep neural network: Hyperparameter tuning, regularization and optimization", at time 4:00, video website.

[111] The word "estimate" is used here for the more general case of stochastic optimization with minibatches; see Section 6.3.1 on stochastic gradient descent and subsequent sections on stochastic algorithms. When deterministic optimization is used with the full batch of dataset, then the cost estimate is the same as the cost, i.e., $\widetilde{J} \equiv J$, and the network parameter estimates are the same as the network parameters, i.e., $\widetilde{\theta} \equiv \theta$.
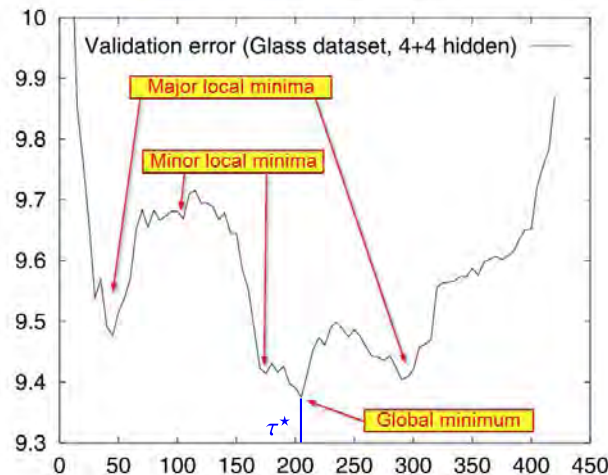
Figure 58: *Validation learning curve* (Section 6.1, Algorithm 4). Validation error vs epoch number. Some validation error could oscillate wildly around the mean, resulting in an "ugly reality". The global minimum validation error corresponded to epoch number $\tau^\star$. Since the stopping criteria may miss this global minimum, it was suggested to monitor the validation learning curve to find the epoch $\tau^\star$ at which the network parameters $\widetilde{\boldsymbol{\theta}}_{\tau^\star}$ would be declared optimal. Adapted from [135], p. 55. (Figure reproduced with permission of the authors.)

to epoch,[112] examples in the validation set are fed as inputs into the network to obtain the outputs for computing the cost function $\widetilde{J}_{val}(\widetilde{\boldsymbol{\theta}}_\tau)$, also called validation error, at predetermined epochs $\{\tau_k\}$ using the network parameters $\{\widetilde{\boldsymbol{\theta}}_{\tau_k}\}$ obtained from the optimization on the training set at those epochs.

Figure 57 shows the different behaviour of the training error versus that of the validation error. The validation error would decrease quickly initially, reaching a global minimum, then gradually increased, whereas the training error continued to decrease and plateaued out, indicating that the gradients got smaller and smaller, and there was not much decrease in the cost. From epoch 100 to epoch 240, the traning error was at about the same level, with litte noise. The validation error, on the other hand, had a lot of noise.

Because of the "asymmetric U-shaped curve" of the validation error, the thinking was that if the optimization process could stop early at the global mininum of the validation error, then the generalization (test) error, i.e., the value of cost function on the test set, would also be small, thus the name "*early stopping*". The test set contains examples that have not been used to train the network, thus simulating inputs never seen before. The validation error could have oscillations with large amplitude around a mean curve, with many local minima; see Figure 58.

The difference between the test (generalization) error and the validation error is called the generalization gap, as shown in the *bias-variance trade-off* [133] Figure 59, which qualitatively delineates these errors versus model capacity, and conceptually explains the optimal model capacity as where the generalization gap equals the training error, or the generalization error is twice the training error.

Remark 6.1. Even the best machine learning generalization capability nowadays still cannot compete with the generalization ability of human babies; see Section 14.6 on "What's new? Teaching machines to think like babies". ∎

Early-stopping criteria. One criterion is to first define the lowest validation error from epoch 1 up to

---

[112] An epoch is when all examples in the dataset had been used in a training session of the optimization process. For a formal definition of "epoch", see Section 6.3.1 on stochastic gradient descent (SGD) and Footnote 145.
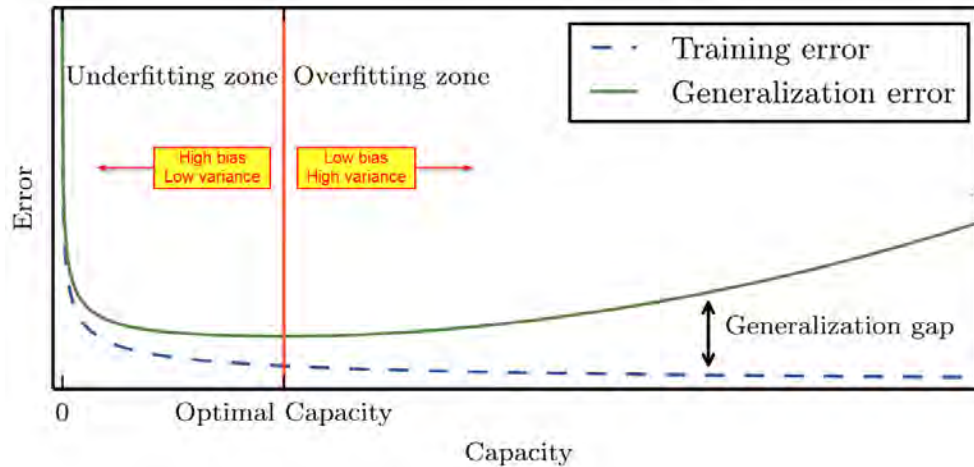
Figure 59: *Bias-variance trade-off* (Section 6.1). Training error (cost) and test error versus model capacity. Two ways to change the model capacity: (1) change the number of network parameters, (2) change the values of these parameters (weight decay). The generalization gap is the difference between the test (generalization) error and the training error. As the model capacity increases from underfit to overfit, the training error decreases, but the generalization gap increases, past the optimal capacity. Figure 72 gives examples of underfit, appropriately fit, overfit. See [78], p. 112. The above is the classical viewpoint, which is still prevalent [136]; see Figure 60 for the modern viewpoint, in which overfitting with high capacity model generalizes well (small test error) in practice. (Figure reproduced with permission of the authors.)

the current epoch τ as:

$$\widetilde{J}^{\star}_{val}(\widetilde{\boldsymbol{\theta}}_{\tau}) = \min_{\tau' \leq \tau}\{\widetilde{J}_{val}(\widetilde{\boldsymbol{\theta}}_{\tau'})\} \ , \tag{116}$$

then define the *generalization loss* (in percentage) at epoch τ as the increase in validation error relative to the minimum validation error from epoch 1 to the present epoch τ:

$$G(\tau) = 100 \cdot \left(\frac{\widetilde{J}_{val}(\widetilde{\boldsymbol{\theta}}_{\tau})}{\widetilde{J}^{\star}_{val}(\widetilde{\boldsymbol{\theta}}_{\tau})} - 1 \right) \ . \tag{117}$$

[135] then defined the "first class of stopping criteria" as follows: Stop the optimization on the training set when the generalization loss exceeds a certain threshold s (generalization loss lower bound):

$$G_{\mathsf{s}} : \text{Stop after epoch } \tau \text{ if } G(\tau) > \mathsf{s} \ . \tag{118}$$

The issue is how to determine the generalization loss lower bound s so not to fall into a local minimum, and to catch the global minimum; see Figure 58. There were many more early-stopping criterion classes in [135]. But it is not clear whether all these increasingly sophisticated stopping criteria would work to catch the validation-error global minimum in Figure 58.

Moreover, the above discussion is for the *classical regime* in Figure 60 (a). In the context of the *modern interpolation regime* in Figure 60 (b), early stopping means that the computation would cease as soon as the training error reaches "its lowest possible value (typically zero [beyond the interpolation threshold], unless two identical data points have two different labels)" [137]. See the green line in Figure 61.

Computational budget, learning curves. A simple method would be to set an epoch budget, i.e., the largest number of epochs for computation sufficiently large for the training error to go down significantly, then monitor graphically both the training error (cost) and the validation error versus epoch number. These plots are called the *learning curves*; see Figure 57, for which an epoch budget of 240 was used. Select the

(a)   (b)
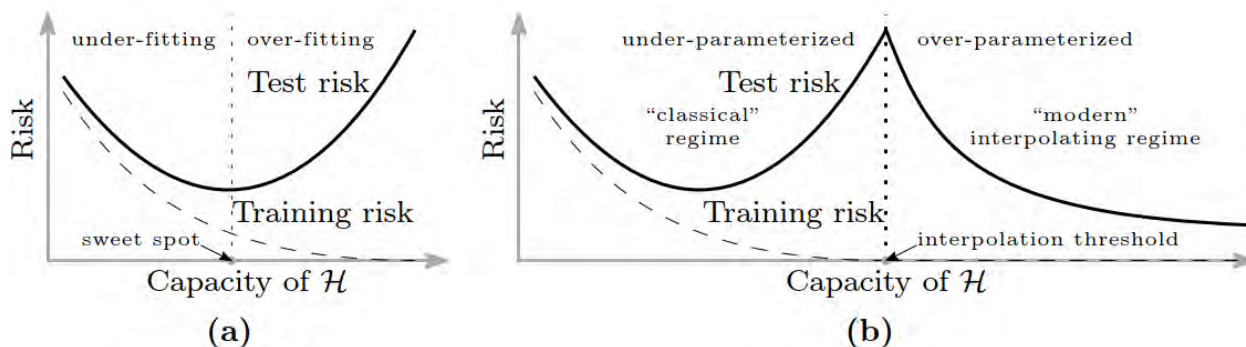
Figure 60: *Modern interpolation regime* (Sections 6.1, 14.2). Beyond the interpolation threshold, the test error goes down as the model capacity (e.g., number of parameters) increases, describing the observation that networks with high capacity beyond the interpolation threshold generalize well, even though overfit in training. Risk = error or cost. Capacity = number of parameters (but could also be increased by weight decay). Figures 57, 59 corresponds to the classical regime, i.e., old method (thinking) [136]. See Figure 61 for experimental evidence of the modern interpolation regime, and Figure 148 for a shallow network with infinite width. Permission of NAS.

global minimum of the validation learning curve, with epoch number $\tau^\star$ (Figure 57), and use the corresponding network parameters $\widetilde{\boldsymbol{\theta}}_{\tau^\star}$, which were saved periodically, as optimal paramters for the network.[113]

Remark 6.2. Since it is important to monitor the validation error during training, a whole section is devoted in [78] (Section 8.1, p. 268) to expound on "How Learning Differs from Pure Optimization". And also for this reason, it is not clear yet what *global optimization* algorithms such as in [138] could bring to network training, whereas the stochastic gradient descent (SGD) in Section 6.3.1 is quite efficient; see also Section 6.5.9 on criticism of adaptive methods. ∎

Remark 6.3. *Epoch budget, global iteration budget.* For stochastic optimization algorithms—Sections 6.3, 6.5, 6.6, 6.7—the epoch counter is τ and the epoch budget τ_{max}. Numerical experiments in Figure 73 had an epoch budget of $\tau_{max} = 250$, whereas numerical experiments in Figure 74 had an epoch budget of $\tau_{max} = 1800$. The computational budget could be specified in terms of global iteration counter j as j_{max}. Figure 71 had a global iteration budget of $j_{max} = 5000$. ∎

Before presenting the stochastic gradient-descent (SGD) methods in Section 6.3, it is important to note that classical deterministic methods of optimization in Section 6.2 continue to be useful in the age of deep learning and SGD.

> "One should not lose sight of the fact that [full] batch approaches possess some intrinsic advantages. First, the use full gradient information at each iterate opens the door for many deterministic gradient-based optimization methods that have been developed over the past decades, including not only the full gradient method, but also accelerated gradient, conjugate gradient, quasi-Newton, inexact Newton methods, and can benefit from parallelization." [80], p. 237.

6.2 Deterministic optimization, full batch

Once the gradient $\partial J / \partial \boldsymbol{\theta}^{(\ell)} \in \mathbb{R}^{m_{(\ell)} \times [m_{(\ell-1)}+1]}$ of the cost function J has been computed using back-propagation described in Section 5.2, the layer parameters $\boldsymbol{\theta}^{(\ell)}$ are updated to decrease cost function J using

---

[113] See also "Method for early stopping in a neural network", StackExchange, 2018.03.05, Original website, Internet archive. [78], p. 287, also suggested to monitor the training learning curve to adjust the step length (learning rate).
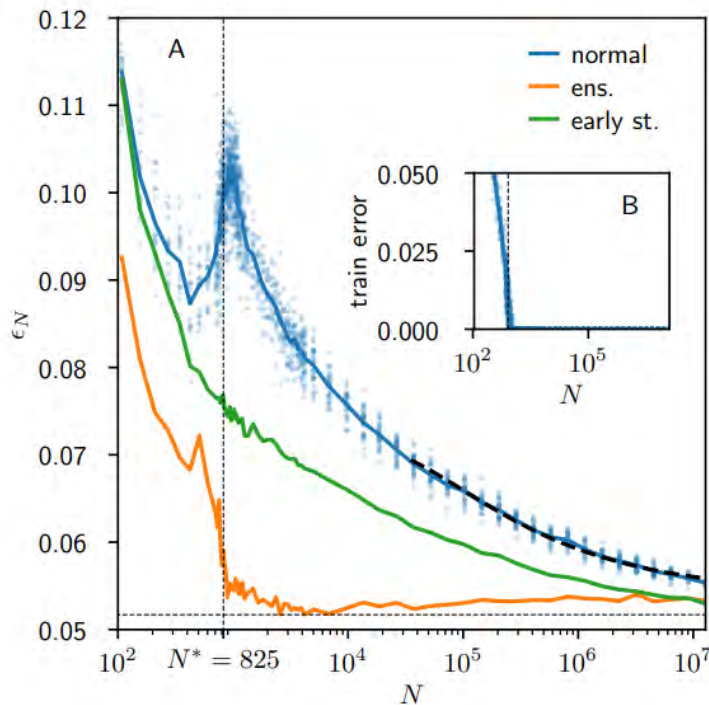
Figure 61: *Empirical test error vs Number of paramesters* (Sections 6.1, 14.2). Experiments using the MNIST handwritten digit database in [137] confirmed the modern interpolation regime in Figure 60 [136]. *Blue:* Average over 20 runs. *Green:* Early stopping. *Orange:* Ensemble average on $n = 20$ samples, trained independently. See Figure 148 for a shallow network with infinite width. (Figure reproduced with permission of the authors.)

gradient descent as follows:

$$\boldsymbol{\theta}^{(\ell)} \leftarrow \boldsymbol{\theta}^{(\ell)} - \epsilon\, \partial J/\partial \boldsymbol{\theta}^{(\ell)} = \boldsymbol{\theta}^{(\ell)} - \epsilon\, \boldsymbol{g}^{(\ell)} \, , \text{ with } \boldsymbol{g}^{(\ell)} := \partial J/\partial \boldsymbol{\theta}^{(\ell)} \in \mathbb{R}^{m_{(\ell)} \times [m_{(\ell-1)}+1]} \, . \tag{119}$$

being the gradient direction, and ϵ called the learning rate.[114] The layer-by-layer update in Eq. (119) as soon as the gradient $\boldsymbol{g}^{(\ell)} = \partial J/\partial \boldsymbol{\theta}^{(\ell)}$ had been computed is valid when the learning rate ϵ does not depend on the gradient $\boldsymbol{g} = \partial J/\partial \boldsymbol{\theta}$ with respect to the whole set of network parameters $\boldsymbol{\theta}$.

Otherwise, the update of the whole network parameter $\boldsymbol{\theta}$ would be carried out after the complete gradient $\boldsymbol{g} = \partial J/\partial \boldsymbol{\theta}$ had been obtained, and the learning rate ϵ had been computed based on the gradient $\boldsymbol{g}$:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon\, \partial J/\partial \boldsymbol{\theta} = \boldsymbol{\theta} - \epsilon\, \boldsymbol{g} \, , \text{ with } \boldsymbol{g} := \partial J/\partial \boldsymbol{\theta} \in \mathbb{R}^{P_T \times 1} \, , \tag{120}$$

where P_T is the total number of network paramters defined in Eq. (34). An example of a learning-rate computation that depends on the complete gradient $\boldsymbol{g} = \partial J/\partial \boldsymbol{\theta}$ is gradient descent with Armijo line search; see Section 6.2.3 and line 8 in Algorithm 1.

> "Neural network researchers have long realized that the learning rate is reliably one of the most difficult to set hyperparameters because it significantly affects model performance." [78], p. 298.

---

[114] See Figure 151 in Section 14.7 on "Lack of transparency and irreproducibility of results" in recent deep-learning papers.

In fact, it is well known in the field of optimization, where the learning rate is often mnemonically denoted by λ, being Greek for "l" and standing for "step length"; see, e.g., Polak (1971) [139].

> "We can choose ϵ in several different ways. A popular approach is to set ϵ to a small constant. Sometimes, we can solve for the step size that makes the directional derivative vanish. Another approach is to evaluate $f(\boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}))$ for several values of ϵ and choose the one that results in the smallest objective function value. This last strategy is called a **line search**." [78], p. 82.

Choosing an arbitrarily small ϵ, without guidance on how small is small, is not a good approach, since exceedingly slow convergence could result for too small ϵ. In general, it would not be possible to solve for the step size to make the directional derivative vanish. There are several variants of line search for computing the step size to decrease the cost function, based on the "decrease" conditions,[115] among which some are mentioned below.[116]

Remark 6.4. *Line search in deep-learning training.* Line search methods are not only important for use in deterministic optimization with full batch of examples,[117] but also in stochastic optimization (see Section 6.3) with random mini-batches of examples [80]. The difficulty of using stochastic gradient coming from random mini-batches is the presence of noise or "discontinuities"[118] in the cost function and in the gradient. Recent stochastic optimization methods—such as the sub-sampled Hessian-free Newton method reviewed in [80], the probabilisitic line search in [143], the first-order stochastic Armijo line search in [144], the second-order sub-sampling line search method in [145], quasi-Newton method with probabilitic line search in [146], etc.—where line search forms a key subprocedure, are designed to address or circumvent the noisy gradient problem. For this reason, claims that line search methods have "fallen out of favor"[119] would be misleading, as they may encourage students not to learn the classics. A classic never dies; it just re-emerges in a different form with additional developments to tackle new problems. ∎

In view of Remark 6.4, a goal of this section is to develop a feel for some classical deterministic line search methods for readers not familiar with these concepts to prepare for reading extensions of these methods to stochastic line search methods.

6.2.1 Exact line search

Find a positive step length ϵ that minimizes the cost function J along the descent direction $\boldsymbol{d}$ such that the scalar (dot) product between $\boldsymbol{g}$ and $\boldsymbol{d} \in \mathbb{R}^{P_T \times 1}$:

$$< \boldsymbol{g}, \boldsymbol{d} > = \boldsymbol{g} \bullet \boldsymbol{d} = \sum_i \sum_j g_{ij} d_{ij} < 0 , \tag{121}$$

---

[115] See, e.g., [140], [141].

[116] See also [139], p. 243.

[117] A full batch contains all examples in the training set. There is a confusion in the use of the word "batch" in terminologies such as "batch optimization" or "batch gradient descent", which are used to mean the full training set, and not a subset of the training set; see, e.g., [78], p. 271. Hence we explicitly use "full batch" for full training set, and mini-batch for a small subset of the training set.

[118] See, e.g., [80]. Noise is sometimes referred to as "discontinuities" as in [142]. See also the lecture video "Understanding mini-batch gradient descent," at time 1:20, by Andrew Ng on Coursera website.

[119] In [78], a discussion on line search methods, however brief, was completely bypassed to focus on stochastic gradient-descent methods with learning-rate tuning and scheduling, such as AdaGrad, Adam, etc. Ironically, it is disconcerting to see these authors, who made important contributions to deep learning, thus helping thawing the last "AI winter", regard with skepticism "most guidance" on learning-rate selection; see [78], p. 287, and Section 6.3. Since then, fully automatic stochastic line-search methods, without tuning runs, have been developed, apparently starting with [147]. In the abstract of [142], where an interesting method using only gradients, without function evaluations, was presented, one reads "Due to discontinuities induced by mini-batch sampling, [line searches] have largely fallen out of favor".

is negative, i.e., the descent direction $\boldsymbol{d}$ and the gradient $\boldsymbol{g}$ form an obtuse angle bounded away from $90°$,[120] and

$$J(\boldsymbol{\theta} + \epsilon\,\boldsymbol{d}) = \min_{\lambda}\{J(\boldsymbol{\theta} + \lambda\,\boldsymbol{d})\,|\,\lambda \geq 0\}\,. \tag{122}$$

The minimization problem in Eq. (122) can be implemented using the Golden section search (or infinite Fibonacci search) for unimodal functions.[121] For more general non-convex cost functions, a minimizing step length may be non-existent, or difficult to compute exactly.[122] In addition, a line search for a minimizing step length is only an auxilliary step in an overall optimization algorithm. It is therefore sufficient to find an approximate step length satisfying some decrease conditions to ensure convergence to a local minimum, while keeping the step length from being too small that would hinder a reasonable advance toward such local minimum. For these reasons, inexact line search methods (rules) were introduced, first in [150], followed by [151], then [152] and [153]. In view of Remark 6.4 and Footnote 119, as we present these deterministic line-search rules, we will also immediately recall, where applicable, the recent references that generalize these rules by adding stochasticity for use as a subprocedure (inner loop) for the stochastic gradient-descent (SGD) algorithm.

6.2.2 Inexact line-search, Goldstein's rule

The method is inexact since the search for an acceptable step length would stop before a minimum is reached, once the rule is satisfied.[123] For a fixed constant $\alpha \in (0, \frac{1}{2})$, select a learning rate (step length) ϵ such that[124]

$$1 - \alpha \leq \frac{J(\boldsymbol{\theta} + \epsilon\,\boldsymbol{d}) - J(\boldsymbol{\theta})}{\epsilon\,\boldsymbol{g}\bullet\boldsymbol{d}} \leq \alpha\,, \tag{123}$$

where both the numerator and the denominator are negative, i.e., $J(\boldsymbol{\theta} + \epsilon\,\boldsymbol{d}) - J(\boldsymbol{\theta}) < 0$ and $\boldsymbol{g}\bullet\boldsymbol{d} < 0$ by Eq. (121). Eq. (123) can be recast into a slightly more general form: For $0 < \alpha < \beta < 1$, choose a learning rate ϵ such that[125]

$$\beta\,\epsilon\,\boldsymbol{g}\bullet\boldsymbol{d} \leq \Delta J(\epsilon) \leq \alpha\,\epsilon\,\boldsymbol{g}\bullet\boldsymbol{d}\,,\text{ with }\Delta J(\epsilon) := J(\boldsymbol{\theta} + \epsilon\,\boldsymbol{d}) - J(\boldsymbol{\theta})\,. \tag{124}$$

---

[120] Or equivalently, the descent direction $\boldsymbol{d}$ forms an acute angle with the gradient (or steepest) descent direction $[-\boldsymbol{g} = -\partial J/\partial\boldsymbol{\theta}]$, i.e., the negative of the gradient direction.

[121] See, e.g., [139], p. 31, for the implementable algorithm, with the assumption that the cost function was convex. Convexity is, however, not needed for this algorithm to work; only unimodality is needed. A unimodal function has a unique minimum, and is decreasing on the left of the minimum, and increasing on the right of the minimum. Convex functions are necessarily unimodal, but not vice versa. Convexity is a particular case of unimodality. See also [148], p. 216, on Golden section search as infinite Fibonacci search and curve fitting line-search methods.

[122] See [149], p. 29, for this assertion, without examples. An example of non-existent minimizing step length $\epsilon = \arg\min_{\lambda}\{f(\theta + \lambda d)|\lambda \geq 0\}$ could be f being a concave function. If we relax the continuity requirement, then it is easy to construct a function with no mininum and no maximum, e.g., $f(x) = |x|$ on open intervals $(-1, 0) \cup (0, +1)$, and $f(-1) = f(0) = f(1) = 0.5$; this function is "essentially" convex, except on the set $\{-1, 0, +1\}$ of measure zero where it is discontinuous. An example of a function whose minimum is difficult to compute exactly could be one with a minimum at the bottom of an extremely narrow crack.

[123] The book [154] was cited in [139], p. 33, but not the papers [150] and [155], where Goldstein's rule was explicitly presented in the form: Step length $\gamma \to g(x^k, \gamma) = [f(x^k) - f(x^k - \gamma\varphi(x^k))]/\{\gamma[\nabla f(x^k), \varphi(x^k)]\}$, with φ being the descent direction, $[a, b]$ the scalar (dot) product between vector a and vector b, and $\delta \leq g(x^k, \gamma) \leq 1 - \delta$, if $g(x^k, 1) \leq \delta$. The same relation was given in [150], with different notation.

[124] See also [139], p. 33.

[125] See [149], p. 55, and [156], p. 256, where the equality $\alpha = \beta$ was even allowed, provided the step length ϵ satisfied the equality $f(\theta + \epsilon\,d) - f(\theta) = \alpha\,\epsilon\,\boldsymbol{g}\bullet\boldsymbol{d}$. But that would make the computation unnecessarily stringent and costly, since the step length ϵ as the root of this equation has to be solved for accurately. Again, the idea should be to make the sector bounded by the two lines, $\beta\,\epsilon\,\boldsymbol{g}\bullet\boldsymbol{d}$ from below and the line $\alpha\,\epsilon\,\boldsymbol{g}\bullet\boldsymbol{d}$ from above, as large as possible in inexact line search. See discussion below Eq. (124).

A reason could be that the sector bounded by the two lines $(1 - \alpha)\epsilon\, \boldsymbol{g} \bullet \boldsymbol{d}$ and $\alpha\epsilon\, \boldsymbol{g} \bullet \boldsymbol{d}$ may be too narrow when α is close to 0.5 from below, making $(1 - \alpha)$ also close to 0.5 from above. For example, it was recommended in [139], p. 33 and p. 37, to use $\alpha = 0.4$, and hence $1 - \alpha = 0.6$, making a tight sector, but we could enlarge such sector by choosing $0.6 < \beta < 1$.
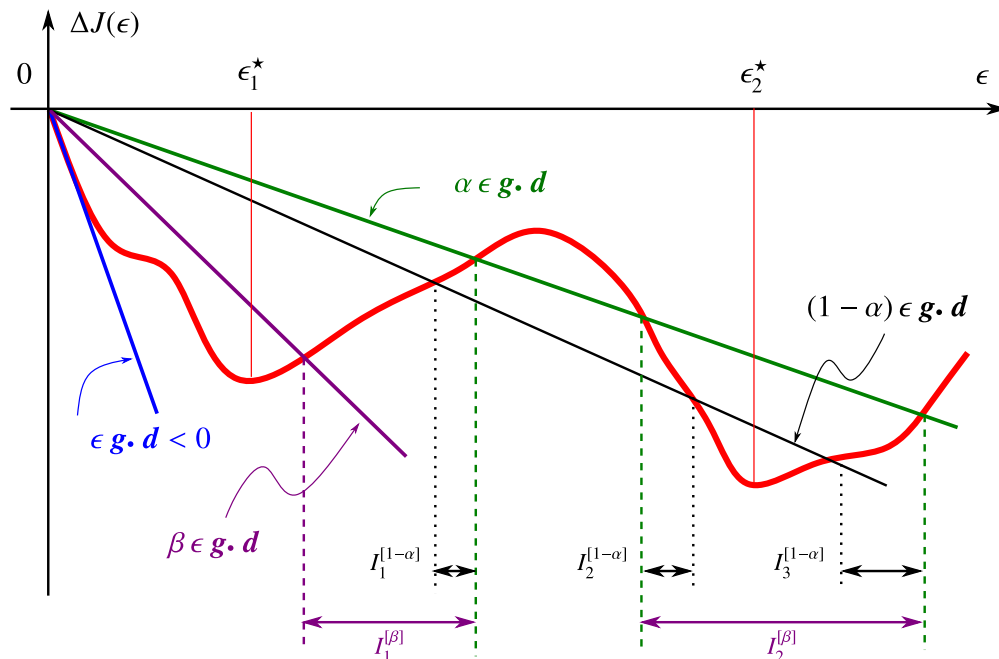
Figure 62: *Inexact line search, Goldstein's rule* (Section 6.2.4). acceptable step lengths would be such that a decrease in the cost function J, denoted by ΔJ in Eq. (124), falls into an acceptable sector formed by an upper-bound line and a lower-bound line. the upper bound is given by the straight line $\alpha\epsilon\, \boldsymbol{g} \bullet \boldsymbol{d}$ (green), with fixed constant $\alpha \in (0, \frac{1}{2})$ and $\epsilon\, \boldsymbol{g} \bullet \boldsymbol{d} < 0$ being the slope to the curve $\Delta J(\epsilon)$ at $\epsilon = 0$. The lower-bound line $(1 - \alpha)\epsilon\, \boldsymbol{g} \bullet \boldsymbol{d}$ (black), adopted in [150] and [155], would be too narrow when α is close to $\frac{1}{2}$, leaving all local minimizers such as $\epsilon_1^\star$ and $\epsilon_2^\star$ outside of the acceptable intevals $I_1^{[1-\alpha]}$, $I_2^{[1-\alpha]}$, and $I_3^{[1-\alpha]}$ (black), which are themselves narrow. The lower-bound line $\beta\epsilon\, \boldsymbol{g} \bullet \boldsymbol{d}$ (purple) proposed in [156], p. 256, and [149], p. 55, with $(1 - \alpha) < \beta < 1$, would enlarge the acceptable sector, which then may contain the minimizers inside the corresponding acceptable intervals $I_1^{[\beta]}$ and $I_2^{[\beta]}$ (purple).

The search for an appropriate step length that satisfies Eq. (123) or Eq. (124) could be carried out by a subprocedure based on, e.g., the bisection method, as suggested in [139], p. 33. Goldstein'rule—also designated as *Goldstein principle* in the classic book [156], p. 256, since it ensured a decrease in the cost function—has been "used only occasionally" per Polak (1997) [149], p. 55, largely superceded by Armijo's rule, and has not been generalized to add stochasticity. On the other hand, the idea behind Armijo's rule is similar to Goldstein's rule, but with a convenient subprocedure[126] to find the appropriate step length.

6.2.3 Inexact line-search, Armijo's rule

Apparently without the knowledge of [150], it was proposed in [151] the following highly popular Armijo step-length search,[127] which recently forms the basis for stochastic line search for use in stochastic

---

[126] See [139], p. 36, Algorithm 36.

[127] As of 2022.07.09, [151] was cited 2301 times in various publications (books, papers) according to Google Scholar, and 1028 times in archival journal papers according to Web of Science. There are references that mention the name

gradient-descent algorithm described in Section 6.3: Stochasticity was added to Armijo's rule in [144], and the concept was extended to second-order line search [145]. Line search based on Armijo's rule is also applied to quasi-Newton method for noisy functions in [158], and to exact and inexact subsampled Newton methods in [159].[128]

Armijo's rule is stated as follows: For $\alpha \in (0, 1)$, $\beta \in (0, 1)$, and $\rho > 0$, use the step length ϵ such that:[129]

$$\epsilon(\boldsymbol{\theta}) = \min_a \{\beta^a \rho \,|\, J(\boldsymbol{\theta} + \beta^j \boldsymbol{d}) - J(\boldsymbol{\theta}) \leq \alpha \beta^a \rho \, \boldsymbol{g} \bullet \boldsymbol{d}\} \tag{125}$$

where the decrease in the cost function along the descent direction $\boldsymbol{d}$, denoted by ΔJ, was defined in Eq. (124), and the descent direction $\boldsymbol{d}$ is related to the gradient $\boldsymbol{g}$ via Eq. (121). The Armijo condition in Eq. (125) can be rewritten as

$$J(\boldsymbol{\theta} + \epsilon \boldsymbol{d}) \leq J(\boldsymbol{\theta}) + \alpha \epsilon \, \boldsymbol{g} \bullet \boldsymbol{d} \,, \tag{126}$$

which is also known as the Armijo sufficient decrease condition, the first of the two Wolfe conditions presented below; see [152], [149], p. 55.[130]

Regarding the paramters α, β, and ρ in the Armijo's rule Eq. (125), [151] selected to fix

$$\alpha = \beta = \frac{1}{2} \,, \text{ and } \rho \in (0, +\infty) \,, \tag{127}$$

and proved a convergence theorem. In practice, ρ cannot be arbitrarily large. Polak (1971) [139], p. 36, also fixed $\alpha = \frac{1}{2}$, but recommended to select $\beta \in (0.5, 0.8)$, based on numerical experiments.[131], and to select[132] $\rho = 1$ to minimize the rate r of geometric progression (from the iterate $\boldsymbol{\theta}_i$, for $i = 0, 1, 2, \ldots$, toward the local minimizer $\boldsymbol{\theta}^\star$) for linear convergence:[133]

$$|J(\boldsymbol{\theta}_{i+1}) - J(\boldsymbol{\theta}^\star)| \leq r^i |J(\boldsymbol{\theta}_0) - J(\boldsymbol{\theta}^\star)| \,, \text{ with } r = 1 - \left(\frac{\rho m}{M}\right)^2 \,, \tag{128}$$

where m and M are the lower and upper bounds of the eigenvalues[134] of the Hessian $\nabla^2 J$, thus $\frac{m}{M} < 1$. In summary, [139] recommended:

$$\alpha = \frac{1}{2} \,, \beta \in (0.5, 0.8) \,, \text{ and } \rho = 1 \,. \tag{129}$$

The pseudocode for deterministic gradient descent with Armijo line search is Algorithm 2, and the pseudocode for deterministic quasi-Newton / Newton with Armijo line search is Algorithm 3. When the Hessian $\boldsymbol{H}(\boldsymbol{\theta}) = \partial^2 J(\boldsymbol{\theta})/(\partial\boldsymbol{\theta})^2$ is positive definite, the Newton descent direction is:

$$\boldsymbol{d} = -\boldsymbol{H}^{-1}(\boldsymbol{\theta}) \, \boldsymbol{g} \,, \tag{130}$$

Armijo, but without referring to the original paper [151], such as [157], clearly indicating that Armijo's rule is a classic, just like there is no need to refer to Newton's original work for Newton's method.

[128] All of these stochastic optimization methods are considered as part of a broader class known as derivative-free optimization methods [160].

[129] [156], p. 491, called the constructive technique in Eq. (125) to obtain the step length ϵ the Goldstein-Armijo algorithm, since [150] and [155] did not propose a method to solve for the step length, while [151] did. See also below Eq. (124) where it was mentioned that a bisection method can be used with Goldstein's rule.

[130] See also [157], p. 34, [148], p. 230.

[131] See [139], p. 301.

[132] To satisfy the condition in Eq. (121), the descent direction $\boldsymbol{d}$ is required to satisfy $(-\boldsymbol{g}) \bullet \boldsymbol{d} \geq \rho \parallel \boldsymbol{d} \parallel \parallel \boldsymbol{g} \parallel$, with $\rho > 0$, so to form an obtuse angle, bounded away from $90°$, with the gradient $\boldsymbol{g}$. But $\rho > 1$ would violate the Schwarz inequality, which requires $|\boldsymbol{g} \bullet \boldsymbol{d}| \leq \parallel \boldsymbol{d} \parallel \parallel \boldsymbol{g} \parallel$.

[133] The inequality in Eq. (128) leads to linear convergence in the sense that $|J(\boldsymbol{\theta}_{i+1}) - J(\boldsymbol{\theta}^\star)| \leq r|J(\boldsymbol{\theta}_i) - J(\boldsymbol{\theta}^\star)|$, for $i = 0, 1, 2, \ldots$, with $|J(\boldsymbol{\theta}_0) - J(\boldsymbol{\theta}^\star)|$ being a constant. See [139], p. 245.

[134] A narrow valley with the minimizer $\boldsymbol{\theta}^\star$ at the bottom would have a very small ratio m/M. See also the use of "small heavy sphere" (also known as "heavy ball") method to accelerate convergence in the case of narrow valley in Section 6.3.2 on stochastic gradient descent with momentum.

1 Gradient descent with Armijo line-search, deterministic
Data:
- Select an initial guess $\boldsymbol{\theta}_0$ (which is part of the network initialization)
- Select 3 Armijo parameters $\alpha \in (0, 1)$, $\beta \in (0, 1)$, and $\rho > 0$, as in Eq. (127)
    - ⋆ Recommend using $\alpha = \frac{1}{2}$, $\beta \in (0.5, 0.8)$, and $\rho = 1$, as in Eq. (129)
Result: Local minimizer $\boldsymbol{\theta}^\star$

2 Define gradient function $\boldsymbol{g}(\cdot) := \partial J(\cdot)/\partial \boldsymbol{\theta}$, obtained through backpropagation; see Figure 48 ;
3 ① **for** $k = 0, 1, 2, \dots$ **do**
4   Compute steepest descent direction $\boldsymbol{d}_k = -\boldsymbol{g}_k = -\boldsymbol{g}(\boldsymbol{\theta}_k)$ as in Eq. (131) ;
5   **if** *Descent direction is not zero*, $\| \boldsymbol{d}_k \| \neq 0$ **then**
6   ▶ Compute step length (learning rate) ϵ_k using Armijo's rule.
7   Initialize step length $\epsilon_k = \rho$;
8   ② **for** *a=1,2,...* **do**
9   **if** *Armijo decrease condition Eq. (126) not satisfied* **then**
10   ▶ Decrease step length
11   Set $\epsilon_k \leftarrow \beta \epsilon_k$;
12   **else**
13   ▶ Armijo decrease condition Eq. (126) satisfied.
14   Stop ② **for** loop, use current step length ϵ_k.
15   **end**
16   **end**
17   ▶ Update network parameter $\boldsymbol{\theta}_k$ to $\boldsymbol{\theta}_{k+1}$ as in Eq. (120)
18   Set $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \epsilon_k \boldsymbol{d}_k$;
19   **else**
20   ▶ Descent direction is zero, $\| \boldsymbol{d}_k \| = 0$;
21   Set the minimizer $\boldsymbol{\theta}^\star = \boldsymbol{\theta}_k$;
22   Stop ① **for** loop.
23   **end**
24   ▶ Continue to next iteration $(k + 1)$
25   Set $k \leftarrow k + 1$;
26 **end**

Algorithm 2: *Steepest gradient descent with Armijo line search, deterministic* (Section 6.2.3, Algorithms 6, 7). Even though Polak (1971) [139], p. 36, recommended to select $\rho = 1$, this parameter is kept here to compare with the stochastic optimization methods in [144] (Algorithm 6) and [145] (Algorithm 7), where there was no such recommendation. See Table 4 for a comparison of the notations used by several authors in relation to Armijo line search.

1 Quasi-Newton / Newton methods with Armijo line-search, deterministic
Data:

- Select an initial guess $\boldsymbol{\theta}_0$ (which is part of the network initialization)
- Select 3 Armijo parameters $\alpha \in (0, 1)$, $\beta \in (0, 1)$, and $\rho > 0$, as in Eq. (127)
    ⋆ Recommend using $\alpha = \frac{1}{2}$, $\beta \in (0.5, 0.8)$, and $\rho = 1$, as in Eq. (129)
- Select stabilization parameter $\delta > 0$ if regularized Newton used, Eq. (132)

Result: Local minimizer $\boldsymbol{\theta}^\star$

2	① **for** $k = 0, 1, 2, \ldots$ **do**
3	Compute gradient $\boldsymbol{g}_k = \boldsymbol{g}(\boldsymbol{\theta}_k)$ at current network parameter $\boldsymbol{\theta}_k$;
4	▶ Find descent direction $\boldsymbol{d}_k$
5	**if** *Gradient not zero*, $\parallel \boldsymbol{g}_k \parallel \neq 0$ **then**
6	Compute Hessian $\boldsymbol{H}_k = \boldsymbol{H}(\boldsymbol{\theta}_k)$ at current network parameter $\boldsymbol{\theta}_k$;
7	**if** *Hessian inverse $\boldsymbol{H}_k^{-1}$ exists* **then**
8	▶ Compute Newton descent direction $\boldsymbol{d}_k = -\boldsymbol{H}_k^{-1}\boldsymbol{g}_k$, Eq. (130)
9	**else**
10	▶ Hessian inverse not existing
11	**if** *Quasi-Newton used* **then**
12	Set descent direction $\boldsymbol{d}_k = -\boldsymbol{g}_k$, Eq. (131) ;
13	**end**
14	**if** *Regularized-Newton used* **then**
15	Compute descent direction $\boldsymbol{d}_k = -\left[\boldsymbol{H}_k + \delta\boldsymbol{I}\right]^{-1}\boldsymbol{g}_k$ Eq. (132) ;
16	**end**
17	**end**
18	▶ Compute step length ϵ_k using Armijo's rule Eq. (125), lines 6-16 in Algorithm 2.
19	▶ Update network parameter $\boldsymbol{\theta}_k$ to $\boldsymbol{\theta}_{k+1}$: Set $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k + \epsilon_k\boldsymbol{d}_k$, Eq. (120) ;
20	**else**
21	▶ Gradient is zero, $\parallel \boldsymbol{g}_k \parallel = 0$;
22	Set the minimizer $\boldsymbol{\theta}^\star = \boldsymbol{\theta}_k$; Stop ① **for** loop.
23	**end**
24	▶ Continue to next iteration $(k + 1)$; Set $k \leftarrow k + 1$;
25	**end**

Algorithm 3: *Quasi-Newton / Newton methods with Armijo line search, deterministic.* Even though [139], p. 36, recommended to select $\rho = 1$, this parameter is kept here to compare with the pseudocodes by [144] (Algorithm 6) and by [145] (Algorithm 7), in which there is no such recommendation. See Table 4 for a comparison of the notations used by several authors in relation to Armijo line search.

When the Hessian $\boldsymbol{H}(\boldsymbol{\theta})$ is not positive definite, e.g., near a saddle point, then quasi-Newton method uses the gradient descent direction $(-\boldsymbol{g} = -\partial J/\partial \boldsymbol{\theta})$ as the descent direction $\boldsymbol{d}$ as in Eq. (120),

$$\boldsymbol{d} = -\boldsymbol{g} = -\partial J(\boldsymbol{\theta})/\partial \boldsymbol{\theta} , \tag{131}$$

and regularized Newton method uses a descent direction based on a regularized Hessian of the form:

$$\boldsymbol{d} = - \left[\boldsymbol{H}(\boldsymbol{\theta}) + \delta \boldsymbol{I} \right]^{-1} \boldsymbol{g} , \tag{132}$$

where δ is a small perturbation parameter (line 15 in Algorithm 3 for deterministic Newton and line 17 in Algorithm 7 for stochastic Newton).[135]

6.2.4 Inexact line-search, Wolfe's rule

The rule introduced in [152] and [153],[136] sometimes called the Armijo-Goldstein-Wolfe's rule (or conditions), particularly in [140] and [141],[137] has been extended to add stochasticity [143],[138] is stated as follows: For $0 < \alpha < \beta < 1$, select the step length (learning rate) ϵ such that (see, e.g., [149], p. 55):

$$J(\boldsymbol{\theta} + \epsilon \boldsymbol{d}) \leq J(\boldsymbol{\theta}) + \alpha \epsilon \boldsymbol{g} \bullet \boldsymbol{d} , \tag{133}$$

$$\frac{\partial J(\boldsymbol{\theta} + \epsilon \boldsymbol{d})}{\partial \boldsymbol{\theta}} \bullet \boldsymbol{d} \geq \beta \boldsymbol{g} \bullet \boldsymbol{d} . \tag{134}$$

The first Wolfe's rule in Eq. (133) is the same as the Armijo's rule in Eq. (126), which ensures that at the updated point $(\boldsymbol{\theta} + \epsilon \boldsymbol{d})$ the cost function value $J(\boldsymbol{\theta} + \epsilon \boldsymbol{d})$ is below the green line $\alpha \epsilon \boldsymbol{g} \bullet \boldsymbol{d}$ in Figure 62.

The second Wolfe's rule in Eq. (134) is to ensure that at the updated point $(\boldsymbol{\theta} + \epsilon \boldsymbol{d})$ the slope of the cost function cannot fall below the (negative) slope of the purple line $\beta \epsilon \boldsymbol{g} \bullet \boldsymbol{d}$ in Figure 62.

For other variants of line search, we refer to [161].

6.3 Stochastic gradient-descent (1st-order) methods

To avoid confusion,[139] we will use the terminology "full batch" (instead of just "batch") when the entire training set is used for training. a minibatch is a small subset of the training set.

In fact, as we shall see, and as mentioned in Remark 6.4, classical optimization methods mentioned in Section 6.2 have been developed further to tackle new problems, such as noisy gradients, encountered in deep-learning training with random mini-batches. There is indeed much room for new research on learning rate since:

> "The learning rate may be chosen by trial and error. This is more of an art than a science, and most guidance on this subject should be regarded with some skepticism." [78], p. 287.

At the time of this writing, we are aware of two review papers on optimization algorithms for machine learning, and in particular deep learning, aiming particularly at experts in the field: [80], as mentioned

---

[135] See, e.g., [149], p. 35, and [78], p. 302, where both cited the Levenberg-Marquardt algorithm as the first to use regularized Hessian.

[136] As of 2022.07.09, [152] was cited 1336 times in various publications (books, papers) according to Google Scholar, and 559 times in archival journal papers according to Web of Science.

[137] The authors of [140] and [141] may not be aware that Goldstein's rule appeared before Armijo's rule, as they cited Goldstein's 1967 book [154], instead of Goldstein's 1965 paper [150], and referred often to Polak (1971) [139], even though it was written in [139], p. 32, that a "step size rule [Eq. (124)] probably first introduced by Goldstein (1967) [154]" was used in an algorithm. See also Footnote 123.

[138] An earlier version of the 2017 paper [143] is the 2015 preprint [147].

[139] See [78], p. 271, about this terminology confusion. The authors of [80] used "stochastic" optimization to mean optimization using random "minibatches" of examples, and "batch" optimization to mean optimization using "full batch" or full training set of examples.

above, and [162]. Our review complements these two review papers. We are aiming here at bringing first-time learners up to speed to benefit from, and even to hopefully enjoy, reading these and others related papers. To this end, we deliberately avoid the dense mathematical-programming language, not familiar to readers outside the field, as used in [80], while providing more details on algorithms that have proved important in deep learning than [162].

Listed below are the points that distinguish the present paper from other reviews. Similar to [78], both [80] and [162]:

- Only mentioned briefly in words the connection between SGD with momentum to mechanics without detailed explanation using the equation of motion of the "heavy ball", a name not as accurate as the original name "small heavy sphere" by Polyak (1964) [3]. These references also did not explain how such motion help to accelerate convergence; see Section 6.3.2.

- Did not discuss recent practical add-on improvements to SGD such as step-length tuning (Section 6.3.3) and step-length decay (Section 6.3.4), as proposed in [55]. This information would be useful for first-time learners.

- Did not connect step-length decay to simulated annealing, and did not explain the reason for using the name "annealing"[140] in deep learning by connecting to stochastic differential equation and physics; see Remark 6.9 in Section 6.3.5.

- Did not review an alternative to step-length decay by increasing minibatch size, which could be more efficient, as proposed in [164]; see Section 6.3.5.

- Did not point out that the exponential smoothing method (or running average) used in adaptive learning-rate algorithms dated since the 1950s in the field of forecasting. None of these references acknowledged the contributions made in [165] and [166], in which exponential smoothing from time series in forecasting was probably first brought to machine learning. See Section 6.5.3.

- Did not discuss recent adaptive learning-rate algorithms such as AdamW [56].[141] These authors also did not discuss the criticism of adaptive methods in [55]; see Section 6.5.10.

- Did not discuss classical line-search rules—such as [150], [151],[142] [152] (Sections 6.2.2, 6.2.3, 6.2.4)—that have been recently generalized to add stochasticity, e.g., [143], [144], [145]; see Sections 6.6, 6.7.

6.3.1 Standard SGD, minibatch, fixed learning-rate schedule

The stochastic gradient descent algorithm, originally introduced by Robbins & Monro (1951a) [167] (another classic) according to many sources,[143] has been playing an important role in training deep-learning networks:

> "Nearly all of deep learning is powered by one very important algorithm: stochastic gradient descent (SGD). Stochastic gradient descent is an extension of the gradient descent algorithm." [78], p. 147.

---

[140] The authors of [162] only cited [163] for a brief mention of "simulated annealing" as an example of "heuristic optimizers", with no discussion, and no connection to step length decay. See also Remark 6.10 on "Metaheuristics".
[141] The authors of [162] only cited [56] in passing, without reviewing AdamW, which was not even mentioned.
[142] The authors of [80] only cited Armijo (1966) [151] once for a pseudocode using line search.
[143] See, e.g., [80]—in which there was a short bio of Robbins, the first author of [167]—and [162] [144].

Minibatch. The number M of examples in a training set $\mathbb{X}$ could be very large, rendering prohibitively expensive to evaluate the cost function and to compute the gradient of the cost function with respect to the number of parameters, which by itself could also be very large. At iteration k within a training session τ, let $\mathbb{I}_k^{|\mathsf{m}|}$ be a randomly selected set of m indices, which are elements of the training-set indices $[\mathsf{M}] = \{1, \ldots, \mathsf{M}\}$. Typically, m is much smaller than M:[144]

> "The minibatch size m is typically chosen to be a relatively small number of examples, ranging from one to a few hundred. Crucially, m is usually held fixed as the training set size M grows. We may fit a training set with billions of examples using updates computed on only a hundred examples." [78], p. 148.

Generated as in Eq. (136), the random-index sets $\mathbb{I}_k^{|\mathsf{m}|}$, for $k = 1, 2, \ldots$, are non-overlapping such that after $k_{max} = \mathsf{M}/\mathsf{m}$ iterations, all examples in the training set are covered, and a training session, or training epoch,[145] is completed (line 6 in Algorithm 4). At iteration k of a training epoch τ, the random minibatch $\mathbb{B}_k^{|\mathsf{m}|}$ is a set of m examples pulled out from the much larger training set $\mathbb{X}$ using the random indices in $\mathbb{I}_k^{|\mathsf{m}|}$, with the corresponding targets in the set $\mathbb{T}_k^{|\mathsf{m}|}$:

$$M_1 = \{1, \ldots, \mathsf{M}\} =: [\mathsf{M}] \; , \; k_{max} = \mathsf{M}/\mathsf{m} \tag{135}$$

$$\mathbb{I}_k^{|\mathsf{m}|} = \{i_{1,k}, \ldots, i_{\mathsf{m},k}\} \subseteq M_k \; , \; M_{k+1} = M_k - \mathbb{I}_k^{|\mathsf{m}|} \text{ for } k = 1, \ldots, k_{max} \tag{136}$$

$$\mathbb{B}_k^{|\mathsf{m}|} = \{\boldsymbol{x}^{|i_{1,k}|}, \cdots, \boldsymbol{x}^{|i_{\mathsf{m},k}|}\} \subseteq \mathbb{X} = \{\boldsymbol{x}^{|1|}, \cdots, \boldsymbol{x}^{|\mathsf{M}|}\} \tag{137}$$

$$\mathbb{T}_k^{|\mathsf{m}|} = \{\boldsymbol{y}^{|i_{1,k}|}, \cdots, \boldsymbol{y}^{|i_{\mathsf{m},k}|}\} \subseteq \mathbb{Y} = \{\boldsymbol{y}^{|1|}, \cdots, \boldsymbol{y}^{|\mathsf{M}|}\} \tag{138}$$

Note that once the random index set $\mathbb{I}_k^{|\mathsf{m}|}$ had been selected, it was deleted from its superset M_k to form M_{k+1} so the next random set $\mathbb{I}_{j+1}^{|\mathsf{m}|}$ would not contain indices already selected in $\mathbb{I}_k^{|\mathsf{m}|}$.

Unlike the iteration counter k within a training epoch τ, the global iteration counter j is not reset to 1 at the beginning of a new training epoch $\tau + 1$, but continues to increment for each new minibatch. Plots versus epoch counter τ and plots versus global iteration counter j could be confusing; see Remark 6.17 and Figure 78.

Cost and gradient estimates. The cost-function estimate is the average of the cost functions, each of which is the cost function of an example $\boldsymbol{x}^{|i_k|}$ in the minibatch for iteration k in training epoch τ:

$$\widetilde{J}(\boldsymbol{\theta}) = \frac{1}{\mathsf{m}} \sum_{a=1}^{a=\mathsf{m}\leq\mathsf{M}} J_{i_a}(\boldsymbol{\theta}) \; , \; \text{with } J_{i_a}(\boldsymbol{\theta}) = J(f(\boldsymbol{x}^{|i_a|}, \boldsymbol{\theta}), \boldsymbol{y}^{|i_a|}) \; , \; \text{and } \boldsymbol{x}^{|i_a|} \in \mathbb{B}_k^{|\mathsf{m}|} \; , \boldsymbol{y}^{|i_a|} \in \mathbb{T}_k^{|\mathsf{m}|} \; , \tag{139}$$

---

[144] As of 2010.04.30, the ImageNet database contained more than 14 million images; see Original website, Internet archive, Figure 3 and Footnote 14. There is a slight inconsistency in notation in [78], where on p. 148, m and m' denote the number of examples in the training set and in the minibatch, respectively, whereas on p. 274, m denote the number of examples in a minibatch. In our notation, m is the dimension of the output array $\boldsymbol{y}$, whereas m (in a different font) is the minibatch size; see Footnote 88. In theory, we write $\mathsf{m} \leq \mathsf{M}$ in Eq. (136); in practice, $\mathsf{m} \ll \mathsf{M}$.

[145] An epoch, or training session, τ is explicitly defined here as when the minibatches as generated in Eqs. (135)-(137) covered the whole dataset. In [78], the first time the word "epoch" appeared was in Figure 7.3 caption, p. 239, where it was defined as a "training iteration", but there was no explicit definition of "epoch" (when it started and when it ended), except indirectly as a "training pass through the dataset", p. 274. See Figure 151 in Section 14.7 on "Lack of transparency and irreproducibility of results" in recent deep-learning papers.

where we wrote the random index as i_a instead of $i_{a,k}$ as in Eq. (136) to alleviate the notation. The corresponding gradient estimate is:

$$\widetilde{\boldsymbol{g}}(\boldsymbol{\theta}) = \frac{\partial \widetilde{J}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \frac{1}{\mathsf{m}} \sum_{a=1}^{a=\mathsf{m} \leq \mathsf{M}} \frac{\partial J_{i_a}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \frac{1}{\mathsf{m}} \sum_{a=1}^{a=\mathsf{m} \leq \mathsf{M}} \boldsymbol{g}_{i_a} \ . \tag{140}$$

The pseudocode for the standard SGD[146] is given in Algorithm 4. The epoch stopping criterion (line 1 in Algorithm 4) is usually determined by a computation "budget", i.e., the maximum number of epochs allowed. For example, [145] set a budget of 1,600 epochs maximum in their numerical examples.

Problems and resurgence of SGD. There are several known problems with SGD:

> "Despite the prevalent use of SGD, it has known challenges and inefficiencies. First, the direction may not represent a descent direction, and second, the method is sensitive to the step-size (learning rate) which is often poorly overestimated." [144]

For the above reasons, it may not be appropriate to use the norm of the gradient estimate being small as stationarity condition, i.e., where the local minimizer or saddle point is located; see the discussion in [145] and stochastic Newton Algorithm 7 in Section 6.7.

Despite the above problems, SGD has been brought back to the forefront state-of-the-art algorithm to beat, surpassing the performance of adaptive methods, as confirmed by three recent papers: [55], [168], [56]; see Section 6.5.9 on criticism of adaptive methods.

Add-on tricks to improve SGD. The following tricks can be added onto the vanilla (standard) SGD to improve its performance; see also the pseudocode in Algorithm 4:

- Momentum and accelerated gradient: Improve (accelerate) convergence in narrow valleys, Section 6.3.2

- Initial-step-length tuning: Find effective initial step length ϵ_0, Section 6.3.3

- Step-length decaying or annealing: Find an effective learning-rate schedule[147] to decrease the step length ϵ as a function of epoch counter τ or global iteration counter j, cyclic annealing, Section 6.3.4

- Minibatch-size increase, keeping step length fixed, equivalent annealing, Section 6.3.5

- Weight decay, Section 6.3.6

6.3.2 Momentum and fast (accelerated) gradient

The standard update for gradient descent is Eq. (120) would be slow when encountering deep and narrow valley, as shown in Figure 63, and can be replaced by the general update with momentum as follows:

$$\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k - \epsilon_k \widetilde{\boldsymbol{g}}(\widetilde{\boldsymbol{\theta}}_k + \gamma_k(\widetilde{\boldsymbol{\theta}}_k - \widetilde{\boldsymbol{\theta}}_{k-1})) + \zeta_k(\widetilde{\boldsymbol{\theta}}_k - \widetilde{\boldsymbol{\theta}}_{k-1}) \ , \tag{141}$$

from which the following methods are obtained (line 10 in Algorithm 4):

- Standard SGD update Eq. (120) with $\gamma_k = \zeta_k = 0$ [49]

---

[146] See also [78], p. 286, Algorigthm 8.1; [80], p. 243, Algorithm 4.1.

[147] See Figure 151 in Section 14.7 on "Lack of transparency and irreproducibility of results" in recent deep-learning papers.

1 SGD, standard, momentum, accelerated gradient

Data:

• Select an initial guess $\widetilde{\boldsymbol{\theta}}_0$ (which is part of the network initialization)

• Select variable † as epoch τ or as global iteration j, Eq. (148), and budget τ_{max} or j_{max}

• Select learning-rate schedule $\epsilon(†) \in \{$ Eq. (147)-Eq. (151) $\}$, and parameters ϵ_0, k_c, ϖ

• Select parameter ζ for momentum, and γ for Nesterov Accelerated Gradient, Eq. (141), if use

• Select cosine annealing multiplier $a(†)$, Eq. (154), if use

• Select a value for m the number of examples in the minibatches

Result: Local minimizer estimate $\widetilde{\boldsymbol{\theta}}^\star$, satisfying a stopping criterion, Section 6.1.

2 Initialize network parameters $\widetilde{\boldsymbol{\theta}}_1^\star = \widetilde{\boldsymbol{\theta}}_0$ and global iteration counter $j = 1$;

3 ① **for** $\tau = 1, 2, \ldots, \tau_{max}$ **do**

4   ▶ Begin training epoch τ

5   Initialize $\widetilde{\boldsymbol{\theta}}_1 = \widetilde{\boldsymbol{\theta}}_\tau^\star$ (from previous training epoch) ;

6   ② **for** $k = 1, \ldots, k_{max}$, *Eq. (135)* **do**

7   Obtain random minibatch $\mathbb{B}_k^{|m|}$ containing m examples, Eq. (137) ;

8   Compute steepest-descent direction estimate $\widetilde{\boldsymbol{d}}_k = -\widetilde{\boldsymbol{g}}(\widetilde{\boldsymbol{\theta}}_k)$, Eq. (140) ;

9   Compute learning rate $\epsilon_k = \epsilon(†)$;

10   ▶ Update network parameter $\widetilde{\boldsymbol{\theta}}_k$ to $\widetilde{\boldsymbol{\theta}}_{k+1}$, Eq. (183)

11   **if** $j < j_{max}$ *(if use global-iteration budget)* **then**

12   Increment global iteration counter $j \leftarrow j + 1$

13   **else**

14   Stop computation; exit ;

15   **end**

16   **end**

17   **if** *Stopping criterion not met (if used, Section 6.1)* **then**

18   ▶ Continue to next training epoch $(\tau + 1)$

19   Update minimizer estimate for epoch $(\tau + 1)$: $\widetilde{\boldsymbol{\theta}}_{\tau+1}^\star = \widetilde{\boldsymbol{\theta}}_{k_{max}}$

20   Set $\tau \leftarrow \tau + 1$;

21   **else**

22   ▶ Stopping criterion met

23   Set optimal paramter $\widetilde{\boldsymbol{\theta}}^\star = \widetilde{\boldsymbol{\theta}}_\tau$; Stop computation, exit ;

24   **end**

25 **end**

26 **if** *Stopping criterion NOT used (Section 6.1)* **then**

27   Find validation-error global minimum at epoch $\tau^\star$, and set optimal parameter $\widetilde{\boldsymbol{\theta}}^\star = \widetilde{\boldsymbol{\theta}}_{\tau^\star}$ at $\tau^\star$;
     see Figure 58 in Section 6.1.

28 **end**

Algorithm 4: *SGD, standard, momentum, accelerated gradient, cyclic annealing* (Sections 6.1, 6.3.1, 6.3.2, 6.3.7, 6.5.1, 6.6, 6.7, Algorithms 5, 6, 7), with fixed learning-rate schedule $\epsilon(†)$ and fixed minibatch size m. The case with constant learning rate $\epsilon_k = \epsilon_0$, and minibatch-size increasing in Section 6.3.5 is not implemented here to keep the pseudocode simple. For adaptive learning-rate methods, see Section 6.5. For SGD with stochastic Armijo length search and adaptive minibatch, see Algorithm 6, where the outer ① **for** loop over training epochs is omitted for conciseness, and similarly for Algorithm 5 and Algorithm 7.
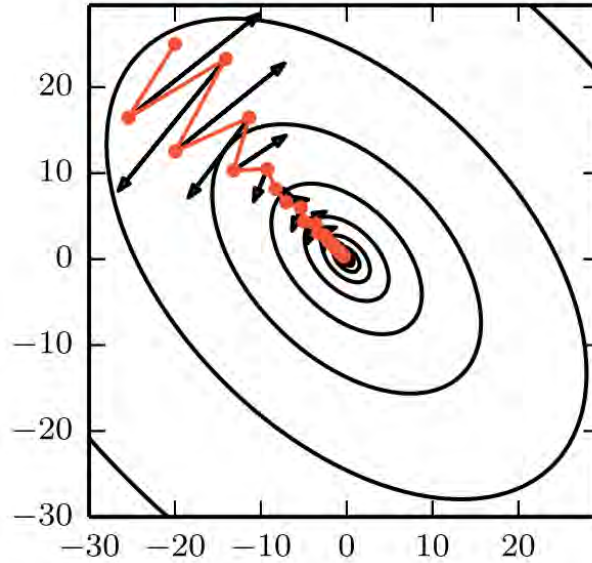
Figure 63: *SGD with momentum, small heavy sphere* Section 6.3.2. The descent direction (negative gradient, black arrows) bounces back and forth between the steep slopes of a deep and narrow valley. The small-heavy-sphere method, or SGD with momentum, follows a faster descent (red path) toward the bottom of the valley. See the cost-function landscape with deep valleys in Figure 55. Figure from [78], p. 289. (Figure reproduced with permission of the authors.)

- SGD with classical momentum: $\gamma_k = 0$ and $\zeta_k \in (0, 1)$ ("small heavy sphere" or heavy point mass)[148] [3]

- SGD with fast (accelerated) gradient:[149] $\gamma_k = \zeta_k \in (0, 1)$, Nesterov (1983 [50], 2018 [51])

The continuous counterpart of the parameter update Eq. (141) with classical momentum, i.e., when $\gamma_k \in (0, 1)$ and $\zeta_k = 0$, is the equation of motion of a heavy point mass (thus no rotatory inertia) under viscous friction at slow motion (proportional to velocity) and applied force $-\widetilde{g}$ given below with its discretization by finite difference in time, where h_k and h_{k-1} are the time-step sizes [169]:

$$\frac{d^2\widetilde{\boldsymbol{\theta}}}{(dt)^2} + \nu \frac{d\widetilde{\boldsymbol{\theta}}}{dt} = -\widetilde{\boldsymbol{g}} \Rightarrow \frac{\frac{\widetilde{\boldsymbol{\theta}}_{k+1} - \widetilde{\boldsymbol{\theta}}_k}{h_k} - \frac{\widetilde{\boldsymbol{\theta}}_k - \widetilde{\boldsymbol{\theta}}_{k-1}}{h_{k-1}}}{h_k} + \nu \frac{\widetilde{\boldsymbol{\theta}}_{k+1} - \widetilde{\boldsymbol{\theta}}_k}{h_k} = -\widetilde{\boldsymbol{g}}_k \ , \tag{142}$$

$$\widetilde{\boldsymbol{\theta}}_{k+1} - \widetilde{\boldsymbol{\theta}}_k - \zeta_k(\widetilde{\boldsymbol{\theta}}_k - \widetilde{\boldsymbol{\theta}}_{k-1}) = -\epsilon_k \widetilde{\boldsymbol{g}}_k \ , \text{ with } \zeta_k = \frac{h_{k-1}}{h_k} \frac{1}{1 + \nu h_k} \text{ and } \epsilon_k = (h_k)^2 \frac{1}{1 + \nu h_k} \ , \tag{143}$$

which is the same as the update Eq. (141) with $\gamma_k = 0$. The term $\zeta_k(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1})$ is often called the "momentum" term since it is proportional to (discretized) velocity. [3] on the other hand explained the term $\zeta_k(\boldsymbol{\theta}_k - \boldsymbol{\theta}_{k-1})$ as "giving inertia to the motion, [leading] to motion along the "essential" direction, i.e. along 'the bottom of the trough' ", and recommended to select $\zeta_k \in (0.8, 0.99)$, i.e., close to 1, without explanation. The reason is to have low friction, i.e., ν small, but not zero friction ($\nu = 0$), since friction is important to slow down the motion of the sphere up and down the valley sides (like skateboarding from side to side in a half-pipe), thus accelerate convergence toward the trough of the valley; from Eq. (143), we have

$$h_k = h_{k-1} = h \text{ and } \nu \in [0, +\infty) \Rightarrow \zeta_k \in (0, 1] \ , \text{ with } \nu = 0 \Rightarrow \zeta_k = 1 \tag{144}$$

---

[148] Often called by the more colloquial "heavy ball" method; see Remark 6.6.
[149] Sometimes referred to as Nesterov's Accelerated Gradient (NAG) in the deep-learning literature.

Remark 6.5. The choice of the momentum parameter ζ in Eq. (141) is not trivial. If ζ is too small, the signal will be too noisy; if ζ is too large, "the average will lag too far behind the (drifting) signal" [165], p. 212. Even though Polyak (1964) [3] recommended to select $\zeta \in (0.8, 0.99)$, as explained above, it was reported in [78], p. 290: "Common values of ζ used in practice include 0.5, 0.9, and 0.99. Like the learning rate, ζ may also be adapted over time. Typically it begins with a small value and is later raised. Adapting ζ over time is less important than shrinking ϵ over time". The value of $\zeta = 0.5$ would correspond to relatively high friction μ, slowing down the motion of the sphere, compared to $\zeta = 0.99$.

Figure 68 from [170] shows the convergence of some adaptive learning-rate algorithms: AdaGrad, RMSProp, SGDNesterov (accelerated gradient), AdaDelta, Adam.

In their remarkable paper, the authors of [55] used a constant momentum parameter $\zeta = 0.9$; see criticism of adaptive methods in Section 6.5 and Figure 73 comparing SGD, SGD with momentum, AdaGrad, RMSProp, Adam.[150]

See Figure 151 in Section 14.7 on "Lack of transparency and irreproducibility of results" in recent deep-learning papers. ∎

For more insight into the update Eq. (143), consider the case of constant coefficients $\zeta_k = \zeta$ and $\epsilon_k = \epsilon$, and rewrite this recursive relation in the form:

$$\widetilde{\boldsymbol{\theta}}_{k+1} - \widetilde{\boldsymbol{\theta}}_k = -\epsilon \sum_{i=0}^{k} \zeta^i \widetilde{\boldsymbol{g}}_{k-i} \, , \text{ using } \widetilde{\boldsymbol{\theta}}_1 - \widetilde{\boldsymbol{\theta}}_0 = -\epsilon \widetilde{\boldsymbol{g}}_0 \, , \tag{145}$$

i.e., without momentum for the first term. So the effective gradient is the sum of all gradients from the beginning $i = 0$ until the present $i = k$ weighted by the exponential function ζ^i so there is a fading memory effect, i.e., gradients that are farther back in time have less influence than those closer to the present time.[151] The summation term in Eq. (145) also provides an explanation of how the "inertia" (or momentum) term work: (1) Two successive opposite gradients would cancel each other, whereas (2) Two successive gradients in the same direction (toward the trough of the valley) would reinforce each other. See also [171], pp. 104-105, and [172] who provided a similar explanation:

> "Momentum is a simple method for increasing the speed of learning when the objective function contains long, narrow and fairly straight ravines with a gentle but consistent gradient along the floor of the ravine and much steeper gradients up the sides of the ravine. The momentum method simulates a heavy ball rolling down a surface. The ball builds up velocity along the floor of the ravine, but not across the ravine because the opposing gradients on opposite sides of the ravine cancel each other out over time."

In recent years, Polyak (1964) [3] (English version)[152] has often been cited for the classical momentum ("small heavy sphere") method to accelerate the convergence in gradient descent, but not so before, e.g., the authors of [22] [175] [176] [177] [172] used the same method without citing [3]. Several books on optimization not related to neural networks, many of them well-known, also did not mention this method: [139] [178] [149] [157] [148] [179]. Both the original Russian version and the English translated version

---

[150] A nice animation of various optimizers (SGD, SGD with momentum, AdaGrad, AdaDelta, RMSProp) can be found in S. Ruder, 'An overview of gradient descent optimization algorithms', updated on 2018.09.02 (Original website).

[151] See also Section 6.5.3 on time series and exponential smoothing.

[152] Polyak (1964) [3]'s English version appeared before 1979, as cited [173], where a similar classical dynamics of a "small heavy sphere" or heavy point mass was used to develop an iterative method to solve nonlinear systems. There, the name Polyak was spelled as "Poljak" as in the Russian version. The earliest citing of the Russian version, with the spelling "Poljak" was in [156] and in [174], but the terminology "small heavy sphere" was not used. See also [171], p. 104 and p. 481, where the Russian version of [3] was cited.

[3] (whose author's name was spelled as "Poljak" before 1990) were cited in the book on neural networks [180], in which another neural-network book [171] was referred to for a discussion of the formulation.[153]

Remark 6.6. *Small heavy sphere, or heavy point mass, is better name.* Because the rotatory motion is not considered in Eq. (142), the name "small heavy sphere" given in [3] is more precise than the more colloquial name "heavy ball" often given to the SGD with classical momentum,[154] since "small" implies that rotatory motion was neglected, and a "heavy ball" could be as big as a bowling ball[155] for which rotatory motion cannot be neglected. For this reason, "heavy point mass" would be a precise alternative name. ∎

Remark 6.7. For Nesterov's fast (accelerated) gradient method, many references referred to [50].[156] The authors of [78], p. 291, also referred to Nesterov's 2004 monograph, which was mentioned in the Preface of, and the material of which was included in, [51]. For a special class of strongly convex functions,[157] the step length can be kept constant, while the coefficients in Nesterov's fast gradient method varied, to achieve optimal performance, [51], p. 92. "Unfortunately, in the stochastic gradient case, Nesterov momentum does not improve the rate of convergence" [78], p. 292. ∎

6.3.3 Initial-step-length tuning

The initial step length ϵ_0, or learning-rate initial value, is one of the two most influential hyperparameters to tune, i.e., to find the best performing values. During tuning, the step length ϵ is kept constant at ϵ_0 in the parameter update Eq. (120) throughout the optimization process, i.e., a fixed step length is used, without decay as in Eqs. (147-150) in Section 6.3.4.

The following simple tuning method was proposed in [55]:

> "To tune the step sizes, we evaluated a logarithmically-spaced grid of five step sizes. If the best performance was ever at one of the extremes of the grid, we would try new grid points so that the best performance was contained in the middle of the parameters. For example, if we initially tried step sizes 2, 1, 0.5, 0.25, and 0.125 and found that 2 was the best performing, we would have tried the step size 4 to see if performance was improved. If performance improved, we would have tried 8 and so on."

The above logarithmically-spaced grid was given by 2^k, with $k = 1, 0, -1, -2, -3$. This tuning method appears effective as shown in Figure 73 on the CIFAR-10 dataset mentioned above, for which the following values for ϵ_0 had been tried for different optimizers, even though the values did not always belong to the sequence $\{a^k\}$, but could include close, rounded values:

- SGD (Section 6.3.1): 2, 1, 0.5 (best), 0.25, 0.05, 0.01[158]

- SGD with momentum (Section 6.3.2): 2, 1, 0.5 (best), 0.25, 0.05, 0.01

- AdaGrad (Section 6.5): 0.1, 0.05, 0.01 (best, default), 0.0075, 0.005

---

[153] See [180], p. 159, p. 115, and [171], p. 104, respectively. The name "Polyak" was spelled as "Poljak" before 1990, [171], p. 481, and sometimes as "Polyack", [169]. See also [181].

[154] See, e.g., [171], p. 104, [180], p. 115, [169], [181].

[155] Or the "Times Square Ball", Wikipedia, version 05:17, 29 December 2019.

[156] Reference [50] cannot be found from the Web of Science as of 2020.03.18, perhaps because it was in Russian, as indicated in Ref. [35] in [51], p. 582, where Nesterov's 2004 monograph was Ref. [39].

[157] A function $f(\cdot)$ is strongly convex if there is a constant $\mu > 0$ such that for any two points x and y, we have $f(y) \geq f(x) + \langle \nabla f(x), y - x \rangle + \frac{1}{2}\mu \| y - x \|^2$, where $\langle \cdot, \cdot \rangle$ is the inner (or dot) product, [51], p. 74.

[158] The last two values $\{0.05, 0.01\}$ did not belong to the sequence 2^k, with k being integers, since $2^{-3} = 0.125$, $2^{-4} = 0.0625$ and $2^{-5} = 0.03125$.

- RMSProp (Section 6.5): 0.005, 0.001, 0.0005, 0.0003 (best), 0.0001

- Adam (Section 6.5): 0.005, 0.001 (default), 0.0005, 0.0003 (best), 0.0001, 0.00005

6.3.4 Step-length decay, annealing and cyclic annealing

In the update of the parameter $\boldsymbol{\theta}$ as in Eq. (120), the learning rate (step length) ϵ has to be reduced gradually as a function of either the epoch counter τ or of the global iteration counter j. Let $\dagger$ represents either τ or j, depending on user's choice.[159] If the learning rate ϵ is a function of epoch τ, then ϵ is held constant in all iterations $k = 1, \ldots, k_{max}$ within epoch τ, and we have the relation:

$$j = (\tau - 1) * k_{max} + k \,. \tag{146}$$

The following learning-rate scheduling, linear with respect to $\dagger$, is one option:[160]

$$\epsilon(\dagger) = \begin{cases} \left(1 - \dfrac{\dagger}{\dagger_c}\right) \epsilon_0 + \dfrac{\dagger}{\dagger_c} \epsilon_{\dagger_c} & \text{for } 0 \le \dagger \le \dagger_c \\ \epsilon_{\dagger_c} & \text{for } \dagger_c \le \dagger \end{cases} \tag{147}$$

$$\dagger = \text{epoch } \tau \text{ or global iteration } j \tag{148}$$

where ϵ_0 is the learning-rate initial value, and $\epsilon_{\dagger_c}$ the constant learning-rate value when $\dagger \ge \dagger_c$. Other possible learning-rate schedules are:[161]

$$\epsilon(\dagger) = \frac{\epsilon_0}{\sqrt{\dagger}} \to 0 \text{ as } \dagger \to \infty \,, \tag{149}$$

$$\epsilon(\dagger) = \frac{\epsilon_0}{\dagger} \to 0 \text{ as } \dagger \to \infty \,, \tag{150}$$

with $\dagger$ defined as in Eq. (147), even though authors such as [182] and [183] used Eq. (149) and Eq. (150) with $\dagger = j$ as global iteration counter.

Another step-length decay method proposed in [55] is to reduce the step length $\epsilon(\tau)$ for the current epoch τ by a factor $\varpi \in (0, 1)$ when the cost estimate $\widetilde{J}_{\tau-1}$ at the end of the last epoch $(\tau - 1)$ is greater than the lowest cost in all previous global iterations, with $\widetilde{J}_j$ denoting the cost estimate at global iteration j, and $k_{max}(\tau - 1)$ the global iteration number at the end of epoch $(\tau - 1)$:

$$\epsilon(\tau) = \begin{cases} \varpi \, \epsilon(\tau - 1) & \text{if } \widetilde{J}_{\tau-1} > \min_j \{\widetilde{J}_j, j = 1, \ldots, k_{max}(\tau - 1)\} \\ \epsilon(\tau - 1) & \text{Otherwise} \end{cases} \tag{151}$$

Recall, k_{max} is the number of non-overlapping minibatches that cover the training set, as defined in Eq. (135). [55] set the step-length decay parameter $\varpi = 0.9$ in their numerical examples, in particular Figure 73.

---

[159] The Avant Garde font $\dagger$ is used to avoid confusion with t, the time variable used in relation to recurrent neural networks; see Section 7 on "Dynamics, sequential data, sequence modeling", and Section 13.2.2 on "Dynamics, time dependence, Volterra series". Many papers on deep-learning optimizers used t as global iteration counter, which is denoted by j here; see, e.g., [182], [56].

[160] See [78], p. 287, where it was suggested that $\dagger_c$ in Eq. (147) would be "set to the number of iterations required to make a few hundred passes through the training set," and $\epsilon_{\dagger_c}$ "should be set to roughly 1 percent the value of ϵ_0". A "few hundred passes through the training set" means a few hundred epochs; see Footnote 161. In [78], p. 286, Algorithm 1 SGD for "training iteration k" should mean for "training epoch k", and the learning rate "ϵ_k" would be held constant within "epoch k".

[161] See [182], p. 3, below Algorithm 1 and just below the equation labeled "(Sgd)". After, say, 400 global iterations, i.e., $\dagger = j = 400$, then $\epsilon_{400} = 5\% \, \epsilon_0$ according to Eq. (149), and $\epsilon_{400} = 0.25\% \, \epsilon_0$ according to Eq. (150), whereas $\epsilon_{400} = 1\% \, \epsilon_0$ according to Eq. (147). See Footnote 160, and also Figure 151 in Section 14.7 on "Lack of transparency and irreproducibility of results" in recent deep-learning papers.

Cyclic annealing. In additional to decaying the step length ϵ, which is already annealing, cyclic annealing is introduced to further reduce the step length down to zero ("cooling"), quicker than decaying, then bring the step length back up rapidly (heating), and doing so for several cycles. The cosine function is typically used, such as shown in Figure 75, as a multiplicative factor $\mathfrak{a}_k \in [0, 1]$ to the step length ϵ_k in the parameter update, and thus the name "cosine annealing":

$$\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k - \mathfrak{a}_k \epsilon_k \widetilde{\boldsymbol{g}}_k \, , \tag{152}$$

as an add-on to the parameter update for vanilla SGD Eq. (120), or

$$\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k - \mathfrak{a}_k \epsilon_k \widetilde{\boldsymbol{g}}(\widetilde{\boldsymbol{\theta}}_k + \gamma_k(\widetilde{\boldsymbol{\theta}}_k - \widetilde{\boldsymbol{\theta}}_{k-1})) + \zeta_k(\widetilde{\boldsymbol{\theta}}_k - \widetilde{\boldsymbol{\theta}}_{k-1}) \tag{153}$$

as an add-on to the parameter update for SGD with momentum and accelerated gradient Eq. (141). The cosine annealing factor can take the form [56]:

$$\mathfrak{a}_k = 0.5 + 0.5 \cos(\pi T_{cur}/T_p) \in [0, 1] \, , \text{ with } T_{cur} := j - \sum_{q=1}^{q=p-1} T_q \tag{154}$$

where T_{cur} is the number of epochs from the start of the last warm restart at the end of epoch $\sum_{q=1}^{q=p-1} T_q$, where $\mathfrak{a}_k = 1$ ("maximum heating"), j the current global iteration counter, T_p the maximum number of epochs allowed for the current pth annealing cycle, during which T_{cur} would go from 0 to T_p, when $\mathfrak{a}_k = 0$ ("complete cooling"). Figure 75 shows 4 annealing cycles, which helped reduce dramatically the number of epochs needed to achieve the same lower cost as obtained without annealing.

Figure 74 shows the effectiveness of cosine annealing in bringing down the cost rapidly in the early stage, but there is a diminishing return, as the cost reduction decreases with the number of annealing cycle. Up to a point, it is no longer as effective as SGD with weight decay in Section 6.3.6.

Convergence conditions. The sufficient conditions for convergence, for convex functions, are[162]

$$\sum_{j=1}^{\infty} \epsilon_j^2 < \infty \, , \text{ and } \sum_{j=1}^{\infty} \epsilon_j = \infty \, . \tag{155}$$

The inequality on the left of Eq. (155), i.e., the sum of the squared of the step lengths being finite, ensures that the step length would decay quickly to reach the minimum, but is valid only when the minibatch size is fixed. The equation on the right of Eq. (155) ensures convergence, no matter how far the initial guess was from the minimum [164].

In Section 6.3.5, the step-length decay is shown to be equivalent to minibatch-size increase and simulated annealing in the sense that there would be less fluctuation, and thus lower "temperature" (cooling) by analogy to the physics governed by the Langevin stochastic differential equation and its discrete version, which is analogous to the network parameter update.

---

[162] Eq. (155) are called the "stepsize requirements" in [80], and "sufficient condition for convergence" in [78], p. 287, and in [184]. Robbins & Monro (1951b) [49] were concerned with solving $M(x) = \alpha$ when the function $M(\cdot)$ is not known, but the distribution of the output, as a random variable, $\mathrm{y} = \mathrm{y}(x)$ is assumed known. For the network training problem at hand, one can think of $M(x) = \| \nabla J(x) \|$, i.e., the magnitude of the gradient of the cost function J at $x = \| \widetilde{\boldsymbol{\theta}} - \boldsymbol{\theta} \|$, the distance from a local minimizer, and $\alpha = 0$, i.e., the stationarity point of $J(\cdot)$. In [49]—in which there was no notion of "epoch τ" but only global iteration counter j—Eq. (6) on p. 401 corresponds to Eq. (155)$_1$ (first part), and Eq. (27) on p. 404 corresponds to Eq. (155)$_2$ (second part). Any sequence $\{\epsilon_k\}$ that satisfied Eq. (155) was called a sequence of type $1/k$; the convergence Theorem 1 on p. 404 and Theorem 2 on p. 405 indicated that the sequence of step length $\{\epsilon_k\}$ being of type $1/k$ was only one among other sufficient conditions for convergence. In Theorem 2 of [49], the additional sufficient conditions were Eq. (33), $M(x) = \| \nabla J(x) \|$ non decreasing, Eq. (34), $M(0) = \| \nabla J(0) \| = 0$, and Eq. (35), $M'(0) = \| \nabla^2 J(0) \| > 0$, i.e., the iterates $\{x_k \,|\, k = 1, 2, \ldots\}$, fell into a local convex bowl.

6.3.5 Minibatch-size increase, fixed step length, equivalent annealing

The minibatch parameter update from Eq. (141), without momentum and accelerated gradient, which becomes Eq. (120), can be rewritten to introduce the error due to the use of the minibatch gradient estimate $\widetilde{g}$ instead of the full-batch gradient g as follows:

$$\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k - \epsilon_k \widetilde{\boldsymbol{g}}_k \tag{120}$$

$$= \widetilde{\boldsymbol{\theta}}_k - \epsilon_k \left[\boldsymbol{g}_k + (\widetilde{\boldsymbol{g}}_k - \boldsymbol{g}_k) \right] \Rightarrow \frac{\Delta \widetilde{\boldsymbol{\theta}}_k}{\epsilon_k} = \frac{\widetilde{\boldsymbol{\theta}}_{k+1} - \widetilde{\boldsymbol{\theta}}_k}{\epsilon_k} = -\boldsymbol{g}_k + (\boldsymbol{g}_k - \widetilde{\boldsymbol{g}}_k) = -\boldsymbol{g}_k + \boldsymbol{\mathfrak{e}}_k , \tag{156}$$

where $\boldsymbol{g}_k = \boldsymbol{g}(\boldsymbol{\theta}_k)$ and $\widetilde{\boldsymbol{g}}_k = \widetilde{\boldsymbol{g}}_b(\boldsymbol{\theta}_k)$, with $b = k$, and $\widetilde{\boldsymbol{g}}_b(\cdot)$ is the gradient estimate function using minibatch $b = 1, \ldots, k_{max}$.

To show that the gradient error has zero mean (average), based on the linearity of the expectation function $\mathbb{E}(\cdot) = \langle \cdot \rangle$ defined in Eq. (67) (Footnote 89), i.e.,

$$\langle \alpha \boldsymbol{u} + \beta \boldsymbol{v} \rangle = \alpha \langle \boldsymbol{u} \rangle + \beta \langle \boldsymbol{u} \rangle , \tag{157}$$

$$\langle \boldsymbol{\mathfrak{e}}_k \rangle = \langle \boldsymbol{g}_k - \widetilde{\boldsymbol{g}}_k \rangle = \langle \boldsymbol{g}_k \rangle - \langle \widetilde{\boldsymbol{g}}_k \rangle = \boldsymbol{g}_k - \langle \widetilde{\boldsymbol{g}}_k \rangle , \tag{158}$$

from Eqs. (135)-(137) on the definition of minibatches and Eqs. (139)-(140) on the definition of the cost and gradient estimates (without omitting the iteration counter k), we have

$$\boldsymbol{g}_k = \boldsymbol{g}(\boldsymbol{\theta}_k) = \frac{1}{k_{max}} \sum_{b=1}^{b=k_{max}} \frac{1}{\mathsf{m}} \sum_{a=1}^{a=\mathsf{m}} \boldsymbol{g}_{i_{a,b}} \Rightarrow \langle \boldsymbol{g}_k \rangle = \boldsymbol{g}_k = \frac{1}{k_{max}} \sum_{b=1}^{b=k_{max}} \frac{1}{\mathsf{m}} \sum_{a=1}^{a=\mathsf{m}} \langle \boldsymbol{g}_{i_{a,b}} \rangle = \langle \boldsymbol{g}_{i_{a,b}} \rangle \tag{159}$$

$$\widetilde{\boldsymbol{g}}_k = \frac{1}{\mathsf{m}} \sum_{a=1}^{a=\mathsf{m} \leq \mathsf{M}} \boldsymbol{g}_{i_{a,k}} \Rightarrow \langle \widetilde{\boldsymbol{g}}_k \rangle = \frac{1}{k_{max}} \sum_{k=1}^{k=k_{max}} \langle \boldsymbol{g}_{i_{a,k}} \rangle = \frac{1}{k_{max}} \sum_{k=1}^{k=k_{max}} \langle \boldsymbol{g}_k \rangle = \boldsymbol{g}_k \Rightarrow \langle \boldsymbol{\mathfrak{e}}_k \rangle = 0 . \tag{160}$$

Or alternatively, the same result can be obtained with:

$$\boldsymbol{g}_k = \frac{1}{k_{max}} \sum_{b=1}^{b=k_{max}} \widetilde{\boldsymbol{g}}_b(\boldsymbol{\theta}_k) \Rightarrow \boldsymbol{g}_k = \langle \widetilde{\boldsymbol{g}}_b(\boldsymbol{\theta}_k) \rangle = \langle \widetilde{\boldsymbol{g}}_k(\boldsymbol{\theta}_k) \rangle = \langle \widetilde{\boldsymbol{g}}_k \rangle \Rightarrow \langle \boldsymbol{\mathfrak{e}}_k \rangle = 0 . \tag{161}$$

Next, the mean value of the "square" of the gradient error, i.e., $\langle \boldsymbol{\mathfrak{e}}^T \boldsymbol{\mathfrak{e}} \rangle$, in which we omitted the iteration counter subscript k to alleviate the notation, relies on some identities related to the covariance matrix $\langle \boldsymbol{\mathfrak{e}}, \boldsymbol{\mathfrak{e}} \rangle$. The mean of the square matrix $\mathsf{x}_i^T \mathsf{x}_j$, where $\{\mathsf{x}_i, \mathsf{x}_j\}$ are two random *row* matrices, is the sum of the product of the mean values and the covariance matrix of these matrices[163]

$$\langle \mathsf{x}_i^T \mathsf{x}_j \rangle = \langle \mathsf{x}_i \rangle^T \langle \mathsf{x}_j \rangle + \langle \mathsf{x}_i, \mathsf{x}_j \rangle , \text{ or } \langle \mathsf{x}_i, \mathsf{x}_j \rangle = \langle \mathsf{x}_i^T \mathsf{x}_j \rangle - \langle \mathsf{x}_i \rangle^T \langle \mathsf{x}_j \rangle , \tag{162}$$

where $\langle \mathsf{x}_i, \mathsf{x}_j \rangle$ is the covariance matrix of x_i and x_j, and thus the covariance operator $\langle \cdot, \cdot \rangle$ is bilinear due to the linearity of the mean (expectation) operator $\langle \cdot \rangle$ in Eq. (157):

$$\left\langle \sum_i \alpha_i \boldsymbol{u}_i, \sum_j \beta_j \boldsymbol{v}_j \right\rangle = \sum_i \alpha_i \beta_j \langle \boldsymbol{u}_i, \boldsymbol{v}_j \rangle , \forall \alpha_i, \beta_j \in \mathbb{R} \text{ and } \forall \boldsymbol{u}_i, \boldsymbol{v}_j \in \mathbb{R}^n \text{ random } . \tag{163}$$

Eq. (163) is the key relation to derive an expression for the square of the gradient error $\langle \boldsymbol{\mathfrak{e}}^T \boldsymbol{\mathfrak{e}} \rangle$, which can be rewritten as the sum of four covariance matrices upon using Eq. (162)$_1$ and either Eq. (160) or Eq. (161), i.e., $\langle \widetilde{\boldsymbol{g}}_k \rangle = \langle \boldsymbol{g}_k \rangle = \boldsymbol{g}_k$, as the four terms $\boldsymbol{g}_k^T \boldsymbol{g}_k$ cancel each other out:

$$\langle \boldsymbol{\mathfrak{e}}^T \boldsymbol{\mathfrak{e}} \rangle = \langle (\widetilde{\boldsymbol{g}} - \boldsymbol{g})^T (\widetilde{\boldsymbol{g}} - \boldsymbol{g}) \rangle = \langle \widetilde{\boldsymbol{g}}, \widetilde{\boldsymbol{g}} \rangle - \langle \widetilde{\boldsymbol{g}}, \boldsymbol{g} \rangle - \langle \boldsymbol{g}, \widetilde{\boldsymbol{g}} \rangle + \langle \boldsymbol{g}, \boldsymbol{g} \rangle , \tag{164}$$

---

[163] See, e.g., [185], p. 36, Eq. (2.8.3).

where the iteration counter k had been omitted to alleviate the notation. Moreover, to simplify the notation further, the gradient related to an example is simply denoted by $\boldsymbol{g}_a$ or $\boldsymbol{g}_b$, with $a, b = 1, \ldots, \mathsf{m}$ for a minibatch, and $a, b = 1, \ldots, \mathsf{M}$ for the full batch:

$$\widetilde{\boldsymbol{g}} = \widetilde{\boldsymbol{g}}_k = \frac{1}{\mathsf{m}} \sum_{a=1}^{\mathsf{m}} \boldsymbol{g}_{i_a,k} = \frac{1}{\mathsf{m}} \sum_{a=1}^{\mathsf{m}} \boldsymbol{g}_{i_a} = \frac{1}{\mathsf{m}} \sum_{a=1}^{\mathsf{m}} \boldsymbol{g}_a \,, \tag{165}$$

$$\boldsymbol{g} = \frac{1}{k_{max}} \sum_{k=1}^{k_{max}} \frac{1}{\mathsf{m}} \sum_{a=1}^{\mathsf{m}} \boldsymbol{g}_{i_a,k} = \frac{1}{\mathsf{M}} \sum_{b=1}^{\mathsf{M}} \boldsymbol{g}_b \,. \tag{166}$$

Now assume the covariance matrix of any pair of single-example gradients $\boldsymbol{g}_a$ and $\boldsymbol{g}_b$ depends only on the parameters $\boldsymbol{\theta}$, and is of the form:

$$\langle \boldsymbol{g}_a, \boldsymbol{g}_b \rangle = \mathfrak{C}(\boldsymbol{\theta})\delta_{ab} \,, \forall a, b \in \{1, \ldots, \mathsf{M}\} \,, \tag{167}$$

where δ_{ab} is the Kronecker delta. Using Eqs. (165)-(166) and Eq. (167) in Eq. (164), we obtain a simple expression for $\langle \boldsymbol{\mathfrak{e}}^T \boldsymbol{\mathfrak{e}} \rangle$:[164]

$$\langle \boldsymbol{\mathfrak{e}}^T \boldsymbol{\mathfrak{e}} \rangle = \left(\frac{1}{\mathsf{m}} - \frac{1}{\mathsf{M}} \right) \mathfrak{C} \,. \tag{168}$$

The authors of [164] introduced the following stochastic differential equation as a continuous counterpart of the discrete parameter update Eq. (156), as $\epsilon_k \to 0$:

$$\frac{d\boldsymbol{\theta}}{dt} = -\boldsymbol{g} + \mathfrak{n}(t) = -\frac{dJ}{d\boldsymbol{\theta}} + \mathfrak{n}(t) \,, \tag{169}$$

where $\mathfrak{n}(t)$ is the noise function, the continuous counterpart of the gradient error $\boldsymbol{\mathfrak{e}}_k := (\boldsymbol{g}_k - \widetilde{\boldsymbol{g}}_k)$. The noise $\mathfrak{n}(t)$ is assumed to be Gaussian, i.e., with zero expectation (mean) and with covariance function of the form (see Remark 6.9 on Langevin stochastic differential equation):

$$\langle \mathfrak{n}(t) \rangle = 0 \,, \text{ and } \langle \mathfrak{n}(t)\mathfrak{n}(t') \rangle = \mathscr{F}\mathfrak{C}(\boldsymbol{\theta})\delta(t - t') \,, \tag{170}$$

where $\mathbb{E}[\cdot] = \langle \cdot \rangle$ is the expectation of a function, $\mathscr{F}$ the "noise scale" or fluctuation factor, $\mathfrak{C}(\boldsymbol{\theta})$ the same gradient-error covariance matrix in Eq. (167), and $\delta(t - t')$ the Dirac delta. Integrating Eq. (169), we obtain:

$$\int_{t=0}^{t=\epsilon_k} \frac{d\boldsymbol{\theta}_k}{dt} dt = \boldsymbol{\theta}_{k+1} - \boldsymbol{\theta}_k = -\epsilon_k \boldsymbol{g}_k + \int_{t=0}^{t=\epsilon_k} \mathfrak{n}(t) dt \,, \text{ and } \left\langle \int_{t=0}^{t=\epsilon_k} \mathfrak{n}(t) dt \right\rangle = \int_{t=0}^{t=\epsilon_k} \langle \mathfrak{n}(t) \rangle dt = 0 \,. \tag{171}$$

The fluctuation factor $\mathscr{F}$ can be identified by equating the square of the error in Eq. (156) to that in Eq. (171), i.e.,

$$\epsilon^2 \langle \boldsymbol{\mathfrak{e}}^T \boldsymbol{\mathfrak{e}} \rangle = \int_{t=0}^{t=\epsilon} \int_{t'=0}^{t'=\epsilon} \langle \mathfrak{n}(t)\mathfrak{n}(t') \rangle dt' dt \Rightarrow \epsilon^2 \left(\frac{1}{\mathsf{m}} - \frac{1}{\mathsf{M}} \right) \mathfrak{C} = \epsilon \mathscr{F} \mathfrak{C} \Rightarrow \mathscr{F} = \epsilon \left(\frac{1}{\mathsf{m}} - \frac{1}{\mathsf{M}} \right) \,, \tag{172}$$

Remark 6.8. *Fluctuation factor for large training set.* For large M, our fluctuation factor $\mathscr{F}$ is roughly proportional to the ratio of the step length over the minibatch size, i.e., $\mathscr{F} \approx \epsilon/\mathsf{m}$. Thus step-length ϵ decay, or equivalenly minibatch size m increase, corresponds to a decrease in the fluctuation factor $\mathscr{F}$. On the other hand, [164] obtained their fluctuation factor $\mathscr{G}$ as[165]

$$\mathscr{G} = \epsilon \left(\frac{\mathsf{M}}{\mathsf{m}} - 1 \right) = \epsilon \mathsf{M} \left(\frac{1}{\mathsf{m}} - \frac{1}{\mathsf{M}} \right) = \mathsf{M}\mathscr{F} \tag{173}$$

---

[164] Eq. (168) possesses a simplicity elegance compared to the expression $\langle \alpha^2 \rangle = N(N/B - 1)F(\omega)$ in [164], based on a different definition of gradient, with $N \equiv \mathsf{M}$, $B \equiv \mathsf{m}$, $\omega \equiv \boldsymbol{\theta}$, but $F(\omega) \neq \mathfrak{C}(\boldsymbol{\theta})$.

[165] In [186] and [164], the fluctuation factor was expressed, in original notation, as $g = \epsilon(N/B - 1)$, where the equivalence with our notation is $g \equiv \mathscr{G}$ (fluctuation factor), $N \equiv \mathsf{M}$ (training-set size), $B \equiv \mathsf{m}$ (minibatch size).
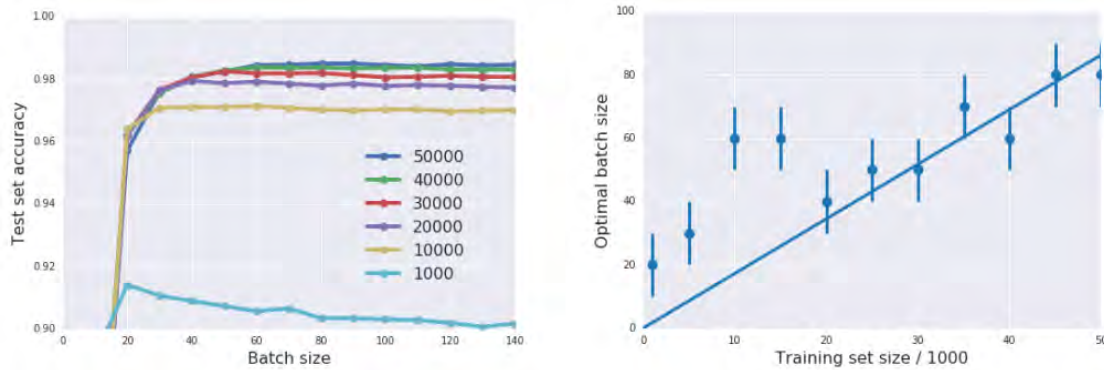
Figure 64: *Optimal minibatch size vs. training-set size* (Section 6.3.5). For a given training-set size, the smallest minibatch size that achieves the highest accuracy is optimal. Left figure: The optimal mimibatch size was moving to the right with increasing training-set size M. Right figure: The optimal minibatch size in [186] is linearly proportional to the training-set size M for large training sets (i.e., $M \to \infty$), Eq. (173), but our fluctuation factor $\mathscr{F}$ is independent of M when $M \to \infty$; see Remark 6.8. (Figure reproduced with permission of the authors.)

since their cost function was not an average, i.e., not divided by the minibatch size m, unlike our cost function in Eq. (139). When $M \to \infty$, our fluctuation factor $\mathscr{F} \to \epsilon/m$ in Eq. (172), but their fluctuation factor $\mathscr{G} \approx \epsilon M/m \to \infty$, i.e., for increasingly large M, our fluctuation factor $\mathscr{F}$ is bounded, but not their fluctuation factor $\mathscr{G}$. [186] then went on to show empirically that their fluctation factor $\mathscr{G}$ was proportional to the training-set size M for large M, as shown in Figure 64. On the other hand, our fluctuation factor $\mathscr{F}$ does not depend on the training set size M. As a result, unlike [186] in Figure 64, our optimal minibatch size would not depend of the training-set size M. ∎

It was suggested in [164] to follow the same step-length decay schedules[166] $\epsilon(\dagger)$ in Section 6.3.4 to adjust the size of the minibatches, while keeping the step length constant at its initial value ϵ_0. To demonstrate the equivalence between decreasing the step length and increasing minibatch size, the CIFAR-10 dataset with three different training schedules as shown in Figure 65 was used in [164].

The results are shown in Figure 66, where it was shown that the number of updates decreased drastically with minibatch-size increase, allowing for significantly shortening the training wall-clock time.

Remark 6.9. *Langevin stochastic differential equation, annealing.* Because the fluctuation factor $\mathscr{F}$ was proportional to the step length, and in physics, fluctuation decreases with temperature (cooling), "decaying the learning rate (step length) is simulated annealing"[168] [164]. Here, we will connect the step length to "temperature" based on the analogy of Eq. (169), the continuous counterpart of the parameter update Eq. (156). In particular, we point exact references that justify the assumptions in Eq. (170).

---

[166] See Figure 151 in Section 14.7 on "Lack of transparency and irreproducibility of results" in recent deep-learning papers.

[167] See Figure 151 in Section 14.7 on "Lack of transparency and irreproducibility of results" in recent deep-learning papers.

[168] "In metallurgy and materials science, annealing is a heat treatment that alters the physical and sometimes chemical properties of a material to increase its ductility and reduce its hardness, making it more workable. It involves heating a material above its recrystallization temperature, maintaining a suitable temperature for a suitable amount of time, and then allow slow cooling." Wikipedia, 'Annealing (metallurgy)', Version 11:06, 26 November 2019. The name "simulated annealing" came from the highly cited paper [163], which received more than 20,000 citations on Web of Science and more than 40,000 citations on Google Scholar as of 2020.01.17. See also Remark 6.10 on "Metaheuristics".
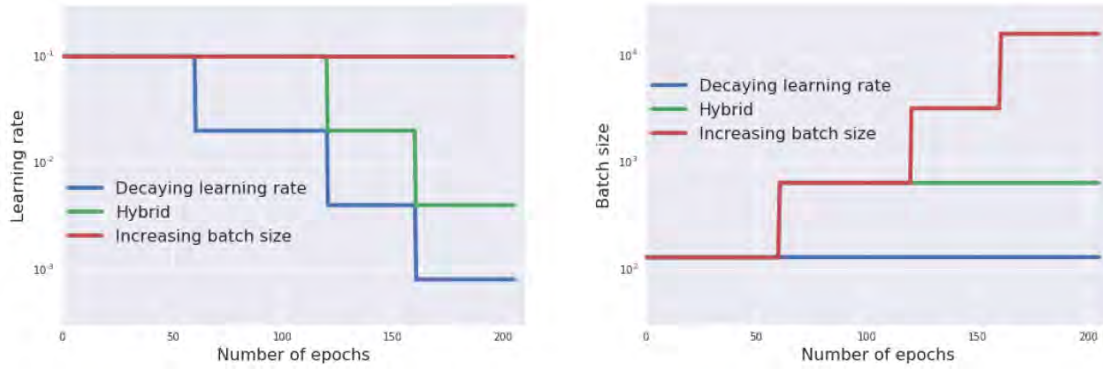
Figure 65: *Minibatch-size increase vs. step-length decay, training schedules* (Section 6.3.5). Left figure: Step length (learning rate) vs. number of epochs. Right figure: Minibatch size vs. number of epochs. Three learning-rate schedules[167] were used for training: (1) The step length was decayed by a factor of 5, from an initial value of 10^{-1}, at specific epochs (60, 120, 160), while the minibatch size was kept constant (blue line); (2) Hybrid, i.e., the step length was initially kept constant until epoch 120, then decreased by a factor of 5 at epoch 120, and by another factor of 5 at epoch 160 (green line); (3) The step length was kept constant, while the minibatch size was increased by a factor of 5, from an initial value of 128, at the same specific epochs, 60, 120, 160 (red line). See Figure 66 for the results using the CIFAR-10 dataset [164] (Figure reproduced with permission of the authors.)

Even though the authors of [164] referred to [187] for Eq. (169), the decomposition of the parameter update in [187]:

$$\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k - \epsilon_k \boldsymbol{g}_k + \sqrt{\epsilon_k} \mathfrak{v}_k \ , \ \text{with} \ \mathfrak{v}_k := \sqrt{\epsilon_k} \left[\boldsymbol{g}_k - \widetilde{\boldsymbol{g}}_k \right] = \sqrt{\epsilon_k} \mathfrak{e}_k \ , \tag{174}$$

with the intriguing factor $\sqrt{\epsilon_k}$ was consistent with the equivalent expression in [185], p. 53, Eq. (3.5.10),[169] which was obtained from the Fokker-Planck equation:

$$\boldsymbol{\theta}(t + \Delta t) = \boldsymbol{\theta}(t) + \boldsymbol{A}(\boldsymbol{\theta}(t), t) \Delta(t) + \sqrt{\Delta t} \boldsymbol{\eta}(t) \ , \tag{175}$$

where $\boldsymbol{A}(\boldsymbol{\theta}(t), t)$ is a nonlinear operator. The noise term $\sqrt{\Delta t} \boldsymbol{\eta}(t)$ is *not* related to the gradient error as in Eq. (174), and is Gaussian with zero mean and covariance matrix of the form:

$$\sqrt{\Delta t} \langle \boldsymbol{\eta}(t) \rangle = 0 \ , \ \text{and} \ \Delta t \langle \boldsymbol{\eta}(t), \boldsymbol{\eta}(t) \rangle = \Delta t \boldsymbol{B}(t) \ . \tag{176}$$

The column matrix (or vector) $\boldsymbol{A}(\boldsymbol{\theta}(t), t)$ in Eq. (175) is called the *drift vector*, and the square matrix $\boldsymbol{B}$ in Eq. (176) the *diffusion matrix*, [185], p. 52. Eq. (175) implies that $\boldsymbol{\theta}(\cdot)$ is a continuous function, called the "sample path".

To obtain a differential equation, Eq. (175) can be rewritten as

$$\frac{\boldsymbol{\theta}(t + \Delta t) - \boldsymbol{\theta}(t)}{\Delta t} = \boldsymbol{A}(\boldsymbol{\theta}(t), t) + \frac{\boldsymbol{\eta}(t)}{\sqrt{\Delta t}} \ , \tag{177}$$

which shows that the derivative of $\boldsymbol{\theta}(t)$ does not exist when taking the limit as $\Delta t \to 0$, not only due to the factor $1/\sqrt{\Delta t} \to \infty$, but also due to the noise $\boldsymbol{\eta}(t)$, [185], p. 53.

The last term $\boldsymbol{\eta}/\sqrt{\Delta t}$ in Eq. (177) corresponds to the random force X exerted on a pollen particle by the viscous fluid molecules in the 1-D equation of motion of the pollen particle, as derived by Langevin and

---

[169] In original notation used in [185], p. 53, Eq. (3.5.10) reads as $\boldsymbol{y}(t + \Delta t) = \boldsymbol{y}(t) + \boldsymbol{A}(\boldsymbol{y}(t), t) \Delta t + \boldsymbol{\eta}(t) \sqrt{\Delta t}$, in which the noise $\boldsymbol{\eta}(t)$ has zero mean, i.e., $\langle \boldsymbol{\eta} \rangle = 0$, and covariance matrix $\langle \boldsymbol{\eta}(t), \boldsymbol{\eta}(t) \rangle = \boldsymbol{B}(t)$.
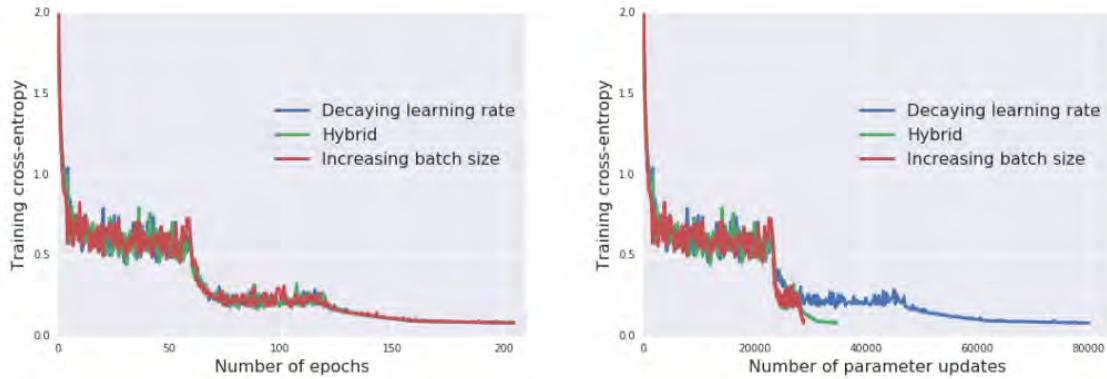
Figure 66: *Minibatch-size increase, fewer parameter updates, faster comutation* (Section 6.3.5). For each of the three training schedules in Figure 65, the same learning curve is plotted in terms of the number of epochs (left figure), and again in terms of the number of parameter updates (right figure), which shows the significant decrease in the number of parameter updates, and thus computational cost, for the training schedule with minibatch-size decrease. The blue curve ends at about 80,000 parameter updates for step-length decrease, whereas the red curve ends at about 29,000 parameter updates for minibatch-size decrease [164] (Figure reproduced with permission of the authors.)

in his original notation, [188]:[170]

$$m\frac{d^2x}{dt^2} = -6\pi\mu a\frac{dx}{dt} + X(t) \Rightarrow m\frac{dv}{dt} = -\mathfrak{f}v + X(t) \text{ , with } v = \frac{dx}{dt} \text{ and } \mathfrak{f} = 6\pi\mu a \text{ ,} \tag{178}$$

where m is the mass of the pollen particle, $x(t)$ its displacement, μ the fluid viscosity, a the particle radius, $v(t)$ the particle velocity, and $\mathfrak{f}$ the friction coefficient between the particle and the fluid. The random (noise) force $X(t)$ by the fluid molecules impacting the pollen particle is assumed to (1) be independent of the position x, (2) vary extremely rapidly compared to the change of x, (3) have zero mean as in Eq. (176). The covariance of this noise force X is proportional to the absolute temperature T, and takes the form, [189], p. 12,[171]

$$\langle X(t), X(t')\rangle = 2\mathfrak{f}kT\delta(t-t') \text{ ,} \tag{179}$$

where k denotes the Boltzmann constant.

The covariance of the noise $\mathfrak{n}(t)$ in Eq. (170) is similar to the covariance of the noise $X(t)$ in Eq. (179), and thus the fluctuation factor $\mathscr{F}$, and hence the step length ϵ in Eq. (172), can be interpreted as being proportional to temperature T. Therefore, decaying the step length ϵ, or increasing the minibatch size m, is equivalent to cooling down the temperature T, and simulating the physical annealing, and hence the name *simulated annealing* (see Remark 6.10).

---

[170] See also "Langevin equation", Wikipedia, version 17:40, 3 December 2019.

[171] For first-time learners, here a guide for further reading on a derivation of Eq. (179). It is better to follow the book [189], rather than Coffey's 1985 long review paper, cited for equation $\overline{X(t)X(t')} = 2\mathfrak{f}kT\delta(t-t')$ on p. 12, not exactly the same left-hand side as in Eq. (179), and for which the derivation appeared some 50 pages later in the book. The factor $2\mathfrak{f}kT$ was called the *spectral density*. The time average $\overline{X(t)X(t')}$ was defined, and the name *autocorrelation function* introduced on p. 13. But a particular case of the more general autocorrelation function is $\langle X(t)X(t')\rangle$ was defined on p. 59, and by the ergodic theorem, $\overline{X(t)X(t')} = \langle X(t)X(t')\rangle$ for stationary processes, p. 60, where the spectral density of $X(t)$, denoted by $\Phi_X(\omega)$ was defined as the Fourier transform of the autocorrelation function $\langle X(t)X(t')\rangle$. The derivation of Eq. (179) started from the beginning of Section 1.7, on p. 60, with the result obtained on p. 62, where the confusion of using the same notation D in $2D = 2\mathfrak{f}kT$, the spectral density, and in $D = kT/\mathfrak{f}$, the diffusion coefficient [[189], p. 20, [185], p. 7] was noted.

Eq. (178) cannot be directly integrated to obtain the velocity v in terms of the noise force X since the derivative does not exist, as interpreted in Eq. (177). Langevin went around this problem by multiplying Eq. (178) by the displacement $x(t)$ and take the average to obtain, [188]:

$$\frac{m}{2}\frac{dz}{dt} = -3\pi\mu az + \frac{RT}{N} \, , \tag{180}$$

where $z = \overline{d(x^2)}/dt$ is the time derivative of the mean square displacement, R the ideal gas constant, and N the Avogadro number. Eq. (180) can be integrated to yield an expression for z, which led to Einstein's result for Brownian motion. ∎

Remark 6.10. *Metaheuristics and nature-inspired optimization algorithms.* There is a large class of nature-inspired optimization algorithms that implemented the general conceptual *metaheuristics*—such as neighborhood search, multi-start, hill climbing, accepting negative moves, etc.—and that include many well-known methods such as Evolutionary Algorithms (EAs), Artificial Bee Colony (ABC), Firefly Algorithm, etc. [190].

The most famous of these nature-inspired algorithms would be perhaps simulated annealing in [163], which is described in [191], p. 18, as being "inspired by the annealing process of metals. It is a trajectory-based search algorithm starting with an initial guess solution at a high temperature and gradually cooling down the system. A move or new solution is accepted if it is better; otherwise, it is accepted with a probability, which makes it possible for the system to escape any local optima", i.e., the metaheuristic "accepting negative moves" mentioned in [190]. "It is then expected that if the system is cooled down slowly enough, the global optimal solution can be reached", [191], p. 18; that's step-length decay or minibatch-size increase, as mentioned above. See also Footnotes 140 and 168.

For applications of these nature-inspired algorithms, we cite the following works, without detailed review: [191] [192] [193] [194] [195] [196] [197] [198] [199]. ∎

6.3.6 Weight decay, avoiding overfit

Reducing, or decaying, the network parameters $\boldsymbol{\theta}$ (which include the weights and the biases) is one method to avoid overfitting by adding a parameter-decay term to the update equation:

$$\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k + \epsilon_k \widetilde{\boldsymbol{d}}_k - \mathfrak{d}\widetilde{\boldsymbol{\theta}}_k \, , \tag{181}$$

where $\mathfrak{d} \in (0, 1)$ is the decay parameter, and there the name "weight decay", which is equivalent to SGD with L_2 regularization, by adding an extra penalty term in the cost function; see Eq. (248) in Section 6.5.10 on the adaptive learning-rate method AdamW, where such equivalence is explained following [56]. "Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error" [78], p. 117. Weight decay is only one among other forms of regularization, such as large learning rates, small batch sizes, and dropout, [200]. The effects of the weight-decay parameter $\mathfrak{d}$ in avoiding network model overfit is shown in Figure 67.

It was written in [201] that: "In the neural network community the two most common methods to avoid overfitting are early stopping and weight decay [175]. Early stopping has the advantage of being quick, since it shortens the training time, but the disadvantage of being poorly defined and not making full use of the available data. Weight decay, on the other hand, has the advantage of being well defined, but the disadvantage of being quite time consuming" (because of tuning). For examples of tuning the weight decay parameter $\mathfrak{d}$, which is of the order of 10^{-3}, see [201] [56].

In the case of weight decay with cyclic annealing, both the step length ϵ_k and the weight decay parameter $\mathfrak{d}$ are scaled by the annealing multiplier $\mathfrak{a}_k$ in the parameter update [56]:

$$\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k + \mathfrak{a}_k(\epsilon_k \widetilde{\boldsymbol{d}}_k - \mathfrak{d}\widetilde{\boldsymbol{\theta}}_k) = \widetilde{\boldsymbol{\theta}}_k - \mathfrak{a}_k(\epsilon_k \widetilde{\boldsymbol{g}}_k + \mathfrak{d}\widetilde{\boldsymbol{\theta}}_k) \, . \tag{182}$$

The effectiveness of SGD with weight decay, with and without cyclic annealing, is presented in Figure 74.
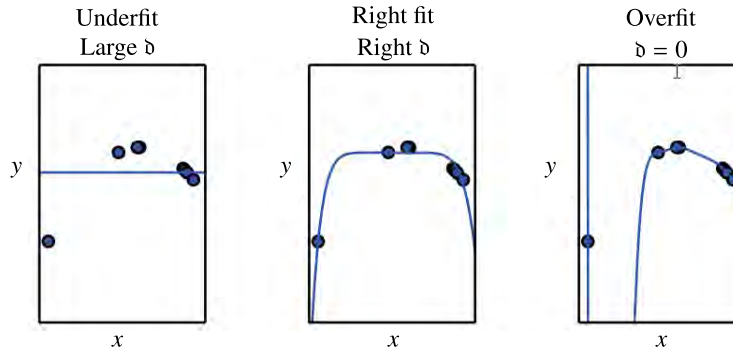
Figure 67: *Weight decay* (Section 6.3.6). Effects of magnitude of weight-decay parameter ɗ. Adapted from [78], p. 116. (Figure reproduced with permission of the authors.)

6.3.7 Combining all add-on tricks

To have a general parameter-update equation that combines all of the above add-on improvement tricks, start with the parameter update with momentum and accelerated gradient Eq. (141)

$$\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k - \epsilon_k \widetilde{\boldsymbol{g}}(\widetilde{\boldsymbol{\theta}}_k + \gamma_k(\widetilde{\boldsymbol{\theta}}_k - \widetilde{\boldsymbol{\theta}}_{k-1})) + \zeta_k(\widetilde{\boldsymbol{\theta}}_k - \widetilde{\boldsymbol{\theta}}_{k-1}) , \tag{141}$$

and add the weight-decay term $\eth\widetilde{\boldsymbol{\theta}}_k$ from Eq. (181), then scale both the weight-decay term and the gradient-descent term $(-\epsilon_k \widetilde{\boldsymbol{g}}_k)$ by the cyclic annealing multiplier $\mathfrak{a}_k$ in Eq. (182), leaving the momentum term $\zeta_k(\widetilde{\boldsymbol{\theta}}_k - \widetilde{\boldsymbol{\theta}}_{k-1})$ alone, to obtain:

$$\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k - \mathfrak{a}_k[\epsilon_k \widetilde{\boldsymbol{g}}(\widetilde{\boldsymbol{\theta}}_k + \gamma_k(\widetilde{\boldsymbol{\theta}}_k - \widetilde{\boldsymbol{\theta}}_{k-1})) + \eth_k\widetilde{\boldsymbol{\theta}}_k] + \zeta_k(\widetilde{\boldsymbol{\theta}}_k - \widetilde{\boldsymbol{\theta}}_{k-1}) , \tag{183}$$

which is included in Algorithm 4.

6.4 Kaiming He initialization

All optimization algorithms discussed above share a crucial step, which crucially affects the convergence of training, especially as neural networks become 'deep': The *initialization* of the network's parameters $\widetilde{\boldsymbol{\theta}}_0$ is not only related to the speed of convergence, it "can determine whether the algorithm converges at all."[172] Convergence problems are typically related to the scaling of initial parameters. Large parameters result in large activations, which leads to exploding values during forward and backward propagation, i.e., evaluation of the loss function and computation of its gradient with respect to the parameters. Small parameters, on the other hand, may result in *vanishing gradients*, i.e., the loss function becomes insensitive to parameters, which causes the training process to stall. To be precise, considerations regarding initialization are related to weight-matrices $\boldsymbol{W}^{(\ell)}$, see Section 4.4 on "Network layer, detailed construct"; bias vectors $\boldsymbol{b}^{(\ell)}$ are usually initialized to zero, which is also assumed in the subsequent considerations.[173]

The *Kaiming He initialization* provides equally effective as simple means to overcome scaling issues observed when weights are randomly initialized using a normal distribution with fixed standard deviation. The key idea of the authors [127] is to have the same variance of weights for each of the network's layers. As opposed to the *Xavier initialization*[174], the nonlinearity of activation functions is accounted for. Consider the ℓ-th layer of a feedforward neural network, where the vector $\boldsymbol{z}^{(\ell)}$ follows as affine function of inputs $\boldsymbol{y}^{(\ell-1)}$

$$\boldsymbol{z}^{(\ell)} = \boldsymbol{W}^{(\ell)}\boldsymbol{y}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}, \qquad \boldsymbol{y}^{(\ell)} = a(\boldsymbol{z}^{(\ell)}), \tag{184}$$

---

[172] See [78], p. 292.
[173] Situations favoring a nonzero initialization of weights are explained in [78], p. 297.
[174] See [202].

where $\boldsymbol{W}^{(\ell)}$ and $\boldsymbol{b}^{(\ell)}$ denote the layer's weight matrix and bias vector, respectively. The output of the layer $\boldsymbol{y}^{(\ell)}$ is given by element-wise application of the activation function a, see Sections 4.4.1 and 4.4.2 for a detailed presentation. All components of the weight matrix $\boldsymbol{W}^{(\ell)}$ are assumed to be independent of each other and to share the same probability distribution. The same holds for the components of the input vector $\boldsymbol{y}^{(\ell-1)}$ and the output vector $\boldsymbol{y}^{(\ell)}$. Additionally, elements of $\boldsymbol{W}^{(\ell)}$ and $\boldsymbol{y}^{(\ell)}$ shall be mutually independent. Further, it is assumed that $\boldsymbol{W}^{(\ell)}$ and $\boldsymbol{y}^{(\ell)}$ have zero mean (i.e., expectation, cf. Eq. (67)) and are symmetric around zero. In this case, the variance of $\boldsymbol{z}^{(\ell)} \in \mathbb{R}^{m_{(\ell)} \times 1}$ is given by

$$\operatorname{Var} z_i^{(\ell)} = m_{(\ell)} \operatorname{Var}\left(W_{ij}^{(\ell)} y_j^{(\ell-1)}\right) \tag{185}$$

$$= m_{(\ell)} \left(\operatorname{Var} W_{ij}^{(\ell)} \operatorname{Var} y_j^{(\ell-1)} + \mathbb{E}(W_{ij}^{(\ell)})^2 \operatorname{Var} y_j^{(\ell-1)} + \operatorname{Var} W_{ij}^{(\ell)} \mathbb{E}(y_j^{(\ell-1)})^2\right) \tag{186}$$

$$= m_{(\ell)} \operatorname{Var} W_{ij}^{(\ell)} \left(\operatorname{Var} y_j^{(\ell-1)} + \mathbb{E}(y_j^{(\ell-1)})^2\right), \tag{187}$$

where $m_{(\ell)}$ denotes the width of the ℓ-th layer and the fundamental relation $\operatorname{Var}(XY) = \operatorname{Var} X \operatorname{Var} Y + \mathbb{E}(X)^2 \operatorname{Var} Y + \mathbb{E}(Y)^2 \operatorname{Var} X$ has been used along with the assumption of weights having a zero mean, i.e., $\mathbb{E}(W_{ij}^{(\ell)}) = 0$. The variance of some random variable X, which is the expectation of the squared deviation of X from its mean, i.e.,

$$\operatorname{Var} X = \mathbb{E}((X - \mathbb{E}(X))^2), \tag{188}$$

is a measure of the "dispersion" of X around its mean value. The variance is the square of the standard deviation σ of the random variable X, or, conversely,

$$\sigma(X) = \sqrt{\operatorname{Var} X} = \sqrt{\mathbb{E}((X - \mathbb{E}(X))^2}. \tag{189}$$

As opposed to the variance, the standard deviation of some random variable X has the same physical dimension as X itself.

The elementary relation $\operatorname{Var} X^2 = \mathbb{E}(X^2) - \mathbb{E}(X)^2$ gives

$$\operatorname{Var} z_i^{(\ell)} = m_{(\ell)} \operatorname{Var} W_{ij}^{(\ell)} \mathbb{E}((y_j^{(\ell-1)})^2), \tag{190}$$

Note that the mean of inputs does not vanish for activation functions that are not symmetric about zero as, e.g., the ReLU functions (see Section 5.3.2). For the ReLU activation function, $y = a(x) = max(0, x)$ the mean value of the squared output and the variance of the input are related by

$$\mathbb{E}(y^2) = \int_{-\infty}^{\infty} y^2 P(y) \mathrm{d}y \tag{191}$$

$$= \int_{-\infty}^{\infty} \max(0, x)^2 P(x) \mathrm{d}x = \int_0^{\infty} x^2 P(x) \mathrm{d}x = \frac{1}{2} \int_{-\infty}^{\infty} x^2 P(x) \mathrm{d}x \tag{192}$$

$$= \frac{1}{2} \operatorname{Var} x. \tag{193}$$

Substituting the above result in Eq. (190) provides the following relationship among the variances of the inputs to the activation function of two consecutive layers, i.e., $\operatorname{Var} \boldsymbol{z}^{(\ell)}$ and $\operatorname{Var} \boldsymbol{z}^{(\ell-1)}$, respectively:

$$\operatorname{Var} z_i^{(\ell)} = \frac{m_{(\ell)}}{2} \operatorname{Var} W_{ij}^{(\ell)} \operatorname{Var} z_j^{(\ell-1)}. \tag{194}$$

For a network with L layers, the following relation between the variance of inputs $\operatorname{Var} \boldsymbol{z}^{(1)}$ and outputs $\operatorname{Var} \boldsymbol{z}^{(L)}$ is obtained:

$$z_i^{(L)} = \operatorname{Var} z_j^{(1)} \prod_{\ell=2}^{L} \frac{m_{(\ell)}}{2} \operatorname{Var} W_{ij}^{(\ell)}. \tag{195}$$

To preserve the variance through all layers of the network, the following condition must be fulfilled regarding the variance of weight matrices:

$$\frac{m_{(\ell)}}{2} \operatorname{Var} W_{ij}^{(\ell)} = 1 \quad \forall \ell \qquad \leftrightarrow \qquad \boldsymbol{W}^{(\ell)} \sim \mathcal{N}\left(0, \frac{2}{m_{(\ell)}}\right), \tag{196}$$

where $\mathcal{N}(0, \sigma^2)$ denotes the normal (or Gaussian) distribution with zero mean and $\sigma^2 = 2/m_{(\ell)}$ variance. The above result, which is known as *Kaiming He initialization*, implies that the width of a layer $m_{(\ell)}$ needs to be regarded in the initialization of weight matrices. Preserving the variance of inputs mitigates exploding or vanishing gradients and improves convergence in particular for deep networks. The authors of [127] provided analogous results for the parametric rectified linear unit (PReLU, see Section 5.3.3).

6.5 Adaptive methods: Adam, variants, criticism

The Adam algorithm was introduced in [170] (version 1), and updated in 2017 (version 9), and has been "immensely successful in development of several state-of-the-art solutions for a wide range of problems," as stated in [182]. "In the area of neural networks, the ADAM-Optimizer is one of the most popular adaptive step size methods. It was invented in [170]. The 5865 citations in only three years shows additionally the importance of the given paper"[175] [203]. The authors of [204] concurred: "Adam is widely used in both academia and industry. However, it is also one of the least well-understood algorithms. In recent years, some remarkable works provided us with better understanding of the algorithm, and proposed different variants of it."

6.5.1 Unified adaptive learning-rate pseudocode

It was suggested in [182] a unified pseudocode, adapted in Algorithm 5, that included not only the standard SGD in Algorithm 4, but also a number of successful adaptive learning-rate methods: AdaGrad, RMSProp, AdaDelta, Adam, the recent AMSGrad, AdamW. Our adaptation in Algorithm 5 also includes Nostalgic Adam and AdamX.[176]

Four new quantities are introduced for iteration k in SGD: (1) $\boldsymbol{m}_k$ at SGD iteration k, as the first moment, and (2) its correction $\widehat{\boldsymbol{m}}_k$

$$\boldsymbol{m}_k = \phi_k(\widetilde{\boldsymbol{g}}_1, \ldots, \widetilde{\boldsymbol{g}}_k), \quad \widehat{\boldsymbol{m}}_k = \chi_{\phi_k}(\boldsymbol{m}_k), \tag{197}$$

and (3) the second moment (variance)[177] $\boldsymbol{V}_k$ and (4) its correction $\widehat{\boldsymbol{V}}_k$

$$\boldsymbol{V}_k = \psi_k(\widetilde{\boldsymbol{g}}_1, \ldots, \widetilde{\boldsymbol{g}}_k), \quad \widehat{\boldsymbol{V}}_k = \chi_{\psi_k}(\boldsymbol{V}_k). \tag{198}$$

The descent direction estimate $\widetilde{\boldsymbol{d}}_k$ at SGD iteration k for each training epoch is

$$\widetilde{\boldsymbol{d}}_k = -\widehat{\boldsymbol{m}}_k. \tag{199}$$

The adaptive learning rate ϵ_k is obtained from rescaling the fixed learning-rate schedule $\epsilon(k)$, also called the "global" learning rate particularly when it is a constant, using the 2nd moment $\widehat{\boldsymbol{V}}_k$ as follows:

$$\boldsymbol{\epsilon}_k = \frac{\epsilon(k)}{\sqrt{\widehat{\boldsymbol{V}}_k} + \delta}, \text{ or } \boldsymbol{\epsilon}_k = \frac{\epsilon(k)}{\sqrt{\widehat{\boldsymbol{V}}_k + \delta}} \text{ (element-wise operations)} \tag{200}$$

---

[175] Reference [170] introduced the Adam algorithm in 2014, received 34,535 citations on 2019.12.11, after 5 years, and a whopping 112,797 citations on 2022.07.11, after an additional period of more than 2.5 years later, according to Google Scholar.

[176] In lines 12-13 of Algorithm 5, use Eq. (183), but replacing scalar learning rate ϵ_k with matrix learning rate $\boldsymbol{\epsilon}_k$ in Eq. (200) to update parameters; the result includes Eq. (201) for vanilla adaptive methods and Eq. (251) for AdamW.

[177] The uppercase letter $\boldsymbol{V}$ is used instead of the lowercase letter $\boldsymbol{v}$, which is usually reserved for "velocity" used in a term called "momentum", which is added to the gradient term to correct the descent direction. Such algorithm is called gradient descent with momentum in deep-learning optimization literature; see, e.g., [78], Section 8.3.2 Momentum, p. 288.

1 Unified adaptive learning-rate, momentum, weight decay, cyclic annealing (for epoch τ)
Data:
- Network parameters $\widetilde{\boldsymbol{\theta}}_0$ obtained from previous epoch $(\tau - 1)$
- Select a small stabilization parameter $\delta > 0$, e.g., 10^{-8}
- Select † as epoch τ or global iteration j, Eq. (148), and budget τ_{max} or j_{max}
- Select learning-rate schedule $\epsilon(†) \in \{$ Eqs. (147)-(151) $\}$, and parameters ϵ_0, k_c, ϖ
- Select ζ if use standard momentum, and γ if use Nesterov Accelerated Gradient, Eq. (141)
- Select a value for m the number of examples in the minibatches

Result: Updated network parameters $\widetilde{\boldsymbol{\theta}}_0$ for the next epoch $(\tau + 1)$.

2 ▶ Begin training epoch τ

3 ▶ Initialize $\widetilde{\boldsymbol{\theta}}_1 = \widetilde{\boldsymbol{\theta}}_\tau^\star$ (from previous training epoch) (line 5 in Algorithm 4)

4 ② **for** $k = 1, 2, \ldots, k_{max}$ **do**

5 | Obtain random minibatch $\mathbb{B}_k^{|\mathsf{m}|}$ containing m examples, Eq. (137) ;

6 | Compute gradient estimate $\widetilde{\boldsymbol{g}}_k = -\widetilde{\boldsymbol{g}}(\widetilde{\boldsymbol{\theta}}_k)$ per Eq. (140) ;

7 | Compute $\boldsymbol{m}_k = \phi_t(\widetilde{\boldsymbol{g}}_1, \ldots, \widetilde{\boldsymbol{g}}_k)$ and $\widehat{\boldsymbol{m}}_k = \chi_{\phi_t}(\widetilde{\boldsymbol{g}}_1, \ldots, \widetilde{\boldsymbol{g}}_k)$ as in Eq. (197) ;

8 | Compute $\boldsymbol{V}_k = \psi_t(\widetilde{\boldsymbol{g}}_1, \ldots, \widetilde{\boldsymbol{g}}_k)$ and $\widehat{\boldsymbol{V}}_k = \chi_{\psi_t}(\widetilde{\boldsymbol{g}}_1, \ldots, \widetilde{\boldsymbol{g}}_k)$ as in Eq. (198) ;

9 | Compute learning rate ϵ_k as in Eq. (200) ;

10 | Set descent direction $\widetilde{\boldsymbol{d}}_k = (-\widehat{\boldsymbol{m}}_k)$;

11 | **if** *Stopping-criterion not met (if used, Section 6.1)* **then**

12 | | ▶ Update network parameter $\widetilde{\boldsymbol{\theta}}_k$ to $\widetilde{\boldsymbol{\theta}}_{k+1}$: Use Eq. (183)

13 | | with Eq. (200), which includes Eq. (201) and Eq. (251) for AdamW ;

14 | **else**

15 | | ▶ Stopping criterion met

16 | | Reset initial minimizer-estimate for next training epoch: $\widetilde{\boldsymbol{\theta}}_0 \leftarrow \widetilde{\boldsymbol{\theta}}_k$;

17 | | Stop ② **for** loop ;

18 | **end**

19 end

Algorithm 5: *Unified adaptive learning-rate algorithm* (Section 6.5.1, 6.5.2, 6.5.4, 6.5.5, 6.5.6, 6.5.7, 6.5.10, Algorithm 4) which includes momentum, accelerated gradient, weight decay, cyclic annealing. The outer ① **for** $\tau = 1, \ldots, \tau_{max}$ loop over the training epochs, as shown in Algorithm 4 for SGD, is not presented to focus on the ② **for** $k = 1, \ldots, k_{max}$ inner loop of one pass over the training set, for a typical training epoch τ. The differences with Algorithm 4 are highlighted in purple color. For the parameter update in lines 12-13, see also Footnote 176.

where $\epsilon(k)$ can be either Eq. (147)[178] (which includes $\epsilon(k) = \epsilon_0 = $ constant) or Eq. (149);[179] δ is a small number to avoid division by zero;[180] the operations (square root, addition, division) are element-wise, with both ϵ_0 and $\delta = O(10^{-6})$ to $O(10^{-8})$ (depending on the algorithm) being constants.

Remark 6.11. A particular case is the AdaDelta algorithm, in which $\boldsymbol{m}_k$ in Eq. (197) is the second moment for the network parameter increments $\{\Delta\boldsymbol{\theta}_i, i = 1, \ldots, k\}$, and $\widehat{\boldsymbol{m}}_k$ also Eq. (197) the corrected gradient. ∎

All of the above arrays—such as $\boldsymbol{m}_k$ and $\widehat{\boldsymbol{m}}_k$ in Eq. (197), $\boldsymbol{V}_k$ and $\widehat{\boldsymbol{V}}_k$ in Eq. (198), and $\widetilde{\boldsymbol{d}}_k$ in Eq. (199)—together the resulting array $\boldsymbol{\epsilon}_k$ in Eq. (200) has the same structure as the network parameter array $\boldsymbol{\theta}$ in Eq. (31), with P_T in Eq. (34) being the total number of parameters. The update of the network parameter estimate in $\boldsymbol{\theta}$ is written as follows:

$$\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k + \boldsymbol{\epsilon}_k \odot \widetilde{\boldsymbol{d}}_k = \widetilde{\boldsymbol{\theta}}_k + \boldsymbol{\epsilon}_k \widetilde{\boldsymbol{d}}_k = \widetilde{\boldsymbol{\theta}}_k + \Delta\widetilde{\boldsymbol{\theta}}_k \text{ (element-wise operations) ,} \tag{201}$$

where the Hadamard operator $\odot$ (element-wise multiplication) is omitted to alleviate the notation.[181]

Remark 6.12. The element-wise operations in Eq. (200) and Eq. (201) would allow each parameter in array $\boldsymbol{\theta}$ to have its own learning rate, unlike in traditional deterministic optimization algorithms, such as in Algorithm 2 or even in the Standard SGD Algorithm 4, where the same learning rate is applied to all parameters in $\boldsymbol{\theta}$. ∎

It remains to define the functions ϕ_t and χ_{ϕ_t} in Eq. (197), and ψ_t and χ_{ψ_t} in Eq. (198) for each of the particular algorithms covered by the unified Algorithm 5.

SGD. To obtain Algorithm 4 as a particular case, select the following functions for Algorithm 5:

$$\phi_k(\widetilde{\boldsymbol{g}}_1, \ldots, \widetilde{\boldsymbol{g}}_k) = \widetilde{\boldsymbol{g}}_k \text{ , with } \chi_{\phi_k} = \boldsymbol{I} \text{ (Identity)} \Rightarrow \boldsymbol{m}_k = \widetilde{\boldsymbol{g}}_k = \widehat{\boldsymbol{m}}_k \text{ ,} \tag{202}$$

$$\psi_k(\widetilde{\boldsymbol{g}}_1, \ldots, \widetilde{\boldsymbol{g}}_k) = \chi_{\psi_k} = \boldsymbol{I} \text{ and } \delta = 0 \Rightarrow \widehat{\boldsymbol{V}}_k = \boldsymbol{I} \text{ and } \boldsymbol{\epsilon}_k = \epsilon(k)\boldsymbol{I} \text{ ,} \tag{203}$$

together with learning-rate schedule $\epsilon(k)$ presented in Section 6.3.4 on step-length decay and annealing. In other words, from Eqs. (199)-(201), the parameter update reduces to that of the vanilla SGD with the fixed learning-rate schedule $\epsilon(k)$, without scaling:

$$\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k + \boldsymbol{\epsilon}_k \widetilde{\boldsymbol{d}}_k = \widetilde{\boldsymbol{\theta}}_k - \boldsymbol{\epsilon}_k \widetilde{\boldsymbol{g}}_k \text{ .} \tag{204}$$

Similarly for SGD with momentum and accelerated gradient (Section 6.3.2), step-length decay and cyclic annealing (Section 6.3.4), weight decay (Section 6.3.6).

---

[178] Suggested in [78], p. 287.

[179] Suggested in [182], p. 3.

[180] AdaDelta and RMSProp used the first form of Eq. (200), with δ outside the square root, whereas AdaGrad and Adam used the second part, with δ inside the square root. AMSGrad, Nostalgic Adam, AdamX did not use δ, i.e., set $\delta = 0$.

[181] There are no symbols similar to the Hadamard operator symbol $\odot$ for other operations such as square root, addition, and division, as implied in Eq. (200), so there is no need to use the symbol $\odot$ just for multiplication.
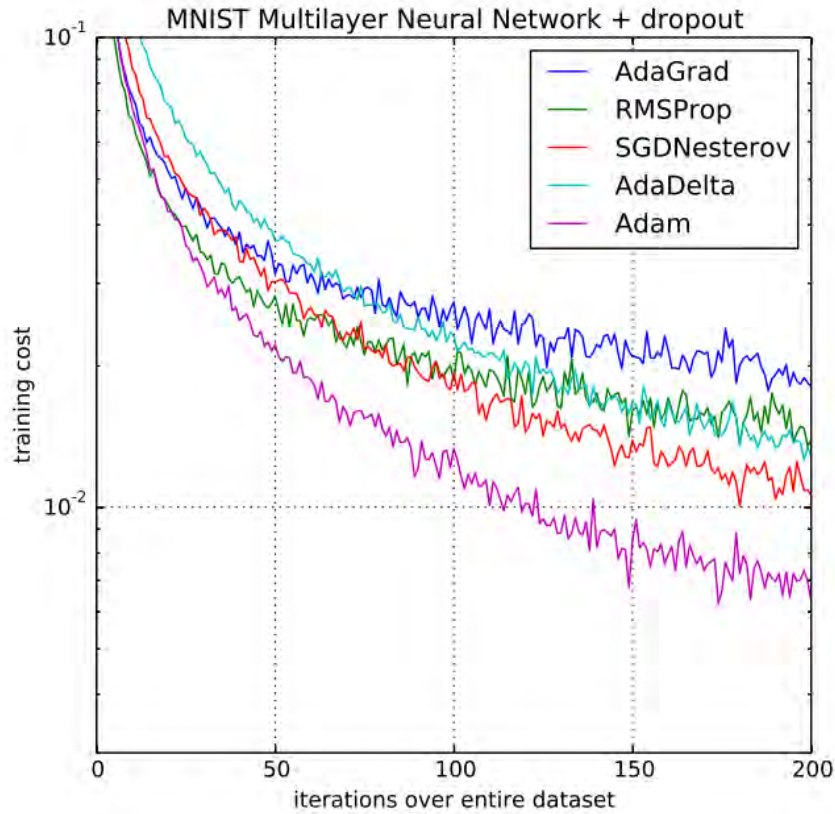
Figure 68: *Convergence of adaptive learning-rate algorithms* (Section 6.3.2): AdaGrad, RM-SProp, SGDNesterov, AdaDelta, Adam [170]. (Figure reproduced with permission of the authors.)

6.5.2 AdaGrad: Adaptive Gradient

Starting the line of research on adaptive learning-rate algorithms, the authors of [52][182] selected the following functions for Algorithm 5:

$$\phi_k(\widetilde{\boldsymbol{g}}_1, \ldots, \widetilde{\boldsymbol{g}}_k) = \widetilde{\boldsymbol{g}}_k \text{ , with } \chi_{\phi_k} = \boldsymbol{I} \text{ (Identity)} \Rightarrow \boldsymbol{m}_k = \widetilde{\boldsymbol{g}}_k = \widehat{\boldsymbol{m}}_k \text{ ,} \tag{205}$$

$$\psi_k(\widetilde{\boldsymbol{g}}_1, \ldots, \widetilde{\boldsymbol{g}}_k) = \sum_{i=1}^{k} \widetilde{\boldsymbol{g}}_i \odot \widetilde{\boldsymbol{g}}_i = \sum_{i=1}^{k} (\widetilde{\boldsymbol{g}}_i)^2 \text{ (element-wise square) ,} \tag{206}$$

$$\chi_{\psi_k} = \boldsymbol{I} \text{ and } \delta = 10^{-7} \Rightarrow \widehat{\boldsymbol{V}}_k = \sum_{i=1}^{k} (\widetilde{\boldsymbol{g}}_i)^2 \text{ and } \boldsymbol{\epsilon}_k = \frac{\epsilon(k)}{\sqrt{\widehat{\boldsymbol{V}}_k} + \delta} \text{ (element-wise operations) ,} \tag{207}$$

leading to an update with adaptive scaling of the learning rate

$$\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k - \frac{\epsilon(k)}{\sqrt{\widehat{\boldsymbol{V}}_k} + \delta} \widetilde{\boldsymbol{g}}_k \text{ (element-wise operations) ,} \tag{208}$$

in which each parameter in $\widetilde{\boldsymbol{\theta}}_k$ is updated with its own learning rate. For a given network parameter, say, θ_{pq}, its learning rate $\epsilon_{k,pq}$ is essentially $\epsilon(k)$ scaled with the inverse of the square root of the sum of all historical

---

[182] As of 2019.11.28, [52] was cited 5,385 times on Google Scholars, and 1,615 times on Web of Science. By 2022.07.11, [52] was cited 10,431 times on Google Scholars, and 3,871 times on Web of Science.

values of the corresponding gradient component (pq), i.e., $(\delta + \sum_{i=1}^{k} \widetilde{g}_{i,pq}^2)^{-1/2}$, with $\delta = 10^{-7}$ being very small. A consequence of such scaling is that a larger gradient component would have a smaller learning rate and a smaller per-iteration decrease in the learning rate, whereas a smaller gradient component would have a larger learning rate and a higher per-iteration decrease in the learning rate, even though the relative decrease is about the same.[183] Thus progress along different directions with large difference in gradient amplitudes is evened out as the number of iterations increases.[184]

Figure 68 shows the convergence of some adaptive learning-rate algorithms: AdaGrad, RMSProp, SGDNesterov, AdaDelta, Adam.

6.5.3 Forecasting time series, exponential smoothing

At this point, all of the subsequent adaptive learning-rate algorithms made use of an important technique in forecasting known as exponential smoothing of time series, without using this terminology, but instead referred to such technique as "exponential decaying average" [54], [78], p. 300, "exponentially decaying average" [80], "exponential moving average" [170], [182], [205], [162], "exponential weight decay" [56].

Figure 69: *Dow Jones Industrial Average* (DJIA, Section 6.5.3) stock index year-to-date (YTD) chart as from 2019.01.01 to 2019.11.30, Google Finance.

"Exponential smoothing methods have been around since the 1950s, and are still the most popular forecasting methods used in business and industry" such as "minute-by-minute stock prices, hourly temperatures at a weather station, daily numbers of arrivals at a medical clinic, weekly sales of a product, monthly unemployment figures for a region, quarterly imports of a country, and annual turnover of a company" [206]. See Figure 69 for the chart of a stock index showing noise.

"Exponential smoothing was proposed in the late 1950s (Brown, 1959; Holt, 1957; Winters, 1960), and has motivated some of the most successful forecasting methods. Forecasts produced using exponential smoothing methods are weighted averages of past observations, with the weights decaying exponentially as the observations get older. In other words, the more recent the observation the higher the associated weight. This framework generates reliable forecasts quickly and for a wide range of time series, which is a great

---

[183] See [54]. For example, compare the sequence $\{\frac{1}{5}, \frac{1}{5+5}\}$ to the sequence $\{\frac{1}{2}, \frac{1}{2+2}\}$. The authors of [78], p. 299, mistakenly stated "The parameters with the largest partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate."

[184] It was stated in [54]: "progress along each dimension evens out over time. This is very beneficial for training deep neural networks since the scale of the gradients in each layer is often different by several orders of magnitude, so the optimal Learning rate should take that into account." Such observation made more sense than saying "The net effect is greater progress in the more gently sloped directions of parameter space" as did the authors of [78], p. 299, who referred to AdaDelta in Section 8.5.4, p. 302, through the work of other authors, but might not read [54].

advantage and of major importance to applications in industry" [207], Chap. 7, "Exponential smoothing". See Figure 70 for an example of "exponential-smoothing" curve that is not "smooth".
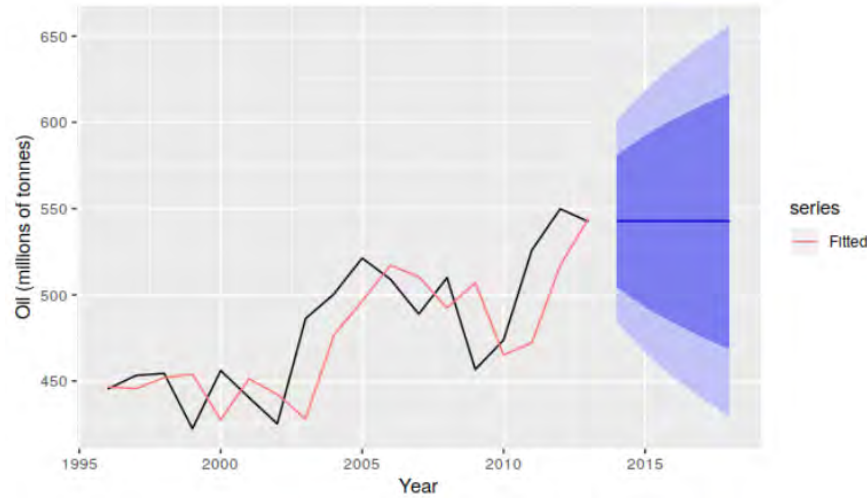
Figure 70: *Saudi Arabia oil production during 1996-2013* (Section 6.5.3). Piecewise linear data (black) and fitted curve (red), despite the name "smoothing". From [207], Chap. 7. (Figure reproduced with permission of the authors.)

For neural networks, early use of exponential smoothing dates back at least to 1998 in [165] and [166].[185]

For adaptive learning-rate algorithms further below (RMSProp, AdaDelta, Adam, etc.), let $\{x(t), t = 1, 2, \ldots\}$ be a noisy raw-data time series as in Figure 69, and $\{s(t), t = 1, 2, \ldots\}$ its smoothed-out counterpart. The following recurrence relation is an exponential smoothing used to predict $s(t + 1)$ based on the known value $s(t)$ and the data $x(t + 1)$:

$$s(t + 1) = \beta s(t) + (1 - \beta)x(t + 1) \text{ , with } \beta \in [0, 1) \tag{209}$$

Eq. (209) is a convex combination between $s(t)$ and $x(t + 1)$. a value of β closer to 1, e.g., $\beta = 0.9$ and $1 - \beta = 0.1$, would weigh the smoothed-out past data $s(t)$ more than the future raw data point $x(t + 1)$. From Eq. (209), we have

$$s(1) = \beta s(0) + (1 - \beta)x(1) \tag{210}$$

$$s(2) = \beta^2 s(0) + (1 - \beta)[\beta x(1) + x(2)] \tag{211}$$

$$\vdots$$

$$s(t) = \beta^t s(0) + (1 - \beta) \sum_{i=1}^{t} \beta^{t-i} x(i) , \tag{212}$$

where the first term in Eq. (212) is called the bias, which is set by the initial condition:

$$s(0) = x(0) \text{ , and as } t \to \infty \Rightarrow \beta^t x(0) \to 0 \text{ .} \tag{213}$$

For finite time t, if the series started with $s(0) = 0 \neq x(0)$, then there is a need to correct the estimate for the non-zero bias $x(0) \neq 0$ (see Eq. (227) and Eq. (229) in the Adam algorithm below). The coefficients in the series in Eq. (212) are exponential functions, and thus the name "exponential smoothing".

Eq. (212) is the discrete counterpart of the linear part of Volterra series in Eq. (497), used widely in neuroscientific modeling; see Remark 13.2. See also the "small heavy sphere" method or SGD with momentum Eq. (145).

---

[185] We thank Lawrence Aitchison for informing us about these references; see also [168].

It should be noted, however, that for *forecasting* (e.g., [207]), the following recursive equation, slightly different from Eq. (209), is used instead:

$$s(t + 1) = \beta s(t) + (1 - \beta)x(t) , \text{ with } \beta \in [0, 1) , \tag{214}$$

where $x(t)$ (shown in purple) is used instead of $x(t + 1)$, since if the data at $(t + 1)$ were already known, there would be no need to forecast.

6.5.4 RMSProp: Root Mean Square Propagation

Since "AdaGrad shrinks the learning rate according to the entire history of the squared gradient and may have made the learning rate too small before arriving at such a convex structure",[186] The authors of [53][187] fixed the problem of continuing decay of the learning rate by introducing RMSProp[188] with the following functions for Algorithm 5:

$$\phi_k(\widetilde{\boldsymbol{g}}_1, \ldots, \widetilde{\boldsymbol{g}}_k) = \widetilde{\boldsymbol{g}}_k , \text{ with } \chi_{\phi_k} = \boldsymbol{I} \text{ (Identity)} \Rightarrow \boldsymbol{m}_k = \widetilde{\boldsymbol{g}}_k = \widehat{\boldsymbol{m}}_k , \tag{215}$$

$$\boldsymbol{V}_k = \beta \boldsymbol{V}_{k-1} + (1 - \beta)(\widetilde{\boldsymbol{g}}_k)^2 \text{ with } \beta \in [0, 1) \text{ (element-wise square) }, \tag{216}$$

$$\boldsymbol{V}_k = (1 - \beta) \sum_{i=1}^{k} \beta^{k-i} (\widetilde{\boldsymbol{g}}_i)^2 = \psi_k(\widetilde{\boldsymbol{g}}_1, \ldots, \widetilde{\boldsymbol{g}}_k) , \tag{217}$$

$$\chi_{\psi_k} = \boldsymbol{I} \text{ and } \delta = 10^{-6} \Rightarrow \widehat{\boldsymbol{V}}_k = \boldsymbol{V}_k \text{ and } \boldsymbol{\epsilon}_k = \frac{\epsilon(k)}{\sqrt{\widehat{\boldsymbol{V}}_k + \delta}} \text{ (element-wise operations) }, \tag{218}$$

where the running average of the squared gradients is given in Eq. (216) for efficient coding, and in Eq. (217) in fully expanded form as a series with exponential coefficients β^{k-i}, for $i = 1, \ldots, k$. Eq. (216) is the exact counterpart of exponential smoothing recurrence relation in Eq. (209), and Eq. (217) has its counterpart in Eq. (212) if $\widetilde{\boldsymbol{g}}_0 = \widetilde{\boldsymbol{g}}_0^2 = 0$; see Section 6.5.3 on forecasting time series and exponential smoothing.

Figure 68 shows the convergence of some adaptive learning-rate algorithms: AdaGrad, RMSProp, SGDNesterov, AdaDelta, Adam.

RMSProp still depends on a global learning rate $\epsilon(k) = \epsilon_0$ constant, a tuning hyperparameter. Even though RMSProp was one of the go-to algorithms for machine learning, the pitfalls of RMSProp, along with other adaptive learning-rate algorithms, were revealed in [55].

6.5.5 AdaDelta: Adaptive Delta (parameter increment)

The name "AdaDelta" comes from the adaptive parameter increment $\Delta\boldsymbol{\theta}$ in Eq. (223). In parallel and independently, AdaDelta proposed in [54] not only fixed the problem of continuing decaying learning rate of AdaGrad, but also removed the need for a global learning rate ϵ_0, which RMSProp still used. By accumulating the squares of the parameter increments, i.e., $(\Delta\boldsymbol{\theta}_k)^2$, AdaDelta would fit in the unified framework of Algorithm 5 if the symbol $\boldsymbol{m}_k$ in Eq. (197) were interpreted as the accumulated 2nd moment $(\Delta\boldsymbol{\theta}_k)^2$, per Remark 6.11.

The weaknesses of AdaGrad was observed in [54]: "Since the magnitudes of gradients are factored out in AdaGrad, this method can be sensitive to initial conditions of the parameters and the corresponding

---

[186] See [78], p. 300.

[187] Almost all authors, e.g., [80] [55] [162], attributed RMSProp to Ref. [53], except for Ref. [78], where only Hinton's 2012 Coursera lecture was referred to. Tieleman was Hinton's student; see the video and the lecture notes in [53], where Tieleman's contribution was noted as unpublished. The authors of [162] indicated that both RMSProp and AdaDelta (next section) were developed independently at about the same time to fix problems in AdaGrad.

[188] Neither [162], nor [80], nor [78], p. 299, provided the meaning of the acronym RMSProp, which stands for "Root Mean Square Propagation".

gradients. If the initial gradients are large, the learning rates will be low for the remainder of training. This can be combatted by increasing the global learning rate, making the AdaGrad method sensitive to the choice of learning rate. Also, due to the continual accumulation of squared gradients in the denominator, the learning rate will continue to decrease throughout training, eventually decreasing to zero and stopping training completely."

AdaDelta was then introduced in [54] as an improvement over AdaGrad with two goals in mind: (1) to avoid the continuing decay of the learning rate, and (2) to avoid having to specify $\epsilon(k)$, called the "global learning rate", as a constant. Instead of summing past squared gradients over a finite-size window, which is not efficient in coding, exponential smoothing was employed in [54] for both the squared gradients $(\widetilde{g}_k)^2$ and for the squared increments $(\Delta\theta_k)^2$, with the increment used in the update $\theta_{k+1} = \theta_k + \Delta\theta_k$, by choosing the following functions for Algorithm 5:

$$V_k = \beta V_{k-1} + (1-\beta)(\widetilde{g}_k)^2 \text{ with } \beta \in [0,1) \text{ (element-wise square)} \tag{219}$$

$$V_k = (1-\beta)\sum_{i=1}^{k} \beta^{k-i}(\widetilde{g}_i)^2 = \psi_k(\widetilde{g}_1, \ldots, \widetilde{g}_k), \tag{220}$$

$$m_k = \beta m_{k-1} + (1-\beta)(\Delta\theta_k)^2 \text{ (element-wise square)} \tag{221}$$

$$m_k = (1-\beta)\sum_{i=1}^{k} \beta^{k-i}(\Delta\theta_i)^2. \tag{222}$$

Thus, exponential smoothing (Section 6.5.3) is used for two second-moment series: $\{(\widetilde{g}_i)^2,\ i = 1, 2, \ldots\}$ and $\{(\Delta\theta_i)^2,\ i = 1, 2, \ldots\}$. The update of the network parameters from θ_k to θ_{k+1} is carried out as follows:

$$\Delta\theta_k = -\frac{1}{\sqrt{\widehat{V}_k}+\delta}\,\widehat{m}_k\,, \text{ with } \widehat{m}_k = [\sqrt{m_{k-1}}+\delta]\,\widetilde{g}_k \text{ and } \theta_{k+1} = \theta_k + \Delta\theta_k\,, \tag{223}$$

where $\epsilon(k)$ in Eq. (218) is fixed to 1 in Eq. (223), eliminating the hyperparameter $\epsilon(k)$. Another nice feature of AdaDelta is the consistency of units (physical dimensions), in the sense that the fraction factor of the gradient $\widetilde{g}_k$ in Eq. (223) has the unit of step length (learning rate):[189]

$$\left[\frac{\sqrt{m_{k-1}}+\delta}{\sqrt{\widehat{V}_k}+\delta}\right] = \left[\frac{\Delta\theta}{\widetilde{g}}\right] = [\epsilon]\,, \tag{224}$$

where the enclosing square brackets denote units (physical dimensions), but that was not the case in Eq. (218) of RMSProp:

$$[\epsilon_k] = \left[\frac{\epsilon(k)}{\sqrt{\widehat{V}_k}+\delta}\right] \neq [\epsilon]\,. \tag{225}$$

---

[189] In spite of the nice features in AdaDelta, neither [78], nor [80], nor [162], had a review of AdaDelta, except for citing [54], even though the authors of [78], p. 302, wrote: "While the results suggest that the family of algorithms with adaptive learning rates (represented by RMSProp and AdaDelta) performed fairly robustly, no single best algorithm has emerged," and "Currently, the most popular optimization algorithms actively in use include SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta, and Adam." The authors of [80], p. 286, did not follow the historical development, briefly reviewed RMSProp, then cited in passing references for AdaDelta and Adam, and then mentioned the "popular AdaGrad algorithm" as a "member of this family"; readers would lose sight of the gradual progress made starting from AdaGrad, to RMSProp, AdaDelta, then Adam, and to the recent AdamW in [56], among others, alternating with pitfalls revealed and subsequent fixes, and then more pitfalls revealed and more fixes.

Figure 68 shows the convergence of some adaptive learning-rate algorithms: AdaGrad, RMSProp, SGDNesterov, AdaDelta, Adam.

Despite this progress, AdaDelta and RMSProp, along with other adaptive learning-rate algorithms, shared the same pitfalls as revealed in [55].

6.5.6 Adam: Adaptive moments

Both both 1st moment Eq. (226) and 2nd moment Eq. (228) are adaptive. To avoid possible large step sizes and non-convergence of RMSProp, the following functions were selected for Algorithm 5 [170]:

$$\boldsymbol{m}_k = \beta_1 \boldsymbol{m}_{k-1} + (1 - \beta_1)\widetilde{\boldsymbol{g}}_k \ , \ \text{with } \beta_1 \in [0, 1) \text{ and } \boldsymbol{m}_0 = 0 \ , \tag{226}$$

$$\widehat{\boldsymbol{m}}_k = \frac{1}{1 - (\beta_1)^k} \, \boldsymbol{m}_k \ (\text{bias correction}) \ , \tag{227}$$

$$\boldsymbol{V}_k = \beta_2 \boldsymbol{V}_{k-1} + (1 - \beta_2)(\widetilde{\boldsymbol{g}}_k)^2 \ (\text{element-wise square}) \ , \ \text{with } \beta_2 \in [0, 1) \text{ and } \boldsymbol{V}_0 = 0 \ , \tag{228}$$

$$\widehat{\boldsymbol{V}}_k = \frac{1}{1 - (\beta_2)^k} \, \boldsymbol{V}_k \ (\text{bias correction}) \ , \tag{229}$$

$$\boldsymbol{\epsilon}_k = \frac{\epsilon(k)}{\sqrt{\widehat{\boldsymbol{V}}_k + \delta}} \ (\text{element-wise operations}) \ . \tag{230}$$

with the following recommended values of the parameters:

$$\beta_1 = 0.9 \, , \beta_2 = 0.999 \, , \epsilon_0 = 0.001 \, , \delta = 10^{-8} \ . \tag{231}$$

Remark 6.13. RMSProp is a particular case of Adam, when $\beta_1 = 0$, together with the absence of the bias-corrected 1st moment Eq. (227) and bias-corrected 2nd moment Eq. (229). Moreover, the get RMSProp from Adam, choose the constant δ and the learning rate ϵ_k as in Eq. (218), instead of Eq. (230) above, but this choice is a minor point, since either choice should be fine. On the other hand, for deep-learning applications, having the 1st moment (or momentum), and thus requiring $\beta > 0$, would be useful to "significantly boost the performance" [182], and hence an advantage of Adam over RMSProp.   ∎

It follows from Eq. (212) in Section 6.5.3 on exponential smoothing of time series that the recurrence relation for gradients (1st moment) in Eq. (226) leads to the following series:

$$\boldsymbol{m}_k = (1 - \beta_1) \sum_{i=1}^{k} \beta^{k-i} \, \widetilde{\boldsymbol{g}}_i \ , \tag{232}$$

since $\boldsymbol{m}_0 = 0$. Taking the expectation, as defined in Eq. (67), on both sides of Eq. (232) yields

$$\mathbb{E}[\boldsymbol{m}_k] = (1 - \beta_1) \sum_{i=1}^{k} \beta^{k-i} \, \mathbb{E}[\widetilde{\boldsymbol{g}}_i] = \mathbb{E}[\widetilde{\boldsymbol{g}}_i] \cdot (1 - \beta_1) \sum_{i=1}^{k} \beta^{k-i} + \text{D} = \mathbb{E}[\widetilde{\boldsymbol{g}}_i] \cdot [1 - (\beta_1)^2] + \text{D} \ , \tag{233}$$

where D is the drift from the expected value, with D = 0 for stationary random processes.[190] For non-stationary processes, it was suggested in [170] to keep D small by choosing small β_1 so only past gradients close to the present iteration k would contribute, so to keep any change in the mean and standard deviation in subsequent iterations small. By dividing both sides by $[1 - (\beta_1)^2]$, the bias-corrected 1st moment $\widehat{\boldsymbol{m}}_k$ shown in Eq. (227) is obtained, showing that the expected value of $\widehat{\boldsymbol{m}}_k$ is the same as the expected value of the gradient $\widetilde{\boldsymbol{g}}_i$ plus a small number, which could be zero for stationary processes:

$$\mathbb{E}[\widehat{\boldsymbol{m}}_k] = \mathbb{E}\left[\frac{\boldsymbol{m}_k}{1 - (\beta_1)^2}\right] = \mathbb{E}[\widetilde{\boldsymbol{g}}_i] + \frac{\text{D}}{1 - (\beta_1)^2} \ . \tag{234}$$

---

[190] A random process is stationary when its mean and standard deviation stay constant over time.

The argument to obtain the bias-corrected 2nd moment $\widehat{V}_k$ in Eq. (229) is of course the same.

The authors of [170] pointed out the lack of bias correction in RMSProp (Remark 6.13), leading to "very large step sizes and often divergence", and provided numerical experiment results to support their point.

Figure 68 shows the convergence of some adaptive learning-rate algorithms: AdaGrad, RMSProp, SGDNesterov, AdaDelta, Adam. Their results show the superior performance of Adam compared to other adaptive learning-rate algorithms. See Figure 151 in Section 14.7 on "Lack of transparency and irreproducibility of results" in recent deep-learning papers.

6.5.7 AMSGrad: Adaptive Moment Smoothed Gradient

The authors of [182] stated that Adam (and other variants such as RMSProp, AdaDelta, Nadam) "failed to converge to an optimal solution (or a critical point in non-convex settings)" in many applications with large output spaces, and constructed a simple convex optimization for which Adam did not converge to the optimal solution.

An earlier version of [182] received one of the three Best Papers at the ICLR 2018[191] conference, in which it was suggested to fix the problem by endowing the mentioned algorithms with "long-term memory" of past gradients, and by selecting the following functions for Algorithm 5:

$$\boldsymbol{m}_k = \beta_{1k}\boldsymbol{m}_{k-1} + (1 - \beta_{1k})\widetilde{\boldsymbol{g}}_k = \phi_k \ , \ \text{with } \boldsymbol{m}_0 = 0 \ , \tag{235}$$

$$\beta_{1k} = \beta_{11} \in [0, 1) \ , \ \text{or } \beta_{1k} = \beta_{11}\lambda^{k-1} \text{ with } \lambda \in (0, 1) \ , \ \text{or } \beta_{1k} = \frac{\beta_{11}}{k} \ , \tag{236}$$

$$\widehat{\boldsymbol{m}}_k = \boldsymbol{m}_k \ \text{(no bias correction)} \ , \tag{237}$$

$$\boldsymbol{V}_k = \beta_2\boldsymbol{V}_{k-1} + (1 - \beta_2)(\widetilde{\boldsymbol{g}}_k)^2 = \psi_k \ \text{(element-wise square)} \ , \ \text{with } \boldsymbol{V}_0 = 0 \ , \tag{238}$$

$$\beta_2 \in [0, 1) \text{ and } \frac{\beta_{11}}{\sqrt{\beta_2}} < 1 \Leftrightarrow \beta_2 > (\beta_1)^2 \ , \tag{239}$$

$$\widehat{\boldsymbol{V}}_k = \max(\widehat{\boldsymbol{V}}_{k-1}, \boldsymbol{V}_k) \ \text{(element-wise max, "long-term memory")} \ , \ \text{and } \delta \text{ not used} \ , \tag{240}$$

The parameter λ was not defined in Corollary 1 of [182]; such omission could create some difficulty for first-time leaners. It has to be deduced from reading the Corollary that $\lambda \in (0, 1)$. For the step-length (or size) schedule $\epsilon(k)$, even though only Eq. (149) was considered in [182] for the convergence proofs, Eq. (147) (which includes $\epsilon(k) = \epsilon_0 = \text{constant}$) and Eq. (150) could also be used.[192]

First-time learners in this field could be overwhelmed by complex-looking equations in this kind of paper, so it would be helpful to elucidate some key results that led to the above expressions, particularly for β_{1k}, which can be a constant or a function of the iteration number k, in Eq. (236).

It was stated in [182] that "one typically uses a constant β_{1k} in practice (although, the proof requires a decreasing schedule for proving convergence of the algorithm)," and hence the first choice $\beta_{1k} = \beta_{11} \in (0, 1)$.

The second choice $\beta_{1k} = \beta_{11}\lambda^{k-1}$, with $\lambda \in (0, 1)$ and $\epsilon(k) = \epsilon_0/\sqrt{k}$ as in Eq. (149), was the result stated in Corollary 1 in [182], but without proof. We fill this gap here to explain this unusual expression for β_{1k}. Only the second term on the right-hand side of the inequality in Theorem 4 needs to be bounded by

---

[191] Sixth International Conference on Learning Representations (Website).
[192] The authors of [182] distinguished the step size $\epsilon(k)$ from the scaled step size ϵ_k in Eq. (147) or Eq. (149), which were called learning rate.

this choice of β_{1k}, and is written in our notation as:[193]

$$\frac{D_\infty^2}{(1-\beta_1)^2} \sum_{k=1}^{k_{max}} \sum_{i=1}^{P_T} \frac{\beta_{1k} \widehat{V}_{k,i}^{1/2}}{\epsilon(k)} \ , \tag{241}$$

where k_{max} is the maximum number of iterations in the ② **for** loop in Algorithm 5, and P_T is the total number of network parameters defined in Eq. (34). The factor $D_\infty^2/(1-\beta_1)^2$ is a constant, and the following bound on the component $\widehat{V}_{k,i}^{1/2}$ is a consequence of an assumption in Theorem 4 in [182]:

$$\| \nabla J_k \|_\infty \le G_\infty \text{ for all } k \in \{1, \ldots, k_{max}\} \Rightarrow \widehat{V}_{k,i}^{1/2} \le G_\infty \text{ for any } i \in \{1, \ldots, P_T\} \ , \tag{242}$$

where $\| \cdot \|_\infty$ is the infinity (max) norm, which is clearly consistent with the use of element-wise maximum components for "long-term memory" in Eq. (240). Intuitively, $\widehat{V}_{k,i}$ has the unit of gradient squared, and thus $\widehat{V}_{k,i}^{1/2}$ has the unit of gradient. Since G_∞ is the upperbound of the maximum component of the gradient[194] ∇J_k of the cost function J_k at any iteration k, it follows that $\widehat{V}_{k,i}^{1/2} \le G_\infty$. We refer to [183], p. 10, Lemma 4.2, for a formal proof of the inequality in Eq. (242). Once $\widehat{V}_{k,i}^{1/2}$ is replaced by G_∞^2, which is then pulled out as a common factor in expression (241), where upon substituting $\beta_{1k} = \beta_{11}\lambda^{k-1}$, with $\beta_{11} = \beta_1$, and $\epsilon(k) = \epsilon_0/\sqrt{k}$, we obtain:

$$(241) \le \frac{D_\infty^2 G_\infty}{(1-\beta_1)^2} \sum_{k=1}^{k_{max}} \sum_{i=1}^{P_T} \frac{\beta_1 \sqrt{k}\lambda^{k-1}}{\epsilon_0} = \frac{D_\infty^2 G_\infty \beta_1 P_T}{(1-\beta_1)^2 \epsilon_0} \sum_{k=1}^{k_{max}} \sqrt{k}\lambda^{k-1} \tag{243}$$

$$\le \frac{D_\infty^2 G_\infty \beta_1 P_T}{(1-\beta_1)^2 \epsilon_0} \sum_{k=1}^{k_{max}} k\lambda^{k-1} = \frac{D_\infty^2 G_\infty \beta_1 P_T}{(1-\beta_1)^2 \epsilon_0} \frac{1}{(1-\lambda)^2} \ , \tag{244}$$

where the following series expansion had been used:[195]

$$\frac{1}{(1-\lambda)^2} = \sum_{k=1}^{k_{max}} k\lambda^{k-1} \ . \tag{245}$$

Comparing the bound on the right-hand side of (244) to the corresponding bound shown in [182], Corollary 1, second term, it can be seen that two factors, P_T and ϵ_0 (in purple), were missing in the numerator and in the denominator, respectively. In addition, there should be no factor β_1 (in blue) as pointed out in [183] in their correction of the proof in [182].

On the other hand, there were some slight errors in the theorem statements and in the proofs in [182] that were corrected in [183], whose authors did a good job of not skipping any mathematical details that rendered the understanding and the verification of the proofs obscure and time consuming. It is then recommended to read [182] to get a general idea on the main convergence results of AMSGrad, then read [183] for the details, together with their variant of AMSGrad called AdamX.

The authors of [203], like those of [182], pointed out errors in the convergence proof in [170], and proposed a fix to this proof, but did not suggest any new variant of Adam.

In the two large numerical experiments on the MNIST dataset in Figure 71,[196] the authors of [182] used contant $\beta_{1k} = \beta_1 = 0.9$, with $\beta_2 \in \{0.99, 0.999\}$; they chose the step size schedule $\epsilon(k) = \epsilon_0/\sqrt{k}$ in

---

[193] To write this term in the notation used in[182], Theorem 4 and Corollary 1, simply make the following changes in notation: $k \to t$, $k_{max} \to T$, $P_T \to d$, $V \to v$, $\epsilon(k) \to \alpha_t$.

[194] The notation "G" is clearly mnemonic for "gradient", and the uppercase is used to designate upperbound.

[195] See also [183], p. 4, Lemma 2.4.

[196] See a description of the NMIST dataset in Section 5.3 on "Vanishing and exploding gradients". For the difference between logistic regression and neural network, see, e.g., [208], Raschka, "Machine Learning FAQ: What is the relation between Logistic Regression and Neural Networks and when to use which?" Original website, Internet archive. See also [78], p. 200, Figure6.8b, for the computational graph of Logistic Regression (one-layer network).
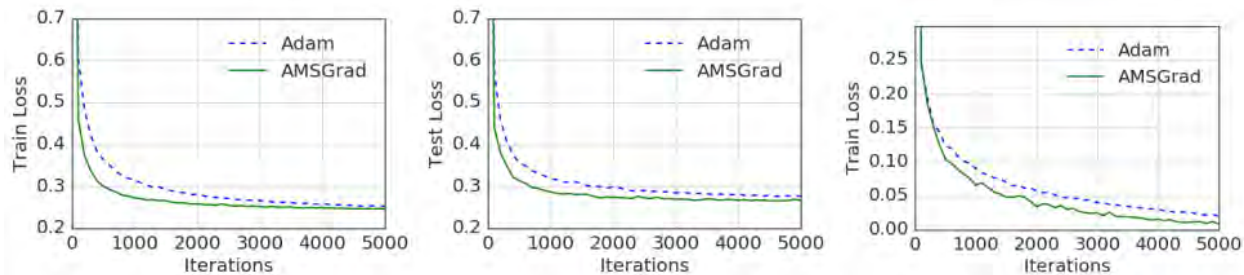
Figure 71: *AMSGrad vs Adam, numerical examples* (Sections 6.1, 6.5.7). The MNIST dataset is used. The first two figures on the left were the results of using logistic regression (network with one layer with logistic sigmoid activation function), whereas the figure on the right is by using a neural network with three layers (input layer, hidden layer, output layer). The cost function decreased faster for AMSGrad compared to that of Adam. For logistic regression, the difference between the two cost values also decreased with the iteration number, and became very small at iteration 5000. For the three-layer neural network, the cost difference between AMSGrad and Adam stayed more or less constant, as the cost went down to more than one tenth of the initial cost at about 0.3, and after 5000 iterations, the AMSGrad cost (≈ 0.01) was about 50% of the Adam cost (≈ 0.02). See [182]. (Figure reproduced with permission of the authors.)

the logistic regression experiment, and constant step size $\epsilon(k) = \epsilon_0$ in a network with three layers (input, hidden, output). There was no single set of optimal parameters, which appeared to be problem dependent.

The authors of [182] also did not provide any numerical example with $\beta_{1k} = \beta_1 \lambda^{k-1}$; such numerical examples can be found, however, in [183] in connection with AdamX below.

Unfortunately, when comparing AMSGrad to Adam and AdamW (further below), it was remarked in [209] that AMSGrad generated "a lot of noise for nothing", meaning AMSGrad did not live up to its potential and best-paper award when tested on "real-life problems".

6.5.8 AdamX and Nostalgic Adam

AdamX. The authors of [183], already mentioned above in connection to errors in the proofs in [182] for AMSGrad, also pointed out errors in the proofs by [170] (Theorem 10.5), [203] (Theorem 4.4), and by others, and suggested a fix for these proofs, and a new variant of AMSGrad called AdamX.

Reference [183] is more convenient to read, compared to [182], as the authors provided all mathematical details for the proofs, without skipping important details.

A slight change to Eq. (240) was proposed in [183] as follows:

$$\widehat{V}_1 = V_1 \text{ , and } \widehat{V}_k = \max\left(\frac{(1 - \beta_{1,k})^2}{(1 - \beta_{1,k-1})^2} \widehat{V}_{k-1}, V_k \right) \text{ for } K \geq 2 \text{ ,} \tag{246}$$

In addition, numerical examples were provided in [183] with $\beta_{1k} = \beta_1 \lambda^{k-1}$, $\beta_1 = 0.9$, $\lambda = 0.001$, $\beta_2 = 0.999$, and $\delta = 10^{-8}$ in Eq. (200), even though the pseudocode did not use δ (or set $\delta = 0$). The authors of [183] showed that both AMSGrad and AdamX converged with similar results, thus supporting their theoretical investigation, in particular, correcting the errors in the proofs of [182].

Nostalgic Adam. The authors of [204] also fixed the non-convergence of Adam by introducing "long-term memory" to the second-moment of the gradient estimates, similar to the work in [182] on AMSGrad and in [183] on AdamX.

There are many more variants of Adam. But how are Adam and its variants compared to good old SGD

with new add-on tricks ? (See the end of Section 6.3.1)
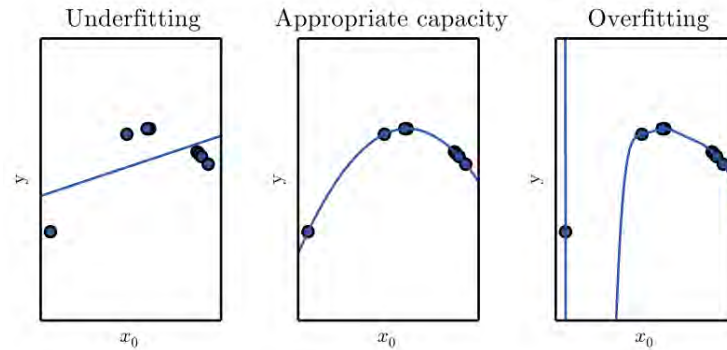
Figure 72: *Overfitting* (Section 6.5.9, 6.5.10). *Left:* Underfitting with 1st-order polynomial. Middle: Appropriate fitting with 2nd-order polynomial. *Right:* Overfitting with 9th-order polynomial. See [78], p. 110, Figure5.2. (Figure reproduced with permission of the authors.)

6.5.9 Criticism of adaptive methods, resurgence of SGD

Yet, despite the claim that RMSProp is "currently one of the go-to optimization methods being employed routinely by deep learning practitioners," and that "currently, the most popular optimization algorithms actively in use include SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta, and Adam",[197] the authors of [55] through their numerical experiments, that adaptivity can overfit (Figure 72), and that standard SGD with step-size tuning performed better than adaptive learning-rate algorithms such as AdaGrad, RMSProp, and Adam. The total number of parameters, P_T in Eq. (34), in deep networks could easily exceed 25 times the number of output targets m (Figure 23), i.e.,[198]

$$P_T \geq 25m ,\tag{247}$$

making it prone to overfit without employing special techniques such as regularization or weight decay (see AdamW below).

It was observed in [55] that adaptive methods tended to have larger generalization (test) errors[199] compared to SGD: "We observe that the solutions found by adaptive methods generalize worse (often significantly worse) than SGD, even when these solutions have better training performance," (see Figure 73), and concluded that:

> "Despite the fact that our experimental evidence demonstrates that adaptive methods are not advantageous for machine learning, the Adam algorithm remains incredibly popular. We are not sure exactly as to why, but hope that our step-size tuning suggestions make it easier for practitioners to use standard stochastic gradient methods in their research."

The work of [55] has encouraged researchers who were enthusiastic with adaptive methods to take a fresh look at SGD again to tease something more out of this classic method.[200]

---

[197] See [78], pp. 301-302.

[198] See [55], p. 4, Section 3.3 "Adaptivity can overfit".

[199] See [78], p. 107, regarding training error and test (generalization) error. "The ability to perform well on previously unobserved inputs is called generalization."

[200] See, e.g., [210], where [55] was not referred to directly, but through a reference to [164], in which there was a reference to [55].

(a) CIFAR-10 (Train)   **(b)** CIFAR-10 (Test)

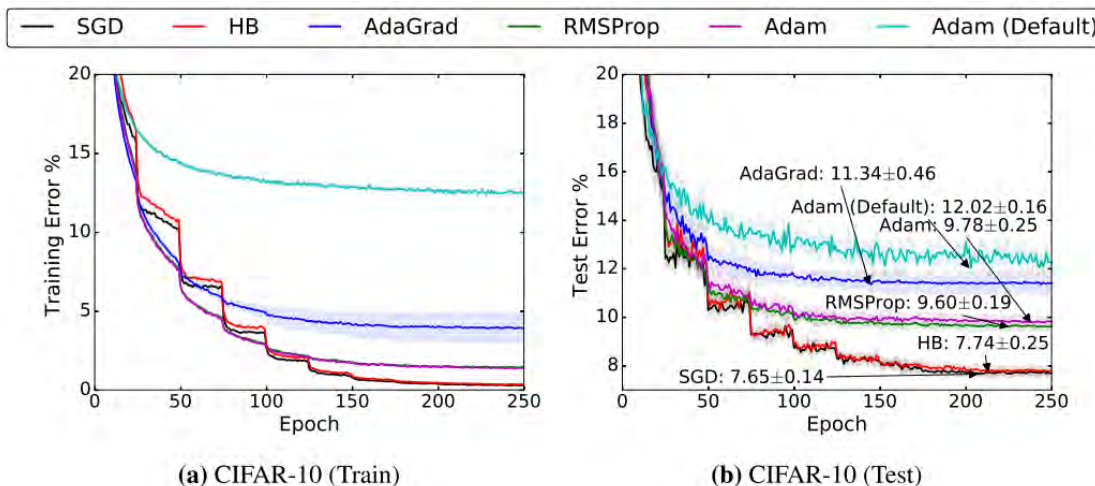**Figure 73:** *Standard SGD and SGD with momentum vs AdaGrad, RMSProp, Adam on CIFAR-10 dataset* (Sections 6.1, 6.3.2, 6.5.9). From [55], where a method for step-size tuning and step-size decaying was proposed to achieve lowest training error and generalization (test) error for both Standard SGD and SGD with momentum ("Heavy Ball" or better yet "Small Heavy Sphere" method) compared to adaptive methods such as AdaGrad, RMSProp, Adam. (Figure reproduced with permission of the authors.)

6.5.10 AdamW: Adaptive moment with weight decay

The authors of [56], aware of the work in [55], wrote: It was suggested in [55] "that adaptive gradient methods do not generalize as well as SGD with momentum when tested on a diverse set of deep learning tasks, such as image classification, character-level language modeling and constituency parsing." In particular, it was shown in [56] that " a major factor of the poor generalization of the most popular adaptive gradient method, Adam, is due to the fact that L_2 regularization is not nearly as effective for it as for SGD," and proposed to move the weight decay from the gradient (L_2 regularization) to the parameter update (original weight decay regularization). So what is "L_2 regularization" and what is "weight decay" ? (See also Section 6.3.6 on weight decay.)

Briefly, "L_2 regularization" is aiming at decreasing the overall weights to avoid overfitting, which simply means that the network model tries to fit through as many data points as possible, including noise.[201]

$$\widetilde{J}_{reg}(\widetilde{\boldsymbol{\theta}}) = \widetilde{J}(\widetilde{\boldsymbol{\theta}}) + \frac{\mathfrak{d}}{2}(\| \widetilde{\boldsymbol{\theta}} \|_2)^2 \tag{248}$$

The magnitude of the coefficient $\mathfrak{d}$ regulates (or regularizes) the behavior of the network: $\mathfrak{d} = 0$ would lead to overfitting (Figure 72 right), a moderate $\mathfrak{d}$ may yield appropriate fitting (Figure 72 middle), a large $\mathfrak{d}$ may lead to underfitting (Figure 72 left). The gradient of the regularized cost $\widetilde{J}_{reg}$ is then

$$\widetilde{\boldsymbol{g}}_{reg} := \frac{\partial \widetilde{J}_{reg}}{\partial \widetilde{\boldsymbol{\theta}}} = \frac{\partial \widetilde{J}}{\partial \widetilde{\boldsymbol{\theta}}} + \mathfrak{d}\widetilde{\boldsymbol{\theta}} = \widetilde{\boldsymbol{g}} + \mathfrak{d}\widetilde{\boldsymbol{\theta}} \tag{249}$$

and the update becomes:

$$\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k - \epsilon_k(\widetilde{\boldsymbol{g}}_k + \mathfrak{d}\widetilde{\boldsymbol{\theta}}_k) = (1 - \epsilon_k\mathfrak{d})\widetilde{\boldsymbol{\theta}}_k - \epsilon_k\widetilde{\boldsymbol{g}}_k , \tag{250}$$

---

[201] See [78], p. 107, Section 5.2 on "Capacity, overfitting and underfitting", and p. 115 provides a good explanation and motivation for regularization, as in Gupta 2017, 'Deep Learning: Overfitting', 2017.02.12, Original website, Internet archive.
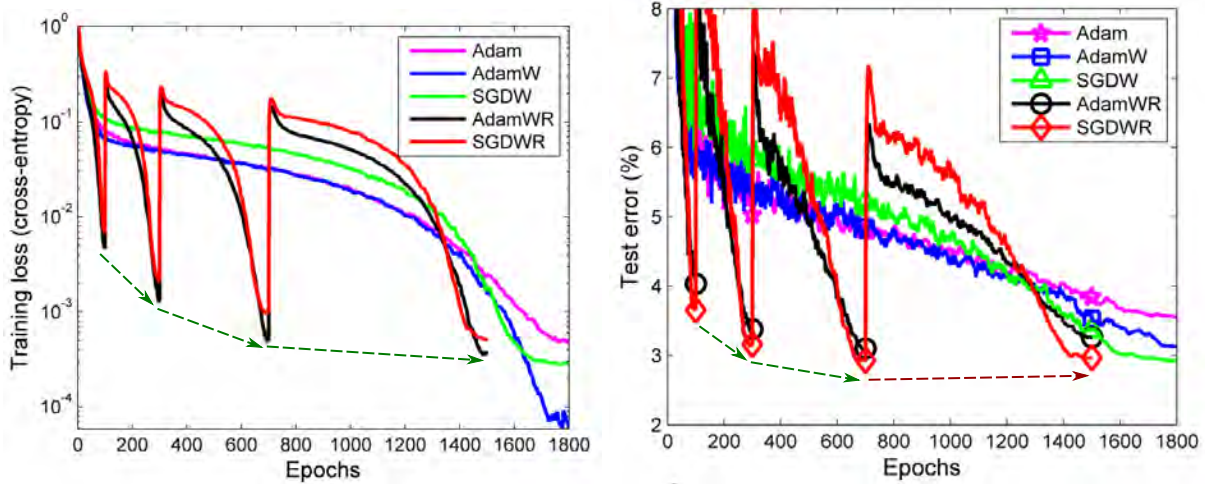
Figure 74: *AdamW vs Adam, SGD, and variants on CIFAR-10 dataset* (Sections 6.1, 6.5.10). While AdamW achieved lowest training loss (error) after 1800 epochs, the results showed that SGD with weight decay (SGDW) and with warm restart (SGDWR) achieved lower test (generalization) errors than Adam, AdamW, AdamWR. See Figure 75 for the scheduling of the annealing multiplier $\mathfrak{a}_k$, for which the epoch numbers $(100, 300, 700, 1500)$ for complete cooling ($\mathfrak{a}_k = 0$) coincided with the same epoch numbers for the sharp minima. There was, however, a diminishing return beyond the 4th cycle as indicated by the dotted arrows, for both training error and test error, which actually increased at the end of the 4th cycle (right subfigure, red arrow), see Section 6.1 on early-stopping criteria and Remark 6.14. Adapted from [56]. (Figure reproduced with permission of the authors.)

which is equivalent to decaying the parameters (including the weights in) $\widetilde{\boldsymbol{\theta}}_k$ when $(1 - \epsilon_k \mathfrak{d}) \in (0, 1)$, but with varying decay parameter $\mathfrak{d}_k = \epsilon_k \mathfrak{d} \in (0, 1)$ depending on the step length ϵ_k, which itself would decrease toward zero.

The same equivalence between L_2 regularization Eq. (248) and weight decay Eq. (250), which is linear with respect to the gradient, cannot be said for adaptive methods due to the nonlinearity with respect to the gradient in the update procedure using Eq. (200) and Eq. (201). See also the parameter update in lines 12-13 of the unified pseudocode for adaptive methods in Algorithm 5 and Footnote 176. Thus, it was proposed in [56] to explicitly add weight decay to the parameter update Eq. (201) for AdamW (lines 12-13 in Algorithm 5) as follows:

$$\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k + \mathfrak{a}_k(\epsilon_k \widetilde{\boldsymbol{d}}_k - \mathfrak{d}\widetilde{\boldsymbol{\theta}}_k) \text{ (element-wise operations) ,} \tag{251}$$

where the parameter $\mathfrak{a}_k \in [0, 1]$ is an annealing multiplier defined in Section 6.3.4 on weight decay:

$$\mathfrak{a}_k = 0.5 + 0.5\cos(\pi T_{cur}/T_p) \in [0, 1] \text{ , with } T_{cur} := j - \sum_{q=1}^{q=p-1} T_q \text{ ,} \tag{154}$$

The results of the numerical experiments on the CIFAR-10 dataset using Adam, AdamW, SGDW (Weight decay), AdamWR (Warm Restart), and SGDWR were reported in Figure 74 [56].

Remark 6.14. *Limitation of cyclic annealing.* The 5th cycle of annealing not shown in Figure 74 would end at epoch $1500 + 1600 = 3100$, which is well beyond epoch budget of 1800. In view of the diminishing return in the decrease of the training error at the end of each cycle, in addition to an increase in the test error by the end of the 4th cycle, as shown in Figure 74, it is unlikely that it is worthwhile to start the 5th cycle, since not only the computation would be more expensive, due to the warm restart, the increase in the test
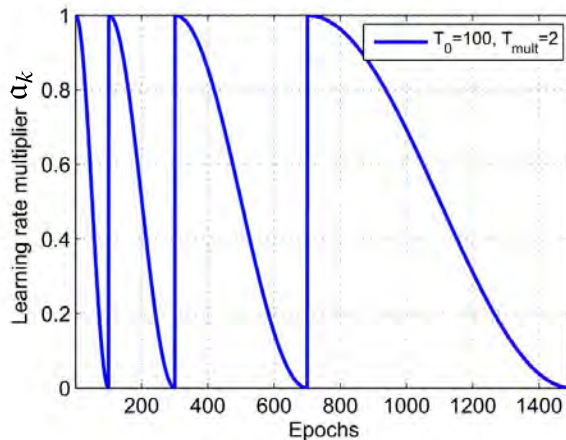
Figure 75: *Cosine annealing* (Sections 6.3.4, 6.5.10). Annealing factor $\mathfrak{a}_k$ as a function of epoch number. Four annealing cycles $p = 1, \ldots, 4$, with the following schedule for T_p in Eq. (154): (1) Cycle 1, $T_1 = 100$ epochs, epoch 0 to epoch 100, (2) Cycle 2, $T_2 = 200$ epochs, epoch 101 to epoch 300, (3) Cycle 3, $T_3 = 400$ epochs, epoch 301 to epoch 700, (4) Cycle 4, $T_4 = 800$ epochs, epoch 701 to epoch 1500. From [56]. See Figure 74 in which the curves for AdamWR and SGDWR ended at epoch 1500. (Figure reproduced with permission of the authors.)

error indicated that the end of the 3rd cycle was optimal, and thus the reason for [56] to stop at the end of the 4th cycle. ∎

The results for test errors in Figure 74 appeared to confirm the criticism in [55] that adaptive methods brought about "marginal value" compared to the classic SGD. Such observation was also in agreement with [168], where it was stated:

> "In our experiments, either AdaBayes or AdaBayes-SS outperformed other adaptive methods, including AdamW (Loshchilov & Hutter, 2017), and Ada/AMSBound (Luo et al., 2019), though SGD frequently outperformed all adaptive methods." (See Figure 76 and Figure 77)

If AMSGrad generated "a lot of noise for nothing" compared to Adam and AdamW, according to [209], then does "marginal value" mean that adaptive methods in general generated a lot of noise for not much, compared to SGD ?

The work in [55], [168], and [56] proved once more that a classic like SGD introduced by Robbins & Monro (1951b) [49] never dies, and would be motivation to generalize classical deterministic first and second-order optimization methods together with line search methods to add stochasticity. We will review in detail two papers along this line: [144] and [145].

6.6 SGD with Armijo line search and adaptive minibatch

In parallel to the deterministic choice of step length based on Armijo's rule in Eq. (125) and Eq. (126), we have the following respective stochastic version proposed by [144]:[202]

$$\epsilon(\boldsymbol{\theta}) = \min_{j} \{ \beta^j \rho \mid \widetilde{J}(\boldsymbol{\theta} + \beta^j \, \widetilde{\boldsymbol{d}}) - \widetilde{J}(\boldsymbol{\theta}) \le \alpha \beta^j \rho \, \widetilde{\boldsymbol{g}} \bullet \widetilde{\boldsymbol{d}} \} \, , \tag{252}$$

$$\widetilde{J}(\boldsymbol{\theta} + \epsilon \widetilde{\boldsymbol{d}}) \le \widetilde{J}(\boldsymbol{\theta}) + \alpha \epsilon \, \widetilde{\boldsymbol{g}} \bullet \widetilde{\boldsymbol{d}} \, , \tag{253}$$

---

[202] It is not until Section 4.7 in [144] that this version is presented for the general descent of nonconvex case, whereas the pseudocode in their Algorithm 1 at the beginning of their paper, and referred to in Section 4.5, was restricted to steepest descent for convex case.
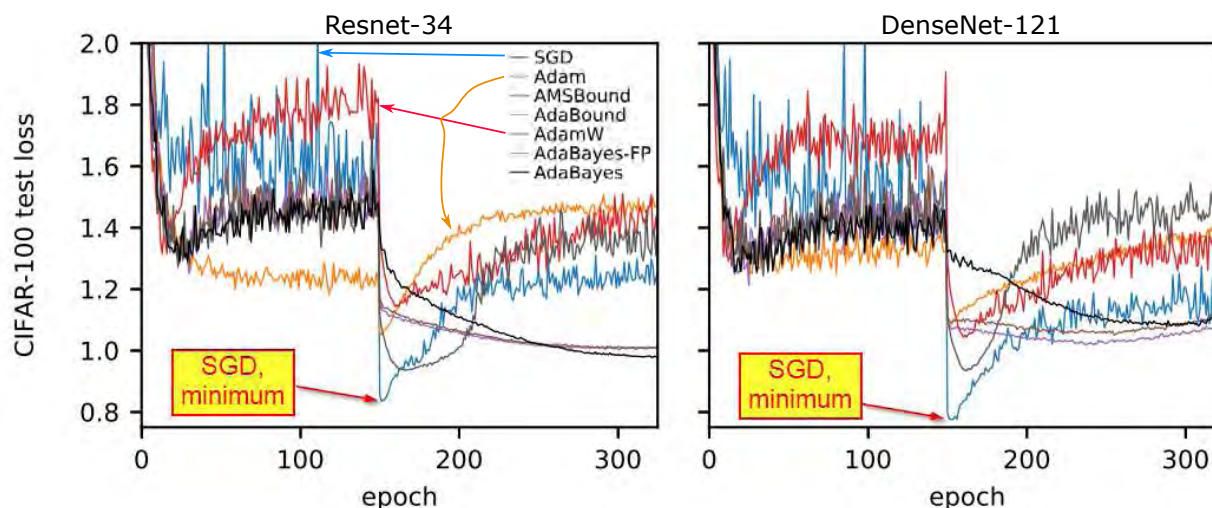
Figure 76: *CIFAR-100 test loss using Resnet-34 and DenseNet-121* (Section 6.5.10). Comparison between various optimizers, including Adam and AdamW, showing that SGD achieved the lowest global minimum loss (blue line) compared to all adaptive methods tested as shown [168]. See also Figure 77 and Section 6.1 on early-stopping criteria. (Figure reproduced with permission of the authors.)

| | CIFAR-10 | | | | CIFAR-100 | | | |
| | ResNet | | DenseNet | | ResNet | | DenseNet | |
optimizer	error (%)	loss	error (%)	loss	error (%)	loss	error (%)	loss
SGD	5.170	0.174	5.580	0.177	22.710	0.833	21.290	0.774
Adam	7.110	0.239	6.690	0.230	27.590	1.049	26.640	1.074
AdaGrad	6.840	0.307	7.490	0.338	30.350	1.347	30.110	1.319
AMSGrad	6.720	0.239	6.170	0.234	27.430	1.033	25.850	1.103
AdaBound	5.140	0.220	4.850	0.210	23.060	1.004	22.210	1.050
AMSBound	4.940	0.210	4.960	0.219	23.000	1.003	22.360	1.017
AdamW	5.080	0.239	5.190	0.214	24.850	1.142	23.480	1.043
AdaBayes-SS	5.230	**0.187**	4.910	**0.176**	23.120	**0.935**	22.600	**0.934**
AdaBayes	**4.840**	0.229	**4.560**	0.222	**22.920**	0.969	**22.090**	1.079

Figure 77: *SGD frequently outperformed all adaptive methods* (Section 6.5.10). The table contains the global minimum for each optimizer, for each of the two datasets CIFAR-10 and CIFAR-100, using two different networks. For each network, an error percentage and the loss (cost) were given. Shown in red are the lowest global minima obtained by SGD in the corresponding columns. Even in the three columns in which SGD results were not the lowest, two SGD results were just slightly above those of AdamW (1st and 3rd columns), and one even smaller (4th column). SGD clearly beat Adam, AdaGrad, AMSGrad [168]. See also Figure 76. (Figure reproduced with permission of the authors.)

1 SGD with Armijo line-search and adaptive minibatch (for training epoch τ)

Data:

• Parameter estimate $\widetilde{\boldsymbol{\theta}}_\tau^\star$ from previous epoch $(\tau - 1)$ (line 5 in Algorithm 4)

• Select 3 Armijo parameters $\alpha \in (0, 1)$, $\beta \in (0, 1)$, $\rho > 0$, and reliability bound $\delta_1 = \delta > 0$

• Select probability $p_f \in (0, 1]$ for cost estimate $\widetilde{J}$ to be close to true cost J

• Select probability $p_g \in (0, 1]$ for gradient estimate $\widetilde{g}$ to be close to true gradient $\boldsymbol{g}$

Result: Parameter estimate for next epoch $(\tau + 1)$: $\widetilde{\boldsymbol{\theta}}_{\tau+1}^\star$.

2 Define cost-estimate function $\widetilde{J}(\widetilde{\boldsymbol{\theta}}, \epsilon, p_J)$ using adaptive minibatch, Eq. (254) ;

3 Define gradient-estimate function $\widetilde{g}(\widetilde{\boldsymbol{\theta}}, \epsilon, p_g)$ using adaptive minibatch, Eq. (254) ;

4 ▶ Begin training epoch τ (see Algorithm 4 for standard SGD)

5 Initialize parameter estimate $\widetilde{\boldsymbol{\theta}}_1 = \widetilde{\boldsymbol{\theta}}_\tau^\star$ and step length $\epsilon_1 = \rho$;

6 ② ③ **for** $k = 1, 2, \ldots$ **do**

7   Compute gradient estimate $\widetilde{g}_k = \widetilde{g}(\widetilde{\boldsymbol{\theta}}_k, \epsilon_k, p_g)$ with adaptive minibatch ;

8   Set descent-direction estimate $\widetilde{\boldsymbol{d}}_k = -\widetilde{g}_k$ as in Eq. (256) ;

9   Compute cost estimates $\widetilde{J}(\widetilde{\boldsymbol{\theta}}_k, \epsilon_k, p_J)$ and $\widetilde{J}(\widetilde{\boldsymbol{\theta}}_k + \epsilon_k \widetilde{\boldsymbol{d}}_k, \epsilon_k, p_J)$ with adaptive minibatch ;

10   **if** *Stopping criterion not satisfied* **then**

11   ▶ Compute step length (learning rate) ϵ_k using stochastic Armijo's rule.

12   **if** *Stochastic Armijo decrease condition Eq. (253) not satisfied* **then**

13   ▶ Decrease step length: Set $\epsilon_k \leftarrow \beta \epsilon_k$;

14   **else**

15   ▶ Stochastic Armijo decrease condition Eq. (253) satisfied.

16   ▶ Update network parameter $\widetilde{\boldsymbol{\theta}}_k$ to $\widetilde{\boldsymbol{\theta}}_{k+1}$: Set $\widetilde{\boldsymbol{\theta}}_{k+1} \leftarrow \widetilde{\boldsymbol{\theta}}_k + \epsilon_k \boldsymbol{d}_k$;

17   ▶ Increase the next step length ϵ_{k+1}: Set $\epsilon_{k+1} = \max\{\rho, (\epsilon_k/\beta)\}$;

18   ▶ Check for reliability of step length

19   **if** *Step length reliable,* $\epsilon_k \parallel \widetilde{g} \parallel^2 \geq \delta_k^2$ **then**

20   ▶ Increase reliability bound: Set $\delta_{k+1}^2 \leftarrow \delta_k^2/\beta$;

21   **else**

22   ▶ Decrease reliability bound: Set $\delta_{k+1}^2 \leftarrow \beta \delta_k^2$;

23   **end**

24   **end**

25   **else**

26   ▶ Stopping criterion satisfied;

27   ▶ Update parameter estimate for next epoch $(\tau + 1)$: $\widetilde{\boldsymbol{\theta}}_{\tau+1}^\star \leftarrow \widetilde{\boldsymbol{\theta}}_k$

28   ▶ Stop ② ③ **for** loop (end minimization for training epoch τ).

29   **end**

30   ▶ Continue to next iteration $(k + 1)$: Set $k \leftarrow k + 1$;

31 **end**

Algorithm 6: *SGD with Armijo line search and adaptive minibatch* (Section 6.3.2, 6.6, 6.7, Algorithms 2, 4, 7). From [144]. The differences compared to the deterministic gradient descent with Armijo line search in Algorithm 2, including the ② ③ **for** loop, are highlighted in purple; See Remark 6.15. For the related stochastic methods with fixed minibatch, see the standard SGD Algorithm 4 and the Newton descent with Armijo-like line search Algorithm 7. Table 4 shows a comparison of the notations used by several authors in relation to Armijo line search.

where the overhead tilde of a quantity designates an estimate of that quantity based on a randomly selected minibatch, i.e., $\widetilde{J}$ is the cost estimate, $\widetilde{\boldsymbol{d}}$ the descent-direction estimate, and $\widetilde{\boldsymbol{g}}$ the gradient estimate, similar to those in Algorithm 4.

There is a difference though: The standard SGD in Algorithm 4 uses a fixed minibatch for the computation of the cost estimate and the gradient estimate, whereas Algorithm 6 in [144] uses adaptive subprocedure to adjust the size of the minibatches to achieve a desired (fixed) probability p_J that the cost estimate is close to the true cost, and a desired probability p_g that the gradient estimate is close to the true gradient. These adaptive-minibatch subprocedures are also functions of the learning rate (step length) ϵ, conceptually written as:

$$\widetilde{J} = \widetilde{J}(\boldsymbol{\theta}, \epsilon, p_J) \text{ and } \widetilde{\boldsymbol{g}} = \widetilde{\boldsymbol{g}}(\boldsymbol{\theta}, \epsilon, p_g) \,, \tag{254}$$

which are the counterparts to the fixed-minibatch procedures in Eq. (139) and Eq. (140), respectively.

Remark 6.15. Since the appropriate size of the minibatch depends on the gradient estimate, which is not known and which is computed based on the minibatch itself, the adaptive-minibatch subprocedures for cost estimate $\widetilde{J}$ and for gradient estimate $\widetilde{\boldsymbol{g}}$ in Eq. (254) contain a loop, started by guessing the gradient estimate, to gradually increase the size of the minibatch by adding more samples until certain criteria are met.[203]

In addition, since both the cost estimate $\widetilde{J}$ and the gradient estimate $\widetilde{\boldsymbol{g}}$ depend on the step size ϵ, the Armijo line-search loop to determined the step length ϵ—denoted as the ② for loop in the deterministic Algorithm 2—is combined with the iteration loop k in Algorithm 6, where these two combined loops are denoted as the ② ③ **for** loop. ∎

The same relationship between $\widetilde{\boldsymbol{d}}$ and $\widetilde{\boldsymbol{g}}$ as in Eq. (121) holds:

$$\widetilde{\boldsymbol{g}} \boldsymbol{\cdot} \widetilde{\boldsymbol{d}} = \sum_i \sum_j \widetilde{g}_{ij} \widetilde{d}_{ij} < 0. \tag{255}$$

For SGD, the descent-direction estimate $\widetilde{\boldsymbol{d}}$ is identified with the steepest-descent direction estimate $(-\widetilde{\boldsymbol{g}})$:

$$\widetilde{\boldsymbol{d}} = -\widetilde{\boldsymbol{g}} \,, \tag{256}$$

For Newton-type algorithms, such as in [145] [146], the descent direction estimate $\widetilde{\boldsymbol{d}}$ is set to equal to the Hessian estimate $\widetilde{\boldsymbol{H}}$ multiplied by the steepest-descent direction estimate $(-\widetilde{\boldsymbol{g}})$:

$$\widetilde{\boldsymbol{d}} = \widetilde{\boldsymbol{H}} \cdot (-\widetilde{\boldsymbol{g}}) \,, \tag{257}$$

Remark 6.16. In the SGD with Armijo line search and adaptive minibatch Algorithm 6, the reliability parameter δ_k and its use is another difference between Algorithm 6 and Algorithm 2, the deterministic gradient descent with Armijo line search, and similarly for Algorithm 4, the standard SGD. The reason was provided in [144]: Even when the probability of gradient estimate and cost estimate is near 1, it is not guaranteed that the expected value of the cost at the next iterate $\widetilde{\boldsymbol{\theta}}_{k+1}$ would be below the cost at the current iterate $\widetilde{\boldsymbol{\theta}}_k$, due to arbitrary increase of the cost. "Since random gradient may not be representative of the true gradient the function estimate accuracy and thus the expected improvement needs to be controlled by a different quantity," δ_k^2. ∎

The authors of [144] provided a rigorous convergence analysis of their proposed Algorithm 6, but had not implemented their method, and thus had no numerical results at the time of this writing.[204] Without empirical evidence that the algorithm works and is competitive compared to SGD (see adaptive methods and their criticism in Section 6.5.9), there would be no adoption.

---

[203] See [144], p. 7, below Eq. (2.4).
[204] Based on our private communications with the authors of [144] on 2019.11.16.

Table 4: *Armijo parameters* (Section 6.2.3, Algorithms 2, 6, 7). Comparing our notations here as used in Eq. (125) and Eq. (126) to those of other authors: [139] [144] [145] [151]. In [151] [139] [144], the parameters are for the first-order term $(-g) \cdot d = \| d \|^2$ (for steepest descent, $d = -g = -\partial J/\partial \theta$) in the Taylor series expansion of $\Delta J = J(\theta + \epsilon d) - J(\theta)$. In [145], the parameters are for the second-order term in the Taylor series expansion, leading to the cube of the norm of the descent direction, $\| d \|^3$. For the stochastic optimization algorithms presented in this paper, the 4th parameter δ is introduced to represent the reliability parameter δ_0 of [144] (Algorithm 6) and the stability parameter ϵ of [145] (Algorithm 7). Deterministic algorithms in [151] and [139] do not have this 4th parameter.

Parameter type	This paper	Polak [139]	Paquette [144]	Bergou [145]	Armijo [151]
Fixed factor	α	α	θ	$\frac{1}{6}$	$\frac{1}{2}$
Varying-power factor	β	β	γ^{-1}	θ	$\frac{1}{2}$
Fixed factor	ρ	ρ	α_{max}	η	α
Reliability / Stability	δ	—	δ_0	ϵ	—
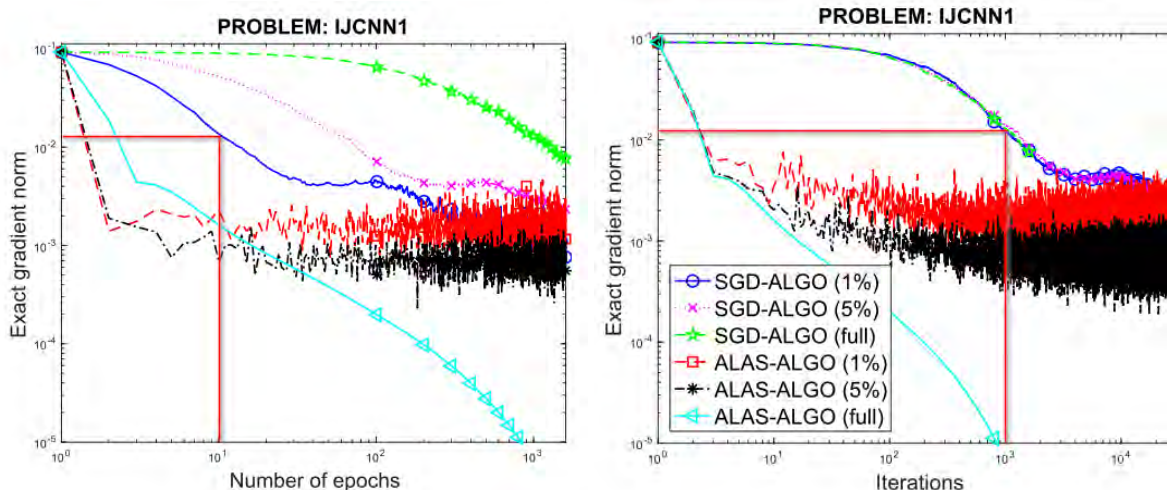
Figure 78: *Stochastic Newton with Armijo-like 2nd order line search* (Section 6.7). IJCNN1 dataset from the LIBSVM library. Three batch sizes were used (1%, 5%, 100%) for both SGD and ALAS (stochastic Newton Algorithm 7). The exact gradient norm for each of these six cases was plotted against the training epochs on the left, and against the iteration numbers on the right. An epoch is the number of non-overlapping minibatches (and thus iterations) to cover the whole training set. One epoch for a minibatch size of s% (respectively 1%, 5%, 100%) of the training set is equivalent to $100/s$ (respectively 100, 20, 1) iterations. Thus, for SGD-ALGO (1%), as shown, 10 epochs is equivalent to 1,000 iterations, with the same gradient norm. The markers on the curves were placed every 100 epochs (left) and 800 iterations (right). For the same number of epochs, say 10, SGD with smaller minibatches yielded lower gradient norm. The same was true for Algorithm 7 for number of epochs less than 10, but the gradient norm plateaued out after that with a lot of noise. Second-order Algorithm 7 converged faster than 1st-order SGD. See [145]. (Figure reproduced with permission of the authors.)

6.7 Stochastic Newton method with 2nd-order line search

The stochastic Newton method in Algorithm 7, described in [145], is a generalization of the deterministic Newton method in Algorithm 3 to add stochasticity via random selection of minibatches and Armijo-like 2nd-order line search.[205]

Upon a random selection of a minibatch as in Eq. (137), the computation of the estimates for the cost function $\widetilde{J}$ in Eq. (139), the gradient $\widetilde{g}$ in Eq. (140), and the Hessian $\widetilde{H}$ (new quantity) can proceed similar to that used for the standard SGD in Algorithm 4, recalled below for convenience:

$$\mathbb{B}^{|\mathsf{m}|} = \{ \boldsymbol{x}^{|i_1|}, \cdots, \boldsymbol{x}^{|i_\mathsf{m}|} \} \subseteq \mathbb{X} = \{ \boldsymbol{x}^{|1|}, \cdots, \boldsymbol{x}^{|\mathsf{M}|} \} \, , \tag{137}$$

$$\widetilde{J}(\boldsymbol{\theta}) = \frac{1}{\mathsf{m}} \sum_{k=1}^{k=\mathsf{m} \leq \mathsf{M}} J_{i_k}(\boldsymbol{\theta}) \, , \text{ with } J_{i_k}(\boldsymbol{\theta}) = J(f(\boldsymbol{x}^{|i_k|}, \boldsymbol{\theta}), \boldsymbol{y}^{|i_k|}) \, , \text{ and } \boldsymbol{x}^{|i_k|} \in \mathbb{B}^{|\mathsf{m}|} \, , \boldsymbol{y}^{|i_k|} \in \mathbb{T}^{|\mathsf{m}|} \, , \tag{139}$$

$$\widetilde{g}(\boldsymbol{\theta}) = \frac{\partial \widetilde{J}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \frac{1}{\mathsf{m}} \sum_{k=1}^{k=\mathsf{m} \leq \mathsf{M}} \frac{\partial \widetilde{J}_{i_k}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \, , \tag{140}$$

$$\widetilde{H}(\boldsymbol{\theta}) = \frac{\partial^2 \widetilde{J}(\boldsymbol{\theta})}{(\partial \boldsymbol{\theta})^2} = \frac{1}{\mathsf{m}} \sum_{k=1}^{k=\mathsf{m} \leq \mathsf{M}} \frac{\partial^2 \widetilde{J}_{i_k}(\boldsymbol{\theta})}{(\partial \boldsymbol{\theta})^2} \, . \tag{258}$$

In the above computation, the minibatch in Algorithm 7 is fixed, not adaptive such as in Algorithm 6.

If the current iterate $\boldsymbol{\theta}_k$ is in a flat region (or plateau) or at the bottom of a local convex bowl, then the smallest eigenvalue $\widetilde{\lambda}$ of the Hessian estimate $\widetilde{H}$ would be close to zero or positive, respectively, and the gradient estimate would be zero (line 7 in Algorithm 7):

$$\widetilde{\lambda} \geq -\delta^{1/2} \text{ and } \parallel \widetilde{g} \parallel = 0 \, , \tag{259}$$

where δ is a small positive number. In this case, no step (or update) would be taken, which is equivalent to setting step length to zero or descent direction to zero[206], then go to the next iteration $(k + 1)$. Otherwise, i.e., the conditions in Eq. (259) are not met, do the remaining steps.

If the current iterate $\boldsymbol{\theta}_k$ is on the downward side of a saddle point, characterized by the condition that the smallest eigenvalue $\widetilde{\lambda}_k$ is clearly negative (line 10 in Algorithm 7):

$$\widetilde{\lambda}_k < -\delta^{1/2} < 0 \, , \tag{260}$$

then find the eigenvector $\widetilde{v}_k$ corresponding to $\widetilde{\lambda}_k$, and scale it such that its norm is equal to the absolute value of this negative smallest eigenvalue, and such that this eigenvector forms an obtuse angle with the gradient $\widetilde{g}_k$, then select such eigenvector as the descent direction $\widetilde{d}_k$:

$$\widetilde{H}_k \widetilde{v}_k = \widetilde{\lambda}_k \widetilde{v}_k \text{ such that } \parallel \widetilde{v}_k \parallel = -\widetilde{\lambda}_k \text{ and } \widetilde{v}_k \bullet \widetilde{g}_k < 0 \Rightarrow \widetilde{d}_k = \widetilde{v}_k \tag{261}$$

When the iterate $\boldsymbol{\theta}_k$ is in a local convex bowl, the Hessian $\widetilde{H}_k$ is positive definite, i.e., the smallest eigenvalue $\widetilde{\lambda}_k$ is strictly positive, then use the Newton direction as descent direction $\widetilde{d}_k$ (line 13 in Algorithm 7):

$$\widetilde{\lambda}_k > \parallel \widetilde{g}_k \parallel^{1/2} > 0 \Rightarrow \widetilde{H}_k \widetilde{d}_k = -\widetilde{g}_k \tag{262}$$

The remaining case is when the iterate $\boldsymbol{\theta}_k$ is close to a saddle point such that the smallest eigenvalue $\widetilde{\lambda}_k$ is bounded below by $-\delta^{1/2}$ and above by $\parallel \widetilde{g}_k \parallel^{1/2}$, then $\widetilde{\lambda}_k$ is nearly zero, and thus the Hessian estimate

---

[205] The Armijo line search itself is 1st order; see Section 6.2 on full-batch deterministic optimization.
[206] See Step 2 of Algorithm 1 in [145].

$\widetilde{H}_k$ is nearly singular. Regularize (or stabilize) the Hessian estimate, i.e, move its smallest eigenvalue away from zero, by adding a small perturbation diagonal matrix using the sum of the bounds $\delta^{1/2}$ and $\parallel \widetilde{g}_k \parallel^{1/2}$. The regularized (or stabilized) Hessian estimate is no longer nearly singular, thus invertible, and can be used to find the Newton descent direction (lines 16-17 in Algorithm 7 for stochastic Newton and line 15 in Algorithm 3 for deterministic Newton):

$$-\delta^{1/2} \le \widetilde{\lambda}_k \le \parallel \widetilde{g}_k \parallel^{1/2} \Rightarrow 0 < \widetilde{\lambda}_k + \parallel \widetilde{g}_k \parallel^{1/2} +\delta^{1/2} \Rightarrow \left[\widetilde{H}_k + \left(\parallel \widetilde{g}_k \parallel^{1/2} +\delta^{1/2}\right) I\right] \widetilde{d}_k = -\widetilde{g}_k . \quad (263)$$

If the stopping criterion is not met, use Armijo's rule to find the step length ϵ_k to update the parameter $\boldsymbol{\theta}_k$ to $\boldsymbol{\theta}_{k+1}$, then go to the next iteration $(k + 1)$. (line 19 in Algorithm 7). The authors of [145] provided a detailed discussion of their stopping criterion. Otherwise, the stopping criterion is met, accept the current iterate $\widetilde{\boldsymbol{\theta}}_k$ as local minimizer estimate $\widetilde{\boldsymbol{\theta}}^\star$, stop the Newton-descent ② **for** loop to end the current training epoch τ.

From Eq. (125), the deterministic 1st-order Armijo's rule for steepest descent can be written as:

$$J(\boldsymbol{\theta}_k + \epsilon_k \, \boldsymbol{d}_k) \le J(\boldsymbol{\theta}_k) - \alpha\beta^a \rho \parallel \boldsymbol{d} \parallel^2 , \text{ with } \boldsymbol{d} = -\boldsymbol{g} , \tag{264}$$

with a being the minimum power for Eq. (264) to be satisfied. In Algorithm 7, the Armijo-like 2nd-order line search reads as follows:

$$\widetilde{J}(\widetilde{\boldsymbol{\theta}}_k + \epsilon_k \, \widetilde{\boldsymbol{d}}_k) \le \widetilde{J}(\widetilde{\boldsymbol{\theta}}_k) - \frac{1}{6}(\beta^a)^3 \rho \parallel \widetilde{\boldsymbol{d}}_k \parallel^3 , \tag{265}$$

with a being the minimum power for Eq. (265) to be satisfied. The parallelism between Eq. (265) and Eq. (264) is clear; see also Table 4.

Figure 78 shows the numerical results of Algorithm 7 on the IJCNN1 dataset ([211]) from the Library for Vector Support Machine (LIBSVM) library by [212]. It is not often to see plots versus epochs side by side with plots versus iterations. Some papers may have only plots versus iterations (e.g., [182]); other papers may rely only on plots versus epochs to draw conclusions (e.g., [56]). Thus Figure 78 provides a good example to see the differences, as noted in Remark 6.17.

Remark 6.17. *Epoch counter vs global iteration counter in plots.* When plotted gradient norm versus epochs (left of Figure 78), the three curves for SGD were separated, with faster convergence for smaller minibatch sizes Eq. (135), but the corresponding three curves fell on top of each other when plotted versus iterations (right of Figure 78). The reason was the scale on the horizontal axis was different for each curve, e.g., 1 iteration for full batch was equivalent to 100 iterations for minibatch size at 1% of full batch. While the plot versus iterations was the zoom-in view, but for each curve separately. To compare the rates of convergence among different algorithms and different minibatch sizes, look at the plots versus epochs, since each epoch covers the whole training set. It is just an optical illusion to think that SGD with different minibatch sizes had the same rate of convergence. ∎

The authors of [145] planned to test their Algorithm 7 on large datasets such as the CIFAR-10, and report the results in 2020.[207]

Another algorithm along the same line as Algorithm 6 and Algorithm 7 is the stochastic quasi-Newton method proposed in [146], where the stochastic Wolfe line search of [143] was employed, but with no numerical experiments on large datasets such as CIFAR-10, etc.

At the time of this writing, due to lack of numerical results with large datasets commonly used in the deep-learning community such as CIFAR-10, CIFAR-100 and the likes, for testing, and thus lack of comparison of performance in terns of cost and accuracy against Adam and its variants, our assessment is

---

[207] Per our private correspondence as of 2019.12.18.

1 Stochastic Newton with 2nd-order line-search and fixed minibatch (for training epoch τ)
   Data:
   • Parameter estimate $\widetilde{\boldsymbol{\theta}}_\tau^\star$ from previous epoch $(\tau - 1)$ (line 5 in Algorithm 4)
   • Select 3 Armijo paramters $\alpha \in (0,1)$, $\beta \in (0,1)$, $\rho > 0$, and stability parameter $\delta > 0$
   Result: Parameter estimate for next epoch $(\tau + 1)$: $\widetilde{\boldsymbol{\theta}}_{\tau+1}^\star$. (line 1 in Algorithm 6)

2 ▶ Begin training epoch τ. Set $\widetilde{\boldsymbol{\theta}}_1 = \widetilde{\boldsymbol{\theta}}_\tau^\star$;
3 ② **for** $k = 1, 2, \ldots$ **do**
4   Obtain minibatch $\mathbb{B}_k^{|\mathsf{m}|}$, Eq. (137) (line 7 in Algorithm 4) ;
5   Compute gradient estimate $\widetilde{\boldsymbol{g}}_k = \widetilde{\boldsymbol{g}}(\widetilde{\boldsymbol{\theta}}_k)$, Eq. (140) ;
6   Compute Hessian estimate $\widetilde{\boldsymbol{H}}_k = \widetilde{\boldsymbol{H}}(\widetilde{\boldsymbol{\theta}}_k)$, Eq. (258), and its smallest eigenvalue $\widetilde{\lambda}_k$;
7   **if** *Plateau or local minimizer,* $\widetilde{\lambda}_k \geq -\delta^{1/2}$ *and* $\| \widetilde{\boldsymbol{g}}_k \| = 0$, *Eq. (259)* **then**
8   ▶ Take zero step length. Go to next iteration $(k+1)$, line 35.
9   **else**
10   **if** *Downward side of saddle point,* $\widetilde{\lambda}_k < -\delta^{1/2}$, *Eq. (260)* **then**
11   ▶ Compute descent direction as eigenvector $\widetilde{\boldsymbol{v}}_k$ for $\widetilde{\lambda}_k$, Eq. (261): Set $\widetilde{\boldsymbol{d}}_k = \widetilde{\boldsymbol{v}}_k$;
12   **end**
13   **if** *Convex bowl* $\widetilde{\lambda}_k > \| \widetilde{\boldsymbol{g}}_k \|^{1/2}$ **then**
14   ▶ Compute Newton descent direction, Eq. (262): $\widetilde{\boldsymbol{H}}_k \widetilde{\boldsymbol{d}}_k = -\widetilde{\boldsymbol{g}}_k$,
15   **end**
16   **if** $\widetilde{\boldsymbol{d}}_k$ *not yet selected,* $-\delta^{1/2} \leq \widetilde{\lambda}_k \leq \| \widetilde{\boldsymbol{g}}_k \|^{1/2}$ **then**
17   ▶ Compute regularized Newton descent direction, Eq. (263)
18   **end**
19   **if** *Stopping criterion not met* **then**
20   ▶ Compute step length ϵ_k using Armijo's rule.
21   Initialize step length $\epsilon_k \leftarrow \rho$;
22   ③ **for** *a=1,2,...* **do**
23   **if** *Armijo decrease condition Eq. (265) not satisfied* **then**
24   ▶ Decrease step length: Set $\epsilon_k \leftarrow \beta\epsilon_k$;
25   **else**
26   ▶ Armijo decrease condition Eq. (265) met.
27   ▶ Update network parameter $\widetilde{\boldsymbol{\theta}}_k$ to $\widetilde{\boldsymbol{\theta}}_{k+1}$: Set $\widetilde{\boldsymbol{\theta}}_{k+1} = \widetilde{\boldsymbol{\theta}}_k + \epsilon_k \widetilde{\boldsymbol{d}}_k$;
28   **end**
29   **end**
30   **else**
31   ▶ Stopping criterion met. Update minimizer estimate for epoch $(\tau+1)$: $\widetilde{\boldsymbol{\theta}}_{\tau+1}^\star \leftarrow \widetilde{\boldsymbol{\theta}}_k$;
32   Stop ② **for** loop. (End training epoch τ.)
33   **end**
34   **end**
35   ▶ Continue to next iteration $(k+1)$: Set $k \leftarrow k+1$;
36 **end**

Algorithm 7: *Stochastic Newton method with 2nd-order Armijo-like line-search and fixed minibatch* (Sections 6.2.3, 6.7, Algorithms 4, 6). Adapted from [145]. See Algorithm 3 for *deterministic* Newton and quasi-Newton algorithm with Armijo line search and fixed minibatch, Algorithm 4 for Standard SGD, Algorithm 6 for SGD with Armijo line search and adaptive minibatch, and Table 4 for a comparison of the notations used by several authors in relation to the Armijo line search.

that SGD and its variants, or Adam and its better variants, particularly AdamW, continue to be the prevalent methods for training.

Time constraint did not allow us to review other stochastic optimization methods such as that with the gradient-only line search in [142] and [179] could not be reviewed here.

7 Dynamics, sequential data, sequence modeling

7.1 Recurrent Neural Networks (RNNs)

In many fields of physics, the respective governing equations that describe the response of a system to (external) stimuli follow a common pattern. The temporal and/or spatial change of some quantity of a system is balanced by sources that cause the change, which is why we refer to equations of this kind as balance relations. The balance of linear momentum in mechanics, for instance, establishes a relationship between the temporal change of a body's linear momentum and the forces acting on the body. Along with kinematic relations and constitutive laws, the balance equations provide the foundation to derive the equations of motion of some mechanical system. For linear problems, the equations of motion constitute a system of second-order ODEs in appropriate (generalized) coordinates $\boldsymbol{d}$, which, in case of continua, is possibly obtained by some spatial discretization,

$$\boldsymbol{M}\ddot{\boldsymbol{d}} + \boldsymbol{D}\dot{\boldsymbol{d}} + \boldsymbol{C}\boldsymbol{d} = \boldsymbol{f}. \tag{266}$$

In the above equation, $\boldsymbol{M}$ denotes the mass matrix, $\boldsymbol{D}$ is a damping matrix and $\boldsymbol{C}$ is the stiffness matrix; $\boldsymbol{f}$ is the vector of (generalized) forces. The equations of motion can be rewritten as a system of first-order ODEs by introducing the vector of (generalized) velocities $\boldsymbol{v}$,

$$\begin{bmatrix} \dot{\boldsymbol{d}} \\ \dot{\boldsymbol{v}} \end{bmatrix} = \underbrace{\begin{bmatrix} \boldsymbol{0} & \boldsymbol{I} \\ -\boldsymbol{M}^{-1}\boldsymbol{C} & -\boldsymbol{M}^{-1}\boldsymbol{D} \end{bmatrix}}_{\boldsymbol{A}} \begin{bmatrix} \boldsymbol{d} \\ \boldsymbol{v} \end{bmatrix} + \underbrace{\begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{M}^{-1} \end{bmatrix}}_{\boldsymbol{B}} \boldsymbol{f} . \tag{267}$$

In control theory, $\boldsymbol{A}$ and $\boldsymbol{b} = \boldsymbol{B}\boldsymbol{f}$ are referred to as state matrix and input vector, respectively.[208] If we gather the (generalized) coordinates and velocities in the state vector $\boldsymbol{q}$,

$$\boldsymbol{q} = \begin{bmatrix} \boldsymbol{d}^T & \boldsymbol{v}^T \end{bmatrix}^T , \tag{268}$$

we obtain a compact representation of the equations of motion, which relates the temporal change of the system's state $\dot{\boldsymbol{q}}$ to its current state $\boldsymbol{q}$ and the input $\boldsymbol{b}$ linearly by means of

$$\dot{\boldsymbol{q}} = \boldsymbol{A}\boldsymbol{q} + \boldsymbol{B}\boldsymbol{f} = \boldsymbol{A}\boldsymbol{q} + \boldsymbol{b}. \tag{269}$$

We found similar relations in neuroscience,[209] where the dynamics of neurons was accounted for, e.g., in the pioneering work [214], whose author modeled a neuron as electrical circuit with capacitances. Time-continuous RNNs were considered in a paper on back-propagation [215]. The temporal change of an RNN's state is related to the current state $\boldsymbol{y}$ and the input $\boldsymbol{x}$ by

$$\dot{\boldsymbol{y}} = -\boldsymbol{y} + a(\boldsymbol{W}\boldsymbol{y}) + \boldsymbol{x}, \tag{270}$$

where a denotes a non-linear activation function as, e.g., the sigmoid function, and $\boldsymbol{W}$ is the weight matrix that describes the connection among neurons.[210]

---

[208] The state-space representation of time-continuous LTI-systems in control theory, see, e.g., Chapter 3 of [213], "State Variables and the State Space Description of Dynamic Systems" is typically written as $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{B}\boldsymbol{u}$, with the output equation $\boldsymbol{y} = \boldsymbol{C}\boldsymbol{x} + \boldsymbol{D}\boldsymbol{u}$. The state vector is denoted by $\boldsymbol{x}$, $\boldsymbol{u}$ and $\boldsymbol{y}$ are the vectors of inputs and outputs, respectively. The ODE describes the (temporal) evolution of the system's state. The output of the system $\boldsymbol{y}$ is a linear combination of states $\boldsymbol{x}$ and the inputs $\boldsymbol{u}$.

[209] See Section 13.2.2 on "Dynamic, time dependence, Volterra series".

[210] See the general time-continuous neural network with a continuous delay described by Eq. (514).

Returning to mechanics, we are confronted with the problem that the equations of motion do not admit closed-form solutions in general. To construct approximate solution, time-integration schemes need to be resorted to, where we mention a few examples such as Newmark's method [216], the Hilber-Hughes-Taylor (HHT) method [217], and the generalized-α method [218]. For simplicity, we have a closer look at the classical trapezoidal rule, in which the time integral of some function f over a time step $\Delta t = t_{n+1} - t_n$ is approximated by the average of the function values $f(t_{n+1})$ and $f(t_n)$. If we apply the trapezoidal rule to the system of ODEs in Eq. (269), we obtain a system of algebraic relations that reads

$$q^{[n+1]} = q^{[n]} + \frac{\Delta t}{2} \left(Aq^{[n]} + b^{[n]} + Aq^{[n+1]} + b^{[n+1]} \right). \tag{271}$$

Rearranging the above relation for the new state gives the update equation for the state vector,

$$q^{[n+1]} = \left(I - \frac{\Delta t}{2} A \right)^{-1} \left(I + \frac{\Delta t}{2} A \right) q^{[n]} + \frac{\Delta t}{2} \left(I - \frac{\Delta t}{2} A \right)^{-1} \left(b^{[n]} + b^{[n+1]} \right), \tag{272}$$

which determines the next state $q^{[n+1]}$ in terms of the state $q^{[n]}$ as well as the inputs $b^{[n]}$ and $b^{[n+1]}$.[211] To keep it short, we introduce the matrices W and U as

$$W = \left(I - \frac{\Delta t}{2} A \right)^{-1} \left(I + \frac{\Delta t}{2} A \right), \qquad U = \frac{\Delta t}{2} \left(I - \frac{\Delta t}{2} A \right)^{-1}, \tag{273}$$

which allows us to rewrite Eq. (272) as

$$q^{[n+1]} = Wq^{[n]} + U \left(b^{[n]} + b^{[n+1]} \right). \tag{274}$$

The update equation of time-discrete RNNs is similar to the discretized equations of motion Eq. (274). Unlike feed-forward neural networks, the state h of an RNN at the n-th time step,[212] which is denoted by $h^{[n]}$, does not only depend on the current input $x^{[n]}$, but also on the state $h^{[n-1]}$ of the previous time step $n - 1$. Following the notation in [78], we introduce a *transition function* f that produces the new state,

$$h^{[n]} = f(h^{[n-1]}, x^{[n]}; \theta). \tag{275}$$

Remark 7.1. In [78], there is a distinction between the "hidden state" of an RNN cell at the n-th step denoted by $h^{[n]}$, where "h" is mnemonic for "hidden", and the cell's output $y^{[n]}$. The output of a multi-layer RNN is a linear combination of the last layer's hidden state. Depending on the application, the output is not necessarily computed at every time step, but the network can "summarize" sequences of inputs to produce an output after a certain number of steps.[213] If the output is identical to the hidden state,

$$y^{[n]} \equiv h^{[n]}, \tag{276}$$

$h^{[n]}$ and $y^{[n]}$ can be used interchangeably. In the current Section 7 on "Dynamics, sequential data sequence modeling", the notation $h^{[n]}$ is used, whereas in Section 4 on "Static, feedforward networks", the notation $y^{(\ell)}$ is used to designate the output of the "hidden layer" (ℓ), keeping in mind the equivalence in Eq. (22) in Remark 4.2. whenever necessary, readers are reminded of the equivalence in Eq. (276) to avoid possible confusion when reading deep-learning literature.   ∎

---

[211] We can regard the trapezoidal rule as a combination of Euler's explicit and implicit methods. The *explicit Euler* method approximates time-integrals by means of rates (and inputs) at the beginning of a time step. The next state (at the end of a time step) is obtained from previous state and the previous input as $q_{n+1} = q_n + \Delta t\, f(q_n, b_n)$ $= (I + \Delta t\, A)\, q_n + \Delta t\, b_n$. On the contrary, the *implicit Euler* method uses rates (and inputs) at the end of a time step, which leads to the update relation $q_{n+1} = q_n + \Delta t\, f(q_{n+1}, b_{n+1}) = (I - \Delta t\, A)^{-1}\, q_n + \Delta t\, b_{n+1}$.

[212] Though the elements $x^{[n]}$ of a sequence are commonly referred to as "time steps", the nature of a sequence is not necessarily temporal. The time step index t then merely refers to a position within some given sequence.

[213] cf. [78], p. 371, Figure 10.5.

The above relation is illustrated as a circular graph in Figure 79 (left), where the delay is explicit in the superscript of $\boldsymbol{h}^{[n-1]}$. The hidden state $\boldsymbol{h}^{[n-1]}$ at $n-1$, in turn, is a function of the hidden state $\boldsymbol{h}^{[n-2]}$ and the input $\boldsymbol{x}^{[n-1]}$,

$$\boldsymbol{h}^{[n]} = f(f(\boldsymbol{h}^{[n-2]}, \boldsymbol{x}^{[n-1]}; \boldsymbol{\theta}), \boldsymbol{x}^{[n]}; \boldsymbol{\theta}). \tag{277}$$

Continuing this *unfolding* process repeatedly until we reach the beginning of a sequence, the recurrence can be expressed as a function $g^{[n]}$,

$$\boldsymbol{h}^{[n]} = g^{[n]}(\boldsymbol{x}^{[n]}, \boldsymbol{x}^{[n-1]}, \boldsymbol{x}^{[n-2]}, \ldots, \boldsymbol{x}^{[2]}, \boldsymbol{x}^{[1]}, \boldsymbol{h}^{[0]}; \boldsymbol{\theta}) = g^{[n]}(\{\boldsymbol{x}^{[k]}, k = 1, \ldots, n\}, \boldsymbol{h}^{[0]}; \boldsymbol{\theta}), \tag{278}$$

which takes the entire sequence up to the current step n, i.e., $\{\boldsymbol{x}^{[k]}, k = 1, \ldots, n\}$ (along with an initial state $\boldsymbol{h}^{[0]}$ and parameters $\boldsymbol{\theta}$), as input to compute the current state $\boldsymbol{h}^{[n]}$. The unfolded graph representation is illustrated in Figure 79 (right).
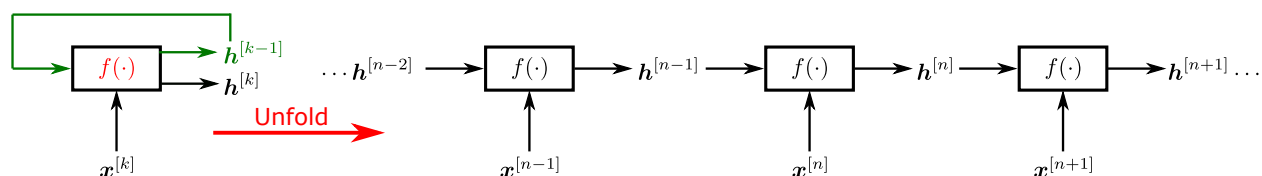

Figure 79: *Folded and unfolded discrete RNN* (Section 7.1, 13.2.2). *Left: Folded discrete RNN* at configuration (or state) number $[k]$, where k is an integer, with input $\boldsymbol{x}^{[k]}$ to a multilayer neural network $f(\cdot) = f^{(1)} \circ f^{(2)} \circ \cdots \circ f^{(L)}(\cdot)$ as in Eq. (18), having a feedback loop $\boldsymbol{h}^{[k-1]}$ with delay by one step, to produce output $\boldsymbol{h}^{[k]}$. *Right: Unfolded discrete RNN*, where the feedback loop is unfolded, centered at $k = n$, as represented by Eq. (275). This graphical representation, with $f(\cdot)$ being a multilayer neural network, is more general than Figure 10.2 in [78], p. 366, and is a particular case of Figure 10.13b in [78], p. 388. See also the continuous RNN explained in Section 13.2.2 on "Dynamic, time dependence, Volterra series", Eq. (514), Figure 135, for which we refer readers to Remark 7.1 and the notation equivalence $\boldsymbol{y}^{[k]} \equiv \boldsymbol{h}^{[k]}$ in Eq. (276).

As an example, consider the default *("vanilla")* single-layer RNN provided by PyTorch[214] and Tensor-Flow[215], which is also described in [78], p. 370:

$$\boldsymbol{z}^{[n]} = \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{[n-1]} + \boldsymbol{U}\boldsymbol{x}^{[n]},$$

$$\boldsymbol{h}^{[n]} = a(\boldsymbol{z}^{[n]}) = \tanh(\boldsymbol{z}^{[n]}). \tag{279}$$

First, $\boldsymbol{z}^{[n]}$ is formed from an affine transformation of the current input $\boldsymbol{x}^{[n]}$, the previous hidden state $\boldsymbol{h}^{[n-1]}$ and the bias $\boldsymbol{b}$ using weight matrices $\boldsymbol{U}$ and $\boldsymbol{W}$, respectively. Subsequently, the hyperbolic tangent is applied to the elements of $\boldsymbol{z}^{[n]}$ as activation function that produces the new hidden state $\boldsymbol{h}^{[n]}$.

A common design pattern of RNNs adds a linear output layer to the simplistic example in Figure 79, i.e., the RNN has a recurrent connection between its hidden units, which represent the state $\boldsymbol{h}$,[216] and produces an output at each time step.[217] Figure 80 shows a two-layer RNN, which extends our above example by a

---

[214] See PyTorch documentation: Recurrent layers, Original website (Internet archive)
[215] See TensorFlow API: TensorFlow Core r1.14: tf.keras.layers.SimpleRNN, Original website (Internet archive)
[216] For this reason, $\boldsymbol{h}$ is typically referred to as the *hidden state* of an RNN.
[217] Such neural network is *universal*, i.e., any function computable by a *Turing machine* can be computed by an RNN of finite size, see [78], p. 368.

second layer, i.e., the first layer is identical to Eq. (279),

$$z_1^{[n]} = b + Wh_1^{[n-1]} + Ux^{[n]},$$

$$h_1^{[n]} = a_1(z_1^{[n]}) = \tanh(z_1^{[n]}). \tag{280}$$

and the second layer forms a linear combination of the first layer's output $h_1^{[n]}$ and the bias c using weights V. Assuming the output $h_2^{[n]}$ is meant to represent probabilities, it is input to a $\mathrm{softmax}(\cdot)$ activation (a derivation of which is provided in Remark 5.3 in Section 5.1.3):

$$z_2^{[n]} = c + Vh_1^{[n]},$$

$$h_2^{[n]} = a_2(z_2^{[n]}) = \mathrm{softmax}(z_2^{[n]}), \tag{281}$$

which was given in Eq. (84); see [78], p. 369. The parameters of the network are the weight matrices W, U and V, as well as the biases b and c of the recurrence layer and the output layer, respectively.
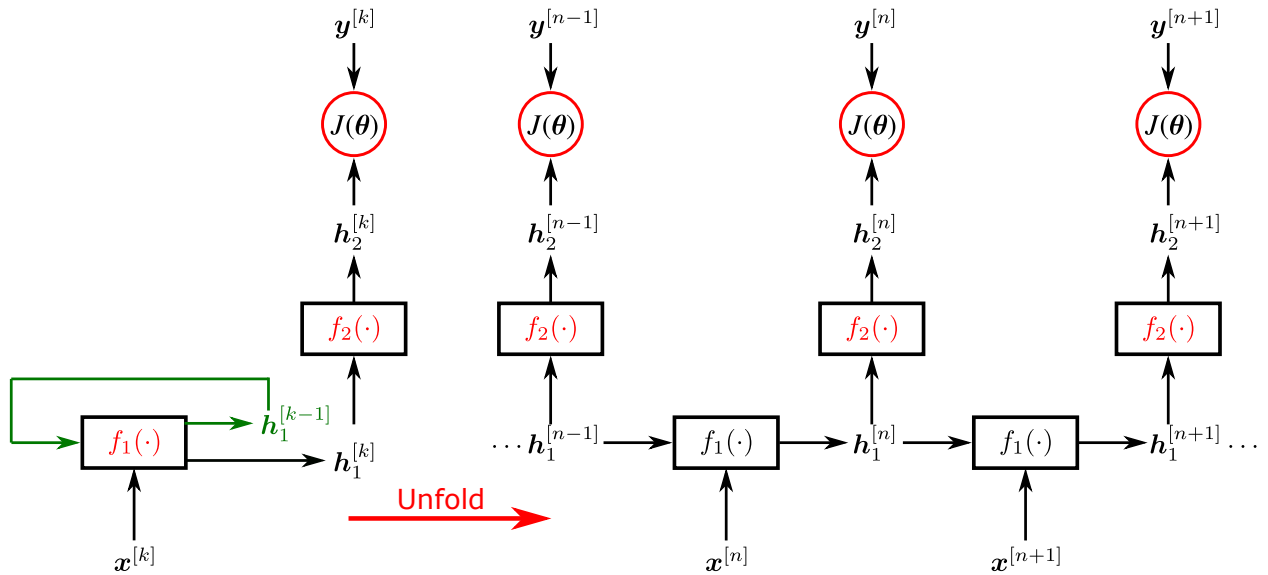

Figure 80: *RNN with two multilayer neural networks (MLNs),* (Section 7.1) denoted by $f_1(\cdot)$ and $f_2(\cdot)$, whose outputs are fed into the loss function for optimization. This RNN is a generalization of the RNN in Figure 79, and includes the RNN in Figure 10.3 in [78], p. 369, as a particular case, where Eq. (282) of the first layer is simply $h_1^{[n]} = f_1(x^{[n]}) = a_1(z_1^{[n]})$, with $z_1^{[n]} = b + Wh_1^{[n-1]} + Ux^{[n]}$ and $a_1(\cdot) = \tanh(\cdot)$, whereas Eq. (282) is $h_2^{[n]} = f_2(x^{[n]}) = a_2(z_2^{[n]})$, with $z_2^{[n]} = c + Vh_1^{[n]}$ and $a_2(\cdot) = \mathrm{softmax}(\cdot)$ for the second layer. The general relation for both MLNs is $h_j^{[n]} = f_j(h_j^{[n-1]}, h_{j-1}^{[n]}) = a_j(z_j^{[n]})$, for $j = 1, 2$.

Irrespective of the number of layers, the hidden state $h_j^{[n]}$ of the j-th layer is gennerally computed from the previous hidden state $h_j^{[n-1]}$ and the input to the layer $h_{j-1}^{[n]}$,

$$h_j^{[n]} = f_j(h_j^{[n-1]}, h_{j-1}^{[n]}) = a_j(z_j^{[n]}). \tag{282}$$

Other design patterns for RNNs show, e.g., recurrent connections between the hidden units but produce a single output only. RNNs may also have recurrent connections from the output at one time step to the hidden unit of next time step.

Comparing the recurrence relation in Eq. (277) and its unfolded representation in Eq. (278), we can make the following observations:

- The unfolded representation after n steps $g^{[n]}$ can be regarded as a factorization into repeated applications of f. Unlike $g^{[n]}$, the transition function f does not depend on the length of the sequence and always has the same input size.

- The same transition function f with the same parameters $\boldsymbol{\theta}$ is used in every time step.

- A state $\boldsymbol{h}^{[n]}$ contains information about the whole past sequence.

Remark 7.2. *Depth of RNNs.* For the above reasons and Figures 79-80, "RNNs, once unfolded in time, can be seen as very deep feedforward networks in which all the layers share the same weights" [13]. See Section 4.6.1 on network depth and Remark 4.5. ■

By nature, RNNs are typically employed for the processing of sequential data $\boldsymbol{x}^{[1]}, \ldots, \boldsymbol{x}^{[\tau]}$, where the sequence length τ typically need not be constant. To process data of variable length, parameter sharing is a fundamental concept that characterizes RNNs. Instead of using separate parameters for each time step in a sequence, the same parameters are shared across several time-steps. The idea of parameter sharing does not only allows us to process sequences of variable length (and possibly not seen during training), the *"statistical strength"*[218] is also shared across different positions in time, which is important if relevant information occurs at different positions within a sequence. A fully-connected feedforward neural network that takes each element of a sequence as input instead needs to learn all its rules separately for each position in the sequence.

Comparing the update equations Eq. (274) and Eq. (279), we note the close resemblance of dynamic systems and RNNs. Let aside the non-linearity of the activation function and the presence of the bias vector, both have state vectors with recurrent connections to the previous states. Employing the trapezoidal rule for the time-discretization, we find a recurrence in the input, which is not present in the type of RNNs described above. The concept of parameter sharing in RNNs translates into the notion of time-invariant systems in dynamics, i.e., the state matrix $\boldsymbol{A}$ does not depend on time. In the computational mechanics context, typical outputs of a simulation could be, e.g., the displacement of a structure at some specified point or the von-Mises stress field in its interior. The computations required for determining the output from the state (e.g., nodal displacements of a finite-element model) depend on the respective nature of the output quantity and need not be linear.

The crucial challenge in RNNs is to learn long-term dependencies, i.e., relations among distant elements in input sequences. For long sequences, we face the problem of vanishing or exploding gradients when training the network by means of back-propagation. To understand vanishing (or exploding) gradients, we can draw analogies between RNNs and dynamic systems once again. For this purpose, we consider an RNN without inputs whose activation function is the identity function:

$$\boldsymbol{h}^{[n]} = \boldsymbol{W} \boldsymbol{h}^{[n-1]}. \tag{283}$$

From the dynamics point of view, the above update equation corresponds to a linear autonomous system, whose time-discrete representation is given by

$$\boldsymbol{h}_{n+1} = \boldsymbol{W} \boldsymbol{h}_n. \tag{284}$$

Clearly, the equilibrium state of the above system is the trivial state $\boldsymbol{h} = \boldsymbol{0}$. Let $\boldsymbol{h}_0$ denote a perturbation of the equilibrium state. An equilibrium state is called *Lyapunov stable* if trajectories of the system, i.e.,

---

[218] see [78], p. 363.

the states at times $t \geq 0$, remain bounded. If the trajectory eventually arrives at the equilibrium state for $t \to \infty$ (i.e., the trajectory is attractive), the equilibrium of the system is called *asymptotically stable*. In other words, an initial perturbation $\boldsymbol{h}_0$ from the equilibrium state (i.e., the initial state) vanishes over time in the case of asymptotic stability. Linear time-discrete systems are asymptotically stable if all eigenvalues λ_i of $\boldsymbol{W}$ have an absolute value smaller than one. From the unfolded representation in Eq. (278) (see also Figure 79 (right)), it is understood that we observe a similar behavior in the RNN described above. At step n, the initial state $\boldsymbol{h}^{[0]}$ has been multiplied n times with the weight matrix $\boldsymbol{W}$, i.e.,

$$\boldsymbol{h}^{[n]} = \boldsymbol{W}^n \boldsymbol{h}^{[0]}. \tag{285}$$

If eigenvalues of $\boldsymbol{W}$ have absolute values smaller than one, $\boldsymbol{h}^{[0]}$ exponentially decays to zero in long sequences. On the other hand, we encounter exponentially increasing values for eigenvalues of $\boldsymbol{W}$ with magnitudes greater than one, which is equivalent to an unstable system in dynamics. When performing back-propagation to train RNNs, gradients of the loss function need to be passed backwards through the unfolded network, where gradients are repeatedly multiplied with $\boldsymbol{W}$ as is the initial state $\boldsymbol{h}^{[0]}$ in the forward pass. The exponential decay (or increase) therefore causes gradients to vanish (or explode) in long sequences, which makes it difficult to learn long-term dependencies among distant elements of a sequence.
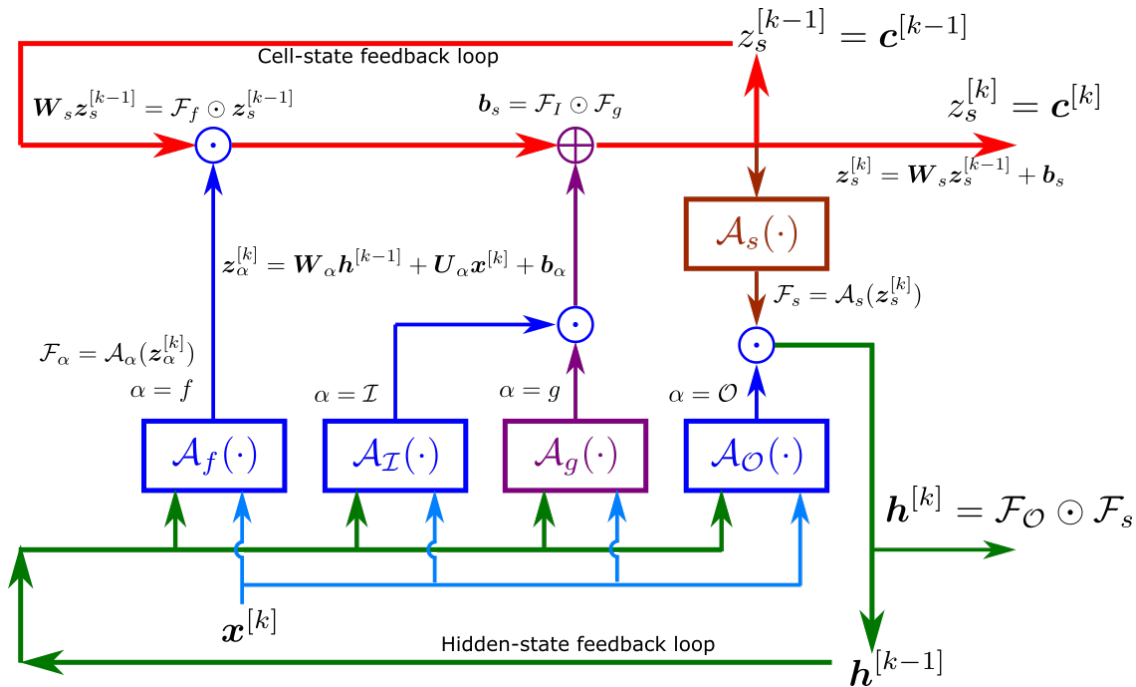

Figure 81: *Folded Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) cell* (Section 7.2, 11.3.3). The cell state at $[k]$ is denoted by $z_s^{[k]} \equiv c^{[k]}$. Two feedback loops, one for cell state $\boldsymbol{z}_s$ and one for hidden state $\boldsymbol{h}$, with one-step delay $[k-1]$. The key unified recurring relation is $\mathcal{F}_\alpha = \mathcal{A}_\alpha(\boldsymbol{z}_\alpha^{[k]})$, with $\alpha \in \{s \text{ (state)}, f \text{ (forget)}, \mathcal{I} \text{ (Input)}, g \text{ (external input)}, \mathcal{O} \text{ (Output)}\}$, where $\mathcal{A}_\alpha$ is a sigmoidal activation (squashing) function, and $\boldsymbol{z}_\alpha^{[k]}$ is a linear combination of some inputs with weights plus biases at cell state $[k]$. See Figure 82 for unfolded RNN with LSTM cells, and also Figure 15 in Section 2.3.2.

7.2 Long Short-Term Memory (LSTM) unit

The vanishing (exploding) gradient problem prevents us from effectively learning long-term dependencies in long input sequences by means of conventional RNNs. Gated RNNs as the long short-term memory

(LSTM) and networks based on the gated recurrent unit (GRU) have proven to successfully overcome the vanishing gradient problem in diverse applications. The common idea of gated RNNs is to create paths through time along which gradients neither vanish nor explode. Gated RNNs can accumulate information in their state over many time steps, but, once the information has been used, they are capable to forget their state by, figuratively speaking, "closing gates" to stop the information flow. This concept bears a resemblance to residual networks, which introduce skip connections to circumvent vanishing gradients in deep feed-forward networks; see Section 4.6.2 on network "Architecture".

Remark 7.3. *What is "short-term" memory?* The vanishing gradient at the earlier states of an RNN (or layers in the case of a multilayer neural network) makes it that information in these earlier states (or layers) did not propagate forward to contribute to adjust the predicted outputs to track the labeled outputs, so to decrease the loss. A state $[k]$ with two feedback loops is depicted in Figure 81. The reason for information in the earlier states not propagating forward was because the weights in these layers did not change much (i.e., did not learn), due to very small gradients, due to repeated multiplications of numbers with magnitude less than 1. As a result, information in earlier states (or "events") played little role in decreasing the loss function, and thus had only "short-term" effects, rather than the needed long-term effect to be carried forward to the output layer. Hence, we had a short-term memory problem. See also Remark 5.5 in Section 5 on back-propagation for vanishing or exploding gradient in multilayer neural networks.   ∎

In their pioneering work on LSTM, the authors of [24] presented a mechanism that allows information (inputs, gradients) to flow over a long duration by introducing additional states, paths and self-loops. The additional components are encapsulated in so-called *LSTM cells*. LSTM cells are the building blocks for LSTM networks, where they are connected recurrently to each other analogously to hidden neurons in conventional RNNs. The introduction of a *cell state*[219] c is one of the key ingredients to LSTM. The schematic cell representation (Figure 81) shows that the cell state can propagate through an LSTM cell without much interference, which is why this path is described as "conveyor belt" for information in [219].

Another way to explain that could contribute to elucidate the concept is: Since information in RNN cannot be stored for a long time, over many subsequent steps, LSTM cell corrected this short-term memory problem by remembering the inputs for a long time:

> "A special unit called the memory cell acts like an accumulator or a gated leaky neuron: it has a connection to itself at the next time step that has a weight of one, so it copies its own real-valued state and accumulates the external signal, but this self-connection is multiplicatively gated by another unit that learns to decide when to clear the content of the memory." [13].

In a unified manner, the various relations in the original LSTM unit depicted in Figure 81 can be expressed in a single key generic recurring-relation that is more easily remembered:

$$\mathcal{F}_\alpha(\boldsymbol{x}, \boldsymbol{h}) = \mathcal{A}_\alpha(\boldsymbol{z}_\alpha) \text{ with } \alpha \in \{s \text{ (state)}, f \text{ (forget)}, \mathcal{I} \text{ (Input)}, g \text{ (external input)}, \mathcal{O} \text{ (Output)}\} , \qquad (286)$$

where $\boldsymbol{x} = \boldsymbol{x}^{[k]}$ is the input at cell state $[k]$, $\boldsymbol{h} = \boldsymbol{h}^{[k-1]}$ the hidden variable at cell state $[k-1]$, $\mathcal{A}_\alpha$ (with "$\mathcal{A}$" being mnemonic for "activation") is a sigmoidal activation (squashing) function—which can be either the logistic sigmoid function or the hyperbolic tangent function (see Section 5.3.1)—and $\boldsymbol{z}_\alpha = \boldsymbol{z}_\alpha^{[k]}$ is a linear combination of some inputs with weights plus biases at cell state $[k]$. The choice of the notation in Eq. (286) is to be consistent with the notation in the relation $\widetilde{\boldsymbol{y}} = f(\boldsymbol{x}) = a(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) = a(\boldsymbol{z})$ in the caption of Figure 32.

In Figure 81, two types of squashing functions are used: One type (three blue boxes with $\alpha \in \{f, \mathcal{I}, \mathcal{O}\}$) squashes inputs into the range $(0, 1)$ (e.g., the logistic sigmoid, Eq. (113)), and the other type (purple box

---

[219] The cell state is denoted with the variable s in [78], p. 399, Eq. (10.41).

with $\alpha = g$, brown box with $\alpha = s$) squashes inputs into the range $(-1, +1)$ (e.g., the hyperbolic tangent, Eq. (114)). The *gates* are the activation functions $\mathcal{A}_\alpha$ with $\alpha \in \{f, \mathcal{I}, g, \mathcal{O}\}$ (3 blue and 1 purple boxes), with argument containing the input $\boldsymbol{x}^{[k]}$ (through $\boldsymbol{z}_\alpha^{[k]}$).

Remark 7.4. The activation function $\mathcal{A}_s$ (brown box in Figure 81) is a hyperbolic tangent, but is not called a gate, since it has the cell state $\boldsymbol{c}^{[k]}$, but not the input $\boldsymbol{x}^{[k]}$, as argument. In other words, a gate has to take in the input $\boldsymbol{x}^{[k]}$ in its argument. ■

There are two feedback loops, each with a delay by one step. The cell-state feedback loop (red) at the top involves the LSTM cell state $\boldsymbol{z}_s^{[k-1]} \equiv \boldsymbol{c}^{[k-1]}$, with a delay by one step. Since the cell state $\boldsymbol{z}_s^{[k-1]} \equiv \boldsymbol{c}^{[k-1]}$ is *not* squashed by a sigmoidal activation function, vanishing or exploding gradient is avoided; the "short-term" memory would last longer, thus the name "Long Short-Term Memory". See Remark 5.5 on vanishing or exploding gradient in back-propagation, and Remark 7.3 on short-term memory.

In the hidden-state feedback loop (green) at the bottom, the combination $\boldsymbol{z}_\mathcal{O}^{[k]}$ of the input $\boldsymbol{x}^{[k]}$ and the previous hidden state $\boldsymbol{h}^{[k-1]}$ is squashed into the range $(0, 1)$ by the processor $\mathcal{F}_\mathcal{O}$ to form a factor that filters out less important information from the cell state $\boldsymbol{z}_s^{[k]} \equiv \boldsymbol{c}^{[k]}$, which had been squashed into the range $(-1, +1)$. See Figure 82 for an unfolded RNN with LSTM cells. See also Appendix 2 and Figure 152 for an alternative block diagram.
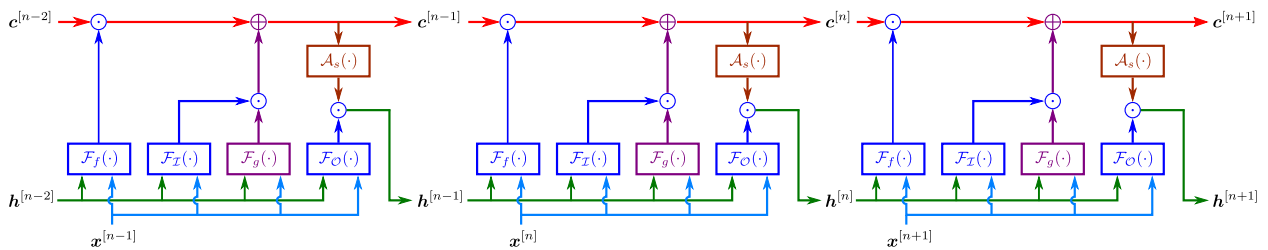

Figure 82: *Unfolded RNN with LSTM cells* (Sections 2.3.2, 7.2, 12.1): In this unfolded RNN, the cell states are centered at the LSTM cell $[k = n]$, preceded by the LSTM cell $[k = n - 1]$, and followed by the LSTM cell $[k = n + 1]$. See Eq. (290) for the recurring relation among the successive cell states, and Figure 81 for a *folded* RNN with details of an LSTM cell. Unlike conventional RNNs, in which the hidden state is repeatedly multiplied with its shared weights, the additional cell state of an LSTM network can propagate over several time steps without much interference. For this reason, LSTM networks typically perform significantly better on long sequences as compared to conventional RNNs, which suffer from the problem of vanishing gradients when being trained. See also Figure 117 in Section 12.2.

As the term suggests, the presence of gates that control the information flow are a further key concept in gated RNNs and LSTM, in particular. Gates are constructed from a linear layer with a sigmoidal function (logistic sigmoid or hyperbolic tangent) as activation function that squashes the components of a vector into the range $(0, 1)$ (logistic sigmoid) or $(-1, +1)$ (hyperbolic tangent). A component-wise multiplication of the sigmoid's output with the cell state controls the evolution of the cell state (forward pass) and the flow of its gradients (backward pass), respectively. Multiplication with 0 suppresses the propagation of a component of the cell state, whereas a gate value of 1 allows a component to pass.

To understand the function of LSTM, we follow the paths information is routed through an LSTM cell. At time n, we assume that the hidden state $\boldsymbol{h}^{[n-1]}$ and the cell state $\boldsymbol{c}^{[n-1]}$ from the previous time step along with the current input $\boldsymbol{x}^{[n]}$ are given. The hidden state $\boldsymbol{h}^{[n-1]}$ and the input $\boldsymbol{x}^{[n]}$ are inputs to the *forget gate*, i.e., a fully connected layer with a sigmoid non-linearity, and the general expression Eq. (286), with $k = n$

(see Figure 82), becomes (Figure 81, where the superscript $[k]$ on $\mathcal{F}_\alpha$ was omitted to alleviate the notation)

$$\alpha = f \Rightarrow \mathcal{F}_f^{[n]} = \mathcal{A}_f(\boldsymbol{z}_f^{[n]}) \to \boldsymbol{f}^{[n]} = \mathfrak{s}(\boldsymbol{z}_f^{[n]}), \text{ with } \boldsymbol{z}_f^{[n]} = \boldsymbol{W}_f \boldsymbol{h}^{[n-1]} + \boldsymbol{U}_f \boldsymbol{x}^{[n]} + \boldsymbol{b}_f. \tag{287}$$

The weights associated with the hidden state and the cell state are $\boldsymbol{W}_f$ and $\boldsymbol{U}_f$, respectively; the bias vector of the forget gate is denoted by $\boldsymbol{b}_f$. The forget gate determines which and to what extent components of the previous cell state $\boldsymbol{c}^{[n-1]}$ are to be kept subsequently. Knowing which information of the cell state to keep, the next step is to determine how to update the cell state. For this purpose, the hidden state $\boldsymbol{h}^{[n-1]}$ and the input $\boldsymbol{x}^{[n]}$ are input to a linear layer with a hyperbolic tangent as activation function, called the *external input gate*, and the general expression Eq. (286), with $k = n$, becomes (Figure 81)

$$\alpha = g \Rightarrow \mathcal{F}_g^{[n]} = \mathcal{A}_g(\boldsymbol{z}_g^{[n]}) \to \boldsymbol{g}^{[n]} = \tanh(\boldsymbol{z}_g^{[n]}), \text{ with } \boldsymbol{z}_g^{[n]} = \boldsymbol{W}_g \boldsymbol{h}^{[n-1]} + \boldsymbol{U}_g \boldsymbol{x}^{[n]} + \boldsymbol{b}_g. \tag{288}$$

Again, $\boldsymbol{W}_g$ and $\boldsymbol{U}_g$ are linear weights and $\boldsymbol{b}_g$ represents the bias. The output vector of the $\tanh$ layer, which is also referred to as *cell gate*, can be regarded as a candidate for updates to the cell state.

The actual updates are determined by a component-wise multiplication of the candidate values $\boldsymbol{g}^{[n]}$ with the *input gate*, which has the same structure as the forget gate but has its own parameters (i.e., $\boldsymbol{W}_i$, $\boldsymbol{U}_i$, $\boldsymbol{b}_i$), and the general expression Eq. (286) becomes (Figure 81)

$$\alpha = i \Rightarrow \mathcal{F}_i^{[n]} = \mathcal{A}_i(\boldsymbol{z}_i^{[n]}) \to \boldsymbol{i}^{[n]} = \mathfrak{s}(\boldsymbol{z}_i^{[n]}), \text{ with } \boldsymbol{z}_i^{[n]} = \boldsymbol{W}_i \boldsymbol{h}^{[n-1]} + \boldsymbol{U}_i \boldsymbol{x}^{[n]} + \boldsymbol{b}_i . \tag{289}$$

The new cell state $\boldsymbol{c}^{[n]}$ is formed by summing the scaled (by the forget gate) values of the previous cell state and scaled (by the input gate) values of the candidate values,

$$\boldsymbol{c}^{[n]} = \boldsymbol{f}^{[n]} \odot \boldsymbol{c}^{[n-1]} + \boldsymbol{i}^{[n]} \odot \boldsymbol{g}^{[n]} , \tag{290}$$

where the component-wise multiplication of matrices, which is also known by the name Hadamard product, is indicated by a "$\odot$".

Remark 7.5. The path of the cell state $\boldsymbol{c}^{[k]}$ is reminiscent of the identity map that jumps over some layers to create a residual map inside the jump in the building block of a residual network; see Figure 44 and also Remark 4.7 on the identity map in residual networks.   ∎

Finally, the hidden state of the LSTM cell is computed from the cell state $\boldsymbol{c}^{[n]}$. The cell state is first squashed into the range $(-1, +1)$ by a hyperbolic tangent $\tanh$, i.e., the general expression Eq. (286) becomes (Figure 81)

$$\alpha = s \Rightarrow \mathcal{F}_s^{[n]} = \mathcal{A}_s(\boldsymbol{z}_s^{[n]}) = \mathcal{A}_s(\boldsymbol{c}^{[n]}) = \tanh \boldsymbol{c}^{[n]} , \text{ with } \boldsymbol{z}_s^{[n]} \equiv \boldsymbol{c}^{[n]} , \tag{291}$$

before the result $\mathcal{F}_s^{[n]}$ is multiplied with the output of a third sigmoid gate, i.e., the *output gate*, for which the general expression Eq. (286) becomes (Figure 81)

$$\alpha = \mathcal{O} \Rightarrow \mathcal{F}_\mathcal{O}^{[n]} = \mathcal{A}_\mathcal{O}(\boldsymbol{z}_\mathcal{O}^{[n]}) \to \boldsymbol{o}^{[n]} = \mathfrak{s}(\boldsymbol{z}_o^{[n]}), \text{ with } \boldsymbol{z}_o^{[n]} = \boldsymbol{W}_o \boldsymbol{h}^{[n-1]} + \boldsymbol{U}_o \boldsymbol{x}^{[n]} + \boldsymbol{b}_o . \tag{292}$$

Hence, the output, i.e., the new hidden state $\boldsymbol{h}^{[n]}$, is given by

$$\boldsymbol{h}^{[n]} = \mathcal{F}_\mathcal{O}^{[n]} \odot \mathcal{F}_s^{[n]} = \boldsymbol{o}^{[n]} \odot \tanh \boldsymbol{c}^{[n]}. \tag{293}$$

For the LSTM cell, we get an intuition for respective choice of the activation function. The hyperbolic tangent is used to normalize and center information that is to be incorporated into the cell state or the hidden state. The forget gate, input gate and output gate make use of the sigmoid function, which takes values between 0 and 1, to either discard information or allow information to pass by.
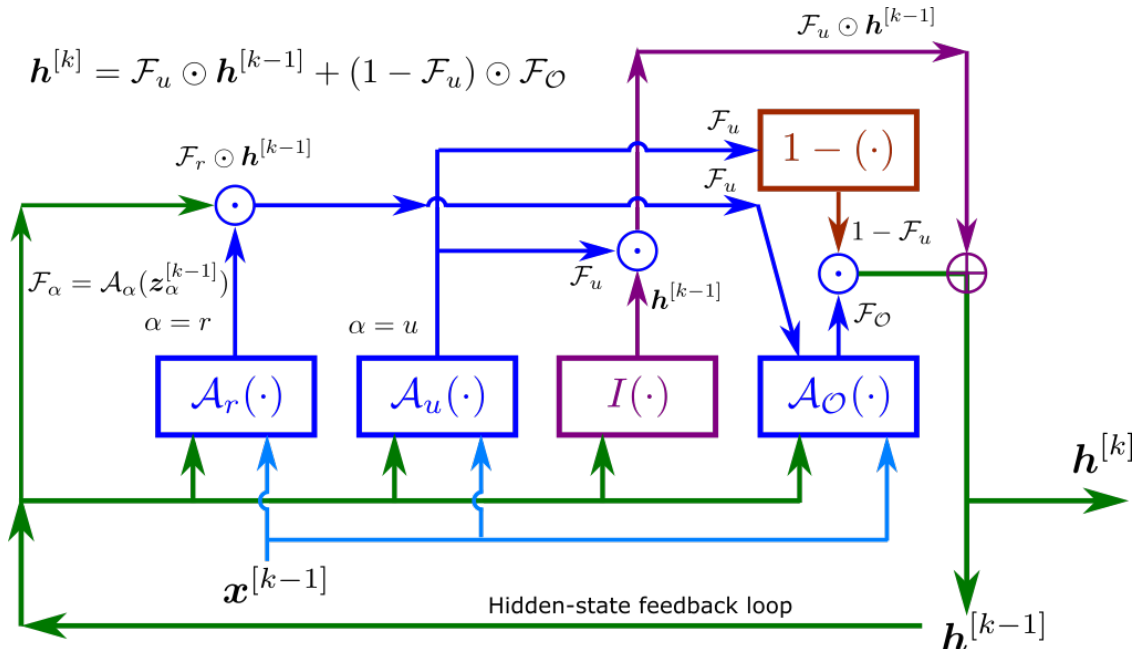
Figure 83: *Folded RNN with Gated Recurrent Unit (GRU)* (Section 7.3). The cell state at $[k-1]$, i.e., $(\boldsymbol{x}^{[k-1]}, \boldsymbol{h}^{[k-1]})$ are inputs to produce the hidden state $\boldsymbol{h}^{[k]}$. One feedback loop for the hidden state $\boldsymbol{h}$, with one-step delay $[k-1]$. The key unified recurring relation is $\mathcal{F}_\alpha = \mathcal{A}_\alpha(\boldsymbol{z}_\alpha^{[k-1]})$, with $\alpha \in \{r \text{ (reset)}, u \text{ (update)}, \mathcal{O} \text{ (Output)}\}$, where $\mathcal{A}_\alpha$ is a logistic sigmoi activation function, and $\boldsymbol{z}_\alpha^{[k-1]}$ is a linear combination of some inputs with weights plus biases at cell state $[k-1]$. Compare to the LSTM cell in Figure 81.

7.3 Gated Recurrent Unit (GRU)

In a unified manner, the various relations in the GRU depicted in Figure 83 can be expressed in a single key generic recurring-relation similar to the LSTM Eq. (286):

$$\mathcal{F}_\alpha(\boldsymbol{x}, \boldsymbol{h}) = \mathcal{A}_\alpha(\boldsymbol{z}_\alpha) \text{ with } \alpha \in \{r \text{ (reset)}, u \text{ (update)}, \mathcal{O} \text{ (Output)}\} , \tag{294}$$

where $\boldsymbol{x} = \boldsymbol{x}^{[k-1]}$ is the input at cell state $[k-1]$, $\boldsymbol{h} = \boldsymbol{h}^{[k-1]}$ the hidden variable at cell state $[k-1]$, $\mathcal{A}_\alpha$ the logistic sigmoid activation function, and $\boldsymbol{z}_\alpha = \boldsymbol{z}_\alpha^{[k-1]}$ a linear combination of some inputs with weights plus biases at cell state $[k-1]$.

To facilitate a direct comparison between the GRU cell and the LSTM cell, the locations of the boxes (gates) in the GRU cell in Figure 83 are identical to those in the LSTM in Figure 81. It can be observed that in the GRU cell (1) There is no feedback loop for the cell state, (2) The input is $\boldsymbol{x}^{[k-1]}$ (instead of $\boldsymbol{x}^{[k]}$), (3) The *reset gate* replaces the LSTM forget gate, (4) The *update gate* replaces the LSTM input gate, (5) The *identity map* (no effect on $\boldsymbol{h}^{[k-1]}$) replaces the LSTM external input gate, (6) The complement of the update gate, i.e., $(1 - \mathcal{F}_u)$ replaces the LSTM state activation $\mathcal{A}_s = \tanh$. There are fewer activations in the GRU cell compared to the LSTM cell.

The GRU was introduced in [220], and tested against LSTM and $\tanh$-RNN in [221], with concise GRU schematics, and thus not easy to follow, unlike Figure 83. The GRU relations below follow [78], p. 400.

The hidden variable $\boldsymbol{h}^{[n]}$, with $k = n$, is obtained from the GRU cell state at $[n-1]$, including the input $\boldsymbol{x}^{[n-1]}$, and is a convex combination of $\boldsymbol{h}^{[n-1]}$ and the GRU *output-gate* effect $\mathcal{F}_\mathcal{O}^{[n-1]}$, using the GRU

update-gate effect $\mathcal{F}_u^{[n-1]}$ as coefficient

$$\boldsymbol{h}^{[n]} = \mathcal{F}_u^{[n-1]} \odot \boldsymbol{h}^{[n-1]} + (1 - \mathcal{F}_u^{[n-1]}) \odot \mathcal{F}_{\mathcal{O}}^{[n-1]} \ . \tag{295}$$

For the GRU *update-gate* effect $\mathcal{F}_u^{[n-1]}$, the generic relation Eq. (294) becomes (Figure 83, where the superscript $[k-1]$ on $\mathcal{F}_\alpha$ was omitted to alleviate the notation), with $\alpha = u$

$$\mathcal{F}_u^{[n-1]} = \mathcal{A}_u(\boldsymbol{z}_u^{[n-1]}) \to \boldsymbol{u}_{[n-1]} = \mathfrak{s}(\boldsymbol{z}_u^{[n-1]}), \ \text{with} \ \boldsymbol{z}_u^{[n-1]} = \boldsymbol{W}_u \boldsymbol{h}^{[n-1]} + \boldsymbol{U}_u \boldsymbol{x}^{[n-1]} + \boldsymbol{b}_u \ . \tag{296}$$

For the GRU *output-gate* effect $\mathcal{F}_{\mathcal{O}}^{[n-1]}$, the generic Eq. (294) becomes (Figure 83), with $\alpha = \mathcal{O}$,

$$\mathcal{F}_{\mathcal{O}}^{[n-1]} = \mathcal{A}_{\mathcal{O}}(\boldsymbol{z}_{\mathcal{O}}^{[k-1]}) \to \boldsymbol{o}^{[n]} = \mathfrak{s}(\boldsymbol{z}_o^{[n-1]}), \ \text{with} \ \boldsymbol{z}_o^{[n-1]} = \boldsymbol{W}_o(\mathcal{F}_r^{[n-1]} \odot \boldsymbol{h}^{[n-1]}) + \boldsymbol{U}_o \boldsymbol{x}^{[n-1]} + \boldsymbol{b}_o. \tag{297}$$

For the GRU *reset-gate* effect $\mathcal{F}_r^{[n-1]}$, the generic Eq. (294) becomes (Figure 83), with $\alpha = r$

$$\mathcal{F}_r^{[n-1]} = \mathcal{A}_r(\boldsymbol{z}_r^{[n-1]}) \to \boldsymbol{r}^{[n-1]} = \mathfrak{s}(\boldsymbol{z}_r^{[n-1]}), \ \text{with} \ \boldsymbol{z}_r^{[n-1]} = \boldsymbol{W}_r \boldsymbol{h}^{[n-1]} + \boldsymbol{U}_r \boldsymbol{x}^{[n-1]} + \boldsymbol{b}_r \ . \tag{298}$$

Remark 7.6. GRU has fewer activation functions compared to LSTM, and is thus likely to be more efficient, even though it was stated in [221] that no concrete conclusion could be made as to "which of the two gating units was better." See Remark 7.7 on the use of GRU to solve hyperbolic problems with shock waves. ∎

7.4 Sequence modeling, attention mechanisms, Transformer

RNN and LSTM have been well-established for use in sequence modeling and "transduction" problems such as language modeling and machine translation. Attention mechanism, introduced in [57] [222], allowed for "modeling of dependencies without regard to their distance in the input and output sequences," and has been used together with RNNs [31]. Transformer is a much more efficient architecture that only uses an attention mechanism, but without the RNN architecture, to "draw global dependencies between input and output." Each of these concepts is discussed in detail below.

7.4.1 Sequence modeling, encoder-decoder

The term *neural machine translation* describes the approach of using a single neural network to translate a sentence.[220] Machine translation is a special kind of sequence-to-sequence modeling problem, in which a *source sequence* is "translated" into a *target sequence*. Neural machine translation typically relies on *encoder–decoder* architectures.[221]

The encoder network converts (encodes) the essential information of the input sequence $\boldsymbol{\mathcal{X}} = \{\boldsymbol{x}^{[k]}, k = 1, \ldots, n\}$ into an intermediate, typically fixed-length vector representation, which is also referred to as *context*. The decoder network subsequently generates (decodes) the output sequence $\boldsymbol{\mathcal{Y}} = \{\boldsymbol{y}^{[k]}, k = 1, \ldots, n\}$ from the encoded intermediate vector. The intermediate context vector in the encoder–decoder structure allows for input and output sequences of different length. Consider, for instance, an RNN composed from LSTM cells (Section 7.2) as encoder network that generates the intermediate vector by sequentially processing the elements (words, characters) of the input sequence (sentence). The decoder is a second RNN that accepts the fixed-length intermediate vector generated by the encoder, e.g., as the initial hidden state, and subsequently generates the output sequence (sentences, words) element by element (words, characters).[222]

To make things clearer, we briefly sketch the structure of a typical RNN encoder-decoder model following [57]. The encoder $\mathcal{E}$ is composed from a recurrent neural network $\mathbb{f}$ and some non-linear feedforward

---

[220] See, e.g., [57].

[221] *Autoencoders* are a special kind of encoder-decoder networks, which are trained to reproduce the input sequence, see Section 12.4.3.

[222] For more details on different types of RNNs, see, e.g., [78], p. 385, Section 10.4 "Encoder-decoder sequence-to-sequence architectures."

function $\mathbb{e}$. In terms of our notation, let $\boldsymbol{x}^{[k]}$ denote the k-th element in the sequence of input vectors, $\boldsymbol{\mathcal{X}}$, and let $\boldsymbol{h}^{[k-1]}$ denote the corresponding hidden state at time $k-1$, the hidden state at time k follows from the recurrence relation (Figure 79)

$$\boldsymbol{h}^{[k]} = \mathbb{f}(\boldsymbol{h}^{[k-1]}, \boldsymbol{x}^{[k]}). \tag{299}$$

In Section 7.1, we referred to $\mathbb{f}$ as the transition function (denoted by f in Figure 79). The context vector $\boldsymbol{c}$ is generated by another generally non-linear function $\mathbb{e}$, which takes the sequence of hidden states $\boldsymbol{x}^{[k]}$, $k = 1, \ldots, n$ as input:

$$\boldsymbol{c} = \mathbb{e}(\{\boldsymbol{h}^{[k]}, k = 1, \ldots, n\}). \tag{300}$$

Note that $\mathbb{e}$ could as well be a function of just the final hidden state in the encoder-RNN.[223]

From a probabilistic point of view, the joint probability of the entire output sequence (i.e., the "translation") $\mathbf{y}$ can be decomposed into conditional probabilities of the k-th output item $\boldsymbol{y}^{[k]}$ given its predecessors $\boldsymbol{y}^{[1]}, \ldots, \boldsymbol{y}^{[k-1]}$ and the input sequence $\mathbf{x}$, which in term can be approximated using the context vector $\boldsymbol{c}$:

$$P(\boldsymbol{y}^{[1]}, \ldots, \boldsymbol{y}^{[n]}) = \prod_{k=1}^{n} P(\boldsymbol{y}^{[k]} \mid \boldsymbol{y}^{[1]}, \ldots, \boldsymbol{y}^{[k-1]}, \mathbf{x}) \approx \prod_{k=1}^{n} P(\boldsymbol{y}^{[k]} \mid \boldsymbol{y}^{[1]}, \ldots, \boldsymbol{y}^{[k-1]}, \boldsymbol{c}). \tag{301}$$

Accordingly, the decoder is trained to predict the next item (word, character) in the output sequence $\boldsymbol{y}^{[k]}$ given the previous items $\boldsymbol{y}^{[1]}, \ldots, \boldsymbol{y}^{[k-1]}$ and the context vector $\boldsymbol{c}$. In analogy to the encoder $\mathcal{E}$, the decoder $\mathcal{D}$ comprises a recurrence function $\mathbb{g}$ and a non-linear feedforward function $\mathbb{d}$. Practically, RNNs provide an intuitive means to realize functions of variable-length sequences, since the current hidden state in RNNs contains information of all previous inputs. Accordingly, the decoder's hidden state $\boldsymbol{s}^{[k]}$ at step k follows from the recurrence relation $\mathbb{g}$ as

$$\boldsymbol{s}^{[k]} = \mathbb{g}(\boldsymbol{s}^{[k-1]}, \boldsymbol{y}^{[k-1]}, \boldsymbol{c}). \tag{302}$$

To predict the conditional probability of the next item $\boldsymbol{y}^{[k]}$ by means of the function $\mathbb{d}$, the decoder can therefore use only the previous item $\boldsymbol{y}^{[k-1]}$ and the current hidden state $\mathbf{s}^{[k]}$ as inputs (along with the context vector $\boldsymbol{c}$) rather than all previously predicted items $\boldsymbol{y}^{[1]}, \ldots, \boldsymbol{y}^{[k-1]}$:

$$P(\boldsymbol{y}^{[k]} \mid \boldsymbol{y}^{[1]}, \ldots, \boldsymbol{y}^{[k-1]}, \boldsymbol{c}) = \mathbb{d}(\boldsymbol{y}^{[k-1]}, \mathbf{s}^{[k]}, \boldsymbol{c}). \tag{303}$$

We have various choices of how the context vector $\boldsymbol{c}$ and inputs $\boldsymbol{y}^{[k]}$ are fed into to the decoder-RNN.[224] The context vector $\boldsymbol{c}$, for instance, can either be used as the decoder's initial hidden state $\boldsymbol{s}^{[1]}$ or, alternatively, as the first input.

7.4.2 Attention

As the authors of [57] emphasized, the encoder *"needs to be able to compress all the necessary information of a source sentence into fixed-length vector"*. For this reason, long sentences pose a challenge in neural machine translation, in particular, as sentences to be translated are longer than the sentences networks have seen during training, which was confirmed by the observations in [223]. To cope with long sentences, an encoder–decoder architecture, *"which learns to align and translate jointly,"* was proposed in [57]. Their approach is motivated by the observation that individual items of the target sequence correspond to different parts of the source sequence. To account for the fact that only a subset of the source sequence is relevant when generating a new item of the target sequence, two key ingredients (alignment and translation) to the conventional encoder–decoder architecture described above were introduced in [57], and will be presented below.

---

[223] See, e.g., [78], p. 385.
[224] See, e.g., [78], p. 386.

☛ The *first key ingredient* to their concept of "alignment" is the idea of using a distinct context vector c_k for each output item $y^{[k]}$ instead of a single context c. Accordingly, the recurrence relation of the decoder, Eq. (302), is modified and takes the context c_k as argument

$$s^{[k]} = g(s^{[k-1]}, y^{[k-1]}, c_k), \tag{304}$$

as is the conditional probability of the output items, Eq. (303),

$$P(y^{[k]} \mid y^{[1]}, \ldots, y^{[k-1]}, \mathbf{x}) = P(y^{[k]} \mid y^{[1]}, \ldots, y^{[k-1]}, c_k) \approx \mathrm{d}(y^{[k-1]}, s^{[k]}, c_k), \tag{305}$$

i.e., it is conditioned on distinct context vectors c_k for each output $y^{[k]}$, $k = 1, \ldots, n$.

The k-th context vector c_k is supposed to capture the information of that part of the source sequence $\mathbf{x}$ which is relevant to the k-th target item $y^{[k]}$. For this purpose, c_k is computed as weighted sum of all hidden states $h^{[l]}$, $l = 1, \ldots, n$ of the encoder:

$$c_k = \sum_{l=1}^{n} \alpha_{kl} h^{[l]}. \tag{306}$$

The k-th hidden state of a conventional RNN obeying the recurrence given by Eq. (299) only includes information about the preceding items $(1, \ldots, k)$ in the source sequence, since the remaining items $(k + 1, \ldots, n)$ still remain to be processed. When generating the k-th output item, however, we want information about all source items, before *and* after, to be contained in the k-th hidden state $h^{[k]}$.

For this reason, using a *bidirectional RNN*[225] as encoder was proposed in [57]. A bidirectional RNN combines two RNNs, i.e., a *forward RNN* and *backward RNN*, which independently process the source sequence in the original and in reverse order, respectively. The two RNNs generate corresponding sequences of forward and backward hidden states,

$$h_{\mathrm{fwd}}^{[k]} = \mathbb{f}_{\mathrm{fwd}}(h_{\mathrm{fwd}}^{[k-1]}, x^{[k]}), \qquad h_{\mathrm{rev}}^{[k]} = \mathbb{f}_{\mathrm{rev}}(h_{\mathrm{rev}}^{[k-1]}, x^{[n-k]}). \tag{307}$$

In each step, these vectors are concatenated to a single hidden state vector $h^{[k]}$, which the authors of [57] refer to as "*annotation*" of the k-th source item:

$$h^{[k]} = \left[\left(h_{\mathrm{fwd}}^{[k]} \right)^T, \left(h_{\mathrm{rev}}^{[k]} \right)^T \right]^T. \tag{308}$$

They mentioned *"the tendency of RNNs to better represent recent inputs"* as reason why the annotation $h^{[k]}$ focuses around the k-th encoder input $x^{[k]}$.

☛ As the *second key ingredient*, the authors of [57] proposed a so-called *"alignment model"*, i.e., a function $\mathfrak{a}$ to compute weights α_{kl} needed for the context c_k, Eq. (306),

$$a_{kl} = \mathfrak{a}(s^{[k-1]}, h^{[l]}), \tag{309}$$

which is meant to quantify ("score") the relation, i.e., the *alignment*, between the k-th decoder output (target) and inputs *"around"* the l-th position of the source sequence. The score is computed from the decoder's hidden state of the previous output $s^{[k-1]}$ and the annotation $h^{[l]}$ of the l-th input item, so it *"reflects the importance of the annotation $h^{[l]}$ with respect to the previous hidden state $s^{[k-1]}$ in deciding the next state $s^{[k]}$ and generating $y^{[k]}$."*

The alignment model is represented by a feedforward neural network, which is jointly trained along with all other components of the encoder–decoder architecture. The weights of the annotations, in turn, follow from the alignment scores upon exponentiation and normalization (through $\mathrm{softmax}(\cdot)$ (Section 5.1.3) along the second dimension):

$$\alpha_{kl} = \frac{\exp(a_{kl})}{\sum_{j=1}^{n} \exp(a_{kj})}. \tag{310}$$

---

[225] See [224].

The weighting in Eq. (306) is interpreted as a way *"to compute an expected annotation, where the expectation is over possible alignments"* [57]. From this perspective, α_{kl} represents the probability of an output (target) item $\boldsymbol{y}^{[k]}$ being aligned to an input (source) item $\boldsymbol{x}^{[l]}$.

In neural machine translation it was possible to show that the attention model in [57] significantly outperformed conventional encoder–decoder architectures, which encoded the entire source sequence into a single fixed-length vector. In particular, this proposed approach turned out to perform better in translating long sentences, where it could achieve performance on par with phrase-based statistical machine translation approaches of that time.

7.4.3 Transformer architecture

Despite improvements, the attention model in [57] shared the fundamental drawback intrinsic to all RNN-based models: The sequential nature of RNNs is adverse to parallel computing making training less efficient as with, e.g., feed-forward or convolutional neural networks, which lend themselves to massive parallelization. To overcome this drawback, a novel model architecture, which entirely dispenses with recurrence, was proposed in [31]. As the title *"Attention Is All You Need"* already reveals, their approach to neural machine translation is exclusively based on the concept of attention (and some feedforward-layers), which is repeatedly used in the proposed architecture referred to as *"Transformer"*.

In what follows, we describe the individual components of the Transformer architecture, see Figure 85. Among those, the *scaled dot-product attention*, see Figure 84, is a fundamental building block. Scaled-dot product attention, which is represented by a function $\mathscr{A}$, compares a query vector $\boldsymbol{q} \in \mathbb{R}^{d_k}$ and a set of m key vectors $\boldsymbol{k}_i \in \mathbb{R}^{d_k}$ to determine the weighting of value vectors $\boldsymbol{v}_i \in \mathbb{R}^{d_v}$ corresponding to the keys. As opposed to the additive alignment model used in [57], see Eq. (309), scaled-dot product attention combines query and key vectors in a multiplicative way.

Let $\boldsymbol{K} = [\boldsymbol{k}_1, \ldots, \boldsymbol{k}_m]^T \in \mathbb{R}^{m \times d_k}$ and $\boldsymbol{V} = [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m]^T \in \mathbb{R}^{m \times d_v}$ denote key and value matrices formed from the individual vectors, where query and key vectors share the dimension d_k and value vectors are d_v-dimensional. The attention model produces a context vector $\boldsymbol{c} \in \mathbb{R}^{d_v}$ by weighting the values $\boldsymbol{v}_i$, $i = 1, \ldots, m$, according to the multiplicative alignment of the query $\boldsymbol{q}$ with keys $\boldsymbol{k}_i$, $i = 1, \ldots, m$:

$$\boldsymbol{c} = \mathscr{A}(\boldsymbol{q}, \boldsymbol{K}, \boldsymbol{V}) = \frac{1}{\sqrt{d_k}} \sum_{i=1}^{m} \frac{\exp(\boldsymbol{q} \cdot \boldsymbol{k}_i)}{\sum_{j=1}^{m} \exp(\boldsymbol{q} \cdot \boldsymbol{k}_j)} \boldsymbol{v}_i = \frac{1}{\sqrt{d_k}} \operatorname{softmax}\left(\boldsymbol{q}^T \boldsymbol{K}^T\right) \boldsymbol{V}. \tag{311}$$

Scaling with the square root of the query/key dimension is supposed to prevent pushing the $\operatorname{softmax}(\cdot)$ function to regions of small gradients for large d_k as scalar products grow with the dimension of queries and keys. The above attention model can be simultaneously applied to multiple queries. For this purpose, let $\boldsymbol{Q} = [\boldsymbol{q}_1, \ldots, \boldsymbol{q}_k]^T \in \mathbb{R}^{k \times d_k}$ and $\boldsymbol{C} = [\boldsymbol{c}_1, \ldots, \boldsymbol{c}_k]^T \in \mathbb{R}^{k \times d_v}$ denote matrices of query and context vectors, respectively. We can rewrite the attention model using matrix multiplication as follows:

$$\boldsymbol{C} = \mathscr{A}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \frac{1}{\sqrt{d_k}} \operatorname{softmax}\left(\boldsymbol{Q}\boldsymbol{K}^T\right) \boldsymbol{V} \in \mathbb{R}^{k \times d_k}. \tag{312}$$

Note that $\boldsymbol{Q}\boldsymbol{K}^T$ gives a $k \times m$ matrix, for which the $\operatorname{softmax}(\cdot)$ is computed along the second dimension.

Based on the concept of scaled dot-product attention, the idea of using multiple attention functions in parallel rather than just a single one was proposed in [31], see Figure 84. In their concept of *"Multi-Head Attention"*, each "head" represents a separate context $\boldsymbol{C}_j$ computed from scaled dot-product attention, Eq. (312), on queries $\boldsymbol{Q}_j$, keys $\boldsymbol{K}_j$ and values $\boldsymbol{V}_j$, respectively. The inputs to the individual scaled dot-product attention functions $\boldsymbol{Q}_j$, $\boldsymbol{K}_j$ and $\boldsymbol{V}_j$, $j = 1, \ldots, h$, in turn, are head-specific (learned) projections of the queries $\mathcal{Q} \in \mathbb{R}^{k \times d}$, keys $\mathcal{K} \in \mathbb{R}^{m \times d}$ and values $\mathcal{V} \in \mathbb{R}^{m \times d}$. Assuming that queries $\boldsymbol{q}_i$, $\boldsymbol{k}_i$ and $\boldsymbol{v}_i$ share the dimension d, the projections are represented by matrices $\boldsymbol{W}_j^Q \in \mathbb{R}^{d \times d_k}$, $\boldsymbol{W}_j^K \in \mathbb{R}^{d \times d_k}$ and

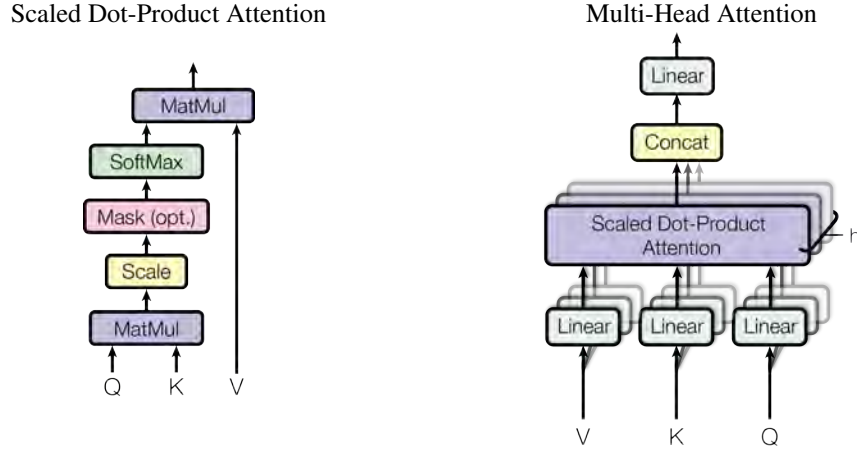Scaled Dot-Product Attention

Multi-Head Attention

Figure 84: *Scaled dot-product attention and multi-head attention* (Section 7.4.3). Scaled-dot product attention (left) is the elementary building block of the Transformer model. It compares query vectors (Q) against a set of key vectors (K) to produce a context vector by weighting to value vectors (V) that correspond to the keys. For this purpose, $\mathrm{softmax}(\cdot)$ function is applied to the inner product (MatMul) of the query and key vectors (scaled by a constant). The output of the $\mathrm{softmax}(\cdot)$ represent the weighting by which value vectors are scaled taking the inner product (MatMul), see Eq.(311). To prevent attention functions of the decoder, which generates the output sequence item by item, from using future items of the output sequence, a masking layer is introduced. By the masking, all scores beyond the current time/position are set to $-\infty$. Multi-head attention (right) combines several (h) scaled-dot product attention functions in parallel, each of which is referred to as "head". For this purpose, queries, keys and values are projected by means of a head-specific linear layers (Linear), whose outputs are input to the individual scaled dot-product attention functions, see Eq.(313). The context vectors produced by each head are concatenated before being fed into one more linear layer, see Eq.(314). (Figure reproduced with permission of the authors.)

$\boldsymbol{W}_j^V \in \mathbb{R}^{d \times d_v}$:[226]

$$Q_j = \mathcal{Q}\boldsymbol{W}_j^Q \in \mathbb{R}^{k \times d_k}, \qquad K_j = \mathcal{K}\boldsymbol{W}_j^K \in \mathbb{R}^{m \times d_k}, \qquad \boldsymbol{V}_j = \mathcal{V}\boldsymbol{W}_j^V \in \mathbb{R}^{m \times d_k}. \qquad (313)$$

Multi-head attention combines the individual "heads" $\boldsymbol{C}_i$ through concatenation (along the second dimension) and subsequent projection by means of $\boldsymbol{W}^O \in \mathbb{R}^{d_v h \times d}$,

$$\mathcal{H} = \mathscr{A}_{\mathrm{mh}}(\mathcal{Q}, \mathcal{K}, \mathcal{V}) = [\boldsymbol{C}_1, \ldots, \boldsymbol{C}_h]\boldsymbol{W}^O \in \mathbb{R}^{k \times d}, \qquad \boldsymbol{C}_i = \mathscr{A}(\boldsymbol{Q}_i, \boldsymbol{K}_i, \boldsymbol{V}_i), \qquad (314)$$

where h denotes the number of heads, i.e., individual scaled-dot product attention functions used in parallel. Note that the output of multi-head attention $\mathcal{H}$ has the same dimensions of as the input queries $\mathcal{Q}$, i.e., $\mathcal{H}, \mathcal{Q} \in \mathbb{R}^{k \times d}$.

To understand why the projection is essential in the Transformer architecture, we shift our attention (no pun intended) to the encoder-structure illustrated in Figure 85. The encoder combines a stack of N identical layers, which, in turn, are composed from two sub-layers each. To stack the layers without further projection, all inputs and outputs of the encoder layers share the same dimension. The same holds true for each of the sub-layers, which are also designed to preserve the input dimensions. The first sub-layer is a multi-head *self-attention* function, in which the input sequence attends to itself. The concept of self-attention

---

[226] Unlike the use of notation in [31], we use different symbols for the arguments of the scaled-dot product attention function, Eq. (312), and those of the multi-head attention, Eq. (314), to emphasize their distinct dimensions.
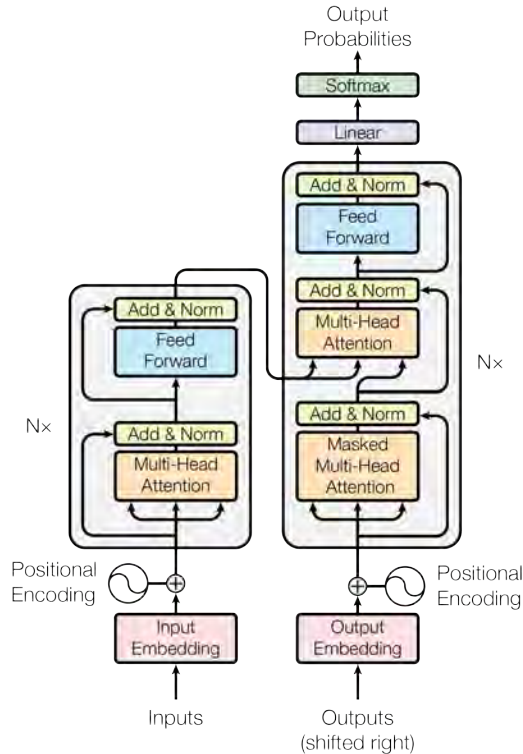
Figure 85: *Transformer architecture* (Section 7.4.3). The Transformer is a sequence-to-sequence model without recurrent connections. Encoder and decoder are entirely built upon scaled dot-product attention. Items of source and target sequences are numerically represented as vectors, i.e., embeddings. Positional encodings furnish embeddings with information on their positions within the respective sequences. The encoder stack comprises N layers, each of which is composed from two sub-layers: The first sub-layer is a multi-head attention function used as a *self-attention*, by which relations among the items of one and the same sequence are learned. The second sub-layer is a *position-wise* fully-connected network. The decoder stack is also built from N layers consisting of three sub-layers. The first and third sub-layers are identical to the encoder except for masking in the self-attention function. The second sub-layer is a multi-head attention function using the encoder's output as keys and values to relate items of the source and target sequences. (Figure reproduced with permission of the authors.)

is based on the idea to relate different items of a single sequence to generate a representation of the input, i.e., *"each position in the encoder can attend to all positions in the previous layer of the encoder."*

In the context of multi-head attention, self-attention implies that one and the same sequence multiply serves as queries, keys and values, respectively. Let $\boldsymbol{\mathcal{X}} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n] \in \mathbb{R}^{n \times d}$ denote the input to the self-attention (sub-)layer and $\boldsymbol{\mathcal{H}} \in \mathbb{R}^{n \times d}$ the corresponding output, where n is the length of the sequence and d is the dimension of single items (*"positions"*), self-attention can be expressed in terms of the multi-head attention function as

$$\boldsymbol{\mathcal{H}} = \mathscr{A}_{\mathrm{self}}(\boldsymbol{\mathcal{X}}) = \mathscr{A}_{\mathrm{mh}}(\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{X}}) \in \mathbb{R}^{n \times d}. \tag{315}$$

The authors of [31] introduced a residual connection around the self-attention (sub-)layer, which, in view of the same dimensions of heads and queries, reduces to a simple addition.

To prevent values from growing upon summation, the residual connection is followed by *layer normal-*

ization as proposed in [225], which scales the input to zero mean and unit variance:

$$\mathcal{N}(\boldsymbol{\mathcal{T}}) = \frac{1}{\sigma}\left(\boldsymbol{\mathcal{T}} - \mu\mathbb{I}\right), \tag{316}$$

where μ and σ denote the mean value and the standard deviation of the input components; $\mathbb{I}$ is a matrix of ones with the same dimension as the input $\boldsymbol{\mathcal{T}}$. The normalized output of the encoder's first sub-layer therefore follows from the sum of inputs to self-attention function $\boldsymbol{\mathcal{X}}$ and its outputs $\boldsymbol{\mathcal{H}}$ as

$$\boldsymbol{\mathcal{Z}} = \mathcal{N}(\boldsymbol{\mathcal{X}} + \boldsymbol{\mathcal{H}}) \in \mathbb{R}^{n\times d}. \tag{317}$$

The output of the first sub-layer is the input to the second sub-layer within the encoder stack, i.e., a *"position-wise feed-forward network."* Position-wise means that a fully connected feedforward network (see Section 4 on "Static, feedforward networks"), which is subsequently represented by the function $\mathcal{F}$, is applied to each item (i.e., *"position"*) of the input sequence $\boldsymbol{z}_i \in \boldsymbol{\mathcal{Z}} = [\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n]^T \in \mathbb{R}^{n\times d}$ *"separately and identically."* In particular, a network with a single hidden layer and a linear rectifier unit (see Section 5.3.2 on "Rectified linear function (ReLU)") as activation function was used in [31] to compute the vectors $\boldsymbol{y}_i$:[227]

$$\boldsymbol{y}_i = \mathcal{F}(\boldsymbol{z}_i) = \boldsymbol{W}_2 \max(\boldsymbol{0}, \boldsymbol{W}_1\boldsymbol{z}_i + \boldsymbol{b}_1) + \boldsymbol{b}_2 \in \mathbb{R}^d, \tag{318}$$

where $\boldsymbol{W}_1 \in \mathbb{R}^{h\times d_{ff}}$, $\boldsymbol{W}_2 \in \mathbb{R}^{d_{ff}\times h}$ denote the connection weights and $\boldsymbol{b}_1 \in \mathbb{R}^h$ and $\boldsymbol{b}_2 \in \mathbb{R}^d$ are bias vectors. The individual outputs $\boldsymbol{y}_i$, which correspond to the respective items $\boldsymbol{x}_i$ of the input sequence $\boldsymbol{\mathcal{X}}$, form the matrix $\boldsymbol{\mathcal{Y}} = [\boldsymbol{y}_1, \ldots, \boldsymbol{y}_n]^T \in \mathbb{R}^{n\times d}$.

As for the first sub-layer, the authors of [31] introduced a residual connection followed by layer normalization around the feedforward network. The ouput of the encoder's second sublayer, which, at the same time, is the output of the encoder layer, is given by

$$\boldsymbol{\mathcal{E}} = \mathcal{N}(\boldsymbol{\mathcal{Z}} + \boldsymbol{\mathcal{Y}}) \in \mathbb{R}^{n\times d}. \tag{319}$$

Within the transformer architecture (Figure 85), the encoder is composed from N encoder layers, which are "stacked", i.e., the output of the ℓ-th layer is input to the subsequent layer $\boldsymbol{\mathcal{X}}^{(\ell+1)} = \boldsymbol{\mathcal{E}}^{(\ell)}$. Let $\mathcal{E}^{(\ell)}$ denote the ℓ-th encoder layer composed from the layer-specific self-attention function $\mathcal{A}_{\text{self}}^{(\ell)}$, which takes $\boldsymbol{\mathcal{X}}^{(\ell)} \in \mathbb{R}^{n\times d}$ as input, and the layer-specific feedforward network $\mathcal{F}^{(\ell)}$, the layer's output $\boldsymbol{\mathcal{E}}^{(\ell)}$ is computed as follows:

$$\boldsymbol{\mathcal{E}}^{(\ell)} = \mathcal{E}^{(\ell)}(\boldsymbol{\mathcal{X}}^{(\ell)}) = \mathcal{N}(\mathcal{F}^{(\ell)}(\mathcal{N}(\boldsymbol{\mathcal{X}}^{(\ell)} + \mathcal{A}_{\text{self}}^{(\ell)}(\boldsymbol{\mathcal{X}}^{(\ell)}))) + \mathcal{N}(\boldsymbol{\mathcal{X}}^{(\ell)} + \mathcal{A}_{\text{self}}^{(\ell)}(\boldsymbol{\mathcal{X}}^{(\ell)}))), \tag{320}$$

or, alternatively, using a step-wise representation,

$$\boldsymbol{\mathcal{H}}^{(\ell)} = \mathcal{A}_{\text{self}}^{(\ell)}(\boldsymbol{\mathcal{X}}^{(\ell)}) \in \mathbb{R}^{n\times d}, \tag{321}$$

$$\boldsymbol{\mathcal{Z}}^{(\ell)} = \mathcal{N}(\boldsymbol{\mathcal{X}}^{(\ell)} + \boldsymbol{\mathcal{H}}^{(\ell)}) \in \mathbb{R}^{n\times d}, \tag{322}$$

$$\boldsymbol{\mathcal{Y}}^{(\ell)} = \mathcal{F}^{(\ell)}(\boldsymbol{\mathcal{Z}}^{(\ell)}) \in \mathbb{R}^{n\times d}, \tag{323}$$

$$\boldsymbol{\mathcal{E}}^{(\ell)} = \mathcal{N}(\boldsymbol{\mathcal{Z}}^{(\ell)} + \boldsymbol{\mathcal{Y}}^{(\ell)}) \in \mathbb{R}^{n\times d}. \tag{324}$$

Note that inputs and outputs of all components of an encoder layer share the same dimensions, which facilitates several layers to be stacked without additional projections in between.

---

[227] Eq. (318) is meant to reflect the idea off *position-wise* computations of the second sub-layer, since single vectors $\boldsymbol{z}_i$ are input to the feedforward network. From the computational point of view, however, all "positions" can be processed simultaneously using the same weights and biases as in Eq. (318): $\boldsymbol{\mathcal{Y}} = \max(\boldsymbol{0}, \boldsymbol{\mathcal{Z}}\boldsymbol{W}_1^T + \boldsymbol{b}_1)\boldsymbol{W}_2^T + \boldsymbol{b}_2 \in \mathbb{R}^{n\times d}$. Note that the addition of bias vectors $\boldsymbol{b}_1$, $\boldsymbol{b}_2$ needs to be computed for all positions, i.e., they are added row-wisely to the matrix-valued projections of the layer inputs by means $\boldsymbol{W}_1^T$ and $\boldsymbol{W}_2^T$, respectively.

The decoder's structure within the transformer architecture is similar to that of the encoder, see the right part in Figure 85. As the encoder, it is composed from N identical layers, each of which combines three sub-layers (as opposed to two sub-layers in the encoder). In addition to the self-attention sub-layer (first sub-layer) and the fully-connected position-wise feed-forward network (third sub-layer), the decoder inserts a second sub-layer that *"performs multi-head attention over the output of the encoder stack"* in between. Attending to outputs of the encoder enables the decoder to relate items of the source sequence to items of the target sequence. Just as in the encoder, residual connections are introduced around each of the decoder's sub-layers.

Let $\boldsymbol{\mathcal{Y}}_1^{(\ell)} \in \mathbb{R}^{n \times d}$ denote the input to the ℓ-th decoder layer, which, for $\ell > 1$, is the output $\boldsymbol{\mathcal{D}}^{(\ell-1)} \in \mathbb{R}^{n \times d}$ of the previous layer. The output $\boldsymbol{\mathcal{D}}^{(\ell)}$ of the ℓ-th decoder layer is then obtained through the following computations. The first sub-layer performs (multi-headed) self-attention over items of the output sequence, i.e., it establishes a relation among them:

$$\boldsymbol{\mathcal{H}}_1^{(\ell)} = \mathscr{A}_{\mathrm{self}}^{(\ell)}(\boldsymbol{\mathcal{Y}}_1^{(\ell)}) \in \mathbb{R}^{n \times d}, \tag{325}$$

$$\boldsymbol{\mathcal{Z}}_1^{(\ell)} = \mathscr{N}(\boldsymbol{\mathcal{Y}}^{(\ell)} + \boldsymbol{\mathcal{H}}_1^{(\ell)}) \in \mathbb{R}^{n \times d}. \tag{326}$$

The second sub-layer relates items of the source and the target sequences by means of multi-head attention:

$$\boldsymbol{\mathcal{Y}}_2^{(\ell)} = \mathscr{A}_{\mathrm{mh}}^{(\ell)}(\boldsymbol{\mathcal{E}}^{(\ell)}, \boldsymbol{\mathcal{E}}^{(\ell)}, \boldsymbol{\mathcal{Z}}_1^{(\ell)}) \in \mathbb{R}^{n \times d}, \tag{327}$$

$$\boldsymbol{\mathcal{Z}}_2^{(\ell)} = \mathscr{N}(\boldsymbol{\mathcal{Z}}_1^{(\ell)} + \boldsymbol{\mathcal{Y}}_2^{(\ell)}) \in \mathbb{R}^{n \times d}. \tag{328}$$

The third sub-layer is a fully-connected feed-forward network (with a single hidden layer) that is applied to each element of the sequence (position-wisely)

$$\boldsymbol{\mathcal{Y}}_3^{(\ell)} = \mathscr{F}^{(\ell)}(\boldsymbol{\mathcal{Z}}_2^{(\ell)}) \in \mathbb{R}^{n \times d}, \tag{329}$$

$$\boldsymbol{\mathcal{D}}^{(\ell)} = \mathscr{N}(\boldsymbol{\mathcal{Z}}_2^{(\ell)} + \boldsymbol{\mathcal{Y}}_3^{(\ell)}) \in \mathbb{R}^{n \times d}. \tag{330}$$

The output of the last ($\ell = N$) decoder-layer within the decoder-stack is projected onto a vector that has the dimension of the "vocabulary", i.e., the set of all feasible output items. Taking the $\mathrm{softmax}(\cdot)$ over the components of the vector produces probabilities over elements of the vocabulary to be item of the output sequence (see Section 5.1.3).

A Transformer-model is trained on the complete source and target sequences, which are input to the encoder and the decoder, respectively. The target sequence is shifted right by one position, such that a special token indicating the start of a new sequence can be placed at the beginning, see Fig. 85. To prevent the decoder from attending to future items of the output sequence, the (multi-headed) self-attention sub-layer needs to be *"masked"*. The masking is realized by setting those inputs to the $\mathrm{softmax}(\cdot)$ function of the scaled dot-product attention, see Eq. (311), for which query vectors are aligned with keys that correspond to items in the output sequence beyond the respective query's position, set to $-\infty$. As loss function, the negative log-likelihood is typically used; see Section 5.1.2 on maximum likelihood (probability cost) and Section 5.1.3 on classification loss functions.

As the transformer architecture does not have recurrent connections, *positional encodings* were added to the inputs of both encoder and decoder [31], see Figure 85. The positional encodings supply the individual items of the inputs to the encoder and the decoder with information about their positions within the respective sequences. For this purpose, the authors of [31] proposed to add vector-valued positional encodings $\boldsymbol{p}_i \in \mathbb{R}^d$, which have the same dimension d as the input embedding of an item, i.e., its numerical representation as a vector. They used sine and cosine functions of an items position (i) within respective sequence, whose frequency varies (decreases) with the component index (j):

$$p_{i,2j} = \sin\theta_{ij}, \qquad p_{i,2j-1} = \cos\theta_{ij}, \qquad \theta_{ij} = \frac{j}{10000^{2i/d}}, \qquad j = 1, \ldots, d. \tag{331}$$

The trained Transformer-model produces one item of the output sequence at a time. Given the input sequence and all outputs already generated in previous steps, the Transformer predicts probabilities for the next item in the output sequence. Models of this kind are referred to as "auto-regressive".

The authors of [31] varied parameters of the Transformer model to study the importance of individual components. Their "base model" used $N = 6$ encoder and decoder layers. Inputs and outputs of all sub-layers are sequences of vectors, which have a dimension of $d = 512$. All multi-head attention functions of the Transformer, see Eq.(321), Eq.(325) and Eq.(327), have $h = 8$ heads each. Before the alignment scores are computed by each head, the dimensions of queries, keys and values were reduced to $d_k = d_v = 64$. The hidden layer of the fully-connected feedforward network, see Eq.(318), was chosen as $d_{ff} = 2048$ neurons wide. With this set of (hyper-)parameters, the Transformer model features a total of 65×10^6 parameters. The training cost (in terms of FLOPS) of the attention-based Transformer was shown to be (at least) two orders of magnitudes smaller than comparable models at that time while the performance was maintained. A refined ("big") model was able to outperform all previous approaches (based on RNNs or convolutional neural networks) in English-to-French and English-to-German translation tasks.

As of 2022, the *Generative Pre-trained Transformer 3* (GPT-3) model [226], which is based on the Transformer architecture, belongs to the most powerful language models. GPT-3 is an autoregressive model that produces text from a given (initial) text prompt, whereby it can deal with different tasks as translation, question-answering, cloze-tests[228] and word-unscrambling, for instance. The impressive capabilities of GPT-3 are enabled by its huge capacity of 175 *billion* parameters, which is 10 times more than preceding language models.

Remark 7.7. *Attention mechanism, kernel machines, physics-informed neural networks (PINNs).* In [227], a new attention architecture (mechanism) was proposed by using kernel machines discussed in Section 8, whereas in [228], the gated recurrent units (GRU, Section 7.3) and the attention mechanism (Section 7.4.1) were used in conjunction with Physics-Informed Neural Networks (PINNs, Section 9.5) to solve hyperbolic problems with shock waves; Remark 9.5 and Remark 11.11. ∎

8 Kernel machines (methods, learning)

Researchers have observed that as the number of parameters increased beyond the interpolation threshold, or as the number of hidden units in a layer (i.e., layer width) increased, the test error decreased, i.e., such network generalized well; see Figures 60 and 61. So, as a first step to try to understand why deep-learning networks work (Section 14.2 on "Lack of understanding"), it is natural to study the limniting case of infinite layer width first, since it would be relatively easier than the case of finite layer width [229]; see Figure 148.

In doing so, a connection between networks with infinite width and the kernel machines or kernel methods, was revealed [230] [231] [232]. See also the connection between kernel methods and Support Vector Machines (SVM) in Footnote 31.

"A neural network is a little bit like a Rube Goldberg machine. You don't know which part of it is really important. ... reducing [them] to kernel methods—because kernel methods don't have all this complexity—somehow allows us to isolate the engine of what's going on" [230].

Quanta Magazine described the discovery of such connection as the 2021 breakthrough in computer science [233].

Covariance functions, or covariance matrices, are kernels in Gaussian processes (Section 8.3), an important class of methods in machine learning [234] [130]. A kernel method in terms of the time variable

---

[228] A cloze is "a form of written examination in which candidates are required to provide words that have been omitted from sentences, thereby demonstrating their knowledge and comprehension of the text", see Wiktionary version 11:23, 2 January 2022.

was discussed in Section 13.2.2 in connection with the continuous temporal summation in neuroscience. Our aim here is only to provide first-time learners background material on kernel methods in terms of space variables (specifically the "Setup" in [235]) in preparation to read more advanced references mentioned in this section, such as [235] [232] [236] etc.

8.1 Reproducing kernel: General theory

A kernel $\mathcal{K}(\cdot, y)$ is called *reproducing* if its scalar (L_2) product with a function $f(y)$ reproduces the same function $f(\cdot)$ itself [237]:

$$f(x) = \langle f(y), \mathcal{K}(x, y), \rangle_y = \int f(y)\mathcal{K}(x, y)dy \tag{332}$$

where the subscript y in the scalar product $\langle \cdot, \cdot \rangle_y$ indicates the integrand.

Let $\{\phi_1(x), \ldots, \phi_n(x)\}$ be a set of linearly independent basis functions. A function $f(x)$ can be expressed in this basis as follows:

$$f(x) = \sum_{k=1}^{n} \zeta_k \phi_k(x) . \tag{333}$$

The scalar product of two functions f and g is given by:

$$g(x) = \sum_{k=1}^{n} \eta_k \phi_k(x) \Rightarrow \langle f, g \rangle = \sum_{i,j=1}^{n} \zeta_i \eta_i \langle \phi_i, \phi_j \rangle = \sum_{i,j=1}^{n} \zeta_i \eta_i \Gamma_{ij} , \tag{334}$$

where

$$\Gamma_{ij} = \langle \phi_i, \phi_j \rangle , \quad \mathbf{\Gamma} = [\Gamma_{ij}] > 0 \in \mathbb{R}^{n \times n} \tag{335}$$

is the Gram matrix,[229] which is strictly positive definite, with its inverse (also strictly positive definite) denoted by

$$\mathbf{\Delta} = \mathbf{\Gamma}^{-1} = [\Delta_{ij}] > 0 \in \mathbb{R}^{n \times n} . \tag{336}$$

Then a reproducing kernel can be written as [237][230]

$$\mathcal{K}(x, y) = \sum_{i,j=1}^{n} \Delta_{ij} \phi_i(x) \phi_j(y) = \mathbf{\Phi}^T(x) \mathbf{\Delta} \mathbf{\Phi}(y) , \text{ with } \mathbf{\Phi}^T(x) = [\phi_1(x), \ldots, \phi_n(x)] \in \mathbb{R}^{1 \times n} . \tag{337}$$

Remark 8.1. It is easy to verify that the function $\mathcal{K}(x, y)$ in Eq. (337) is a reproducing kernel:

$$\langle f(y), \mathcal{K}(y, x) \rangle_y = \langle \sum_i \zeta_i \phi_i(y), \sum_{j,k} \Delta_{jk} \phi_j(y) \phi_k(x) \rangle_y = \sum_i \zeta_i \sum_{j,k} \langle \phi_i(y), \phi_j(y) \rangle_y \Delta_{jk} \phi_k(x) \tag{338}$$

$$= \sum_i \zeta_i \sum_{j,k} \Gamma_{ij} \Delta_{jk} \phi_k(x) = \sum_i \zeta_i \sum_k \delta_{ik} \phi_k(x) = \sum_i \zeta_i \phi_i(x) = f(x) , \tag{339}$$

using Eqs. (335) and (336). ■

From Eq. (334), the norm of a function f can be defined as

$$\| f \|_{\mathcal{K}}^2 = \langle f, f \rangle = \sum_{i,j=1}^{n} \zeta_i \Gamma_{ij} \zeta_j = \boldsymbol{\zeta}^T \mathbf{\Gamma} \boldsymbol{\zeta} = \boldsymbol{\zeta}^T \mathbf{\Delta}^{-1} \boldsymbol{\zeta} , \text{ with } \boldsymbol{\zeta}^T = [\zeta_1, \ldots, \zeta_n] , \tag{340}$$

---

[229] The stiffness matrix in the displacement finite element method is a Gram matrix.
[230] See Eq. (6), p. 346, in [237].

where $\| \cdot \|_{\mathcal{K}}$ denotes the norm induced by the kernel $\mathcal{K}$, and where the matrix notation in Eq. (7) was used.

When the basis functions in $\mathbf{\Phi}$ are mutually orthogonal, then the Gram matrix $\mathbf{\Gamma}$ in Eq. (335) is diagonal, and the reproducing kernel $\mathcal{K}$ in Eq. (337) takes the simple form:

$$\mathcal{K}(x,y) = \sum_{k=1}^{n} \Delta_k \phi_k(x)\phi_k(y) \,,\ \text{with } \Delta_k = \Delta_{ik}\delta_{ik} = \Delta_{(kk)} > 0 \,,\ \text{for } k = 1,\ldots,\infty \,, \tag{341}$$

where δ_{ij} is the Kronecker delta, and where the summation convention on repeated indices, except when enclosed in parentheses, was applied. In the case of infinite-dimensional space of functions, Eq. (341), Eq. (333) and Eq. (334) would be written with $n \to \infty$:[231]

$$f(x) = \sum_{k=1}^{\infty} \zeta_k \phi_k(x) \,,\quad \langle f,g \rangle = \sum_{k=1}^{\infty} \zeta_k \eta_k \Gamma_k = \sum_{k=1}^{\infty} \zeta_k \eta_k / \Delta_k = \boldsymbol{\zeta}^T \mathbf{\Gamma} \boldsymbol{\eta} \tag{342}$$

$$\| f \|_{\mathcal{K}}^2 = \sum_{k=1}^{\infty} (\zeta_k)^2 \Gamma_k = \sum_{k=1}^{\infty} (\zeta_k)^2 / \Delta_k = \boldsymbol{\zeta}^T \mathbf{\Gamma} \boldsymbol{\zeta} \,,\quad \mathcal{K}(x,y) = \sum_{k=1}^{\infty} \Delta_k \phi_k(x)\phi_k(y) \,. \tag{343}$$

Let $L(y, f(x))$ be the loss (cost) function, with x being the data, $f(x)$ the predicted output, and y the label. Consider the regularized minimization problem:

$$\min_{f} \left[\sum_{i=1}^{n} L(y_i, f(x_i)) + \lambda \| f \|_{\mathcal{K}}^2 \right] = \min_{\{\zeta_i\}_1^{\infty}} \left[\sum_{i=1}^{n} L(y_i, \sum_{j=1}^{\infty} \zeta_j \phi_j(x_i)) + \lambda \sum_{j=1}^{\infty} (\zeta_j)^2/\Delta_j \right] \,, \tag{344}$$

which is a "ridge" penalty method[232] with λ being the penalty coefficient (or regularization parameter), with the aim of forcing the norm of the minimizer, i.e, $\| f^\star \|^2$, to be as small as possible, by penalizing the objective function (cost L plus penalty $\lambda \| f \|_{\mathcal{K}}^2$) if it were not.[233] What is remarkable is that even though the minimization problem is *infinite* dimensional, the minimizer (solution) $f^\star$ of Eq. (344) is *finite* dimensional [238]:

$$f^\star(x) = \sum_{i=1}^{n} \alpha_i^\star \mathcal{K}(x, x_i) \,, \tag{345}$$

where $\mathcal{K}(\cdot, x_i)$ is a basis function, and $\alpha_i^\star$ the corresponding coefficient, for $i = 1, \ldots, n$.

Remark 8.2. *Finite-dimensional solution to infinite-dimentional problem.* Since $f(x)$ is expressed as in Eq. (342)$_1$, and $\mathcal{K}(x, y)$ as in Eq. (343)$_2$, it follows that

$$\langle \mathcal{K}(y, x_i), f(y) \rangle_y = f(x_i) \,, \tag{346}$$

following the same argument as in Remark 8.1. As a result,

$$\langle \mathcal{K}(y, x_i), \mathcal{K}(y, x_j) \rangle_y = \mathcal{K}(x_i, x_j) \,,\ \text{with } i, j = 1, \ldots, n \,. \tag{347}$$

---

[231] Eq. (342)$_1$, and Eqs. (343)$_{1,2}$ were given as Eq. (5.46), Eq. (5.47), and Eq. (5.45), respectively, in [238], pp. 168-169, where the basis functions $\phi_i(x)$ were the "eigen-functions," and where the general case non-orthogonal basis presented in Eq. (337) was not discussed. Technically, Eq. (343)$_2$ is accompanied by the conditions $\Delta_i \geq 0$, for $i = 1, \ldots, \infty$, and $\sum_{k=1}^{\infty} (\Delta_k)^2 < \infty$, i.e., the sum of the squared coefficients is finite [238], p. 188.

[232] See [238], Eq. (3.41), p. 61, which is in Section 3.4.1 on "Ridge regression," which "shrinks the regression coefficients by imposing a penalty on them." Here the penalty is imposed on the kernel-induced norm of f, i.e., $\| f \|_{\mathcal{K}}^2$.

[233] In classical regularization, the loss function (1st term) in Eq. (344) is called the "empirical risk" and the penalty term (2nd term) the "stabilizer" [239].

Thus $\boldsymbol{\mathcal{K}} = [\mathcal{K}(x_i, x_j)] \in \mathbb{R}^{n \times n}$ is the Gram matrix of the set of functions $\{\mathcal{K}(\cdot, x_1), \ldots, \mathcal{K}(\cdot, x_n)\}$. To show that $\boldsymbol{\mathcal{K}} > 0$, i.e., positive definite, for any set $\{a_1, \ldots, a_n\}$, consider

$$\sum_{i,j=1}^{n} a_i \mathcal{K}(x_i, x_j) a_j = \sum_{i,j=1}^{n} a_i \sum_{p=1}^{\infty} \Delta_p \phi_p(x_i) \phi_p(x_j) a_j = \sum_{p=1}^{\infty} \Delta_p (b_p)^2 \geq 0 , \tag{348}$$

$$b_p := \sum_{i=1}^{n} a_i \phi_p(x_i) , \text{ for } p = 1, \ldots, \infty , \tag{349}$$

which is equivalent to the matrix $\boldsymbol{\mathcal{K}}$ being positive definite, i.e., $\boldsymbol{\mathcal{K}} > 0$,[234] and thus the functions $\{\mathcal{K}(\cdot, x_1), \ldots, \mathcal{K}(\cdot, x_n)\}$ are linearly independent, and form a basis, making expression such as Eq. (345) possible.

A goal now is to show that the solution to the *infinite*-dimensional regularized minimization problem Eq. (344) is *finite* dimensional, for which the coefficients $\alpha_i^\star$, $i = 1, \ldots, n$, in Eq. (345) are to be determined. It is also remarkable that the solution of the form Eq. (345) holds in general for *any* type of differentiable loss function L in Eq. (344), and not necessarily restricted to the squared-error loss [241] [239].

For notation compactness, let the objective function (loss plus penalty) in Eq. (344) be written as

$$\overline{L}[f] := \sum_{k=1}^{n} L(y_k, f(x_k)) + \lambda \parallel f \parallel_{\mathcal{K}}^2 = \sum_{k=1}^{n} L(y_k, \sum_{j=1}^{\infty} \zeta_j \phi_j(x_k)) + \lambda \sum_{k=1}^{\infty} (\zeta_k)^2 / \Delta_j , \tag{350}$$

and set the derivative of $\overline{L}$ with respect to the coefficients ζ_p, for $p = 1, \ldots, \infty$, to zero to solve for these coefficients:

$$\frac{\partial \overline{L}}{\partial \zeta_p} = -\sum_{k=1}^{n} \frac{\partial L(y_k, f(x_k))}{\partial f} \frac{\partial f(x_k)}{\partial \zeta_p} + 2\lambda \frac{\partial \parallel f \parallel_{\mathcal{K}}}{\partial \zeta_p} = -\sum_{k=1}^{n} \frac{\partial L(y_k, f(x_k))}{\partial f} \phi_p(x_k) + 2\lambda \frac{\zeta_p}{\Delta_p} = 0 \quad (351)$$

$$\Rightarrow \zeta_p^\star = \frac{\Delta_p}{2\lambda} \sum_{k=1}^{n} \frac{\partial L(y_k, f(x_k))}{\partial f} \phi_p(x_k) = \Delta_p \sum_{k=1}^{n} \alpha_k^\star \phi_p(x_k) , \text{ with } \alpha_k^\star := \frac{1}{2\lambda} \frac{\partial L(y_k, f(x_k))}{\partial f} , \tag{352}$$

$$\Rightarrow f^\star(x) = \sum_{k=1}^{\infty} \zeta_k^\star \phi_k(x) = \sum_{k=1}^{\infty} \Delta_k \sum_{i=1}^{n} \alpha_i^\star \phi_k(x_i) \phi_k(x) = \sum_{i=1}^{n} \alpha_i^\star \sum_{k=1}^{\infty} \Delta_k \phi_k(x_i) \phi_k(x) = \sum_{i=1}^{n} \alpha_i^\star \mathcal{K}(x_i, x) , \tag{353}$$

where the last expression in Eq. (353) came from using the kernel expression in Eq. (343)$_2$, and the end result is Eq. (345), i.e., the solution (minimizer) $f^\star$ is of finite dimension.[235]   ∎

For the squared-error loss,

$$L(\boldsymbol{y}, f(\boldsymbol{x})) = \parallel \boldsymbol{y} - f(\boldsymbol{x}) \parallel_{\mathcal{K}}^2 = \sum_{k=1}^{n} [y_k - f(x_k)]^2 = [\boldsymbol{y} - \boldsymbol{\mathcal{K}} \boldsymbol{\alpha}^\star]^T [\boldsymbol{y} - \boldsymbol{\mathcal{K}} \boldsymbol{\alpha}^\star] , \text{ with} \tag{354}$$

$$\boldsymbol{y}^T = [y_1, \ldots, y_n] \in \mathbb{R}^{1 \times n} , \quad \boldsymbol{\mathcal{K}} = [\mathcal{K}_{ij}] = [\mathcal{K}(x_i, x_j)] \in \mathbb{R}^{n \times n} , \quad \boldsymbol{\alpha}^{\star T} = [\alpha_1^\star, \ldots, \alpha_n^\star] \in \mathbb{R}^{n \times n} , \quad (355)$$

the coefficients $\alpha_k^\star$ in Eq. (353) (or Eq. (345)) can be computed using Eq. (352)$_2$:

$$\alpha_k^\star = \frac{1}{\lambda} [y_k - f^\star(x_k)] \Rightarrow \lambda \alpha_k^\star = y_k - \sum_{j=1}^{n} \mathcal{K}_{kj} \alpha_j^\star \Rightarrow [\boldsymbol{\mathcal{K}} + \lambda \boldsymbol{I}] \boldsymbol{\alpha}^\star = \boldsymbol{y} \Rightarrow \boldsymbol{\alpha}^\star = [\boldsymbol{\mathcal{K}} + \lambda \boldsymbol{I}]^{-1} \boldsymbol{y} . \tag{356}$$

---

[234] See, e.g., [240], p. 11.
[235] See also [238], p. 169, Eq. (5.50), and p. 185.

It is then clear from the above that the "Setup" section in [235] simply corresponded to the particular case where the penalty parameter was zero:

$$\lambda = 0 \Rightarrow y_k = f^{\star}(x_k) = \sum_{j=1}^{n} \mathcal{K}_{kj} \alpha_j^{\star} \, , \tag{357}$$

i.e., $f^{\star}$ is interpolating.

For technical jargon such as Reproducing Kernel Hilbert Space (RKHS), Riesz Representation Theorem, $\mathcal{K}(\cdot, x_i)$ as a representer of evaluation at x_i, etc. to describe several concepts presented above, see [237] [242] [240] [238].[236]

Table 5: *Some reproducing kernels* (Section 8). See [239], [130], p. 296, p. 305, [244].

Regularization network	Kernel function
Gaussian Radial Basis Function	$\exp\left[-\parallel \boldsymbol{x} - \boldsymbol{y} \parallel^2\right]$
Exponential (Laplacian)	$\exp\left[-\parallel \boldsymbol{x} - \boldsymbol{y} \parallel\right]$
Inverse multiquadric	$\left[\parallel \boldsymbol{x} - \boldsymbol{y} \parallel^2 + c^2\right]^{-1/2}$
Multiquadric	$\left[\parallel \boldsymbol{x} - \boldsymbol{y} \parallel^2 + c^2\right]^{1/2}$
Thin plate spline (a)	$\parallel \boldsymbol{x} - \boldsymbol{y} \parallel^{2n+1}$
Thin plate spline (b)	$\parallel \boldsymbol{x} - \boldsymbol{y} \parallel^{2n} \log \parallel \boldsymbol{x} - \boldsymbol{y} \parallel$
Multilayer perceptron (for some values of θ)	$\tanh\left(\boldsymbol{x} \cdot \boldsymbol{y} - \theta\right)$
Polynomial of degree d	$\left[1 + \boldsymbol{x} \cdot \boldsymbol{y}\right]^d$

8.2 Exponential functions as reproducing kernels

A list of reproducing kernels is given in, e.g., [241] [239], such as those in Table 5. Two reproducing kernels with exponential function were used to understand how deep learning works [235], and are listed in Eq. (358): (1) the popular smooth Gaussian kernel $\mathcal{K}_G$ in in Eq. (358)$_1$, and (2) the non-smooth Laplacian (exponential) kernel[237] $\mathcal{K}_L$ in Eq. (358)$_2$:

$$\mathcal{K}_G(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(-\frac{\parallel \boldsymbol{x} - \boldsymbol{y} \parallel^2}{2\sigma^2}\right) \, , \quad \mathcal{K}_L(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(-\frac{\parallel \boldsymbol{x} - \boldsymbol{y} \parallel}{\sigma}\right) \, , \tag{358}$$

where σ is the standard deviation. The method in Remark 8.1 is not suitable to show that these exponential functions are reproducing kernels. We provide a verificatrion of the reproducing property of the Laplacian kernel in Eq. (358)$_2$.

Remark 8.3. *Laplacian kernel is reproducing.* Consider the Laplacian kernel in Eq. (358)$_2$ for the scalar case (x, y) with $\sigma = 1$ for simplicity, without loss of generality. The goal is to show that the reproducing

---

[236] A succinct introduction to Hilbert space and the Riesz Representation theorem, with detailed proofs, starting from the basic definitions, can be found in [243].

[237] In [130], p. 305, the kernel $\mathcal{K}_L$ in Eq. (358)$_2$ was called "exponential kernel," without a reference to Laplace, but referred to "the Ornstein-Uhlenbeck process originally introduced by Uhlenbeck and Ornstein (1930) to describe Brownian motion." Similarly, in [234], p. 85, the term "exponential covariance function" (or kernel) was used for the kernel $k(r) = \exp(-r/\ell)$, with $r = |x - y|$, in connection with the Ornstein-Uhlenbeck process. Even though the name "kernel" came from the theorie of integral operator [234], p. 80, the attribution of the exponential kernel to Laplace came from the Laplace probability distribution (Wikipedia, version 06:55, 24 August 2022), also called the "double exponential" distribution, but not from the different kernel used in the Laplace transform. See also Remark 8.3.1 and Figure 86.

property in Eq. $(332)_1$ holds for such kernel expressed in Eq. $(359)_1$ below.

$$\mathcal{K}(x,y) = \exp\left[-|x-y|\right] \Rightarrow \mathcal{K}'(x,y) := \frac{\partial \mathcal{K}(x,y)}{\partial y} = \begin{cases} -\mathcal{K}(x,y) \text{ for } y > x \\ \mathcal{K}(x,y) \text{ for } y < x \end{cases} \tag{359}$$

$$\Rightarrow \mathcal{K}''(x,y) := \frac{\partial^2 \mathcal{K}(x,y)}{(\partial y)^2} = \mathcal{K}(x,y) \ , \text{ for } y \neq x \ . \tag{360}$$

The method is by using integration by parts and by using a function norm different from that in Eq. $(332)_2$; see [240], p. 8. Now start with the integral in Eq. $(332)_2$, and do integration by parts:

$$\int f(y)\mathcal{K}(x,y)dy = \int f(y)\mathcal{K}''(x,y)dy = \int \left[f\mathcal{K}'\right]' dy - \int f'\mathcal{K}'dy = \left[f\mathcal{K}'\right]_{y=-\infty}^{y=+\infty} - \int f'\mathcal{K}'dy \tag{361}$$

$$\left[f\mathcal{K}'\right]_{y=-\infty}^{y=+\infty} = \left[f(+\mathcal{K})\right]_{y=-\infty}^{y=x^-} + \left[f(-\mathcal{K})\right]_{y=x^+}^{y=+\infty} = f(x^-)\mathcal{K}(x^-,x) + f(x^+)\mathcal{K}(x^+,x) = 2f(x) \tag{362}$$

$$\Rightarrow \int f(y)\mathcal{K}(x,y)dy + \int f'(y)\mathcal{K}'(x,y)dy = 2f(x) \ , \tag{363}$$

where $x^- = x - \epsilon$ and $x^+ = x + \epsilon$ with $\epsilon > 0$ being very small. The scalar product on the space of functions that are differentiable almost everywhere, i.e.,

$$\langle f, g \rangle = \frac{1}{2} \left[\int f(y)g(y)dy + \int f'(y)g'(y)dy \right] \ , \tag{364}$$

together with Eq. (363), and $g(y) = \mathcal{K}(x,y)$, show that the Laplacian kernel is reproducing.[238]   ∎

8.3 Gaussian processes

The Kalman filter, well-known in engineering, is an example of a Gaussian-process model. See also Remark 9.6 in Section 9.5 on the 2021 US patent on Physics-Informed Learning Machine that was based on Gaussian processes (GPs),[239] which possess the "most pleasant resolution imaginable" to the question of how to computationally deal with infinite-dimensional objects like functions [234], p. 2:

> "If you ask only for the properties of the function at a finite number of points, then the inference from the Gaussian process will give you the same answer if you ignore the infinitely many other points, as if you would have taken them into account! And these answers are consistent with any other finite queries you may have. One of the main attractions of the Gaussian process framework is precisely that it unites a sophisticated and consistent view with computational tractability."

A simple example of a Gaussian process is the linear model $f(\cdot)$ below [130]:

$$y = f(x) = \sum_{k=0}^{k=n} w_k x^k = \boldsymbol{w}^T \boldsymbol{\phi}(x) \ , \text{ with } \boldsymbol{w}^T = \lfloor w_0, \ldots, w_n \rfloor \in \mathbb{R}^{1 \times (n+1)} \ , \ w_k \sim \mathcal{N}(0,1) \ , \tag{365}$$

$$\boldsymbol{\phi}(x) = [\phi_0(x), \ldots, \phi_n(x)]^T \in \mathbb{R}^{(n+1) \times 1} \ , \text{ and } \phi_k(x) = x^k \ , \tag{366}$$

---

[238] It is possible to define the Laplacian kernel in Eq. $(359)_1$ with the factor $\frac{1}{2}$, which then will not appear in the definition of the scalar product in Eq. (364). See also [240], p. 8.

[239] Non-Gaussian processes, such as in [245] [246], are also important, but more advanced, and thus beyond the scope of the present review. See also the Gaussian-Process Summer-School videos (Youtube) 2019 and 2021. We thank David Duvenaud for noting that we did not review non-Gaussian processes.

with random coefficients w_k, $k = 0, \ldots, n$, being normally distributed with zero mean $\mu = 0$ and unit variance $\sigma^2 = 1$, and with basis functions $\{\phi_k(x) = x^k, k = 0, \ldots, n\}$.

More generally, $\phi(x)$ could be any basis of nonlinear functions in x, and the weights in $\boldsymbol{w} \in \mathbb{R}^{n \times 1}$ could have a joint Gaussian distribition with zero mean and a given covariance marix $\boldsymbol{C_{ww}} = \boldsymbol{\Sigma} \in \mathbb{R}^{n \times n}$, i.e, $\boldsymbol{w} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$. If w_k, $k = 1, \ldots, n$, are idependently and identically distributed (i.i.d.), with variance σ^2, then the covariance matrix is diagonal (and is called "isotropic" [130], p. 84), i.e., $\boldsymbol{\Sigma} = \sigma^2 \boldsymbol{I}$ and $\boldsymbol{w} \sim \mathcal{N}(\mathbf{0}, \sigma^2 \boldsymbol{I})$, with $\boldsymbol{I}$ being the identity matrix.

Formally, a *Gaussian process* is a probability distribution over the functions $f(x)$ such that, for any given arbitrary set of input training points $\boldsymbol{x} = [x_1, \ldots, x_m]^T \in \mathbb{R}^{m \times 1}$, the set of output values of $f(\cdot)$ at input training points, i.e., $\boldsymbol{y} = [y_1, \ldots, y_m]^T = [f(x_1), \ldots, f(x_m)]^T \mathbb{R}^{m \times 1}$, such that $y_i = f(x_i)$, which from Eq. (365) can be written as

$$\boldsymbol{y} = (\boldsymbol{w}^T \boldsymbol{\Phi})^T = \boldsymbol{\Phi}^T \boldsymbol{w} , \quad y_j = f(x_j) = \sum_{i=1}^{n} w_i \phi_i(x_j) , \; j = 1, \ldots, m , \text{ such that } \boldsymbol{w} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}) , \quad (367)$$

and the *design* matrix $\boldsymbol{\Phi} = [\phi_i(x_j)] \in \mathbb{R}^{n \times m}$, has a joint probability distribution; see [130], p. 305.

Another way to put it succinctly, a Gaussian process describes a distribution over functions, and is defined as a collection of random variables (representing the values of the function $f(\boldsymbol{x})$ at location $\boldsymbol{x}$), such that any finite subset of which has a joint probability distribution [234], p. 13.

The multivariate (joint probability) Gaussian distribution for an $m \times 1$ matrix $\boldsymbol{y}$, with mean $\boldsymbol{\mu}$ (element-wise expectation of $\boldsymbol{y}$) and covariance matrix $\boldsymbol{C_{yy}}$ (element-wise expectation of $\boldsymbol{yy}^T$) is written as

$$\mathcal{N}(\boldsymbol{y}|\boldsymbol{\mu}, \boldsymbol{C_{yy}}) = \frac{1}{(2\pi)^{m/2} \det \boldsymbol{C_{yy}}} \exp\left[-\frac{1}{2}(\boldsymbol{y} - \boldsymbol{\mu})^T \boldsymbol{C_{yy}}^{-1}(\boldsymbol{y} - \boldsymbol{\mu})\right] , \text{ with} \quad (368)$$

$$\boldsymbol{\mu} = \mathbb{E}[\boldsymbol{y}] = \mathbb{E}\left[\boldsymbol{w}^T\right] \boldsymbol{\Phi} = \mathbf{0} , \quad (369)$$

$$\boldsymbol{C_{yy}} = \mathbb{E}\left[\boldsymbol{yy}^T\right] = \boldsymbol{\Phi}^T \mathbb{E}\left[\boldsymbol{ww}^T\right] \boldsymbol{\Phi} = \boldsymbol{\Phi}^T \boldsymbol{C_{ww}} \boldsymbol{\Phi} = \boldsymbol{\Phi}^T \boldsymbol{\Sigma} \boldsymbol{\Phi} = [\mathcal{K}(x_i, x_j)] \in \mathbb{R}^{m \times m} , \quad (370)$$

where $\boldsymbol{C_{ww}} = \boldsymbol{\Sigma} \in \mathbb{R}^{n \times n}$ is a given covariance matrix of the weight matrix $\boldsymbol{w} \in \mathbb{R}^{n \times 1}$. The covariance matrix $\boldsymbol{C_{yy}}$ in Eq. (370) has the same mathematical structure as Eq. (337), and is therefore a reproducing kernel, with kernel function $k(\cdot, \cdot)$

$$\text{cov}(y_i, y_j) = \text{cov}(f(x_i), f(x_j)) = \mathbb{E}[f(x_i), f(x_j)] = \mathcal{K}(x_i, x_j) = \sum_{p=1}^{n} \sum_{q=1}^{n} \phi_p(x_i) \Sigma_{pq} \phi_q(x_j) . \quad (371)$$

In the case of an isotropic covariance matrix $\boldsymbol{C_{ww}}$, the kernel function $k(\cdot, \cdot)$ takes a simple form:[240]

$$\boldsymbol{C_{ww}} = \sigma^2 \boldsymbol{I} \Rightarrow \mathcal{K}(y_i, y_j) = \mathcal{K}(f(x_i), f(x_j)) = \sigma^2 \sum_{p=1}^{n} \phi_p(x_i) \phi_p(x_j) = \sigma^2 \boldsymbol{\phi}^T(x_i) \boldsymbol{\phi}(x_j) , \quad (372)$$

with $\boldsymbol{\phi}(x) = [\phi_1(x), \ldots, \phi_n(x)]^T \in \mathbb{R}^{n \times 1}$. $\quad (373)$

The Gaussian (normal) probability distribution $\mathcal{N}(\boldsymbol{y}|\boldsymbol{\mu}, \boldsymbol{C_{yy}})$ in Eq. (368) is the *prior* probability distribution for $\boldsymbol{y}$, before any conditioning with observed data (i.e., before specifying the actual observed values of the outputs in $\boldsymbol{y} = f(\boldsymbol{x})$).

Remark 8.4. *Zero mean.* In a Gaussian process, the joint distribution Eq. (368) over the outputs, i.e., the n random variables in $\boldsymbol{y} = f(\boldsymbol{x}) \in \mathbb{R}^{n \times 1}$, is defined completely by "second-order statistics," i.e., the mean $\boldsymbol{\mu} = \mathbf{0}$ and the covariance $\boldsymbol{C_{yy}}$. In practice, the mean of $\boldsymbol{y} = f(\boldsymbol{x})$ is not available a priori, and "so by

---

[240] The "precision" is the inverse of the variance, i.e., σ^{-2} [130], p. 304.
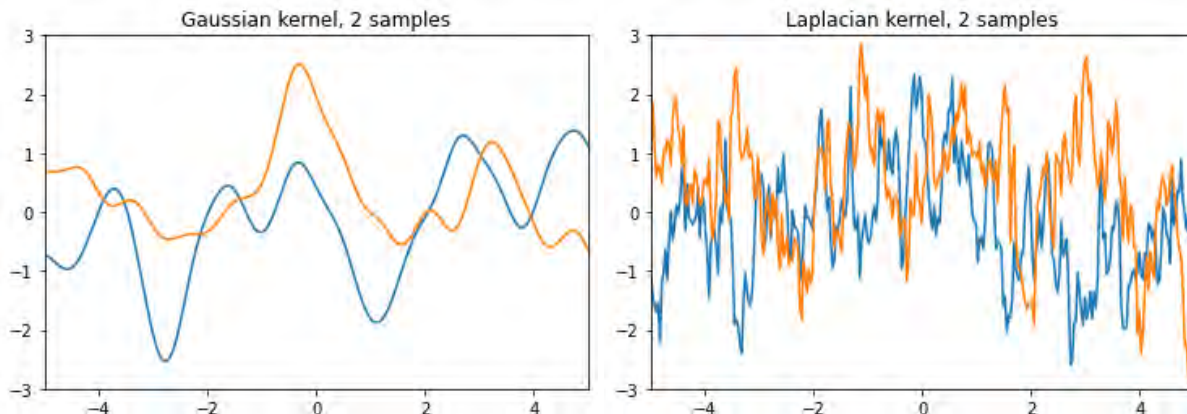
Figure 86: *Gaussian process priors* (Section 8.3). *Left:* Two samples with Gaussian kernel. *Right:* Two samples with Laplacian kernel. Parameters for both kernels: Kernel precision (inverse of variance) $\gamma = \sigma^{-2} = 0.2$ in Eq. (358), isotropic noise variance $\nu^2 I = 10^{-6} I$ added to covariance matrix C_{yy} of output y and isotropic weight covariance matrix $C_{ww} = \Sigma = I$ in Eq. (374).

symmetry, we take it to be zero" [130], p. 305, or equivalently, specify that the weights in $w \in \mathbb{R}^{n \times 1}$ have zero mean, as in Eqs. (365), (367), (369). ∎

8.3.1 Gaussian-process priors and sampling

Instead of defining the kernel function $\mathcal{K}(\cdot, \cdot)$ by selecting a basis of functions as in Eq. (371), the kernel function can be directly defined using the analytical expressions in Table 5 and Section 8.2. Figure 86 provides a comparison of samples from two Gaussian-process priors, one with Gaussian kernel $\mathcal{K}_G$ and one with Laplacian kernel $\mathcal{K}_L$, as given in Eq. (358) with precision $\gamma = \sigma^{-2}$. Since the Gram matrix $K = [\mathcal{K}(x_i, x_j)]$ for a kernel $\mathcal{K}$ is positive semidefinite[241] ([130], p. 295), to make the Gram matrix $K = C_{yy}$ in Eq. (367) positive definite before performing a Choleski decomposition, a noise with isotropic variance $\nu^2 I$ is added to K ([130], p. 314), meaning that the noise is the same and independent in each direction. Then the sample function values can be obtained as follows ([130], p. 528):

$$C_{yy,\nu} = \Phi_\nu^T \Sigma \Phi_\nu = K + \nu^2 I = LL^T \text{ , with } w \sim \mathcal{N}(0, \Sigma = I) \text{ , then } y = \Phi_\nu^T w = Lw \text{ .} \tag{374}$$

In other words, the GP prior samples in Figure 86 were drawn from the Gaussian distribution with zero mean and covariance matrix $C_{yy,\nu}$ in Eq. (374):

$$\widetilde{y} := f(\widetilde{x}) \sim \mathcal{N}(y|\mu, C_{yy,\nu}) = \mathcal{N}(0, K + \nu^2 I) \text{ ,} \tag{375}$$

where $\widetilde{x}$ contains the test inputs, and $\widetilde{y} = f(\widetilde{x})$ contains the predictive values of $f(\cdot)$ at $\widetilde{x}$.

It can be observed from Figure 86 that samples obtained with the Gaussian kernel were smooth, with slow variations, whereas samples obtained with the Laplacian kernel had high jiggling with rapid variations, appropriate to model Brownian motion; see Footnote 237.

8.3.2 Gaussian-process posteriors and sampling

Let $y = [y_1, \ldots, y_m]^T \in \mathbb{R}^{m \times 1}$ be the observed data (or target values) at the training points $x = [x_1, \ldots, x_m]^T \in \mathbb{R}^{m \times 1}$, and let $\widetilde{x} \in \mathbb{R}^{\widetilde{m} \times 1}$ contain the $\widetilde{m}$ test input points, with the predictive values in

---

[241] That the Gram matrix is positive semidefinite should be familiar with practitioners of the finite element method, in which the Gram matrix is the stiffness matrix (for elliptic differential operators), and is positive semidefinite before applying the essential boundary conditions.
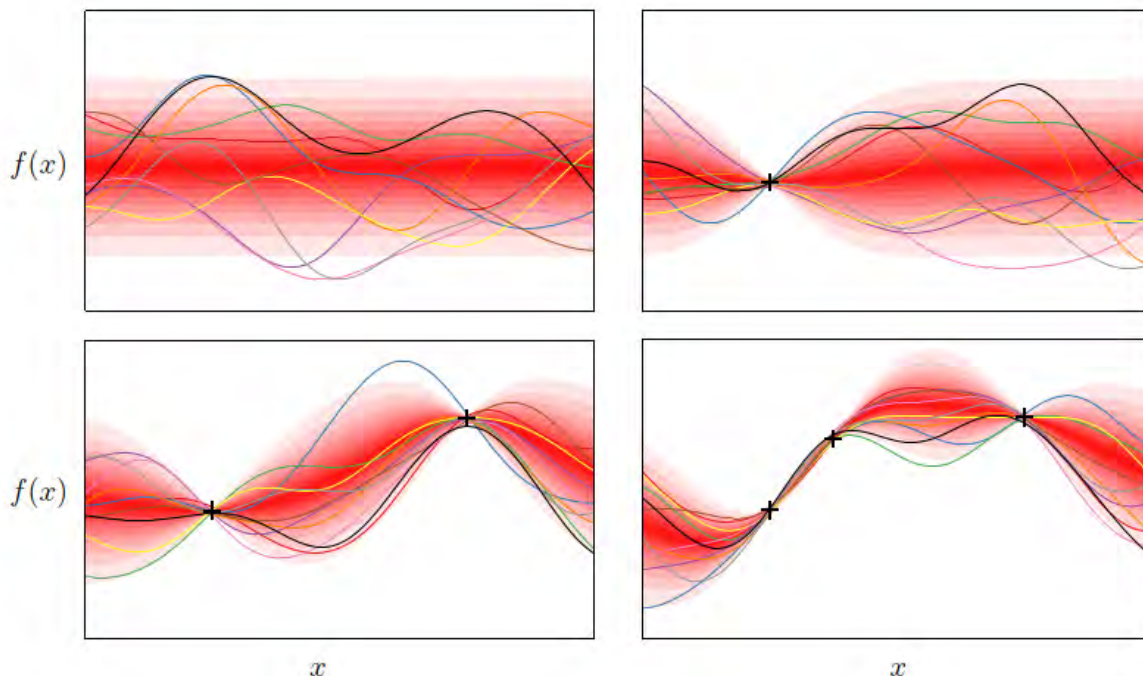
Figure 87: *Gaussian process prior and posterior samplings, Gaussian kernel* (Section 8.3). *Top left:* Gaussian-prior samples (Section 8.3.1). The shaded red zones represent the predictive density of at each input location. *Top right:* Gaussian-posterior samples with 1 data point. *Bottom left:* Gaussian-posterior samples with 2 data points. *Bottom right:* Gaussian-posterior samples with 3 data points [247]. See Figure 88 for the noise effects and Figure 89 for animation of GP priors and posteriors. (Figure reproduced with permission of the authors.)

$\widetilde{\boldsymbol{y}} = f(\widetilde{\boldsymbol{x}}) \in \mathbb{R}^{\widetilde{m} \times 1}$. The combined function values in the matrix $[\boldsymbol{y}, \widetilde{\boldsymbol{y}}]^T \in \mathbb{R}^{(m+\widetilde{m}) \times 1}$, as random variables, are distributed "normally" (Gaussian), i.e.,

$$\begin{Bmatrix} f(\boldsymbol{x}) \\ f(\widetilde{\boldsymbol{x}}) \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{y} \\ \widetilde{\boldsymbol{y}} \end{Bmatrix} = \mathcal{N} \left(\begin{Bmatrix} \boldsymbol{\mu}(\boldsymbol{x}) \\ \boldsymbol{\mu}(\widetilde{\boldsymbol{x}}) \end{Bmatrix}, \begin{bmatrix} \boldsymbol{K}(\boldsymbol{x}, \boldsymbol{x}) + \nu^2 \boldsymbol{I} & \boldsymbol{K}(\boldsymbol{x}, \widetilde{\boldsymbol{x}}) \\ \boldsymbol{K}^T(\boldsymbol{x}, \widetilde{\boldsymbol{x}}) & \boldsymbol{K}(\widetilde{\boldsymbol{x}}, \widetilde{\boldsymbol{x}}) \end{bmatrix} \right). \tag{376}$$

The Gaussian-process posterior distribution, i.e., the conditional Gaussian distribution for the test output $\widetilde{\boldsymbol{y}}$ given the training data $(\boldsymbol{x}, \boldsymbol{y} = f(\boldsymbol{x}))$ is then (See Appendix 3 for the detailed derivation, which is simpler than in [130] and in [248])[242]

$$p(\widetilde{\boldsymbol{y}} \,|\, \boldsymbol{x}, \boldsymbol{y}, \widetilde{\boldsymbol{x}}) = \mathcal{N}(\widetilde{\boldsymbol{y}} \,|\, \widetilde{\boldsymbol{\mu}}, \boldsymbol{C}_{\widetilde{\boldsymbol{y}}\widetilde{\boldsymbol{y}}}), \tag{377}$$

$$\widetilde{\boldsymbol{\mu}} = \boldsymbol{\mu}(\widetilde{\boldsymbol{x}}) + \boldsymbol{K}(\widetilde{\boldsymbol{x}}, \boldsymbol{x}) \left[\boldsymbol{K}(\boldsymbol{x}, \boldsymbol{x}) + \nu^2 \boldsymbol{I} \right]^{-1} [\boldsymbol{y} - \boldsymbol{\mu}(\boldsymbol{x})] = \boldsymbol{K}(\widetilde{\boldsymbol{x}}, \boldsymbol{x}) \boldsymbol{K}^{-1}(\boldsymbol{x}, \boldsymbol{x}) \, \boldsymbol{y}, \tag{378}$$

$$\boldsymbol{C}_{\widetilde{\boldsymbol{y}}\widetilde{\boldsymbol{y}}} = \boldsymbol{K}(\widetilde{\boldsymbol{x}}, \widetilde{\boldsymbol{x}}) - \boldsymbol{K}(\widetilde{\boldsymbol{x}}, \boldsymbol{x}) \left[\boldsymbol{K}(\boldsymbol{x}, \boldsymbol{x}) + \nu^2 \boldsymbol{I} \right]^{-1} \boldsymbol{K}(\boldsymbol{x}, \widetilde{\boldsymbol{x}}), \tag{379}$$

where the mean was set to zero by Remark 8.4. In Figure 87, the number m of training points varied from 1 to 3. The Gaussian posterior sampling follows the same method as in Eq. (374), but with the covariance

---

[242] See, e.g., [234], p. 16, [130], p. 87, [247], p. 4. The authors of [234], in their Appendix A.2, p. 200, referred to [248] "sec. 9.3" for the derivation of Eqs. (378)-(379), but there were several sections numbered "9.3" in [248]; the correct referencing should be [248], Chapter XIII "More on distributions," Sec. 9.3 "Marginal distributions and conditional distributions," p. 427.
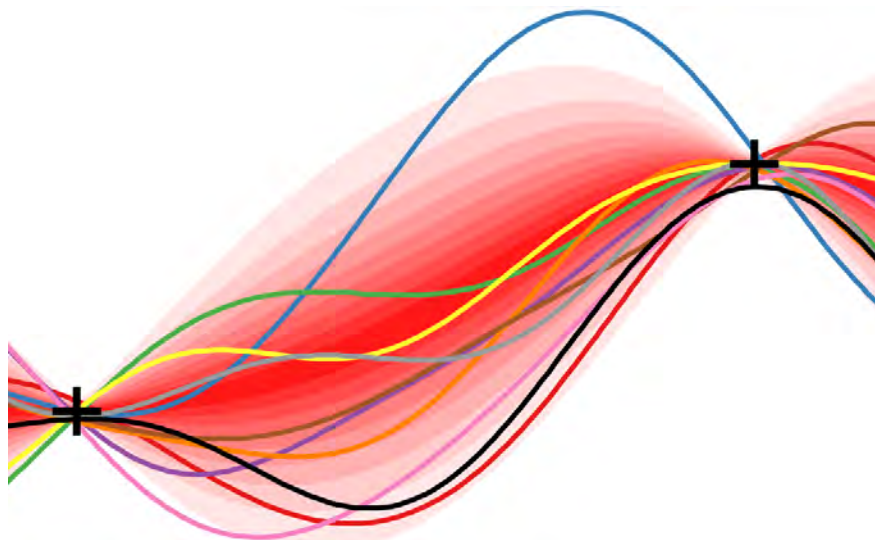
Figure 88: *Gaussian process posterior samplings, noise effects* (Section 8.3). Not all sampled curves in Figure 87 went through the data points, such as the black line in the present zoomed-in view of the bottom-left subfigure [247]. It is easy to make the sampled curves passing closer to the data points simply by reducing the noise variance ν^2 in Eqs. (376), (378), (379). See Figure 89 for animation of GP priors and posteriors.   (Figure reproduced with permission of the authors.)

matrix in Eq. (379):[243]

$$C_{\widetilde{y}\widetilde{y}} = \widetilde{L}\widetilde{L}^T \ , \ \text{ with } w \sim \mathcal{N}(\mathbf{0}, \Sigma = I) \ , \ \text{ then } \widetilde{y} = \widetilde{L}w \ . \tag{380}$$

9 Deep-learning libraries, frameworks, platforms

Among factors that drove the resurgence of AI, see Section 2, the availability of effective computing hardware and libraries that facilitate leveraging that hardware for DL-purposes have played and continue to play an important role in terms of both expansion of research efforts and dissemination in applications. Both commercial software, but primarily open-source libraries, which are backed by major players in software industry and academia, have emerged over the last decade, see, e.g., Wikipedia's "Comparison of deep learning software" version 12:51, 18 August 2022.

Figure 90 compares the popularity (as of 2018) of various software frameworks of the DL-realm by means of their "Power Scores".[244] The "Power Score" metric was computed from the occurrences of the respective libraries on 11 different websites, which range from scientific ones, e.g., as world's largest storage for research articles and preprints arXiv to social media outlets as, e.g., Linkedin.

The impressive pace at which DL research and applications progress is also reflected in changes the software landscape has been subjected to. As of 2018, TensorFlow was clearly dominant, see Figure 90, whereas Theano was the only library around among those only five years earlier according to the author of the 2018 study.

As of August 2022, the picture has once again changed quite a bit. Using Google Trends as metric,[245] PyTorch, which was in third place in 2018, has taken the leading position from TensorFlow as most popular

---

[243] The Matlab code for generating Figures 87 and 88 was provided courtesy of David Duvenaud. On line 25 of the code, the noise variance ν^2 was set as `sigma = 0.02`, which was much larger than $\nu^2 = 10^{-6}$ used in Figure 86.
[244] See [249].
[245] See this link for the latest Google Trends results corresponding to Figure 91.
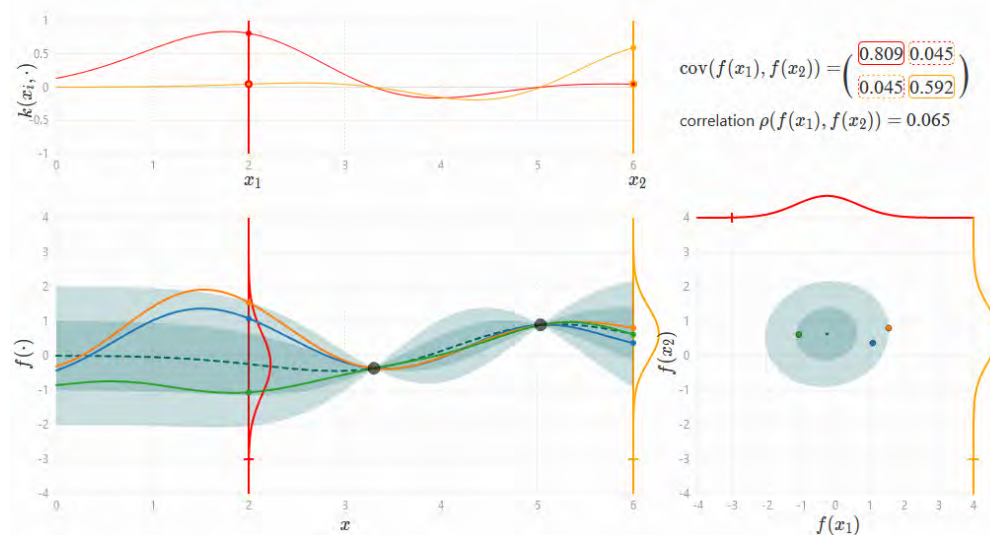
Figure 89: *Gaussian process posterior samplings, animation* (Section 8.3). Interactive Gaussian Process Visualization, Infinite curiosity. Click on the plot area to specify data points. See Figures 87 and 88.

DL-related software framework, see Figure 91.

Although the individual software frameworks do differ in terms of functionality, scope and internals, their overall purpose are clearly the same, i.e., to facilitate creation and training of neural networks and to harness the computational power of parallel computing hardware as, e.g., GPUs. For this reason, libraries share the following ingredients, which are essentially similar but come in different styles:

- *Linear algebra:* In essence, DL boils down to algebraic operations on large sets of data arranged in multi-dimensional arrays, which are supported by all software frameworks, see Section 4.4.[246]

- *Back-propagation:* Gradient-based optimization relies on efficient evaluation of derivatives of loss functions with respect to network parameters. The representation of algebraic operations as computational graphs allows for automatic differentiation, which is typically performed in *reverse-mode*, hence, back-propagation, see Section 5.

- *Optimization:* DL-libraries provide a variety of optimization algorithms that have proven effective in training of neural networks, see Section 6.

- *Hardware-acceleration:* Training deep neural networks is computationally intensive and requires adequate hardware, which allows algebraic computations to be performed in parallel. DL-software frameworks support various kinds of parallel/distributed hardware ranging from multi-threaded CPUs to GPUs to DL-specific hardware as TPUs. Parallelism is not restricted to algebraic computations only, but data also has to be efficiently loaded from storage and transferred to computing units.

- *Frontend and API:* Popular DL-frameworks provide an intuitive API, which supports accessibility for first-time learners and dissemination of novel methods to scientific fields beyond computer science. *Python* has become the prevailing programming language in DL, since it is more approachable for

---

[246] In the context of DL, multi-dimensional arrays are also referred to a tensors, although the data often lacks the defining properties of tensors as algebraic objects. The software framework *TensorFlow* even reflects that by its name.
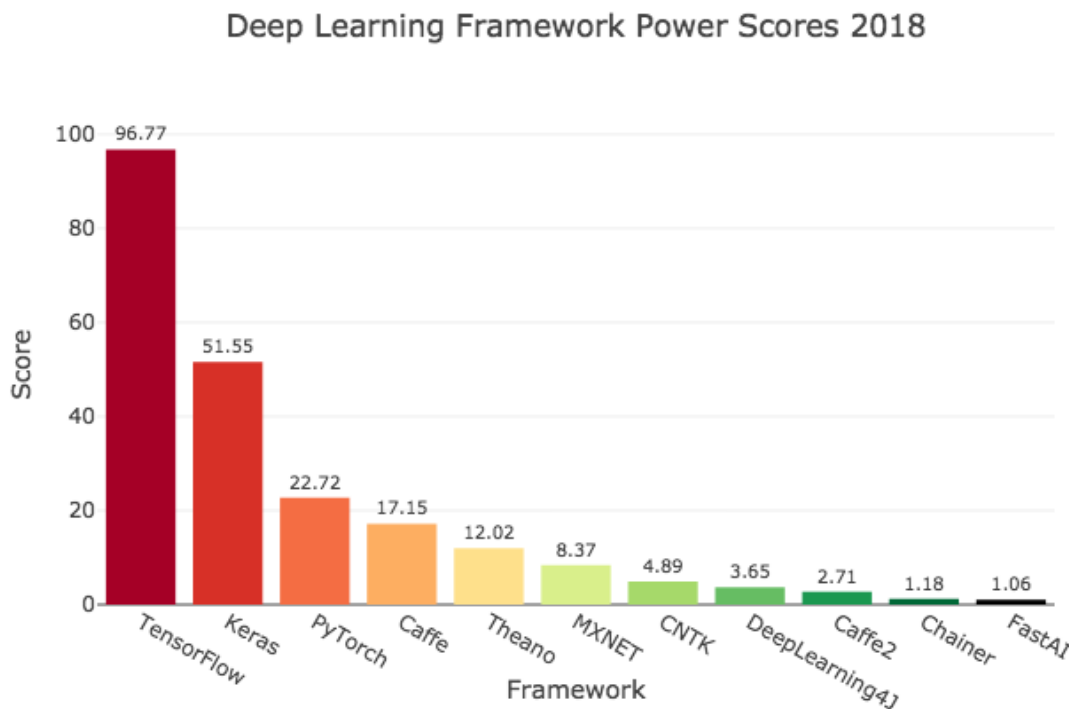
Figure 90: Top deep-learning libraries in 2018 by the "Power Score" in [249]. By 2022, using Google Trends, the popularity of different frameworks is significantly different; see Figure 91.

less-proficient developers as compared to languages traditionally popular in computational science as, e.g., C++ or Fortran. High-level APIs provide all essential building blocks (layers, activations, loss functions, optimizers, etc.) for both construction and training of complex network topologies and fully abstract the underlying algebraic operations from users.

In what follows, a brief description of some of the most popular software frameworks is given.

9.1 TensorFlow

TensorFlow [250] is a free and open-source software library which is being developed by the *Google Brain* research team, which, in turn, is part of Google's AI division. TensorFlow emerged from Google's proprietary predecessor "DistBelief" and was released to public in November 2015. Ever since its release, TensorFlow has rapidly become the most popular software framework in the field of deep-learning and maintains a leading position as of 2022, although it has been outgrown in popularity by its main competitor PyTorch particularly in research.

In 2016, Google presented its own AI accelerator hardware for TensorFlow called "Tensor Processing Unit" (TPU), which is built around an application-specific integrated circuited (ASIC) tailored to computations needed in training and evaluation of neural networks. DeepMind's grandmaster-beating software AlphaGo, see Section 2.3 and Figure 2, was trained using TPUs.[247] TPUs were made available to the public as part of "Google Cloud" in 2018. A single fourth-generation TPU device has a peak computing power of 275 teraflops for 16 bit floating point numbers (`bfloat16`) and 8 bit integers (`int8`). A fourth-generation cloud TPU "pod", which comprises 4096 TPUs offers a peak computing power of 1.1 exaflops.[248]

---

[247] See Google's announcement of TPUs [251].

[248] Recall that the 'exa'-prefix translates into a factor of 1×10^{18}. For comparison, Nvidia's latest H100 GPU-based accelerator has a half-precision floating point (`bfloat16`) performance of 1 petaflops.
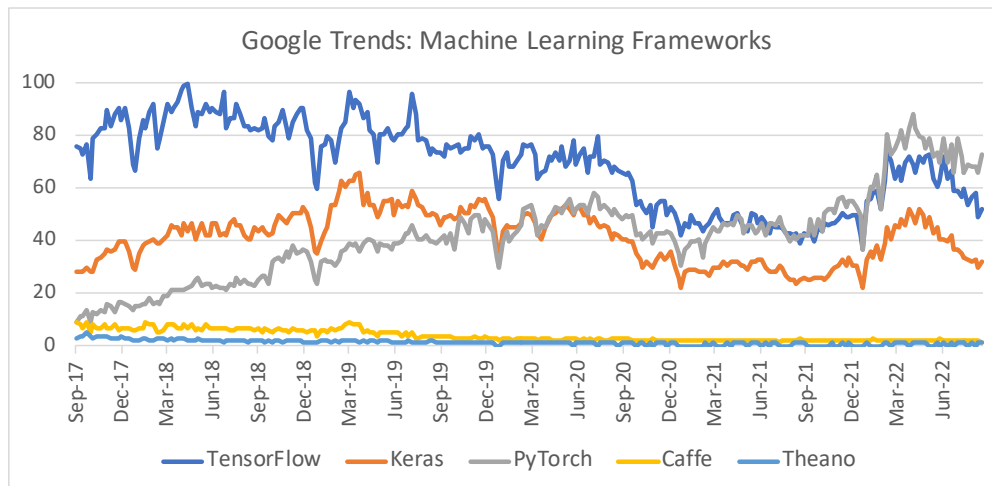
Figure 91: *Google Trends of deep-learning software libraries* (Section 9). The chart shows the popularity of five DL-related software libraries most "powerful" in 2018 over the last 5 years (as of July 2022). See also Figure 90 for the rankings of DL frameworks in 2018.

9.1.1 Keras

Keras [252] plays a special role among the software frameworks discussed here. As a matter of fact, it is not a full-featured DL-library, much rather Keras can be considered as an interface to other libraries providing a high-level API, which was originally built for various backends including TensorFlow, Theano and the (now deprecated) Microsoft Cognitive Toolkit (CNTK). As of version 2.4, TensorFlow is the only supported framework. Keras, which is free and also open-source, is meant to further simplify experimentation with neural networks as compared to TensorFlow's lower level API.

9.2 PyTorch

PyTorch [253], which is a free and open-source library, which was originally released to public in January 2017.[249] As of 2022, PyTorch has evolved from a research-oriented DL-framework to a fully fledged environment for both scientific work and industrial applications, which, as of 2022 has caught up with, if not surpassed, TensorFlow in popularity. Primarily addressing researchers in its early days, PyTorch saw a rapid growth not least for the—at that time—unique feature of *dynamic* computational graphs, which allows for great flexibility and simplifies creation of complex network architectures. As opposed to its competitors as, e.g., TensorFlow, computational graphs, which represent compositions of mathematical operations and allow for automatic differentiation of complex expressions, are created on the fly, i.e., at the very same time as operations are performed. *Static* graphs, on the other hand, need to be created in a first step, before they can be evaluated and automatically differentiated. Some examples of applying PyTorch to computational mechanics are provided in the next two remarks.

Remark 9.1. *Reinforcement Learning (RL)* is a branch of machine-learning, in which computational methods and DL-methods naturally come together. Owing to the progress in DL, reinforcement learning, which has its roots in the early days of cybernetics and machine learning, see, e.g., the survey [256], has gained attention in the fields of automatic control and robotics again. In their opening to a more recent review, the authors of [257] expect no less than *"deep reinforcement-learning is poised to revolutionize artificial intelligence and represents a step toward building autonomous systems with a higher level understanding of the visual world."* RL is based on the concept that an autonomous agent learns complex tasks by trial-and-error.

---

[249] See this blog post on the history of PyTorch [254] and the YouTube talk of Yann LeCun, PyTorch co-creator Sousmith Chintala, Meta's PyTorch lead Lin Qiao and Meta's CTO Mike Schroepfer [255].

Interacting with its environment, the agent receives a reward if it succeeds in solving a given tasks. Not least to speed up training by means of the parallelization, simulation has become are key ingredient to modern RL, where agents are typically trained in virtual environments, i.e., simulation models of the physical world. Though (computer) games are classical benchmarks, in which DeepMind's AlphaGo and AlphaZero models excelled humans (see Section 1, Figure 2), deep RL has proven capable of dealing with real-world applications in the field of control and robotics, see, e.g., [258]. Based on the PyTorch's introductory tutorial (see Original website) of a the classic cart-pole problem, i.e., an inverted pendulum (pole) mounted to a moving base (cart), we developed a RL-model for the control of large-deformable beams, see Figure 92 and the video illustrating the training progress. For some large-deformable beam formulations, see, e.g., [259], [260], [261], [262]. ∎
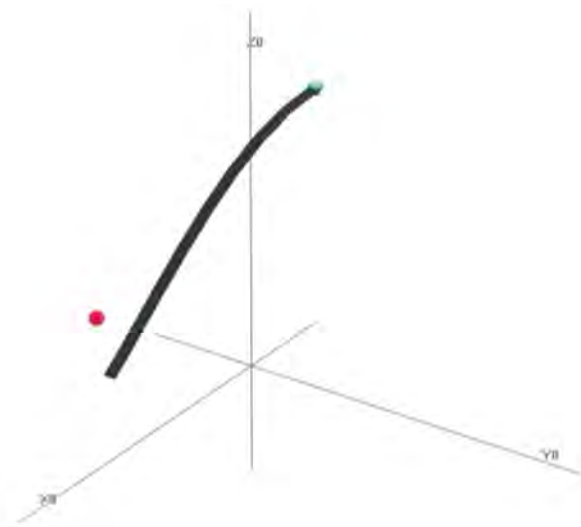
Figure 92: *Positioning and pointing control of large deformable beam* (Section 9, Remark 9.1). Reinforcement learning. The agent is trained to align the tip of the flexible beam with the target position (red ball). For this purpose, the agent can move the base of the cantilever; the environment returns the negative Euclidean distance of the beam's tip to the target position as "reward" in each time-step of the simulation. Simulation video. See also GitLab repository for code and video.

9.3 JAX

JAX [263] is a free and open-source research project driven by Google. Released to the public in 2018, JAX is one of the more recent software frameworks that have emerged during the current wave of AI. It is being described as being "Autograd and XLA" and as "a language for expressing and composing transformations of numerical programs", i.e., JAX focuses on accelerating evaluations of algebraic expression and, in particular, gradient computations. As a matter of fact, its core API, which provides a mostly NumPy-compatible interface to many mathematical operations, is rather trimmed-down in terms of DL-specific functions as compared to the broad scope of functionality offered by TensorFlow and PyTorch, for instance, which, for this reason, are often referred to as *end-to-end* frameworks. JAX, on the other hand, considers itself as system that facilitates "transformations" like gradient computation, just-in-time compilation and automatic vectorization of compositions of functions on parallel hardware as GPUs and TPUs. A higher-level interface to JAX' functionality, which is specifically made for ML-purposes, is available through the *FLAX* framework [264]. FLAX rovides many fundamental building blocks essential for creation and training of neural networks.

9.4 Leveraging DL-frameworks for scientific computing

Software frameworks for deep-learning as, e.g., PyTorch, TensorFlow and JAX share several features which are also essential in scientific computing, in general, and finite-element analysis, in particular. These DL-frameworks are highly optimized in terms of vectorization and parallelization of algebraic operations. Within finite-element methods, parallel evaluations can be exploited in several respects: First and foremost, residual vectors and (tangent) stiffness matrices need to be repeatedly evaluated for all elements of finite-element mesh, into which the domain of interest is discretized. Secondly, the computation of each of these vectors and matrices is based upon numerical quadrature (see Section 10.3.2 for a DL-based approach to improve quadrature), which, from an algorithmic point of view, is computed as a weighted sum of integrands evaluated at a finite set of points. A further key component that proves advantages in complex FE-problems is automatic differentiation, which, in the form of backpropagation (i.e., reverse-mode automatic differentiation), is the backbone of gradient-based training of neural networks, see Section 5. In the context of FE-problems in solid mechanics, automatic differentiation saves us from deriving and implementing derivatives of potentials, whose variation and linearization with respect to (generalized) coordinates give force vectors and tangent-stiffness matrices.[250]

The potential of modern DL-software frameworks in conventional finite-element problems was studied in [266], where Netgen/NGSolve, which is a highly-optimized, OpenMP-parallel finite-element code written in C++, was compared against PyTorch and JAX implementations. In particular, the computational efficiency of the computing and assembling vectors of internal forces and tangent stiffness matrices of a hyperelastic solid was investigated. On the same (virtual) machine, it turned out that both PyTorch and JAX can compete with Netgen/NGSolve when computations are performed on CPU, see the timings shown in Figure 93. Moving computations to a GPU, the Python-based DL-frameworks outperformed Netgen/NGSolve in the evaluation of residual vectors. Regarding tangent-stiffness matrices, which are obtained through (automatic) second derivatives of the strain-energy function with respect to nodal coordinates, both PyTorch and JAX showed (different) bottlenecks, which, however, are likely to be sorted out in future releases.

9.5 Physics-Informed Neural Network (PINN) frameworks

In laying out the roadmap for "Simulation Intelligence" (SI) the authors of [267] considered PINN as a key player in the first of the nine SI "motifs," called "Multi-physics & multi-scale modeling."

The PINN method to solve differential equations (ODEs, PDEs) aims at training a neural network to minimize the total weighted loss L in Figure 94, which describes the PINN concept in more technical details. As an example, the PDE residual L_{PDE} for incompressible flow can be written as follows [268]:

$$L_{\text{PDE}} = \frac{1}{N_f} \sum_{i=1}^{N_f} \sum_{j=1}^{4} \left[r_j(\boldsymbol{x}_f^{(i)}, t_f^{(i)}) \right]^2 \, , \text{ with} \tag{381}$$

$$\boldsymbol{r} = [r_1, r_2, r_3]^T = \frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \boldsymbol{\cdot} \nabla) \, \boldsymbol{u} + \nabla p - \frac{1}{Re} \nabla^2 \boldsymbol{u} \in \mathbb{R}^{3 \times 1} \, , \text{ and } r_4 = \nabla \boldsymbol{\cdot} \boldsymbol{u} \, , \tag{382}$$

where N_f is the number of residual (collocation) points, which could be in the millions generated randomly,[251] the residual $\boldsymbol{r} = [r_1, r_2, r_3]^T$ in Eq. (382)$_1$ is the left-hand side of the balance of linear momentum

---

[250] GPU-computing and automatic differentiation are by no means new to scientific computing not least in the field of computational mechanics. *Project Chrono* (see Original website), for instance, is well known for its support of GPU-computing in problems of flexible multi-body and particle systems. The general purpose finite-element code *Netgen/NGSolve* [265] (see Original website) offers a great degree of flexibility owing to its automatic differentiation capabilities. Well-established commercial codes, on the other hand, are often built on a comparatively old codebase, which dates back to times before the advent of GPU-computing.

[251] For incompressible flow past a cylinder, the computational domain of dimension $[-7.5, 28.5] \times [-20, 20] \times$
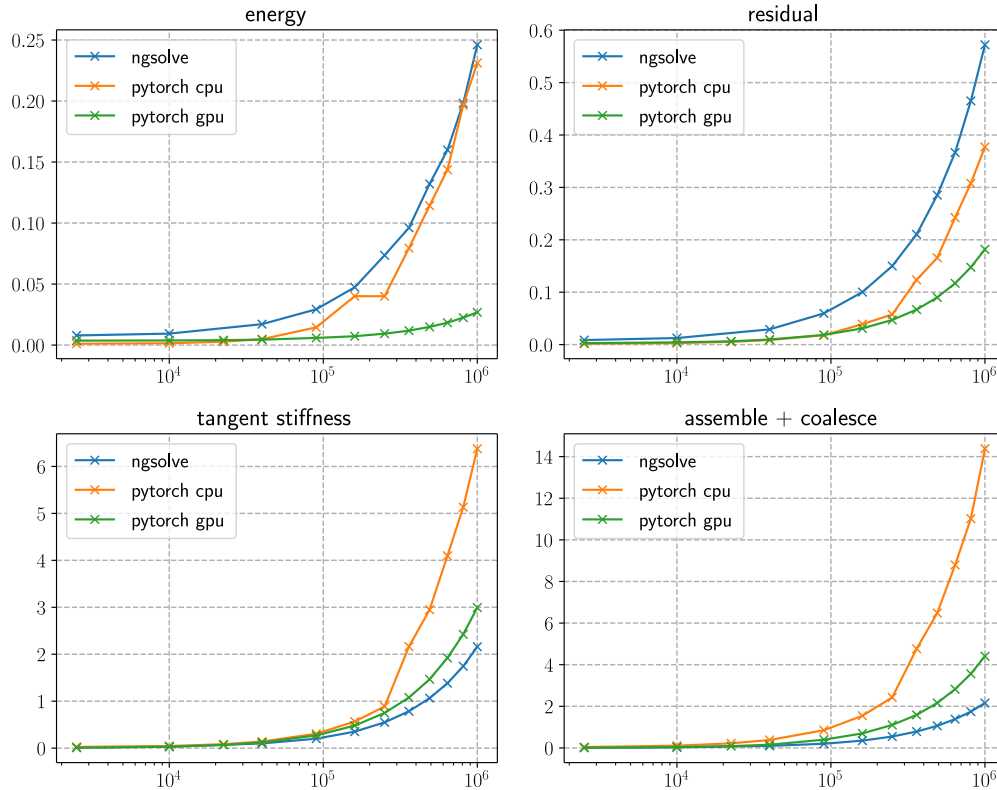
Figure 93: *DL-frameworks in nonlinear finite-element problems* (Section 9.4). The computational efficiency of a PyTorch-based (Version 1.8) finite-element code implemented was compared against the state-of-the-art general purpose Netgen/NGSolve [265] for a problem of nonlinear elasticity, see the slides of the presentation and the corresponding video. The figures show timings (in seconds) for evaluations of the strain energy (top left), the internal forces (residual, top right) and element-stiffness matrices (bottom left) and the global stiffness matrix (bottom right) against the number of elements. Owing to PyTorch's parallel computation capacity, the simple Python implementation could compete with the highly-optimized finite-element code, in particular, as computations were moved to a GPU (NVIDIA Tesla V100).

(the right-hand side being zero), and the residual r_4 in Eq. (382)$_2$ is the left-hand side of the incomrepssibility constraint, and $(\boldsymbol{x}_f^{(i)}, t_f^{(i)})$ the space-time coordinates of the collocation point (i).

Some review papers on PINN are [269] [270], with the latter being more general than [268], which was restricted to fluid mechanics, and touching on many different fields. Table 6 lists PINN frameworks that are currently actively developed, and a few selected *solvers* among which are summarized below.

☛ DeepXDE [271], one of the first PINN framework and a *solver* (Table 6) was developed in Python, with a TensorFlow backend, for both teaching and research. This framework can solve both forward problems ("given initial and boundary conditions") and inverse problems (" given some extra measures"), with domains having complex geometry. According to the authors, DeepXDE is user-friendly, with compact user code resembling the problem mathematical formulation, customizable to different types of mechanics problem. The site contains many published papers with a large number of demo problems: Poisson equation, Burgers equation, diffusion-reaction equation, wave propagation equation, fractional PDEs, etc. In addi-

---

[0, 12.5]—with coordinates (x, y, z) non-dimensionalized by the diameter of the cylinder, with axis along the z direction, and going through the point $(x, y) = (0, 0)$—contained 3×10^6 residual (collocation) points [268].
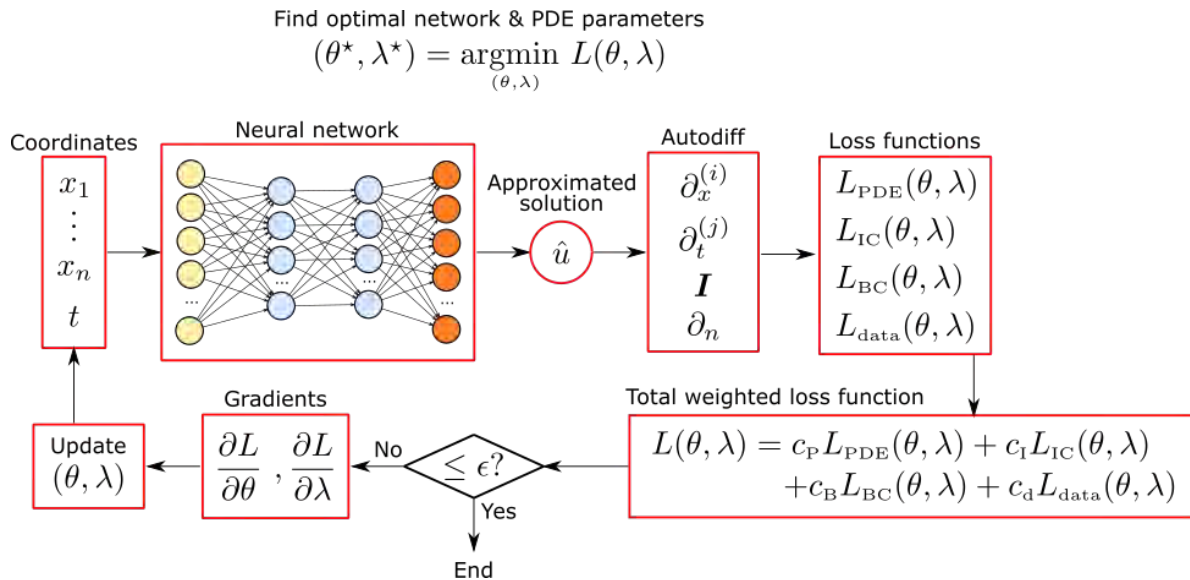
Figure 94: *Physics-Informed Neural Networks (PINN) concept* (Section 9.5). The goal is to find the optimal network parameters $\theta^\star$ (weights) and PDE parameters $\lambda^\star$ that minimize the total weighted loss function $L(\theta, \lambda)$, which is a linear combination of four loss functions: (1) The residual of the PDE, L_{PDE}, (2) Loss due to initial conditions, L_{IC}, (3) Loss due to boundary conditions , L_{BC}, (4) Loss due to known (labeled) data, L_{data}, with $c_{\mathrm{P}}, c_{\mathrm{I}}, c_{\mathrm{B}}, c_{\mathrm{d}}$ being the combination coefficients. With the space-time coordinates $(x_1, \ldots, x_n, t)$ as inputs, the neural network produces an approximated multi-physics solution $\hat{u} = \{u, v, p, \phi\}$, of which the derivatives, estimated by automatic differentiation, are used to evaluate the loss functions. If the total loss L is not less than a tolerance, its gradients with respect to the parameters (θ, λ) are used to update these parameters in a descent direction toward a local minimum of L [268].

tion, there are demos on inverse problems and operator learning. Three more backends beyond TensorFlow, which was reported in [270], have been added to DeepXDE: PyTorch, JAX, Paddle.[252]

☛ NeuroDiffEq [274], a *solver*, was developed about the same time as DeepXDE, with the backend being PyTorch. Even though it was written that the authors were "actively working on extending NeuroDiffEq to support three spatial dimensions," this feature is not ready, and can be worked around by including the 3D boundary conditions in the loss function.[253] Even though in principle, NeuroDiffEq can be used to solve PDEs of interest to engineering (e.g., Navier-Stokes solutions), there were no such examples in the official documentation, except for a 2D Laplace equation and a 1D heat equation. The backend is limited to PyTorch, and the site did not list any papers, either by the developers or by others, using this framework.

☛ NeuralPDE [275], a *solver*, was developed in a relatively new language Julia, which is 20 years younger than Python, has a speed edge over Python in machine learning, but does not in data science.[254] Demos are

---

[252] See Section Working with different backends. See also [269].

[253] "All you need is to import `GenericSolver` from `neurodiffeq.solvers`, and `Generator3D` from `neurodiffeq.generators`. The catch is that currently there is no reparametrization defined in `neurodiffeq.conditions` to satisfy 3D boundary conditions," which can be hacked into the loss function "by either adding another element in your equation system or overwriting the `additional_loss` method of `GenericSolve`." Private communication with a developer of NeuroDiffEq on 2022.10.08.

[254] There are many comparisons of Julia versus Python on the web; one is Julia vs Python: Which is Best to Learn First?, by By Zulie Rane on 2022.02.05, updated on 2022.10.01.

Table 6: *PINN frameworks* (Section 9.5) being actively developed [270] [269]. A *solver* solves the problem defined by users. A *wrapper* does not solve, but only wraps low-level functions from other libraries (e.g., PyTorch) into high-level functions that are convenient for users to implement PINN to solve the problem.

Framework site & Ref.	Usage	Language	Backend
DeepXDE [271]	Solver	Python	TensorFlow, PyTorch, JAX, Paddle
NVIDIA Modulus (SimNet) [272]	Solver	Python	TensorFlow
PyDEns [273]	Solver	Python	TensorFlow
NeuroDiffEq [274]	Solver	Python	PyTorch
NeuralPDE [275]	Solver	Julia	Julia
SciANN [276]	Wrapper	Python	TensorFlow
ADCME [277]	Wrapper	Julia	TensorFlow
GPytorch [278]	Wrapper	Python	PyTorch
Neural Tangents [279]	Wrapper	Python	JAX

given for ODEs, generic PDEs, such as the coupled nonlinear hyperbolic PDEs of the form:

$$\frac{\partial^2 u}{(\partial t)^2} = \frac{a}{n}\frac{\partial}{\partial x}(x^n\frac{\partial u}{\partial x}) + uf(\frac{u}{w}) , \quad \frac{\partial^2 w}{(\partial t)^2} = \frac{b}{n}\frac{\partial}{\partial x}(x^n\frac{\partial u}{\partial x}) + wf(\frac{u}{w}) , \tag{383}$$

with f and g being arbitrary functions. There are initial and boundary conditions, and exact solution to find the error of the numerical solution, Figure 95. The site did not list any papers, either by the developers or by others, using this framework.
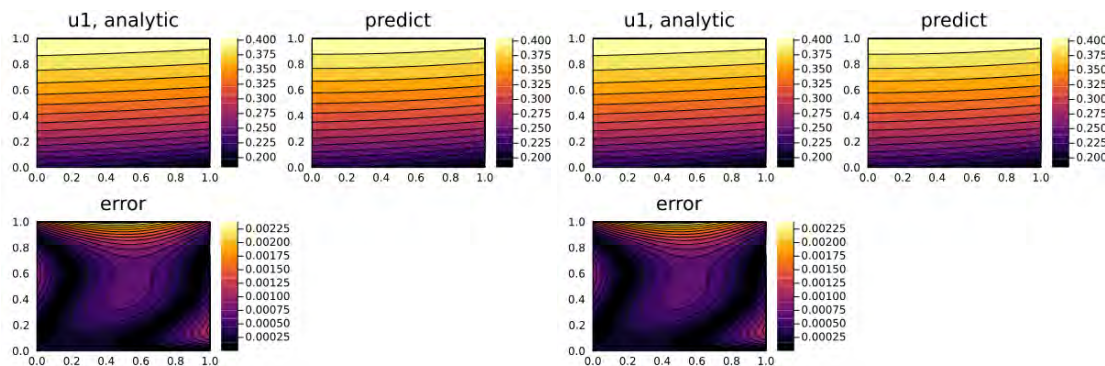
Figure 95: Coupled nonlinear hyperbolic equations (Section 9.5). Analytical solution, predicted solution by NeuralPDE [275] and error for the coupled nonlinear hyperbolic equations in Eq. (383).

Additional PINN software packages other than those in Table 6 are listed and summarized in [269].

Remark 9.2. *PINN and activation functions.* Deep neural networks (DNN), having at least two hidden layers, with ReLU activation function (Figure 24) were shown to correspond to linear finite element interpolation [280], since piecewise linear functions can be written as DNN with ReLU activation functions [281].

But using the *strong* form such as the PDE in Eq. (383), which has the second partial derivative with respect to x, and since the second derivative of ReLU is zero, activation functions such as the logistic sigmoid (Figure 30), hyperbolic tangent (Figure 31), or the swish function (Figure 139) are recommended.

Because of the presence of the second partial derivatives with respect to the spatial coordinates in the general PDE Eq. (384), particularized to the 2D Navier-Stokes Eq. (385):

$$\boldsymbol{u}_t + \mathcal{N}[\boldsymbol{u}; \boldsymbol{\lambda}] = \boldsymbol{0} \ \text{ for } \boldsymbol{x} \in \Omega \, t \in [0, T] \, , \text{ with} \tag{384}$$

$$\boldsymbol{u}_t = \begin{Bmatrix} u_t \\ v_t \end{Bmatrix} \, , \quad \mathcal{N}[\boldsymbol{u}; \boldsymbol{\lambda}] = \begin{Bmatrix} \lambda_1(uu_x + vu_y) + p_x - \lambda_2(u_{xx} + u_{yy}) \\ \lambda_1(uv_x + vv_y) + p_y - \lambda_2(v_{xx} + v_{yy}) \end{Bmatrix} \, , \quad \boldsymbol{\lambda} = \begin{Bmatrix} \lambda_1 \\ \lambda_2 \end{Bmatrix} \, , \tag{385}$$

for which the hyperbolic tangent (tanh) was used as activation function [282]. ∎

Remark 9.3. *Variational PINN.* Similar to the finite element method, in which the *weak* form, not the strong form as in Remark 9.2, of the PDE allows for a reduction in the requirement of differentiability of the trial solution, and is discretized with numerical integration used to evaluate the resulting coefficients for various matrices (e.g, mass, stiffness, etc), PINN can be formulated using the *weak* form, instead of the strong form such as Eq. (385), at the expense of having to perform numerical integration (quadrature) [283] [284] [285].

Examples of 1-D PDEs were given in [283] in which the activation function was a sine function defined over the interval $(-1, 1)$. To illustrate the concept, consider the following simple 1-D problem [283] (which could model the axial displacement $u(x)$ of an elastic bar under distributed load $f(x)$, with prescribed end displacements):

$$-u''(x) = -\frac{d^2u(x)}{(dx)^2} = f(x), \text{ for } x \in (-1, 1) \, , \quad u(-1) = g \, , \quad u(1) = h \, , \tag{386}$$

where g and h are constants. Three variational forms of the strong form Eq. (386) are:

$$\text{A}_k(u, v) = \text{B}(f, v) := \int_{-1}^{+1} f(x)v(x)dx \, , \forall v \text{ with } v(-1) = v(+1) = 0 \, , \text{ and} \tag{387}$$

$$\text{A}_1(u, v) := -\int_{-1}^{+1} u''(x)v(x)dx \, , \tag{388}$$

$$\text{A}_2(u, v) := \int_{-1}^{+1} u'(x)v'(x)dx \, , \tag{389}$$

$$\text{A}_3(u, v) := -\int_{-1}^{+1} u(x)v''(x)dx + u(x)v'(x)\big|_{x=-1}^{x=+1} \, , \tag{390}$$

where the familiar symmetric operator A_2 in Eq. (389) is the weak form, with the non-symmetric operator A_1 in Eq. (388) retaining the second derivation on the solution $u(x)$, and the non-symmetric operator A_3 in Eq. (390) retaining the second derivative on the test function $v(x)$ in addition to the boundary terms. Upon replacing the solution $u(x)$ by its neural network (NN) approximation $u_{NN}(x)$ obtained using one single hidden layer ($L = 1$ in Figure 23) with $y = f^{(1)}(x)$ and layer width $m_{(1)}$, and using the sine activation function on the interval $(-1, +1)$:

$$u_{NN}(x) = y(x) = f^{(1)}(x) = \sum_{i=1}^{m_{(1)}} u_{NN_i}(x) = \sum_{i=1}^{m_{(1)}} c_i \sin(w_i x + b_i) \, , \tag{391}$$

which does not satisfy the essential boundary conditions (whereas the solution $u(x)$ does), the loss function for the VPINN method is then the squared residual of the variational form plus the squared residual of the essential boundary conditions:

$$L(\theta) = [\text{A}_k(u_{NN}, v) - \text{B}(f, v)]^2 + [u_{NN}(x) - u(x)]_{x \in \{-1, +1\}}^2 \, , \text{ and} \tag{392}$$

$$\theta^\star = \{\boldsymbol{w}^\star, \boldsymbol{b}^\star, \boldsymbol{c}^\star\} = \underset{\theta}{\operatorname{argmin}} \, L(\theta) \, , \tag{393}$$

where $\theta^\star = \{\boldsymbol{w}^\star, \boldsymbol{b}^\star, \boldsymbol{c}^\star\}$ are the optimal parameters for Eq. (391), with the goal of enforcing the variational form and essential boundary conditions in Eq. (387). More details are in [283] [284].

For a symmetric variational form such as $\mathtt{A}_2(u, v)$ in Eq. (389), a potential energy exists, and can be minimized to obtain the neural approximate solution $u_{NN}(x)$:

$$J(u) = \frac{1}{2}\mathtt{A}_2(u, u) - \mathtt{B}(f, u) , \quad \tilde{J}(\theta) = \frac{1}{2}\mathtt{A}_2(u_{NN}, u_{NN}) - \mathtt{B}(f, u_{NN}) , \tag{394}$$

$$\theta^\star = \{\boldsymbol{w}^\star, \boldsymbol{b}^\star, \boldsymbol{c}^\star\} = \underset{\theta}{\operatorname{argmin}} \, L(\theta) , \quad L(\theta) = \tilde{J}(\theta) + [u_{NN}(x) - u(x)]^2_{x \in \{-1, +1\}} , \tag{395}$$

which is similar to the approach taken in [280], where the ReLU activation function (Figure 24) was used, and where a constraint on the NN parameters was used to satisfy an essential boundary condition. ∎

Remark 9.4. *PINN, kernel machines, training, convergence problems.* There is a relationship between PINN and kernel machines in Section 8. Specifically, the neural tangent kernel [232], which "captures the behavior of fully-connected neural networks in the infinite width limit during training via gradient descent" was used to understand when and why PINN failed to train [286], whose authors found a "remarkable discrepancy in the convergence rate of the different loss components contributing to the total training error," and proposed a new gradient descent algorithm to fix the problem.

It was often reported that PINN optimization converged to "solutions that lacked physical behaviors," and "reduced-domain methods improved convergence behavior of PINNs"; see [287], where a dynamical system of the form below was studied:

$$\frac{du_{NN}(x)}{dx} = f(u_{NN}(x)) , \quad L(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\frac{du_{NN}(x^{(i)})}{dx} - f(u_{NN}(x^{(i)}))\right]^2 , \tag{396}$$

with $L(\theta)$ being called the "physics" loss function. An example with $f(u(x)) = u(x)\left[1 - u^2(x)\right]$ was studied to show that the "physics loss optimization predominantly results in convergence issues, leading to incorrectly learned system dynamics"; see [287], where it was found that "solutions corresponding to nonphysical system dynamics [could] be dominant in the physics loss landscape and optimization," and that "reducing the computational domain [lowered] the optimization complexity and chance of getting trapped with nonphysical solutions."

See also [288] for incorporating the Lyapunov stability concept into PINN formulation for CFD to "improve the generalization error and reduce the prediction uncertainty." ∎

Remark 9.5. *PINN and attention architecture.* In [228], PIANN, Physics-Informed Attention-based Neural Network, was proposed to connect PINN to attention architecture, discussed in Section 7.4.3 to solve hyperbolic PDE with shock wave. See Remark 7.7 and Remark 11.11. ∎

Remark 9.6. *"Physics-Informed Learning Machine" (PILM) 2021 US Patent* [289]. First note that the patent title used the phrase "learning machine," instead of "machine learning," indicating that the emphasis of the patent appeared to be on "machine," instead of on "learning" [289]. PINN was not mentioned, as it was first invented in [290] [291], which were cited by the patent authors in their original PINN paper [282].[255] The abstract of this 2021 PILM US Patent [289] reads as follows:

> "A method for analyzing an object includes modeling the object with a differential equation, such as a linear partial differential equation (PDE), and sampling data associated with the differential equation. The method uses a probability distribution device to obtain the solution to the differential equation. The method eliminates use of discretization of the differential equation."

---

[255] The 2019 paper [282] was a merger of a two-part preprint [292] [293].

The first sentence is nothing new to the readers. In the second sentence, a "probability distribution device" could be replaced by a neural network, which would make PILM into PINN. This patent mainly focused on the Gaussian processes (Section 8.3), as an exemple of probability distribution (see Figure 4 in [289]). The third sentence would be the claim-to-fame of PILM, and also of PINN.   ∎

Remark 9.7. *Using PINN frameworks.* While undergraduates with limited knowledge on the theory of the Finite Element Method could run FE Analysis of complicated structures and complex domain geometries on a laptop using commercial FE codes, solving problems with exceedingly simple domain geometry using a PINN framework such as DeepXDE does require knowledge of governing PDEs, initial and boundary conditions, artificial neural networks and frameworks (such as PyTorch, TensorFlow, etc.), the Python language, and having a more powerful computer. In addition, because there are many parameters to fiddle with, beyond the sample problems posted on the DeepXDE website, first-time users could encounter disappointment and doubt when trying to solve a new problem. It is not clear when the PINN methods would reach the level of FE commercial codes that undergraduates could use, or would they just fade away after an initial period of excitement like the meshless methods before them.   ∎

10 Application 1: Enhanced numerical quadrature for finite elements

The results and deep-learning concepts used in [38] were presented in Section 2.3.1 above. In this section, we discuss some details of the formulation.

The finite element method (FEM) has become the most important numerical method for the approximation of solutions to partial differential equations, in particular, the governing equations in solid mechanics. As for any mesh-based method, the discretization of a continuous (spatial, temporal) domain into a finite-element mesh, i.e., a disjoint set of finite elements, is a vital ingredient that affects the quality of the results and therefore has emerged as a field of research on its own. Being based on the weak formulation of the governing balance equations, numerical integration is a second key ingredient of FEM, in which integrals over the physical domain of interest are approximated by the sum of integrals over the individual finite elements. In real-world problems, regularly shaped elements, e.g., triangles and rectangles in 2-D, tetrahedra and hexahedra in 3-D, typically no longer suffice to represent the complex shape of bodies or physical domains. By distorting basic element shapes, finite elements of more arbitrary shapes are obtained, while the interpolation functions of the "parent" elements can be retained.[256] The mapping represents a coordinate transformation by which the coordinates of a "parent" or "reference" element are mapped onto distorted, possibly curvilinear, physical coordinates of the actual elements in the mesh. Conventional finite element formulations use polynomials as interpolation functions, for which Gauss-Legendre quadrature is the most efficient way to integrate numerically. Efficiency in numerical quadrature is immediately related the (finite) number of integration points that is required to exactly integrate a polynomial of a given degree.[257] For distorted elements, the Jacobian of the transformation describing the distortion of the parent element renders the integrand of, e.g., the element stiffness matrix non-polynomial. Therefore, the integrals generally are not integrated exactly using Gauss-Legendre quadrature, but the accuracy depends, roughly speaking, on the degree of distortion.

10.1 Two methods of quadrature, 1-D example

To motivate their approaches, the authors of [38] presented an illustrative 1-D example of a simple integral, which was analytically integrated as

$$\int_{-1}^{1} x^{10} \mathrm{d}x = \left[\frac{1}{11} x^{11} \right]_{-1}^{1} = \frac{2}{11}. \tag{397}$$

---

[256] See, e.g., [39], p.61, Section "3.5 Isoparametric form" and p.170, Section "6.5 Mapping: Parametric forms."
[257] Using Gauss-Legendre quadrature, p integration points integrate polynomials up to a degree of $2p - 1$ exactly.

Using 2 integration points, Gauss-Legendre quadrature yields significant error

$$\int_{-1}^{1} x^{10} \mathrm{d}x \approx 1\left(-\frac{1}{\sqrt{3}}\right)^{10} + 1\left(\frac{1}{\sqrt{3}}\right)^{10} = \frac{2}{243}, \tag{398}$$

which is owed to the insufficient number of integration points.

Method 1 to improve accuracy, which is reflected in Application 1.1 (see Section 10.2), is to increase the number of integration points. In the above example, 6 integration points are required to obtain the exact value of the integral. By increasing the accuracy, however, we sacrifice computational efficiency due the need for 6 evaluations of the integrand instead of the original 2 evaluations.

Method 2 is to retain 2 integration points, and to adjust the quadrature weights at the integration points instead. If the same quadrature weights of $243/11$ are used instead of 1, the integral evaluates to the exact result:

$$\int_{-1}^{1} x^{10} \mathrm{d}x = \frac{243}{11}\left(-\frac{1}{\sqrt{3}}\right)^{10} + \frac{243}{11}\left(\frac{1}{\sqrt{3}}\right)^{10} = \frac{2}{11}. \tag{399}$$

By adjusting the quadrature weights rather than the number of integration points, which is the key concept of Application 1.2 (see Section 10.3), the computational efficiency of the original approach Eq. (398) is retained.

In this study [38], hexahedral elements with linear shape functions were considered. To exactly integrate the element stiffness matrix of an undistorted element $2 \times 2 \times 2 = 8$ integration points are required.[258] Both applications summarized subsequently required the integrand, i.e., the element shape to be identified in an unique way. Gauss-Legendre quadrature is performed in the local coordinates of the reference element with accuracy invariant to both rigid-body motion and uniform stretching of the actual elements. To account for these invariances, a "normalization" of elements was proposed in [38] (Figure 96), i.e., hexahedra were re-located to the origin of a frame of reference, re-oriented along with the coordinate planes and scaled by means of the average length of two of its edges.

To train the neural networks involved in their approaches, a large set of distorted elements was created by randomly displacing seven nodes of a regular cube [38],

$$B(1 \pm r_1 d, 0, 0), \quad C(1 \pm r_2 d, 1 \pm r_3 d, \pm r_4 d), \quad D(\pm r_5 d, 1 \pm r_6 d, 0), \quad E(\pm r_7 d, \pm r_8 d, 1 \pm r_9 d),$$

$$F(1 \pm r_{10} d, \pm r_{11} d, 1 \pm r_{12} d), \quad G(1 \pm r_{13} d, 1 \pm r_{14} d, 1 \pm r_{15} d), \quad H(\pm r_{16} d, 1 \pm r_{17} d, 1 \pm r_{18} d),$$

$$\tag{400}$$

where $r_i \in [0, 1]$, $i = 1, \ldots, 18$ were 18 random numbers, see Figure 97. The elements were collected into five groups according to five different degrees of maximum distortion (maximum *possible* nodal displacement) $d \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$. Elements in the set $d = 0.5$ would only be highly distorted with r_i having values closer to 1, but may only be slightly distorted with r_i having values closer to 0.[259] To avoid

---

[258] Conventional linear hexahedra are known to suffer from locking, see, e.g., [39] Section "10.3.2 Locking," which can be alleviated by "reduced integration," i.e., using a single integration point ($1 \times 1 \times 1$). The concept in [38], however, immediately translates to higher-order shape functions and non-conventional finite element formulations (e.g., mixed formulation).

[259] In fact, a somewhat ambiguous description of the random generation of elements was provided in [38]. On the one hand, the authors stated that a "... *coordinates of nodes are changed using a uniform random number r* ...," and did not distinguish the random numbers in Eq. (400) by subscripts. On the other hand, they noted that exaggerated distortion may occur if nodal coordinates of an element were changed independently, and introduced the constraints on the distortion mentioned above in that context. If the same random number r were used for all nodal coordinates, all elements generated would exhibit the same mode of distortion.
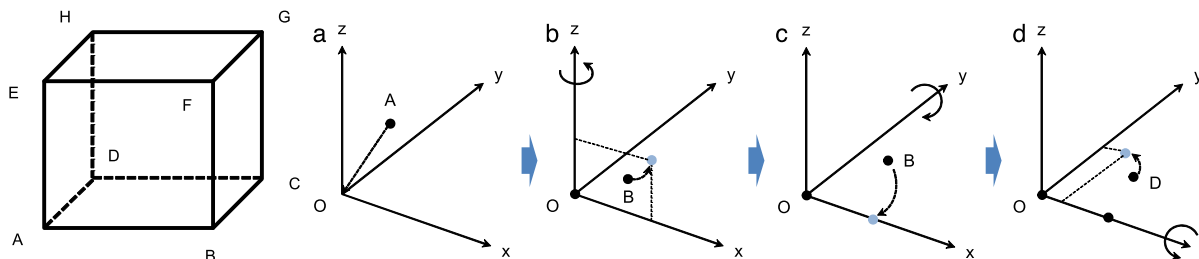
Figure 96: *Normalization procedure for hexahedra* (Section 10.1). Numerical integration is performed in local element coordinates. The accuracy of Gauss-Legendre quadrature only depends on the element shape, i.e., it is invariant with respect to rigid-body motion and uniform stretching deformation. For this reason, a normalization procedure was introduced involving one translation and three rotations (second from left to right) for linear hexahedral elements, whose nodes are labelled as shown for the regular hexahedron (left) [38]. (1) The element is displaced such that node A coincides with the origin O of the global frame. (2) The hexahedron is rotated about the z-axis of the global frame to place node B in the xz-plane, and then (3) rotated about the y-axis such that node B lies on the x-axis. (4) A third rotation about the x-axis relocates node D to the xy-plane of the global frame. (5) Finally (not shown), the element is scaled by a factor of $1/l_0$, where $l_0 = (AB + AD)/2$. (Figure reproduced with permission of the authors.)

large distortion, the displacement of each node was restricted to the range of 0.5 to 2, and the angle between adjacent faces must lie within the range of 90° to 120°. Applying the normalization procedure, an element was characterized by a total of 18 nodal coordinates randomly distributed, but in a specific manner according to Eq. (400).

To quantify the quadrature error, the authors of [38] introduced $e(q)$, a relative measure of accuracy for the components of the stiffness matrix k_{ij} as a function of the number integration points q along each local coordinate,[260] defined as

$$e(q) = \frac{1}{\max_{i,j} |k_{ij}^{q_{\max}}|} \sum_{i,j} |k_{ij}^q - k_{ij}^{q_{\max}}| , \tag{401}$$

where k_{ij}^q denotes the component in the i-th row and j-th column of the stiffness matrix obtained with q integration points. The error $e(q)$ is measured with respect to reference values $k_{ij}^{q_{\max}}$, which are obtained using the Gauss-Legendre quadrature with $q_{\max} = 30$ integration points, i.e., a total of $30^3 = 27{,}000$ integration points for 3-D elements.

10.2 Application 1.1: Method 1, Optimal number of integration points

The details of this particular deep-learning application, mentioned briefly in Section 2.3 on motivation via applications of deep learning—specifically Section 2.3.1, item (1)—are provided here. The idea is to have a neural network predict, for each element (particularly distorted elements), the number of integration points that provides accurate integration within a given error tolerance e^{tol}, [38].

10.2.1 Method 1, feasibility study

In the example in [38], the quadrature error is required to be smaller than $e^{tol} = 1 \times 10^{-3}$. For this purpose, a fully-connected feed-forward neural network with N hidden layers of 50 neurons each was used. The non-trivial nodal coordinates were fed as inputs to the network, i.e., $\boldsymbol{x} = \boldsymbol{y}^{(0)} \in \mathbb{R}^{18 \times 1}$. This

---

[260] For example, for a $2 \times 2 \times 2$ integration with a total of 8 integration points, $q = 2$.
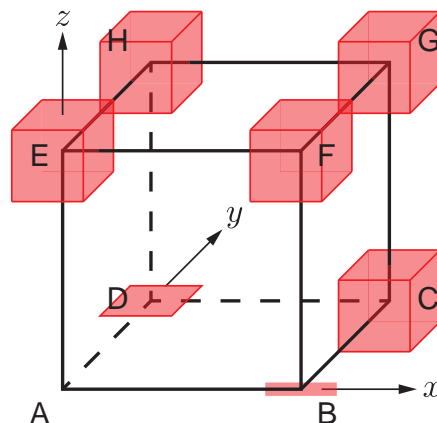
Figure 97: *Creation of randomly distorted elements* (Section 10). Hexahedra forming the train-ing and validation sets are created by randomly displacing the nodes of a regular hexahedral. To comply with the normalization procedure, node A remains fixed, node B is shifted along the x-axis and node C is displaced with the xy-plane. For each of the remaining nodes (E, F, G, H), all three nodal coordinates are varied randomly. The elements are grouped according to the maximum *possible* nodal displacement $d \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$, from which each of the 18 nodal displacements was obtained upon multiplication with a random number $r_i \in [0, 1]$, $i = 1, \ldots, 18$ [38], see Eq. (400). Each of the 8 nodal zones in which the corresponding node can be placed randomly is shown in red; all nodal zones are cubes, except for Node B (an interval on the x axis) and for Node C (a square in the plane (x, y)).

neural network performed a classification task,[261] where each class corresponded to the minimum number of integration points q along a local coordinate axis for a maximum error $e^{\text{tol}} = 10^{-3}$. Figure 98 presents the distribution of 10,000 elements generated randomly for two degrees of maximum possible distortion, $d = 0.1$ and $d = 0.5$, using the method of Figure 97, and classified the minimum number of integration points. Similar results for $d = 0.3$ were presented in [38], in which the conclusion that "as the shape is distorted, more integration points are needed" is expected.

10.2.2 Method 1, training phase

To train the network, 2000 randomly distorted elements were generated for each of the five degrees of maximum distortion, $d \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$, to obtain a total of 10,000 elements [38]. Before the network was trained, the minimum number of integration points q^{opt} to meet the accuracy tolerance e^{tol} was determined for each element.

The whole dataset was partitioned[262] into a training set $\mathbb{X} = \{\boldsymbol{x}^{|1|}, \cdots, \boldsymbol{x}^{|\mathsf{M}|}\}$, $\mathbb{Y} = \{\boldsymbol{y}^{|1|}, \cdots, \boldsymbol{y}^{|\mathsf{M}|}\}$ of $\mathsf{M} = 5000$ elements and an equally large validation set,[263] by which the training progress was monitored [38], as described in Section 6.1. The authors of [38] explored several network architectures (with depth

---

[261] The authors of [38] used the squared-error loss function (Section 5.1.1) for the classification task, for which the softmax loss function can also be used, as discussed in Section 5.1.3.

[262] There was no indication in [38] on how these 5,000 elements were selected from the total of 10,000 elements, perhaps randomly.

[263] In contrast to the terminology of the present paper, in which "training set" and "validation set" are used (Sec-tion 6.1), the terms "training patterns" and "test patterns", respectively, were used in [38]. The "test patterns" were used in the training process, since the authors of [38] "... *terminated the training before the error for test patterns started to increase*" (p.331). These "test patterns", based on their use as stated, correspond to the elements of the *validation set* in the present paper, Figure 100. Technically, there was no test set in [38]; see Section 6.1.
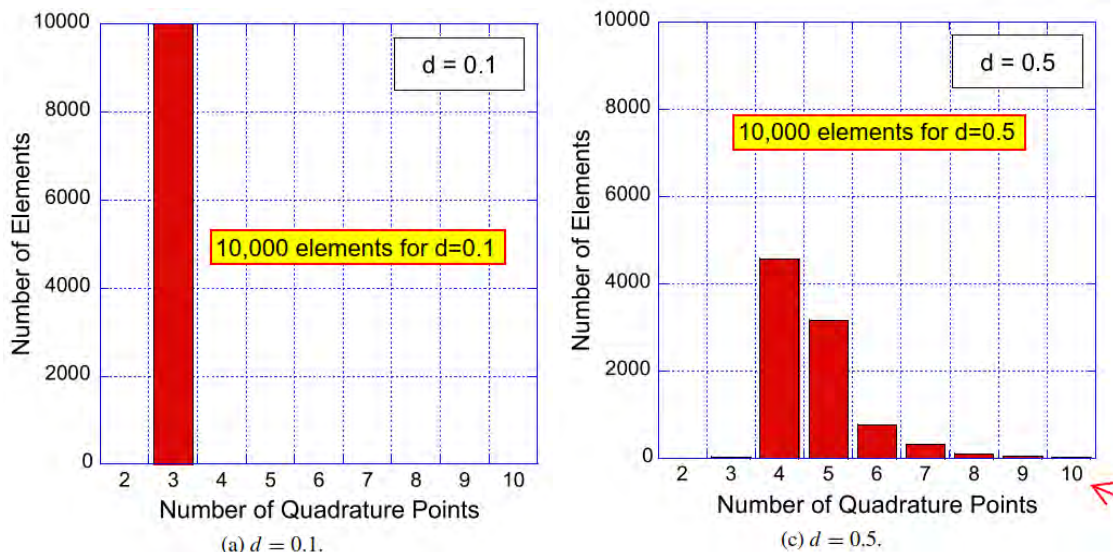
Figure 98: *Method 1, Optimal number of integration points, feasibility* (Section 10.2.1). Distribution of minimum numbers of integration points on a local coordinate axes for a maximum error of $e^{\text{tol}} = 10^{-3}$ among 10,000 elements generated randomly using the method in Figure 97. For $d = 0.1$, all elements were only slightly distorted, and required 3 integration points each. For $d = 0.5$, close to 5,000 elements required 4 integration points each; very few elements required 3 or 10 integration points [38]. (Figure reproduced with permission of the authors.)

ranging from 1 to 5, keeping the width fixed at 50) to determine the optimal structure of their classifier network, which used the logistic sigmoid as activation function.[264,265] Their optimal feed-forward network, composed of 3 hidden layers of 50 neurons each, correctly predicted the number of integration points needed for 98.6% of the elements in the training set, and for 81.6 % of the elements in the validation set, Figure 99.

10.2.3 Method 1, application phase

The correct number of quadrature points and the corresponding number of points predicted by the neural network are illustrated in Figure 100 for both the training set ("patterns") in Table (a) and the validation set ("test patterns") in Table (b).

Both the training set and the validation set, each had 5000 distorted element shapes. As an example to interpret these tables, take Table (a), Row 2 (red underline, labeled "3" in red circle) of the matrix (blue box): Out of a Total of 1562 element shapes (last column) in the training set that were ideally integrated using 3 quadrature points (in red circle), the neural network correctly estimated a need of 3 quadrature points (Column 2, labeled "3" in red circle) for 1553 element shapes, and 4 quadrature points (Column 3, labeled "4" in red circle) for 9 element shapes. That's an accuracy of 99.4% for Row 2. The accuracy varies, however, by row, 0% for Row 1 (2 integration points), 99.6% for Row 3, ..., 71.4% for Row 7, 0% for Row 8 (9 integration points). The numbers in column "Total" add up to 5 + 1562 + 2557 + 636 + 162 + 55 + 21 + 2 = 5000 elements in the training set. The diagonal coefficients add up to 1553 + 2548 + 616 + 153 + 46 + 15 = 4931 elements with correctly predicted number of integration points, yielding the overall accuracy

---

[264] The first author of [38] provided the information on the activation function through a private communication to the authors on 2018 Nov 16. Their tests with the ReLU did not show improved performance (in terms of accuracy) as compared to the logistic sigmoid.

[265] Even though the squared-error loss function (Section 5.1.1) was used in [38], we also discuss the softmax loss function for classification tasks; see Section 5.1.3.

Classification accuracy versus network architecture.

Number of hidden layers	Number of units per hidden layer	Results (%)	
		Training patterns	Test patterns
1	50	91.4	79.2
2	50	95.9	80.4
3	50	98.6	81.6
4	50	97.8	81.4
5	50	97.4	81.2

Figure 99: *Method 1, Optimal network architecture for training* (Section 10.2.2). The number of hidden layers varies from 1 to 5, keeping the number of neurons per hidden layer constant at 50. The network with 3 hidden layers provided the highest accuracy for both the training set ("patterns") at 98.6% and for the validation set ("test patterns") at 81.6%. Increase the network depth does not necessarily increase the accuracy [38]. (Table reproduced with permission of the authors.)

of 4931 / 5000 = 98.6% in training, Figure 99.

For Table (b) in Figure 100, the numbers in column "Total" add up to 5 + 1553 + 2574 + 656 + 135 + 56 + 21 + 5 = 5005 elements in the validation set, which should have 5000, as written by [38]. Was there a misprint ? The diagonal coefficients add up to 1430 + 2222 + 386 + 36 + 6 = 4080 elements with correctly predicted number of integration points, yielding the accuracy of 4080 / 5000 = 81.6%, which agrees with Row 3 in Figure 99. As a result of this agreement, the number of elements in the validation set ("test patterns") should be 5000, and not 5005, i.e., there was a misprint in column "Total".

10.3 Application 1.2: Method 2, optimal quadrature weights

The details of this particular deep-learning application, mentioned briefly in Section 2.3 on motivation via applications of deep learning, Section 2.3.1, item (2), are provided here. As an alternative to increasing the number of quadrature points, the authors of [38] proposed to compensate for the quadrature error introduced by the element distortion by adjusting the quadrature weights at a fixed number of quadrature points. For this purpose, they introduced correction factors $\{w_{i,j,k}\}$ that were predicted by a neural network, and were multipliers for the corresponding standard quadrature weights $\{H_{i,j,k}\}$ of an undistorted hexahedron. To exactly compute the components of the stiffness matrix, undistorted linear hexahedra require eight quadrature points ($i = j = k = 2$) at local positions $\xi, \eta, \zeta = \pm 1/\sqrt{3}$ with uniform weights $H_{i,j,k} = 1$.

The data preparation here was similar to that in Method 1, Section 10.2, except that 20,000 randomly distorted elements were generated, with 4000 elements in each of the five groups, each group having a different degree of maximum distortion $d \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$, as depicted in Figure 97 [38].

10.3.1 Method 2, feasibility study

Using the above 20,000 elements, the feasibility of improving integration accuracy by quadrature weight correction was established in Figure 101. To obtain these results, a brute-force search was used: For each of the 20,000 elements, 1 million sets of random correction factors $\{w_{i,j,k}\} \in [0.95, 1.05]$ were tested to find the optimal set of correction factors for that single element [38]. The effectiveness of quadrature weight correction was quantified by the error reduction ratio defined by

$$R_{\text{error}} = \frac{\sum_{i,j} |k_{ij}^{q,opt} - k_{ij}^{q_{\max}}|}{\sum_{i,j} |k_{ij}^{q} - k_{ij}^{q_{\max}}|}, \tag{402}$$

i.e., the ratio between the quadrature error defined in Eq. (401) obtained using the *optimal* ("*opt*") corrected quadrature weights and the quadrature error obtained using the standard quadrature weights of Gauss-Legendre quadrature. Accordingly, a ratio $R_{\text{error}} < 1$ indicates that the modified quadrature weights im-

(a) Training patterns

| | Number of quadrature points (estimated by neural network) | | | | | | | | |
Number of quadrature points (correct)	2	3	4	5	6	7	8	9	Total
2		5							5
3		1553	9						1562
4		7	2548	2					2557
5			20	616					636
6			5	2	153	2			162
7			2	5	1	46	1		55
8				5	1	1	15		21
9					1	1			2

(b) Test patterns

| | Number of quadrature points (estimated by neural network) | | | | | | | | |
Number of quadrature points (correct)	2	3	4	5	6	7	8	9	Total
2		5							5
3		1430	123						1553
4		135	2222	201	15	1			2574
5			206	386	56	7	1		656
6			16	70	36	13			135
7			1	29	19	6	1		56
8				10	8	3			21
9				3	2				5

Figure 100: *Method 1, application phase* (Section 10.2.3). The numbers of quadrature points predicted by the neural network was compared to the minimum numbers of quadrature points for maximum error $e^{\text{tol}} = 10^{-3}$ [38]. Table (a) shows the results for the training set ("patterns"), and Table (b) for the validation set. (Table reproduced with permission of the authors.)

proved the accuracy. For each element, the optimal correction factors that yielded the smallest error ratio, i.e.,

$$\{w_{i,j,k}^{opt}\} = \arg \min_{w_{i,j,k}} R_{\text{error}}, \tag{403}$$

were retained as target values for training, and identified with the superscript "*opt*", standing for "optimal". The corresponding optimally integrated coefficients in the element stiffness matrix are denoted by $k_{ij}^{q,opt}$, with $i, j = 1, 2, \cdots$, as appeared in Eq. (402).

It turns out that a reduction of the quadrature error by correcting the quadrature weights is not feasible for all element shapes. Undistorted elements, for instance, which are already integrated exactly using standard quadrature weights, naturally do not admit improvements. These 20,000 elements were classified into two categories A and B [38], Figure 101. Quadrature weight correction was not effective for Category A ($R_{\text{error}} \geq 1$), but effective for Category B ($R_{\text{error}} < 1$). Figure 101 shows that elements with a higher degree of maximum distortion were more likely to benefit from the quadrature weight correction as compared to weakly distorted elements. Recall that among the elements of the group $d = 0.5$ were elements that were only slightly distorted (due to the random factors r_{ij} in Eq. (400) being close to zero), and therefore would not benefit from quadrature weight correction (Category A); there were 1489 such elements, Figure 101.

10.3.2 Method 2, training phase

Because the effectiveness of the quadrature weight correction strongly depends on the degree of maximum distortion, d, the authors of [38] proposed a two-stage approach for the correction of quadrature weights, which relied on two fully-connected feedforward neural networks.

In the first stage, a *first* neural network, a binary classifier, was trained to predict whether an element shape admits improved accuracy by quadrature weight correction (Category B) or not (Category A). The

		Category		
		A	B	Total
	0.1	4000	0	4000
	0.2	3973	27	4000
d	0.3	3350	650	4000
	0.4	2320	1680	4000
	0.5	1489	2511	4000

Figure 101: *Method 2, Quadrature weight correction, feasibility* (Section 10.3.1). Each element was tested 1 million times with randomly generated sets of quadrature weights. There were 4000 elements in each of the 5 groups with different degrees of maximum distortion, d. Quadrature weight correction effectiveness increased with element distortion. Weakly distorted elements ($d = 0.1$) did not have any improvement, and thus belonged to Category A (error ratio $R_{\text{error}} \geq 1$). As d increased, the size of Category A decreased, while the size of Category B increased. Among the 4000 elements in the group with $d = 0.5$, the stiffness matrices of 2511 elements could be integrated more accurately by correcting their quadrature weights (Category B, $R_{\text{error}} < 1$) [38]. (Table reproduced with permission of the authors.)

neural network to perform the classification task took the 18 non-trivial nodal coordinates obtained upon the proposed normalization procedure for linear hexahedra as inputs, i.e., $\boldsymbol{x} = \boldsymbol{y}^{(0)} \in \mathbb{R}^{18 \times 1}$. The output of the neural network was single scalar $\widetilde{y} \in \mathbb{R}$, where $\widetilde{y} = 0$ indicated an element of Category A and $\widetilde{y} = 1$ an element of Category B.

Out of the 20,000 elements generated, 10,000 elements were selected to train the classifier network, for which both the training set and the validation set comprised 5000 elements each [38].[266] The optimal neural network in terms of classification accuracy for this application had 4 hidden layers with 30 neurons per layer; see Figure 102. The trained neural network succeeded in predicting the correct category for 98 % and 92 % of the elements in the training set and in the validation set, respectively.

In the second stage, a *second* neural network was trained to predict the corrections to the quadrature weights for all those elements, which allowed a reduction of the quadrature error. Again, the 18 non-trivial nodal coordinates of a normalized hexahedron were input to the neural network, i.e., $\boldsymbol{x} = \boldsymbol{y}^{(0)} \in \mathbb{R}^{18 \times 1}$. The outputs of the neural network $\widetilde{\boldsymbol{y}} \in \mathbb{R}^{8 \times 1}$ represented the eight correction factors to the standard weights $\{w_{i,j,k}^{opt}\}$, $i, j, k = 1, 2$, of the Gauss-Legendre quadrature. The authors of [38] stated that 10,000 elements with an error reduction ration $R_{\text{error}} < 1$ formed equally large training set and validation set, comprising 5000 elements each.[267] A neural network with 5 hidden layers of 50 neurons was reported to perform best in predicting the corrections to the quadrature weights. The normalized error[268] distribution of the correction factors are illustrated in Figure 103.

10.3.3 Method 2, application phase

The effectiveness of the numerical quadrature with corrected quadrature weights was already presented in Figure 10 in Section 2.3.1, with the distribution of the error-reduction ratio R_{error} defined in Eq. (402) for the training set (patterns) (a) and the validation set (test patterns) (b). Further explanation is provided here

---

[266] Even though a reason for not using the entire set of 20,000 elements was not given, it could be guessed that the authors of [38] would want the size of the training set and of the validation set to be the same as that in Method 1, Section 10.2. Moreover, even though details were not given, the selection of these 10,000 elements would likely be a random process.

[267] According to Figure 101, only 4868 out of in total 20,000 elements generated belonged to Category B, for which $R_{\text{error}} < 1$ held. Further details on the 10,000 elements that were being used for training and validation were not proviced in [38].

[268] No details on the normalization procedure were provided in the paper.

(a) Training patterns

		Category (estimated by neural network)		
		A	B	Total
Category (correct)	A	3707	29	3736
	B	70	1194	1264

(b) Test patterns

		Category (estimated by neural network)		
		A	B	Total
Category (correct)	A	3682	155	3837
	B	224	939	1163

Figure 102: *Method 2, training phase, classifier network* (Section 10.3.2). The training and validation sets comprised 5000 elements each, of which 3707 and 3682, respectively, belonged to Category A (no improvements upon weight correction). A *first* neural network with 4 hidden layers of 30 neurons correctly classified $(3707 + 1194)/5000 \approx 98\,\%$ elements in the training set (a) and $(3682 + 939)/5000 \approx 92\,\%$ elements in the validation set (b). See also Figure 10 in Section 2.3.1 [38]. (Table reproduced with permission of the authors.)

for Figure 10.[269]

The red bars ("Optimized") in Figure 10 represent the distribution of the error-reduction ratio R_{error} in Eq. (402), using the optimal correction factors, which were themselves obtained by a brute force method (Sections 10.3.1-10.3.2). While these optimal correction factors were used as targets for training (Figure 10 (a)), they were used only to compute the error-reduction ratios R_{error} for elements in the validation set (Figure 10 (b)).

The blue bars ("Estimated by Neuro") correspond to the error reduction ratios achieved with the corrected quadrature weights that were predicted by the trained neural network.

The error-reduction ratios $R_{\mathrm{error}} < 1$ indicate improved quadrature accuracy, which occurred for 99 % of the elements in the training set, and for 97 % of the elements in the validation set. Very few elements had their accuracy worsened ($R_{\mathrm{error}} > 1$) when using the predicted quadrature weights.

There were no red bars (optimal weights) with $R_{\mathrm{error}} > 1$ since only elements that admitted improved quadrature accuracy by optimal quadrature weight correction were included in the training set and validation set. The blue bars with error ratios $R_{\mathrm{error}} > 1$ corresponded to the distorted hexahedra for which the accuracy worsened as compared to the standard weights of Gauss-Legendre quadrature.

The authors of [38] concluded their paper with a discussion on the computational efforts, and in particular the evaluation of trained neural networks in the application phase. As opposed to computational mechanics, where we are used to double precision floating-point arithmetics, deep neural networks have proven to perform well with reduced numerical precision.[270] To speed up the evaluation of the trained networks, the least significant bits from all parameters (weights, biases) and the inputs were simply removed [38]. In both the estimation of the number of quadrature points and the prediction of the weight correction factors, half precision floating-point numbers (16 bit) turn out to show sufficient accuracy almost on par with single precision floats (32 bit).

---

[269] The caption of Figure 10, if it were in Section 10.3.3, would begin with *"Method 2, application phase"* in parallel to the caption of Figure 100.
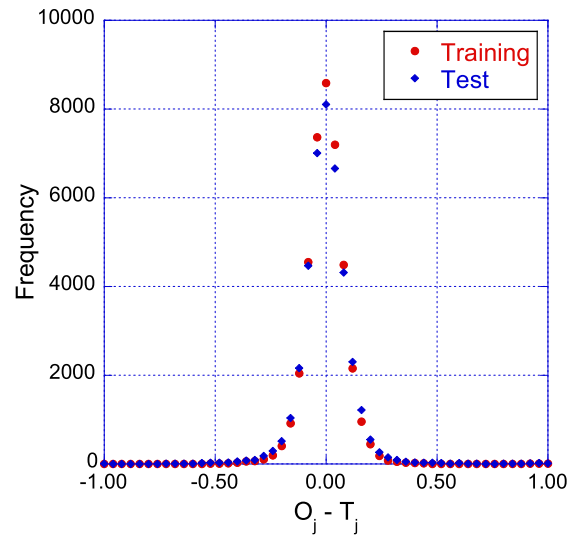
[270] See, e.g., [294] [295] [296].

Figure 103: *Method 2, training phase, regression network* (Section 10.3.2). A *second* neural network estimated 8 correction factors $\{w_{i,j,k}\}$, with $i, j, k \in \{1, 2\}$, to be multiplied by the standard quadrature weights for each element. Distribution of normalized errors, i.e., the normalized differences between the predicted weights (outputs) O_j and the true weights T_j for the elements of the training set (red) and the test set (blue). For both sets, which consist of $5000 \times 8 = 40,000$ correction factors each, the error has a mean of zero and seems to obey a normal distribution [38]. (Figure reproduced with permission of the authors.)

11 Application 2: Solid mechanics, multi-scale, multi-physics

The results and deep-learning concepts used in [25] were presented in Section 2.3.2 further above. In this section, we discuss some details of the formulation.

11.1 Multiscale problems

Multiscale problems are characterized by the fact that couplings of physical processes occurring on different scales of length and/or time needs to be considered. In the field of computational mechanics, multiscale models are often used to accurately capture the constitutive behavior on a macroscopic length scale, since resolving the entire domain under consideration on the smallest relevant scale if often intractable. To reduce the computational costs, multiscale techniques as, e.g., coupled DEM-FEM or coupled FEM-FEM (known as FEM^2),[271] have been proposed for bridging length scales by deducing constitutive laws from micro-scale models for Representative Volume Elements (RVEs).[272] These approaches no longer require the entire macroscopic domain to be resolved on the micro-scale. In FEM^2, for instance, the microscale model to deduce the constitutive behavior on the macroscopic scale is evaluated at the quadrature points of the microscale model.

The multiscale problem in the mechanics of porous media tackled in [25] is represented in Figure 104, where the relative orientations among the three models at microscale, mesoscale, and macroscale in Figure 14 are indicated. The method of analysis (DEM or FEM) in each scale is also indicated in the figure.

11.2 Data-driven constitutive modeling, deep learning

Despite the diverse approaches proposed, multiscale problems remain a computationally challenging task, which was tackled in [25] by means of a hybrid data-driven method by combining deep neural net-

---

[271] DEM = Discrete Element Method. FEM = Finite Element Method. See references in [25].
[272] RVEs are also referred to as *representative elementary volumes* (REVs) or, simply, *unit cells*.
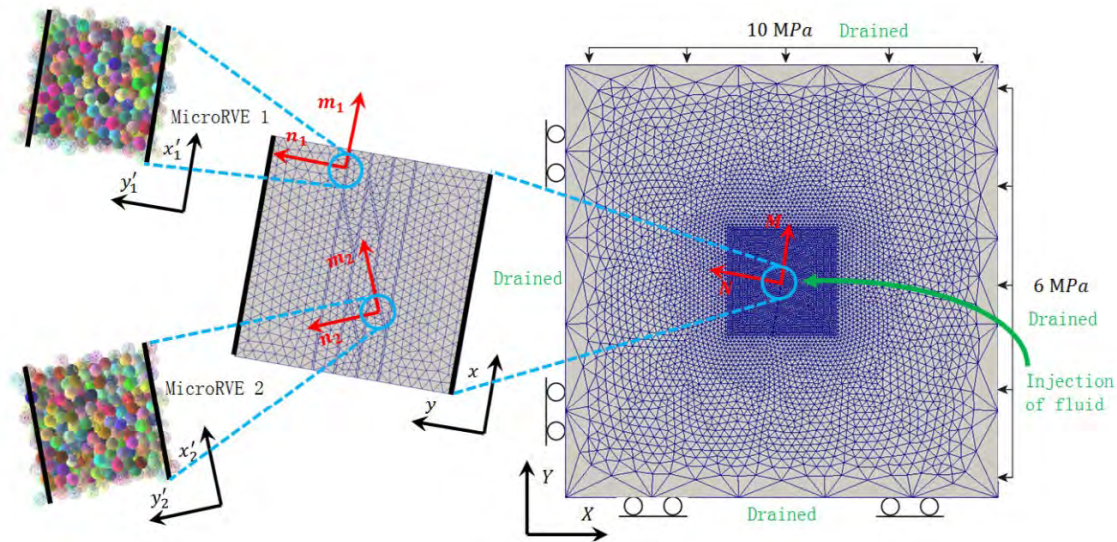
Figure 104: *Three scales in data-driven fault-reactivation simulations* (Sections 2.3.2, 11.1, 11.3.5). Relative orientation of Representative Volume Elements (RVEs). *Left:* Microscale (μ) RVE using Discrete Element Method (DEM), Figure 106 and Row 1 of Figure 14. *Center:* Mesoscale (cm) REV using FEM; Row 2 of Figure 14. *Right:* Field-size macroscale (km) FEM model; Row 3 of Figure 14 [25]. (Figure reproduced with permission of the authors.)

Figure 105: *Single-physics block diagram* (Section 11.2). Single physics is an easiest way to see the role of deep learning in modeling complex nonlinear constitutive behavior (stress-strain relation, red arrow), as first realized in [23], where balance of linear momentum and strain-displacement relation are definitions or accepted "universal principles" (black arrows) [25]. (Figure reproduced with permission of the authors.)

works and conventional constitutive models. To illustrate the hierarchy among the relations of models and to identify, the authors of [25] used directed graphs, which also indicated the nature of the individual relations by the colors of the graph edges. Black edges correspond to "universal principles" whereas red edges represent phenomenological relations, see, e.g., the classical problem in solid mechanics shown in Figure 105. Within classical mechanics, the balance of linear momentum is axiomatic in nature, i.e., it represents a well-accepted premise that is taken to be true. The relation between the displacement field and the strain tensor represents a definition. The constitutive law describing the stress response, which, in the elastic case, is an algebraic relation among stresses and strains, is the only phenomenological part in the "single-physics" solid mechanics problem and, therefore, highlighted in red.

In many engineering problems, stress-strain relations of, possibly nonlinear, elasticity, which are parameterized by a set of elastic moduli, can be used in the modeling. For heterogeneous materials as, e.g., in composite structures, even the "single physics" problem of elastic solid mechanics may necessitate multiscale approaches, in which constitutive laws are replaced by RVE simulations and homogenization. This approach was extended to multiphysics models of porous media, in which multiple scales needed to be considered [25]. The counterpart of Figure 105 for the mechanics of porous media is complex, could be confusing for readers not familiar with the field, does not add much to the understanding of the use of deep learning in this study, and therefore not included here; see [25].

The hybrid approach in [25], which was described as graph-based machine learning model, retained those parts of the model which represented universal principles or definitions (black arrows). Phenomenological relations (red arrows), which, in conventional multiscale approaches, followed from microscale models, were replaced by computationally efficient data-driven models. In view of the path-dependency of the constitutive behavior in the poromechanics problem considered, it was proposed in [25] to use recurrent neural networks (RNNs), Section 7.1, constructed with Long Short-Term Memory (LSTM) cells, Section 7.2.

11.3 Multiscale multiphysics problem: Porous media

The problem of hydro-mechanical coupling in deformable porous media with multiple permeabilities is characterized by the presence of two or more pore systems with different typical sizes or geometrical features of the host matrix [25]. The individual pore systems may exchange fluid depending on whether the pores are connected or not. If the (macroscopic) deformation of the solid skeleton is large, plastic deformation and cracks may occur, which result in anisotropic evolution of the effective permeability. As a consequence, problems of this kind are not characterized by a single effective permeability, and, to identify the material parameters on the macroscopic scale, micro-structural models need to be incorporated.

The authors of [25] considered a saturated porous medium, which features two dominant pore scales: The regular solid matrix was characterized by micropores, whereas macropores may result, e.g., from cracks and fissures. Both the volume and the partial densities of each constituent in the mixture of solid, micropores, macropores and voids were characterized by the porosity, i.e., the (local) ratio of pores and the total volume, as well as the fractions of the respective pore systems.

11.3.1 Recurrent neural networks for scale bridging

Recurrent neural networks (RNNs, Section 7.1), which are equivalent to "very deep feedforward networks" (Remark 7.2), were used in [25] as a scale-bridging method to efficiently simulate multiscale problems of poroplasticity. In Figure 14, Section 2.3.2, three scales were considered: Microscale (μ), mesoscale (cm), macroscale (km).

At the microscale, Discrete Element Method (DEM) is used to simulate a mesoscale Representative Volume Element (RVE) that consists of a cubic pack of microscale particles, with different loading conditions to generate a training set for a mesoscale RNN with LSTM architecture to model mesoscale constitutive response to produce loading histories ($\epsilon_{meso}, \sigma'_{meso}$) on demand.

At the mesocale, Finite Element Method (FEM), combined with mesoscale loading histories (ϵ_{meso}, σ'_{meso}) produced by the above mesoscale RNN, are used to model a macroscale RVE, subjected to different loading conditions to generate a training set for a macroscale RNN with LSTM architecture to model macroscale constitutive response to produce loading histories ($\epsilon_{macro}, \sigma'_{macro}$) on demand.

At the macroscale, Finite Element Method (FEM), combined with macroscale loading histories (ϵ_{macro}, σ'_{macro}) produced by the above macroscale RNN, are used to model an actual domain at kilometer size (macroscale).

11.3.2 Microstructure and principal direction data

To train the mesoscale RNN with LSTM units (which was called the "Mesoscale data-driven constitutive model" in [25]), incorporating microstucture data—such as the fabric tensor $\boldsymbol{F}$ of the 1st kind of rank 2 (motivated in Section 2.3.2, Figure 16 and Figure 17) defined in Eq. (404) [102]—into the training set, whose data were generated by a discrete element assembly as in Figure 106 depicting a microscale RVE, was important to obtain good network prediction, as mentioned in Section 2.3.2 in relation to Figure 17:

$$\boldsymbol{F} := \boldsymbol{A}_F \cdot \boldsymbol{A}_F := \frac{1}{N_c} \sum_{c=1}^{c=N_c} \boldsymbol{n}_c \otimes \boldsymbol{n}_c \,, \tag{404}$$
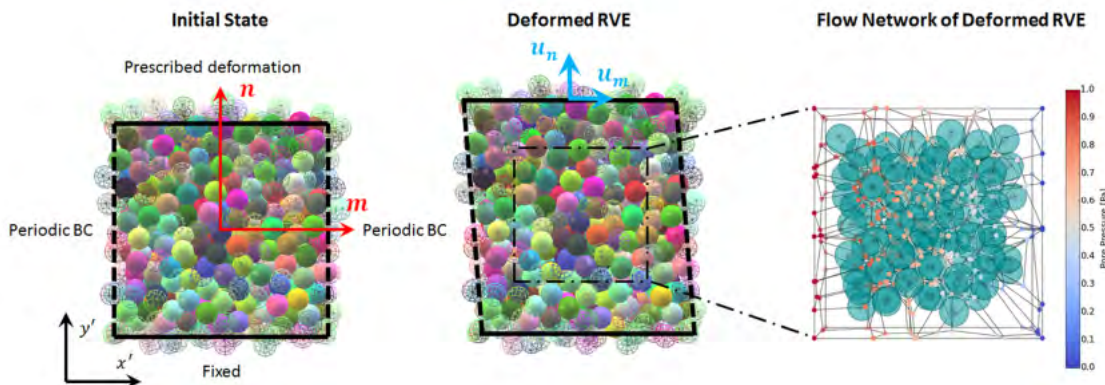
Figure 106: *Microscale RVE* (Sections 11.3.2, 11.3.3, 11.3.5). A 10 cm × 10 cm × 5 cm box of identical spheres of 0.5 cm diameter (Figure 14, row 1, Remark 11.1), subjected to imposed displacements. (a) Initial configuration of the granular assembly; (b) Imposed displacement, deformed configuration; (c) Flow network generated from deformed configuration used to predict anisotropic effective permeability [25]. See also Figure 104 and Figure 110. (Figure reproduced with permission of the authors.)

where N_c is the number of contact points (which is the same as the coordination number CN), and $\boldsymbol{n}_c$ the unit normal vector at contact point c.

Remark 11.1. Even though in Figure 14 (row 1) in Section 2.3.2, the microscale RVE was indicated to be of micron size, but the microscale RVE in Figure 106 was of size 10 cm × 10 cm × 5 cm with particles of 0.5 cm in diameter, many orders of magnitude larger. See Remark 11.9. ∎

Other microstructure data such as the porosity and the coordination number (or number of contact point), being scalars and did not incorporate directional data like the fabric tensor, did not help to improve the accuracy of the network prediction, as noted in the caption of Figure 17.

To enforce objectivity of constitutive models realized as neural networks, (the history of) principal strains and incremental rotation parameters that describe the orientation of principal directions served as inputs to the network. Accordingly, principal stresses and incremental rotations for the principal directions were outputs of what was referred to as *Spectral RNNs* [25], which preserved objectivity of constitutive models.

Different ANN architectures for evaluation of training performance.

ANN Configuration	Number of hidden layers	Number of LSTM neurons per hidden layer	Activation function for output layer
1	2	50	Sigmoid
2	2	80	Sigmoid
3	2	100	Sigmoid
4	3	80	Sigmoid
5	2	80	ReLU

Figure 107: *Optimal RNN-LSTM architecture* (Section 11.3.3). 5 different configurations of RNNs with LSTM units [25]. (Table reproduced with permission of the authors.)

11.3.3 Optimal RNN-LSTM architecture

Using the same discrete element assembly of microscale RVE in Figure 106 to generate data, the authors of [25] tried out 5 different configurations of RNNs with LSTM units, with 2 or 3 hidden layers, 50 to 100
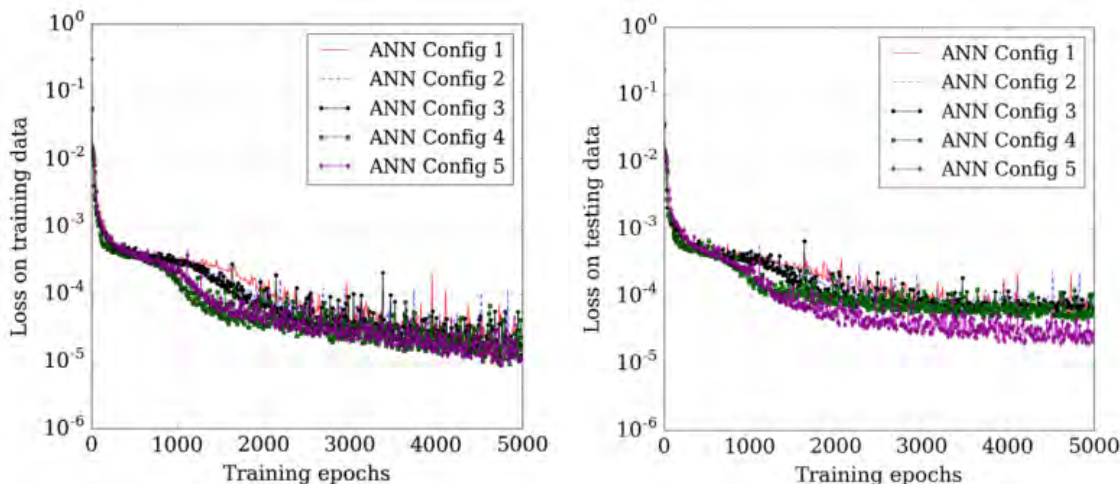
Figure 108: *Optimal RNN-LSTM architecture* (Section 11.3.3). Training error and test errors for 5 different configurations of RNN with LSTM units, see Figure 107 [25]. (Figure reproduced with permission of the authors.)

LSTM units (Figure 15, Section 2.3.2, and Figure 81 for the detailed original LSTM cell), and either logistic sigmoid or ReLU as activation function, Figure 107.

Configuration 1 has 2 hidden layers with 50 LSTM units each, and with the logistic sigmoid as activation function. Config 2 is similar, but with 80 LSTM units per hidden layer. Config 3 is similar, but with 100 LSTM units per hidden layer. Config 4 is similar to Config 2, but with 3 hidden layers. Config 5 is similar to Config 4, but with ReLU activation function.

The training error and test error obtained from using these 5 configurations are shown in Figure 108. The zoomed-in views of the training error and test error from epoch 3000 to epoch 5000 in Figure 109 show that Config 5 was the optimal with smaller errors, and since ReLU would be computationally more efficient than the logistic sigmoid. But Config 2 was, however, selected in [25], whose authors cited that the discrepancy was "not significant", and that Config 2 gave "good training and prediction performances".

Remark 11.2. The above search for an optimal network architecture is similar to searching for an appropriate degree of a polynomial function for a best fit, avoiding overfit and underfit, over a given set of data points in a least-square curve fittings. See Figure 72 in Section 6.5.9 for an explanation of underfit and overfit, and Figure 99 in Section 10 for a similar search of an optimal network for numerical integration by ANN. ∎

Remark 11.3. Referring to Remark 4.3 and the neural network in Figure 14 and to our definition of action depth as total number of action layers L in Figure 23 in Section 4.3.1 and Remark 4.5 in Section 4.6, it is clear that a network layer in [25] is a state layer, i.e., an input matrix $\boldsymbol{y}^{(\ell)}$, with $\ell = 0, \ldots, L$. Thus all configs in Figure 107 with 2 hidden layers have an action depth of $L = 3$ layers and a state depth of $L + 1 = 4$ layers, whereas Config 4 with 3 hidden layers has an action depth of $L = 4$ layers and a state depth of $L + 1 = 5$ layers. On the other hand, since RNNs with LSTM units were used, in view of Remark 7.2, these networks were equivalent to "very deep feedforward networks". ∎

Remark 11.4. The same *selected* architecture of RNN with LSTM units on both the microscale RVE (Figure 106, Figure 110) and the mesoscale RVE (Figure 112, Figure 113) was used to produce the mesoscale RNN with LSTM units ("Mesoscale data-driven constitutive model") and the macroscale RNN with LSTM units ("Macroscale data-driven constitutive model"), respectively [25]. ∎
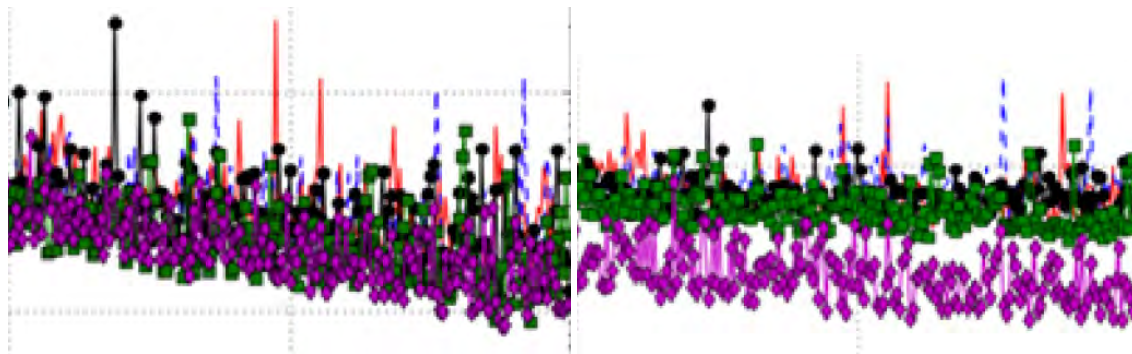
Figure 109: *Optimal RNN-LSTM architecture* (Section 11.3.3). Training error (a) and testing error (b), close-up views of Figure 108 from epoch 3000 to epoch 5000: Config 5 (purple line with dots) was optimal, with smaller errors than those of Config 2 (blue dashed line). See Figure 107 for config details [25]. (Figure reproduced with permission of the authors.)

11.3.4 Dual-porosity dual-permeability governing equations

The governing equations for media with dual-porosity dual-permeability in this section are only applied to macroscale (field-size) simulations, i.e., not for simulations with the microscale RVE (Figure 106, Figure 110) and with the mesoscale RVE (Figure 112, Figure 113).

For field-size simulations, assuming stationary conditions, small deformations, incompressibility, no mass exchange among solid and fluid constituents, the problem is governed by the balance of linear momentum and the balance of fluid mass in micropores and macropores, respectively. The displacement field of the solid $\boldsymbol{u}$, the micropore pressure p_m and the macropore pressure p_M constitute the primary unknowns of the problem. The (total) Cauchy stress tensor $\boldsymbol{\sigma}$ is the sum of the effective stress tensor $\boldsymbol{\sigma}'$ on the solid skeleton and the pore fluid pressure p_f, which was in between p_M and p_m, and assumed to be a convex combination of the latter two pore pressures [25],

$$\boldsymbol{\sigma} = \boldsymbol{\sigma}' - p_f \boldsymbol{I} \Rightarrow \boldsymbol{\sigma}' = \boldsymbol{\sigma} + \{\psi \, p_M + (1 - \psi) \, p_m\} \, \boldsymbol{I}, \tag{405}$$

where ψ denoted the ratio of macropore volume over total pore volume, and $(1 - \psi)$ the ratio of micropore volume over total pore volume.

The *balance of linear momemtum* equation is written as

$$\operatorname{div} \boldsymbol{\sigma} + \rho \boldsymbol{g} = c_0 (\widetilde{\boldsymbol{v}}_m - \widetilde{\boldsymbol{v}}_M) \,, \tag{406}$$

where ρ is the total mass density, $\boldsymbol{g}$ the acceleration of gravity,

$$\widetilde{\boldsymbol{v}}_M := \boldsymbol{v}_M - \boldsymbol{v} \,, \quad \widetilde{\boldsymbol{v}}_m := \boldsymbol{v}_m - \boldsymbol{v} \,, \tag{407}$$

where $\boldsymbol{v}$ is the velocity of the solid skeleton, $\boldsymbol{v}_M$ and $\boldsymbol{v}_m$ are the fluid velocities in the macropores and in the micropores, respectively. In Eq. (406), the coefficient c_0 of fluid mass transfer between macropores and micropores was assumed to obey the "semi-empirical" relation [25]:

$$c_0 = \frac{\bar{\alpha}}{\mu_f} \, (p_M - p_m) \,, \tag{408}$$

with $\bar{\alpha}$ being a parameter that characterized the interface permeability between macropores and micropores, and μ_f the dynamic viscosity of the fluid.

In the 1-D case, Darcy's law is written as

$$\frac{q(x,t)}{\rho_f} = v(x,t) = -\frac{k}{\mu_f} \frac{\partial p(x,t)}{\partial x} \,, \tag{409}$$

where q is the fluid mass flux (kg/(m^{2}s)), ρ_f the fluid mass density (kg / m^3), v the fluid velocity (m/s), k the medium permeability (m^2), μ_f the fluid dynamic viscosity (Pa $\cdot$ s = N s/m^2), p the pressure (N/m^2), and x the distance (m).
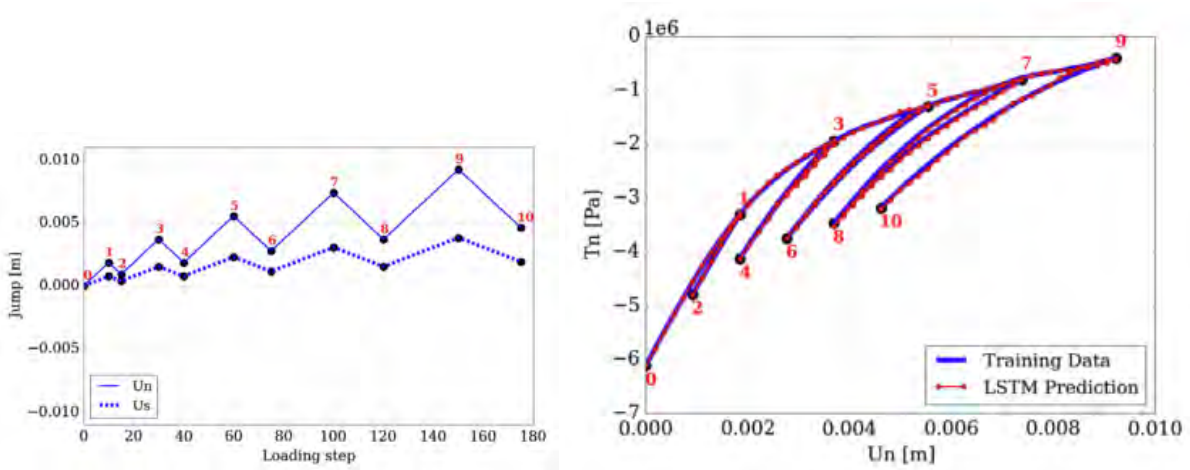

Figure 110: *Mesoscale RNN with LSTM units. Traction-separation law* (Sections 11.3.3, 11.3.5). *Left:* Sequence of imposed displacement jumps on microscale RVE (Figure 106), normal (solid line) and tangential (dotted line, with $u_s \equiv u_m$ in Figure 106, center). *Right:* Normal traction vs. normal displacement. Cyclic loading and unloading. Microscale RVE training data (blue) vs. Mesoscale RNN with LSTM prediction (red, Section 11.3.3), with mean squared error 3.73×10^{-5}. See also Figure 115 on the macroscale RNN with LSTM units [25]. (Figure reproduced with permission of the authors.)

Remark 11.5. *Dimension of $\bar{\alpha}$ in Eq.* (408). Each term in the balance of linear momentum Eq. (406) has force per unit volume (F/L^3) as dimension, which is therefore the dimension of the right-hand side of Eq. (406), where c_0 appears. As a result, c_0 has the dimension of mass (M) per unit volume (L^3) per unit time (T):

$$[c_0(\widetilde{\boldsymbol{v}}_m - \widetilde{\boldsymbol{v}}_M)] = \frac{F}{L^3} = \frac{M}{L^2 T^2} \Rightarrow [c_0] = \frac{M}{L^2 T^2}\frac{T}{L} = \frac{M}{L^3 T} \ . \tag{410}$$

Another way to verify is to identify the right-hand side of Eq. (406) with the usual inertia force per unit volume:

$$[c_0(\widetilde{\boldsymbol{v}}_m - \widetilde{\boldsymbol{v}}_M)] = \left[\rho\frac{\partial \boldsymbol{v}}{\partial t}\right] \Rightarrow [c_0]\,[\boldsymbol{v}] = [\rho]\frac{[\boldsymbol{v}]}{T} \Rightarrow [c_0] = \frac{[\rho]}{T} = \frac{M}{L^3 T} \ . \tag{411}$$

The empirical relation Eq. (408) adopted in [25] implies that the dimension of $\bar{\alpha}$ was

$$[\bar{\alpha}] = \frac{[c_0]\,[\mu_f]}{[(p_M - p_m)]} = \frac{\left(M/(L^3 T)\right)\left(FT/L^2\right)}{F/L^2} = \frac{M}{L^3} \ , \tag{412}$$

i.e., mass density, whereas permeability has the dimension of area (L^2), as seen in Darcy's law Eq. (409). It is not clear why it is written in [25] that $\bar{\alpha}$ characterized the "interface permeability between the macropores and the micropores". A reason could be that the "semi-empirical" relation Eq. (408) was assumed to be analogous to Darcy's law Eq. (409). Moreover, as a result of Eq. (412), the dimension of $\mu_f/\bar{\alpha}$ is therefore the same as that of the kinematic viscosity $\nu_f = \mu_f/\rho$. If $\bar{\alpha}$ had the dimension of permeability, then the choice of the right-hand side of the balance of linear momentum Eq. (406) was inconsistent, dimensionally speaking. See Remark 11.6. ∎
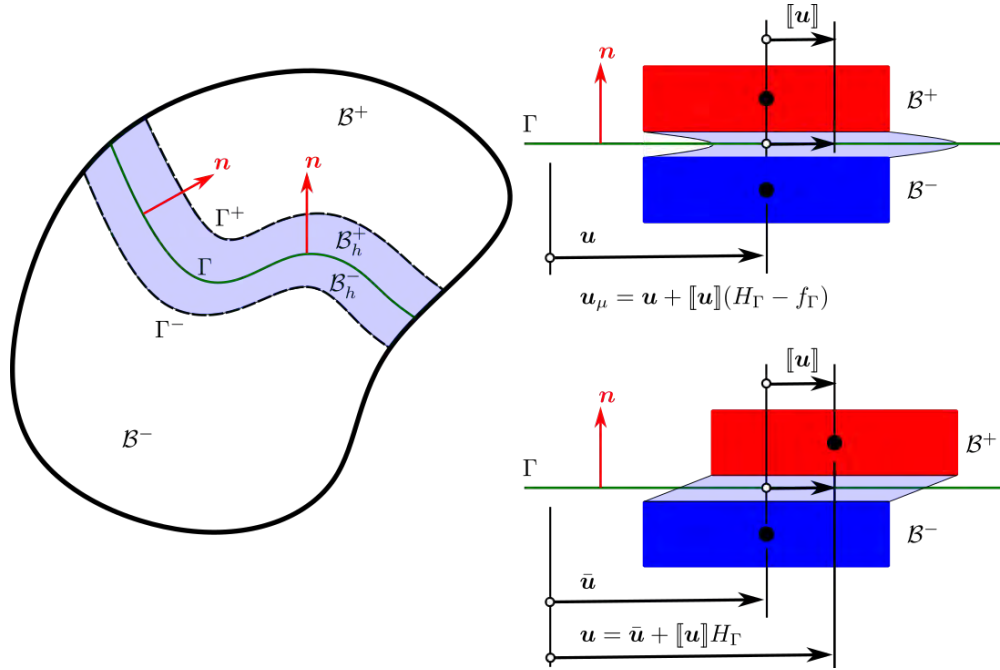
Figure 111: *Continuum with embedded strong discontinuity* (Section 11.3.5). Domain $\mathcal{B} = \mathcal{B}^+ \cup \mathcal{B}^-$ with embedded discontinuity surface Γ, running through the middle of a narrow band (light blue) $\mathcal{B}_h = (\mathcal{B}_h^+ \cup \mathcal{B}_h^-) \subset \mathcal{B}$ between the parallel surfaces Γ^+ and Γ^-. Objects behind Γ in the negative direction of the normal $\boldsymbol{n}$ to Γ are designated with the minus sign, and those in front of Γ with the plus sign. The narrow band $\mathcal{B}_h$ represents an embedded strong discontinuity, as shown in the mesoscale RVE in Figure 113, where the discretized strong discontinuity zones were a network of straight narrow bands. *Top right:* No sliding, tnterpretation of $\boldsymbol{u}_\mu = \boldsymbol{u} + [\![\boldsymbol{u}]\!](H_\Gamma - f_\Gamma)$ in [25]. *Bottom right:* Sliding, interpretation of $\boldsymbol{u} = \bar{\boldsymbol{u}} + [\![\boldsymbol{u}]\!]H_\Gamma$ in [297].

The (local) porosity ϕ is the ratio of the void volume dV_v over the total volume dV. Within the void volume, let ψ be the percentage of macropores, and $(1 - \psi)$ the percentage of micropores; we have:

$$\phi = \frac{dV_v}{dV} , \quad \psi = \frac{dV_M}{dV_v} , \quad 1 - \psi = \frac{dV_m}{dV_v} . \tag{413}$$

The absolute macropore flux $\boldsymbol{q}_M$ and the absolute micropore flux $\boldsymbol{q}_m$ are defined as follows:

$$e_M = \rho_f \phi \psi , \quad \boldsymbol{q}_M = e_M \boldsymbol{v}_M , \quad e_m = \rho_f \phi (1 - \psi) , \quad \boldsymbol{q}_m = e_m \boldsymbol{v}_m , \tag{414}$$

There is a conservation of fluid transfer between the macropores and the micropores across any closed surface Γ, i.e.,

$$\int_\Gamma (\boldsymbol{q}_M + \boldsymbol{q}_m) \bullet \boldsymbol{n} \, d\Gamma = 0 \Rightarrow \operatorname{div}(\boldsymbol{q}_M + \boldsymbol{q}_m) = 0 . \tag{415}$$

Generalizing Eq. (409) to 3-D, Darcy's law in tensor form governs the fluid mass fluxes $\boldsymbol{q}_M$, $\boldsymbol{q}_m$ is written as

$$\widetilde{\boldsymbol{q}}_M = e_M \widetilde{\boldsymbol{v}}_M = -\rho_f \frac{\boldsymbol{k}_M}{\mu_f} \cdot (\nabla p_M - \rho_f \boldsymbol{g}) , \quad \widetilde{\boldsymbol{q}}_m = e_M \widetilde{\boldsymbol{v}}_m = -\rho_f \frac{\boldsymbol{k}_m}{\mu_f} \cdot (\nabla p_m - \rho_f \boldsymbol{g}) , \tag{416}$$

where $\boldsymbol{k}_M$, $\boldsymbol{k}_m$ denote the permeability tensors on the respective scales and $\boldsymbol{g}$ is the gravitational acceleration.

From Eq. (415), assuming that

$$\operatorname{div} \boldsymbol{q}_M = -\operatorname{div} \boldsymbol{q}_m = c_0 , \tag{417}$$

where c_0 is given in Eq. (408), then two more governing equations are obtained:

$$\text{div } \boldsymbol{q}_M = e_M \text{div } \boldsymbol{v} + \text{div } \widetilde{\boldsymbol{q}}_M = c_0, \tag{418}$$

$$\text{div } \boldsymbol{q}_m = e_m \text{div } \boldsymbol{v} + \text{div } \widetilde{\boldsymbol{q}}_m = -c_0, \tag{419}$$

agreeing with [25].

Remark 11.6. It can be verified from Eq. (417) that the dimension of c_0 is

$$[c_0] = [\text{div } \boldsymbol{q}_M] = \frac{[\boldsymbol{q}_M]}{L} = \frac{[\rho]\,[\boldsymbol{v}]}{L} = \frac{M}{L^3 T}\,, \tag{420}$$

agreeing with Eq. (411). In view of Remark 11.5, for all three governing Eq. (406), Eqs. (418)-(419) to be dimensionally consistent, $\bar{\alpha}$ in Eq. (408) should have the same dimension as mass density, as indicated in Eq. (412). Our consistent notation for fluxes $(\boldsymbol{q}_M, \boldsymbol{q}_m)$ and $(\widetilde{\boldsymbol{q}}_M, \widetilde{\boldsymbol{q}}_m)$ differ in meaning compared to [25].   ∎
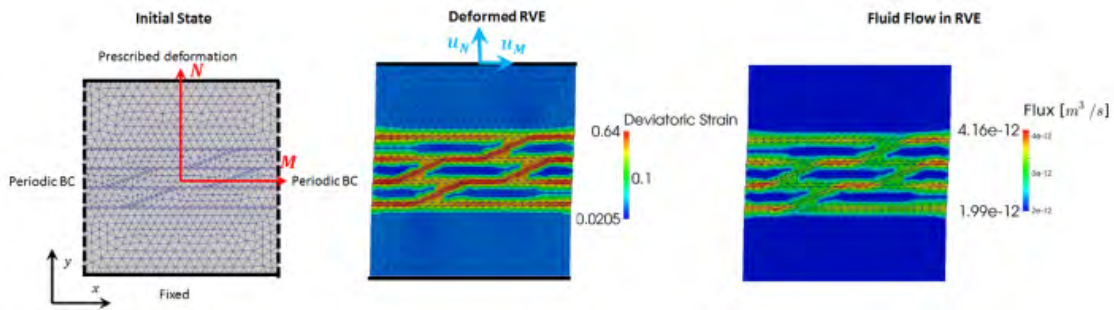
Figure 112: *Mesoscale RVE* (Sections 11.3.3, 11.3.5). A 2-D domain of size 1 m × 1 m (Remark 11.9). See Figure 14 (row 2, left), and also Figure 111 for a conceptual representation. Embedded strong discontinuity zones where damage occurred formed a network of straight narrow bands surrounded by elastic material. Both the strong discontinuity narrow bands and the elastic domain were discretized into finite elements. Imposed displacements $(\boldsymbol{u}_N, \boldsymbol{u}_M)$, with $\boldsymbol{u}_S \equiv \boldsymbol{u}_N$, at the top (center). See Figure 113 for the deformation (strains and displacement jumps) [25]. (Figure reproduced with permission of the authors.)

Remark 11.7. For field-size simulations, the above equations do not include the changing size of the pores, which were assumed to be of constant size, and thus constant porosity, in [25]. As a result, the collapse of the pores that leads to nonlinearity in the stress-strain relation observed in experiments (Figure 13) is not modelled in [25], where the nonlinearity essentially came from the embedded strong discontinuities (displacement jumps) and the associated traction-separation law obtained from DEM simulations using the micro RVE in Figure 106 to train the meso RNN with LSTM; see Section 11.3.5. See also Remark 11.10. ∎

11.3.5 Embedded strong discontinuities, traction-separation law

Strong discontinuities are embedded at both the mesoscale and the macroscale. Once a fault is formed through cracks in rocks, it could become inactive (no further slip) due to surrounding stresses, friction between the two surfaces of a fault, cohesive bond, and low fluid pore pressure. A fault can be reactivated (onset of renewed fault slip) due to changing stress state, loosened fault cohesion, high fluid pore-pressure. Conventional models for fault reactivation are based on effective stresses and Coulomb law [298]:

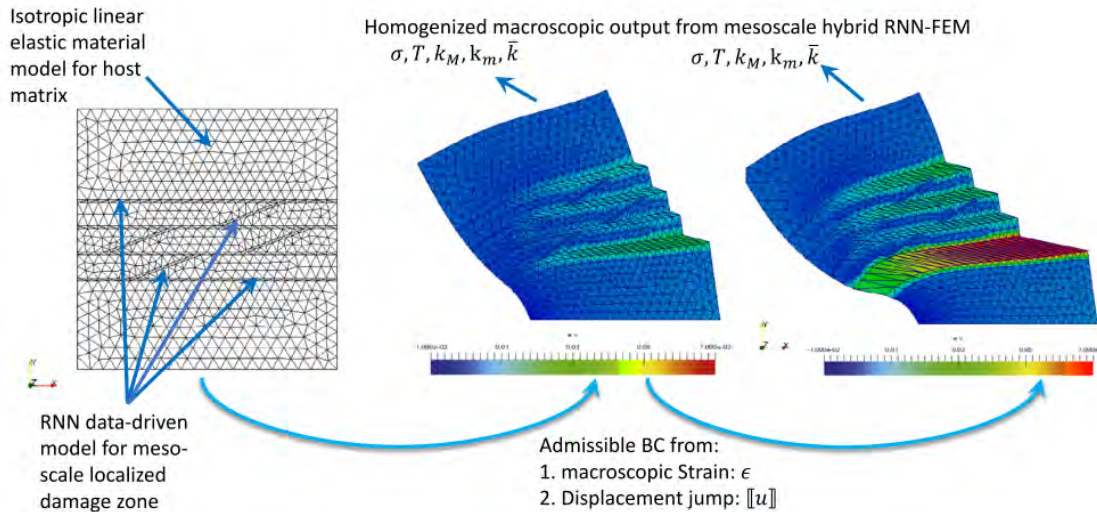$$\tau \geq \tau_p = C + \mu\sigma' = C + \mu(\sigma - p)\,, \tag{421}$$

Figure 113: *Mesoscale RVE* (Section 11.3.3). Strains and displacement jumps [25]. (Figure reproduced with permission of the authors.)

where τ is the shear stress along the fault line, τ_p the critical shear stress for the onset of fault reactivation, C the cohesion strength, μ the coefficient of friction, σ' the effective stress normal to the fault line, σ the normal stress, and p the fluid pore pressure. The authors of [299] demonstrated that increase in fluid injection rate led to increase in peak fluid pressure and, as a result, fault reactivation, as part of a study on why there was an exponential increase in seismic activities in Oklahoma, due to wastewater injection for use in hydraulic fracturing (i.e., fracking) [300].

But criterion Eq. (421) involves only stresses, with no displacement, and thus cannot be used to quantify the amount of fault slip. To allow for quantitative modeling of fault slip, or displacement jump, in a displacement-driven FEM environment, the so-called "cohesive traction-separation laws," expressing traction (stress) vector on fault surface as a function of fault slip, similar to those used in modeling cohesive zone in nonlinear fracture mechanics [301], is needed. But these classical "cohesive traction-separation law" are not appropriate for handling loading-unloading cycles.

To model a continuum with displacement jumps, i.e., embedded strong discontinuities, the traction-separation law was represented in [25] as

$$\boldsymbol{T}(\llbracket \boldsymbol{u} \rrbracket) = \boldsymbol{\sigma}'(\llbracket \boldsymbol{u} \rrbracket) \boldsymbol{\cdot} \boldsymbol{n} \ , \tag{422}$$

where $\boldsymbol{T}$ is the traction vector on the fault surface, $\boldsymbol{u}$ the displacement field, $\llbracket \cdot \rrbracket$ the jump operator, making $\llbracket \boldsymbol{u} \rrbracket$ the displacement jump (fault slip, separation), $\boldsymbol{\sigma}'$ the effective stress tensor at the fault surface, $\boldsymbol{n}$ the normal to the fault surface. To obtain the traction-separation law represented by the function $\boldsymbol{T}(\llbracket \boldsymbol{u} \rrbracket)$, a neural network can be used provided that data are available for training and testing.

It was assumed in [25] that cracks were pre-existing and did not propagate (see the mesoscale RVE in Figure 113), then set out to use the microscale RVE in Figure 106 to generate training data and test data for the mesocale RNN with LSTM units, called the "Mesoscale data-driven constitutive model", to represent the traction-separation law for porous media. Their results are shown in Figure 110.

Remark 11.8. The microscale RVE in Figure 106 did not represent any real-world porous rock sample such the Majella limestone with macroporosity $11.4\% \approx 0.1$ and microporosity $19.6\% \approx 0.2$ shown in Figure 12 in Section 2.3.2, but was a rather simple assembly of mono-disperse (identical) solid spheres with no information on size and no clear contact force-displacement relation; see [302]. Another problem was that realistic porosity of 0.1 or 0.2 could not be achieved with this microscale RVE, yielding a porosity above

0.3, which was the total porosity (= macroporosity + microporosity) of the highly porous Majella limestone. A goal of [25] was only to demonstrate the methodology, not presenting realistic results. ∎
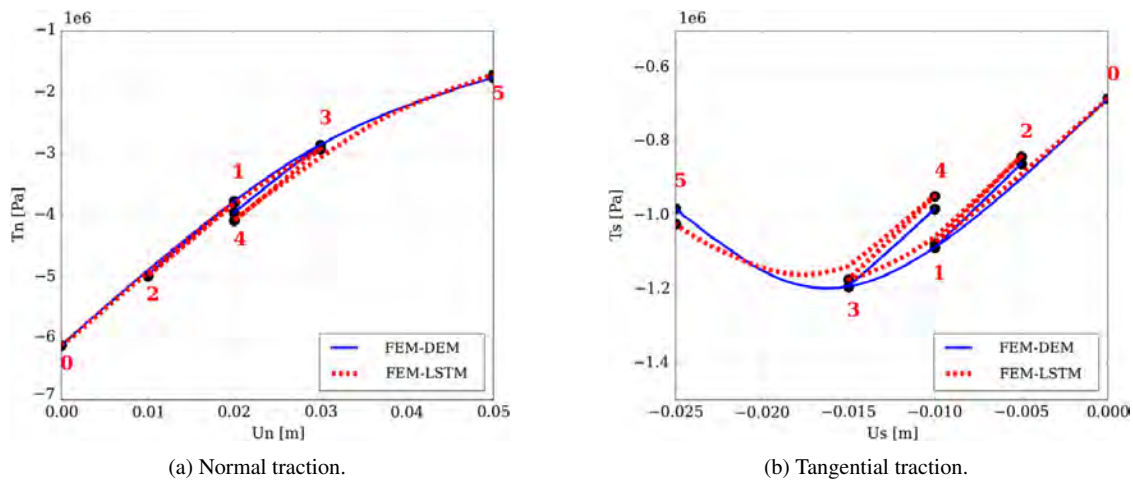

(a) Normal traction.   (b) Tangential traction.

Figure 114: *Mesoscale RVE* (Section 11.3.5). Validation of coupled FEM and RNN with LSTM units (FEM-LSTM, red dotted line) against coupled FEM and DEM (FEM-DEM, blue line) to analyze the mesoscale RVE in Figure 113 under a sequence of imposed displacement jumps at the top (represented by numbers). (a) Normal traction (Tn) vs normal displacement (Un). (b) Tangential traction (Ts) vs tangential displacement (Us) [25]. (Figure reproduced with permission of the authors.)

Remark 11.9. Even though in Figure 14 (row 2) in Section 2.3.2, the mesoscale RVE was indicated to be of centimeter size, but the mesoscale RVE in Figure 113 was of size 1 m × 1 m, many orders of magnitude larger. See Remark 11.1. ∎

To analyze the mesoscale RVE in Figure 113 (Figure 104, center) and the macroscale (field-size) model (Figure 104, right) by finite elements, both with embedded strong discontinuities (Figure 111), the authors of [25] adopted a formulation that looked similar to [297] to represent strong discontinuities, which could result from fractures or shear bands, by the *local* displacement field u_μ as[273]

$$u_\mu = u + [\![u]\!] (H_\Gamma - f_\Gamma) , \quad \bar{u} := u - [\![u]\!] f_\Gamma \Rightarrow u_\mu = \bar{u} + [\![u]\!] H_\Gamma , \tag{423}$$

which differs from the global smooth displacement field u by the displacement jump vector $[\![u]\!]$ across the singular surface Γ that represents a discontinuity, multiplied by the function $(H_\Gamma - f_\Gamma)$, where H_Γ is the Heaviside function, such that $H_\Gamma = 0$ in $\mathcal{B}^-$ and $H_\Gamma = 1$ in $\mathcal{B}^+$, and f_Γ a smooth ramp function equal to zero in $(\mathcal{B}^- - \mathcal{B}_h^-)$ and going smoothly up to 1 in $(\mathcal{B}^+ - \mathcal{B}_h^+)$, as defined in [297]. So Eq. (423) means that the displacement field u only had a smooth "bump" with support being the band $\mathcal{B}_h$, as a result of a local displacement jump, with *no* fault sliding, as shown in the top right subfigure in Figure 111.

Eq. (423)₃ was the starting point in [297], but *without* using the definition of $\bar{u}$ in Eq. (423)₂:[274]

$$u = \bar{u} + [\![u]\!] H_\Gamma \Rightarrow \dot{u} = \dot{\bar{u}} + [\![\dot{u}]\!] H_\Gamma , \tag{424}$$

with u being the total displacement field (including the jump), $\bar{u}$ the smooth part of u, and the overhead dot representing the rate or increment. As such, Eq. (424) can describe the sliding between $\mathcal{B}_-$ and $\mathcal{B}_+$, as

---

[273] The subscript μ in u_μ probably meant "micro", and was used to designate the *local* nature of u_μ.
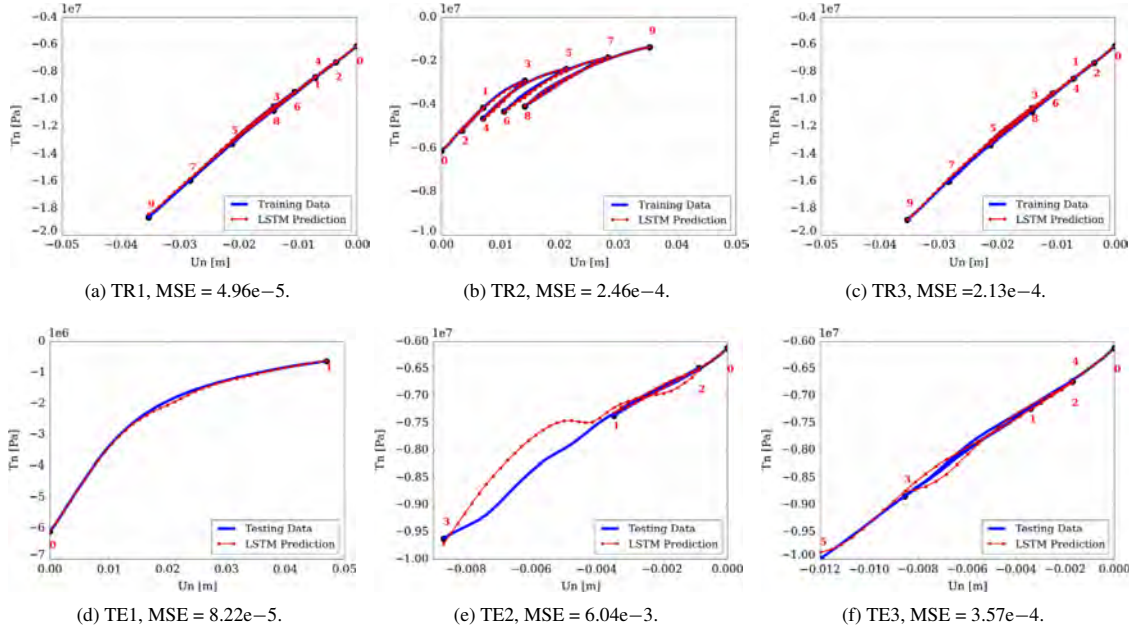[274] [297], Eq. (2.1).

Figure 115: *Macroscale RNN with LSTM units* (Section 11.3.5). Normal traction (Tn) vs imposed displacement jumps (Un) on mesoscale RVE (Figure 112, Figure 113). *Blue:* Training data (TR1-TR3) and test data (TE1-TE3) from the mesoscale FEM-LSTM model, where numbers indicate the sequence of loading-unloading steps similar to those in Figure 110 for the mesoscale RNN with LSTM units. *Red:* Corresponding predictions of the trained macroscale RNN with LSTM units. The mean squared error (MSE) was used as loss function [25]. (Figure reproduced with permission of the authors.)

shown in the bottom right subfigure in Figure 111. Assuming that the jump $[\![u]\!]$ has zero gradient in $\mathcal{B}_+$, take the gradient of rate form in Eq. $(424)_2$ and symmetrize to obtain the small-strain rate[275]

$$(\nabla[\![u]\!] = 0 \text{ and } \nabla H_\Gamma = n\delta_\Gamma) \Rightarrow \dot{\epsilon} = \text{sym}(\nabla\dot{u}) + \text{sym}([\![\dot{u}]\!] \otimes n\delta_\Gamma), \text{ with sym}(b) := \frac{1}{2}(b + b^T). \quad (425)$$

Later in [297], an equation that looked similar to Eq. $(423)_1$, but in rate form, was introduced:[276]

$$\dot{u} = \dot{\bar{u}} + [\![\dot{u}]\!]H_\Gamma = (\dot{\bar{u}} + [\![\dot{u}]\!]f_\Gamma) + [\![\dot{u}]\!](H_\Gamma - f_\Gamma) = \dot{\tilde{u}} + [\![\dot{u}]\!](H_\Gamma - f_\Gamma), \text{ with } \dot{\tilde{u}} := \dot{\bar{u}} + [\![\dot{u}]\!]f_\Gamma, \quad (426)$$

where $\tilde{u}$, defined with the term $+[\![u]\!]$, is not the same as $\bar{u}$ defined with the term $-[\![u]\!]$ in Eq. $(423)_2$ of [25], even though Eq. $(423)_3$ looked similar to Eq. $(426)_1$. From Eq. $(426)_3$, the small-strain rate $\dot{\epsilon}$ in Eq. $(425)_3$ in terms of the smoothed velocity $\dot{\tilde{u}}$ is[277]

$$\dot{\epsilon} = \text{sym}(\nabla\dot{\tilde{u}}) + \text{sym}([\![\dot{u}]\!] \otimes n\delta_\Gamma) - \text{sym}([\![\dot{u}]\!] \otimes \nabla f_\Gamma), \quad (427)$$

which, when removing the overhead dot, is similar to, but different from, the small strain expression in [25], written as

$$\epsilon = \text{sym}(\nabla u) + \text{sym}([\![u]\!] \otimes n\delta_\Gamma) - \text{sym}([\![u]\!] \otimes \nabla f_\Gamma), \quad (428)$$

where the first term was u and not $\tilde{u} = u + [\![u]\!]f_\Gamma$ in the notation of [25].[278]

---

[275] [297], Eq. (2.2).

[276] [297], Eq. (2.15).

[277] [297], Eq. (2.17).

[278] Recall that u in [25] (Eq. $(423)_1$), the "large-scale (or conformal) displacement field" without displacement jump, is equivalent to $\bar{u}$ in [297] (Eq. $(424)_1$), but is of course not the $\bar{u}$ in [25] (Eq. $(423)_2$).

Typically, in this type of formulation [297], once the traction-separation law $T(\llbracket u \rrbracket)$ in Eq. (422) was available (e.g., Figure 110), then given the traction T, the displacement jump $\llbracket u \rrbracket$ was solved for using Eq. (422) at each Gauss point within a constant-strain triangular (CST) element [25].

At this point, it is no longer necessary to review further this continuum formulation for displacement jumps to return to the training of the macroscale RNN with LSTM units, which the authors of [25] called the "Macroscale data-driven constitutive model" (Figure 14, row 3, right), using the data generated from the simulations using the mesoscale RVE (Figure 113) and the mesoscale RNN with LSTM units, called the "Mesoscale data-driven constitutive model" (Figure 14, row 2, right), obtained earlier.

The mesoscale RNN with LSTM units ("mesoscale data-driven constitutive model") was first validated using the mesoscale RVE with embedded discontinuities (Figure 113), discretized into finite elements, and subjected to imposed displacements at the top. This combined FEM and RNN with LSTM units on the mesoscale RVE is denoted FEM-LSTM, with results compared well with those obtained from the coupled FEM and DEM (denoted as FEM-DEM), as shown in Figure 114.

Once validated, the FEM-LSTM model for the mesocale RVE was used to generate data to train the macroscale RNN with LSTM units (called "Macroscale data-driven constitutive model") by imposing displacement jumps at the top of the mesoscale RVE (Figure 112), very much like what was done with the microscale RVE (Figure 106, Figure 110), just at a larger scale.

The accuracy of the macroscale RNN with LSTM units ("Macroscale data-driven constitutive model") is illustrated in Figure 115, where the normal tractions under displacement loading were compared to results obtained with the mesoscale RVE (Figure 112, Figure 113), which was used for generating the training data. Once established, the macroscale RNN with LSTM units is used in field-size macroscale simulations. Since there is no further interesting insights into the use of deep learning, we stop of review of [25] here.

Remark 11.10. *No non-linear stress-strain relation.* In the end, the authors of [25] only used Figure 12 to motivate the double porosity (in Majella limestone) in their macroscale modeling and simulations, which did not include the characteristic non-linear stress-strain relation found experimentally in Majella limestone as shown in Figure 13. All nonlinear responses considered in [25] came from the nonlinear traction-separation law obtained from DEM simulations in which the particles themselves were elastic, even though the Hertz contact force-displacement relation was nonlinear [303] [304] [302]. See Remark 11.7.   ∎

Remark 11.11. *Physics-Informed Neural Networks (PINNs) applied to solid mechanics.* The PINN method discussed in Section 9.5 has been applied to problems in solid mechanics [305]: Linear elasticity (square plate, plane strain, trigonometric body force, with exact solution), nonlinear elasto-plasticity (perforated plate with circular hole, under plane-strain condition and von-Mises elastoplasticity, subjected to uniform extension, showing localized shear band). Less accuracy was encountered for solutions that presented discontinuiies (localized high gradients) in the materials properties or at the boundary conditions; see Remark 7.7 and Remark 9.5.   ∎

12 Application 3: Fluids, turbulence, reduced-order models

The general ideas behind the work in [26] were presented in Section 2.3.3 further above. In this section, we discuss some details of the formulation, starting with a brief primer on Proper Orthogonal Decomposition (POD) for unfamiliar readers.

12.1 Proper orthogonal decomposition (POD)

The presentation of the *continuous* formulation of POD in this section follows [306]. Consider the separation of variables of a time-dependent function $u(\boldsymbol{x}, t)$, which could represent a component of a velocity
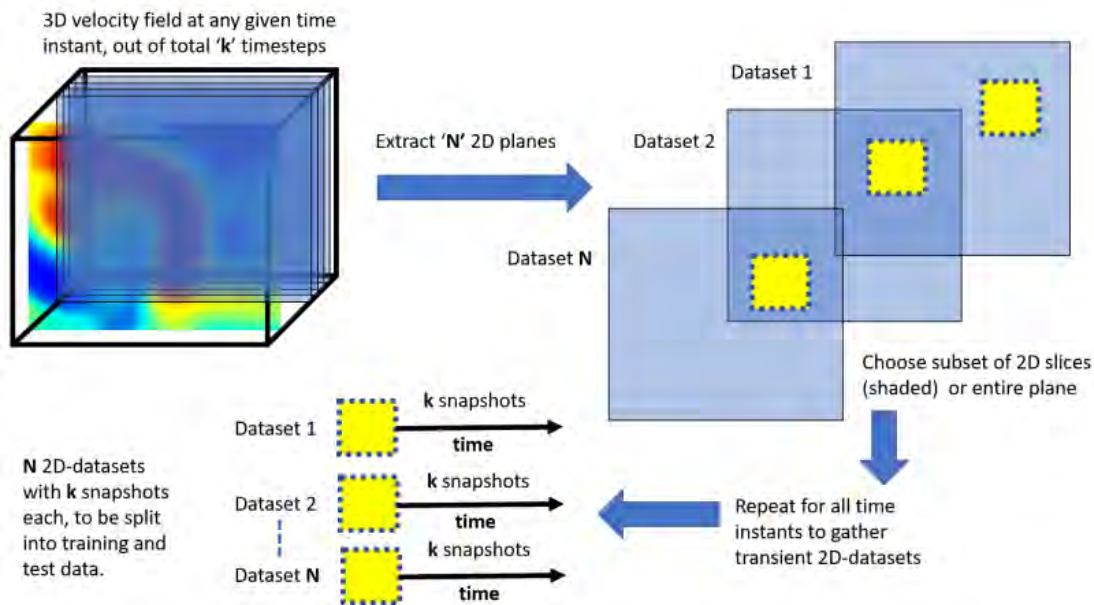
Figure 116: *2-D datasets for training neural networks* (Sections 2.3.3, 12.1). Extract 2-D datasets from 3-D turbulent flow field evolving in time. From the 3-D flow field, extract N equidistant 2-D planes (slices). Within each 2-D plane, select a region (yellow square), and k temporal snapshots of this region as it evolves in time to produce a dataset. Among these N datasets, each containing k snapshots of the same region within each 2-D plane, the majority of the datasets is used for training, and the rest for testing; see Remark 12.1. For each dataset, the reduced POD basis consists of $m \ll k$ POD modes with highest eigenvalues of a matrix constructed from the k snapshots (Figure 18) [26]. (Figure reproduced with permission of the author.)

field or a magnetic potential in a 3-D domain $\mathcal{B}$, where $\boldsymbol{x} \in \mathcal{B}$, and t a time parameter:

$$u(\boldsymbol{x}, t) = \alpha(t)\phi(\boldsymbol{x}) , \tag{429}$$

where $\alpha(t)$ is the time-dependent amplitude, and $\phi(\boldsymbol{x})$ a function of $\boldsymbol{x}$ representing the most typical spatial structure of $u(\boldsymbol{x}, t)$. The goal is to find the function $\phi(\boldsymbol{x})$ that maximizes the square of the amplitude $\alpha(t)$ (or component of $u(\boldsymbol{x}, t)$ along $\phi(\boldsymbol{x})$):

$$\alpha(t) = \frac{(u, \phi)}{(\phi, \phi)} , \text{ with } (u, \phi) := \int_{\mathcal{B}} u(\boldsymbol{y}, t)\phi(\boldsymbol{y})d\mathcal{B}(\boldsymbol{y}) , \tag{430}$$

$$(\phi, \phi) = 1 \Rightarrow \alpha(t) = (u, \phi) , \tag{431}$$

i.e., if $\phi(\boldsymbol{x})$ is normalized to 1, then the amplitude $\alpha(t)$ is the spatial scalar product between $u(\boldsymbol{x}, t)$ and $\phi(\boldsymbol{x})$, with $(\cdot, \cdot)$ designating the spatial scalar product operator. Let $\langle \cdot \rangle$ designate the time average operator, i.e.,

$$\langle f(t) \rangle = \frac{1}{T} \int_{t=0}^{t=T} f(t) \, dt , \tag{432}$$

where T designates the maximum time. The goal now is to find $\phi(\boldsymbol{x})$ such that the time average of the square of the amplitude $\alpha(t)$ is maximized:

$$\phi(\boldsymbol{x}) = \underset{\phi}{\operatorname{argmax}}(\langle \alpha^2(t) \rangle) = \underset{\phi}{\operatorname{argmax}}(\langle (u, \phi)^2 \rangle) , \tag{433}$$
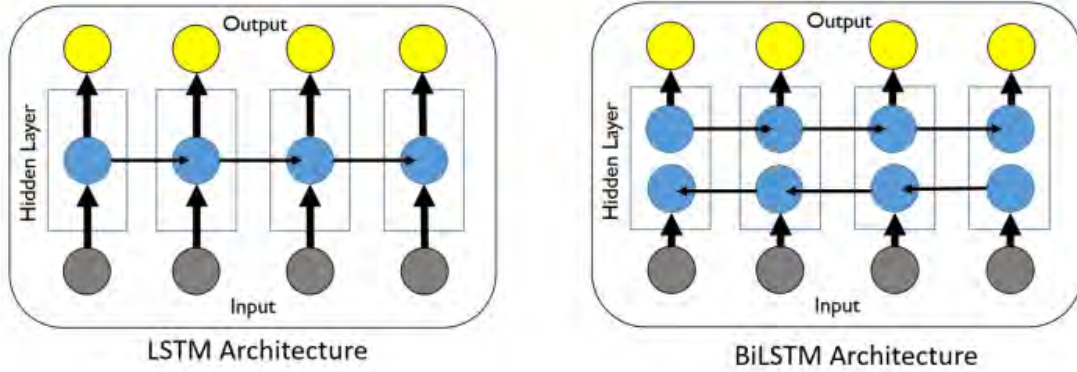
Figure 117: *LSTM unit and BiLSTM unit* (Sections 2.3.2, 2.3.3, 7.2, 12.2). Each blue dot is an original LSTM unit (in *folded* form Figure 81 in Section 7.2, without peepholes as shown in Figure 15), thus a single hidden layer. The above LSTM architecture (left) in *unfolded* form corresponds to Figure 82, with the inputs at state $[k]$ designated by $\boldsymbol{x}^{[k]}$ and the corresponding outputs by $\boldsymbol{h}^{[k]}$, for $k = \ldots, n-1, n, n+1, \ldots$. In the BiLSTM architecture (right), there are two LSTM units in the hidden layer, with the forward flow of information in the bottom LSTM unit, and the backward flow in the top LSTM unit [26]. (Figure reproduced with permission of the author.)

which is equivalent to maximizing the amplitude $\alpha(t)$, and thus the information content of $u(\boldsymbol{x}, t)$ in $\phi(\boldsymbol{x})$, which in turn is also called "coherent structure". The square of the amplitude in Eq. (433) can be written as

$$\lambda := \alpha^2(t) = (u, \phi)^2 = \int_{\mathcal{B}} \left[\int_{\mathcal{B}} u(\boldsymbol{x}, t) u(\boldsymbol{y}, t) \phi(\boldsymbol{y}) \, d\boldsymbol{y} \right] \phi(\boldsymbol{x}) \, d\boldsymbol{x} , \tag{434}$$

so that λ is the component (or projection) of the term in square brackets in Eq. (434) along the direction $\phi(\boldsymbol{x})$. The component λ is maximal if this term in square brackets is colinear with (or "parallel to") $\phi(\boldsymbol{x})$, i.e.,

$$\int_{\mathcal{B}} u(\boldsymbol{x}, t) u(\boldsymbol{y}, t) \phi(\boldsymbol{y}) \, d\boldsymbol{y} = \lambda \phi(\boldsymbol{x}) \Rightarrow \int_{\mathcal{B}} \langle u(\boldsymbol{x}, t) u(\boldsymbol{y}, t) \rangle \phi(\boldsymbol{y}) \, d\boldsymbol{y} = \lambda \phi(\boldsymbol{x}) , \tag{435}$$

which is a *continuous* eigenvalue problem with the eigenpair being $(\lambda, \phi(\boldsymbol{x}))$. In practice, the dynamic quantity $u(\boldsymbol{x}, t)$ is sampled at k discrete times $\{t_1, \ldots, t_k\}$ to produce k snapshots, which are functions of $\boldsymbol{x}$, assumed to be linearly independent, and ordered in matrix form as follows:

$$\{u(\boldsymbol{x}, t_1) \ldots, u(\boldsymbol{x}, t_k)\} =: \{u_1(\boldsymbol{x}), \ldots, u_k(\boldsymbol{x})\} =: \boldsymbol{u}(\boldsymbol{x}) . \tag{436}$$

The coherent structure $\phi(\boldsymbol{x})$ can be expressed on the basis of the snapshots as

$$\phi(\boldsymbol{x}) = \sum_{i=1}^{i=k} \beta_i u_i(\boldsymbol{x}) = \boldsymbol{\beta} \bullet \boldsymbol{u} , \text{ with } \boldsymbol{\beta} = \{\beta_1, \ldots, \beta_k\} . \tag{437}$$

As a result of the discrete nature of Eq. (436) and Eq. (437), the eigenvalue problem in Eq. (435) is discretized into

$$\boldsymbol{C}\boldsymbol{\beta} = \lambda \boldsymbol{\beta} , \text{ with } \boldsymbol{C} := \left[\frac{1}{k} \int_{\mathcal{B}} \boldsymbol{u}(\boldsymbol{y}) \otimes \boldsymbol{u}(\boldsymbol{y}) d\boldsymbol{y} \right] , \tag{438}$$

where the matrix $\boldsymbol{C}$ is symmetric, positive definite, leading to positive eigenvalues in k eigenpairs $(\lambda_i, \boldsymbol{\beta}_i)$, with $i = 1, \ldots, k$. With $\phi(\boldsymbol{x})$ now decomposed into k linearly independent directions $\phi_i(\boldsymbol{x}) := \beta_i u_i(\boldsymbol{x})$
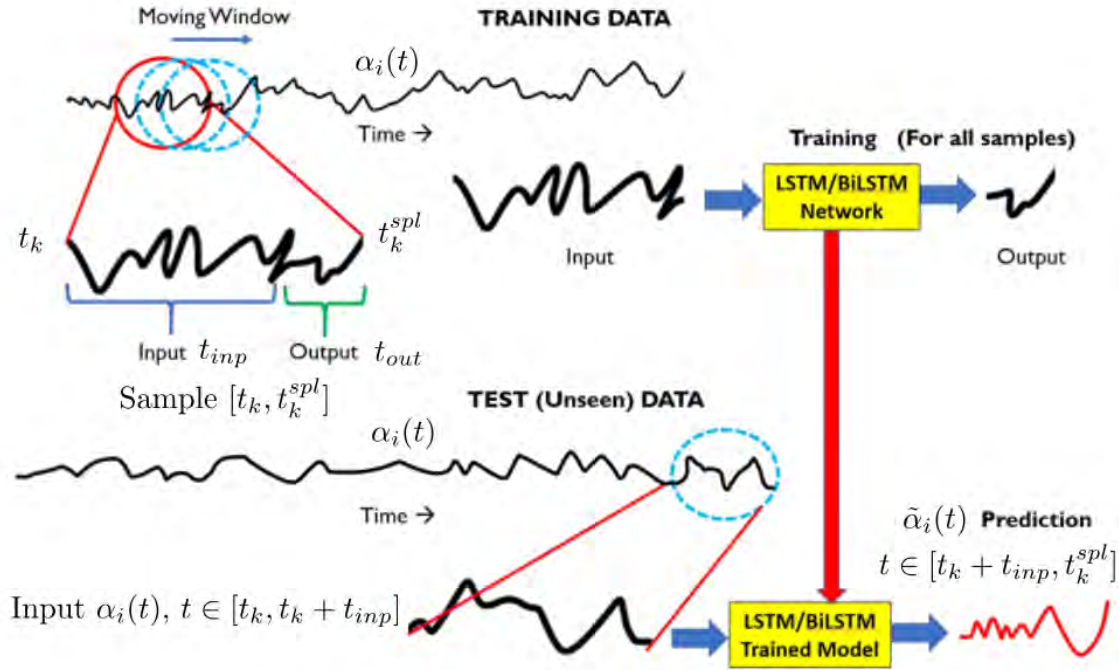
Figure 118: *LSTM/BiLSTM training strategy* (Sections 12.2.1, 12.2.2). From the 1-D time series $\alpha_i(t)$ of each dominant mode ϕ_i, for $i = 1, \ldots, m$, use a moving window to extract thousands of samples $\alpha_i(t)$, $t \in [t_k, t_k^{spl}]$, with t_k being the time of snapshot k. Each sample is subdivided into an input signal $\alpha_i(t)$, $t \in [t_k, t_k + t_{inp}]$ and an output signal $\alpha_i(t)$, $t \in [t_k + t_{inp}, t_k^{spl}]$, with $t_k^{spl} - t_k = t_{inp} + t_{out}$ and $0 < t_{out} \le t_{inp}$. The windows can be overlapping. These thousands of input/output pairs were then used to train LSTM-ROM networks, in which LSTM can be replaced by BiLSTM (Figure 117). The trained LSTM/BiLSTM-ROM networks were then used to predict $\tilde{\alpha}_i(t)$, $t \in [t_k + t_{inp}, t_k^{spl}]$ of the test datasets, given $\alpha_i(t)$, $t \in [t_k, t_k + t_{inp}]$ [26]. (Adapted with permission of the author.)

according to Eq. (437), the dynamic quantity $u(\boldsymbol{x}, t)$ in Eq. (429) can now be written as a linear combination of $\phi_i(\boldsymbol{x})$, with $i = 1, \ldots, k$, each with a different time-dependent amplitude $\alpha_i(t)$, i.e.,

$$u(\boldsymbol{x}, t) = \sum_{i=1}^{i=k} \alpha_i(t)\phi_i(\boldsymbol{x}) , \tag{439}$$

which is called a proper orthogonal decomposition of $u(\boldsymbol{x}, t)$, and is recorded in Figure 18 as "Full POD reconstruction". Technically, Eq. (437) and Eq. (439) are approximations of infinite-dimensional functions by a finite number of linearly-independent functions.

Usually, a subset of $m \ll k$ POD modes are selected such that the error committed by truncating the basis as done in Eq. (4) would be small compared to Eq. (439), and recalled here for convenience:

$$u(\boldsymbol{x}, t) \approx \sum_{j=1}^{j=m} \phi_{i_j}(\boldsymbol{x})\alpha_{i_j}(t) , \text{ with } m \ll k \text{ and } i_j \in \{1, \ldots, k\} . \tag{4}$$

One way is to select the POD modes corresponding to the highest eigenvalues (or energies) in Eq. (435); see Step (2) in Section 12.2.2.

Remark 12.1. *Reduced-order POD.* Data for two physical problems were available from numerical simulations: (1) the Force Isotropic Turbulence (ISO) dataset, and (2) the Magnetohydrodynamic Turbulence
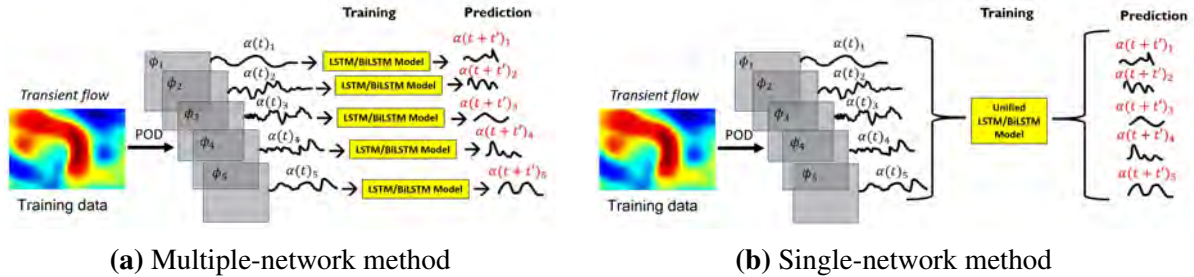
(a) Multiple-network method   **(b)** Single-network method

Figure 119: *Two methods of developing LSTM-ROM* (Section 2.3.3, 12.2.2). For each physical problem (ISO or MHD): (a) Each dominant POD mode has a network to predict $\alpha_i(t + t')$, with t', given $\alpha_i(t)$; (b) All m POD dominant modes share the same network to predict $\{\alpha_i(t + t'),\ i = 1, \ldots, m\}$, with t', given $\{\alpha_i(t),\ i = 1, \ldots, m\}$ [26]. (Figure reproduced with permission of the author.)

(MHD) dataset [105]. For each physical problem, the authors of [26] employed $N = 6$ equidistant 2-D planes (slices, Figure 116), with 5 of those 2-D planes used for training, and 1 remaining 2-D plane used for testing (see Section 6.1). The same sub-region of the 6 equidistant 2-D plane (yellow squares in Figure 116) was used to generate 6 training / testing datasets. For each training / testing dataset, $k = 5,023$ snapshots for the ISO dataset and $k = 1,024$ snapshots for the MHD dataset were used in [26]. The reason was because the ISO dataset contained 5,023 time steps, whereas the MHD dataset contained 1,024 time steps. So the number of snapshots was the same as the number of time steps. These snapshots were reduced to $m \in \{5, \ldots, 10\}$ POD modes with highest eigenvalues (thus energies), which were much fewer than the original k snapshots, since $m \ll k$. See Remark 12.3. ∎

Remark 12.2. *Another method of finding the POD modes* without forming the symmetric matrix C in Eq. (438) is by using the Singular Value Decomposition (SVD) directly on the rectangular matrix of the sampled snapshots, discrete in both space and time. The POD modes are then obtained from the left singular vectors times the corresponding singular values. A reduced POD basis is obtained next based on an information-content matrix. See [306], where POD was applied to efficiently solve nonlinear electromagnetic problems governed by Maxwell's equations with nonlinear hysteresis at low frequency (10 kHz), called static hysteresis, discretized by a finite-element method. See also Remark 12.4. ∎

12.2 POD with LSTM-Reduced-Order-Model

Typically, once the dominant POD modes of a physical problem (ISO or MHD) were identified, a reduced-order model (ROM) can be obtained by projecting the governing partial differential equations (PDEs) onto the basis of the dominant POD modes using, e.g., Galerkin projection (GP). Using this method, the authors of [306] employed full-order simulations of the governing electro-magnetic PDE with certain input excitation to generate POD modes, which were then used to project similar PDE with different parameters and solved for the coefficients $\alpha_i(t)$ under different input excitations.

12.2.1 Goal for using neural network

Instead of using GP on the dominant POD modes of a physical problem to solve for the coefficients $\alpha_i(t)$ as described above, deep-learning neural network was used in [26] to predict the next value of $\alpha_i(t+t')$, with $t' > 0$, given the current value $\alpha_i(t)$, for $i = 1, \ldots, m$.

To achive this goal, LSTM/BiLSTM networks (Figure 117) were trained using thousands of paired short input / output signals obtained by segmenting the time-dependent signal $\alpha_i(t)$ of the dominant POD mode ϕ_i [26].

(a) Hurst Exponent of each POD mode

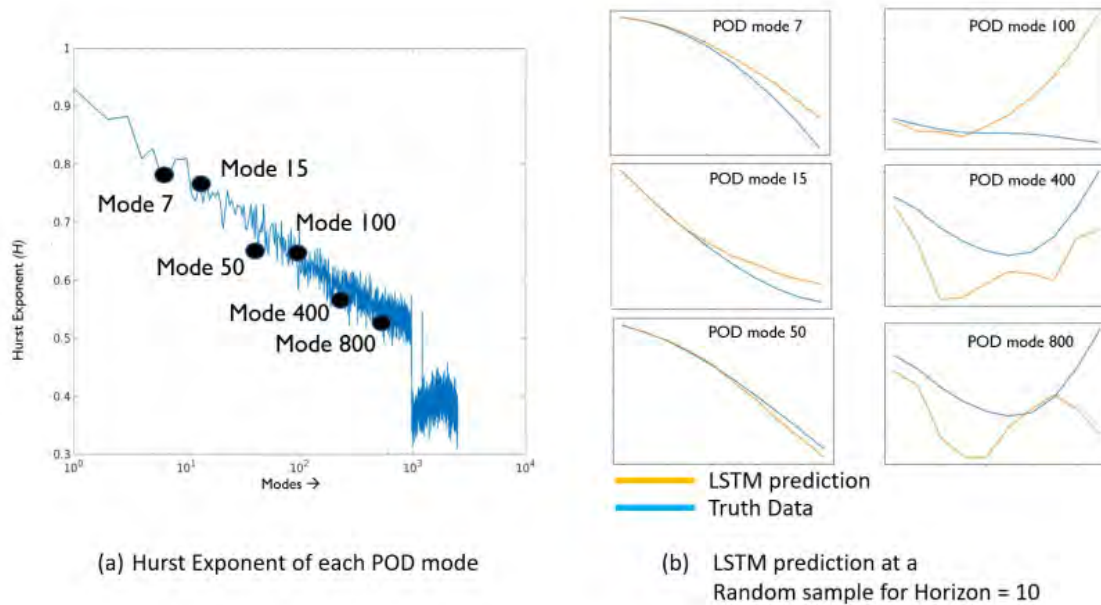(b) LSTM prediction at a
     Random sample for Horizon = 10

Figure 120: *Hurst exponent vs POD-mode rank for Isotropic Turbulence (ISO)* (Sections 12.3). POD modes with larger eigenvalues (Eq. (438)) are higher ranked, and have lower rank number, e.g., POD mode rank 7 has larger eigenvalue, and thus more dominant, than POD mode rank 50. The Hurst exponent, even though fluctuating, trends downward with the POD mode rank, but not monotonically, i.e., for two POD modes sufficiently far apart (e.g., mode 7 and mode 50), a POD mode with lower rank generally has a lower Hurst exponent. The first 800 POD modes for the ISO problem have the Hurst exponents higher than 0.5, and are thus persistent [26]. (Adapted with permission of the author.)

12.2.2 Data generation, training and testing procedure

The following procedure was adopted in [26] to develop their LSTM-ROM for two physical problems, ISO and MHD; see Remark 12.1. For each of the two physical problems (ISO and MHD), the following steps were used:

(1) From the 3-D computational domain of a physical problem (ISO or MHD), select N equidistant 2-D planes that slice through this 3-D domain, and select the same subregion for all of these planes, the majority of which is used for the training datasets, and the rest for the test datasets (Figure 116 and Remark 12.1 for the actual value of N and the number of training datasets and test datasets employed in [26]).

(2) For each of the training datasets and test datasets, extract from k snapshot a few $m \ll k$ dominant POD modes ϕ_i, $i = 1, \ldots, m$ (with the highest energies / eigenvalues) and their corresponding coefficients $\alpha_i(t)$, $i = 1, \ldots, m$, by solving the eigenvalue problem Eq. (438), then use Eq. (437) to obtain the POD modes ϕ_i, and Eq. (431) to obtain $\alpha_i(t)$ for use in Step (3).

(3) The time series of the coefficient $\alpha_i(t)$ of the dominant POD mode $\phi_i(\boldsymbol{x})$ of a *training* dataset is chunked into thousands of small samples with $t \in [t_k, t_k^{spl}]$, where t_k is the time of the *kth* snapshot, by a moving window. Each sample is subdivided into two parts: The input part with time length t_{inp} and the output part with time length t_{out}, Figure 118. These thousands of input/output pairs were then used to train LSTM/BiLSTM networks in Step (4). See Remark 12.3.

(4) Use the input/output pairs generated from the training datasets in Step (3) to train LSTM/BiLSTM-ROM networks. Two methods were considered in [26]:

    (a) *Multiple-network method:* Use a separate RNN for each of the m dominant POD modes to separately predict the coefficient $\alpha_i(t)$, for $i = 1, \ldots, m$, given a sample input, see Figure 119a and Figure 118. Hyperparameters (layer width, learning rate, batch size) are tuned for the most dominant POD mode and reused for training the other neural networks.

    (b) *Single-network method:* Use the same RNN to predict the coefficients $\alpha_i(t)$, $i = 1, \ldots, m$ of all m dominant POD modes at once, given a sample input, see Figure 119b and Figure 118.

The single-network method better captures the inter-modal interactions that describe the energy transfer from larger to smaller scales. Vortices that spread over multiple dominant POD modes also support the single-network method, which does not artificially constrain flow features to separate POD modes.

(5) Validation: Input/output pairs similar to those for training in Step (3) were generated from the *test* dataset for validation.[279] With a short time series of the coefficient $\alpha_i(t)$ of the dominant POD mode $\phi_i(\boldsymbol{x})$ of the *test* dataset, with $t \in [t_k, t_k + t_{inp}]$, where t_k is the time of the *kth* snapshot, and $0 < t_{inp}$, as input, use the trained LSTM/BiLSTM networks in Step (4) to predict the values of $\tilde{\alpha}_i(t)$, with $t \in [t_k + t_{inp}, t_k^{spl}]$, where t_k^{spl} is the time at the end of the sample, such that the sample length is $t_k^{spl} - t_k = t_{inp} + t_{out}$ and $0 < t_{out} \leq t_{inp}$; see Figure 118. Compute the error between the predicted value $\tilde{\alpha}_i(t + t_{out})$ and the target value of $\alpha_i(t + t_{out})$ from the test dataset. Repeat the same prediction procedure for all m dominant POD modes chosen in Step (2). See Remark 12.3.

(6) At time t, use the predicted coefficients $\tilde{\alpha}_i(t)$ together with the dominant POD modes $\phi_i(\boldsymbol{x})$, for $i = 1, \ldots, m$, to compute the flow field dynamics $u(\boldsymbol{x}, t)$ using Eq. (4).

12.3 Memory effects of POD coefficients on LSTM models

The results and deep-learning concepts used in [26] were presented in the motivational Section 2.3.3 above, Figure 20. In this section, we discuss some details of the formulation.

Remark 12.3. Even though the value of $t_{inp} = t_{out} = 0.1\,sec$ was given in [26] as an example, all LSTM/BiLSTM networks in their numerical examples were trained using input/output pairs with $t_{inp} = t_{out} = 10 \times \Delta t$, i.e., 10 time steps for both input and output samples. With the overall simulation time for both physical problems (ISO and MHD) being $2.056\,secs$, the time step size is $\Delta t_{\text{ISO}} = 2.056/5023 = 4.1 \times 10^{-4}\,sec$ for ISO, and $\Delta t_{\text{MHD}} = 2.056/1024 = 2 \times 10^{-3}\,sec = 5\Delta t_{\text{ISO}}$ for MHD. See also Remark 12.1. ∎

The "U-velocity field for all results" was mentioned in [26], but without a definition of "U-velocity", which was possibly the x component of the 2-D velocity field in the 2-D planes (slices) used for the datasets, with "V-velocity" being the corresponding y components.

The BiLSTM networks in the numerical examples were not as accurate as the LSTM networks for both physical problems (ISO and MHD), despite having more computations; see Figure 117 above and Figure 20 in Section 2.3.3. The authors of [26] conjectured that a reason could be due to the randomness nature of turbulent flows, as opposed to the high long-term correlation found in natural human languages, for which BiLSTM was designed to address.

Since LSTM architecture was designed specifically for sequential data with memory, it was sought in [26] to quantify whether there was "memory" (or persistence) in the time series of the coefficients $\alpha_i(t)$ of

---

[279] The authors of [26] did not have a validation dataset as defined in Section 6.1.

the POD mode $\phi_i(\boldsymbol{x})$. To this end, the Hurst exponent[280] was used to quantify the presence or absence of long-term memory in the time series $\mathcal{S}_i(k,n)$:

$$\mathcal{S}_i(k,n) := \{\alpha_i(t_{k+1}),\ldots,\alpha_i(t_{k+n})\}, \quad \underset{i,n \text{ fixed}}{\mathbb{E}}\left(\frac{R_i(k,n)}{\sigma_i(k,n)}\right) = C_i e^{H_i}, \tag{440}$$

$$R_i(k,n) = \max \mathcal{S}_i(k,n) - \min \mathcal{S}_i(k,n), \ \sigma_i(k,n) = \text{stdev}\,[\mathcal{S}_i(k,n)], \tag{441}$$

where $\mathcal{S}_i(k,n)$ is a sequence of n steps of $\alpha_i(t)$, starting at snapshot time t_{k+1}, $\mathbb{E}$ is the expectation of the ratio of range R_i over standard deviation σ_i for many samples $\mathcal{S}_i(k,n)$, with different values of k keeping (i,n) fixed, $R_i(k,n)$ the range of sequence $\mathcal{S}_i(k,n)$, $\sigma_i(k,n)$ the standard deviation of sequence $\mathcal{S}_i(k,n)$, C_i a constant, H_i the Hurst exponent for POD mode $\phi_i(\boldsymbol{x})$.

A Hurst coefficient of $H = 1$ indicates persistent behavior, i.e., an upward trend in a sequence is followed by an upward trend. If $H = 0$ the behavior that is represented by time series data is anti-persistent, i.e., an upward trend is followed by a downward trend (and vice versa). The case $H = 0.5$ indicates random behavior, which implies a lack of memory in the underlying process.

The effects of prediction horizon and persistence on the prediction accuracy of LSTM network were studied in [26]. Horizon is the number of steps after the input sample that a LSTM model would predict the values of $\alpha_i(t)$, and is proportional to the output time length t_{out}, assuming constant time step size. Persistence refers to the amount of correlation among subsequent realizations within sequential data, i.e., the presence of the long-term memory.

To this end, they selected one dataset (training or testing), and followed the multiple-network method in Step (4) of Section 12.2.2 to develop a different LSTM network model for each POD mode with "non-negligible eigenvalue". For both the ISO (Figure 120) and MHD problems, the 800 highest ranked POD modes were used.

A baseline horizon of 10 steps were used, for which the prediction errors were $\{1.08, 1.37, 1.94, 2.36, 2.03, 2.00\}$ for POD ranks $\mathcal{R} = \{7, 15, 50, 100, 400, 800\}$, and for Hurst exponents $\{0.78, 0.76, 0.652, 0.653, 0.56, 0.54\}$, respectively. So the prediction error increased from 1.08 for POD rank 7 to 2.36 for POD rank 100, then decreased slightly for POD ranks 400 and 800. The time histories of the corresponding coefficients $\alpha_i(t)$, with $i \in \mathcal{R}$ on the right of Figure 120 provided only some qualitative comparison between the predicted values and the true values, but did not provide the scale of the actual magnitude of $\alpha_i(t)$, nor the time interval of these plots. For example, the magnitude of $\alpha_{800}(t)$ could be very small compared to that of $\alpha_7(t)$, given that the POD modes were normalized in the sense of Eq. (431). Qualitatively, the predicted values compared well with the true values for POD ranks 7, 15, 50. So even though there was a divergence between the predicted value and the true value for POD ranks 100 and 800, but if the magnitude of $\alpha_{100}(t)$ and $\alpha_{800}(t)$ were very small compared to those of the dominant POD modes, there would be less of a concern.

Another expected result was that for the same POD mode rank lower than 50, the error increased dramatically with the prediction horizon. For example, for POD rank 7, the errors were $\{1.08, 6.86, 15.03, 19.42, 21.40\}$ for prediction horizons of $\{10, 25, 50, 75, 100\}$ steps, respectively. Thus the error (at 21.40) for POD rank 7 with horizon 100 steps was more than ten times higher than the error (at 2.00) for POD rank 800 with horizon 10 steps. For POD mode rank 50 and higher, the error did not increase as much with the horizon, but stayed at about the same order of magnitude as the error for horizon 25 steps.

A final note is whether the above trained LSTM-ROM networks could produce accurate prediction for flow dynamics with different parameters such as Reynolds number, mass density, viscosity, geometry, initial conditions, etc., particularly that both the ISO and MHD datasets were created for a single Reynolds

---

[280] "Hurst exponent", Wikipedia, version 22:09, 30 October 2020.

number, which was not mentioned in [26], who mentioned that their method (and POD in general) would work for a "narrow range of Reynolds numbers" for which the flow dynamics is qualitatively similar, and for "simplified flow fields and geometries".

Remark 12.4. *Use of POD-ROM for different systems.* The authors of [306] studied The flexibility of POD reduced-order models to solve nonlinear electromagnetic problems by varying the excitation form (e.g., square wave instead of sine wave) and by using the undamped (without the first-order time derivative term) snapshots in the simulation of the damped case (with the first-order time derivative term). They demonstrated via numerical examples involving nonlinear power-magnetic-component simulations that the reduced-order models by POD are quite flexible and robust. See also Remark 12.2.   ■

Remark 12.5. *pyMOR - Model Order Reduction with Python.* Finally, we mention the software pyMOR,[281] which is "a software library for building model order reduction applications with the Python programming language. Implemented algorithms include reduced basis methods for parametric linear and non-linear problems, as well as system-theoretic methods such as balanced truncation or IRKA (Iterative Rational Krylov Algorithm). All algorithms in pyMOR are formulated in terms of abstract interfaces for seamless integration with external PDE (Partial Differential Equation) solver packages. Moreover, pure Python implementations of FEM (Finite Element Method) and FVM (Finite Volume Method) discretizations using the NumPy/SciPy scientific computing stack are provided for getting started quickly." It is noted that pyMOR includes POD and "Model order reduction with artificial neural networks", among many other methods; see the documentation of pyMOR. Clearly, this software tool would be applicable to many physical problems, e.g., solids, structures, fluids, electromagnetics, coupled electro-thermal simulation, etc.   ■

FOM   NM-LSPG-HR   LS-LSPG-HR
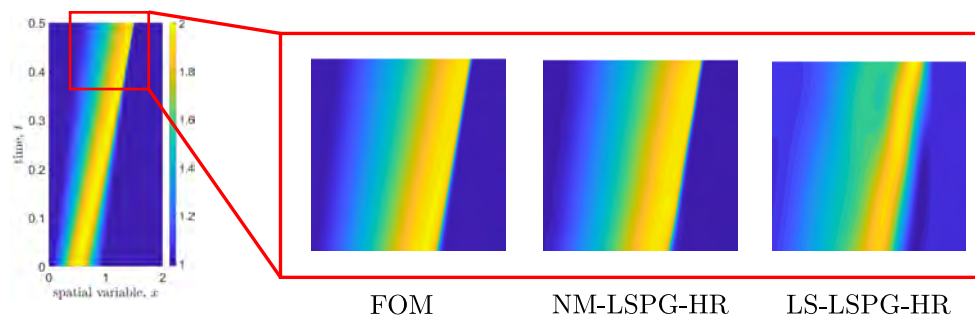
Figure 121: *Space-time solution of inviscid 1D-Burgers' equation* (Section 12.4.1). The solution shows a characteristic steep spatial gradient, which shifts and further steepens in the course of time. The FOM solution (left) and the solution of the proposed hyper-reduced ROM (center), in which the solution subspace is represented by a nonlinear manifold in the form of a feed-forward neural network (Section 4) (NM-LSPG-HR), show an excellent agreement, whereas the spatial gradient is significantly blurred in the solution obtained with a hyper-reduced ROM based on a linear subspace (LS-LSPG-HR) (right). The FOM is a obtained upon a finite-difference approximation in the spatial domain with $N_s = 1001$ grid points (i.e., degrees of freedom); the backward Euler scheme is employed for time-integration using a step size $\Delta t = 1 \times 10^{-3}$. Both ROMs only use $n_s = 5$ generalized coordinates. The NM-LSPG-HR achieves a maximum relative error of less than 1%, while the maximum relative error of the LS-LSPG-HR is approximately 6%. (Figure reproduced with permission of the authors.)

---

[281] See pyMOR website https://pymor.org/.

12.4 Reduced order models and hyper-reduction

Reduction of computational expense also is the main aim of *nonlinear manifold reduced-order models* (NM-ROMs), which have recently been proposed in [47],[282] where the approach belongs to the class of *projection-based methods*. Projection-based methods rely on the idea that solutions to physical simulations lie in a subspace of small dimensionality as compared to the dimensionality of high-fidelity models, which we obtain upon discretization (e.g., by finite elements) of the governing equations. In classical projection-based methods, such *"intrinsic solution subspaces"* are spanned by a set of appropriate basis vectors that capture the essential features of the full-order model (FOM), i.e., the subspace is assumed to be linear. We refer to [307] for a survey on projection-based linear subspace methods for parametric systems.[283]

12.4.1 Motivating example: 1D Burger's equation

The effectiveness of linear subspace methods is directly related to the dimensionality of the basis to represent solutions with sufficient accuracy. Advection-dominated problems and problems with solutions that exhibit large ("sharp") gradients, however, are characterized by a large Kolmogorov n-width,[284] which is adverse to linear subspace methods. As examples, The authors of [47] mention hyperbolic equations with large Reynolds number, Boltzmann transport equations and traffic flow simulations. Many approaches to construct efficient ROMs in adverse problems are based on the idea to enhance the "solution representability" of the linear subspace, e.g., by using adaptive schemes tailored to particular problems. Such problem-specific approaches, however, suffer from limited generality and a the necessity of a-priori knowledge as, e.g., the (spatial) direction of advection. In view of these drawbacks, the transition to a solution representation by *nonlinear manifolds* rather than using linear subspaces in projection-based ROMs was advocated in [47].

Burger's equation serves as a common prototype problem in numerical methods for nonlinear partial differential equations (PDEs) and MOR, in particular. The inviscid Burgers' equation in one spatial dimension is given by

$$\frac{\partial u(x,t;\mu)}{\partial t} + u(x,t;\mu)\frac{\partial u}{\partial x} = 0, \qquad x \in \Omega = [0,2], \qquad t \in [0,T]. \tag{442}$$

Burgers' equation is a first-order hyperbolic equation which admits the formation of shock waves, i.e., regions with steep gradients in field variables, which propagate in the domain of interest. Its left-hand side corresponds to material time-derivatives in Eulerian descriptions of continuum mechanics. In the balance of linear momentum, for instance, the field u represents the velocity field. Periodic boundary conditions and non-homogeneous initial conditions were assumed [47]:

$$u(2,t;\mu) = u(0,t;\mu), \qquad u(x,0;\mu) = \begin{cases} 1 + \frac{\mu}{2}\left(\sin\left(2\pi x - \frac{\pi}{2}\right) + 1\right) & \text{if } 0 \le x \le 1 \\ 1 & \text{otherwise} \end{cases}, \tag{443}$$

The above initial conditions are governed by a scalar parameter $\mu \in \mathcal{D} = [0.9, 1.1]$. In parametric MOR, ROMs are meant to be valid for not just a single value of the parameter μ, but it is supposed to be valid for a range of values of μ in the domain $\mathcal{D}$. For this purpose, the reduced-order space, irrespective of whether a linear subspace of FOM or a nonlinear manifold is used, is typically constructed using data obtained for different values of the parameter μ. In the example in [47], the solutions for the parameter set $\mu \in [0.9, 1.1]$

---

[282] Note that the authors also published a second condensed, but otherwise similar version of their article, see [48].

[283] As opposed to the classical *non-parametric* case, in which all parameters are fixed, *parametric* methods aim at creating ROMs which account for (certain) parameters of the underlying governing equations to vary in some given range. Optimization of large-scale systems, for which repeated evaluations are computationally intractable, is a classical use-case for methods in parametric MOR, see Benner *et al.* [307].

[284] Mathematically, the dimensionality of linear subspace that 'best' approximates a nonlinear manifold is described by the *Kolmogorov n-width*, see, e.g., [308] for a formal definition.

were used to construct the individual ROMs. Note that solution data corresponding to the parameter value $\mu = 1$, for which the ROMs were evaluated, was not used in this example.

Figure 121 shows a zoomed view of solutions to the above problem obtained with the FOM (left), the proposed nonlinear manifold-based ROM (NM-LSPG-HR, center) and a conventional ROM, in which the full-order solution is represented by a linear subspace. The initial solution is characterized by a "bump" in the left half of the domain, which is centered at $x = 1/2$. The advective nature of Burgers' problem causes the bump to move right, which also results in slopes of the bump to increase in movement direction but decrease on the averted side. The zoomed view in Figure 121 shows the region at the end ($t = 0.5$) of the considered time-span, in which the (negative) gradient of the solution has already steepened significantly. With as little as $n_s = 5$ generalized coordinates, the proposed nonlinear manifold-based approach (NM-LSPG-HR) succeeds in reproducing the FOM solution, which is obtained by a finite-difference approximation of the spatial domain using $N_s = 1001$ grid points; time-integration is performed by means of the backward Euler scheme with a constant step size of $\Delta t = 1 \times 10^{-3}$, which translates into a total of $n_t = 500$ time steps.

The ROM based on a linear subspace of the full-order solution (LS-LSPG-HR) fails to accurately reproduce the steep spatial gradient that develops over time, see Figure 121, right Instead, the bump is substantially blurred in the linear subspace-based ROM as compared to the FOM's solution (left). The maximum error over all time steps t_n relative to the full-order solution defined as:

$$\text{Maximum relative error} = \max_{n \in \{1,\dots,n_t\}} \frac{\left\|\widetilde{\boldsymbol{x}}(t_n; \mu) - \boldsymbol{x}(t_n; \mu)\right\|_2}{\left\|\boldsymbol{x}(t_n; \mu)\right\|_2}, \tag{444}$$

where $\boldsymbol{x}$ and $\widetilde{\boldsymbol{x}}$ denote FOM and ROM solution vectors, respectively, was considered in [47]. In terms of the above metric, the proposed nonlinear-manifold-based ROM achieves a maximum error of approximately 1 %, whereas the linear-subspace-based ROM shows a maximum relative error of 6 %. For the given problem, the linear-subspace-based ROM is approximately 5 to 6 times faster than the FOM. The nonlinear-manifold-based ROM, however, does not achieve any speed unless *hyper-reduction* is employed. Hyper-reduction (HR) methods provide means to efficiently evaluated nonlinear terms in ROMs without evaluations of the FOM (see Hyper-reduction, Section 12.4.4). Using hyper-reduction, a factor 2 speed-up is achieved for both the nonlinear-manifold and linear-subspace-based ROMs, i.e., the effective speed-ups amount to factors of 2 and 9–10, respectively.

The solution manifold can be represented by means of a shallow, sparsely connected feed-forward neural network [47] (see Statics, feedforward networks, Sections 4, 4.6). The network is trained in an unsupervised manner using the concept of autoencoders (see Autoencoder, Section 12.4.3).

12.4.2 Nonlinear manifold-based (hyper-)reduction

The NM-ROM approach proposed in [47] addressed nonlinear dynamical systems, whose evolution was governed by a set of nonlinear ODEs, which had been obtained by a semi-discretization of the spatial domain, e.g., by means of finite elements:

$$\dot{\boldsymbol{x}} = \frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \boldsymbol{f}(\boldsymbol{x}, t; \boldsymbol{\mu}), \qquad \boldsymbol{x}(0; \boldsymbol{\mu}) = \boldsymbol{x}_0(\boldsymbol{\mu}). \tag{445}$$

In the above relation, $\boldsymbol{x}(t; \boldsymbol{\mu})$ denotes the parameterized solution of the problem, where $\boldsymbol{\mu} \in \mathcal{D} \subseteq \mathbb{R}^{n_\mu}$ are an n_μ-dimensional vector of parameters; $\boldsymbol{x}_0(\boldsymbol{\mu})$ is the initial state. The function $\boldsymbol{f}$ represents the rate of change of the state, which is assumed to be nonlinear in the state $\boldsymbol{x}$ and possibly also in its other arguments,

i.e., time t and the vector of parameters $\boldsymbol{\mu}$:[285]

$$\boldsymbol{x} : [0, T] \times \mathcal{D} \to \mathbb{R}^{N_s}, \qquad \boldsymbol{f} : \mathbb{R}^{N_s} \times [0, T] \times \mathcal{D} \to \mathbb{R}^{N_s}. \tag{446}$$

The fundamental idea of any projection-based ROM is to approximate the original solution space of the FOM by a comparatively low-dimensional space $\mathcal{S}$. In view of the aforementioned shortcomings of linear subspaces, the authors of [47] proposed a representation in terms of a *nonlinear manifold*, which was described by the vector-valued function $\boldsymbol{g}$, whose dimensionality was supposed to be much smaller than that of the FOM:

$$\mathcal{S} = \{\boldsymbol{g}(\hat{\boldsymbol{v}}) \mid \hat{\boldsymbol{v}} \in \mathbb{R}^{n_s}\}, \qquad \boldsymbol{g} : \mathbb{R}^{n_s} \to \mathbb{R}^{N_s}, \qquad \dim(\mathcal{S}) = n_s \ll N_s. \tag{447}$$

Using the nonlinear function $\boldsymbol{g}$, an approximation $\widetilde{\boldsymbol{x}}$ to the FOM's solution $\boldsymbol{x}$ was constructed using a set of generalized coordinates $\widehat{\boldsymbol{x}} \in \mathbb{R}^{n_s}$:

$$\boldsymbol{x} \approx \widetilde{\boldsymbol{x}} = \boldsymbol{x}_{ref} + \boldsymbol{g}(\widehat{\boldsymbol{x}}), \tag{448}$$

where $\boldsymbol{x}_{ref}$ denoted a (fixed) reference solution. The rate of change $\dot{\boldsymbol{x}}$ was approximated by

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} \approx \frac{\mathrm{d}\widetilde{\boldsymbol{x}}}{\mathrm{d}t} = \boldsymbol{J}_g(\widehat{\boldsymbol{x}})\frac{\mathrm{d}\widehat{\boldsymbol{x}}}{\mathrm{d}t}, \qquad \boldsymbol{J}_g(\widehat{\boldsymbol{x}}) = \frac{\partial \boldsymbol{g}}{\partial \widehat{\boldsymbol{x}}} \in \mathbb{R}^{N_s \times n_s}, \tag{449}$$

where the Jacobian $\boldsymbol{J}_g$ spanned the tangent space to the manifold at $\widehat{\boldsymbol{x}}$. The initial conditions for the generalized coordinates were given by $\widehat{\boldsymbol{x}}_0 = \boldsymbol{g}^{-1}(\boldsymbol{x}_0 - \boldsymbol{x}_{ref})$, where $\boldsymbol{g}^{-1}$ denoted the inverse function to $\boldsymbol{g}$.

Note that *linear subspace* methods are included in the above relations as the special case where $\boldsymbol{g}$ is a linear function, which can be written in terms of a *constant* matrix $\boldsymbol{\Phi} \in \mathbb{R}^{N_s \times n_s}$, i.e., $\boldsymbol{g}(\widehat{\boldsymbol{x}}) = \boldsymbol{\Phi}\widehat{\boldsymbol{x}}$. In this case, the approximation to the solution in Eq. (448) and their respective rates in Eq. (449) are given by

$$\boldsymbol{g}(\widehat{\boldsymbol{x}}) = \boldsymbol{\Phi}\widehat{\boldsymbol{x}} \Rightarrow \boldsymbol{x} \approx \widetilde{\boldsymbol{x}} = \boldsymbol{x}_{ref} + \boldsymbol{\Phi}\widehat{\boldsymbol{x}}, \qquad \frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} \approx \frac{\mathrm{d}\widetilde{\boldsymbol{x}}}{\mathrm{d}t} = \boldsymbol{\Phi}\frac{\mathrm{d}\widehat{\boldsymbol{x}}}{\mathrm{d}t}. \tag{450}$$

As opposed to the nonlinear manifold Eq. (449), the tangent space of the (linear) solution manifold is constant, i.e., $\boldsymbol{J}_g = \boldsymbol{\Phi}$; see [307]. We also mention the example of eigenmodes as a classical choice for basis vectors of reduced subspaces [309].

The authors of [47] defined a *residual function* $\widetilde{\boldsymbol{r}} : \mathbb{R}^{n_s} \times \mathbb{R}^{n_s} \times \mathbb{R} \times \mathcal{D} \to \mathbb{R}^{N_s}$ in the reduced set of coordinates by rewriting the governing ODE Eq. (445) and substituting the approximation of the state Eq. (448) and its rate Eq. (449):[286]

$$\widetilde{\boldsymbol{r}}(\dot{\widehat{\boldsymbol{x}}}, \widehat{\boldsymbol{x}}, t; \boldsymbol{\mu}) = \boldsymbol{J}_g(\widehat{\boldsymbol{x}})\dot{\widehat{\boldsymbol{x}}} - \boldsymbol{f}(\boldsymbol{x}_{ref} + \boldsymbol{g}(\widehat{\boldsymbol{x}}), t; \boldsymbol{\mu}) \tag{451}$$

As $N_s > n_s$, the system of equations $\widetilde{\boldsymbol{r}}(\dot{\widehat{\boldsymbol{x}}}, \widehat{\boldsymbol{x}}, t; \boldsymbol{\mu}) = \boldsymbol{0}$ is over-determined and no solution exists in general. For this reason, a least-squares solution that minimized the square of the residual's Euclidean norm, which was denoted by $\|(\cdot)\|_2$, was sought instead [47]:

$$\dot{\widehat{\boldsymbol{x}}} = \underset{\widehat{\boldsymbol{v}} \in \mathbb{R}^{n_s}}{\arg\min} \left\| \widetilde{\boldsymbol{r}}(\widehat{\boldsymbol{v}}, \widehat{\boldsymbol{x}}, t; \boldsymbol{\mu}) \right\|_2^2. \tag{452}$$

Requiring the derivative of the (squared) residual to vanish, we obtain the following set of equations,[287]

$$\frac{\partial}{\partial \dot{\widehat{\boldsymbol{x}}}} \left\| \widetilde{\boldsymbol{r}}(\dot{\widehat{\boldsymbol{x}}}, \widehat{\boldsymbol{x}}, t; \boldsymbol{\mu}) \right\|_2^2 = 2\boldsymbol{J}_g^T \left(\boldsymbol{J}_g\dot{\widehat{\boldsymbol{x}}} - \boldsymbol{f}(\boldsymbol{x}_{ref} + \boldsymbol{g}(\widehat{\boldsymbol{x}}), t; \boldsymbol{\mu}) \right) = \boldsymbol{0}, \tag{453}$$

---

[285] In solid mechanics, we typically deal with second-order ODEs, which can be converted into a system of first-order ODEs by including velocities in the state space. For this reason, we prefer to use the term 'rate' rather than 'velocity' of $\boldsymbol{x}$ in what follows.

[286] The tilde above the symbol for the residual function $\widetilde{\boldsymbol{r}}$ in Eq. (451) was used to indicate an approximation, consistent with the use of the tilde in the approximation of the state in Eq. (448), i.e., $\boldsymbol{x} \approx \widetilde{\boldsymbol{x}}$.

[287] As the authors of [47] omitted a step-by-step derivation, we introduce it here for the sake of completeness.

which can be rearranged for the rate of the reduced vector of generalized coordinates as:

$$\dot{\widehat{\boldsymbol{x}}} = \boldsymbol{J}_g^\dagger(\widehat{\boldsymbol{x}})\boldsymbol{f}(\boldsymbol{x}_{ref} + \boldsymbol{g}(\widehat{\boldsymbol{x}}), t; \boldsymbol{\mu}), \qquad \widehat{\boldsymbol{x}}(0; \boldsymbol{\mu}) = \widehat{\boldsymbol{x}}_0(\boldsymbol{\mu}), \qquad \boldsymbol{J}_g^\dagger = (\boldsymbol{J}_g^T \boldsymbol{J}_g)^{-1} \boldsymbol{J}_g^T. \tag{454}$$

The above system of ODEs, in which $\boldsymbol{J}_g^\dagger$ denotes the *Moore-Penrose inverse* of the Jacobian $\boldsymbol{J}_g \in \mathbb{R}^{N_s \times n_s}$, which is also referred to as *pseudo-inverse*, constitutes the ROM corresponding to the FOM governed by Eq. (445). Eq. (453) reveals that the minimization problem is equivalent to a projection of the full residual onto the low-dimensional subspace $\mathcal{S}$ by means of the Jacobian $\boldsymbol{J}_g$. Note that the rate of the generalized coordinates $\dot{\widehat{\boldsymbol{x}}}$ lies in the same tangent space, which is spanned by $\boldsymbol{J}_g$. Therefore, the authors of [47] described the projection Eq. (453) as *Galerkin projection* and the ROM Eq. (454) as *nonlinear manifold (NM) Galerkin ROM*. To construct the solutions, suitable time-integration schemes needed to be applied to the semi-discrete ROM Eq. (454).

An alternative approach was also presented in [47] for the construction of ROMs, in which time-discretization was performed prior to the projection onto the low-dimensional solution subspace. For this purpose a uniform time-discretization with a step size Δt was assumed; the solution at time $t_n = t_{n-1} + \Delta t = n\Delta t$ was denoted by $\boldsymbol{x}_n = \boldsymbol{x}(t_n; \boldsymbol{\mu})$. To develop the method for implicit integration schemes, the *backward Euler* method was chosen as example [47]:

$$\boldsymbol{x}_n - \boldsymbol{x}_{n-1} = \Delta t \boldsymbol{f}_n, \tag{455}$$

The above integration rule implies that span of rate $\boldsymbol{f}_n$, which is given by evaluating (or by taking the 'snapshot' of) the nonlinear function $\boldsymbol{f}$, is included in the span of the state ('solution snapshot') $\boldsymbol{x}_n$:

$$\mathrm{span}\,\{\boldsymbol{f}_n\} \subseteq \mathrm{span}\,\{\boldsymbol{x}_{n-1}, \boldsymbol{x}_n\} \qquad \rightarrow \qquad \mathrm{span}\,\{\boldsymbol{f}_1, \ldots, \boldsymbol{f}_{N_t}\} \subseteq \mathrm{span}\,\{\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{N_t}\}. \tag{456}$$

For the backward Euler scheme Eq. (455), a residual function was defined in [47] as the difference

$$\widetilde{\boldsymbol{r}}_{BE}^n(\widehat{\boldsymbol{x}}_n; \widehat{\boldsymbol{x}}_{n-1}, \boldsymbol{\mu}) = \boldsymbol{g}(\widehat{\boldsymbol{x}}_n) - \boldsymbol{g}(\widehat{\boldsymbol{x}}_{n-1}) - \Delta t \boldsymbol{f}(\boldsymbol{x}_{ref} + \boldsymbol{g}(\widehat{\boldsymbol{x}}_n), t_n; \boldsymbol{\mu}). \tag{457}$$

Just as in the time-continuous domain, the system of equations $\widetilde{\boldsymbol{r}}_{BE}^n(\widehat{\boldsymbol{x}}_n; \widehat{\boldsymbol{x}}_{n-1}, \boldsymbol{\mu}) = \boldsymbol{0}$ is over-determined and, hence, was reformulated as a least-squares problem for the generalized coordinates $\widehat{\boldsymbol{x}}_n$:

$$\widehat{\boldsymbol{x}}_n = \frac{1}{2} \underset{\widehat{\boldsymbol{v}} \in \mathbb{R}^{n_s}}{\arg\min} \left\| \widetilde{\boldsymbol{r}}_{BE}^n(\widehat{\boldsymbol{v}}; \widehat{\boldsymbol{x}}_{n-1}, \boldsymbol{\mu}) \right\|_2^2. \tag{458}$$

To solve the least-squares problem, the Gauss-Newton method with starting point $\widehat{\boldsymbol{x}}_{n-1}$ was applied, i.e., the residual Eq. (457) was expanded into a first-order Taylor polynomial in the generalized coordinates [47]:

$$\widetilde{\boldsymbol{r}}_{BE}^n(\widehat{\boldsymbol{x}}_n; \widehat{\boldsymbol{x}}_{n-1}, \boldsymbol{\mu}) \approx \widetilde{\boldsymbol{r}}_{BE}^n(\widehat{\boldsymbol{x}}_{n-1}; \widehat{\boldsymbol{x}}_{n-1}, \boldsymbol{\mu}) + \left.\frac{\partial \widetilde{\boldsymbol{r}}_{BE}^n}{\partial \widehat{\boldsymbol{x}}_n}\right|_{\widehat{\boldsymbol{x}}_n = \widehat{\boldsymbol{x}}_{n-1}} (\widehat{\boldsymbol{x}}_n - \widehat{\boldsymbol{x}}_{n-1})$$

$$= -\Delta t \boldsymbol{f}_{n-1} + (\boldsymbol{J}_g(\widehat{\boldsymbol{x}}_{n-1}) - \Delta t \boldsymbol{J}_f(\boldsymbol{x}_{ref} + \boldsymbol{g}(\widehat{\boldsymbol{x}}_{n-1}))\boldsymbol{J}_g(\widehat{\boldsymbol{x}}_{n-1})) (\widehat{\boldsymbol{x}}_n - \widehat{\boldsymbol{x}}_{n-1})$$

$$= -\Delta t \boldsymbol{f}_{n-1} + (\boldsymbol{I} - \Delta t \boldsymbol{J}_f(\boldsymbol{x}_{ref} + \boldsymbol{g}(\widehat{\boldsymbol{x}}_{n-1}))) \boldsymbol{J}_g(\widehat{\boldsymbol{x}}_{n-1}) (\widehat{\boldsymbol{x}}_n - \widehat{\boldsymbol{x}}_{n-1}). \tag{459}$$

Unlike the time-continuous case, the Gauss-Newton method results in a projection involving not only the Jacobian $\boldsymbol{J}_g$ defined in Eq. (449), but also the Jacobian of the nonlinear function $\boldsymbol{f}$, i.e., $\boldsymbol{J}_f = \partial \boldsymbol{f} / \partial \boldsymbol{x}_n$. Therefore, the resulting reduced set of algebraic equations obtained upon a projection of the fully discrete FOM was referred to as *nonlinear manifold least-squares Petrov-Galerkin (NM-LSPG) ROM* [47].

12.4.3 Autoencoder

Within the NS-ROM approach, it was proposed in [47] to construct $\boldsymbol{g}$ by training an *autoencoder*, i.e., a neural network that reproduces its input vector:

$$\boldsymbol{x} \approx \widetilde{\boldsymbol{x}} = \mathbf{ae}(\boldsymbol{x}) = \mathbf{de}(\mathbf{en}(\boldsymbol{x})). \tag{460}$$

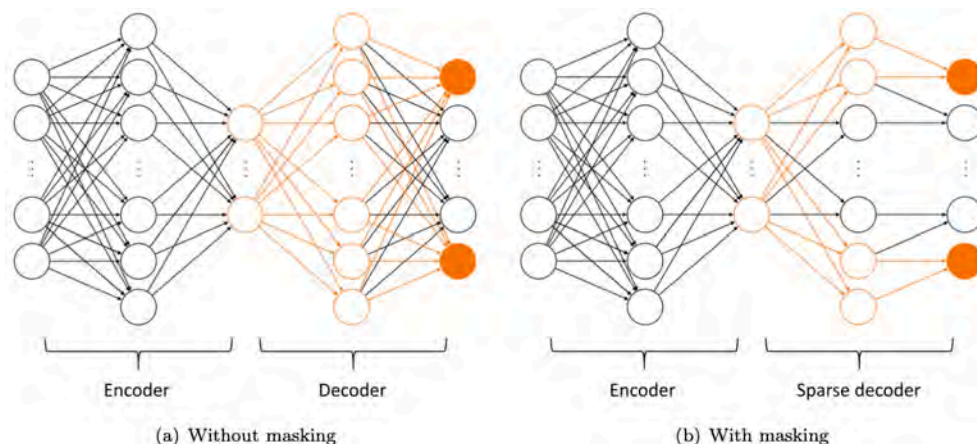(a) Without masking   (b) With masking

Figure 122: *Dense vs. shallow decoder networks* (Section 12.4.3). Contributing neurons (orange "nodes") and connections (orange "edges") lie in the "active" paths arriving at the selected outputs (solid orange "nodes") from the decoder's inputs. In dense networks as the one in (a), each neuron in a layer is connected to all other neurons in both the preceeding layer (if it exists) and in the succeeding layer (if it exists). Fully-connected networks are characterized by dense weight matrices, see Section 4.4. In sparse networks as the decoder in (b), several connections among successive layers are dropped, resulting in sparsely populated weight matrices. (Figure reproduced with permission of the authors.)

As the above relation reveals, autoencoders are typically composed from two parts, i.e., $\mathbf{ae}(\boldsymbol{x}) = (\mathbf{de} \circ \mathbf{en})(\boldsymbol{x})$: The *encoder* 'codes' inputs $\boldsymbol{x}$ into a so-called *latent state* $\boldsymbol{h} = \mathbf{en}(\boldsymbol{x})$ (not to be confused with hidden states of neural networks). The *decoder* then reconstructs ('decodes') an approximation of the input from the latent state, i.e., $\widetilde{\boldsymbol{x}} = \mathbf{de}(\boldsymbol{h})$. Both encoder and decoder can be represented by neural networks, e.g., feed-forward networks as done in [47]. As the authors of [78] noted, an autoencoder is "not especially useful" if it exactly learns the identity mapping for all possible inputs $\boldsymbol{x}$. Instead, autoencoders are typically restricted in some way, e.g., by reducing the dimensionality of the latent space as compared to the dimension of inputs, i.e., $\dim(\boldsymbol{h}) < \dim(\boldsymbol{x})$. The restriction forces an autoencoder to focus on those aspects of the input, or input 'features', which are essential for the reconstruction of the input. The encoder network of such *undercomplete autoencoder*[288] performs a *dimensionality reduction*, which is exactly the aim of projection-based methods for constructing ROMs. Using nonlinear encoder/decoder networks, undercomplete autoencoders can be trained to represent low-dimensional subspaces of solutions to high-dimensional dynamical systems governed by Eq. (445).[289] In particular, the decoder network represents the nonlinear manifold $\mathcal{S}$, which described by the function $\boldsymbol{g}$ that maps the generalized coordinates of the ROM $\widehat{\boldsymbol{x}}$ onto the corresponding element $\widetilde{\boldsymbol{x}}$ of FOM's solution space. Accordingly, the generalized coordinates are identified with the autoencoder's latent state, i.e., $\widehat{\boldsymbol{x}} = \boldsymbol{h}$; the encoder network represent the inverse mapping $\boldsymbol{g}^{-1}$, which "captures the most salient features" of the FOM.

The input data, which was formed by the snapshots of solutions $\boldsymbol{X} = [\boldsymbol{x}_1, \dots, \boldsymbol{x}_m]$, where m denoted the number of snapshots, was normalized when training the network [47]. The shift by the referential state $\boldsymbol{x}_{ref}$ and scaling by $\boldsymbol{x}_{scale}$, i.e.,

$$\boldsymbol{x}_{normal} = \boldsymbol{x}_{scale} \odot (\boldsymbol{x} - \boldsymbol{x}_{ref}) \in \mathbb{R}^{N_s}, \tag{461}$$

is such that each component of the normalized input ranges from $[-1, 1]$ or $[0, 1]$.[290] The encoder maps a

---

[288] See [78], Chapter 14, p.493.

[289] In fact, linear decoder networks combined with MSE-loss are equivalent to the *Principal Components Analysis* (PCA) (see [78], p.494), which, in turn, is equivalent to the discrete variant of POD by means of singular value decomposition (see Remark 12.2).

[290] The authors of [47] did not provide further details.

FOM's (normalized) solution snapshot onto a corresponding low-dimensional latent state $\widehat{\boldsymbol{x}}$, i.e., vector of generalized coordinates:

$$\widehat{\boldsymbol{x}} = \boldsymbol{g}^{-1}(\boldsymbol{x}) = \mathbf{en}(\boldsymbol{x}_{normal}) = \mathbf{en}(\boldsymbol{x}_{scale} \odot (\boldsymbol{x} - \boldsymbol{x}_{ref})) \in \mathbb{R}^{n_s}. \tag{462}$$

The decoder reconstructs the high-dimensional solution $\widetilde{\boldsymbol{x}}$ from the low-dimensional state $\widehat{\boldsymbol{x}}$ inverting the normalization Eq. (461):

$$\widetilde{\boldsymbol{x}} = \boldsymbol{g}(\widehat{\boldsymbol{x}}) = \boldsymbol{x}_{ref} + \mathbf{de}(\widehat{\boldsymbol{x}}) \oslash \boldsymbol{x}_{scale} \in \mathbb{R}^{N_s}, \tag{463}$$

where the operator $\oslash$ denotes the element-wise division. The composition of encoder and decoder give the autoencoder, i.e.,

$$\widetilde{\boldsymbol{x}} = \mathbf{ae}(\boldsymbol{x}) = \boldsymbol{x}_{ref} + \mathbf{de}(\mathbf{en}(\boldsymbol{x}_{scale} \odot (\boldsymbol{x} - \boldsymbol{x}_{ref}))) \oslash \boldsymbol{x}_{scale}, \tag{464}$$

which is trained in an *unsupervised* way to (approximately) reproduce states $\widetilde{\boldsymbol{X}} = [\widetilde{\boldsymbol{x}}_1, \ldots, \widetilde{\boldsymbol{x}}_m]$ from input snapshots $\boldsymbol{X}$. To train autoencoders, the *mean squared error* (see Sec. 5.1.1) is the natural choice for the loss function, i.e., $J = \|\boldsymbol{X} - \widetilde{\boldsymbol{X}}\|_F/2 \to \min$, where the Frobenius norm of matrices is used.

Both the encoder and the decoder are feed-forward neural networks with a single hidden layer. As activation function, it was proposed in [47] to use the sigmoid function (see Figure 30 and Section 13.3.1) or the swish function (see Figure 139). It remains unspecified, however, which of the two were used in the numerical examples in [47]. Though the representational capacity of neural networks increases with depth, which is what 'deep learning' is all about, the authors of [47] deliberately used shallow networks to minimize the (computational) complexity of computing the decoder's Jacobian $\boldsymbol{J}_g$. To further improve computational efficiency, they proposed to use a "masked/sparse decoder", in which the weight matrix that mapped the hidden state onto the output $\widetilde{\boldsymbol{x}}$ was sparse. As opposed to a fully populated weight matrix of fully-connected networks (Figure 122 (a)), in which each component of the output vector depends on all components of the hidden state, the outputs of a sparse decoder (Figure 122 (b)) only depend on the selected components of the hidden state.

(a) Mask matrix for 1D Burgers equation   (b) Mask matrix for 2D Burgers equation
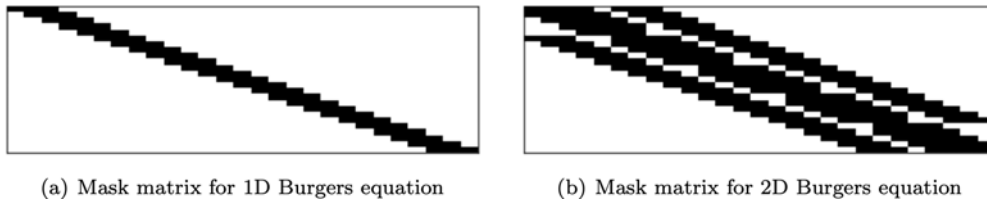
Figure 123: *Sparsity masks* (Section 12.4.3) used to realize sparse decoders in one- and two-dimensional problems. The structure of the respective binary-valued mask matrices $\boldsymbol{S}$ is inspired by grid-points required in the finite-difference approximation of the Laplace operator in one and two dimensions, respectively. (Figure reproduced with permission of the authors.)

Using our notation for feedforward networks and activation functions introduced in Section 4.4, the encoder network has the following structure:

$$\widehat{\boldsymbol{x}} = \mathbf{en}(\boldsymbol{x}_{normal}) := \boldsymbol{W}_{\mathrm{en}}^{(2)} \boldsymbol{y}^{(1)} + \boldsymbol{b}_{\mathrm{en}}^{(2)}, \qquad \boldsymbol{y}^{(1)} = \mathfrak{s}(\boldsymbol{z}^{(1)}),$$

$$\boldsymbol{z}^{(1)} = \boldsymbol{W}_{\mathrm{en}}^{(1)} \boldsymbol{y}^{(0)} + \boldsymbol{b}_{\mathrm{en}}^{(1)}, \qquad \boldsymbol{y}^{(0)} = \boldsymbol{x}_{normal}, \tag{465}$$

where $\boldsymbol{W}_{\mathrm{de}}^{(i)}$, $i = 1, 2$ are dense matrices. The masked decoder, on the contrary, is characterized by sparse connections between the hidden layer and the output layer, which is realized as element-wise multiplication of a dense weight matrix $\boldsymbol{W}_{\mathrm{de}}^{(2)}$ with a binary-valued "mask matrix" $\boldsymbol{S}$ reflecting the connectivity among the two layers:

$$\widetilde{\boldsymbol{x}} \odot \boldsymbol{x}_{scale} - \boldsymbol{x}_{ref} = \mathbf{de}(\widehat{\boldsymbol{x}}) := \boldsymbol{S} \odot \boldsymbol{W}_{\mathrm{de}}^{(2)} \boldsymbol{y}^{(1)} + \boldsymbol{b}_{\mathrm{de}}^{(2)}, \qquad \boldsymbol{y}^{(1)} = \mathfrak{s}(\boldsymbol{z}^{(1)}),$$

$$\boldsymbol{z}^{(1)} = \boldsymbol{W}_{\mathrm{de}}^{(1)} \boldsymbol{y}^{(0)} + \boldsymbol{b}_{\mathrm{de}}^{(1)}, \qquad \boldsymbol{y}^{(0)} = \widehat{\boldsymbol{x}}. \tag{466}$$

The structure of the sparsity mask $\boldsymbol{S}$, in turn, is inspired by the pattern ("stencil") of grid-points involved in a finite-difference approximation of the Laplace operator, see Figure 123 for the one- and two-dimensional cases.

12.4.4 Hyper-reduction

Irrespective of how small the dimensionality of the ROM's solution subspace $\mathcal{S}$ is, we cannot expect a reduction in computational efforts in nonlinear problems. Both the time-continuous NM-Galerkin ROM in Eq. (454) and the time-discrete NM-LSPG ROM in Eq. (458) require repeated evaluations of the non-linear function $\boldsymbol{f}$ and, in case implicit time-integration schemes, also its Jacobian $\boldsymbol{J}_f$, whose computational complexity is determined by the FOM's dimensionality. The term *hyper-reduction* subsumes techniques in model-order reduction to overcome the necessity for evaluations that scale with the FOM's size. Among hyper-reduction techniques, the *Discrete Empirical Interpolation Method* (DEIM) [310] and *Gauss-Newton with Approximated Tensors* (GNAT) [311] have gained significant attention in recent years.

In [47], a variant of GNAT relying on solution snapshots for the approximation of the nonlinear residual term $\widetilde{\boldsymbol{r}}$ was used, and therefore the reason to extend the GNAT acronym with "SNS" for *'solution-based subspace'* (SNS), i.e., *GNAT-SNS*, see [312]. The idea of DEIM, GNAT and their SNS variants takes up the leitmotif of projection-based methods: The approximation of the full-order residual $\widetilde{\boldsymbol{r}}$ is in turn linearly interpolated by a low-dimensional vector $\widehat{\boldsymbol{r}}$ and an appropriate set of basis vectors $\boldsymbol{\phi}_{r,i} \in \mathbb{R}^{N_s}$, $i = 1, \ldots, n_r$:

$$\widetilde{\boldsymbol{r}} \approx \boldsymbol{\Phi}_r \widehat{\boldsymbol{r}}, \qquad \boldsymbol{\Phi}_r = [\boldsymbol{\phi}_{r,1}, \ldots, \boldsymbol{\phi}_{r,n_r}] \in \mathbb{R}^{N_s \times n_r}, \qquad n_s \leq n_r \ll N_s. \tag{467}$$

In DEIM methods, the residual $\widetilde{\boldsymbol{r}}$ of the time-continuous problem in Eq. (451) is approximated, whereas $\widetilde{\boldsymbol{r}}$ it to be replaced by $\widetilde{\boldsymbol{r}}_{BE}^n$ in the above relation when applying GNAT, which builds upon Petrov-Galerkin ROMs as in Eq. (459). Both DEIM and GNAT variants use *gappy POD* to determine the basis $\boldsymbol{\Phi}_r$ for the interpolation of the nonlinear residual. Gappy POD originates in a method for image reconstruction proposed [313] under the name of a Karhunen-Loève procedure, in which images were reconstructed from individual pixels, i.e., from *gappy data*. In the present context of MOR, gappy POD aims at reconstructing the full-order residual $\widetilde{\boldsymbol{r}}$ from a small, n_r-dimensional subset of its components.

The matrix $\boldsymbol{\Phi}_r$ was computed by a singular value decomposition (SVD) on the snapshots of data. Unlike original DEIM and GNAT methods, their SNS variants did not use snapshots of the nonlinear residual $\widetilde{\boldsymbol{r}}$ and $\widetilde{\boldsymbol{r}}_{BE}^n$, respectively, but SVD was performed on solution snapshots $\boldsymbol{X}$ instead. The use of solution snapshots was motivated by the fact that the span of snapshots of the nonlinear term $\boldsymbol{f}$ was included in the span of of solution snapshots for conventional time-integration schemes, see Eq. (456). The vector of *"generalized coordinates of the nonlinear residual term"* minimizes the square of the Euclidean distance of selected components of full-order residual $\widetilde{\boldsymbol{r}}$ and respective components of its reconstruction:

$$\widehat{\boldsymbol{r}} = \underset{\widehat{\boldsymbol{v}} \in \mathbb{R}^{n_r}}{\arg\min} \frac{1}{2} \left\| \boldsymbol{Z}^T \left(\widetilde{\boldsymbol{r}} - \boldsymbol{\Phi}_r \widehat{\boldsymbol{v}} \right) \right\|_2^2, \quad \boldsymbol{Z}^T = \left[\boldsymbol{e}_{p_1}, \ldots, \boldsymbol{e}_{p_{n_z}} \right]^T \in \mathbb{R}^{n_z \times N_s}, \quad n_s \leq n_r \leq n_z \ll N_s. \tag{468}$$

The matrix $\boldsymbol{Z}^T$, which was referred to as *sampling matrix*, extracts a set of components from the full-order vectors $\boldsymbol{v} \in \mathbb{R}^{N_s}$. For this purpose, the sampling matrix was built from unit vectors $\boldsymbol{e}_{p_i} \in \mathbb{R}^{N_s}$, having the value one at the p_i-th component, which corresponded to the component of $\boldsymbol{v}$ to be selected ('sampled'), and the value zero otherwise. The components selected by $\boldsymbol{Z}^T$ are equivalently described by the (ordered) set of *sampling indices* $\mathcal{I} = \{p_1, \ldots, p_{n_z}\}$, which are represented by the vector $\boldsymbol{i} = [p_1, \ldots, p_{n_z}]^T \in \mathbb{R}^{n_z}$. As the number of components of $\widetilde{\boldsymbol{r}}$ used for its reconstruction may be larger than the dimensionality of the reduced-order vector $\widehat{\boldsymbol{r}}$, i.e., $n_z \geq n_r$, Eq. (468) generally constitutes a least-squares problem, the solution of which follows as

$$\widehat{\boldsymbol{r}} = (\boldsymbol{Z}^T \boldsymbol{\Phi}_r)^{\dagger} \boldsymbol{Z}^T \widetilde{\boldsymbol{r}}. \tag{469}$$

Substituting the above result in Eq. (467), the FOM's residual can be interpolated using an *oblique projection matrix* $\mathcal{P}$ as

$$\widetilde{r} \approx \mathcal{P}\widetilde{r}, \qquad \mathcal{P} = \Phi_r(Z^T\Phi_r)^\dagger Z^T \in \mathbb{R}^{N_s \times N_s}. \tag{470}$$

The above representation in terms of the projection matrix $\mathcal{P}$ somewhat hides the main point of hyperreduction. In fact, we do not apply $\mathcal{P}$ to the full-order residual $\widetilde{r}$, which would be tautological. Unrolling the definition of $\mathcal{P}$, we note that $Z^T\widetilde{r} \in \mathbb{R}^{n_z}$ is a vector containing only a small subset of components of the full-order residual $\widetilde{r}$. In other words, to evaluate the approximation in Eq. (470) only $n_z \ll N_s$ of components of $\widetilde{r}$ need to be computed, i.e., the computational cost no longer scales with the FOM's dimensionality.

Several methods have been proposed to efficiently construct a suitable set of sampling indices $\mathcal{I}$, see, e.g., [310, 311, 314]. These methods share the property of being *greedy algorithms*, i.e., algorithms that sequentially ("inductively") create optimal sampling indices using some suitable metric. For instance, the authors of [310] selected the j-th index corresponding to the component of the gappy reconstruction of the j-th POD-mode $\widehat{\phi}_{r,j}$ showing the largest error compared to the original mode $\phi_{r,j}$. For the reconstruction, the first $j-1$ POD-modes were used, i.e.,

$$\phi_{r,j} \approx \widetilde{\phi}_{r,j} = \Phi_r\widehat{\phi}_{r,j}, \qquad \Phi_r = [\phi_{r,1}, \ldots, \phi_{r,j-1}] \in \mathbb{R}^{N_s \times j-1}. \tag{471}$$

The reduced-order vector $\widehat{\phi}_{r,j} \in \mathbb{R}^{j-1}$ minimizes the square of the Euclidean distance between $j-1$ components (selected by $Z^T \in \mathbb{R}^{j \times N_s}$) of the j-th mode $\phi_{r,j}$ and its gappy reconstruction $\widetilde{\phi}_{r,j}$:

$$\widehat{\phi}_{r,j} = \operatorname*{arg\,min}_{\widehat{v}\in\mathbb{R}^{j-1}} \frac{1}{2}\left\| Z^T\left(\phi_{r,j} - \Phi_r\widehat{v}\right)\right\|_2^2. \tag{472}$$

The key idea of the greedy algorithm is to select additional indices to minimize the error of the gappy reconstruction. Therefore, the component of the reconstructed mode that differs most (in terms of magnitude) from the original mode defines the j-th sampling index:

$$p_j = \operatorname*{arg\,max}_{i\in\{1,\ldots,N_s\}\setminus\mathcal{I}} \left|(\phi_{r,j} - \Phi_r(Z^T\Phi_r)^\dagger Z^T\phi_{r,j})_i\right|. \tag{473}$$

A pseudocode representation of the greedy approach for selecting sampling indices is given in Algorithm 8. To start the algorithm, the first sampling index is chosen according to the largest (in terms of magnitude) component of the first POD-mode, i.e., $p_1 = \operatorname{arg\,max}_{i\in\{1,\ldots,N_s\}} |(\phi_{r,1})_i|$.

The authors of [47] substituted the residual vector $\widetilde{r}$ by its gappy reconstruction, i.e., $\mathcal{P}\widetilde{r}$ in the minimization problem in Eq. (452). Unlike the original Galerkin ROM in Eq. (452), the rate of the reduced vector of generalized coordinate does not minimize the (square of the) FOM's residual $\widetilde{r}$, but the corresponding reduced residual $\widehat{r}$, which are related by $\widetilde{r} = \Phi_r\widehat{r}$:

$$\dot{\widehat{x}} = \operatorname*{arg\,min}_{\widehat{v}\in\mathbb{R}^{n_s}} \|\widehat{r}(\widehat{v},\widehat{x},t;\mu)\|_2^2 = \operatorname*{arg\,min}_{\widehat{v}\in\mathbb{R}^{n_s}} \left\|(Z^T\Phi_r)^\dagger Z^T\widetilde{r}(\widehat{v},\widehat{x},t;\mu)\right\|_2^2. \tag{474}$$

From the above minimization problem, the ROM's ODEs were determined in [47] by taking the derivative with respect to the reduced vector of generalized velocities $\widehat{v}$, which was evaluated at $\widehat{v} = \dot{\widehat{x}}$, i.e.,[291]

$$2\widehat{r}(\dot{\widehat{x}},\widehat{x},t;\mu)\left.\frac{\partial\widehat{r}(\widehat{v},\widehat{x},t;\mu)}{\partial\widehat{v}}\right|_{\widehat{v}=\dot{\widehat{x}}} = 2(Z^T\Phi_r)^\dagger Z^T\widetilde{r}\left\{(Z^T\Phi_r)^\dagger Z^T\frac{\partial\widetilde{r}}{\partial\dot{\widehat{x}}}\right\}$$

$$= 2(Z^T\Phi_r)^\dagger Z^T\left(J_g\dot{\widehat{x}} - f(x_{ref} + g(\widehat{x}),t;\mu)\right)(Z^T\Phi_r)^\dagger Z^T J_g$$

$$= 2\left((Z^T\Phi_r)^\dagger Z^T J_g\right)^T(Z^T\Phi_r)^\dagger Z^T\left(J_g\dot{\widehat{x}} - f(x_{ref} + g(\widehat{x}),t;\mu)\right) = 0, \tag{475}$$

---

[291] Note again that the step-by-step derivations of the ODEs in Eq. (477) were not given by the authors of [47], which is why we provide it in our review paper for the sake of clarity.

1 DEIM algorithm for construction of sampling indices
    Data: POD-basis $\{\phi_{r,1}, \ldots, \phi_{r,n_r}\}$, in descending order according to singular values
    Result: Set of sampling indices $\mathcal{I} = \{p_1, \ldots, p_{n_z}\}$
2 ▶ Initialization
3 Select first sampling index $p_1 = \arg\max_{i \in \{1, \ldots, N_s\}} |(\phi_{r,1})_i|$;
4 Set $\boldsymbol{\Phi}_r \leftarrow [\phi_{r,1}]$, $\boldsymbol{Z} \leftarrow [\boldsymbol{e}_{p_1}]$, $\boldsymbol{i} = [p_1]$, $\mathcal{I} = \{p_1\}$;
5 **for** $j = 2, \ldots, n_z$ **do**
6   ▶ Select j-th sampling index p_j
7   Solve $(\boldsymbol{Z}^T\boldsymbol{\Phi}_r)\widehat{\boldsymbol{v}} = \boldsymbol{Z}^T\phi_{r,j}$ for $\widehat{\boldsymbol{v}}$ (cf. Eq. (472)) ;
8   Set $p_j = \arg\max_{i \in \{1, \ldots, N_s\} \setminus \mathcal{I}} |(\phi_{r,j} - \boldsymbol{\Phi}_r\widehat{\boldsymbol{v}})_i|$;
9   Update $\boldsymbol{\Phi}_r \leftarrow \begin{bmatrix} \boldsymbol{\Phi}_r & \phi_{r,j} \end{bmatrix}$, $\boldsymbol{Z} \leftarrow \begin{bmatrix} \boldsymbol{Z} & \boldsymbol{e}_{p_j} \end{bmatrix}$, $\boldsymbol{i} \rightarrow \begin{bmatrix} \boldsymbol{i}^T & p_j \end{bmatrix}^T$, $\mathcal{I} \leftarrow \mathcal{I} \cup \{p_j\}$;
10 **end**

Algorithm 8: *DEIM algorithm for construction of sampling indices* (Section 12.4.4). Starting from the initial ($j = 1$) sampling index p_1, a sequence of indices $\mathcal{I} = \{p_1, \ldots, p_{n_z}\}$ is iteratively constructed in a *greedy* way: Given the first index p_1, subsequent indices are iteratively chosen according to the error in the gappy reconstruction of POD-modes. In particular, the j-th sampling index p_j corresponds to that component of the j-th POD mode, for which the gappy reconstruction by means of the $j - 1$ preceding modes shows the largest error.

where the definition of the residual vector $\widetilde{\boldsymbol{r}}$ has been introduced in Eq. (451). We therefore obtain the following linear system of equations for $\dot{\widehat{\boldsymbol{x}}}$,

$$\left((\boldsymbol{Z}^T\boldsymbol{\Phi}_r)^\dagger \boldsymbol{Z}^T\boldsymbol{J}_g\right)^T (\boldsymbol{Z}^T\boldsymbol{\Phi}_r)^\dagger \boldsymbol{Z}^T\boldsymbol{J}_g\dot{\widehat{\boldsymbol{x}}} = \left((\boldsymbol{Z}^T\boldsymbol{\Phi}_r)^\dagger \boldsymbol{Z}^T\boldsymbol{J}_g\right)^T (\boldsymbol{Z}^T\boldsymbol{\Phi}_r)^\dagger \boldsymbol{Z}^T\boldsymbol{f}(\boldsymbol{x}_{ref} + \boldsymbol{g}(\widehat{\boldsymbol{x}}), t; \boldsymbol{\mu}), \quad (476)$$

which, using the notion of the pseudo-inverse, is resolved for $\dot{\widehat{\boldsymbol{x}}}$ to give, complemented by proper initial conditions $\widehat{\boldsymbol{x}}_0(\boldsymbol{\mu})$, the governing systems of ODEs of the *hyper-reduced nonlinear manifold, least-squares-Galerkin* (NM-LS-Galerkin-HR) ROM [47]:

$$\dot{\widehat{\boldsymbol{x}}} = \left((\boldsymbol{Z}^T\boldsymbol{\Phi}_r)^\dagger \boldsymbol{Z}^T\boldsymbol{J}_g\right)^\dagger (\boldsymbol{Z}^T\boldsymbol{\Phi}_r)^\dagger \boldsymbol{Z}^T\boldsymbol{f}(\boldsymbol{x}_{ref} + \boldsymbol{g}(\widehat{\boldsymbol{x}}), t; \boldsymbol{\mu}), \qquad \widehat{\boldsymbol{x}}(0; \boldsymbol{\mu}) = \widehat{\boldsymbol{x}}_0(\boldsymbol{\mu}). \quad (477)$$

Remark 12.6. *Equivalent minimization problems.* Note the subtle difference between the minimization problems that govern the ROMs with and without hyper-reduction, see Eqs. (452) and (474), respectively: For the case without hyper-reduction, see Eq. (452), the minimum is sought for the approximate *full-dimensional* residual $\widetilde{\boldsymbol{r}}$. In the hyper-reduced variant Eq. (474), the authors of [47] aimed, however, at minimizing the projected residual $\widehat{\boldsymbol{r}}$, which was related to its full-dimensional counterpart by the residual basis matrix $\boldsymbol{\Phi}_r$, i.e., $\widetilde{\boldsymbol{r}} = \boldsymbol{\Phi}_r\widehat{\boldsymbol{r}}$. Using the full-order residual also in the hyper-reduced ROM translates into the following minimization problem:

$$\dot{\widehat{\boldsymbol{x}}} = \arg\min_{\widehat{\boldsymbol{v}} \in \mathbb{R}^{n_s}} \|\boldsymbol{\Phi}_r\widehat{\boldsymbol{r}}(\widehat{\boldsymbol{v}}, \widehat{\boldsymbol{x}}, t; \boldsymbol{\mu})\|_2^2 = \arg\min_{\widehat{\boldsymbol{v}} \in \mathbb{R}^{n_s}} \left\|\boldsymbol{\Phi}_r(\boldsymbol{Z}^T\boldsymbol{\Phi}_r)^\dagger \boldsymbol{Z}^T\widetilde{\boldsymbol{r}}(\widehat{\boldsymbol{v}}, \widehat{\boldsymbol{x}}, t; \boldsymbol{\mu})\right\|_2^2. \quad (478)$$

Repeating the steps of the derivation in Eq. (475) then gives

$$\dot{\widehat{\boldsymbol{x}}} = \left(\boldsymbol{\Phi}_r(\boldsymbol{Z}^T\boldsymbol{\Phi}_r)^\dagger \boldsymbol{Z}^T\boldsymbol{J}_g\right)^\dagger \boldsymbol{\Phi}_r(\boldsymbol{Z}^T\boldsymbol{\Phi}_r)^\dagger \boldsymbol{Z}^T\boldsymbol{f}(\boldsymbol{x}_{ref} + \boldsymbol{g}(\widehat{\boldsymbol{x}}), t; \boldsymbol{\mu})$$

$$= \left((\boldsymbol{Z}^T\boldsymbol{\Phi}_r)^\dagger \boldsymbol{Z}^T\boldsymbol{J}_g\right)^\dagger \boldsymbol{\Phi}_r^\dagger\boldsymbol{\Phi}_r(\boldsymbol{Z}^T\boldsymbol{\Phi}_r)^\dagger \boldsymbol{Z}^T\boldsymbol{f}(\boldsymbol{x}_{ref} + \boldsymbol{g}(\widehat{\boldsymbol{x}}), t; \boldsymbol{\mu}) \quad (479)$$

$$= \left((\boldsymbol{Z}^T\boldsymbol{\Phi}_r)^\dagger \boldsymbol{Z}^T\boldsymbol{J}_g\right)^\dagger (\boldsymbol{Z}^T\boldsymbol{\Phi}_r)^\dagger \boldsymbol{Z}^T\boldsymbol{f}(\boldsymbol{x}_{ref} + \boldsymbol{g}(\widehat{\boldsymbol{x}}), t; \boldsymbol{\mu}),$$

i.e., using the identity $\mathbf{\Phi}_r^\dagger \mathbf{\Phi}_r = \mathbf{I}_{n_r}$, we recover exactly the same result as in the hyper-reduced case in Eq. (477). The only requirement is that the residual basis vectors $\Phi_{r,j}$, $j = 1, \ldots, n_r$ need to be linearly independent.   ∎

Remark 12.7. *Further reduction of the system operator?*   At first glance, the operator in the ROM's governing equations in Eq. (477) appears to be further reducible:

$$\left((\mathbf{Z}^T\mathbf{\Phi}_r)^\dagger \mathbf{Z}^T \mathbf{J}_g \right)^\dagger (\mathbf{Z}^T\mathbf{\Phi}_r)^\dagger = (\mathbf{Z}^T \mathbf{J}_g)^\dagger (\mathbf{Z}^T\mathbf{\Phi}_r)(\mathbf{Z}^T\mathbf{\Phi}_r)^\dagger. \tag{480}$$

Note that the product $(\mathbf{Z}^T\mathbf{\Phi}_r)(\mathbf{Z}^T\mathbf{\Phi}_r)^\dagger$ generally does not, however, evaluate to identity, since our particular definition of the pseudo inverse of $\mathbf{A} \in \mathbb{R}^{m \times n}$ is a *left inverse*, for which $\mathbf{A}^\dagger \mathbf{A} = \mathbf{I}_n$ holds, but $\mathbf{A}\mathbf{A}^\dagger \neq \mathbf{I}_m$.   ∎

We first consider linear subspace methods, for which the Jacobian $\mathbf{J}_g(\widehat{\boldsymbol{x}}) = \mathbf{\Phi}$ that spans the tangent space of the solution manifold is constant, see Eq. (450). Substituting $\mathbf{\Phi}\widehat{\boldsymbol{x}}$ for $\boldsymbol{g}(\widehat{\boldsymbol{x}})$ and $\mathbf{\Phi}$ for $\mathbf{J}_g$, the above set of ODEs in Eq. (477) governing the NM-Galerkin-HR ROM reduces to the corresponding *linear subspace* (LS-Galerkin-HR) ROM:

$$\dot{\widehat{\boldsymbol{x}}} = \left\{ (\mathbf{Z}^T\mathbf{\Phi}_r)^\dagger \mathbf{Z}^T \mathbf{\Phi} \right\}^\dagger (\mathbf{Z}^T\mathbf{\Phi}_r)^\dagger \mathbf{Z}^T \boldsymbol{f}(\boldsymbol{x}_{ref} + \mathbf{\Phi}\widehat{\boldsymbol{x}}, t; \boldsymbol{\mu}). \tag{481}$$

Note that, in linear subspace methods, the operator $\left\{ (\mathbf{Z}^T\mathbf{\Phi}_r)^\dagger \mathbf{Z}^T \mathbf{\Phi} \right\}^\dagger (\mathbf{Z}^T\mathbf{\Phi}_r)^\dagger$ is independent of the solution and, hence, *"can be pre-computed once for all"*. The products $\mathbf{Z}^T\mathbf{\Phi}_r$, $\mathbf{Z}^T\mathbf{\Phi}$ and $\mathbf{Z}^T\boldsymbol{f}$ need not be evaluated explicitly.
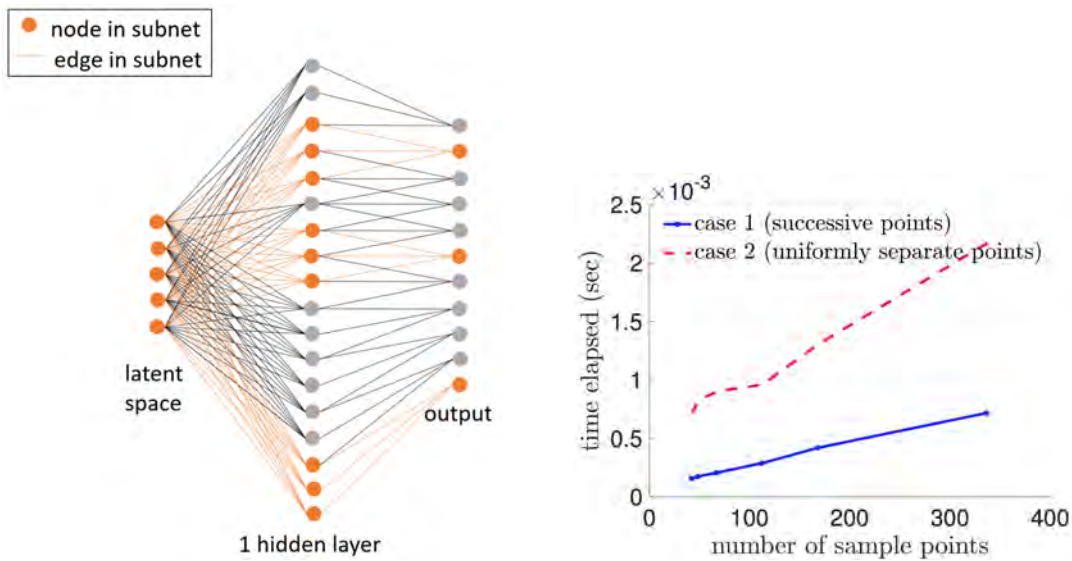
Figure 124: *Subnet construction* (Section 12.4.4). To reduce computational cost, a subnet representing the set of active paths, which comprise all neurons and connections needed for the evaluation of selected outputs (highlighted in orange), i.e., the reduced residual $\widehat{r}$, is constructed (left). The size of the hidden layer of the subnet depends on which output components of the decoder are needed for the reconstruction of the full-order residual. If the full-order residual is reconstructed from from successive outputs of the decoder, the number of neurons in the hidden layer involved in the evaluation becomes minimal due to the specific sparsity patterns proposed. Uniformly distributed components show the least overlap in terms of hidden-layer neurons required, which is why the subnet and therefore the computational cost in the hyper-reduction approach is maximal (right). (Figure reproduced with permission of the authors.)

Instead, only those rows of $\boldsymbol{\Phi}_r$, $\boldsymbol{\Phi}$ and $\boldsymbol{f}$ which are selected by the sampling matrix $\boldsymbol{Z}^T$ need to be extracted or computed when evaluating the above operator. In the context of MOR, pre-computations as, e.g., the above operator, but, more importantly, also the computationally demanding collection of full-order solutions and residual snapshots and subsequent SVDs, are attributed to the *offline phase* or *stage*, see, e.g., [307]. Ideally, the *online phase* only requires evaluations of quantities that scale with the dimensionality of the ROM.

By keeping track of which components of full-order solution $\widetilde{\boldsymbol{x}}$ are involved in the computation of selected components of the nonlinear term, i.e., $\boldsymbol{Z}^T \boldsymbol{f}$, the computational cost can be reduced even further. In other words, we need not reconstruct all components of the full-order solution $\widetilde{\boldsymbol{x}}$, but only those components, which are required for the evaluation of $\boldsymbol{Z}^T \boldsymbol{f}$. However, the number of components of $\widetilde{\boldsymbol{x}}$ that is needed for this purpose, which translates into the number of products of rows in $\boldsymbol{\Phi}$ with the reduced-order solution $\widehat{\boldsymbol{x}}$, is typically much larger than the number of sampling indices p_i, i.e., the cardinality of the set $\mathcal{I}$.

To explain this discrepancy, assume the full-order model to be obtained upon a finite-element discretization. Given some particular nodal point, all finite elements sharing the node contribute to the corresponding components of the nonlinear function $\boldsymbol{f}$. So we generally must evaluate several element when computing a single component of $\boldsymbol{f}$ corresponding to a single sampling index $p_i \in \mathcal{I}$, which, in turn, involves coordinates of all elements associated with the p_i-th degree of freedom.[292]

For nonlinear manifold methods, we cannot expect much improvement in computational efficiency by the hyper-reduction. As a matter of fact, the 'nonlinearity' becomes twofold if the reduced subspace is a nonlinear manifold: We do not only have to compute selected components of the nonlinear term $\boldsymbol{Z}^T \boldsymbol{f}$, we need to evaluate the nonlinear manifold $\boldsymbol{g}$. More importantly, from a computational point of view, also relevant rows of the Jacobian of the nonlinear manifold, which are extracted by $\boldsymbol{Z}^T \boldsymbol{J}_g(\widehat{\boldsymbol{x}})$, must be re-computed for every update of the (reduced-order) solution $\widehat{\boldsymbol{x}}$, see Eq. (477).

For Petrov-Galerkin-type variants of ROMs, hyper-reduction works in exactly the same way as with their Galerkin counterparts. The residual in the minimization problem in Eq. (458) is approximated by a gappy reconstruction, i.e.,

$$\widehat{\boldsymbol{x}}_n = \operatorname*{arg\,min}_{\widetilde{\boldsymbol{v}} \in \mathbb{R}^{n_s}} \frac{1}{2} \left\| (\boldsymbol{Z}^T \boldsymbol{\Phi}_r)^\dagger \boldsymbol{Z}^T \widetilde{\boldsymbol{r}}_{BE}^n(\widetilde{\boldsymbol{v}}; \widehat{\boldsymbol{x}}_{n-1}, \boldsymbol{\mu}) \right\|_2^2. \tag{482}$$

From a computational point of view, the same implications apply to Petrov-Galerkin ROMs as for Galerkin-type ROMs, which is why we focus on the latter in our review.

In the approach of [47], the nonlinear manifold $\boldsymbol{g}$ was represented by a feed-forward neural network, i.e., essentially the decoder of the proposed sparse autoencoder, see Eq. (463).

The computational cost of evaluating the decoder and its Jacobian scales with the number of parameters of the neural network. Both shallowness and sparsity of the decoder network already account for computational efficiency in regard of the number of parameters.

Additionally, the authors of [47] traced "active paths" when evaluating selected components of the decoder and its Jacobian of the hyper-reduced model. The set of active paths comprises all those connections and neurons of the decoder network which are involved in evaluations of its outputs. Figure 124 (left) highlights the active paths for the computations of the components of the reduced residual $\widehat{\boldsymbol{r}}$, from which the full residual vector $\widetilde{\boldsymbol{r}}$ is reconstructed within the hyper-reduction method, in orange.

Given all active paths, a subnet of the decoder network is constructed to only evaluate those components of the full-order state which are required to compute the hyper-reduced residual. The computational costs to

---

[292] The "Unassembled DEIM" (UDEIM) method proposed in [315] provides a partial remedy for that issue in the context of finite-element problems. In UDEIM, the algorithm is applied to the unassembled residual vector, i.e., the set of element residuals, which restricts the dependency among generalized coordinates to individual elements.

compute the residual and its Jacobian depends on the size of the subnet. As both input and output dimension are given, size translates into width of the (single) hidden layer. The size of the hidden layer, in turn, depends on the distribution of the sampling indices p_i, i.e., from which components the full residual $\hat{r}$ is reconstructed.

For the sparsity patterns assumed, successive output components show the largest overlap in terms of the number of neurons in the hidden layer involved in the evaluation, whereas the overlap is minimal in case of equally spaced outputs.

The cases of successive and equally distributed sampling indices constitute extremal cases, for which the computational time for the evaluation of both the residual and its Jacobian of the 2D-example (Section 12.4.5)) are illustrated as a function of the dimensionality of the reduced residual ("number of sampling points") in Figure 124 (right).
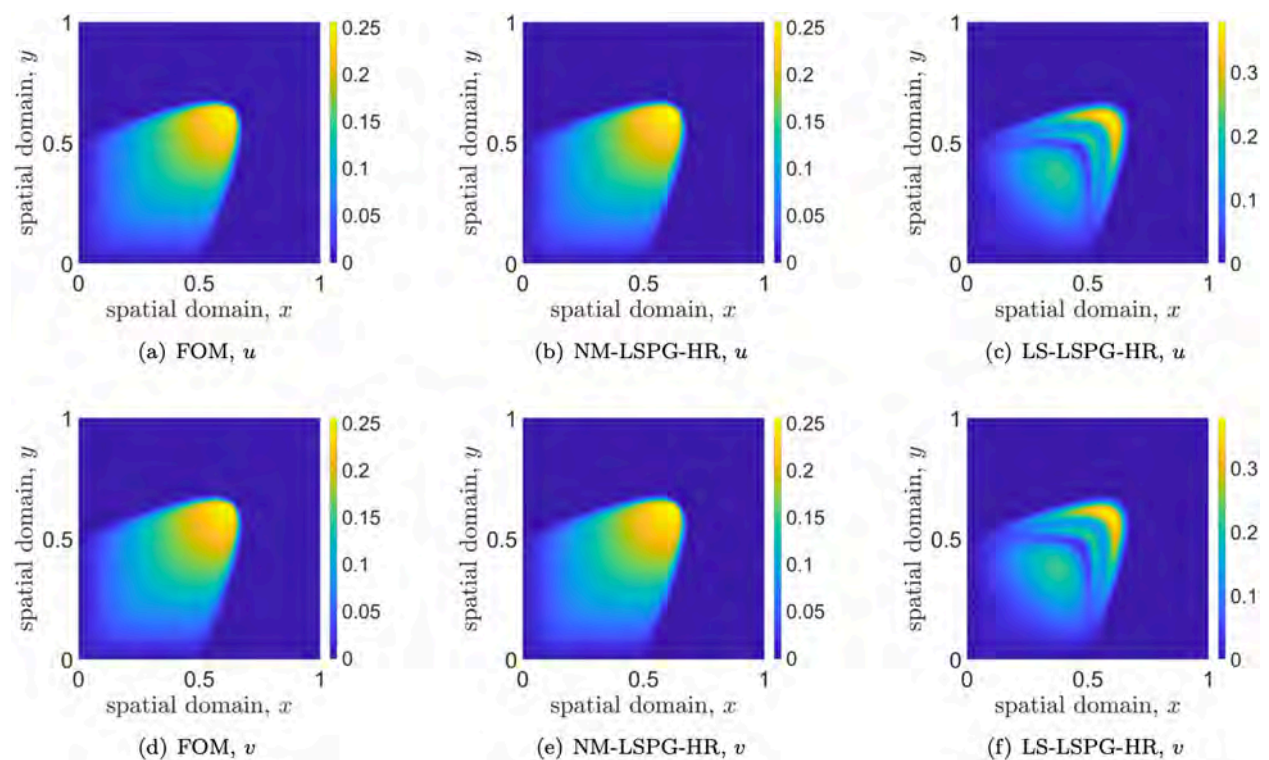
Figure 125: *2-D Burger's equation. Solution snapshots of full and reduced-order models* (Section 12.4.5). From left to right, the components u (top row) and v (bottom row) of the velocity field at time $t = 2$ are shown for the FOM, the hyper-reduced nonlinear-manifold-based ROM (NM-LSPG-HR) and the hyper-reduced linear-subspace-based ROM (LS-LSPG-HR). Both ROMs have a dimension of $n_s = 5$; with respect to hyper-reduction, the residual basis of the NM-LSPG-HR ROM has a dimension of $n_r = 55$ and $n_z = 58$ components of the full-order residual ("sampling indices") are used in the gappy reconstruction of the reduced residual. For the LS-LSPG-HR ROM, both the dimension of the residual basis and the number of sampling indices are $n_r = n_z = 59$. Due to the advection, a steep, shock wave-like gradient develops. While FOM and NM-LSPG-HR solutions are visually indistinguishable, the LS-LSPG-HR fails to reproduce the FOM's solution by a large margin (right column). Spurious oscillation patterns characteristic of advection-dominated problems (Brooks & Hughes (1982) [316]) occur. (Figure reproduced with permission of the authors.)

12.4.5 Numerical example: 2D Burger's equation

As a second example, consider now Burgers' equation in two (spatial) dimensions [47], instead of in one dimension as in Section 12.4.1. Additionally, *viscous behavior*, which manifests as the Laplace term on the right-hand side of the following equation, was included as opposed to the one-dimensional problem in Eq. (442):

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\text{grad } \boldsymbol{u}) \cdot \boldsymbol{u} = \frac{1}{Re} \text{ div} (\text{grad } \boldsymbol{u}), \qquad \boldsymbol{x} = (x, y)^T \in \Omega = [0, 1] \times [0, 1], \qquad t \in [0, 2]. \qquad (483)$$

The problem was solved on a square (unit) domain Ω, where homogeneous Dirichlet conditions for the velocity field $\boldsymbol{u}$ were assumed at all boundaries $\Gamma = \partial\Omega$:

$$\boldsymbol{u}(\boldsymbol{x}, t; \mu) = \boldsymbol{0} \quad \text{on} \quad \Gamma = \partial\Omega. \qquad (484)$$

An inhomogeneous flow profile under 45° was prescribed as initial conditions ($t = 0$),

$$u(\boldsymbol{x}, 0; \mu) = v(\boldsymbol{x}, 0; \mu) = \begin{cases} \mu \sin(2\pi x) \sin(2\pi y) & \text{if } \boldsymbol{x} \in [0, 0.5] \times [0, 0.5] \\ 0 & \text{otherwise} \end{cases}, \qquad (485)$$

where u, v denoted the Cartesian components of the velocity field, and $\mu \in \mathcal{D} = [0.9, 1.1]$ was a parameter describing the magnitude of the initial velocity field. Viscosity was governed by the Reynolds number Re, i.e., the problem was advection-dominated for high Reynolds number, whereas diffusion prevailed for low Reynolds number.
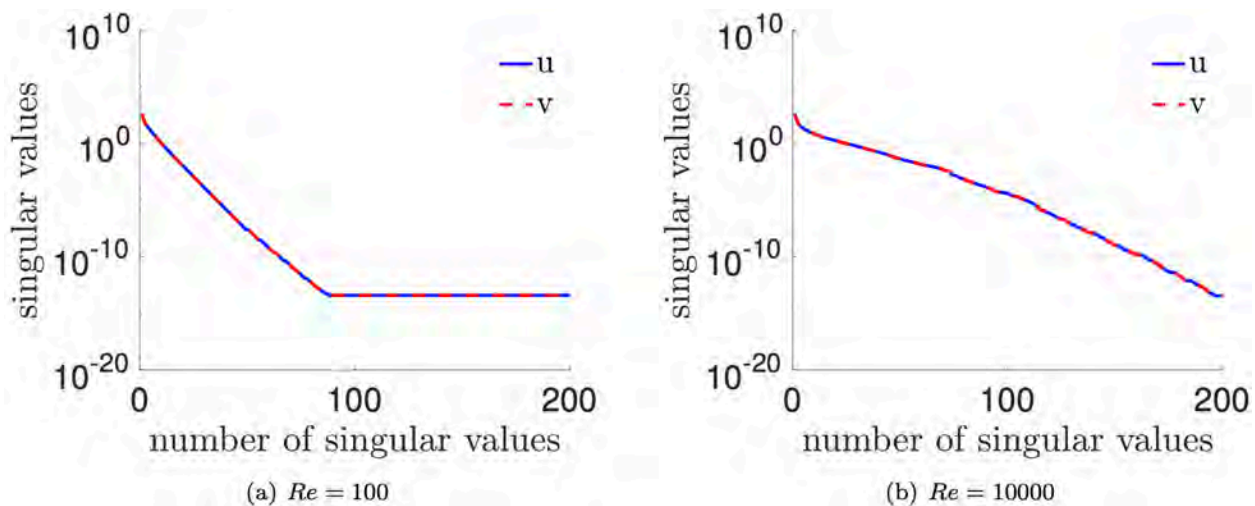
Figure 126: *2-D Burger's equation. Reynolds number vs. singular values* (Section 12.4.5). Performing SVD on FOM solution snapshots, which were partitioned into x and y-components, the influence of the Reynolds number on the singular values is illustrated. In diffusion-dominated problems, which are characterized by low Reynolds number, a rapid decay of singular values was observed. Less than 100 singular values were non-zero (in terms of double precision accuracy) in the present example for $Re = 100$. In problems with high Reynolds number, in which advection dominates over diffusive processes, the decay of singular values was much slower. As many as 200 singular values were different from zero in the case $Re = 1 \times 10^4$. (Figure reproduced with permission of the authors.)

The semi-discrete FOM was a finite-difference approximation in the spatial dimension on a uniform 60×60 grid of points (x_i, y_j), where $i \in \{1, 2, \dots, 60\}$, $j \in \{1, 2, \dots, 60\}$. First spatial derivatives were approximated by backward differences; central differences were used to approximate second derivatives. A spatial discretization led to a set of ODEs, which was partitioned into two subsets that corresponded to the

two spatial directions:

$$\dot{\boldsymbol{U}} = \boldsymbol{f}_u(\boldsymbol{U}, \boldsymbol{V}), \qquad \dot{\boldsymbol{V}} = \boldsymbol{f}_u(\boldsymbol{U}, \boldsymbol{V}), \qquad \boldsymbol{U}, \boldsymbol{V} \in \mathbb{R}^{(n_x-2)\times(n_y-2)}, \tag{486}$$

where $\boldsymbol{U}, \boldsymbol{V}$ comprised the components of velocity vectors at the grid points in x and y-direction, respectively. For the nonlinear functions $\boldsymbol{f}_u, \boldsymbol{f}_v : \mathbb{R}^{(n_x-2)\times(n_y-2)} \times \mathbb{R}^{(n_x-2)\times(n_y-2)} \to \mathbb{R}^{(n_x-2)\times(n_y-2)}$, which follow from the spatial discretization of the advection and diffusion terms [47]. In line with the partitioning the system of equations (486), two separate autoencoders were trained for $\boldsymbol{U}$ and $\boldsymbol{V}$, respectively, since less memory was required as opposed to a single autoencoder for the full set of unknowns $(\boldsymbol{U}^T, \boldsymbol{V}^T)^T$.

For time integration, the backward Euler scheme with a constant step size $\Delta t = 2/n_t$ was applied, where $n_t = 1500$ was used in the example. The solutions corresponding to the parameter values $\mu \in \{0.9, 0.95, 1.05, 1.1\}$ were collected as training data, which amounted to a total of $4 \times (n_t + 1) = 6004$ snapshots. Ten percent (10%) of the snapshots were retained as validation set (see Sec. 6.1); no test set was used.

Figure 126 shows the influence of the Reynolds number on the singular values obtained from solution snapshots of the FOM. For $Re = 100$, i.e., when diffusion was dominant, the singular values decayed rapidly as compared to an advection-dominated problem with a high Reynolds number of $Re = 1 \times 10^4$, for which the reduced-order models were constructed in what follows. In other words, the dimensionality of the tangent space of the FOM's solution was more than twice as large in the advection-dominated case, which limited the feasible reduction in dimensionality by means of linear subspace methods. Note that the singular values were the same for both components of the velocity field u and v. The problem was symmetric about the diagonal from the lower-left ($x = 0$, $y = 0$) to the upper-right ($x = 1$, $y = 1$) corner of the domain, which was also reflected in the solution snapshots illustrated in Figure 125. For this reason, we only show the results related to the x-component of the velocity field in what follows.

Both autoencoders (for $\boldsymbol{U}$ and $\boldsymbol{V}$, respectively) had the same structure. In each encoder, the (single) hidden layer had a width of 6728 neurons, which were referred to as "nodes" in [47]; the hidden layer of the (sparse) decoder networks was 33,730 neurons wide. To train the autoencoders, the Adam algorithm (see Sec. 6.5.6 and Algorithm 5) was used with an initial learning rate of 0.001. The learning rate was decreased by a factor of 10 when the training loss did not decrease for 10 successive epochs. The batch size was 240, and training was stopped either after the maximum of 10,000 epochs or, alternatively, once the validation loss had stopped to decrease for 200 epochs. The (single) hidden layers of the encoder networks had a width of 6728 neurons; with 33,730 neurons, the decoders' hidden layers were almost five times wider. The parameters of all neural networks were initialized according to the *Kaiming He* initialization [61].

To evaluate the accuracy of the NM-ROMs proposed in [47], the Burgers' equation was solved for the target parameter $\mu = 1$, for which no solution snapshots were included in the training data. Figure 127 compares the relative errors (Eq. (444)) of the nonlinear-manifold-based and linear-projection-based ROMs as a function of the reduced dimension n_s. Irrespective of whether a Galerkin or Petrov-Galerkin approach was used, nonlinear-manifold based ROMs (NM-Galerkin, NM-LSPG) were superior to their linear-subspace counterparts (LS-Galerkin, LS-LSPG). The figure also shows the so-called *projections errors*, which are lower bounds for the relative errors of linear-subspace and nonlinear-manifold-based errors, see [47] for their definitions. Note that the relative error of the NM-LSPG was smaller than the lower error bound of linear-subspace ROMs. As noted in [47], linear-subspace ROMs performed relatively poorly for the problem at hand, and even failed to converge. Both, the LS-Galerkin and the LS-LSPG-ROM showed relative errors of 1 if their dimension was 10 or more in the present problem. The NM-Galerkin ROM fell behind the NM-LSPG ROM in terms of accuracy. We also note that both NM-based ROMs hardly showed any reduction in error if their dimension was increased beyond five.

The authors of [47] also studied the impact how the size of the parameter set $\mathcal{D}_{\text{train}}$, which translated into the amount of training data, affected the accuracy of ROMs. For this purpose, parameter sets with
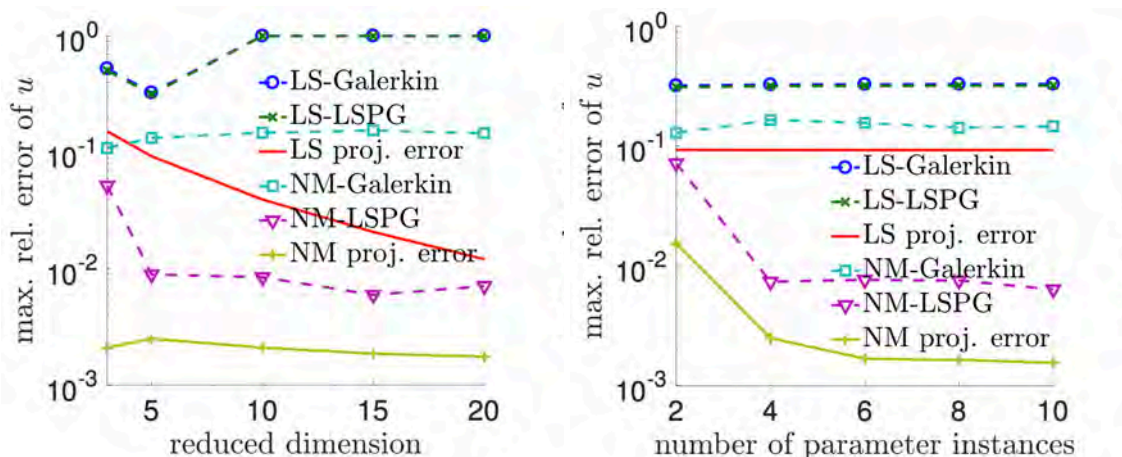
Figure 127: *2D-Burgers' equation: relative errors of nonlinear manifold and linear subspace ROMs* (Section 12.4.5). (Figure reproduced with permission of the authors.)

	NM-LSPG-HR						LS-LSPG-HR					
Residual basis	55	56	51	53	54	44	59	53	53	53	53	53
Residual samples	58	59	54	56	57	47	59	58	59	56	55	53
Max. rel. error (%)	0.93	0.94	0.95	0.97	0.97	0.98	34.38	37.73	37.84	37.95	37.96	37.97
Wall-clock time (sec)	12.15	12.35	12.09	12.14	12.29	12.01	5.26	5.02	4.86	5.05	4.75	7.18
Speed-up	11.58	11.39	11.63	11.58	11.44	11.71	26.76	28.02	28.95	27.83	29.61	19.58

Table 7: *2-D Burger's equation. Juxtaposition of hyper-reduced ROMs: speed-up and accuracy* (Section 12.4.5). The 6 respective best least-squares Petrov-Galerkin ROMs built upon nonlinear manifold approximation by means of autoencoders and linear subspaces are compared, where 'best' refers to the maximum error relative to the FOM (Eq. (444)). The optimal dimension of the basis Φ_r and the number of sampling indices p_j used in the gappy reconstruction of the nonlinear residual lie in similar ranges for all ROMs listed. The hyper-reduced ROMs achieve speed-up in wall-clock time of factors of approximately 11 for the nonlinear-manifold-based approach up to a factor of almost 30 in linear-subspace ROMs. While the former show a maximum relative error of below 1 %, the latter fail to reproduce the FOM's behavior by a large margin. (Table reproduced with permission of the authors.)

$n_{\text{train}} = 2, 4, 6, 8$ values of $\mu \in \mathcal{D}$, which were referred to as "parameter instances," were created, where the target value $\mu = 1$ remained excluded, i.e.,

$$\mathcal{D}_{\text{train}} = \{0.9 + 0.2i/n_{\text{train}}, i = 0, \ldots, n_{\text{train}}\} \setminus \{1\}; \tag{487}$$

the reduced dimension was set to $n_s = 5$. Figure 127 (right) reveals that, for the NM-LSPG ROM, 4 "parameter instances" were sufficient to reduce the maximum relative error below 1 % in the present problem. None of the ROMs benefited from increasing the parameter set, for which the training data were generated.

Hyper-reduction turned out to be crucial with respect to computational efficiency. For a reduced dimension of $n_s = 5$, all ROMs except for the NM-LSPG ROM, which achieved a minor speedup, were less efficient than the FOM in terms of wall-clock time. The dimension of the residual basis n_r and the number of sampling indices n_z were both varied in the range from 40 to 60 to quantify their relation to the maximum relative error. For this purpose, the number of training instances was again set to $n_{\text{train}} = 4$ and the reduced dimension was fixed to $n_s = 5$. Table 7 compares the 6 best—in terms of maximum error relative to the FOM—hyper-reduced least-squares Petrov-Galerkin ROMs based on nonlinear manifolds and linear subspaces, respectively. The NM-LSPG-HR ROM in [47] was able to achieve a speed-up of more than a factor of 11 while keeping the maximum relative error below 1 %. Though the speed-up of the linear-subspace

counterpart was more than twice as large, relative errors beyond 34 % rendered these ROMs worthless.
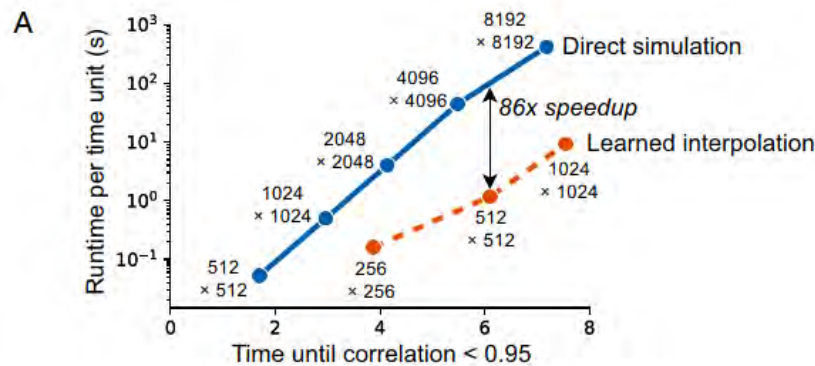
Figure 128: *Machine-learning accelerated CFD* (Section 12.4.5). Speed-up factor, compared to direct integration, was much higher than those obtained from nonlinear model-order reduction in Table 7 [317]. Permission of NAS.
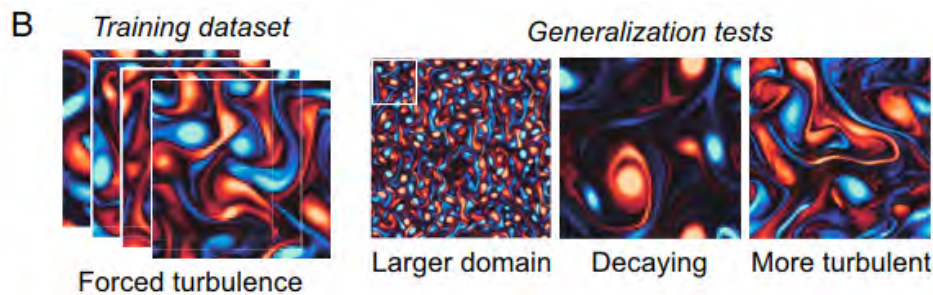
Figure 129: *Machine-learning accelerated CFD* (Section 12.4.5). Good accuracy and good generalization, devoiding of non-physical solutions [317]. Permission of NAS.
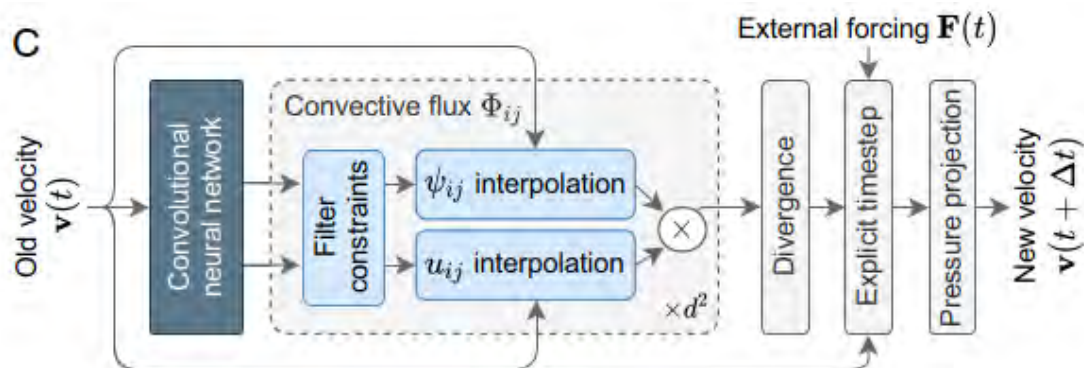
Figure 130: *Machine-learning accelerated CFD* (Section 12.4.5). The neural network generates interpolation coefficients based on local-flow properties, while ensuring at least first-order accuracy relative to the grid spacing [317]. Permission of NAS.

Remark 12.8. *Machine-learning accelerated CFD.* A hybrid method between traditional direct integration of the Navier-Stokes equation and machine learning (ML) interpolation was presented in [317] (Figure 130), where a speed-up factor close to 90, many times higher than those in Table 7, was obtained, Figure 128, while generalizing well (Figure 129). Grounded on the traditional direct integration, such hybrid method

would avoid non-physical solutions of pure machine-learning methods, such as the physics-inspired machine learning (Section 9.5, Remark 9.4), maintain higher accuracy as obtained with direct integration, and at the same time benefit from an acceleration from the learned interpolation.   ∎

Remark 12.9. In concluding this section, we mention the 2023 review paper [318], brought to our attention by a reviewer, on "A state-of-the-art review on machine learning-based multiscale modeling, simulation, homogenization and design of materials." This review paper would nicely complement our present review paper.   ∎

13 Historical perspective

13.1 Early inspiration from biological neurons

In the early days, many papers on artificial neural networks, particularly for applications in engineering, started to motivate readers with a figure of a biological neuron as in Figure 131 (see, e.g., [23], Figure 1a), before displaying an artificial neuron (e.g, [23], Figure 1b). When artificial neural networks took a foothold
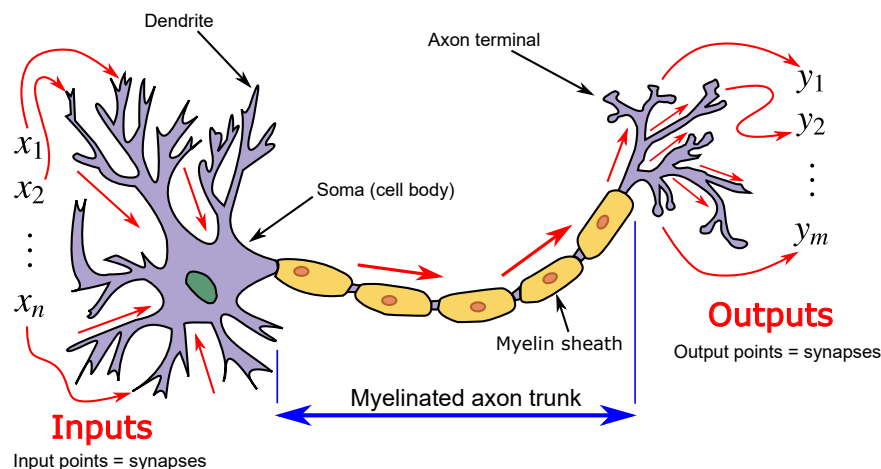
Figure 131: *Biological Neuron and signal flow* (Sections 4.4.4, 13.1, 13.2.2) along myelinated axon, with inputs at the synapses (input points) in the dendrites and with outputs at the axon terminals (output points,which are also the synapses for the next neuron). Each input current x_i is multiplied by the weight w_i, then all weighted input currents are summed together (linear combination), with $i = 1, \ldots, n$, to form the total synaptic input current I_s into the soma (cell body). The corresponding artificial neuron is in Figure 36 in Section 4.4.4. (Figure adapted from Wikipedia version 14:29, 2 May 2019).

in the research community, there was no need to motivate with a biological neuron, e.g., [38] [20], which began directly with an artificial neuron.

13.2 Spatial / temporal combination of inputs, weights, biases

Both [21] and [78] referred to Rosenblatt (1958) [119], who first proposed using a linear combination of inputs with weights, and with biases (thresholds). The authors of [78], p. 14, only mentioned the "linear model" defined—using the notation convention in Eq. (16)—as

$$f(\boldsymbol{x}, \boldsymbol{w}) = \sum_{i}^{n} x_i w_i = \boldsymbol{w}\boldsymbol{x} \tag{488}$$

without the bias. On the other hand, it was written in [21] that "Rosenblatt proposed a simple rule to compute the output. He introduced weights, real numbers expressing the importance of the respective inputs to the

output" and "some threshold value," and attributed the following equation to Rosenblatt

$$\text{output} = \begin{cases} 0 \text{ if } \sum_j w_j x_j \leq \text{threshold} \\ 1 \text{ if } \sum_j w_j x_j > \text{threshold} \end{cases} \tag{489}$$

where the threshold is simply the negative of the bias $b_i^{(\ell)}$ in Eq. (26)[293] or $(-b)$ in Figure 132, which is a graphical representation of Eq. (489). The author of [21] went on to say, "That's all there is to how a

$$\boldsymbol{X} \longrightarrow \boxed{z^{(1)}} \longrightarrow \boxed{H(\cdot)} \longrightarrow \widetilde{\boldsymbol{y}}$$
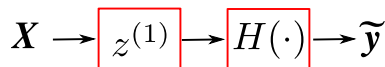
Figure 132: The *perceptron network* (Sections 4.5, 13.2)—introduced by Rosenblatt (1958) [119], (1962) [120]—has a linear combination with weights and bias as expressed in $z^{(1)}(\boldsymbol{x}_i) = \boldsymbol{w}\boldsymbol{x}_i + b \in \mathbb{R}$, but differs from the one-layer network in Figure 37 in that it is activated by the Heaviside function. That the Rosenblatt perceptron cannot represent the XOR function; see Section 4.5.

perceptron works!" Such statement could be highly misleading for first-time learners in discounting Rosenblatt's important contributions, which were extensively inspired from neuroscience, and were not limited to the perceptron as a machine-learning algorithm, but also to the development of the Mark I computer, a hardware implementation of the perceptron; see Figure 133 [319] [120] [320].

Moreover, adding to the confusion for first-time learners, another error and misleading statement about the "Rosenblatt perceptron" in connection with Eq. (488) and Eq. (489)—which represent a single neuron—is in [78], p. 13, where it was stated that the "Rosenblatt perceptron" involved only a "single neuron":

> "The first wave started with cybernetics in the 1940s-1960s, with the development of theories of biological learning (McCulloch and Pitts, 1943; Hebb, 1949) and implementations of the first models, such as the perceptron (Rosenblatt, 1958), enabling the training of a single neuron." [78], p. 13.

The error of considering the Rosenblatt perceptron as having a "single neuron" is also reported in Figure 42, which is Figure 1.11 in [78], p. 23. But the Rosenblatt perceptron as described in the cited reference Rosenblatt (1958) [119] and in Rosenblatt (1960) [319] was a network, called a "nerve net":

> "Any perceptron, or nerve net, consists of a network of "cells," or signal generating units, and connections between them."

---

[293] "The Perceptron's design was much like that of the modern neural net, except that it had only one layer with adjustable weights and thresholds, sandwiched between input and output layers" [77]. In the neuroscientific terminology that Rosenblatt (1958) [119] used, the input layer contains the sensory units, the middle (hidden) layer contains the "association units," and the output layer contains the response units. Due to the difference in notation and due to "neurodynamics" as a new field for most readers, we provide here some markers that could help track down where Rosenblatt used linear combination of the inputs. Rosenblatt (1962) [2], p. 82, defined the "transmission function $c_{ij}^\star$" for the connection between two "units" (neurons) u_i and u_j, with $c_{ij}^\star$ playing the same role as that of the term $w_{ij}^{(\ell)} y_j^{(\ell-1)}$ in $z_i^{(\ell)}$ in Eq. (26). Then for an "elementary perceptron", the transmission function $c_{ij}^\star$ was defined in [2], p. 85, to be equal to the output of unit u_i (equivalent to $y_i^{(\ell-1)}$) multiplied by the "coupling coefficient" v_{ij} (between unit u_i and unit u_j), with v_{ij} being the equivalent of the weight $w_{ij}^{(\ell)}$ in Eq. (26), ignoring the time dependence. The word "weight," meaning coefficient, was not used often in [2], and not at all in [119].

Figure 133: *Rosenblatt and the Mark I computer* (Sections 4.6.1, 13.2) based on the perceptron, described in the New York Times article titled "New Navy device learns by doing" on 1958 July 8 (Internet archive), as a "computer designed to read and grow wiser", and would be able to "walk, talk, see, write, reproduce itself and be conscious of its existence. The first perceptron will have about 1,000 electronic "association cells" [A-units] receiving electrical impulses from an eye-like scanning device with 400 photo cells". See also the Youtube video "Perceptron Research from the 50's & 60's, clip". Sometimes, it is incorrectly thought that Rosenblatt's network had only one neuron (A-unit); see Figure 42, Section 4.6.1.

Such a "nerve net" would surely not just contain a "single neuron". Indeed, the report by Rosenblatt (1957) [1] that appeared a year earlier mentioned a network (with one layer) containing as many a thousand neurons, called "association cells" (or A-units):[294]

> "Thus with 1000 A-units connected to each R-unit [response unit or output], and a system in which 1% of the A-units respond to stimuli of a given size (i.e., $P_a = .01$), the probability of making a correct discrimination with one unit of training, after 10^6 stimuli have been associated to each response in the system, is equal to the 2.23 sigma level, or a probability of 0.987 of being correct." [1], p. 16.

The perceptron with one thousand A-units mentioned in [1] was also reported in the New York Times article "New Navy device learns by doing" on 1958 July 8 (Internet archive); see Figure 133. Even if the report by Rosenblatt (1957) [1] were not immediately accessible, it was stated in no uncertain terms that the perceptron was a machine with many neurons:

---

[294] We were so thoroughly misled in thinking that the "Rosenblatt perceptron" was a single neuron that we were surprised to learn that Rosenblatt had built the Mark I computer with many neurons.

"The organization of a typical photo-perceptron (a perceptron responding To optical patterns as stimuli) is shown In Figure1. ... [Rule] 1. Stimuli impinge on a retina of Sensory units (S-points), which are Assumed to respond on an all-or-nothing basis[295]. [Rule] 2. Impulses are transmitted to a set Of association cells (A-units) [neurons]... If the algebraic sum of excitatory and Inhibitory impulse intensities[296] is equal To or greater than the threshold[297] (θ) of The A-unit, then the A-unit fires, again On an all-or-nothing basis." [119]

Figure 1 in [119] described a network ("nerve net") with many A-units (neurons). Does anyone still read the classics anymore?

Rosenblatt's (1962) book [2], p. 33, provided the following neuroscientific explanation for using a linear combination (weight sum / voting) of the inputs in both time and space:

"The arrival of a single (excitatory) impulse gives rise to a partial depolarization of the post-synaptic[298] membrane surface, which spreads over an appreciable area, and decays exponentially with time. This is called a local excitatory state (l.e.s.). The l.e.s. due to successive impulses is (approximately) additive. Several impulses arriving in sufficiently close succession may thus combine to touch off an impulse in the receiving neuron if the local excitatory state at the base of the axon achieves the threshold level. This phenomenon is called temporal summation. Similarly, impulses which arrive at different points on the cell body or on the dendrites may combine by spatial summation to trigger an impulse if the l.e.s. induced at the base of the axon is strong enough."

The *spatial summation* of the input synaptic currents is also consistent with Kirchhoff's current law of summing the electrical currents at a junction in an electrical network.[299] We first look at linear combination in the static case, followed by the dynamic case with Volterra series.

13.2.1 Static, comparing modern to classic literature

"A classic is something that everybody wants to have read and nobody wants to read."

Mark Twain

Readers not interested in reading the classics can skip this section. Here, we will not review the perceptron algorithm,[300] but focus our attention on the historical details not found in many modern references, and connect Eq. (488) and Eq. (489) to the original paper by Rosenblatt (1958) [119]. But the problem is that such task is not directly obvious for readers of modern literature, such as [78] for the following reasons:

---

[295] Heaviside activation function, see Figure 132 for the case of one neuron.

[296] Weighted sum / voting (or linear combination) of inputs; see Eq. (488), Eq. (493), Eq. (494).

[297] The negative of the bias $b_i^{(\ell)}$ in Eq. (26).

[298] Refer to Figure 131. A *synapse* (meaning "junction") is "a structure that permits a neuron (or nerve cell) to pass an electrical or chemical signal to another neuron", and consists of three parts: the *presynaptic* part (which is an axon terminal of an upstream neuron from which the signal came), the gap called the synaptic cleft, and the *postsynaptic* part, located on a dendrite or on the neuron cell body (called the *soma*); [19], p. 6; "Synapse", Wikipedia, version 16:33, 3 March 2019. A *dendrite* is a conduit for transmitting the electrochemical signal received from another neuron, and passing through a synapse located on that dendrite; "Dendrite", Wikipedia, version 23:39, 15 April 2019. A synapse is thus an input point to a neuron in a biological neural network. An axon, or nerve fiber, is a "long, slender projection of a neuron, that conducts electrical impulses known as action potentials" away from the soma to the axon terminals, which are the presynaptic parts; "Axon terminal", Wikipedia, version 18:13, 27 February 2019.

[299] See "Kirchhoff's circuit laws", Wikipedia, version 14:24, 1 May 2019.

[300] See, e.g., "Perceptron", Wikipedia, version 13:11, 10 May 2019, and many other references.

- Rosenblatt's work in [119] was based on neuroscience, which is confusing to those without this background;

- Unfamiliar notations and concepts for readers coming from deep-learning literature, such as [21], [78];

- The word "weight" was not used at all in [119], and thus cannot be used to indirectly search for hints of equations similar to Eq. (488) or Eq. (489);

- The word "threshold" was used several times, such as in the sentence "If the algebraic sum of excitatory and inhibitory impulse intensities is equal to or greater than the threshold (θ)"[301] of the A-unit, then the A-unit fires, again on an all-or-nothing basis". The threshold θ is used in Eq. (2) of [119]:

$$e - i - l_e + l_i + g_e - g_i \geq \theta \,, \tag{490}$$

  where e is the number excitatory stimulus components received by the A-unit (neuron, or associated unit), i the number of inhibitory stimulus components, l_e the number of lost excitatory components, l_i the number of lost inhibitory components, g_e the number of gained excitatory components, g_i the number of gained inhibitory components. But all these quantities are positive integers, and thus would not be the real-number weights w_j in Eq. (489), of which the inputs x_j also have no clear equivalence in Eq. (490).

As will be shown below, it was misleading to refer to [119] for equations such as Eq. (488) and Eq. (489), even though [119] contained the seed ideas leading to these equations upon refinement as presented in [120], which was in turn based on the book by Rosenblatt (1962) [2].

Instead of a direct reading of [119], we suggest reading key publications in reverse chronological orders. We also use the original notations to help readers to identify quickly the relevant equations in the classic literature.

The authors of [121] introduced a general class of machines, each known under different names, but decided to call all these machines as "perceptrons" in honor of the pioneering work of Rosenblatt. General perceptrons were defined in [121], p. 10, as follows. Let φ_i be the ith image characteristic, called an image predicate, which consists of a verb and an object, such as "is a circle", "is a convex figure", etc. An image predicate is also known as an image feature.[302] For example, let the ith image characteristic is whether an image "is a circle", then $\varphi_i \equiv \varphi_{circle}$. If an image X is a circle, then $\varphi_{circle}(X) = 1$; if not, $\varphi_{circle}(X) = 0$:

$$\varphi_{circle}(X) = \begin{cases} 1 \text{ if } X \text{ is a circle} \\ 0 \text{ if } X \text{ is not a circle} \end{cases} \tag{491}$$

Let Φ be a family of simple image predicates:

$$\Phi = [\varphi_1, \ldots, \varphi_n] \tag{492}$$

A *general* perceptron was defined as a more complex predicate, denoted by ψ, which was a *weighted voting* or *linear combination* of the simple predicates in Φ such that[303]

$$\psi(X) = 1 \iff \alpha_1 \varphi_1 + \ldots + \alpha_n \varphi_n > \theta \tag{493}$$

---

[301] Of course, the notation θ (lightface) here does not designate the set of parameters, denoted by $\boldsymbol{\theta}$ (boldface) in Eq. (31).

[302] See [78], p. 3. Another example of a feature is a piece of information about a patient for medical diagnostics. "For many tasks, it is difficult to know which features should be extracted." For example, to detect cars, we can try to detect the wheels, but "it is difficult to describe exactly what a wheel looks like in terms of pixel values", due to shadows, glares, objects obscuring parts of a wheel, etc.

[303] [121], p. 10.

with α_i being the weight associated with the ith image predicate φ_i, and θ the threshold or the negative of the bias. As such "each predicate of Φ is supposed to provide some evidence about whether ψ is true for any figure X."[304] The expression on the right of the equivalence sign, written with the notations used here, is the general case of Eq. (489). The authors of [121], p. 12, then defined the Rosenblatt perceptron as a special case of Eq. (493) in which the image predicates in Φ were random Boolean functions, generated by a random process according to a probability distribution.

The next paper to read is [120], which was based on the book by Rosenblatt (1962) [2], and from where the following equation[305] can be identified as being similar to Eq. (493) and Eq. (489), again in its original notation as

$$\sum_{\mu=1}^{N_a} y_\mu b_{\mu i} > \Theta \,, \text{ for } i = 1, \ldots, n \,, \tag{494}$$

where y_μ was the weight corresponding to the input $b_{\mu i}$ to the "associated unit" a_μ (neuron) from the stimulus pattern S_i (ith example in the dataset), N_a the number of "associated units" (neurons), n the number of "stimulus patterns" (examples in the dataset), and Θ the second of two thresholds, which were fixed real non-negative numbers, and which corresponded to the negative of the bias $b_i^{(\ell)}$ in Eq. (26), or $(-b)$ in Figure 132.

To discriminate between two classes, the input $b_{\mu i}$ took the value $+1$ or -1, when there was excitation coming from the stimulus pattern S_i to the neuron a_μ, and the value 0 when there was no excitation from S_i to a_μ. When the weighted voting or linear combination in Eq. (494) surpassed the threshold Θ, then the response was correct (or yields the value $+1$).

If the algebraic sum α_μ^i of the connection strengths $C_{\sigma\mu}$ between the neuron (associated unit) a_μ and the sensory unit s_σ inside the pattern (example) S_i surpassed a threshold θ (which was the first of two thresholds, and which does not correspond to the negative of the bias in modern networks), then the neuron a_μ was activated:[306]

$$\alpha_\mu^i := \sum_{\substack{\sigma \\ s_\sigma \in S_i}} C_{\sigma\mu} \geq \theta \tag{495}$$

Eq. (495) in [120] would correspond to Eq. (490) in [119], with the connection strengths $C_{\sigma\mu}$ being "random numbers having the possible values +1, -1, 0".

That Eq. (495) was not numbered in [120] indicates that it played a minor role in this paper. The reason is clear, since the author of [120] stated[307] that "the connections $C_{\sigma\mu}$ do not change", and thus "we may disregard the sensory retina altogether", i.e., Eq. (495).

Moreover, the very first sentence in [120] was "The perceptron is a self-organizing and adaptive system proposed by Rosenblatt", and the book by [2] was immediately cited as Ref. 1, whereas only much later in the fourth page of [120] did the author write "With the Perceptron, Rosenblatt offered for the first time a model...", and cited Rosenblatt's 1958 report first as Ref. 34, followed by the paper [119] as Ref. 35.

In a major work on AI dedicated to Rosenblatt after his death in a boat accident, the authors of [121], p.xi, in the Prologue of their book, referred to Rosenblatt's (1962) book [2] and not Rosenblatt's (1958) paper [119]:

---

[304] [121], p. 11.

[305] Eq. (4) in [120].

[306] First equation, unnumbered, in [120]. That this equation was unnumbered also indicated that it would not be subsequently referred to (and hence perhaps not considered as important).

[307] See above Eq. (1) in [120].

> "**The 1960s: Connectionists and Symbolists**
> Interest in connectionist networks revived dramatically in 1962 with the publication of Frank Rosenblatt's book *Principles of Neurodynamics* in which he defined the machines he named perceptrons and proved many theories about them."

In fact, Rosenblatt's (1958) paper [119] was never referred to in [121], except for a brief mention of the influence of "Rosenblatt's [1958]" work on p. 19, without the full bibliographic details. The authors of [121] wrote:

> "However, it is not our goal here to evaluate these theories [to model brain functioning], but only to sketch a picture of the intellectual stage that was set for the perceptron concept. In this setting, Rosenblatt's [1958] schemes quickly took root, and soon there were perhaps as many as a hundred groups, large and small, experimenting with the model either as a 'learnÂ-ing machine' or in the guise of 'adaptive' or 'self-organizing' networks or 'automatic control' systems."

So why was [119] often referred to for Eq. (488) or Eq. (489), instead of [120] or [2],[308] which would be much better references for these equations? One reason could be that citing [120] would not do justice to [119], which contained the germ of the idea, even though not as refined as four years later in [120] and [2]. Another reason could be the herd effect by following other authors who referred to [119], without actually reading the paper, or without comparing this paper to [120] or [2]. A best approach would be to refer to both [119] and [120], as papers like these would be more accessible than books like [2].

Remark 13.1. *The hype on the Rosenblatt perceptron* Mark I computer described in the 1958 New York Times article shown in Figure 133, together with the criticism of the Rosenblatt perceptron in [121] for failing to represent the XOR function, led to an early great disappointment on the possibilities of AI when overreached expectations for such device did not pan out, and contributed to the first AI winter that lasted until the 1980s, with a resurgence in interest due to the development of backpropagation and application in psychology as reported in [22]. But some sixty years since the Mark I computer, AI still cannot even think like human babies yet: "Understanding babies and young children may be one key to ensuring that the current "AI spring" continues—despite some chilly autumnal winds in the air" [321].   ∎

13.2.2 Dynamic, time dependence, Volterra series

For time-dependent input $x(t)$, the *continuous temporal summation*, mentioned in [2], p. 33, is present in all terms other than the constant term in the Volterra series[309] of the estimated output $z(t)$ as a result of

---

[308] A search on the Web of Science on 2019.07.04 indicated that [119] received 2,346 citations, whereas [120] received 168 citations. A search on Google Books on the same day indicated that [2] received 21 citations.

[309] [19], p. 46. "The Volterra series is a model for non-linear behavior similar to the Taylor series. It differs from the Taylor series in its ability to capture 'memory' effects. The Taylor series can be used for approximating the response of a nonlinear system to a given input if the output of this system depends strictly on the input at that particular time. In the Volterra series the output of the nonlinear system depends on the input to the system at all other times. This provides the ability to capture the 'memory' effect of devices like capacitors and inductors. It has been applied in the fields of medicine (biomedical engineering) and biology, especially *neuroscience*. In mathematics, a Volterra series denotes a functional expansion of a dynamic, nonlinear, time-invariant functional," in "Volterra series", Wikipedia, version 12:49, 13 August 2018.

the input $x(t)$

$$
\begin{aligned}
z(t) & = \mathcal{K}_0 + \sum_{n=1} \int \cdots \int \mathcal{K}_n(\tau_1, \ldots, \tau_n) \prod_{j=1}^{n} x(t - \tau_j) d\tau_j \\
& = \mathcal{K}_0 + \int \mathcal{K}_1(\tau_1) x(t - \tau_1) d\tau_1 + \iint \mathcal{K}_2(\tau_1, \tau_2) x(t - \tau_1) x(t - \tau_2) d\tau_1 d\tau_2 \\
& \quad + \iiint \mathcal{K}_3(\tau_1, \tau_2, \tau_3) x(t - \tau_1) x(t - \tau_2) x(t - \tau_3) d\tau_1 d\tau_2 d\tau_3 + \cdots
\end{aligned}
\tag{496}
$$

where $\mathcal{K}_n(\tau_1, \ldots, \tau_n)$ is the kernel of the nth order, with all integrals going from $\tau_j = 0$ to the current time $\tau_j = +\infty$, for $j = 1, \ldots, n$. The linear-order approximation of the Volterra series in Eq. (496) is then

$$
z(t) \approx \mathcal{K}_0 + \int_{\tau_1=0}^{\tau_1=+\infty} \mathcal{K}_1(\tau_1) x(t - \tau_1) d\tau_1 = \mathcal{K}_0 + \int_{\tau=-\infty}^{\tau=t} \mathcal{K}_1(t - \tau) x(\tau) d\tau
\tag{497}
$$

with the continuous linear combination (weighted sum) appearing in the second term. The convolution integral in Eq. (497) is the basis for convolutional networks for highly effective and efficient image recognition, inspired by mammalian visual system. A review of convolutional networks outside the scope here, despite them being the "greatest success story of biologically inspired artificial intelligence" [78], p. 353.

For biological neuron models, both The input $x(t)$ and the continuous weighted sum $z(t)$ can be either currents, with nA (nano Ampere) as dimension, or firing rates (frequency), with Hz (Hertz) as dimension.

Eq. (497) is the continuous temporal summation, counterpart of the discrete spatial summation in Eq. (26), with the constant term $\mathcal{K}_0$ playing a role similar to that of the bias $b_i^{(\ell)}$. The linear kernel $\mathcal{K}_1(\tau_1)$ (also called the Wiener kernel,[310] or synaptic kernel in brain modeling)[311] is the weight on the input $x(t - \tau_1)$, with τ_1 going from $-\infty$ to the current time t. In other words, the whole history of the input $x(t)$ prior to the current time has an influence on the output $z(t)$, with typically smaller weight for more distant input (fading memory). For this reason, the synaptic kernel used in the biological neuron firing-rate models is often chosen to have an exponential decay of the form:[312]

$$
\mathcal{K}_s(t) := \mathcal{K}_1(t) = \frac{1}{\tau_s} \exp\left(\frac{-t}{\tau_s}\right)
\tag{498}
$$

where τ_s is the synaptic time constant such that the smaller τ_s is, the less memory of past input values, and

$$
\tau_s \to 0 \Rightarrow \mathcal{K}_s \to \delta(t) \Rightarrow z(t) \to \mathcal{K}_0 + x(t)
\tag{499}
$$

i.e., the continuous weighted sum $z(t)$ would correspond to the instantaneous $x(t)$ (without memory of past input values) as the synaptic time constant τ_s goes to zero (no memory).

Remark 13.2. The discrete counterpart of the linear part of the Volterra series in Eq. (497) can be found in the exponential-smoothing time series in Eq. (212), with the kernel $\mathcal{K}_1(t - \tau)$ being the exponential function $\beta^{(t-i)}$; see Section 6.5.3 on exponential smoothing in forecasting. The similarity is even closer when the synaptic kernel $\mathcal{K}_1$ is of exponential form as in Eq. (498). ∎

In firing-rate models of the brain (see Figure 27), the function $x(t)$ represents an input firing-rate at a synapse, $\mathcal{K}_0$ is called the background firing rate, and the weighted sum $z(t)$ has the dimension of firing rate (Hz.

For a neuron with n pre-synaptic inputs $[x_1(t), \ldots, x_n(t)]$ (either currents or firing rates) as depicted in Figure 131 of a biological neuron, the total input $z(t)$ (current or firing rate, respectively) going into the soma

---

[310] Since the first two terms in the Volterra series coincide with the first two terms in the Wiener series; see [19], p. 46.
[311] See [19], p. 234.
[312] See [19], p. 234, below Eq. (7.3).

(cell body, Figure 131), called total somatic input, is a discrete weighted sum of all post-synaptic continuous weighted sums $z_i(t)$ expressed in Eq. (497), assuming the same synaptic kernel $\mathcal{K}_s$ at all synapses:

$$z(t) = \overline{\mathcal{K}}_0 + \sum_{i=1}^{n} w_i \int_{\tau=-\infty}^{\tau=t} \mathcal{K}_s(t-\tau)x_i(\tau)d\tau \tag{500}$$

with $\overline{\mathcal{K}}_0$ being the constant bias,[313] and w_i the synaptic weight associated with the synapse i.

Using the synaptic kernel Eq. (498) in Eq. (500) for the total somatic input $z(t)$, and differentiate,[314] the following ordinary differential equation is obtained

$$\boxed{\tau_s \frac{dz}{dt} = -z + \sum_i w_i x_i} \tag{501}$$

Remark 13.3. The second term in Eq. (501), with time-independent input x_i, is the steady state of the total somatic input $z(t)$:

$$z(t) = [z_0 - z_\infty] \exp\left(\frac{-t}{\tau}\right) + z_\infty , \tag{502}$$

with $z_0 := z(0)$, and $z_\infty := \sum_i w_i x_i \tag{503}$

$$z(t) \rightarrow z_\infty \text{ as } t \rightarrow \infty . \tag{504}$$

As a result, the subscript ∞ is often used to denote as the steady state solution, such as R_∞ in the model of neocortical neurons Eq. (509) [118]. ∎

For constant total somatic input z, the output firing-rate y is given by an activation function $a(\cdot)$ (e.g, scaled ReLU in Figure 25 and Figure 26) through relation[315]

$$\boxed{y = a(z) = c\max(0, z) = c[z]_+} \tag{505}$$

where c is a scaling constant to match the slope of the firing rate (F) vs input current (I) relation called the FI curve obtained from experiments; see Figure 27 and Figure 28. the total somatic input z can be thought of as being converted from current (nA) to frequency (Hz) by multiplying with the converting constant c.[316]

At this stage, there are two possible firing-rate models. The first firing-rate model consists of (1) Eq. (501), the ODE for the total somatic input firing rate z, followed by (2) the "static" relation between output firing-rate y and constant input firing rate z, expressed in Eq. (505), but now used for time-dependent total somatic input firing rate $z(t)$.

Remark 13.4. The steady-state output y_∞ in the first firing-rate model described in Eq. (500) and Eq. (505) in the case of constant inputs x_i is therefore

$$y_\infty = a(z_\infty) , \tag{506}$$

where z_∞ is given by Eq. (503). ∎

---

[313] The negative of the bias $\overline{\mathcal{K}}_0$ is the threshold. The constant bias $\overline{\mathcal{K}}_0$ is called the background firing rate when the inputs $x_i(t)$ are firing rates.

[314] In general, for $I(t) = \int_{x=A(t)}^{x=B(t)} f(x,t)dx$, then $\frac{dI(t)}{dt} = f(B(t),t)\overset{\bullet}{B}(t) - f(A(t),t)\overset{\bullet}{A}(t) + \int_{x=A(t)}^{x=B(t)} \frac{\partial f(x,t)}{\partial x}dx$.

[315] See [19], p. 234, in original notation as $v = F(I_s)$, with v being the output firing rate, F an activation function, and I_s the total synaptic current.

[316] See [19], p. 234, subsection "The Firing Rate".

The second firing-rate model consists of using Eq. (501) for the total somatic input firing rate z, which is then used as input for the following ODE for the output firing-rate y:

$$\tau_r \frac{dy}{dt} = -y + a(z(t)) \,, \tag{507}$$

where the activation function $a(\cdot)$ is applied on the time-dependent total somatic input firing rate $z(t)$, but with a different time constant τ_r, which describes how fast the output firing rate y approaches steady state for *constant* input z.

Eq. (507) is a recurring theme that has been frequently used in papers in neuroscience and artificial neural networks. Below are a few relevant papers for this review, particularly the *continuous* recurrent neural networks (RNNs)—such as Eq. (510), Eq. (512), and Eqs. (515)-(516)—which are the counterparts of the *discrete* RNNs in Section 7.1.
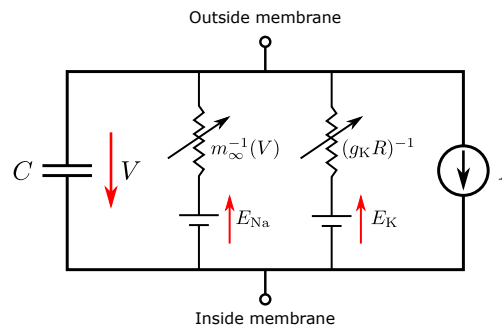
Figure 134: *Model of neocortical neurons* in [118] as a simplification of the model in [322] (Section 13.2.2): A capacitor C with a potential V across its plates, in parallel with the equilibrium potentials E_{Na} (sodium) and E_{K} (potassium) in opposite direction. Two variable resistors $m_\infty^{-1}(V)$ and $[g_{\mathrm{K}} R(V)]^{-1}$ are each in series with one of the mentioned two equilibrium potentials. The capacitor C is also in parallel with a current source I. The notation R is used here for the "recovery variable", not a resistor. See Eqs. (508)-(509).

The model for neocortical neurons in [118], a simplification of the model by [322], was employed in [116] as starting point to develop a formulation that produced SubFigure(b) in Figure 28 consists of two coupled ODE's[317]

$$C \frac{dV}{dt} = -m_\infty(V)(V - E_{\mathrm{Na}}) - g_{\mathrm{K}} R(V - E_{\mathrm{K}}) + I \tag{508}$$

$$\tau \frac{dR}{dt} = -R + R_\infty(V) \tag{509}$$

where $m_\infty(V)$ and $R_\infty(V)$ are prescribed quadratic polynomials in the potential V, making the right-hand side of Eq. (508) of cubic order. Eq. (508) describes the change in the membrane potential V due to the capacitance C in parallel with other circuit elements shown in Figure 134, with (1) $m_\infty(V)$ and E_{Na} being the activation function and equilibrium potential for the sodium ion (Na$^+$), respectively, (2) g_{K}, R, and E_{K} being the conductance, recovery variable, and equilibrium potential for the potassium ion (K$^+$), respectively, and (3) I the stimulating current. Eq. (509) for the recovery variable R has the same form as Eq. (507), with $R_\infty(V)$ being the steady state; see Remark 13.4.

To create a continuous recurrent neural network described by ODEs in Eq. (510), the input $x_i(t)$ in Eq. (500) is replaced by the output $y_j(t)$ (i.e., a feedback), and the bias $\overline{\mathcal{K}}_0$ becomes an input, now denoted by x_i. Electrical circuits can be designed to approximate the dynamical behavior a spatially-discrete,

---

[317] Eq. (1) and Eq. (2) in [116].

temporally-continuous recurrent neural network (RNN) described by Eq. (510), which is Eq. (1) in [32]:[318]

$$\tau_i \frac{dy_i}{dt} = -y_i + \left[x_i + \sum_j w_{ij} y_j \right]_+ . \tag{510}$$

The network is called symmetric if the weight matrix is symmetric, i.e.,

$$w_{ij} = w_{ji} . \tag{511}$$

The difference between Eq. (510) and Eq. (507) is that Eq. (507) is based on the expression for $z(t)$ in Eq. (500), and thus has no feedback loop.

A time-dependent time delay $d(t)$ can be introduced into Eq. (510) leading to spatially-discrete, *temporally-continuous* RNNs with time delay, which is Eq. (1) in [323]:[319]

$$\boldsymbol{T} \frac{d\boldsymbol{y}}{dt} = -\boldsymbol{y} + a[\boldsymbol{x} + \boldsymbol{W} \boldsymbol{y}(t - d(t))] \tag{512}$$

where the diagonal matrix $\boldsymbol{T} = \mathrm{Diag}[\tau_1, ..., \tau_n] \in \mathbb{R}^{n \times n}$ contains the synaptic time constants τ_i as its diagonal coefficients, the matrix $\boldsymbol{y} = [y_1, \ldots, y_n] \in \mathbb{R}^{n \times 1}$ contains the outputs, the bias matrix $\boldsymbol{x} \in \mathbb{R}^{n \times 1}$ plays the role of input matrix (thus denoted by $\boldsymbol{x}$ instead of $\boldsymbol{b}$), $a(\cdot)$ is the activation function, $\boldsymbol{W} \in \mathbb{R}^{n \times n}$ the weight matrix, and $d(t)$ the time-dependent delay; see Figure 135.

For discrete RNNs, the delay is a constant integer set to one, i.e.,

$$d(n) = 1 , \text{ with } n = t \text{ (integer)} , \tag{513}$$

as expressed in Eq. (275) in Section 7.1.

Both Eq. (510) and Eq. (512) can be rewritten in the following form:

$$\boldsymbol{y}(t) = -\boldsymbol{T} \frac{d\boldsymbol{y}(t)}{dt} + a[\boldsymbol{x}(t) + \boldsymbol{W} \boldsymbol{y}(t - d(t))] = f(\boldsymbol{y}(t), \boldsymbol{y}(t - d)\boldsymbol{x}(t), d(t)) . \tag{514}$$

A densely distributed pre-synaptic input points [see Eq. (500) and Figure 131 of a biological neurons] can be approximated by a continuous distribution in space, represented by $x(s, t)$, with s being the space variable, and t the time variable. In this case, a *continuous* RNN in both space and time, called "continuously labeled RNN", can be written as follows:[320]

$$\tau_r \frac{\partial y(s, t)}{\partial t} = -y(s, t) + a(z(s, t)) \tag{515}$$

$$z(s, t) = \rho_s \int [W(s, \gamma) x(\gamma, t) + M(s, \gamma) y(\gamma, t)] \, d\gamma \tag{516}$$

where ρ_s is the neuron density, assumed to be constant. Space-time continuous RNNs such as Eqs. (515)-(516) have been used to model, e.g., the visually responsive neurons in the premotor cortex [19], p. 242.

---

[318] In original notation, Eq. (510) was written as $\tau_i(x_i) \frac{dx_i}{dt} + x_i = \left[b_i + \sum_j W_{ij} x_j \right]_+$ in [32], whose outputs x_i in the previous expression are now rewritten as y_i in Eq. (510), and the biases $b_i(t)$, playing the role of inputs, are rewritten as $x_i(t)$ to be consistent with the notation for inputs and outputs used throughout in the present work; see Section 4.2 on matrix notation, Eq. (7), and Section 7.1 on *discrete* recurrent neural networks, which are discrete in both space and time. The paper [32] was cited in both [19] and [36], with the latter leading us to it.

[319] In original notation, Eq. (512) was written as $\dot{z} = -Az + f(Wz(t - h(t)) + J)$ in [323], where $z = \boldsymbol{y}$, $A = \boldsymbol{T}^{-1}$, $h(t) = d(t)$ and $J = \boldsymbol{x}$.

[320] [19], p. 240, Eq. (7.14).
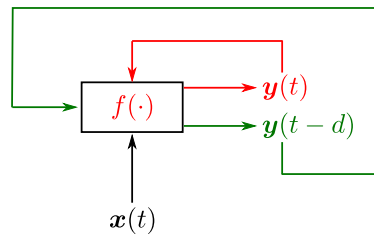
Figure 135: *Continuous recurrent neural network with time-dependent delay $d(t)$ (green feed-back loop, Section* 13.2.2*), as expressed in Eq.* (514)*, where $f(\cdot)$ is the operator with the first derivative term plus a standard static term—which is an activation function acting on linear combination of input and bias, i.e., $a(z(t))$ as in Eq.* (35) *and Eq.* (32)*—$x(t)$ the input, $y(t)$ the output with the red feedback loop, and $y(t-d)$ the delayed output with the green feedback loop. This figure is the more general continuous counterpart of the discrete RNN in Figure* 79*, represented by Eq.* (275)*, which is a particular case of Eq.* (514)*. We also refer readers to Remark* 7.1 *and the notation equivalence $y(t) \equiv h(t)$ as noted in Eq.* (276)*.*

(b) Anatomy. Abdominal nerve cord and ganglions (black). [324]. A portion of the abdominal nerve cord and a ganglion are shown in Figure 137. (Figure reproduced with permission of the authors.)
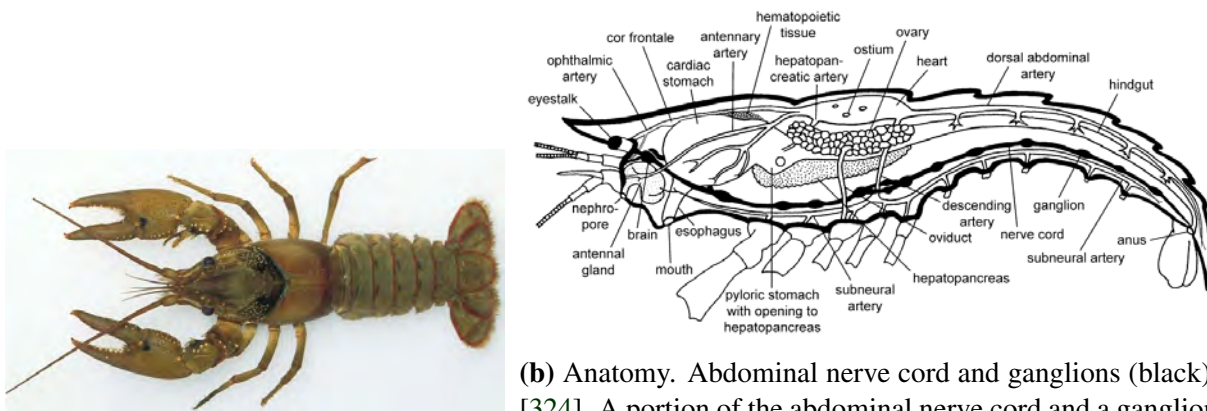
(a) Crayfish. Courtesy of Missouri Dept of Conservation

Figure 136: *Crayfish (Section* 13.3.2*), freshwater crustaceans. Anatomy.*

13.3 Activation functions

13.3.1 Logistic sigmoid

The use of the logistic sigmoid function (Figure 30) in neuroscience dates back since the seminal work of Nobel Laureates Hodgkin & Huxley (1952) [322] in the form of an electrical circuit (Figure 134), and since the work reported in [35] in a form closer to today's network; see also [325] and [37].

The authors of [78], p. 219, remarked: "Despite the early popularity of rectification (see next Section 13.3.2), it was largely replaced by sigmoids in the 1980s, perhaps because sigmoids perform better when neural networks are very small."

The rectified linear function has, however, made a come back and was a key component responsible for the success of deep learning, and helped inspired a variant that in 2015 surpassed human-level performance in image classification, as "it expedites convergence of the training procedure [16] and leads to better solutions [21, 8, 20, 34] than conventional sigmoid-like units" [61]. See Section 5.3.3 on Parametric Rectified Linear Unit.
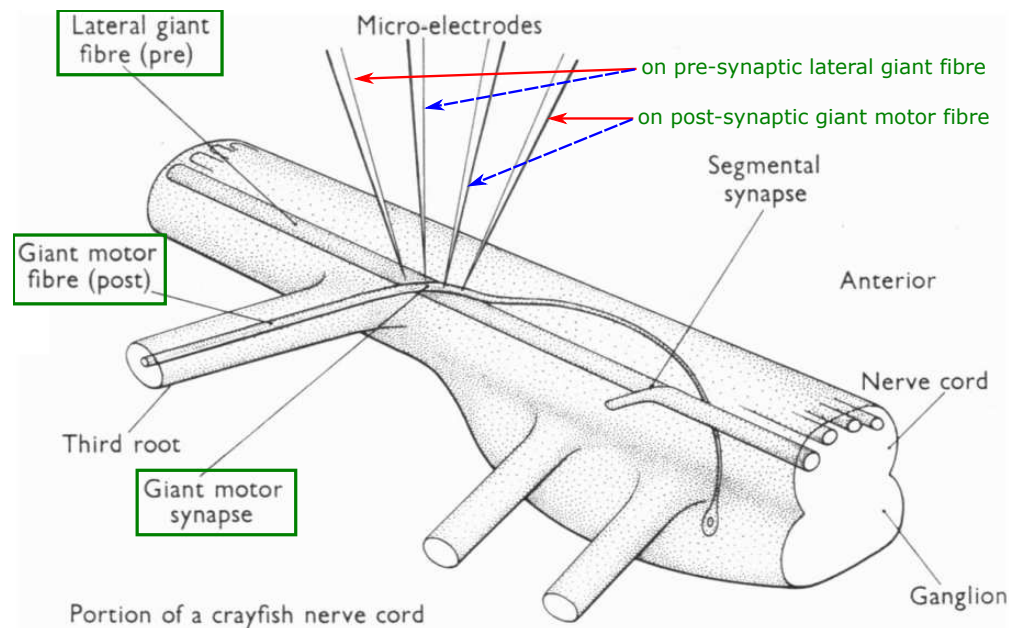
Figure 137: *Crayfish giant motor synapse* (Section 13.3.2). The (pre-synaptic) lateral giant fiber was connected to the (post-synaptic) giant motor fiber through a synapse where the two fibers cross each other at the location annotated by "Giant motor synapse" in the figure. This synapse was right underneath the giant motor fiber, at the crossing and contact point, and thus could not be seen. The two left electrodes (including the second electrode from left) were inserted in the lateral giant fiber, with the two right electrodes in the giant motor fiber. Currents were injected into the two electrodes indicated by solid red arrows, and electrical outputs recorded from the two electrodes indicated by dashed blue arrows [326]. (Figure reproduced with permission of the publisher Wiley.)

13.3.2 Rectified linear unit (ReLU)

Yoshua Bengio, the senior author of [78] and a Turing Award recipient, recounted the rise in popularity of ReLU in deep learning networks in an interview in [79]:

> "The big question was how we could train deeper networks... Then a few years later, we discovered that we didn't need these approaches [Restricted Boltzmann Machines, autoencoders] to train deep networks, we could just change the nonlinearity. One of my students was working with neuroscientists, and we thought that we should try rectified linear units (ReLUs)—we called them rectifiers in those days—because they were more biologically plausible, and this is an example of actually taking inspiration from the brain. We had previously used a sigmoid function to train neural nets, but it turned out that by using ReLUs we could suddenly train very deep nets much more easily. That was another big change that occurred around 2010 or 2011."

The student mentioned by Bengio was likely the first author of [113]; see also the earlier Section 4.4.2 on activation functions.

We were aware of Ref. [32] appearing in year 2000—in which a spatially-discrete, temperally-continuous recurrent neural network was used with a rectified linear function, as expressed in Eq. (510)—through Ref. [36]. On the other hand, prior its introduction in deep neural networks, rectified linear unit had been used in neuroscience since at least 1995, but [327] was a book, as cited in [113]. Research results published in papers would appear in book form several years later:
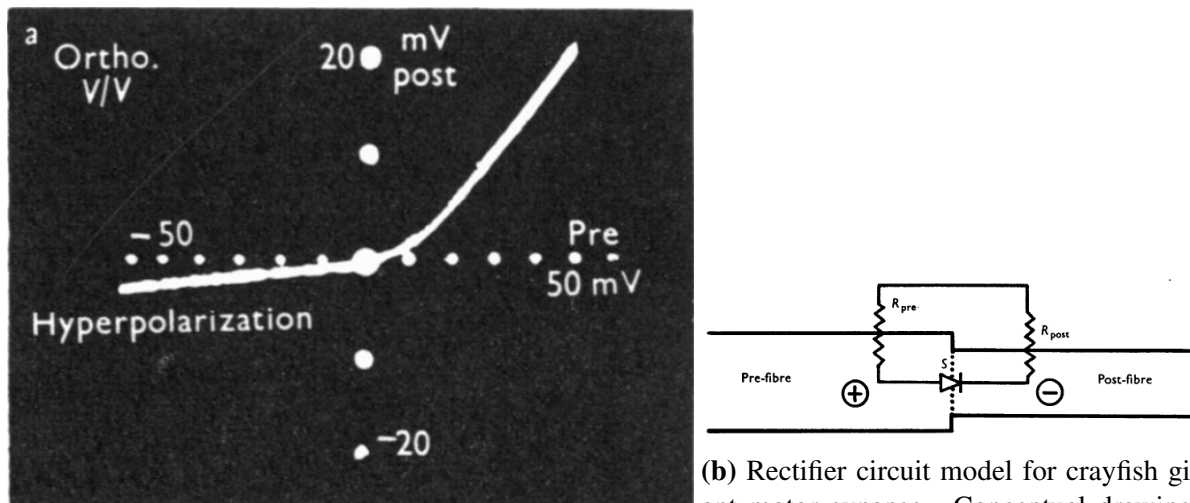
(a) Post-synaptic voltage vs pre-synaptic voltage. Experimental results indicated that currents flow from the pre-synaptic lateral giant fiber, through the synapse, into the post-synaptic giant motor fiber essentially in one direction, with a small leakage when polarity was reversed.

(b) Rectifier circuit model for crayfish giant motor synapse. Conceptual drawing, where the vertical dotted line represents the synapse. The large tube on the left represents the pre-synaptic lateral giant fiber, the smaller tube on the right the post-synaptic giant motor fiber.

Figure 138: *Crayfish Giant Motor Synapse* (Section 13.3.2). The response in SubFigure (a) is similar to that of a rectifier circuit with leaky diode in Figure 25 and Figure 29 [red curve $(-V = V_D + V_R)$ in SubFigure (b)] [326]. (Figure reproduced with permission of the publisher Wiley.)

" The current and third wave, deep learning, started around 2006 (Hinton et al., 2006; Bengio et al., 2007; Ranzato et al., 2007a) and is just now appearing in book form as of 2016. The other two waves [cybernetics and connectionism] similarly appeared in book form much later than the corresponding scientific activity occurred" [78], p. 13.

Another clue that the rectified linear function was a well-known, well accepted concept—similar to the relation $Kd = F$ in the finite element method—is that the authors of [32] did not provide any reference to their own important Eq. (1), which is reproduced in Eq. (510), as if it was already obvious to anyone in neuroscience.

Indeed, more than sixty years ago, in a series of papers [58] [326] [59], Furshpan & Potter established that current flows through a crayfish neuron synapse (Figure 136 and Figure 137) in essentially one direction, thus deducing that the synapse can be modeled as a rectifier, diode in series with resistance, as shown in Figure 138.

13.3.3 New active functions

"The design of hidden units is an extremely active area of research and does not yet have many definitive guiding theoretical principles," [78], p. 186. Indeed, the Swish activation function in Figure 139 of the form

$$a(x) = x \cdot \mathfrak{s}(\beta x) = \frac{x}{1 + \exp(-\beta x)} , \tag{517}$$

with $\mathfrak{s}(\cdot)$ being the logistic sigmoid given in Figure 30 and in Eq. (113), was found in [36] to outperform the rectified linear unit (ReLU) in a number of benchmark tests.

On the other hand, it would be hard to beat the efficiency of the rectified linear function in both evaluating the weighted combination of inputs $z^{(\ell)}$ of layer (ℓ) and in computing the gradient with the first
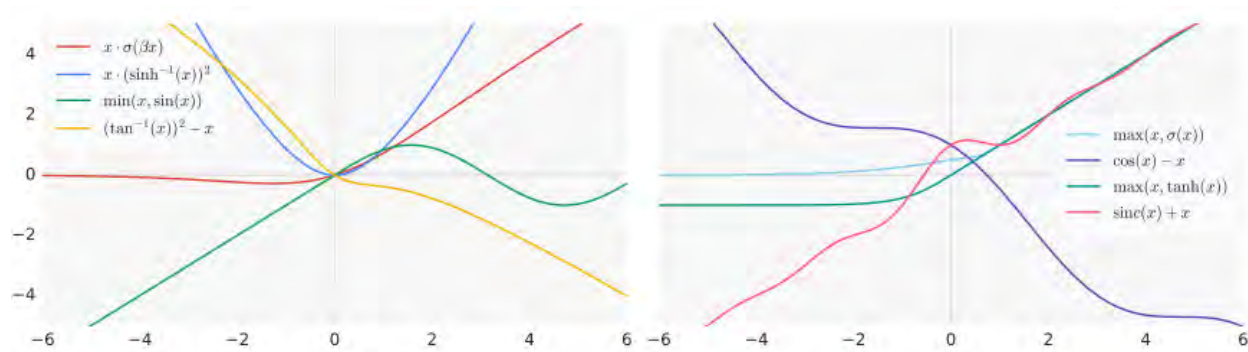
Figure 139: *Swish function* (Section 13.3.3) $x \cdot \mathfrak{s}(\beta x)$, with $\mathfrak{s}(\cdot)$ being the logistic sigmoid in Figure 30, and other activation functions [36]. (Figure reproduced with permission of the authors.)

derivative of ReLU being the Heaviside function; see Figure 24.

A zoo of activation functions is provided in "Activation function", Wikipedia, version 22:46, 24 November 2018 and the more recent version 06:30, 20 July 2022, in which several active functions had been removed, e.g., the "Square Nonlinearity (SQNL)"[321] listed in the 2018 version of this zoo.

13.4 Back-propagation, automatic differentiation

"At its core, backpropagation [Section 5] is simply an efficient and exact method for calculating all the derivatives of a single target quantity (such as pattern classification error) with respect to a large set of input quantities (such as the parameters or weights in a classification rule)" [328].

In a survey on automatic differentiation in [329], it was stated that: "in simplest terms, backpropagation models learning as gradient descent in neural network weight space, looking for the minima of an objective function."

Such statement identified back-propagation with an optimization method by gradient descent. But according to the authors of [78], p. 198, "back-propagation is often misunderstood as meaning the whole learning algorithm for multilayer neural networks", and clearly distinguish back-propagation *only* as a method to compute the gradient of the cost function with respect to the parameters, while another algorithm, such as stochastic gradient descent, is used to perform the learning using this gradient, where performing "learning" meant network training, i.e., find the parameters that minimize the cost function, which "typically includes a performance measure evaluated on the entire training set as well as additional regularization terms."[322]

According to [329], automatic differentiation, or in short "autodiff", is "a family of techniques similar to but more general than backpropagation for efficiently and accurately evaluating derivatives of numeric functions expressed as computer programs."

13.4.1 Back-propagation

In an interview published in 2018 [79], Hinton confirmed that backpropagation was independently invented by many people before his own 1986 paper [22]. Here we focus on information that is not found in the review of backpropagation in [12].

For example, the success reported in [22] laid not in backpropagation itself, but in its use in psychology:

---

[321] The "Square Nonlinearity (SQNL)" activation, having a shape similar to that of the hyperbolic tangent function, appeared in the article "Activation function" for the last time in version 22:00, 17 March 2021, and was was removed from the table of activation functions starting from version 18:13, 11 April 2021 with the comment "Remove SQLU since it has 0 citations; it needs to be broadly adopted to be in this list; Remove SQNL (also from the same author, and this also does not have broad adoption)"; see the article History.

[322] See [78], Chap. 8, "Optimization for training deep models", p. 267.

"Back in the mid-1980s, when computers were very slow, I used a simple example where you would have a family tree, and I would tell you about relationships within that family tree. I would tell you things like Charlotte's mother is Victoria, so I would say Charlotte and mother, and the correct answer is Victoria. I would also say Charlotte and father, and the correct answer is James. Once I've said those two things, because it's a very regular family tree with no divorces, you could use conventional AI to infer using your knowledge of family relations that Victoria must be the spouse of James because Victoria is Charlotte's mother and James is Charlotte's father. The neural net could infer that too, but it didn't do it by using rules of inference, it did it by learning a bunch of features for each person. Victoria and Charlotte would both be a bunch of separate features, and then by using interactions between those vectors of features, that would cause the output to be the features for the correct person. From the features for Charlotte and from the features for mother, it could derive the features for Victoria, and when you trained it, it would learn to do that. The most exciting thing was that for these different words, it would learn these feature vectors, and it was learning distributed representations of words." [79]

For psychologists, "a learning algorithm that could learn representations of things was a big breakthrough," and Hinton's contribution in [22] was to show that "backpropagation would learn these distributed representations, and that was what was interesting to psychologists, and eventually, to AI people." But backpropagation lost ground to other technologies in machine learning:

"In the early 1990s, ... the support vector machine did better at recognizing handwritten digits than backpropagation, and handwritten digits had been a classic example of backpropagation doing something really well. Because of that, the machine learning community really lost interest in backpropagation" [79].[323]

Despite such setback, psychologists still considered backpropagation as an interesting approach, and continued to work with this method:

There is "a distinction between AI and machine learning on the one hand, and psychology on the other hand. Once backpropagation became popular in 1986, a lot of psychologists got interested in it, and they didn't really lose their interest in it, they kept believing that it was an interesting algorithm, maybe not what the brain did, but an interesting way of developing representations" [79].

The 2015 review paper [12] referred to Werbos' 1974 PhD dissertation for a preliminary discussion of backpropagation (BP),

"Efficient BP was soon explicitly used to minimize cost functions by adapting control parameters (weights) (Dreyfus, 1973). Compare some preliminary, NN-specific discussion (Werbos, 1974, Section 5.5.1), a method for multilayer threshold NNs (Bobrowski, 1978), and a computer program for automatically deriving and implementing BP for given differentiable systems (Speelpenning, 1980)."

and explicitly attributed to Werbos early applications of backpropagation in neural networks (NN):

---

[323] See Footnote 31 on how research on kernel methods (Section 8) for Support Vector Machines have been recently used in connection with networks with infinite width to understand how deep learning works (Section 14.2).

> "To my knowledge, the first NN-specific application of efficient backpropagation was described in 1981 (Werbos, 1981, 2006). Related work was published several years later (LeCun, 1985, 1988; Parker, 1985). A paper of 1986 significantly contributed to the popularization of BP for NNs (Rumelhart, Hinton, & Williams, 1986), experimentally demonstrating the emergence of useful internal representations in hidden layers."

See also [112] [328] [330]. The 1986 paper mentioned above was [22].

13.4.2 Automatic differentiation

The authors of [78], p. 214, wrote of backprop as a particular case of automatic differentiation (AD):

> "The deep learning community has been somewhat isolated from the broader computer science community and has largely developed its own cultural attitudes concerning how to perform differentiation. More generally, the field of **automatic differentiation** is concerned with how to compute derivatives algorithmically. The back-propagation algorithm described here is only one approach to automatic differentiation. It is a special case of a broader class of techniques called **reverse mode accumulation**."

Let's decode what was said above. The deep learning community was isolated because it was not in the mainstream of computer science research during the last AI winter, as Hinton described in an interview published in 2018 [79]:

> "This was at a time when all of us would have been a bit isolated in a fairly hostile environment— the environment for deep learning was fairly hostile until quite recently—it was very helpful to have this funding that allowed us to spend quite a lot of time with each other in small meetings, where we could really share unpublished ideas."

If Hinton did not move from Carnegie Mellon University in the US to the University of Toronto in Canada, it would be necessary for him to change research topic to get funding, AI winter would last longer, and he may not get the Turing Award along with LeCun and Bengio [331]:

> "The Turing Award, which was introduced in 1966, is often called the Nobel Prize of computing, and it includes a $1 million prize, which the three scientists will share."

A recent review of AD is given in [329], where backprop was described as a particular case of AD, known as "reverse mode AD"; see also [12].[324]

13.5 Resurgence of AI and current state

The success of deep neural networks in the ImageNet competitions since 2012, particularly when they surpassed human-level performance in 2015 (See Figure 3 and Section 5.3.3 on Parametric ReLU), was preceded by their success in speech recognition, as recounted by Hinton in a 2018 interview [79]:

> "For computer vision, 2012 was the inflection point. For speech, the inflection point was a few years earlier. Two different graduate students at Toronto showed in 2009 that you could make a better speech recognizer using deep learning. They went as interns to IBM and Microsoft, and

---

[324] We only want to point out the connection between backprop and AD, together with a recent review paper on AD, but will not review AD itself here.

a third student took their system to Google. The basic system that they had built was developed further, and over the next few years, all these companies' labs converted to doing speech recognition using neural nets. Many of the best people in speech recognition had switched to believing in neural networks before 2012, but the big public impact was in 2012, when the vision community, almost overnight, got turned on its head and this crazy approach turned out to win."

The mentioned 2009 breakthrough of applying deep learning to speech recognition did not receive much of the non-technical press as the 2012 breakthrough in computer vision (e.g., [75] [74]), and was thus not popularly known, except inside the deep-learning community.

Deep learning is being developed and used to guide consumers in nutrition [332]:

"Using machine learning, a subtype of artificial intelligence, the billions of data points were analyzed to see what drove the glucose response to specific foods for each individual. In that way, an algorithm was built without the biases of the scientists.

There are other efforts underway in the field as well. In some continuing nutrition studies, smartphone photos of participants' plates of food are being processed by deep learning, another subtype of A.I., to accurately determine what they are eating. This avoids the hassle of manually logging in the data and the use of unreliable food diaries (as long as participants remember to take the picture).

But that is a single type of data. What we really need to do is pull in multiple types of data—activity, sleep, level of stress, medications, genome, microbiome and glucose—from multiple devices, like skin patches and smartwatches. With advanced algorithms, this is eminently doable. In the next few years, you could have a virtual health coach that is deep learning about your relevant health metrics and providing you with customized dietary recommendations."

13.5.1 COVID-19 machine-learning diagnostics and prognostics

While it is not possible to review the vast number of papers on deep learning, it would be an important omission if we did not mention a most urgent issue of our times,[325] the COVID-19 (COronaVIrus Disease 2019) pandemic, and how deep learning could help in the diagnostics and prognostics of Covid-19.

Some reviews of Covid-19 models and software. The following sweeping assertion was made in a 2021 MIT Technology Review article titled "Hundreds of AI tools have been built to catch covid. None of them helped" [334], based on two 2021 papers that reviewed and appraised the validity and usefulness of Covid-19 models for diagnostics (i.e., detecting Covid-19 infection) and for prognostics (i.e., forecasting the course of Covid-19 in patients) [335] and [336]:

"The clear consensus was that AI tools had made little, if any, impact in the fight against covid."

A large collection of 37,421 titles (published and preprint reports) on Covid-19 models up to July 2020 were examined in [335], where only 169 studies describing 232 prediction models were selected based on CHARMS (CHecklist for critical Appraisal and data extraction for Systematic Reviews of prediction Modeling Studies) [337] for detailed analysis, with the risk of bias assessed using PROBAST (Pediction model Risk Of Bias ASsessment Tool) [338]. A follow-up study [336] examined 2,215 titles up to Oct 2020, using the same methodology as in [335] with the added requirement of "sufficiently documented

---

[325] As of 2020.12.18, the COVID-19 pandemic was still raging across the entire United States.
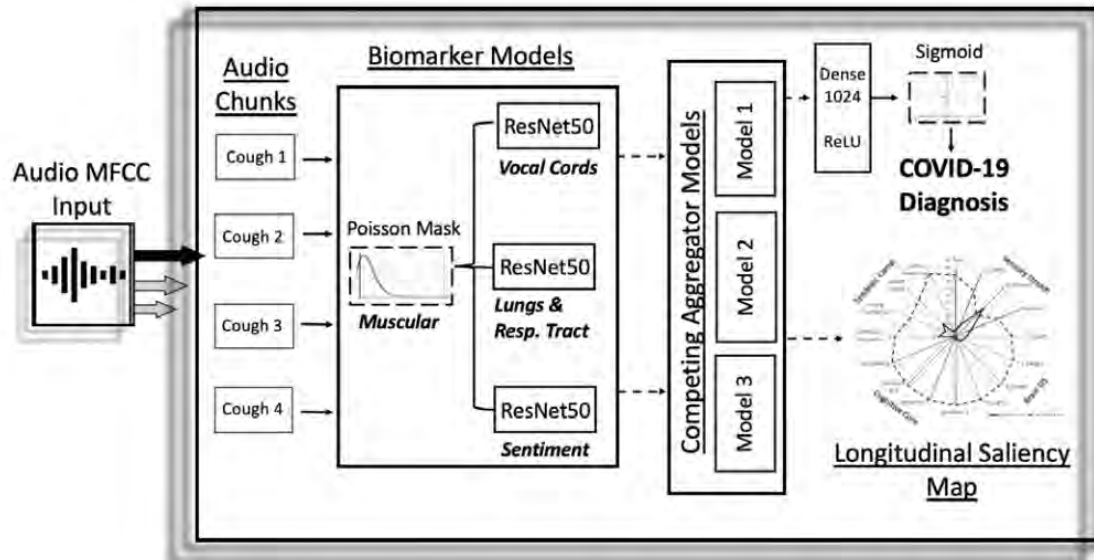
Figure 140: *MIT COVID-19 diagnosis by cough recordings.* Machine learning architecture. Audio Mel Frequency Cepstrum Coefficients (MFCC) as input. Each cough signal is split into 6 audio chunks, processed by the MFCC package, then passed through the Biomarker 1 to check on muscular degradation. The output of Biomarker 1 is input into each of the three Convolutional Neural Networks (CNNs), representing Biomarker 2 (Vocal cords), Biomarker 3 (Lungs & Respiratory Tract), Biomarker 4 (Sentiment). The outputs of these CNNs are concatenated and "pooled" together to serve as (1) input for "Competing Aggregator Models" to produce a "longitudinal saliency map", and as (2) input for a deep and dense network with ReLU activation, followed by a "binary dense layer" with sigmoid activation to produce Covid-19 diagnosis. [333]. (CC By 4.0)

methodologies", to narrow down to 62 titles for review "in most details". In the words of the lead developer of PROBAST, "unfortunately" journals outside the medical field were not included since it would be a "surprise" that "the reporting and conduct of AI health models is better outside the medical literature".[326]

Covid-19 diagnosis from cough recordings. MIT researchers developed a cough-test smartphone app that diagnoses Covid-19 from cough recordings [333], and claimed that their app achieved excellent results:[327]

> "When validated with subjects diagnosed using an official test, the model achieves COVID-19 sensitivity of 98.5% with a specificity of 94.2% (AUC: 0.97). For asymptomatic subjects it achieves sensitivity of 100% with a specificity of 83.2%." [333].

making one wondered why it had not been made available for use by everyone, since "These inventions could help our coronavirus crisis now. But delays mean they may not be adopted until the worst of the

---

[326] Private communication with Karel (Carl) Moons on 2021 Oct 28. In other words, only medical journals included in PROBAST would report Covid-19 models that cannot be beaten by models reported in non-medical journals, such as in [333], which was indeed not "fit for clinical use" to use the same phrase in [334].

[327] "In medical diagnosis, test sensitivity is the ability of a test to correctly identify those with the disease (true positive rate), whereas test specificity is the ability of the test to correctly identify those without the disease (true negative rate)." See "Sensitivity and specificity", Wikipedia version 02:21, 22 February 2021. For the definition of "AUC" (Area Under the ROC Curve), with "ROC" abbreviating for "Receiver Operating characteristic Curve", see "Classification: ROC Curve and AUC", in "Machine Learning Crash Course", Website. Internet archive.

pandemic is behind us" [339].[328]

Unfortunately, we suspected that the model in [333] was also "not fit for clinical use" as described in [334], because it has not been put to use in the real world as of 2022 Jan 12 (we were still spitting saliva into a tube instead of coughing into our phone). In addition, despite being contacted three times regarding the lack of transparency in the description of the model in [333], in particular the "Competing Aggregator Models" in Figure 140, the authors of [333] did not respond to our repeated inquiries, confirming the criticism described in Section 14.7 on "Lack of transparency and irreproducibility of results" of AI models.

Our suspicion was confirmed when we found the critical review paper [340], in which the pitfalls of the model in [333], among other cough audio models, were pointed out, with the single most important question being: Were the audio representations in these machine-learning models, even though correlated with Covid-19 in their respective datasets, the true audio biomarkers originated from Covid-19? The seven grains of salt (pitfalls) listed in [340] were:

(1) Machine-learning models did not detect Covid-19, but only distinguished between healthy people and sick people, a not so useful task.

(2) Surrounding acoustic environment may introduce biases into the cough sound recordings, e.g., Covid-19 positive people tend to stay indoors, and Covid-19 negative people outdoors.

(3) Participants providing coughs for the datasets may know their Covid-19 status, and that knowledge would affect their emotion, and hence the machine learning models.

(4) The machine-learning models can only be as accurate as the cough recording labels, which may not be valid since participants self reported their Covid-19 status.

(5) Most researchers, like the authors of [333], don't share codes and datasets, or even information on their method as mentioned above; see also Section 14.7 "Lack of transparency".

(6) The influence of factors such as comorbidity, ethnicity, geography, socio-economics, on Covid-19 is complex and unequal, and could introduce biases in the datasets.

(7) Lack of population control (participant identity not recorded) led to non-disjoint training set, development set, and test set.

Other Covid-19 machine-learning models. A comprehensive review of machine learning for Covid-19 diagnosis based on medical-data collection, preprocessing of medical images, whose features are extracted, and classified is provided in [341], where methods based on cough sound recordings were not included. Seven methods were reviewed in detail: (1) transfer learning, (2) ensemble learning, (3) unsupervised learning and (4) semi-supervised learning, (5) convolutional neural networks, (6) graph neural networks, (7) explainable deep neural networks.

In [342], deep-learning methods together with transfer learning were reviewed for classification and detection of Covid-19 based on chest X-ray, computer-tomography (CT) images, and lung-ultrasound images. Also reviewed were machine-learning methods for selection of vaccine candidates, natural-language-processing methods to analyze public sentiment during the pandemic.

For multi-disease (including Covid-19) prediction, methods based on (1) logistic regression, (2) machine learning, and in particular (3) deep learning were reviewed, with difficulties encountered forming a basis for future developments pointed out, in [343].

---

[328] One author of the present article (LVQ), more than one year after the preprint of [333], still spit into a tube for Covid test instead of coughing into a phone.

Information on the collection of genes, called genotype, related to Covid-19, was predicted by searching and scoring similarities between the seed genes (obtained from prior knowledge) and candidate genes (obtained from the biomedical literature) with the goal to establish the molecular mechanism of Covid-19 [344].

In [345], the proteins associated with Covid-19 were predicted using ligand[329] designing and modecular modeling.

In [346], after evaluating various computer-science techniques using Fuzzy-Analytic Hierarchy Process integrated with the Technique for Order Performance by Similar to Ideal Solution, it was recommended to use Blockchain as the most effective technique to be used by healthcare workers to address Covid-19 problems in Saudi Arabia.

Other Covid-19 machine learning models include the use of regression algorithms for real-time analysis of Covid-19 pandemic [347], forecasting the number of infected people using the logistic growth curve and the Gompertz growth curve [348], a generalization of the SEIR[330] model and logistic regression for forecasting [349].

13.5.2 Additional applications of deep learning

The use of deep learning as one of several machine-learning techniques for Covid-19 diagnosis was reviewed in [341] [342] [343], as mentioned above.

By growing and pruning deep learning neural networks (DNNs), optimal parameters, such as number of hidden layers, number of neurons, and types of activation functions, were obtained for the diagnosis of Parkinson's disease, with 99.34% accuracy on test data, compared to previous DNNs using specific or random number of hidden layers and neurons [350].

In [351], a deep residual network, with gridded interpolation and Swish activation function (see Section 13.3.3), was constructed to generate a single high-resolution image from many low-resolution images obtained from Fundus Fluorescein Angiography (FFA),[331] resulting in "superior performance metrics and computational time."

Going beyond the use of Proper Orthogonal Decomposition and Generalized Falk Method [352], a hierarchichal deep-learning neural network was proposed in [353] to be used with the Proper Generalized Decomposition as a model-order-reduction method applied to finite element models.

To develop high precision model to forecast wind speed and wind power, which depend on the conditions of the nearby "atmospheric pressure, temperature, roughness, and obstacles", the authors of [354] applied "deep learning, reinforcement learning and transfer learning." The challenges in this area are the randomness, the instantaneity, and the seasonal characteristics of wind and the atmosphere.

Self-driving cars must deal with a large variety of real scenarios and of real behaviors, which deep-learning perception-action models should learn to become robust. But due to a limition of the data, it was proposed in [355] to use a new image style transfer method to generate more varieties in data by modifying texture, contrast ratio and image color, and then extended to scenarios that were unobserved before.

---

[329] A ligand is "usually a molecule which produces a signal by binding to a site on a target protein,' see "Ligand (biochemistry)", Wikipedia, version 11:08, 8 December 2021.

[330] SEIR = Susceptible, Exposed, Infectious, Recovered; see "Compartmental models in epidemiology", Wikipedia, version 15:44, 19 February 2022.

[331] Fundus is "the interior surface of the eye opposite the lens and includes the retina, optic disc, macula, fovea, and posterior pole" (Wikipedia, version 02:49, 7 January 2020). Fluorescein is an organic compound and fluorescent dye (Wikipedia, version 19:51, 6 January 2022). Angiography (angio- "blood vessel" + graphy "write, record", Wikipedia, version 10:19, 2 February 2022) is a medical procedure to visualize the flow of blood (or other biological fluid) by injecting a dye and by using a special camera.

Other applications of deep learning include a real-time maskless-face detector using deep residual networks [356], topology optimization with embedded physical law and physical constraints [357], prediction of stress-strain relations in granular materials from triaxial test results [358], surrogate model for flight-load analysis [359], classification of domestic refuse in medical institutions based on transfer learning and convolutional neural network [360], convolutional neural network for arrhythmia diagnosis [361], e-commerce dynamic pricing by deep reinforcement learning [362], network intrusion detection [363], road pavement distress detection for smart maintenance [364], traffic flow statistics [365], multi-view gait recognition using deep CNN and channel attention mechanism [366], mortality risk assessment of ICU patients [367], stereo matching method based on space-aware network model to reduce the limitation of GPU RAM [368], air quality forecasting in Internet of Things [369], analysis of cardiac disease abnormal ECG signals [370], detection of mechanical parts (nuts, bolts, gaskets, etc.) by machine vision [371], asphalt road crack detection [372], steel commondity selection using bidirectional encoder representations from transformers (BERT) [373], short-term traffic flow prediction using LSTM-XGBoost combination model [374], emotion analysis based on multi-channel CNN in social networks [375].
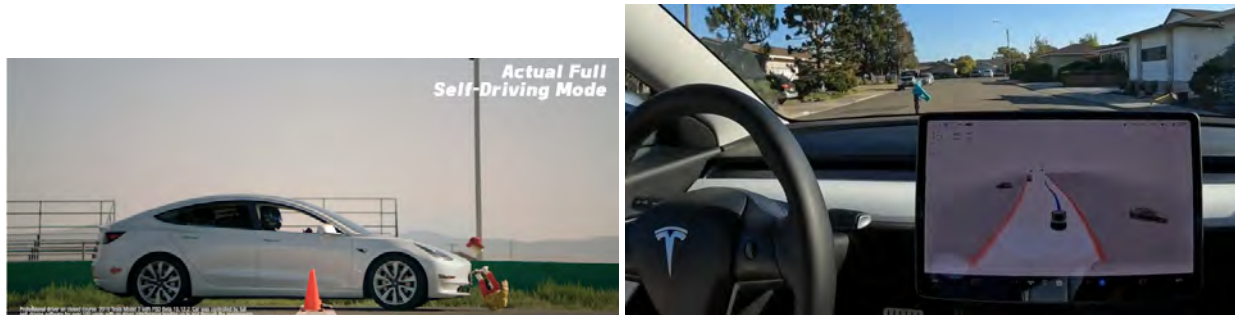
Figure 141: *Tesla Full-Self-Driving (FSD) controversy* (Section 14.1). *Left:* Tesla in FSD mode hit a child-size mannequin, repeatedly in safety tests by The Dawn Project, a software competitor to Tesla, 2022.08.09 [376] [377]. *Right:* Tesla in FSD mode went around a child-size mannequin at 15 mph in a residential area, 2022.08.14 [378] [379]. Would a prudent driver stop completely, waiting for the kid to move out of the road, before proceeding forward? The driver, a Tesla investor, did not use his own child, indicating that his maneuver was not safe.

14 Closure: Limitations and danger of AI

A goal of the present review paper is to bring first-time learners from the beginning level to as close as possible the research frontier in deep learning, with particular connection to, and application in, computational mechanics.

As concluding remarks, we collect here some known limitations and danger of AI in general, and deep learning in particular. As Hinton pointed out himself a limitation of generalization of deep learning [383]:

> "If a neural network is trained on images that show a coffee cup only from a side, for example,
> it is unlikely to recognize a coffee cup turned upside down."

14.1 Driverless cars, crewless ships, "not any time soon"

In 2016, the former U.S. secretary of transportation, Anthony Foxx, described a rosy future just five years down the road: "By 2021, we will see autonomous vehicles in operation across the country in ways that we [only] imagine today" [384].

On 2022.06.09, UPI reported that "Automaker Hyundai and South Korean officials launched a trial service of self-driving taxis in the busy Seoul neighborhood of Gangnam," an event described as "the latest step

Figure 142: *Tesla Full-Self-Driving (FSD) controversy* (Section 14.1). The Tesla was about to run down the child-size mannequin at 23 mph, hitting it at 24 mph. The driver did not hold on, but only kept his hands close, to the driving wheel for safety, and did not put his foot on the accelerator. There were no cones on both sides of the road, and there was room to go around the mannequin. The weather was clear, sunny. The mannequin wore a bright safety jacket. Visibility was excellent, 2022.08.15 [380] [381].

forward in the country's efforts to make autonomous vehicles an everyday reality. The new service, called RoboRide, features Hyundai Ioniq 5 electric cars equipped with Level 4 autonomous driving capabilities. The technology allows the taxis to move independently in real-life traffic without the need for human control, although a safety driver will remain in the car" [385]. According to Huyndai, the safety driver "only intervenes under limited conditions," which were explicitly not specified to the public, whereas the car itself would "perceive, make decisions and control its own driving status."

But what is "Level 4 autonomous driving"? Let's look at the startup autononous-driving company Waymo. Their Level 4 consists of "mapping the territory in a granular fashion (including lane markers, traffic signs and lights, curbs, and crosswalks). The solution incorporates both GPS signals and real-time sensor data to always determine the vehicle's exact location. Further, the system relies on more than 20 million miles of real-world driving and more than 20 billion miles in simulation, to allow the Waymo Driver to anticipate what other road users, pedestrians, or other objects might do" [386].

Yet Level 4 is still far from Level 5, for which "vehicles are fully automated with no need for the driver to do anything but set the destination and ride along. They can drive themselves anywhere under any conditions, safely" [387], and would still be many years later away [386].

Indeed, exactly two months after Huyndai's announcement of their Level 4 test pilot program, on 2022.08.09, *The Guardian* reported that in a series of safety tests, a "professional test driver using Tesla's Full Self-Driving mode repeatedly hit a child-sized mannequin in its path" [377]; Figure 141, left. "It's a lethal threat to all Americans, putting children at great risk in communities across the country," warned The Dawn Project's founder, Dan O'Dowd, who described the test results as "deeply disturbing," as the vehicle tended to "mow down children at crossroads," and who argued for prohibiting Tesla vehicles from running in the street until Tesla self driving software could be proven safe.

The Dawn Project test results were contested by a Tesla investor, who posted a video on 2022.08.14 to prove that the Tesla Full-Self-Driving (FSD) system worked as advertised (Figure 141, right). The next day, 2022.08.15, Dan O'Dowd posted a video proving that the Tesla under FSD mode ran over a child-size mannequin at 24 mph in clear weather, with excellent visibility, no cones on either side of the Tesla, and
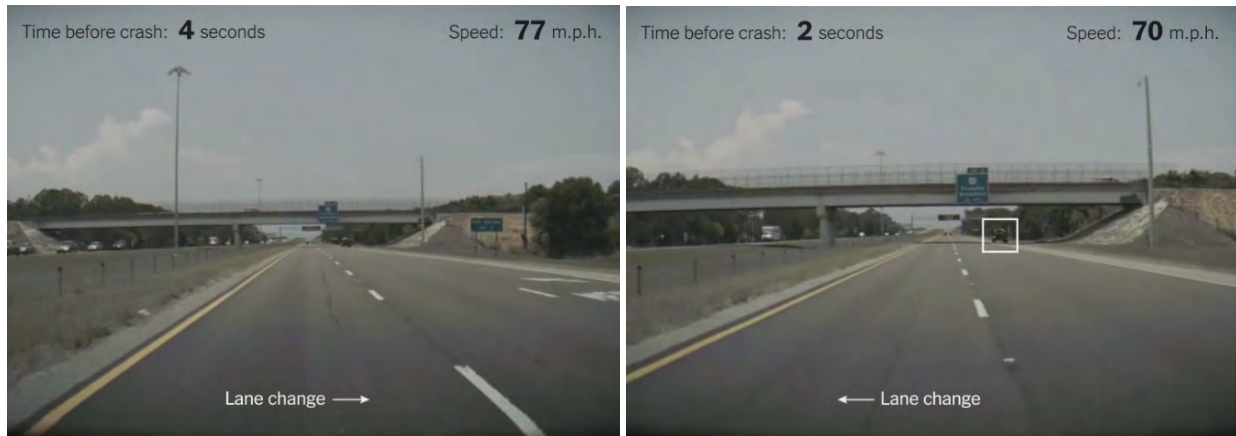
Figure 143: *Tesla crash* (Section 14.1). July 2020. *Left:* "Less than a half-second after [the Tesla driver] flipped on her turn signal, Autopilot started moving the car into the right lane and gradually slowed, video and sensor data showed." *Right:* "Halfway through, the Tesla sensed an obstruction—possibly a truck stopped on the side of the road—and paused its lane change. The car then veered left and decelerated rapidly" [382]. See also Figures 144, 145, 146. (Data and video provided by QuantivRisk.)

without the driver pressing his foot on the accelerator (Figure 142).

"In June [2022], the National Highway Traffic Safety Administration (NHTSA), said it was expanding an investigation into 830,000 Tesla cars across all four current model lines. The expansion came after analysis of a number of accidents revealed patterns in the car's performance and driver behavior" [377]. "Since 2016, the agency has investigated 30 crashes involving Teslas equipped with automated driving systems, 19 of them fatal. NHTSA's Office of Defects Investigation is also looking at the company's autopilot technology in at least 11 crashes where Teslas hit emergency vehicles."

In 2019, it was reported that several car executives thought that driveless cars were still several years in the future because of the difficulty in anticipating human behavior [388]. The progress of Huyndai's driveless taxis has not solved the challenge of dealing with human behavior, as there was still a need for a "safety driver."

"On [2022] May 6, Lyft, the ride-sharing service that competes with Uber sold its Level 5 division, an autonomous-vehicle unit, to Woven Planet, a Toyota subsidiary. After four years of research and development, the company seems to realize that autonomous driving is a tough nut to crack—much tougher than the team had anticipated.

"Uber came to the same conclusion, but even earlier, in December. The company sold Advanced Technologies Group, its self-driving unit, to Aurora Innovation, citing high costs and more than 30 crashes, culminating in a fatality as the reason for cutting its losses.

"Finally, several smaller companies, including Zoox, a robo-taxi company; Ike, an autonomous-trucking startup; and Voyage, a self-driving startup; have also passed the torch to companies with bigger budgets" [384].

"Those startups, like many in the industry, have underestimated the sheer difficulty of "leveling up" vehicle autonomy to the fabled Level 5 (full driving automation, no human required)" [384].

On top of the difficulty in addressing human behavior, there were other problems, perhaps in principle less challenging, so we thought, as reported in [386]: "widespread adoption of autonomous driving is still years away from becoming a reality, largely due to the challenges involved with the development of accurate sensors and cameras, as well as the refinement of algorithms that act upon the data captured by these sensors.
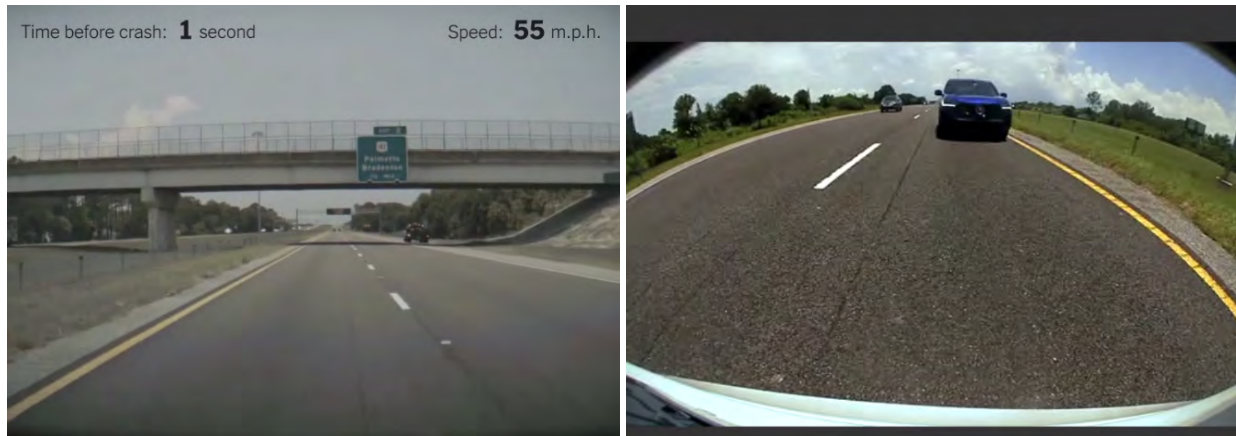
Figure 144: *Tesla crash* (Section 14.1). July 2020. "Less than a second after the Tesla has slowed to roughly 55 m.p.h. [*Left*], its rear camera shows a car rapidly approaching [*Right*]" [382]. There were no moving cars on both lanes in front of the Tesla for a long distance ahead (perhaps a quarter of a mile). See also Figures 143, 145, 146. (Data and video provided by QuantivRisk.)

"This process is extremely data-intensive, given the large variety of potential objects that could be encountered, as well as the near-infinite ways objects can move or react to stimuli (for example, road signs may not be accurately identified due to lighting conditions, glare, or shadows, and animals and people do not all respond the same way when a car is hurtling toward them).

"Algorithms in use still have difficulty identifying objects in real-world scenarios; in one accident involving a Tesla Model X, the vehicle's sensing cameras failed to identify a truck's white side against a brightly lit sky."

In addition to Figure 141, another example was a Tesla crash in July 2020 in clear, sunny weather, with little clouds, as shown in Figures 143, 144, 145, 146. The self-driving system could not detect that a static truck was parked on the side of a highway, and due to the foward and changing-lane motion of the Tesla, the software could have thought that it was running into the truck, and veered left while rapidly decelerating to avoid collision with the truck. As a result, the Tesla was rear-ended by another fast coming car from behind on its left side [382].

"Pony.ai is the latest autonomous car company to make headlines for the wrong reasons. It has just lost its permit to test its fleet of autonomous vehicles in California over concerns about the driving record of the safety drivers it employs. It's a big blow for the company, and highlights the interesting spot the autonomous car industry is in right now. After a few years of very bad publicity, a number of companies have made real progress in getting self-driving cars on the road" [389].

The 2022 article "I'm the Operator': The Aftermath of a Self-Driving Tragedy" [390] described these "few years of very bad publicity" in stunning, tragic details about an Uber autonomous-vehicle operator, Rafela Vasquez, who did not take over the control of the vehicle in time, and killed a jaywalking pedestrian.

The classification software of the Uber autonomous driving system could not recognize the pedestrian, but vacillated between a "vehicle", then "other", then a "bicycle" [390].

"At 2.6 seconds from the object, the system identified it as 'bicycle.' At 1.5 seconds, it switched back to considering it 'other.' Then back to 'bicycle' again. The system generated a plan to try to steer around whatever it was, but decided it couldn't. Then, at 0.2 seconds to impact, the car let out a sound to alert Vasquez that the vehicle was going to slow down. At two-hundredths of a second before impact, traveling at 39 mph, Vasquez grabbed the steering wheel, which wrested the car out of autonomy and into manual

Figure 145: *Tesla crash* (Section 14.1). July 2020. The fast-coming blue car rear-ended the Tesla, indented its own front bumper, with flying broken glass (or clear plastic) cover shards captured by the Tesla rear camera [382]. See also Figures 143, 144, 146. (Data and video provided by QuantivRisk.)

mode. It was too late. The smashed bike scraped a 25-foot wake on the pavement. A person lay crumpled in the road" [390].

The operator training program manager said "I felt shame when I heard of a lone frontline employee has been singled out to be charged of negligent homicide with a dangerous instrument. We owed Rafaela better oversight and support. We also put her in a tough position." Another program manager said "You can't put the blame on just that one person. I mean, it's absurd. Uber had to know this would happen. We get distracted in *regular* driving. It's not like somebody got into their car and decided to run into someone. They were working within a framework. And that framework created the conditions that allowed that to happen." [390].

After the above-mentioned fatality caused by an Uber autonomous car with a single operator in it, "many companies temporarily took their cars off the road, and after it was revealed that only one technician was inside the Uber car, most companies resolved to keep two people in their test vehicles at all times" [391]. Having two operators in a car would help to avoid accidents, but the pandemic social-distancing rule often prevented such arrangement from happening.

"Many self-driving car companies have no revenue, and the operating costs are unusually high. Autonomous vehicle start-ups spend $1.6 million a month on average—four times the rate at financial tech or health care companies" [391].

"Companies like Uber and Lyft, worried about blowing through their cash in pursuit of autonomous technology, have tapped out. Only the deepest-pocketed outfits like Waymo, which is a subsidiary of Google's parent company, Alphabet; auto giants; and a handful of start-ups are managing to stay in the game.

"Late last month, Lyft sold its autonomous vehicle unit to a Toyota subsidiary, Woven Planet, in a deal valued at $550 million. Uber offloaded its autonomous vehicle unit to another competitor in December. And three prominent self-driving start-ups have sold themselves to companies with much bigger budgets over the past year" [392].

Figure 146: *Tesla crash* (Section 14.1). After hitting the Tesla, the blue car "spun across the highway [*Left*] and onto the far shoulder [*Right*]," as another car was coming toward on the right lane (left in photo), but still at a safe distance so not to hit it. [382]. See also Figures 143, 144, 145. (Data and video provided by QuantivRisk.)

Similar problems exist with building autonomous boats to ply the oceans without a need for a crew on board [393]:

> "When compared with autonomous cars, ships have the advantage of not having to make split-second decisions in order to avoid catastrophe. The open ocean is also free of jaywalking pedestrians, stoplights and lane boundaries. That said, robot ships share some of the problems that have bedeviled autonomous vehicles on land, namely, that they're bad at anticipating what humans will do, and have limited ability to communicate with them.

Shipping is a dangerous profession, as there were some 41 large ships lost at sea due to fires, rogue waves, or other accidents, in 2019 alone. But before an autonomous ship can reach the ocean, it must get out of port, and that remains a technical hurdle not yet overcome:

> " 'Technically, it's not possible yet to make an autonomous ship that operates safely and efficiently in crowded areas and in port areas,' says Rudy Negenborn, a professor at TU Delft who researches and designs systems for autonomous shipping.

> Makers of autonomous ships handle these problems by giving humans remote control. But what happens when the connection is lost? Satisfactory solutions to these problems have yet to arrive, adds Dr. Negenborn."

The onboard deep-learning computer vision system was trained to recognize "kayaks, canoes, Sea-Doos", but a person standing on a paddle board would look like someone walking on water to the system [393]. See also Figures 143, 144, 145, 146 on the failure of the Tesla computer vision system in detecting a parked truck on the side of a highway.

Beyond the possible lost of connection in a human remote-control ship, mechanical failure did occur, such as that happened for the Mayflower autonomous ship shown in Figure 147 [394]. Measures would have to be taken when mechanical failure happens to a crewless ship in the middle of a vast ocean.

See also the interview of S.J. Russell in [79] on the need to develop hybrid systems that have classical AI along side with deep learning, which has limitations, even though it is good at classification and perception,[332] and Section 14.3 on the barrier of meaning in AI.

---

[332] S.J. Russell also appeared in the video "AI is making it easier..." mentioned at the end of this closure section.

Figure 147: *Mayflower autonomous ship* (Section 14.1) sailing from Plymouth, UK, planning to arrive at Plymouth, MA, U.S., like the original Mayflower 400 years ago, but instead arriving at Halifax, Nova Scotia, Canada, on 2022 Jun 05, due to mechanical problems [394]. (CC BY-SA 4.0, Wikipedia, version 16:43, 17 July 2022.)

14.2 Lack of understanding on why deep learning worked

Such lack of understanding is described in *The Guardian*'s Editorial on the 2019 New Year Day [67] as follows:

> "Compared with conventional computer programs, [AI that teaches itself] acts for reasons incomprehensible to the outside world. It can be trained, as a parrot can, by rewarding the desired behaviour; in fact, this describes the whole of its learning process. But it can't be consciously designed in all its details, in the way that a passenger jet can be. If an airliner crashes, it is in theory possible to reconstruct all the little steps that led to the catastrophe and to understand why each one happened, and how each led to the next. Conventional computer programs can be debugged that way. This is true even when they interact in baroquely complicated ways. But neural networks, the kind of software used in almost everything we call AI, can't even in principle be debugged that way. We know they work, and can by training encourage them to work better. But in their natural state it is quite impossible to reconstruct the process by which they reach their (largely correct) conclusions."

The 2021 breakthough in computer science, as declared by the Quanta Magazine [233], was the discovery of the connection between shallow networks with infinite width (Figure 148) and kernel machines (or methods) as a first step in trying to understand how deep-learning networks work; see Section 8 on "Kernel machines" and Footnote 31.

14.3 Barrier of meaning

Deep learning could not think like humans do, and could be easily fooled as reported in [395]:

> " Machine learning algorithms don't yet understand things the way humans do—with sometimes disastrous consequences.
>
> Even more worrisome are recent demonstrations of the vulnerability of A.I. systems to so-called adversarial examples. In these, a malevolent hacker can make specific changes to images, sound waves or text documents that while imperceptible or irrelevant to humans will cause a program to make potentially catastrophic errors.
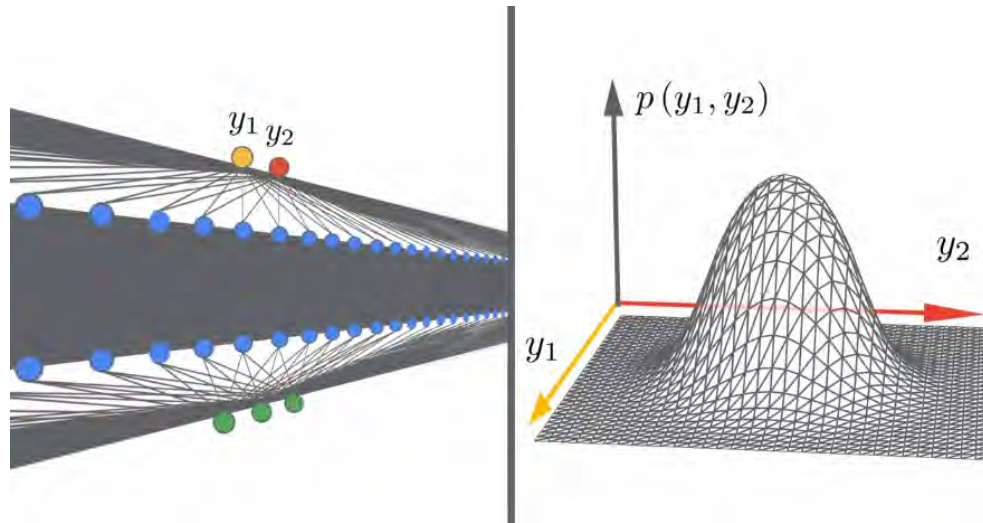
Figure 148: *Network with infinite width* (left) and Gaussian distribution (Right) (Section 6.1, 14.2). "A number of recent results have shown that DNNs that are allowed to become infinitely wide converge to another, simpler, class of models called Gaussian processes. In this limit, complicated phenomena (like Bayesian inference or gradient descent dynamics of a convolutional neural network) boil down to simple linear algebra equations. Insights from these infinitely wide networks frequently carry over to their finite counterparts. As such, infinite-width networks can be used as a lens to study deep learning, but also as useful models in their own right" [279] [231]. See Figures 60 and 61 for the motivation for networks with infinite width. (CC BY-SA 4.0, Wikipedia, version 03:51, 18 June 2022.)

The possibility of such attacks has been demonstrated in nearly every application domain of A.I., including computer vision, medical image processing, speech recognition and language processing. Numerous studies have demonstrated the ease with which hackers could, in principle, fool face- and object-recognition systems with specific minuscule changes to images, put inconspicuous stickers on a stop sign to make a self-driving car's vision system mistake it for a yield sign or modify an audio signal so that it sounds like background music to a human but instructs a Siri or Alexa system to perform a silent command.

These potential vulnerabilities illustrate the ways in which current progress in A.I. is stymied by the barrier of meaning. Anyone who works with A.I. systems knows that behind the facade of humanlike visual abilities, linguistic fluency and game-playing prowess, these programs do not—in any humanlike way—understand the inputs they process or the outputs they produce. The lack of such understanding renders these programs susceptible to unexpected errors and undetectable attacks.

As the A.I. researcher Pedro Domingos noted in his book *The Master Algorithm*, 'People worry that computers will get too smart and take over the world, but the real problem is that they're too stupid and they've already taken over the world.'

Such barrier of meaning is also a barrier for AI to tackle human controversies; see Section 14.5. See also Section 14.1 on driverless cars not coming any time soon, which is related to the above barrier of meaning.

14.4 Threat to democracy and privacy

On the 2019 new-year day, *The Guardian* [67] not only reported the most recent breakthrough in AI research on the development of AlphaZero, a software possessing superhuman performance in several "im-

mensely complex" games such as Go (see Section 13.5 on resurgence of AI and current state), they also reported another breakthrough as a more ominous warning on a "Power struggle" to preserve liberal democracies against authoritarian governments and criminals:

> "The second great development of the last year makes bad outcomes much more likely. This is the much wider availability of powerful software and hardware. Although vast quantities of data and computing power are needed to train most neural nets, once trained a net can run on very cheap and simple hardware. This is often called the democratisation of technology but it is really the anarchisation of it. Democracies have means of enforcing decisions; anarchies have no means even of making them. The spread of these powers to authoritarian governments on the one hand and criminal networks on the other poses a double challenge to liberal democracies. Technology grants us new and almost unimaginable powers but at the same time it takes away some powers, and perhaps some understanding too, that we thought we would always possess."

Nearly three years later, a report of a national poll of 2,200 adults in the U.S., released on 2021.11.15, indicated that three in four adults were concerned about the loss of privacy, "loss of trust in elections (57%), in threats to democracy (52%), and in loss of trust in institutions (56%). Additionally, 58% of respondents say it has contributed to the spread of misinformation" [396].

Figure 149: *Deepfake images* (Section 14.4.1). AI-generated portraits using Generative Adversarial Network (GAN) models. See also [397] [398], Chap. 8, "GAN Fingerprints in Face Image Synthesis." (Images from 'This Person Does Not Exist' site.)

14.4.1 Deepfakes

AI software available online helping to create videos that show someone said or did things that the person did not say or do represent a clear danger to democracy, as these deepfake videos could affect the outcome of an election, among other misdeeds, with risk to national security. Advances in machine learning have made deepfakes "ever more realistic and increasingly resistant to detection" [399]; see Figure 149. The authors of [400] concurred:

> "Deepfake videos made with artificial intelligence can be a powerful force because they make it appear that someone did or said something that they never did, altering how the viewers see politicians, corporate executives, celebrities and other public figures. The tools necessary to make these videos are available online, with some people making celebrity mashups and one app offering to insert users' faces into famous movie scenes."

To be sure, deepfakes do have benefits in education, arts, and individual autonomy [399]. In education, deepfakes could be used to provide information to students in a more interesting manner. For example,

deepfakes make it possible to "manufacture videos of historical figures speaking directly to students, giving an otherwise unappealing lecture a new lease on life". In the arts, deepfake technology allowed to resurrect long dead actors for fresh roles in new movies. An example is a recent *Star Wars* movie with the deceased actress Carrie Fisher. In helping to maintain some personal autonomy, deepfake audio technology could help restore the ability to speak for a person suffered from some form of paralysis that prevents normal speaking.

On the other hand, the authors of [399] cited a long list of harmful uses of deepfakes, from harm to individuals or organizations (e.g., exploitation, sabotage), to harm to society (e.g., distortion of democratic discourse, manipulation of elections, eroding trust in institutions, exacerbating social division, undermining public safety, undermining diplomacy, jeopardizing national security, undermining journalism, crying deepfake news as liar's dividend).[333] See also [402] [403] [404] [405].

Researchers have been in a race to develop methods to detect deepfakes, a difficult technological challenge [406]. One method is to spot the subtle characteristics of how someone spoke to provide a basis to determine whether a video was true or fake [400]. But that method was not a top-five winner of the *Deep-Fake Detection Challenge* (DFDC) [407] organized in the period 2019-2020 by "The Partnership for AI, in collaboration with large companies including Facebook, Microsoft, and Amazon," with a total prize money of one million dollars, divided among the top five winners, out of more than two thousand teams [408].

Human's ability to detect of deepfakes compared well with the "leading model," i.e., the DFCD top winner [408]. The results were "at odds with the commonly held view in media forensics that ordinary people have extremely limited ability to detect media manipulations" [408]; see Figure 150, where the width of a violin plot,[334] at a given accuracy, represents the number of participants. In Col. 2 of Figure 150, the area of the blue violin *above* the leading model accuracy of 65% represents 82% of the participants, represented by the area of the whole violin. A crowd does have a collective accuracy comparable to (or for those who viewed at least 10 videos, better than) the leading model; see Cols. 5, 6, 7 in Figure 150.

While it is difficult to detect AI deepfakes, the MIT Media Lab DeepFake detection project advised to pay attention to the following eight facial features [411]:

(1) "Face. High-end DeepFake manipulations are almost always facial transformations.

(2) "Cheeks and forehead. Does the skin appear too smooth or too wrinkly? Is the agedness of the skin similar to the agedness of the hair and eyes? DeepFakes are often incongruent on some dimensions.

(3) "Eyes and eyebrows. Do shadows appear in places that you would expect? DeepFakes often fail to fully represent the natural physics of a scene.

(4) "Glasses. Is there any glare? Is there too much glare? Does the angle of the glare change when the person moves? Once again, DeepFakes often fail to fully represent the natural physics of lighting.

(5) "Facial hair or lack thereof. Does this facial hair look real? DeepFakes might add or remove a mustache, sideburns, or beard. But, DeepFakes often fail to make facial hair transformations fully natural.

(6) "Facial moles. Does the mole look real?

(7) Eye "blinking. Does the person blink enough or too much?

(8) "Size and color of the lips. Does the size and color match the rest of the person's face?"

---

[333] Watch also Danielle Citron's 2019 TED talk "How deepfakes undermine truth and threaten democracy" [401].

[334] See the classic original paper [409], which was cited 1,554 times on Google Scholar as of 2022.08.24. See also [410] with Python code and resulting images on GitHub.
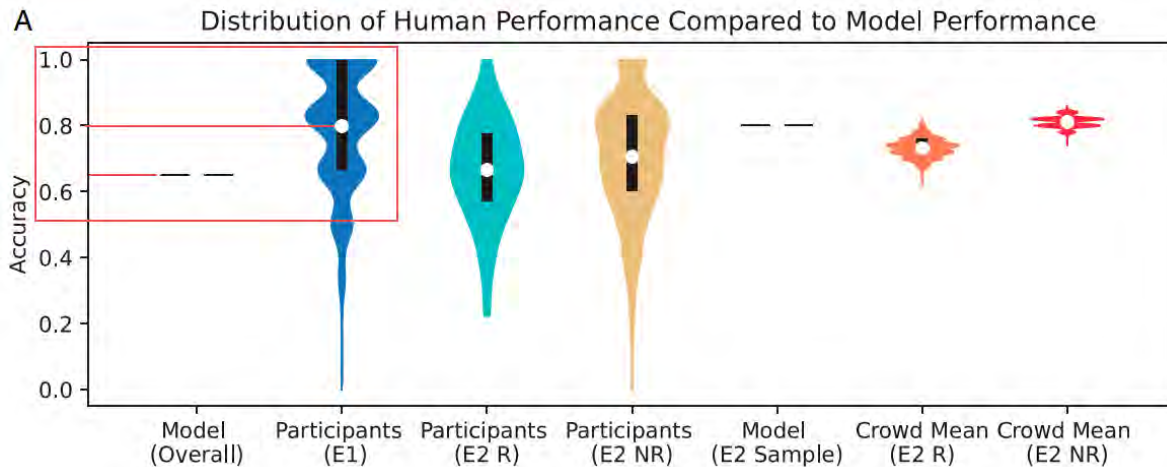
Figure 150: *DeepFake detection* (Section 14.4.1). Violin plots. • *Individual vs machine*. The leading model had an accuracy of 65% on 4,000 videos (Col. 1). In Experiment 1 (E1), 5,524 participants were asked to identify a deepfake from each of 56 pairs of videos. The participants had a mean accuracy of 80% (white dot in Col. 2), with 82% of the participants having an accuracy better than that of the leading model (65%). In Experiment 2 (E2), using a subset of randomly sampled videos, the recruited (R) participants had mean accuracy at 66% (Col. 3), the non-recruited (NR) participants at 69% (Col. 4), and leading model at 80%. • *Crowd wisdom vs machine*. Crowd mean is the average accuracy by participants for each video. R participants had a crowd-mean average accuracy at 74%, NR participants at 80%, which was the same for the leading model, and NR participants who viewed at least 10 videos at 86% [408]. (CC BY-NC-ND 4.0)

14.4.2 Facial recognition nightmare

"We're all screwed" as a Clearview AI, startup company, uses deep learning to identify faces against a large database involving more than three billions photos collected from "Facebook, Youtube, Venmo and millions of other websites" [412]. Their software "could end your ability to walk down the street anonymously, and provided it to hundreds of law enforcement agencies". More than 600 law enforcement agencies have started to use Clearview AI software to "help solve shoplifting, identity theft, credit card fraud, murder and child sexual exploitation cases". On the other hand, the tool could be abused, such as identifying "activists at a protest or an attractive stranger on the subway, revealing not just their names but where they lived, what they did and whom they knew". Some large cities such as San Francisco has banned to use of facial recognition by the police.

A breach of Clearview AI database occurred just a few weeks after the article by [412], an unforeseen, but not surprising, event [413]:

> "Clearview AI, the controversial and secretive facial recognition company, recently experienced its first major data breach—a scary prospect considering the sheer amount and scope of personal information in its database, as well as the fact that access to it is supposed to be restricted to law enforcement agencies."

The leaked documents showed that Clearview AI had a large range of customers, ranging from law-enforcement agencies (both domestic and internatinal), to large retail stores (Macy's, Best Buy, Walmart). Experts describe Clearview AI's plan to produce a publicly available face recognition app as "dangerous". So we got screwed again.

There was a documented wrongful arrest by face-recognition algorithm that demonstrated racism, i.e., a bias toward people of color [414]. A detective showed the wrongful-arrest victim a photo that was clearly not the victim, and asked "Is this you?" to which the victim replied "You think all black men look alike?"

It is well known that AI has "propensity to replicate, reinforce or amplify harmful existing social biases" [415], such as racial bias [416] among others: "An early example arose in 2015, when a software engineer pointed out that Google's image-recognition system had labeled his Black friends as 'gorillas.' Another example arose when Joy Buolamwini, an algorithmic fairness researcher at MIT, tried facial recognition on herself—and found that it wouldn't recognize her, a Black woman, until she put a white mask over her face. These examples highlighted facial recognition's failure to achieve another type of fairness: representational fairness" [417].[335]

A legal measure has been taken against gathering data for facial-recognition software. In May 2022, Clearview AI was slapped with a "$10 million for scraping UK faces from the web. That might not be the end of it"; in addition, "the firm was also ordered to delete all of the data it holds on UK citizens" [419].

There were more of such measures: "Earlier this year, Italian data protection authorities fined Clearview AI €20 million ($21 million) for breaching data protection rules. Authorities in Australia, Canada, France, and Germany have reached similar conclusions.

Even in the US, which does not have a federal data protection law, Clearview AI is facing increasing scrutiny. Earlier this month the ACLU won a major settlement that restricts Clearview from selling its database across the US to most businesses. In the state of Illinois, which has a law on biometric data, Clearview AI cannot sell access to its database to anyone, even the police, for five years" [419].

14.5 AI cannot tackle controversial human problems

If there was a barrier of meaning as described in Section 14.3, it is clear that there are many problems that AI could not be trained to solve since even humans do not agree on how to classify certain activities as offending or acceptable. It was written in [420] the following:

> "Mr. Schroepfer—or Schrep, as he is known internally—is the person at Facebook leading the efforts to build the automated tools to sort through and erase the millions of [hate-speech] posts. But the task is Sisyphean, he acknowledged over the course of three interviews recently.
>
> That's because every time Mr. Schroepfer [Facebook's Chief Technology Officer] and his more than 150 engineering specialists create A.I. solutions that flag and squelch noxious material, new and dubious posts that the A.I. systems have never seen before pop up—and are thus not caught. The task is made more difficult because "bad activity" is often in the eye of the beholder and humans, let alone machines, cannot agree on what that is.
>
> "I don't think I'm speaking out of turn to say that I've seen Schrep cry at work," said Jocelyn Goldfein, a venture capitalist at Zetta Venture Partners who worked with him at Facebook."

14.6 So what's new? Learning to think like babies

Because of AI's inability to understand (barrier of meaning) and to solve controversial human issues, a idea to tackle such problems is to start with baby steps in trying to teach AI to think like babies, as recounted by [321]:

---

[335] See also [418] on a number of relevant AI ethical issues such as: "Who bears responsibility in the event of harm resulting from the use of an AI system; How can AI systems be prevented from reflecting existing discrimination, biases and social injustices based on their training data, thereby exacerbating them; How can the privacy of people be protected, given that personal data can be collected and analysed so easily by many." Perhaps the toughest question is "Who should get to decide which moral intuitions, which values, should be embedded in algorithms?" [417].

"The problem is that these new algorithms are beginning to bump up against significant limitations. They need enormous amounts of data, only some kinds of data will do, and they're not very good at generalizing from that data. Babies seem to learn much more general and powerful kinds of knowledge than AIs do, from much less and much messier data. In fact, human babies are the best learners in the universe. How do they do it? And could we get an AI to do the same?

First, there's the issue of data. AIs need enormous amounts of it; they have to be trained on hundreds of millions of images or games.

Children, on the other hand, can learn new categories from just a small number of examples. A few storybook pictures can teach them not only about cats and dogs but jaguars and rhinos and unicorns.

AIs also need what computer scientists call "supervision." In order to learn, they must be given a label for each image they "see" or a score for each move in a game. Baby data, by contrast, is largely unsupervised.

Even with a lot of supervised data, AIs can't make the same kinds of generalizations that human children can. Their knowledge is much narrower and more limited, and they are easily fooled by what are called "adversarial examples." For instance, an AI image recognition system will confidently say that a mixed-up jumble of pixels is a dog if the jumble happens to fit the right statistical pattern—a mistake a baby would never make."

Regarding early stopping and generalization error in network training, see Remark 6.1 in Section 6.1. To make AIs into more robust and resilient learners, researchers are developing methods to build curiosity into AIs, instead of focusing on immediate rewards.

Table 1 | Essential hyperparameters for reproducing the study for each of the three models

	Lesion	Breast	Case
Learning rate	Missing	0.0001	Missing
Learning rate schedule	Missing	Stated	Missing
Optimizer	Stochastic gradient descent with momentum	Adam	Missing
Momentum	Missing	Not applicable	Not applicable
Batch size	4	Unclear	2
Epochs	Missing	120,000	Missing

Figure 151: *Lack of transparency and irreproducibility* (Section 14.7). The table shows many missing pieces of information for the three networks—Lesion, Breast, and Case models—used to detect breast cancer. Learning rate, Section 6.2. Learning-rate schedule, Section 6.3.1, Figure 65 in Section 6.3.5. SGD with momentum, Section 6.3.2 and Remark 6.5. Adam algorithm, Section 6.5.6. Batch size, Sections 6.3.1 and 6.3.5. Epoch, Footnote 145. [421]. (Figure reproduced with permission of the authors).

14.7 Lack of transparency and irreproducibility of results

For "multiple years now", there have been articles on deep learning that looked more like a promotion/advertisement for newly developed AI technologies, rather than scientific papers in the traditional sense that published results should be replicable and verifiable [422]. But it was only on 2020 Oct 14 that many

scientists [421] had enough and protested the lack of transparency in AI research in a "damning" article in *Nature*, a major scientific journal.

> "We couldn't take it anymore," says Benjamin Haibe-Kains, the lead author of the response, who studies computational genomics at the University of Toronto. "It's not about this study in particular—it's a trend we've been witnessing for multiple years now that has started to really bother us." [422]

The particular contentious study was published by the Google-Health authors of [423] on the use of AI in medical imaging to detect breast cancer. But these authors of [423] provided so little information about their code and how it was tested that their article read more like a "promotion of proprietary tech" than a scientific paper. Figure 151 shows the missing pieces of crucial information to reproduce the results. A question would immediately come to mind: Why would a reputable journal like *Nature* accept such a paper? Was the review rigorous enough?

> "When we saw that paper from Google, we realized that it was yet another example of a very high-profile journal publishing a very exciting study that has nothing to do with science," Haibe-Kains says. "It's more an advertisement for cool technology. We can't really do anything with it." [422]

According to [421], even though McKinney et al [423] stated that "all experiments and implementation details were described in sufficient detail in the supplementary methods section of their Article to 'support replication with non-proprietary libraries'," that was a subjective statement, and replicating their results would be a difficult task, since such textual description can hide a high level of complexity of the code, and nuances in the computer code can have large effects in the training and evaluation results.

> "AI is feeling the heat for several reasons. For a start, it is a newcomer. It has only really become an experimental science in the past decade, says Joelle Pineau, a computer scientist at Facebook AI Research and McGill University, who coauthored the complaint. 'It used to be theoretical, but more and more we are running experiments,' she says. 'And our dedication to sound methodology is lagging behind the ambition of our experiments.' " [422]

No progress in science could be made if results were not verifiable and replicable by independent researchers.

14.8 Killing you!

Oh, one more thing: "A.I. Is Making it Easier to Kill (You). Here's How," New York Times Documentaries, 2019.12.13 (Original website) (Youtube).

And getting better at it every day, e.g., by using "a suite of artificial intelligence-driven systems that will be able to control networked 'loyal wingman' type drones and fully autonomous unmanned combat air vehicles" [424]; see also "Collaborative Operations in Denied Environment (CODE) Phase 2 Concept Video" (Youtube).

References

1. Rosenblatt, F. (1957). The perceptron: A perceiving and recognizing automaton. Technical report, Cornell University. Cornell University, Report No. 85-460-1. Project PARA, January. Internet archive. 1070, 1079, 1123, 1279, 1339

2. Rosenblatt, F. (1962). *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*. Spartan Books. 1070, 1079, 1114, 1123, 1278, 1280, 1281, 1282, 1283, 1339

3. Polyak, B. (1964). Some methods of speeding up the convergence of iteration methods . *USSR Computational Mathematics and Mathematical Physics* , *4*(5), 1–17. DOI 10.1016/0041-5553(64)90137-5. 1070, 1078, 1079, 1153, 1157, 1158, 1159

4. Roose, K. (2022). An A.I.-Generated Picture Won an Art Prize. Artists Aren't Happy. *New York Times*, (*Sep 2*). Original website. 1074, 1075

5. Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., et al. (2021). Highly accurate protein structure prediction with AlphaFold. *Nature*, *596*(7873), 583–589. 1076

6. Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484+. Original website. 1076, 1080, 1081

7. Moyer, C. How Google's AlphaGo Beat a Go World Champion. 2016 Mar 28, Original website. 1076

8. Edwards, B. (2022). DeepMind breaks 50-year math record using AI; new record falls a week later. *Ars Technica*, (*Oct 13*). Original website, Internet archive. 1074

9. Vu-Quoc, L., Humer, A. (2022). Deep learning applied to computational mechanics: A comprehensive review, state of the art, and the classics. arXiv:2212.08989. 1074

10. Roose, K. (2023). Bing (Yes, Bing) Just Made Search Interesting Again. *New York Times*, (*Feb 8*). Original website. 1074

11. Knight, W. (2023). Meet Bard, Google's Answer to ChatGPT. *WIRED*, (*Feb 6*). Original website. 1074

12. Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, *61*, 87–117. 1075, 1104, 1106, 1120, 1291, 1292, 1293, 1340

13. LeCun, Y., Bengio, Y., Hinton, G. (2015). Deep learning. *Nature*, *521*(7553), 436–444. 1075, 1080, 1082, 1106, 1120, 1121, 1122, 1197, 1199

14. Khan, S., Yairi, T. (2018). A review on the application of deep learning in system health management. *Mechanical Systems and Signal Processing*, *107*, 241–265. 1075

15. Sanchez-Lengeling, B., Aspuru-Guzik, A. (2018). Inverse molecular design using machine learning: Generative models for matter engineering. *Science*, *361*(6400, SI), 360–365. 1075

16. Ching, T., Himmelstein, D. S., Beaulieu-Jones, B. K., Kalinin, A. A., Do, B. T., et al. (2018). Opportunities and obstacles for deep learning in biology and medicine. *Journal of the Royal Society Interface*, *15*(141). 1075

17. Quinn, J. A., Nyhan, M. M., Navarro, C., Coluccia, D., Bromley, L., et al. (2018). Humanitarian applications of machine learning with remote-sensing data: review and case study in refugee settlement mapping. *Philosophical Transactions of the Royal Society A-Mathematical Physical and Engineering Sciences*, *376*(2128). 1075

18. Higham, C. F., Higham, D. J. (2019). Deep learning: An introduction for applied mathematicians. *SIAM Review*, *61*(4), 860–891. 1075

19. Dayan, P., Abbott, L. (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press. 1075, 1077, 1079, 1098, 1099, 1106, 1107, 1108, 1109, 1111, 1280, 1283, 1284, 1285, 1287

20. Sze, V., Chen, Y. H., Yang, T. J., Emer, J. S. (2017). Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, *105*(12), 2295–2329. 1075, 1085, 1100, 1106, 1277

21. Nielsen, M. (2015). *Neural Networks and Deep Learning*. Determination Press. Original website. Internet archive. 1076, 1100, 1106, 1134, 1135, 1277, 1278, 1281

22. Rumelhart, D., Hinton, G., Williams, R. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. 1076, 1158, 1283, 1291, 1292, 1293, 1339

23. Ghaboussi, J., Garrett, J., Wu, X. (1991). Knowledge-based modeling of material behavior with neural networks. *Journal of Engineering Mechanics-ASCE, 117(1)*, 132–153. 1076, 1077, 1094, 1100, 1241, 1277, 1340

24. Hochreiter, S., Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9(8)*, 1735–1780. 1077, 1097, 1199

25. Wang, K., Sun, W. C. (2018). A multiscale multi-permeability poroplasticity model linked by recursive homogenizations and deep learning. *Computer Methods in Applied Mechanics and Engineering, 334*, 337–380. 1077, 1079, 1090, 1092, 1093, 1094, 1095, 1096, 1240, 1241, 1242, 1243, 1244, 1245, 1246, 1247, 1248, 1249, 1250, 1251, 1252

26. Mohan, A., Gaitonde, D. (2018). A deep learning based approach to reduced order modeling for turbulent flow control using LSTM neural networks. *arXiv:1804.09269 [physics.comp-ph]*. Apr 24. 1077, 1078, 1079, 1096, 1097, 1098, 1252, 1253, 1254, 1255, 1256, 1257, 1258, 1259, 1260

27. Zaman, M., Zhu, J. (1998). A neural network model for a cohesionless soilIn AttohOkine, NO. *Artificial Intelligence and Mathematical Methods in Pavement and Geomechanical Systems*. International Workshop on Artificial Intelligence and Mathematical Methods in Pavement and Geomechanical Systems, Miami, FL, Nov 05-06, 1998. 1077

28. Su, H., Fan, L., Schlup, J. (1998). Monitoring the process of curing of epoxy/graphite fiber composites with a recurrent neural network as a soft sensor. *Engineering Applications of Artificial Intelligence, 11(2)*, 293–306. 1077

29. Li, C., Huang, T. (1999). Automatic structure and parameter training methods for modeling of mechanical systems by recurrent neural networks. *Applied Mathematical Modelling, 23(12)*, 933–944. 1077

30. Waszczyszyn, Z. (2000). Neural networks in structural engineering: Some recent results and prospects for applications. In topping, BHV. *Computational Mechanics for the Twenty-First Century*. 5th International Conference on Computational Structures Technology/2nd International Conference on Engineering Computational Technology, Leuven, Belgium, Sep 06-08, 2000. 1077

31. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., et al. (2017). Attention Is All You Need. CoRR, abs/1706.03762v5. arXiv:1706.03762v5. See Footnote 337. 1077, 1079, 1203, 1206, 1207, 1208, 1209, 1210, 1211, 1316

32. Hahnloser, R., Sarpeshkar, R., Mahowald, M., Douglas, R., Seung, S. (2000). Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit (vol 405, pg 947, 2000). *Nature, 408(6815)*, 1012–U24. 1077, 1107, 1287, 1289, 1290

33. Jarrett, K., Kavukcuoglu, K., Ranzato, M., LeCun, Y. What is the Best Multi-Stage Architecture for Object Recognition? *2009 IEEE 12th International Conference on Computer Vision (ICCV)*. 1077, 1107

34. Nair, V., Hinton, G. (2010). Rectified linear units improve restricted boltzmann machines. *Proceedings of the 27th International Conference on Machine Learning*, Haifa, Israel. 1077, 1107

35. Little, W. (1974). The existence of persistent states in the brain. *Mathematical Biosciences, 19*, 101–120. In Cabrera, B and Gutfreund, H and Kresin, V (eds.), From High-Temperature Superconductivity to Microminiature Refrigeration, William Little Symposium on From High-Temperature Superconductivity to Microminiature Refrigeration, Stanford Univ, Stanford, CA, Sep 30, 1995.[336] 1077, 1288

36. Ramachandran, P., Barret, Z., Le, Q. (2017). Searching for Activation Functions. CoRR (Computing

---

[336] The landmark paper "Little (1974)" was not listed in the Web of Science database as of Nov 2018, using the search keywords [au=(little) and py=(1974) and ts=(brain)]. On the other hand, [au=(little) and ts=(The existence of persistent states in the brain)], i.e., the author's last name and the full title of the paper, led to the 1995 collection of Little's papers edited by Cabrera *et al.*, in which "Little (1974)"was found.

Research Repository), abs/1710.05941v2. arXiv:1710.05941v2. See Footnote 337. 1077, 1120, 1287, 1289, 1290, 1291

37. Wuraola, A., Patel, N. SQNL: A New Computationally Efficient Activation Function. In *2018 International Joint Conference on Neural Networks (IJCNN)*. 1077, 1288

38. Oishi, A., Yagawa, G. (2017). Computational mechanics enhanced by deep learning. *Computer Methods in Applied Mechanics and Engineering*, *327*, 327–351. 1077, 1079, 1086, 1087, 1088, 1089, 1100, 1114, 1121, 1128, 1231, 1232, 1233, 1234, 1235, 1236, 1237, 1238, 1239, 1240, 1277

39. Zienkiewicz, O., Taylor, R., Zhu, J. (2013). *The Finite Element Method: Its Basis and Fundamentals*. Oxford: Butterworth-Heineman. 7th edition. 1077, 1103, 1231, 1232

40. Barlow, J. (1976). Optimal stress locations in finite-element models. *International Journal for Numerical Methods in Engineering*, *10*(2), 243–251. 1077

41. Barlow, J. (1977). Optimal stress locations in finite-element models - reply. *International Journal for Numerical Methods in Engineering*, *11*(3), 604. 1077

42. Abaqus 6.14. *Theory Guide*. Simulia Systems, Dassault Systèmes. Subsection 3.2.4 Solid isoparametric quadrilaterals and hexahedra. (Website, go to Section Reference, Abaqus Theory Guide, Section 3 Elements, Section 3.2 Continuum elements, then Section 3.2.4.). 1077

43. Ghaboussi, J., Garrett, J., Wu, X. (1990). Material Modeling with Neural NetworksIn Pande, GN and Middleton, J. *Numerical Methods in Engineering: Theory and Applications, Vol 2*. 3rd International Conf on Numerical Methods in Engineering: Theory and Applications (NUMETA 90), Univ Coll Swansea, Swansea, Wales, Jan 07-11, 1990. 1077

44. Chen, C. (1989). Applying and validating neural network technology for nondestructive evaluation of materials. In *1989 IEEE International Conference on Systems, Man, and Cybernetics, Vols 1-3: Conference Proceedings*. 1989 IEEE International Conf on Systems, Man, and Cybernetics: Decision-Making in Large-Scale Systems, Cambridge, MA, Nov 14-17, 1989. 1077

45. Sayeh, M., Viswanathan, R., Dhali, S. (1990). Neural networks for assessment of impact and stress relief on composite-materialsIn Genisio, M. *Sixth Annual Conference on Materials Technology: Composite Technology*. 6th Annual Conf on Materials Technology: Composite Technology, Southern Illinois Univ Carbondale, Carbondale, IL, Apr 10-11, 1990. 1077

46. Chen, C., Leclair, S. (1991). A probability neural network (pnn) estimator for improved reliability of noisy sensor data. *Journal of Reinforced Plastics and Composites*, *10*(4), 379–390. 1077

47. Kim, Y., Choi, Y., Widemann, D., Zohdi, T. (2020). A fast and accurate physics-informed neural network reduced order model with shallow masked autoencoderer. (*Sep 28*). Version 2, 2020.09.28: arXiv:2009.11990v2, 2009.11990. 1077, 1078, 1079, 1261, 1262, 1263, 1264, 1265, 1266, 1267, 1268, 1269, 1271, 1273, 1274, 1275

48. Kim, Y., Choi, Y., Widemann, D., Zohdi, T. (2020). Efficient nonlinear manifold reduced order model. (*Nov 13*). arXiv:2011.07727, 2011.07727. 1077, 1078, 1079, 1261

49. Robbins, H., Monro, S. (1951b). Stochastic approximation. *Annals of Mathematical Statistics*, *22*(2), 316. 1078, 1155, 1161, 1185

50. Nesterov, I. (1983). A method of the solution of the convex-programming problem with a speed of convergence O($1/k^2$). *Doklady Akademii Nauk SSSR*, *269*(3), 543–547. In Russian. 1078, 1157, 1159

51. Nesterov, Y. (2018). *Lecture on Convex Optimization*. 2nd edition. Switzerland: Springer Nature. 1078, 1157, 1159

52. Duchi, J., Hazan, E., Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, *12*, 2121–2159. 1078, 1173

53. Tieleman, T., Hinton, G. (2012). Lecture 6e, rmsprop: Divide the gradient by a running average of its recent magnitude. Youtube video, time 5:54. Lecture notes, p.29: Original website, Internet archive. 1078, 1176

54. Zeiler, M. D. (2012). ADADELTA: An adaptive learning rate method. (*Dec 22*). arXiv:1212.5701. 1078, 1174, 1176, 1177

55. Wilson, A. C., Roelofs, R., Stern, M., Srebro, N., Recht, B. (2018). The marginal value of adaptive gradient methods in machine learning. (*May 22*). arXiv:1705.08292v2. Version 1 appeared in 2017, see also the Reviews for NIPS 2017. 1078, 1141, 1153, 1155, 1158, 1159, 1160, 1176, 1178, 1182, 1183, 1185

56. Loshchilov, I., Hutter, F. (2019). Decoupled weight decay regularization. (*Jan 4*). arXiv:1711.05101v3. OpenReview. 1078, 1153, 1155, 1160, 1161, 1167, 1174, 1177, 1183, 1184, 1185, 1191

57. Bahdanau, D., Cho, K., Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. *CoRR*, *abs/1409.0473*. arXiv:1409.0473. 1079, 1203, 1204, 1205, 1206

58. Furshpan, E., Potter, D. (1957). Mechanism of nerve-impulse transmission at a crayfish synapse. *Nature*, *180(4581)*, 342–343. 1079, 1290

59. Furshpan, E., Potter, D. (1959b). Slow post-synaptic potentials recorded from the giant motor fibre of the crayfish. *Journal of Physiology-London*, *145(2)*, 326–335. 1079, 1290

60. Gershgorn, D. (2017). The data that transformed AI research—and possibly the world. *Quartz*, (*Jul 26*). Original website. Internet archive (blurry images). 1079, 1081

61. He, K., Zhang, X., Ren, S., Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. CoRR, abs/1502.01852. arXiv:1502.01852, 1502.01852. 1080, 1108, 1138, 1274, 1288

62. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., et al. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, *115(3)*, 211–252. 1080, 1081

63. Park, E., Liu, W., Russakovsky, O., Deng, J., Li, F., et al. (2017). ImageNet Large scale visual recognition challenge (ILSVRC) 2017, Overview. *ILSVRC 2017*, (*Jul 26*). Original website Internet archive. 1080, 1081

64. Beckwith, W. Science's 2021 Breakthrough: AI-powered Protein Prediction. 2022 Dec 17, Original website. 1079, 1080

65. AlphaFold reveals the structure of the protein universe. DeepMind, 2022 Jul 28, Original website, Internet archive. 1080

66. Callaway, E. DeepMind's AI predicts structures for a vast trove of proteins. 2021 Jul 21, Original website. 1080

67. Editorial (2019). The Guardian view on the future of AI: Great power, great irresponsibility. *The Guardian*, (*Jan 01*). Original website. Internet archive. 1080, 1304, 1305

68. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, *362(6419)*, 1140+. 1080

69. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, *518(7540)*, 529–533. 1081

70. Racaniere, S., Weber, T., Reichert, D. P., Buesing, L., Guez, A., et al. (2017). Imagination-Augmented Agents for Deep Reinforcement Learning. In Guyon, I and Luxburg, UV and Bengio, S and Wallach, H and Fergus, R and Vishwanathan, S and Garnett, R, editor, *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, volume 30 of *Advances in Neural Information Processing Systems*. 1081

71. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., et al. (2017). Mastering the game of Go without human knowledge. *Nature*, *550(7676)*, 354+. 1081

72. Cellan-Jones, Rory (2017). Artificial intelligence - hype, hope and fear. *BBC*, (*Oct 16*). Original website. Internet archive. 1081

73. Campbell, M. (2018). Mastering board games. A single algorithm can learn to play three hard board games. *Science*, *362(6419)*, 1118. 1081

74. *The Economist* (2016). Why artificial intelligence is enjoying a renaissance. (*Jul 15*). (https://goo.gl/Grkofq). 1081, 1122, 1294

75. *The Economist* (2016). From not working to neural networking. (*Jun 25*). (https://goo.gl/z1c9pc). 1081, 1120, 1122, 1294

76. Dodge, S., Karam, L. (2017). A Study and Comparison of Human and Deep Learning Recognition Performance Under Visual Distortions. (*May 6*). CoRR (Computing Research Repository), abs/1705.02498.[337] arXiv:1705.02498. 1081

77. Hardesty, L. (2017). Explained: Neural networks. *MIT News*, (*Apr 14*). Original website. Internet archive. 1081, 1278

78. Goodfellow, I., Bengio, Y., Courville, A. (2016). *Deep Learning*. Cambridge, MA: The MIT Press. 1082, 1084, 1085, 1095, 1100, 1102, 1103, 1104, 1105, 1106, 1107, 1108, 1112, 1114, 1115, 1116, 1117, 1120, 1121, 1122, 1123, 1124, 1125, 1126, 1127, 1128, 1129, 1130, 1133, 1135, 1137, 1138, 1140, 1141, 1143, 1144, 1145, 1146, 1152, 1153, 1154, 1155, 1157, 1158, 1159, 1160, 1161, 1167, 1168, 1170, 1172, 1174, 1176, 1177, 1180, 1182, 1183, 1194, 1195, 1196, 1197, 1199, 1202, 1203, 1204, 1265, 1277, 1278, 1280, 1281, 1284, 1288, 1289, 1290, 1291, 1293, 1334, 1335, 1336, 1339, 1340, 1343

79. Ford, K. (2018). *Architects of Intelligence: The truth about AI from the people building it*. Packt Publishing. 1082, 1084, 1289, 1291, 1292, 1293, 1303

80. Bottou, L., Curtis, F. E., Nocedal, J. (2018). Optimization Methods for Large-Scale Machine Learning. *SIAM Review*, *60*(2), 223–311. 1082, 1144, 1146, 1152, 1153, 1155, 1161, 1174, 1176, 1177

81. Khullar, D. (2019). A.I. Could Worsen Health Disparities. *New York Times*, (*Jan 31*). Original website. 1082

82. Kornfield, M., Firozi, P. (2020). Artificial intelligence use is growing in the U.S. healthcare system. *Washington Post*, (*Feb 24*). Original website. 1082

83. Lee, K. (2018a). *AI Superpowers: China, Silicon Valley, and the New World Order*. Houghton Mifflin Harcourt. 1082

84. Lee, K. (2018b). How AI can save our humanity. *TED2018*, (*Apr*). Original website. 1082

85. Dunjko, V., Briegel, H. J. (2018). Machine learning & artificial intelligence in the quantum domain: a review of recent progress. *Reports on Progress in Physics*, *81*(7), 074001. 1084, 1085

86. Hinton, G. E., Osindero, S., Teh, Y. W. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, *18*(7), 1527–1554. 1084

87. Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., et al. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, *345*(6197), 668–673. 1085

88. Esser, S. K., Merolla, P. A., Arthur, J. V., Cassidy, A. S., Appuswamy, R., et al. (2016). Convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Sciences of the United States of America*, *113*(41), 11441–11446. 1085

89. Warren, J., Root, P. (1963). The behavior of naturally fractured reservoirs. *Society of Petroleum*

---

[337] In a rather cryptic manner to outsiders, several computer-science papers refer to papers in the Computing Research Repository (CoRR) such as, e.g., "CoRR abs/1706.03762v5", which means that the abstract of paper number "1706.03762v5" (version 5) can be accessed by prepending to "abs/1706.03762v5" the CoRR web address https://arxiv.org/ to form https://arxiv.org/abs/1706.03762v5, which can also be obtained via a web search of "abs/1706.03762v5", and where the PDF of the paper can be downloaded. An equivalent reference is "arXiv preprint arXiv:1706.03762v5", which may be clearer since more non-computer-science readers would have heard of the arXiv rather than the CoRR. Papers such as [31] use both types of references, which are also used in the present review paper so readers become familiar with both. To refer to the specific version 5, use "CoRR abs/1706.03762v5"; to refer to the latest version (which may be different from version 5), remove "v5" to use only "CoRR abs/1706.03762".

Engineers Journal, *3*(*03*), 245–255. 1090

90. Ji, Y., Hall, S. A., Baud, P., Wong, T. F. (2015). Characterization of pore structure and strain localization in Majella limestone by X-ray computed tomography and digital image correlation. *Geophysical Journal International*, *200*(*2*), 701–719. 1091, 1092

91. Christensen, R. (2013). *The Theory of Materials Failure*. 1st edition. Oxford University Press. 1090

92. Balogun, A. S., Kazemi, H., Ozkan, E., Al-Kobaisi, M., Ramirez, B. A., et al. (2007). Verification and proper use of water-oil transfer function for dual-porosity and dual-permeability reservoirs. In *SPE Middle East Oil and Gas Show and Conference*. Society of Petroleum Engineers. 1090

93. Ho, C. K. (2000). Dual porosity vs. dual permeability models of matrix diffusion in fractured rock. Technical report. International High-Level Radioactive Waste Conference, Las Vegas, NV (US), 04/29/2001-05/03/2001. Sandia National Laboratories, Albuquerque, NM (US), Report No. SAND2000-2336C. Office of Scientific & Technical Information Report Number 763324. PDF archived at the International Atomic Energy Agency. 1090, 1091

94. Datta-Gupta, A., King, M. J. (2007). *Streamline simulation: Theory and practice*, volume 11. Society of Petroleum Engineers Richardson. 1090, 1091, 1092

95. Croizé, D., Renard, F., Gratier, J. P. (2013). Chapter 3 - compaction and porosity reduction in carbonates: A review of observations, theory, and experiments. In R. Dmowska, editor, *Advances in Geophysics*, volume 54 of *Advances in Geophysics*. Elsevier, 181 – 238. 1091, 1092

96. Lu, J., Qu, J., Rahman, M. M. (2019). A new dual-permeability model for naturally fractured reservoirs. *Special Topics & Reviews in Porous Media: An International Journal*, *10*(*5*). 1091

97. Gers, F. A., Schmidhuber, J. (2000). Recurrent nets that time and countIn *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks*. IEEE. 1092

98. Santamarina, J. C. (2003). Soil behavior at the microscale: particle forces. In *Soil behavior and soft ground construction*. 25–56. Proc. of the Symposium in honor of Charles C. Ladd, October 2001, MIT. 1095

99. Alam, M. F., Haque, A., Ranjith, P. G. (2018). A study of the particle-level fabric and morphology of granular soils under one-dimensional compression using insitu x-ray ct imaging. *Materials*, *11*(*6*), 919. 1094

100. Karatza, Z., Andò, E., Papanicolopulos, S. A., Viggiani, G., Ooi, J. Y. (2019). Effect of particle morphology and contacts on particle breakage in a granular assembly studied using x-ray tomography. *Granular Matter*, *21*(*3*), 44. 1094

101. Shire, T., O'Sullivan, C., Hanley, K., Fannin, R. J. (2014). Fabric and effective stress distribution in internally unstable soils. *Journal of Geotechnical and Geoenvironmental Engineering*, *140*(*12*), 04014072. 1094

102. Kanatani, K. I. (1984). Distribution of directional data and fabric tensors. *International Journal of Engineering Science*, *22*(*2*), 149–164. 1094, 1242

103. Fu, P., Dafalias, Y. F. (2015). Relationship between void-and contact normal-based fabric tensors for 2d idealized granular materials. *International Journal of Solids and Structures*, *63*, 68–81. 1094

104. Graves, A., Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, *18*(*5–6*), 602–610. 1097

105. Graham, J., Kanov, K., Yang, X., Lee, M., Malaya, N., et al. (2016). A web services accessible database of turbulent channel flow and its use for testing a new integral wall model for les. *Journal of Turbulence*, *17*(*2*), 181–215. 1097, 1256

106. Rossant, C., Goodman, D. F. M., Fontaine, B., Platkiewicz, J., Magnusson, A. K., et al. (2011). Fitting neuron models to spike trains . *Frontiers in Neuroscience*, *Feb 23*. 1099

107. Brillouin, L. (1964). *Tensors in Mechanics and Elasticity*. New York: Academic Press. 1100, 1102

108. Misner, C., Thorne, K., Wheeler, J. (1973). *Gravitation*. New York: W.H. Freeman and Company.

1100

109. Malvern, L. (1969). *Introduction to the Mechanics of a Continuous Medium*. Englewood Cliffs, New Jersey: Prentice Hall. 1102

110. Marsden, J., Hughes, T. (1994). *Mathematical Foundation of Elasticity*. New York: Dover. 1102

111. Vu-Quoc, L., Li, S. (1995). Dynamics of sliding geometrically-exact beams - large-angle maneuver and parametric resonance. *Computer Methods in Applied Mechanics and Engineering*, *120(1-2)*, 65–118. 1102

112. Werbos, P. (1988). Backpropagation: Past and future. IEEE 1988 International Conference on Neural Networks, San Diego, 24-27 July 1988. 1106, 1293

113. Glorot, X., Bordes, A., Bengio, Y. (2011). Deep Sparse Rectifier Neural Networks. In *Proceedings of Machine Learning Research (PMLR), Vol.15, Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS), 11-13 April 2011, Fort Lauderdale, FL, USA* . PMLR Vol.15, AISTATS 2011 Paper pdf. 1107, 1109, 1111, 1112, 1137, 1138, 1289

114. Drion, G., O'Leary, T., Marder, E. (2015). Ion channel degeneracy enables robust and tunable neuronal firing rates. *Proceedings of the National Academy of Sciences of the United States of America*, *112(38)*, E5361–E5370. 1108, 1110

115. van Welie, I., van Hooft, J., Wadman, W. (2004). Homeostatic scaling of neuronal excitability by synaptic modulation of somatic hyperpolarization-activated I-h channels. *Proceedings of the National Academy of Sciences of the United States of America*, *101(14)*, 5123–5128. 1111

116. Steyn-Ross, M. L., Steyn-Ross, D. A. (2016). From individual spiking neurons to population behavior: Systematic elimination of short-wavelength spatial modes. *Physical Review E*, *93(2)*. 1111, 1286

117. Dutta, S., Kumar, V., Shukla, A., Mohapatra, N. R., Ganguly, U. (2017). Leaky Integrate and Fire Neuron by Charge-Discharge Dynamics in Floating-Body MOSFET. *Scientific Reports*, *7*. 1111

118. Wilson, H. (1999). Simplified dynamics of human and mammalian neocortical neurons. *Journal of Theoretical Biology*, *200(4)*, 375–388. 1111, 1285, 1286

119. Rosenblatt, F. (1958). The perceptron - A probabilistic model for information-storage and organization in the brain. *Psychological Review*, *65(6)*, 386–408. 1113, 1114, 1116, 1117, 1122, 1123, 1277, 1278, 1280, 1281, 1282, 1283

120. Block, H. (1962a). Perceptron - A model for brain functioning .1. *Reviews of Modern Physics*, *34(1)*, 123–135. 1113, 1114, 1278, 1281, 1282, 1283

121. Minsky, M., Papert, S. (1969). *Perceptrons: An introduction to computational geometry*. MIT Press. 1988 expanded edition. 2017 edition with foreword by Leon Bottou, Facebook AI. 1114, 1115, 1281, 1282, 1283

122. Herzberger, M. (1949). The normal equations of the method of least squares and their solution. *Quarterly of Applied Mathematics*, *7(2)*, 217–223. (pdf). 1116

123. Weisstein, E. W. Normal equation. From *MathWorld*–A Wolfram Web Resource. URL: http://mathworld.wolfram.com/NormalEquation.html. 1116

124. Dyson, F. (2004). A meeting with Enrico Fermi - How one intuitive physicist rescued a team from fruitless research. *Nature*, *427(6972)*, 297. 1120

125. Mayer, J., Khairy, K., Howard, J. (2010). Drawing an elephant with four complex parameters. *American Journal of Physics*, *78(6)*, 648–649. 1120

126. Hsu, J. (2015). Biggest Neural Network Ever Pushes AI Deep Learning. *IEEE Spectrum*. 1122

127. He, K., Zhang, X., Ren, S., Sun, J. (2015). Deep Residual Learning for Image Recognition. CoRR (Computing Research Repository), abs/1512.03385v1. arXiv:1512.03385v1. See Footnote 337. 1123, 1124, 1125, 1135, 1168, 1170

128. Huang, G., Sun, Y., Liu, Z., Sedra, D., Weinberger, K. (2016). Deep Networks with Stochastic Depth. CoRR (Computing Research Repository), abs/1603.09382v3. arXiv:1603.09382v3. See Footnote 337.

1124

129. Zagoruyko, S., Komodakis, N. (2017). Wide residual networks. (*Jun 17*). CoRR (Computing Research Repository), arXiv:1605.07146v4. 1124

130. Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. New York: Springer Science+Business Media. 1127, 1128, 1129, 1130, 1140, 1211, 1215, 1216, 1217, 1218, 1219, 1337, 1338, 1339

131. Maas, A., Hannun, A., Ng, A. (2013). Rectifier nonlinearities improve neural network acoustic models. ICML Workshop on Deep Learning for Audio, Speech, and Language Processing (WDLASL 2013), Accepted papers. See also leakyReluLayer, Leaky Rectified Linear Unit (ReLU) layer, MathWorks. 1138

132. Li, H., Xu, Z., Taylor, G., Studer, C., Goldstein, T. (2018). Visualizing the loss landscape of neural nets. (*Nov 7*). arXiv:1712.09913v3. 1139

133. Geman, S., Bienenstock, E., Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural computation*, *4*(*1*), 1–58. pdf, pdf. 1140, 1142

134. Hastie, T., Tibshirani, R., Friedman, J. H. (2001). *The elements of statistical learning: Data mining, inference, prediction.* 1st edition. Springer. 2nd edition, corrected, 12 printing, 2017 Jan 13. 1140

135. Prechelt, L. (1998). Early Stopping — But When ? In G. Orr, K. Muller. *Neural Networks: Tricks of the Trade* . Springer . LLCS State-of-the-Art Survey. Paper pdf, Internet archive. 1141, 1142, 1143

136. Belkin, M., Hsu, D., Ma, S., Mandal, S. (2019). Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, *116*(*32*), 15849–15854. Original website, arXiv:1812.11118. 1143, 1144, 1145

137. Geiger, M., Jacot, A., Spigler, S., Gabriel, F., Sagun, L., et al. (2020). Scaling description of generalization with number of parameters in deep learning. *Journal of Statistical Mechanics: Theory and Experiment*, *2020*(*2*), 023401. Original website, arXiv:1901.01608. 1143, 1145

138. Sampaio, P. R. (2020). Deft-funnel: an open-source global optimization solver for constrained grey-box and black-box problems. (*Jan 2020*). arXiv:1912.12637. 1144

139. Polak, E. (1971). *Computational Methods in Optimization: A Unified Approach*. Academic Press. 1146, 1147, 1148, 1149, 1150, 1151, 1152, 1158, 1189

140. Lewis, R., Torczon, V., Trosset, M. (2000). Direct search methods: then and now. *Journal of Computational and Applied Mathematics*, *124*(*1-2*), 191–207. 1146, 1152

141. Kolda, T., Lewis, R., Torczon, V. (2003). Optimization by direct search: New perspectives on some classical and modern methods. *SIAM Review*, *45*(*3*), 385–482. 1146, 1152

142. Kafka, D., Wilke, D. (2018). Gradient-only line searches: An alternative to probabilistic line searches. (*Mar 22*). arXiv:1903.09383. 1146, 1193

143. Mahsereci, M., Hennig, P. (2017). Probabilistic line searches for stochastic optimization. *Journal of Machine Learning Research*, *18*. Article No.1. Also, CoRR, abs/1703.10034v2, Jun 30. arXiv:1703.10034v2, 1703.10034. 1146, 1152, 1153, 1191

144. Paquette, C., Scheinberg, K. (2018). A stochastic line search method with convergence rate analysis. (*Jul 20*). arXiv:1807.07994v1. 1146, 1149, 1150, 1151, 1153, 1155, 1185, 1187, 1188, 1189

145. Bergou, E., Diouane, Y., Kungurtsev, V., Royer, C. W. (2018). A subsampling line-search method with second-order results. (*Nov 21*). arXiv:1810.07211v2. 1146, 1149, 1150, 1151, 1153, 1155, 1185, 1188, 1189, 1190, 1191, 1192

146. Wills, A., Schön, T. (2018). Stochastic quasi-newton with adaptive step lengths for large-scale problems. (*Feb 22*). arXiv:1802.04310v1. 1146, 1188, 1191

147. Mahsereci, M., Hennig, P. (2015). Probabilistic line searches for stochastic optimization. *CoRR*, (*Feb 10*). Abs/1502.02846. arXiv:1502.02846. 1146, 1152

148. Luenberger, D., Ye, Y. (2016). *Linear and Nonlinear Programming*. 4th edition. Springer. 1147, 1149,

1158

149. Polak, E. (1997). *Optimization: Algorithms and Consistent Approximations*. Springer Verlag. 1147, 1148, 1149, 1152, 1158

150. Goldstein, A. (1965). On steepest descent. *SIAM Journal of Control, Series A*, *3(1)*, 147–151. 1147, 1148, 1149, 1152, 1153

151. Armijo, L. (1966). Minimization of functions having lipschitz continuous partial derivatives. *Pacific Journal of Mathematics*, *16(1)*, 1–3. 1147, 1148, 1149, 1153, 1189

152. Wolfe, P. (1969). Convergence conditions for ascent methods. *SIAM Review*, *11(2)*, 226–235. 1147, 1149, 1152, 1153

153. Wolfe, P. (1971). Convergence conditions for ascent methods. II: Some corrections. *SIAM Review*, *13*. 1147, 1152

154. Goldstein, A. (1967). *Constructive Real Analysis*. New York: Harper. 1147, 1152

155. Goldstein, A., Price, J. (1967). An effective algorithm for minimization. *Numerische Mathematik*, *10*, 184–189. 1147, 1148, 1149

156. Ortega, J., Rheinboldt, W. (1970). *Iterative Solution of Nonlinear Equations in Several Variables*. New York: Academic Press. Republished in 2000 by SIAM, Classics in Applied Mathematics, Vol.30. 1147, 1148, 1149, 1158

157. Nocedal, J., Wright, S. (2006). *Numerical Optimization*. Springer. 2nd edition. 1149, 1158

158. Bollapragada, R., Byrd, R. H., Nocedal, J. (2019). Exact and inexact subsampled Newton methods for optimization. *IMA Journal of Numerical Analysis*, *39(2)*, 545–578. 1149

159. Berahas, A. S., Byrd, R. H., Nocedal, J. (2019). Derivative-free optimization of noisy functions via quasi-newton methods. *SIAM Journal on Optimization*, *29(2)*, 965–993. 1149

160. Larson, J., Menickelly, M., Wild, S. M. (2019). Derivative-free optimization methods. (*Jun 25*). arXiv:1904.11585v2. 1149

161. Shi, Z., Shen, J. (2005). Step-size estimation for unconstrained optimization methods. *Computational and Applied Mathematics*, *24(3)*, 399–416. 1152

162. Sun, S., Cao, Z., Zhu, H., Zhao, J. (2019). A survey of optimization methods from a machine learning perspective. (*Oct 23*). arXiv:1906.06821v2. 1153, 1174, 1176, 1177

163. Kirkpatrick, S., Gelatt, C., Vecchi, M. (1983). Optimization by simulated annealing. *Science*, *220(4598)*, 671–680. 1153, 1164, 1167

164. Smith, S. L., Kindermans, P. J., Ying, C., Le, Q. V. (2018). Don't decay the learning rate, increase the batch size. (*Feb 2018*). arXiv:1711.00489v2. OpenReview. 1153, 1161, 1163, 1164, 1165, 1166, 1182

165. Schraudolph, N. (1998). Centering Neural Network Gradient Factors In G. Orr, K. Muller. *Neural Networks: Tricks of the Trade* . Springer . LLCS State-of-the-Art Survey. 1153, 1158, 1175

166. Neuneier, R., Zimmermann, H. (1998). How to Train Neural Networks In G. Orr, K. Muller. *Neural Networks: Tricks of the Trade* . Springer . LLCS State-of-the-Art Survey. 1153, 1175

167. Robbins, H., Monro, S. (1951a). A stochastic approximation method. *Annals of Mathematical Statistics*, *22(3)*, 400–407. 1153

168. Aitchison, L. (2019). Bayesian filtering unifies adaptive and non-adaptive neural network optimization methods. (*Jul 31*). arXiv:1807.07540v4. 1155, 1175, 1185, 1186

169. Goudou, X., Munier, J. (2009). The gradient and heavy ball with friction dynamical systems: The quasiconvex case. *Mathematical Programming*, *116(1-2)*, 173–191. 7th French-Latin American Congress in Applied Mathematics, Univ Chile, Santiago, CHILE, JAN, 2005. 1157, 1159

170. Kingma, D. P., Ba, J. (2014). Adam: A method for stochastic optimization. (*Dec 22*). Version 1, 2014.12.22: arXiv:1412.6980v1. Version 9, 2017.01.30: arXiv:1412.6980v9. 1158, 1170, 1173, 1174, 1178, 1179, 1180, 1181

171. Bertsekas, D., Tsitsiklis, J. (1995). *Neuro-Dynamic Programming* . Athena Scientific. 1158, 1159

172. Hinton, G. (2012). A Practical Guide to Training Restricted Boltzmann Machines In G. Montavon, G. Orr, K. Muller. *Neural Networds: Tricks of the Trade* . Springer . LLCS State-of-the-Art Survey. 1158

173. Incerti, S., Parisi, V., Zirilli, F. (1979). New method for solving non-linear simultaneous equations. *SIAM Journal on Numerical Analysis*, *16*(*5*), 779–789. 1158

174. Voigt, R. (1971). Rates of convergence for a class of iterative procedures. *SIAM Journal on Numerical Analysis*, *8*(*1*), 127–134. 1158

175. Plaut, D. C., Nowlan, S. J., Hinton, G. E. (1986). Experiments on learning by back propagation. Technical Report Technical Report CMU-CS-86-126, June. Website. 1158, 1167

176. Jacobs, R. (1988). Increased rates of convergence through learning rate adaptation. *Neural Networks*, *1*(*4*), 295–307. 1158

177. Hagiwara, M. (1992). Theoretical derivation of momentum term in back-propagation In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'92)*. volume 1. Piscataway, NJ, IEEE . 1158

178. Gill, P., Murray, W., Wright, M. (1981). *Practical Optimization*. Academic Press. 1158

179. Snyman, J., Wilke, D. (2018). *Practical Mathematical Optimization: Basic optimization theory and gradient-based algorithms* . Springer. 1158, 1193

180. Priddy, K., Keller, P. (2005). *Artificial neural network: An introduction* . SPIE. 1159

181. Sutskever, I., Martens, J., Dahl, G., Hinton, G. (2013). On the importance of initialization and momentum in deep learning. *Proceedings of the 30th International Conference on Machine Learning, PMLR*, *28*(*3*). Original website. 1159

182. Reddi, S. J., Kale, S., Kumar, S. (2019). On the convergence of Adam and beyond. (*Oct 23*). arXiv:1904.09237. OpenReview. Best paper ICLR 2018. 1160, 1170, 1172, 1174, 1178, 1179, 1180, 1181, 1191

183. Phuong, T. T., Phong, L. T. (2019). On the convergence proof of AMSGrad and a new version. (*Oct 31*). arXiv:1904.03590v4. 1160, 1180, 1181

184. Li, X., Orabona, F. (2019). On the convergence of stochastic gradient descent with adaptive stepsizes. (*Feb 26*). arXiv:1805.08114v3. 1161

185. Gardiner, C. (2004). *Handbook of Stochastic Methods: for Physics, Chemistry and the Natural Sciences*. Synergetics, 3rd edition. Springer. 1162, 1165, 1166

186. Smith, S. L., Le, Q. V. (2018). A bayesian perspective on generalization and stochastic gradient descent. (*Feb 2018*). arXiv:1710.06451v3. OpenReview. 1163, 1164

187. Li, Q., Tai, C., E, W. (2017). Stochastic modified equations and adaptive stochastic gradient algorithms. (*Jun 20*). arXiv:1511.06251v3. Proceedings of Machine Learning Research, 70:2101-2110, 2017. 1165

188. Lemons, D., Gythiel, A. (1997). Paul Langevin's 1908 paper ''On the theory of Brownian motion''. *American Journal of Physics*, *65*(*11*), 1079–1081. 1166, 1167

189. Coffey, W., Kalmikov, Y., Waldron, J. (2004). *The Langevin Equation* . 2nd edition. World Scientific. 1166

190. Lones, M. A. Metaheuristics in nature-inspired algorithms. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. 1167

191. Yang, X. S. (2014). *Nature-inspired optimization algorithms*. Elsevier. 1167

192. Rere, L. R., Fanany, M. I., Arymurthy, A. M. (2015). Simulated annealing algorithm for deep learning. *Procedia Computer Science*, *72*(*1*), 137–144. 1167

193. Rere, L., Fanany, M. I., Arymurthy, A. M. (2016). Metaheuristic algorithms for convolution neural network. *Computational Intelligence and Neuroscience*, *2016*. 1167

194. Fong, S., Deb, S., Yang, X. (2018). How meta-heuristic algorithms contribute to deep learning in the

hype of big data analytics. In *Progress in Intelligent Computing Techniques: Theory, Practice, and Applications*. Springer, 3–25. 1167

195. Bozorg-Haddad, O. (2018). *Advanced optimization by nature-inspired algorithms*. Springer. 1167

196. Al-Obeidat, F., Belacel, N., Spencer, B. (2019). Combining machine learning and metaheuristics algorithms for classification method proaftn. In *Enhanced Living Environments*. Springer, 53–79. 1167

197. Bui, Q. T. (2019). Metaheuristic algorithms in optimizing neural network: A comparative study for forest fire susceptibility mapping in Dak Nong, Vietnam. *Geomatics, Natural Hazards and Risk*, *10*(*1*), 136–150. 1167

198. Devikanniga, D., Vetrivel, K., Badrinath, N. (2019). Review of meta-heuristic optimization based artificial neural networks and its applications. In *Journal of Physics: Conference Series*. volume 1362. IOP Publishing. 1167

199. Mirjalili, S., Dong, J. S., Lewis, A. (2020). *Nature-Inspired Optimizers*. Springer. 1167

200. Smith, L. N., Topin, N. (2018). Super-convergence: Very fast training of residual networks using large learning rates. (*May 2018*). arXiv:1708.07120v3. OpenReview. 1167

201. Rögnvaldsson, T. S. (1998). A Simple Trick for Estimating the Weight Decay ParameterIn G. Orr, K. Muller. *Neural Networks: Tricks of the Trade*. Springer . LLCS State-of-the-Art Survey. 1167

202. Glorot, X., Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networksIn *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 1168

203. Bock, S., Goppold, J., Weiss, M. (2018). An improvement of the convergence proof of the ADAM-optimizer. (*Apr 27*). arXiv:1804.10587v1. 1170, 1180, 1181

204. Huang, H., Wang, C., Dong, B. (2019). Nostalgic Adam: Weighting more of the past gradients when designing the adaptive learning rate. (*Feb 23*). arXiv:1805.07557v2. 1170, 1181

205. Chen, X., Liu, S., Sun, R., Hong, M. (2019). On the convergence of a class of Adam-type algorithms for non-convex optimization. (*Mar 10*). arXiv:1808.02941v2. OpenReview. 1174

206. Hyndman, R. J., Koehler, A. B., Ord, J. K., Snyder, R. D. (2008). *Forecasting with Exponential Smoothing: A state state approach*. Springer. 1174

207. Hyndman, R. J., Athanasopoulos, G. (2018). *Forecasting: Principles and Practices*. 2nd edition. OTexts: Melbourne, Australia. Original website, open online text. 1175, 1176

208. Dreiseitl, S., Ohno-Machado, L. (2002). Logistic regression and artificial neural network classification models: a methodology review. *Journal of Biomedical Informatics*, *35*, 352–359. 1180

209. Gugger, S., Howard, J. (2018). AdamW and Super-convergence is now the fastest way to train neural nets. *Fast.AI*, (*Jul 02*). Original website, Internet Archive. 1181, 1185

210. Xing, C., Arpit, D., Tsirigotis, C., Bengio, Y. (2018). A walk with sgd. (*May 2018*). arXiv:1802.08770v4. OpenReview. 1182

211. Prokhorov, D. (2001). IJCNN 2001 neural network competition. Slide presentation in IJCNN'01, Ford Research Laboratory, 2001 Internet Archive. 1191

212. Chang, C. C., Lin, C. J. (2011). LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology*, *2*(*3*). Article 27, April 2011. Original website for software (Version 3.24 released 2019.09.11), Internet Archive. 1191

213. Brogan, W. L. (1990). *Modern Control Theory*. 3rd edition. Pearson. 1193

214. Hopfield, J. J. (1984). Neurons with graded response have collective computational properties like those of two-state neurons. *Proceedings of the National Academy of Sciences*, *81*(*10*), 3088–3092. Original website. 1193

215. Pineda, F. J. (1987). Generalization of back-propagation to recurrent neural networks. *Physical Review Letters*, *59*(*19*), 2229–2232. 1193

216. Newmark, N. M. (1959). *A Method of Computation for Structural Dynamics*. Number 85 in A Method of Computation for Structural Dynamics. American Society of Civil Engineers. 1194

217. Hilber, H. M., Hughes, T. J., Taylor, R. L. (1977). Improved numerical dissipation for time integration algorithms in structural dynamics. *Earthquake Engineering & Structural Dynamics*, *5*(*3*), 283–292. Original website. 1194

218. Chung, J., Hulbert, G. M. (1993). A Time Integration Algorithm for Structural Dynamics With Improved Numerical Dissipation: The Generalized-α Method. *Journal of Applied Mechanics*, *60*(*2*), 371. Original website. 1194

219. Olah, C. (2015). Understanding LSTM Networks. *colah's blog*, (*Aug 27*). Original website. Internet archive. 1199

220. Cho, K., van Merrienboer, B., Bahdanau, D., Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. arXiv:1409.1259. 1202

221. Chung, J., Gulcehre, C., Cho, K., Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv:1412.3555. 1202, 1203

222. Kim, Y., Denton, C., Hoang, L., Rush, A. M. (2017). Structured attention networks. International Conference on Learning Representations, OpenReview.net, arXiv:1702.00887. 1203

223. Cho, K., van Merriënboer, B., Bahdanau, D., Bengio, Y. Doha, Qatar. 1204

224. Schuster, M., Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, *45*(*11*), 2673–2681. 1205

225. Ba, J. L., Kiros, J. R., Hinton, G. E. (2016). Layer normalization. arXiv:1607.06450. 1209

226. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., et al. (2020). Language models are few-shot learners. arXiv:2005.14165v4. 1211

227. Tsai, Y. H. H., Bai, S., Yamada, M., Morency, L. P., Salakhutdinov, R. (2019). Transformer dissection: A unified understanding of transformer's attention via the lens of kernel. arXiv:1908.11775. 1211

228. Rodriguez-Torrado, R., Ruiz, P., Cueto-Felgueroso, L., Green, M. C., Friesen, T., et al. (2022). Physics-informed attention-based neural network for hyperbolic partial differential equations: application to the buckley–leverett problem. *Scientific Reports*, *12*(*1*), 1–12. Original website. 1211, 1230

229. Bahri, Y. (2019). Towards an Understanding of Wide, Deep Neural Networks. Youtube. 1211

230. Ananthaswamy, A. (2021). A New Link to an Old Model Could Crack the Mystery of Deep Learning. *Quanta Magazine*, (*Oct 11*). Original website. 1211

231. Lee, J., Bahri, Y., Novak, R., Schoenholz, S. S., Pennington, J., et al. (2018). Deep neural networks as gaussian processes. arXiv:1711.00165. 1211, 1305

232. Jacot, A., Gabriel, F., Hongler, C. (2018). Neural tangent kernel: Convergence and generalization in neural networks. arXiv:1806.07572. 1211, 1212, 1230

233. 2021's Biggest Breakthroughs in Math and Computer Science. Quanta Magazine, 2021 Dec 31. Youtube. 1211, 1304

234. Rasmussen, C. E., Williams, C. K. (2006). *Gaussian processes for machine learning*. MIT press Cambridge, MA. MIT website, GaussianProcess.org. 1211, 1215, 1216, 1217, 1219, 1339

235. Belkin, M., Ma, S., Mandal, S. (2018). To understand deep learning we need to understand kernel learning. arXiv:1802.0139. 1212, 1215

236. Lee, J., Schoenholz, S. S., Pennington, J., Adlam, B., Xiao, L., et al. (2020). Finite versus infinite neural networks: an empirical study. arXiv:2007.15801. 1212

237. Aronszajn, N. (1950). Theory of reproducing kernels. *Transactions of the American mathematical society*, *68*(*3*), 337–404. 1212, 1215

238. Hastie, T., Tibshirani, R., Friedman, Friedman, J. H. (2017). *The elements of statistical learning: Data mining, inference, and prediction*. 2 edition. Springer. Corrected, 12th printing, Jan 13. 1213, 1214, 1215

239. Evgeniou, T., Pontil, M., Poggio, T. (2000). Regularization networks and support vector machines. *Advances in computational mathematics*, *13*(*1*), 1–50. Semantic Scholar. 1213, 1214, 1215

240. Berlinet, A., Thomas-Agnan, C. (2004). *Reproducing kernel Hilbert spaces in probability and statistics*. New York: Springer Science & Business Media. 1214, 1215, 1216

241. Girosi, F. (1998). An equivalence between sparse approximation and support vector machines. *Neural computation*, *10*(*6*), 1455–1480. Original website, Semantic Scholar. 1214, 1215

242. Wahba, G. (1990). *Spline Models for Observational Data*. Philadelphia, Pennsylvania: SIAM. 4th printing 2002. 1215

243. Adler, B. (2021). Hilbert spaces and the Riesz Representation Theorem. The University of Chicago Mathematics REU 2021, Original website, Internet archive. 1215

244. Schaback, R., Wendland, H. (2006). Kernel techniques: From machine learning to meshless methods. *Acta numerica*, *15*, 543–639. 1215

245. Yaida, S. Non-gaussian processes and neural networks at finite widths. In *Mathematical and Scientific Machine Learning*. 1216

246. Sendera, M., Tabor, J., Nowak, A., Bedychaj, A., Patacchiola, M., et al. (2021). Non-gaussian gaussian processes for few-shot regression. *Advances in Neural Information Processing Systems*, *34*, 10285–10298. arXiv:2110.13561. 1216

247. Duvenaud, D. (2014). *Automatic model construction with Gaussian processes*. Ph.D. thesis, University of Cambridge. PhD dissertation. Thesis repository, CC BY-SA 2.0 UK. 1219, 1220

248. von Mises, R. (1964). *Mathematical theory of probability and statistics*. Elsevier. Book site. 1219, 1339

249. Hale, J. (2018). Deep Learning Framework Power Scores 2018. *Towards Data Science*, (*Sep 19*). Original website. Internet archive. 1220, 1222

250. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Whitepaper pdf, Software available from tensorflow.org. 1222

251. Jouppi, N. Google supercharges machine learning tasks with TPU custom chip. Original website. 1222

252. Chollet, F., et al. (2015). Keras. Original website. 1223

253. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, R. Garnett, editors, *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. Paper pdf. 1223

254. Chintala, S. (2022). Decisions and pivots on pytorch. 2022 Jan 19, Original website Internet archive. 1223

255. PyTorch Turns 5! 2022 Jan 20, Youtube. 1223

256. Kaelbling, L. P., Littman, M. L., Moore, A. W. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, *4*, 237–285. 1223

257. Arulkumaran, K., Deisenroth, M. P., Brundage, M., Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, *34*, 26–38. 1223

258. Sünderhauf, N., Brock, O., Scheirer, W., Hadsell, R., Fox, D., et al. (2018). The limits and potentials of deep learning for robotics. *The International Journal of Robotics Research*, *37*, 405–420. 1224

259. Simo, J. C., Vu-Quoc, L. (1988). On the dynamics in space of rods undergoing large motions–a geometrically exact approach. *Computer Methods in Applied Mechanics and Engineering*, *66*, 125–161. 1224

260. Humer, A. (2013). Dynamic modeling of beams with non-material, deformation-dependent boundary conditions. *Journal of Sound and Vibration*, *332*(*3*), 622–641. 1224

261. Steinbrecher, I., Humer, A., Vu-Quoc, L. (2017). On the numerical modeling of sliding beams: A comparison of different approaches. *Journal of Sound and Vibration*, *408*, 270–290. 1224

262. Humer, A., Steinbrecher, I., Vu-Quoc, L. (2020). General sliding-beam formulation: A non-material description for analysis of sliding structures and axially moving beams. *Journal of Sound and Vibration*, *480*, 115341. Original website. 1224

263. Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., et al. (2018). JAX: composable transformations of Python+NumPy programs. Original website. 1224

264. Heek, J., Levskaya, A., Oliver, A., Ritter, M., Rondepierre, B., et al. (2020). Flax: A neural network library and ecosystem for JAX. Original website. 1224

265. Schoeberl, J. (2014). C++11 Implementation of Finite Elements in NGSolve. Scientific report. 1225, 1226

266. Weitzhofer, S., Humer, A. (2021). Machine-Learning Frameworks in Scientific Computing: Finite Element Analysis and Multibody Simulation. Talk slides, Video talk. 1225

267. Lavin, A., Zenil, H., Paige, B., Krakauer, D., Gottschlich, J., et al. (2021). Simulation Intelligence: Towards a New Generation of Scientific Methods. arXiv:2112.03235. 1225

268. Cai, S., Mao, Z., Wang, Z., Yin, M., Karniadakis, G. E. (2021). Physics-informed neural networks (PINNs) for fluid mechanics: A review. *Acta Mechanica Sinica*, *37*(*12*), 1727–1738. Original website, arXiv:2105.09506. 1225, 1226, 1227

269. Cuomo, S., di Cola, V. S., Giampaolo, F., Rozza, G., Raissi, M., et al. (2022). Scientific Machine Learning through Physics-Informed Neural Networks: Where we are and What's next. *Journal of Scientific Computing*, *92*(*3*). Article No. 88, Original website, arXiv:2201.05624. 1226, 1227, 1228

270. Karniadakis, G. E., Kevrekidis, I. G., Lu, L., Perdikaris, P., Wang, S., et al. (2021). Physics-informed machine learning. *Nature Reviews Physics*, *3*(*6*), 422–440. Original website. 1226, 1227, 1228

271. Lu, L., Meng, X., Mao, Z., Karniadakis, G. E. (2021). DeepXDE: A deep learning library for solving differential equations. *SIAM Review*, *63*(*1*), 208–228. Original website, pdf, arXiv:1907.04502. 1226, 1228

272. Hennigh, O., Narasimhan, S., Nabian, M. A., Subramaniam, A., Tangsali, K., et al. (2020). NVIDIA SimNet (tm): an AI-accelerated multi-physics simulation framework. arXiv:2012.07938. The software name "SimNet" has been changed to "Modulus"; see NVIDIA Modulus. 1228

273. Koryagin, A., Khudorozkov, R., Tsimfer, S. (2019). PyDEns: a Python Framework for Solving Differential Equations with Neural Networks. arXiv:1909.11544. 1228

274. Chen, F., Sondak, D., Protopapas, P., Mattheakis, M., Liu, S., et al. (2020). NeuroDiffEq: A python package for solving differential equations with neural networks. *Journal of Open Source Software*, *5*(*46*), 1931. Original website. 1227, 1228

275. Rackauckas, C., Nie, Q. (2017). DifferentialEquations. jl–a performant and feature-rich ecosystem for solving differential equations in Julia. *Journal of Open Research Software*, *5*(*1*). Original website. 1227, 1228

276. Haghighat, E., Juanes, R. (2021). SciANN: A keras/tensorflow wrapper for scientific computations and physics-informed deep learning using artificial neural networks. *Computer Methods in Applied Mechanics and Engineering*, *373*, 113552. 1228

277. Xu, K., Darve, E. (2020). ADCME: Learning Spatially-varying Physical Fields using Deep Neural Networks. arXiv:2011.11955. 1228

278. Gardner, J. R., Pleiss, G., Bindel, D., Weinberger, K. Q., Wilson, A. G. (2018). Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. [v6] Tue, 29 Jun 2021 arXiv:1809.11165. 1228

279. Schoenholz, S. S., Novak, R. (2020). Fast and Easy Infinitely Wide Networks with Neural Tangents. Google AI Blog, 2020 Mar 13, Original website. 1228, 1305

280. He, J., Li, L., Xu, J., Zheng, C. (2020). ReLU deep neural networks and linear finite elements. *Journal of Computational Mathematics*, *38*(*3*), 502–527. arXiv:1807.03973. 1228, 1230

281. Arora, R., Basu, A., Mianjy, P., Mukherjee, A. (2016). Understanding deep neural networks with rectified linear units. arXiv:1611.01491. 1228

282. Raissi, M., Perdikaris, P., Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, *378*, 686–707. Original website. 1229, 1230

283. Kharazmi, E., Zhang, Z., Karniadakis, G. E. (2019). Variational physics-informed neural networks for solving partial differential equations. arXiv:1912.00873. 1229, 1230

284. Kharazmi, E., Zhang, Z., Karniadakis, G. E. (2021). hp-vpinns: Variational physics-informed neural networks with domain decomposition. *Computer Methods in Applied Mechanics and Engineering*, *374*, 113547. See also arXiv:1912.00873. 1229, 1230

285. Berrone, S., Canuto, C., Pintore, M. (2022). Variational physics informed neural networks: the role of quadratures and test functions. *Journal of Scientific Computing*, *92*(*3*), 1–27. Original website. 1229

286. Wang, S., Yu, X., Perdikaris, P. (2020). When and why pinns fail to train: A neural tangent kernel perspective. arXiv:2007.14527. 1230

287. Rohrhofer, F. M., Posch, S., Gössnitzer, C., Geiger, B. C. (2022). Understanding the difficulty of training physics-informed neural networks on dynamical systems. arXiv:2203.13648. 1230

288. Erichson, N. B., Muehlebach, M., Mahoney, M. W. (2019). Physics-informed Autoencoders for Lyapunov-stable Fluid Flow Prediction. arXiv:1905.10866. 1230

289. Raissi, M., Perdikaris, P., Karniadakis, G. E. (2021). Physics informed learning machine. US Patent 10,963,540, Mar 30. Google Patents, pdf. 1230, 1231

290. Lagaris, I. E., Likas, A., Fotiadis, D. I. (1998). Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, *9*(*5*), 987–1000. Original website. 1230

291. Lagaris, I. E., Likas, A. C., Papageorgiou, D. G. (2000). Neural-network methods for boundary value problems with irregular boundaries. *IEEE Transactions on Neural Networks*, *11*(*5*), 1041–1049. Original website. 1230

292. Raissi, M., Perdikaris, P., Karniadakis, G. E. (2017). Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations. arXiv:1711.10561. 1230

293. Raissi, M., Perdikaris, P., Karniadakis, G. E. (2017). Physics Informed Deep Learning (Part II): Data-driven Discovery of Nonlinear Partial Differential Equations. arXiv:1711.10566. 1230

294. Gupta, S., Agrawal, A., Gopalakrishnan, K., Narayanan, P. (2015). Deep learning with limited numerical precisionIn *International Conference on Machine Learning*. arXiv:1502.02551. 1239

295. Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., Bengio, Y. (2016). Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. arXiv:1602.02830. 1239

296. De Sa, C., Feldman, M., Ré, C., Olukotun, K. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 1239

297. Borja, R. I. (2000). A finite element model for strain localization analysis of strongly discontinuous fields based on standard galerkin approximation. *Computer Methods in Applied Mechanics and Engineering*, *190*(*11-12*), 1529–1549. 1247, 1250, 1251, 1252

298. Sibson, R. H. (1985). A note on fault reactivation. *Journal of Structural Geology*, *7*(*6*), 751–754. 1248

299. Passelègue, F. X., Brantut, N., Mitchell, T. M. (2018). Fault reactivation by fluid injection: Controls from stress state and injection rate. *Geophysical Research Letters*, *45*(*23*), 12–837. 1249

300. Kuchment, A. (2019). Even if injection of fracking wastewater stops, quakes won't. *Scientific Ameri-

can. Sep 9. 1249

301. Park, K., Paulino, G. H. (2011). Cohesive zone models: a critical review of traction-separation relationships across fracture surfaces. *Applied Mechanics Reviews*, *64*(*6*). 1249

302. Zhang, X., Vu-Quoc, L. (2007). An accurate elasto-plastic frictional tangential force–displacement model for granular-flow simulations: Displacement-driven formulation. *Journal of Computational Physics*, *225*(*1*), 730–752. 1249, 1252

303. Vu-Quoc, L., Zhang, X. (1999). An accurate and efficient tangential force–displacement model for elastic frictional contact in particle-flow simulations. *Mechanics of Materials*, *31*(*4*), 235–269. 1252

304. Vu-Quoc, L., Zhang, X., Lesburg, L. (2001). Normal and tangential force–displacement relations for frictional elasto-plastic contact of spheres. *International Journal of Solids and Structures*, *38*(*36-37*), 6455–6489. 1252

305. Haghighat, E., Raissi, M., Moure, A., Gomez, H., Juanes, R. (2021). A physics-informed deep learning framework for inversion and surrogate modeling in solid mechanics. *Computer Methods in Applied Mechanics and Engineering*, *379*, 113741. 1252

306. Zhai, Y., Vu-Quoc, L. (2007). Analysis of power magnetic components with nonlinear static hysteresis: Proper orthogonal decomposition and model reduction. *IEEE Transactions on Magnetics*, *43*(*5*), 1888–1897. 1252, 1256, 1260

307. Benner, P., Gugercin, S., Willcox, K. (2015). A Survey of Projection-Based Model Reduction Methods for Parametric Dynamical Systems. *SIAM Review*, *57*(*4*), 483–531. 1261, 1263, 1271

308. Greif, C., Urban, K. (2019). Decay of the Kolmogorov N-width for wave problems. *Applied Mathematics Letters*, *96*, 216–222. Original website. 1261

309. Craig, R. R., Bampton, M. C. C. (1968). Coupling of substructures for dynamic analyses. *AIAA Journal*, *6*(*7*), 1313–1319. 1263

310. Chaturantabut, S., Sorensen, D. C. (2010). Nonlinear Model Reduction via Discrete Empirical Interpolation. *SIAM Journal on Scientific Computing*, *32*(*5*), 2737–2764. 1267, 1268

311. Carlberg, K., Bou-Mosleh, C., Farhat, C. (2011). Efficient non-linear model reduction via a least-squares Petrov-Galerkin projection and compressive tensor approximations. *International Journal for Numerical Methods in Engineering*, *86*(*2*), 155–181. 1267, 1268

312. Choi, Y., Coombs, D., Anderson, R. (2020). SNS: A Solution-Based Nonlinear Subspace Method for Time-Dependent Model Order Reduction. *SIAM Journal on Scientific Computing*, *42*(*2*), A1116–A1146. 1267

313. Everson, R., Sirovich, L. (1995). Karhunen-Loève procedure for gappy data. *Journal of the Optical Society of America A*, *12*(*8*), 1657. 1267

314. Carlberg, K., Farhat, C., Cortial, J., Amsallem, D. (2013). The GNAT method for nonlinear model reduction: Effective implementation and application to computational fluid dynamics and turbulent flows. *Journal of Computational Physics*, *242*, 623–647. 1268

315. Tiso, P., Rixen, D. J. (2013). Discrete empirical interpolation method for finite element structural dynamics. 1271

316. Brooks, A. N., Hughes, T. J. (1982). Streamline upwind/petrov-galerkin formulations for convection dominated flows with particular emphasis on the incompressible navier-stokes equations. *Computer Methods in Applied Mechanics and Engineering*, *32*(*1*), 199–259. Original website. 1272

317. Kochkov, D., Smith, J. A., Alieva, A., Wang, Q., Brenner, M. P., et al. (2021). Machine learning–accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, *118*(*21*), e2101784118. 1276

318. Bishara, D., Xie, Y., Liu, W. K., Li, S. (2023). A state-of-the-art review on machine learning-based multiscale modeling, simulation, homogenization and design of materials. *Archives of Computational Methods in Engineering*, *30*(*1*), 191–222. 1277

319. Rosenblatt, F. (1960). Perceptron simulation experiments. *Proceedings of the Institute of Radio Engineers*, *48*(*3*), 301–309. 1278

320. Block, H., Knight, B., Rosenblatt, F. (1962b). Analysis of a 4-layer series-coupled perceptron .2. *Reviews of Modern Physics*, *34*(*1*), 135–142. 1278

321. Gopnik, A. (2019). The ultimate learning machines. *The Wall Street Journal*, Oct 11. Original website. 1283, 1309

322. Hodgkin, A., Huxley, A. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, *117*(*4*), 500–544. 1286, 1288

323. Dimirovski, G. M., Wang, R., Yang, B. (2017). Delay and recurrent neural networks: Computational cybernetics of systems biology?In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE. Original website. 1287

324. Gherardi, F., Souty-Grosset, C., Vogt, G., Dieguez-Uribeondo, J., Crandall, K. (2009). Infraorder astacidea latreille, 1802 p.p.: The freshwater crayfish. In F. Schram, C. von Vaupel Klein, editors, *Treatise on Zoology - Anatomy, Taxonomy, Biology. The Crustacea, Volume 9 Part A*, chapter 67. Leiden, Netherlands: Brill, 269–423. 1288

325. Han, J., Moraga, C. The Influence of the Sigmoid Function Parameters on the Speed of Backpropagation Learning. In *IWANN '96 Proceedings of the International Workshop on Artificial Neural Networks: From Natural to Artificial Neural Computation, Jun 07-09*. 1288

326. Furshpan, E., Potter, D. (1959a). Transmission at the giant motor synapses of the crayfish. *Journal of Physiology-London*, *145*(*2*), 289–325. 1289, 1290

327. Bush, P., Sejnowski, T. (1995). *The Cortical Neuron*. Oxford University Press. 1289

328. Werbos, P. (1990). Backpropagation through time - what it does and how to do it. *Proceedings of the IEEE*, *78*(*10*), 1550–1560. 1291, 1293

329. Baydin, A. G., Pearlmutter, B. A., Radul, A. A., Siskind, J. M. (2018). Automatic Differentiation in Machine Learning: a Survey. *Journal of Machine Learning Research*, *18*. 1291, 1293

330. Werbos, P. J., Davis, J. J. J. (2016). Regular Cycles of Forward and Backward Signal Propagation in Prefrontal Cortex and in Consciousness. *Frontiers in Systems Neuroscience*, *10*. 1293

331. Metz, C. (2019). Turing Award Won by 3 Pioneers in Artificial Intelligence. *New York Times*, (*Mar 27*). Original website. 1293

332. Topol, E. (2019). The A.I. Diet. *New York Times*, (*Mar 02*). Original website. 1294

333. Laguarta, J., Hueto, F., Subirana, B. (2020). Covid-19 artificial intelligence diagnosis using only cough recordings. *IEEE Open Journal of Engineering in Medicine and Biology*. 1295, 1296

334. Heaven, W. (2021). Hundreds of ai tools have been built to catch covid. none of them helped. *MIT Technological Review*. July 30. 1294, 1295, 1296

335. Wynants, L., Van Calster, B., Collins, G. S., Riley, R. D., Heinze, G., et al. (2021). Prediction models for diagnosis and prognosis of covid-19: systematic review and critical appraisal. *BMJ*, *369*. 1294

336. Roberts, M., Driggs, D., Thorpe, M., Gilbey, J., Yeung, M., et al. (2021). Common pitfalls and recommendations for using machine learning to detect and prognosticate for covid-19 using chest radiographs and ct scans. *Nature Machine Intelligence*, *3*(*3*), 199–217. 1294

337. Moons, K. G., de Groot, J. A., Bouwmeester, W., Vergouwe, Y., Mallett, S., et al. (2014). Critical appraisal and data extraction for systematic reviews of prediction modelling studies: the charms checklist. *PLoS Medicine*, *11*(*10*), e1001744. 1294

338. Wolff, R. F., Moons, K. G., Riley, R. D., Whiting, P. F., Westwood, M., et al. (2019). Probast: a tool to assess the risk of bias and applicability of prediction model studies. *Annals of Internal Medicine*, *170*(*1*), 51–58. 1294

339. Matei, A. (2020). An app could catch 98.5% of all Covid-19 infections. Why isn't it available? *The Guardian*, (*Dec 16*). Original website. 1296

340. Coppock, H., Jones, L., Kiskin, I., Schuller, B. (2021). Covid-19 detection from audio: seven grains of salt. *The Lancet Digital Health*, *3*(*9*), e537–e538. 1296

341. Guo, X., Zhang, Y. D., Lu, S., Lu, Z. (2022). A Survey on Machine Learning in COVID-19 Diagnosis. *CMES-Computer Modeling in Engineering & Sciences*, *130*(*1*), 23–71. 1296, 1297

342. Li, W., Deng, X., Shao, H., Wang, X. (2021). Deep Learning Applications for COVID-19 Analysis: A State-of-the-Art Survey. *CMES-Computer Modeling in Engineering & Sciences*, *129*(*1*), 65–98. 1296, 1297

343. Xie, S., Yu, Z., Lv, Z. (2021). Multi-Disease Prediction Based on Deep Learning: A Survey. *CMES-Computer Modeling in Engineering & Sciences*, *128*(*2*), 489–522. 1296, 1297

344. Gong, L., Zhang, X., Zhang, L., Gao, Z. (2021). Predicting Genotype Information Related to COVID-19 for Molecular Mechanism Based on Computational Methods. *CMES-Computer Modeling in Engineering & Sciences*, *129*(*1*), 31–45. 1297

345. Monajjemi, M., Esmkhani, R., Mollaamin, F., Shahriari, S. (2020). Prediction of Proteins Associated with COVID-19 Based Ligand Designing and Molecular Modeling. *CMES-Computer Modeling in Engineering & Sciences*, *125*(*3*), 907–926. 1297

346. Attaallah, A., Ahmad, M., Seh, A. H., Agrawal, A., Kumar, R., et al. (2021). Estimating the Impact of COVID-19 Pandemic on the Research Community in the Kingdom of Saudi Arabia. *CMES-Computer Modeling in Engineering & Sciences*, *126*(*1*), 419–436. 1297

347. Gupta, M., Jain, R., Gupta, A., Jain, K. (2020). Real-Time Analysis of COVID-19 Pandemic on Most Populated Countries Worldwide. *CMES-Computer Modeling in Engineering & Sciences*, *125*(*3*), 943–965. 1297

348. Areepong, Y., Sunthornwat, R. (2020). Predictive Models for Cumulative Confirmed COVID-19 Cases by Day in Southeast Asia. *CMES-Computer Modeling in Engineering & Sciences*, *125*(*3*), 927–942. 1297

349. Singh, A., Bajpai, M. K. (2020). SEIHCRD Model for COVID-19 Spread Scenarios, Disease Predictions and Estimates the Basic Reproduction Number, Case Fatality Rate, Hospital, and ICU Beds Requirement. *CMES-Computer Modeling in Engineering & Sciences*, *125*(*3*), 991–1031. 1297

350. Akyol, K. (2020). Growing and Pruning Based Deep Neural Networks Modeling for Effective Parkinson's Disease Diagnosis. *CMES-Computer Modeling in Engineering & Sciences*, *122*(*2*), 619–632. 1297

351. Hemalakshmi, G. R., Santhi, D., Mani, V. R. S., Geetha, A., Prakash, N. B. (2020). Deep Residual Network Based on Image Priors for Single Image Super Resolution in FFA Images. *CMES-Computer Modeling in Engineering & Sciences*, *125*(*1*), 125–143. 1297

352. Vu-Quoc, L., Zhai, Y., Ngo, K. D. T. (2021). Model reduction by generalized Falk method for efficient field-circuit simulations. *CMES-Computer Modeling in Engineering & Sciences*, *129*(*3*), 1441–1486. DOI: 10.32604/cmes.2021.016784. 1297

353. Lu, Y., Li, H., Saha, S., Mojumder, S., Al Amin, A., et al. (2021). Reduced Order Machine Learning Finite Element Methods: Concept, Implementation, and Future Applications. *CMES-Computer Modeling in Engineering & Sciences*. DOI: 10.32604/cmes.2021.017719. 1297

354. Deng, X., Shao, H., Hu, C., Jiang, D., Jiang, Y. (2020). Wind Power Forecasting Methods Based on Deep Learning: A Survey. *CMES-Computer Modeling in Engineering & Sciences*, *122*(*1*), 273–301. 1297

355. Liu, D., Zhao, J., Xi, A., Wang, C., Huang, X., et al. (2020). Data Augmentation Technology Driven By Image Style Transfer in Self-Driving Car Based on End-to-End Learning. *CMES-Computer Modeling in Engineering & Sciences*, *122*(*2*), 593–617. 1297

356. Sethi, S., Kathuria, M., Kaushik, T. (2021). A Real-Time Integrated Face Mask Detector to Curtail Spread of Coronavirus. *CMES-Computer Modeling in Engineering & Sciences*, *127*(*2*), 389–409.

1298

357. Luo, J., Li, Y., Zhou, W., Gong, Z., Zhang, Z., et al. (2021). An Improved Data-Driven Topology Optimization Method Using Feature Pyramid Networks with Physical Constraints. *CMES-Computer Modeling in Engineering & Sciences*, *128*(*3*), 823–848. 1298

358. Qu, T., Di, S., Feng, Y. T., Wang, M., Zhao, T., et al. (2021). Deep Learning Predicts Stress-Strain Relations of Granular Materials Based on Triaxial Testing Data. *CMES-Computer Modeling in Engineering & Sciences*, *128*(*1*), 129–144. 1298

359. Li, H., Zhang, Q., Chen, X. (2021). Deep Learning-Based Surrogate Model for Flight Load Analysis. *CMES-Computer Modeling in Engineering & Sciences*, *128*(*2*), 605–621. 1298

360. Guo, D., Yang, Q., Zhang, Y. D., Jiang, T., Yan, H. (2021). Classification of Domestic Refuse in Medical Institutions Based on Transfer Learning and Convolutional Neural Network. *CMES-Computer Modeling in Engineering & Sciences*, *127*(*2*), 599–620. 1298

361. Yang, F., Zhang, X., Zhu, Y. (2020). PDNet: A Convolutional Neural Network Has Potential to be Deployed on Small Intelligent Devices for Arrhythmia Diagnosis. *CMES-Computer Modeling in Engineering & Sciences*, *125*(*1*), 365–382. 1298

362. Yin, C., Han, J. (2021). Dynamic Pricing Model of E-Commerce Platforms Based on Deep Reinforcement Learning. *CMES-Computer Modeling in Engineering & Sciences*, *127*(*1*), 291–307. 1298

363. Zhang, Y., Ran, X. (2021). A Step-Based Deep Learning Approach for Network Intrusion Detection. *CMES-Computer Modeling in Engineering & Sciences*, *128*(*3*), 1231–1245. 1298

364. Park, J., Lee, J. H., Bang, J. (2021). PotholeEye plus: Deep-Learning Based Pavement Distress Detection System toward Smart Maintenance. *CMES-Computer Modeling in Engineering & Sciences*, *127*(*3*), 965–976. 1298

365. Mu, L., Zhao, H., Li, Y., Liu, X., Qiu, J., et al. (2021). Traffic Flow Statistics Method Based on Deep Learning and Multi-Feature Fusion. *CMES-Computer Modeling in Engineering & Sciences*, *129*(*2*), 465–483. 1298

366. Wang, J., Peng, K. (2020). A Multi-View Gait Recognition Method Using Deep Convolutional Neural Network and Channel Attention Mechanism. *CMES-Computer Modeling in Engineering & Sciences*, *125*(*1*), 345–363. 1298

367. Shi, D., Zheng, H. (2021). A Mortality Risk Assessment Approach on ICU Patients Clinical Medication Events Using Deep Learning. *CMES-Computer Modeling in Engineering & Sciences*, *128*(*1*), 161–181. 1298

368. Bian, J., Li, J. (2021). Stereo Matching Method Based on Space-Aware Network Model. *CMES-Computer Modeling in Engineering & Sciences*, *127*(*1*), 175–189. 1298

369. Kong, W., Wang, B. (2020). Combining Trend-Based Loss with Neural Network for Air Quality Forecasting in Internet of Things. *CMES-Computer Modeling in Engineering & Sciences*, *125*(*2*), 849–863. 1298

370. Jothiramalingam, R., Jude, A., Hemanth, D. J. (2021). Review of Computational Techniques for the Analysis of Abnormal Patterns of ECG Signal Provoked by Cardiac Disease. *CMES-Computer Modeling in Engineering & Sciences*, *128*(*3*), 875–906. 1298

371. Yang, J., Xin, L., Huang, H., He, Q. (2021). An Improved Algorithm for the Detection of Fastening Targets Based on Machine Vision. *CMES-Computer Modeling in Engineering & Sciences*, *128*(*2*), 779–802. 1298

372. Dong, J., Liu, J., Wang, N., Fang, H., Zhang, J., et al. (2021). Intelligent Segmentation and Measurement Model for Asphalt Road Cracks Based on Modified Mask R-CNN Algorithm. *CMES-Computer Modeling in Engineering & Sciences*, *128*(*2*), 541–564. 1298

373. Chen, M., Luo, X., Shen, H., Huang, Z., Peng, Q. (2021). A Novel Named Entity Recognition Scheme for Steel E-Commerce Platforms Using a Lite BERT. *CMES-Computer Modeling in Engineering &*

Sciences, *129*(*1*), 47–63. 1298

374. Zhang, X., Zhang, Q. (2020). Short-Term Traffic Flow Prediction Based on LSTM-XGBoost Combination Model. *CMES-Computer Modeling in Engineering & Sciences*, *125*(*1*), 95–109. 1298

375. Lu, X., Zhang, H. (2020). An Emotion Analysis Method Using Multi-Channel Convolution Neural Network in Social Networks. *CMES-Computer Modeling in Engineering & Sciences*, *125*(*1*), 281–297. 1298

376. Safety Test Reveals Tesla's Full Self-Driving Software Repeatedly Hits Child-Sized Mannequin. The Dawn Project, 2022.08.09, Original website, Internet archived on 2022.08.17. 1298

377. Helmore, E. (2022). Tesla's self-driving technology fails to detect children in the road, tests find. *The Guardian*, (*Aug 09*). Original website. 1298, 1299, 1300

378. Does Tesla Full Self-Driving Beta really run over kids? Whole Mars Catalog, 2022.08.14, Tweet. 1298

379. Roth, E. (2022). YouTube removes video that tests Tesla's Full Self-Driving beta against real kids. *The Verge*, (*Aug 20*). Original website. 1298

380. Musk's Full Self-Driving @Tesla ruthlessly mowing down a child mannequin. Dan O'Dowd, The Dawn Project, 2022.08.15, Tweet. 1299

381. Hawkins, A. J. (2022). Tesla wants videos of its cars running over child-sized dummies taken down. *The Verge*, (*Aug 25*). Original website. 1299

382. Metz, C., Koeze, E. (2022). Can Tesla Data Help Us Understand Car Crashes? *New York Times*, (*Aug 18*). Original website. 1300, 1301, 1302, 1303

383. Metz, C. (2017). A New Way for Machines to See, Taking Shape in Toronto. *New York Times*, (*Nov 28*). Original website. 1298

384. Dujmovic, J. (2021). You will not be traveling in a self-driving car anytime soon. Here's what the future will look like. *Market Watch*, (*June 16 - Updated June 19*). Original website. 1298, 1300

385. Maresca, T. (2022). Hyundai's self-driving taxis roll out on the streets of South Korea. *UPI*, (*Jun 09*). Original website. 1299

386. Kirkpatrick, K. (2022). Still Waiting for Self-Driving Cars. *Communications of the ACM*, (*April*). Original website. 1299, 1300

387. Bogna, J. (2022). Is Your Car Autonomous? The 6 Levels of Self-Driving Explained. *PC Magazine*, (*June 14*). Original website. 1299

388. Boudette, N. (2019). Despite High Hopes, Self-Driving Cars Are 'Way in the Future'. *New York Times*, (*Jul 07*). Original website. 1300

389. Guinness, H. (2022). What's going on with self-driving cars right now? *Popular Science*, (*May 28*). Original website. 1301

390. Smiley, L. (2022). 'I'm the Operator': The Aftermath of a Self-Driving Tragedy. *WIRED*, (*Mar 8*). Original website. 1301, 1302

391. Metz, C., Griffith, E. (2022). This Was Supposed to Be the Year Driverless Cars Went Mainstream. *New York Times*, (*May 12 - Updated Sep 15*). Original website. 1302

392. Metz, C. (2022). The Costly Pursuit of Self-Driving Cars Continues On. And On. And On. *New York Times*, (*May 24 - Updated Sep 15*). Original website. 1302

393. Nims, C. (2020). Robot Boats Leave Autonomous Cars in Their Wake — Unmanned ships don't have to worry about crowded roads. But crossing of the Atlantic is still a challenge. *Wall Street Journal*, (*Aug 29*). 1303

394. O'Brien, M. (2022). Autonomous Mayflower reaches American shores — in Canada. *ABC News*, (*Jun 05*). Original website. 1303, 1304

395. Mitchell, M. (2018). Artificial intelligence hits the barrier of meaning. *The New York Times*. 1304

396. New Survey: Americans Think AI Is a Threat to Democracy, Will Become Smarter than Humans and

Overtake Jobs, Yet Believe its Benefits Outweigh its Risks. Stevens Institute of Technology, 2021 Nov 15, Website, Internet archive. 1306

397. Hu, S., Li, Y., Lyu, S. (2020). Exposing gan-generated faces using inconsistent corneal specular highlights. arXiv:2009.11924. 1306

398. Sencar, H. T., Verdoliva, L., Memon, N. (2022). Multimedia forensics. 1306

399. Chesney, R., Citron, D. K. (2019). Deep Fakes: A Looming Challenge for Privacy, Democracy, and National Security. *107 California Law Review 1753*. Original website U of Texas Law, Public Law Research Paper No. 692. U of Maryland Legal Studies Research Paper No. 2018-21. 1306, 1307

400. Ingram, D., Ward, J. (2019). How do you spot a deepfake? A clue hides within our voices, researchers say. *NBC News*, (*Dec 16*). Original website. 1306, 1307

401. Citron, D. How deepfakes undermine truth and threaten democracy. TEDSummit 2019. Website. 1307

402. Manheim, K., Kaplan, L. (2019). Artificial Intelligence: Risks to Privacy and Democracy. *Yale Journal of Law & Technology*, *21*, 106–188. Original website. 1307

403. Hao, K. (2019). Why AI is a threat to democracy—and what we can do to stop it. *MIT Technology Review*, (*Feb 26*). Original website. 1307

404. Feldstein, S. (2019). How Artificial Intelligence Systems Could Threaten Democracy. *Carnegie Endowment for International Peace*, (*Apr 24*). Original website. 1307

405. Pearce, G. (2021). Beware the Privacy Violations in Artificial Intelligence Applications. *ISACA Now Blog*, (*May 28*). Original website. 1307

406. Harwell, D. (2019). Top AI researchers race to detect 'deepfake' videos: 'We are outgunned'. *Washington Post*, (*Jun 12*). Original website. 1307

407. Deepfake Detection Challenge: Identify videos with facial or voice manipulations. 2019-2020, Overview, Leaderboard. 1307

408. Groh, M., Epstein, Z., Firestone, C., Picard, R. (2022). Deepfake detection by human crowds, machines, and machine-informed crowds. *Proceedings of the National Academy of Sciences*, *119(1)*, e2110013119. Original website. 1307, 1308

409. Hintze, J. L., Nelson, R. D. (1998). Violin plots: a box plot-density trace synergism. *The American Statistician*, *52(2)*, 181–184. JSTOR. 1307

410. Lewinson, E. (2019). Violin plots explained. *Toward Data Science*, (*Oct 21*). Original website, GitHub. 1307

411. Detect DeepFakes: How to counteract misinformation created by AI. MIT Media Lab, project contact Matt Groh. Website, Internet archive. 1307

412. Hill, K. (2020). The Secretive Company That Might End Privacy as We Know It. *New York Times*, (*Feb 10, Updated 2021 Nov 2*). Original website. 1308

413. Morrison, S. (2020). The world's scariest facial recognition company is now linked to everybody from ICE to Macy's. *Vox*, (*Feb 28*). Original website. 1308

414. Hill, K. (2020). Wrongfully Accused by an Algorithm. *New York Times*, (*Jun 24, Updated Aug 03*). Original website. 1309

415. Raji, I. D., Smart, A., White, R. N., Mitchell, M., Gebru, T., et al. (2020). Closing the AI accountability gap: Defining an end-to-end framework for internal algorithmic auditingIn *Proceedings of the 2020 conference on fairness, accountability, and transparency*. Original website. 1309

416. Metz, C. (2021). Who Is Making Sure the A.I. Machines Aren't Racist? *New York Times*, (*Mar 15*). Original website. 1309

417. Samuel, S. (2022). Why it's so damn hard to make AI fair and unbiased. *Vox*, (*Apr 19*). Future Perfect, Original website. 1309

418. Heilinger, J. C. (2022). The Ethics of AI Ethics. A Constructive Critique. *Philosophy & Technology*, *35(3)*, 1–20. Original website. 1309

419. Heikkilä, M. (2022). The walls are closing in on Clearview AI as data watchdogs get tough. *MIT Technology Review*, (*May 24*). Original website. 1309

420. Metz, C., Isaac, M. (2019). Facebook's A.I. Whiz Now Faces the Task of Cleaning It Up. Sometimes That Brings Him to Tears. *New York Times*, (*May 17*). Original website. 1309

421. Haibe-Kains, B., Adam, G. A., Hosny, A., Khodakarami, F., Massive Analysis Quality Control (MAQC) Society Board of Directors, et al. (2020). Transparency and reproducibility in artificial intelligence. *Nature*. Oct 14. doi.org/10.1038/s41586-020-2766-y. 1310, 1311

422. Heaven, W. D. (2020). AI is wrestling with a replication crisis. *MIT Technology Review*, (*Aug 29*). Original website. 1310, 1311

423. McKinney, S. M., Sieniek, M., Godbole, V., Godwin, J., Antropova, N., et al. (2020). International evaluation of an AI system for breast cancer screening. *Nature*, *577*(*7788*), 89–94. 1311

424. Trevithick, J. (2020). General Atomics Avenger Drone Flew An Autonomous Air-To-Air Mission Using An AI Brain. *The Drive*, (*Dec 4*). Original website. Internet Archive. 1311

425. Sepulchre, R. (2020). Cybernetics [From the Editor]. *IEEE Control Systems Magazine*, *40*(*2*), 3–4. 1339

426. Copeland, A. (1949). A cybernetic model of memory and recognition. *Bulletin of the American Mathematical Society*, *55*(*7*), 698. 1340

427. Chavalarias, D. (2020). From inert matter to the global society life as multi-level networks of processes. *Philosophical Transactions of the Royal Society B-Biological Sciences*, *375*(*1796*). 1340

428. Togashi, E., Miyata, M., Yamamoto, Y. (2020). The first world championship in cybernetic building optimization. *Journal of Building Performance Simulation*, *13*(*3*), 391–408. 1340

429. Jube, S. (2020). Labour and international accounting standards: A question of social justice. *International Labour Review*. Early Access Date MAR 2020. 1340

430. McCulloch, W., Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, *5*, 115–133. Reprinted in the *Bulletin of Mathematical Biology*, Vol.52, No.1-2, pp.99-115, 1990. 1340, 1343

431. Kline, R. (2015). *The Cybernetic Moment, or why we call our Age the Information Age*. Baltimore: Johns Hopkins University Press. 1340, 1341, 1342, 1343

432. Cariani, P. (2017). The Cybernetics Moment: or Why We Call Our Age the Information Age. *Cognitive Systems Research*, *43*, 119–124. 1340, 1341

433. Eisenhart, C. (1949). Cybernetics - A new discipline. *Science*, *109*(*2834*), 397–399. 1340

434. W. E. H. (1949). Book Review: Cybernetics: Or Control and Communication in the Animal and the Machine. *Quarterly Journal of Experimental Psychology*, *1*(*4*), 193–194. https://doi.org/10.1080/17470214908416765. 1341

Appendices

1 Backprop pseudocodes, notation comparison

To connect the backpropagation Algorithm 1 in Section 5 to Algorithm 6.4 in [78], p.206, Section 6.5.4 on "Back-Propagation Computation in Fully Connected MLP",[338] a different form of Algorithm 1, where the "while" loop is used, is provided in Algorithm 9, where the "for" loop is used. This information would be especially useful for first-time learners. See also Remark 5.4.

1 Backpropagation pseudocode 2
Data:
- Input into network $x = y^{(0)} \in \mathbb{R}^{m_{(0)} \times 1}$
- Learning rate ϵ; see Section 6.2 on deterministic optimization
- Results from *any* forward propagation:
    - ⋆ Network parameters $\theta = \{\theta^{(1)}, \cdots, \theta^{(L)}\}$ (all layers)
    - ⋆ Layer weighted inputs and biases $z^{(\ell)}$, for $\ell = 1, \cdots, L$
    - ⋆ Layer outputs $y^{(\ell)}$, for $\ell = 1, \cdots, L$

Result: Updated network parameters θ to reduce cost function J.

2 Initialize:
3 • Gradient $r = \partial J/\partial y^{(L)} \in \mathbb{R}^{1 \times m_{(L)}}$ (row) on predicted output $\widetilde{y} = y^{(L)}$, Eq.(99) ;

4 for $\ell = L, \ldots, 1$ **do**
5 ▶ Compute gradient on weighted inputs (pre-nonlinear activation) $z^{(\ell)}$:
6 $r \leftarrow \partial J/\partial z^{(\ell)} = r \odot a'(z^{(\ell)})^T \in \mathbb{R}^{1 \times m_{(\ell)}}$ (row), Eq.(104) ;
7 ▶ Compute gradient on layer parameters $\theta^{(\ell)}$:
8 $\partial J/\partial \theta^{(\ell)} = r^T \overline{y}^{(\ell-1)T} \in \mathbb{R}^{m_{(\ell)} \times [m_{(\ell-1)}+1]}$ (row), Eq.(105) ;
9 ▶ Compute gradient on layer outputs $y^{(\ell-1)}$:
10 $r \leftarrow \partial J/\partial y^{(\ell-1)} = r W^{(\ell)} \in \mathbb{R}^{1 \times m_{(\ell-1)}}$ (row), Eq.(106) ;
11 end

Algorithm 9: *Backpropagation pseudocode.* Alternative presentation of Algorithm 1, where the "while" loop is used, to compare to Algorithm 6.4 in [78], p.206, where the "for" loop was used, and where there was no layer-parameter update step (Line 8 of Algorithm 1). See also Table 8 for the correspondence between the notations employed here and those in [78], and the block diagrams in Figure 47 and Figure 48.

In Algorithm 9, the regularization of the cost function J is not considered, i.e., we omit the penalty (or regularization) term $\lambda\Omega(\theta)$ used in Algorithm 6.4 in [78], p.206, where the regularized cost was $J(\theta) + \lambda\Omega(\theta)$. As pointed out in Section 6.3.6, weight decay is more general than L_2 regularization, and would be the preferred method to avoid overfitting.

In Table 8, the correspondence between the notations employed here and those in [78], p.206, is provided.

---

[338] MLP = MultiLayer Perceptron.

Table 8: Equivalence of backprop Algorithm 9 and Algorithm 6.4 in [78], p.206. Comparison of notations. The mathematical expressions in Algorithm 6.4 are reproduced here in their original notations, except for the matrix dimensions, which were not given in Algorithm 6.4 of [78].

Algorithm 9, current notation	Goodfellow Algorithm 6.4, original notation
• Layer index = ℓ	• Layer index = k
• Gradient = $\boldsymbol{r} \in \mathbb{R}^{1 \times m_{(\ell)}}$ (row)	• Gradient = $\boldsymbol{g} \in \mathbb{R}^{m_{(k)} \times 1}$ (column)
• Output of layer $(\ell) = \boldsymbol{y}^{(\ell)} \in \mathbb{R}^{m_{(\ell)} \times 1}$ (column)	• Output of layer $(k) = \boldsymbol{h}^{(k)} \in \mathbb{R}^{m_{(k)} \times 1}$ (column)
• Predicted output (last layer (L)): $\widetilde{\boldsymbol{y}} = \boldsymbol{y}^{(L)}$	• Predicted output (last layer (l)): $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
• Weighted inputs to layer (ℓ): $\boldsymbol{z}^{(\ell)}$	• Pre-nonlinear activation for layer (k): $\boldsymbol{a}^{(k)}$
• Activation function: $a(\cdot)$	• Activation function: $f(\cdot)$
• Gradient on output of layer (ℓ): $\partial J/\partial \boldsymbol{y}^{(\ell)} \in \mathbb{R}^{1 \times m_{(\ell)}}$ (row)	• Gradient on output of layer (k): $\nabla_{\boldsymbol{h}^{(k)}} J \in \mathbb{R}^{m_{(k)} \times 1}$ (column)
• Gradient on weighted inputs: $\boldsymbol{r}^{(\ell)} := \dfrac{\partial J}{\partial \boldsymbol{z}^{(\ell)}} = \dfrac{\partial J}{\partial \boldsymbol{y}^{(\ell)}} \odot a'(\boldsymbol{z}^{(\ell)T}) \in \mathbb{R}^{1 \times m_{(\ell)}}$ (row) Code line 6 in Algorithm 9 $\boldsymbol{r} \leftarrow \partial J/\partial \boldsymbol{z}^{(\ell)} = \boldsymbol{r} \odot a'(\boldsymbol{z}^{(\ell)})^T \in \mathbb{R}^{1 \times m_{(\ell)}}$ (row)	• Gradient on prenonlinear activation: $\boldsymbol{g} = \nabla_{\boldsymbol{a}^{(k)}} J = \nabla_{\boldsymbol{h}^{(k)}} J \odot f'(\boldsymbol{a}^{(k)}) \in \mathbb{R}^{m_{(k)} \times 1}$ (col) Code in Algorithm 6.4, [78] $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)}) \in \mathbb{R}^{m_{(k)} \times 1}$ (column)
• Layer parameters $\boldsymbol{\theta}^{(\ell)} = \left[\boldsymbol{W}^{(\ell)}, \boldsymbol{b}^{(\ell)} \right] \in \mathbb{R}^{m_{(\ell)} \times [m_{(\ell-1)}+1]}$	• Layer parameters $\left[\boldsymbol{W}^{(k)}, \boldsymbol{b}^{(k)} \right] \in \mathbb{R}^{m_{(k)} \times [m_{(k-1)}+1]}$
• Gradient on layer parameters $\boldsymbol{\theta}^{(\ell)}$ $\dfrac{\partial J}{\partial \boldsymbol{\theta}^{(\ell)}} = \left[\dfrac{\partial J}{\partial \boldsymbol{W}^{(\ell)}} \,\Big|\, \dfrac{\partial J}{\partial \boldsymbol{b}^{(\ell)}} \right] \in \mathbb{R}^{m_{(\ell)} \times [m_{(\ell-1)}+1]}$ $\partial J/\partial \boldsymbol{\theta}^{(\ell)} = \boldsymbol{r}^T \overline{\boldsymbol{y}}^{(\ell-1)T} = \left[\boldsymbol{r}^T \boldsymbol{y}^{(\ell-1)T} \,\big|\, \boldsymbol{r}^T \right]$ $\partial J/\partial \boldsymbol{W}^{(\ell)} = \boldsymbol{r}^T \boldsymbol{y}^{(\ell-1)T}$ and $\partial J/\partial \boldsymbol{b}^{(\ell)} = \boldsymbol{r}^T$	• Gradient on layer parameters $\left[\boldsymbol{W}^{(k)}, \boldsymbol{b}^{(k)} \right]$ $\left[\nabla_{\boldsymbol{W}^{(k)}} J \,\big|\, \nabla_{\boldsymbol{b}^{(k)}} J \right] \in \mathbb{R}^{m_{(k)} \times [m_{(k-1)}+1]}$ $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g} \boldsymbol{h}^{(k-1)T}$ and $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g}$ (no regularization in cost function, $\boldsymbol{g} = \boldsymbol{r}^T$)
• Gradient on layer outputs $\boldsymbol{y}^{(\ell-1)}$ $\boldsymbol{r} \leftarrow \partial J/\partial \boldsymbol{y}^{(\ell-1)} = \boldsymbol{r} \boldsymbol{W}^{(\ell)} \in \mathbb{R}^{1 \times m_{(\ell-1)}}$ (row)	• Gradient on layer outputs $\boldsymbol{h}^{(k-1)}$ $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T} \boldsymbol{g} \in \mathbb{R}^{m_{(k-1)} \times 1}$ (column)

2 Another LSTM block diagram

An alternative block diagram for the folded RNN with LSTM cell corresponding to Figure 81 is shown Figure 152 below:
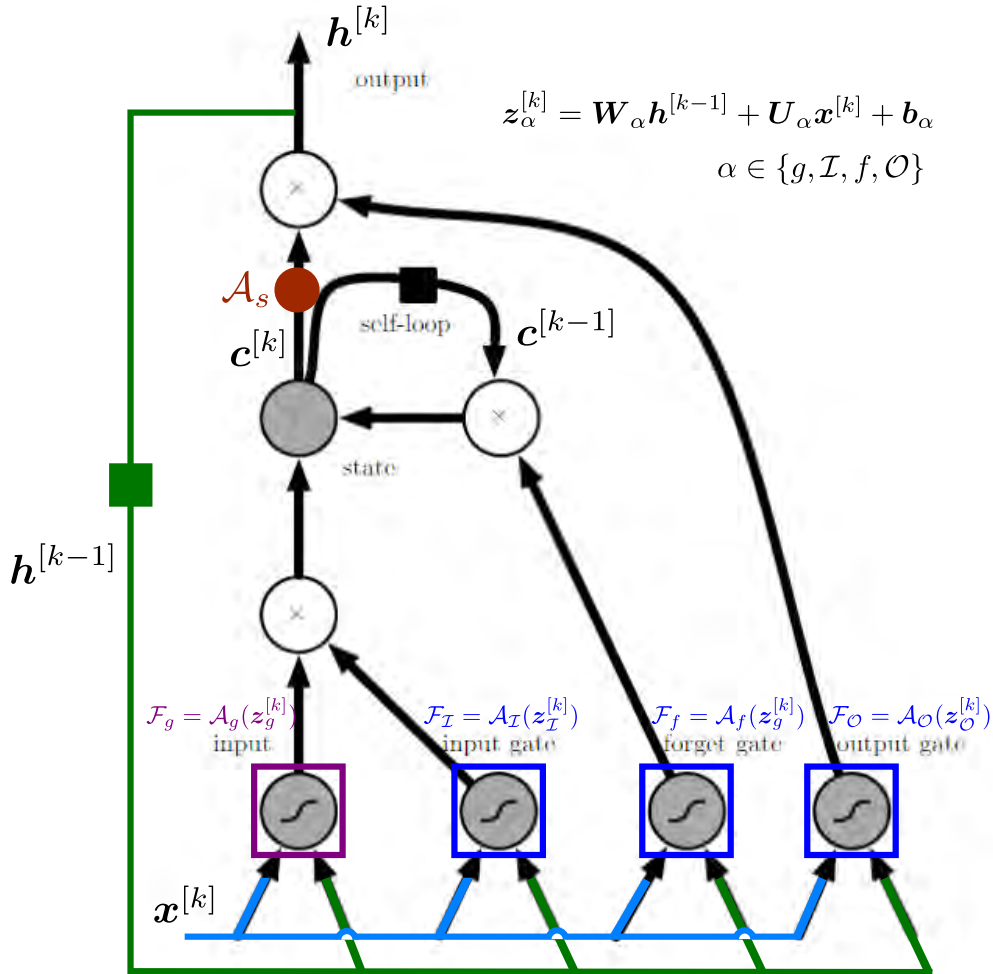

Figure 152: *Folded RNN and LSTM cell, two feedback loops with delay, block diagram.* Typical state $[k]$. Corrections for the figures in (1) Figure 10.16 in the online book Deep Learning by Goodfellow et al 2016, Chap.10, p.405 (referred to here as "DL-A", or "Deep Learning, version A"), and (2) [78], p.398 (referred to as "DL-B"). (The above figure is adapted from a figure reproduced with permission of the authors.)

Figure 10.16 in the online book Deep Learning by Goodfellow et al. (2016), Chap.10, p.405 (referred to here as "DL-A", or "Deep Learning, version A"), was either incomplete with missing important details. Even the *updated* Figure 10.16 in [78], p.398 (referred to as "DL-B"), was still incomplete (or incorrect).

The corrected arrows, added annotations, and colors correspond to those in the equivalent Figure 81. The corrections are described below.

Error 1: The cell state $c^{[k]}$ should be squashed by the state sigmoidal activation function $\mathcal{A}_s$ with range $(-1, +1)$ (brown dot, e.g., tanh function) before being multiplied by the scaling factor $\mathcal{F}_\mathcal{O}$ coming from the output gate to produce the hidden state $h^{[k]}$. This correction is for both DL-A and DL-B.

Error 2: The hidden-state feedback loop (green) should start from the hidden state $h^{[k]}$, delayed by one step, i.e., $h^{[k-1]}$, which is fed to all four gates: (1) The *externally-input* gate g (purple box) with activation function $\mathcal{A}_g$ having the range $(-1,+1)$, (2) the *input* gate $\mathcal{I}$, (3) the *forget* gate f, (4) the *output* gate $\mathcal{O}$. The activations $\mathcal{A}_\alpha$, with $\alpha \in \{\mathcal{I}, f, \mathcal{O}\}$ (3 blue boxes) all have the interval $(0,1)$ as their range. This hidden-state feedback loop was missing in DL-A, whereas the hidden-state feedback loop in DL-B incorrectly started from the summation operation (grey circle, just below $c^{[k]}$) in the cell-state feedback loop, and did not feed into the input gate g.

Error 3: Four pairs of arrows pointing into the four gates $\{g, \mathcal{I}, f, \mathcal{O}\}$, with one pair per gate, were intended as inputs to these gates, but were without annotation, and thus unclear / confusing. Here, for each gate, one arrow is used for the hidden state $h^{[k-1]}$, and the other arrow for the input $x^{[k]}$. This correction is for both DL-A and DL-B.

3 Conditional Gaussian distribution

The derivation of Eqs. (377)-(379) is provided here helps develop a better feel for the Gaussian distribution, and facilitates the understanding of the conditional Gaussian process posterior described in Section 8.3.2.

If two sets of variables have a joint Gaussian distribution, i.e., these two sets are jointly Gaussian, then the conditional probability distribution of one set given the other set is also Gaussian.[339] The two sets of variables considered here is the observed values in y and the test values in $\widetilde{y}$ in Eq. (376). Define

$$\widehat{y} := \begin{Bmatrix} y \\ \widetilde{y} \end{Bmatrix}, \quad \widehat{\mu} := \begin{Bmatrix} \mu \\ \widetilde{\mu} \end{Bmatrix}, \quad C_{\widehat{y}\widehat{y}} := \begin{bmatrix} C_{yy,\nu} & C_{y\widetilde{y}} \\ C_{\widetilde{y}y} & C_{\widetilde{y}\widetilde{y}} \end{bmatrix} = \begin{bmatrix} K(x,x) + \nu^2 I & K(x,\widetilde{x}) \\ K^T(x,\widetilde{x}) & K(\widetilde{x},\widetilde{x}) \end{bmatrix}, \tag{518}$$

$$C_{\widehat{y}\widehat{y}}^{-1} := \begin{bmatrix} C_{yy,\nu} & C_{y\widetilde{y}} \\ C_{\widetilde{y}y} & C_{\widetilde{y}\widetilde{y}} \end{bmatrix}^{-1} = \begin{bmatrix} D_{yy} & D_{y\widetilde{y}} \\ D_{y\widetilde{y}}^T & D_{\widetilde{y}\widetilde{y}} \end{bmatrix}, \text{ with } D_{y\widetilde{y}}^T = D_{\widetilde{y}y}, \tag{519}$$

then expand the exponent in the Gaussian joint probability Eq. (368), with $(\widehat{y} - \widehat{\mu})$ instead of $(y - \mu)$, to have the Mahalanobis distance[340] Δ squared, a quadratic form in terms of $(\widehat{y} - \widehat{\mu})$, written as:

$$\Delta^2 := (\widehat{y} - \widehat{\mu})^T C_{\widehat{y}\widehat{y}}^{-1} (\widehat{y} - \widehat{\mu}) = (y - \mu)^T D_{yy}(y - \mu) + (y - \mu)^T D_{y\widetilde{y}}(\widetilde{y} - \widetilde{\mu})$$

$$+ (\widetilde{y} - \widetilde{\mu})^T D_{\widetilde{y}y}(y - \mu) + (\widetilde{y} - \widetilde{\mu})^T D_{\widetilde{y}\widetilde{y}}(\widetilde{y} - \widetilde{\mu}), \tag{520}$$

which is also a quadratic form in terms of $(\widetilde{y} - \widetilde{\mu})$, based on the symmetry of $C_{\widehat{y}\widehat{y}}^{-1}$, the inverse of the covariance matrix, in Eq. (519), implying that the distribution of the test values $\widetilde{y}$ is Gaussian. The covariance matrix and the mean of the Gaussian distribution over $\widetilde{y}$ are determined by identifying the quadratic term and the linear term in $\widetilde{y}$, compared to the expansion of the general Gaussian distribution $\mathcal{N}(z|m, C)$ over the variable z with mean m and covariance matrix C:

$$(z - m)^T C^{-1} (z - m) = z^T C^{-1} z - 2z^T C^{-1} m + \text{ constant}, \tag{521}$$

where the constant is independent of z. Expand Eq. (520) to have

$$\Delta^2 = \widetilde{y}^T D_{\widetilde{y}\widetilde{y}} \widetilde{y} - 2\widetilde{y}^T \left[D_{\widetilde{y}\widetilde{y}} \widetilde{\mu} - D_{\widetilde{y}y}(y - \mu) \right] + \text{ constant}, \tag{522}$$

and compare to Eq. (521), then for the conditional distribution $p(\widetilde{y}|y)$ of the test values $\widetilde{y}$ at $\widetilde{x}$ given the

---

[339] See, e.g., [130], p. 85.
[340] See, e.g., [130], p. 80.

data (x, y), the covariance matrix $C_{\widetilde{y}|y}$ and the mean $\mu_{\widetilde{y}|y}$ are

$$C_{\widetilde{y}|y}^{-1} = D_{\widetilde{y}\widetilde{y}} \Rightarrow C_{\widetilde{y}|y} = D_{\widetilde{y}\widetilde{y}}^{-1} \tag{523}$$

$$C_{\widetilde{y}|y}^{-1}\mu_{\widetilde{y}|y} = \left[D_{\widetilde{y}\widetilde{y}}\widetilde{\mu} - D_{\widetilde{y}y}(y - \mu)\right] \Rightarrow \mu_{\widetilde{y}|y} = C_{\widetilde{y}|y}\left[D_{\widetilde{y}\widetilde{y}}\widetilde{\mu} - D_{\widetilde{y}y}(y - \mu)\right], \tag{524}$$

$$\Rightarrow \mu_{\widetilde{y}|y} = \widetilde{\mu} - D_{\widetilde{y}\widetilde{y}}^{-1}D_{\widetilde{y}y}(y - \mu), \tag{525}$$

in which Eq. (523)$_2$ had been used.

At this point, the submatrices $D_{\widetilde{y}\widetilde{y}}$ and $D_{\widetilde{y}y}$ can be expressed in terms of the submatrices of the partitioned matrix $C_{\widehat{y}\widehat{y}}$ in Eq. (518) as follows. From the definition of the matrix $D_{\widehat{y}\widehat{y}}$, the inverse of the covariance matrix $C_{\widehat{y}\widehat{y}}$:

$$D_{\widehat{y}\widehat{y}}C_{\widehat{y}\widehat{y}} = \begin{bmatrix} D_{yy} & D_{y\widetilde{y}} \\ D_{\widetilde{y}y} & D_{\widetilde{y}\widetilde{y}} \end{bmatrix} \begin{bmatrix} C_{yy} & C_{y\widetilde{y}} \\ C_{\widetilde{y}y} & C_{\widetilde{y}\widetilde{y}} \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix} \tag{526}$$

the 2nd row gives rise to a system of two equations for two unknowns $D_{\widehat{y}y}$ and $D_{\widehat{y}\widehat{y}}$:

$$\begin{bmatrix} D_{\widetilde{y}y} & D_{\widetilde{y}\widetilde{y}} \end{bmatrix} \begin{bmatrix} C_{yy} & C_{y\widetilde{y}} \\ C_{\widetilde{y}y} & C_{\widetilde{y}\widetilde{y}} \end{bmatrix} = \begin{bmatrix} 0 & I \end{bmatrix}, \tag{527}$$

in which the covariance matrix $C_{\widehat{y}\widehat{y}}$ is symmetric, and is a particular case of the non-symmetric problem of expressing (F, G) in terms of (P, Q, R, S):

$$\begin{bmatrix} F & G \end{bmatrix} \begin{bmatrix} P & Q \\ R & S \end{bmatrix} = \begin{bmatrix} 0 & I \end{bmatrix} \Rightarrow G = -FPR^{-1} \Rightarrow G^{-1}F = -RP^{-1}, \tag{528}$$

from the first equation, and leads to

$$FQ + GS = I \Rightarrow F\left[Q - PR^{-1}S\right] = I, \tag{529}$$

$$\Rightarrow F = \left[Q - PR^{-1}S\right]^{-1}, \quad G = -\left[Q - PR^{-1}S\right]^{-1}PR^{-1}, \quad G^{-1} = S - RP^{-1}Q, \tag{530}$$

which, after using Eq. (527) to identify $(F, G) = (D_{\widetilde{y}y}, D_{\widetilde{y}\widetilde{y}})$ and $(P, Q, R, S) = (C_{yy}, C_{y\widetilde{y}}, C_{y\widetilde{y}}^T, C_{\widetilde{y}\widetilde{y}})$, and then when replaced in Eq. (523) for the conditional covariance $C_{\widetilde{y}|y}$ and Eq. (525) for the conditional mean $\mu_{\widetilde{y}|y}$, yields Eq. (379) and Eq. (378), respectively.

Remark 3.1. Another way to obtain indirectly Eq. (379) and Eq. (378), without derivation, is to use the identity of the inverse of a partitioned matrix in Eq. (531), as done in [130], p. 87:

$$\begin{bmatrix} P & Q \\ R & S \end{bmatrix}^{-1} = \begin{bmatrix} M & -MQS^{-1} \\ -S^{-1}RM & S^{-1} + S^{-1}RMQS^{-1} \end{bmatrix}, \quad M := \left[P - QS^{-1}R\right]^{-1}. \tag{531}$$

This method is less satisfactory since without derivation, there is no feel of where the matrix elements in Eq. (531) came from. In fact, the derivation of the 1st row of Eq. (531) follows exactly the same line as for Eqs.(528)-(530). The 2nd row in Eq. (531) looks complex, but before getting into its derivation, we note that exactly the same line of derivation for the 1st row can be straightforwardly followed to arrive at different, and simpler, expressions of the 2nd-row matrix elements $(2, 1)$ and $(2, 2)$, which are similar to those in the 1st row, and which were already derived in Eq. (530).

It can be easily verified that

$$\begin{bmatrix} M & -MQS^{-1} \\ -S^{-1}RM & S^{-1} + S^{-1}RMQS^{-1} \end{bmatrix} \begin{bmatrix} P & Q \\ R & S \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix}. \tag{532}$$

To derive the 2nd row of Eq. (531), premultiply the 1st row (which had been derived as mentioned above) of Eq. (532) by $(-S^{-1}R)$ to have

$$(-S^{-1}R)\begin{bmatrix} M & -MQS^{-1} \end{bmatrix} \begin{bmatrix} P & Q \\ R & S \end{bmatrix} = (-S^{-1}R)\begin{bmatrix} I & 0 \end{bmatrix} = \begin{bmatrix} (-S^{-1}R) & 0 \end{bmatrix}. \tag{533}$$

To make the right-hand side become $\begin{bmatrix} \mathbf{0} & \mathbf{I} \end{bmatrix}$, add to both sides of Eq. (533) the matrix $\begin{bmatrix} (\mathbf{S}^{-1}\mathbf{R}) & \mathbf{I} \end{bmatrix}$ to obtain Eq. (532)'s 2nd row, whose complex expressions did not contribute to the derivation of the conditional Gaussian posterior mean Eq. (378) and covariance Eq. (379).

Yet another way to derive Eqs. (378)-(379) is to use the more complex proof in [248], p. 429, which was referred to in [234], p. 200 (see also Footnote 242). ∎

In summary, the above derivation is simpler and more direct than in [130], p. 87, and in [248], p. 429.
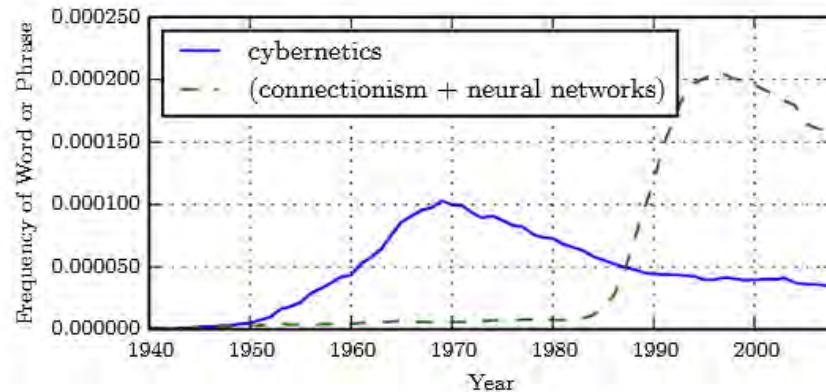
Figure 153: *The first two waves of AI*, according to [78], p.13, showing the "cybernetics" wave (blue line) started in the 1940s peaked before 1970, then gradually declined toward 2006 and beyond. The results were based on a search for frequency of words in Google Books. It was mentioned, incorrectly, that the work of Rosenblatt (1957-1962) [1]-[2] was limited to one neuron; see Figure 42 and Figure 133. (Figure reproduced with permission of the authors.)

4 The ups and downs of AI, cybernetics

The authors of [78], p.13, divided the wax-and-wane fate of AI into three waves, with the first wave called the "cybernetics" that started in the 1940s, peaked before 1970, then began a gradual descent toward 1986, when the second wave picked up with the publication of [22] on an application of backpropagation to psychology; see Section 13.4.1 on a history of backpropagation. Since Goodfellow (the first author of [78]) worked at Google at the time, and would have access to the scanned books in the Google Books collection to do the search. For a concise historical account of "cybernetics", see [425].

We had to rely on Web of Science to do the "topic" search for the keyword "cyberneti*", i.e., using the query `ts=(cyberneti*)`, with "*" being the search wildcard, which can stand for any character that follows. Figure 154 is the result,[341] spanning an astoundingly vast and diverse number of more than 100 categories,[342] listed in descending order of number of papers in parentheses: Computer Science Cybernetics (2,665 papers), Computer Science Artificial Intelligence (601), Engineering Electrical Electronic (459), ..., Philosophy (229), ..., Social Sciences Interdisciplinary (225), ..., Business (132), ..., Psychology Multidisciplinary (128), ..., Psychiatry (90), ..., Art (66), ..., Business Finance (43), ..., Music (31), ..., Religion (27), ..., Cell biology (21), ..., Law (21), ...

---

[341] The total number of papers on the topic "cyberneti*" was 7,962 on 2020.04.15—as shown in Figure 154 obtained upon clicking on the "Citation Report" button in the Web of Science—and 8,991 on 2022.08.08. Since the distribution in Figure 154, the points made in the figure caption and in this section remain the same, there was no need to update the figure to its 2022.08.08 version.

[342] The number of categories has increased to 244 in the Web of Sciecne search on 2022.08.08, mentioned in Footnote 341, with the number of papers in Computer Science Cybernetics at 2,952, representing 32% of the 8,991 papers in this topic.
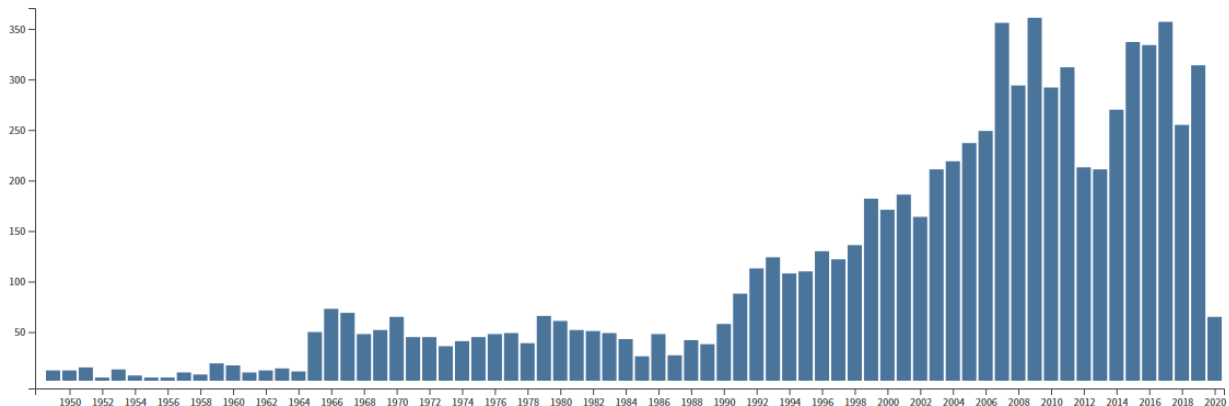
Total Publications

7,962 Analyze

Figure 154: *Cybernetics papers*, (Appendix 4). Web of Science search on 2020.04.15, having more than 100 Web of Science categories. The first paper was [426]. There was no clear wave that crested before 1970, but actually the number of papers in Cybernetics continue to increase over the years.

The first paper in 1949 [426] was categorized as Mathematics. More recent papers include Biological Science, e.g., [427], Building Construction, e.g., [428], Accounting, e.g., [429].

It is interesting to note that McCulloch who co-authored the well-known paper [430] was part of the original cybernetics movement that started in the 1940s, as noted in [431]:

> "Warren McCulloch, the "chronic chairman" and founder of the cybernetics conferences.[343] An eccentric physiologist, McCulloch had coauthored a foundational article of cybernetics on the brain's neural network."

But McCulloch & Pitt's 1943 paper [430]—often cited in artificial-neural-network papers (e.g., [23], [12]) and books (e.g., [78]), and dated six years before [426]—was placed in the Web of Science category "Biology; Mathematical & Computational Biology," and thus did not show up in the search with keyword "cyberneti*" shown in Figure 154. A reason is [430] did not contain the word "cybernetics," which was not invented until 1948 with the famous book by Wiener, and which was part of the title of [426]. Cybernetics was a "new science" with a "mysterious name and universal aspirations" [431], p.5.

> "What exactly is (or was) cybernetics? This has been a perennial ongoing topic of debate within the American Society for Cybernetics throughout its 50-year history. ... the word has a much older history reaching back to Plato, Ampère ("Cybernétique = the art of growing"), and others. "Cybernetics" comes from the Greek word for governance, *kybernetike*, and the related word, *kybernetes*, steersman or captain" [432].

Steering a ship is controlling its direction. [433] defined cybernetics as

---

[343] These cybernetics conferences were called the Macy conferences, held during a short period from 1946 to 1953, and involved researchers from diverse fields: not just mathematics, physics, engineering, but also anthropology and physiology, [431], pp.2-3.

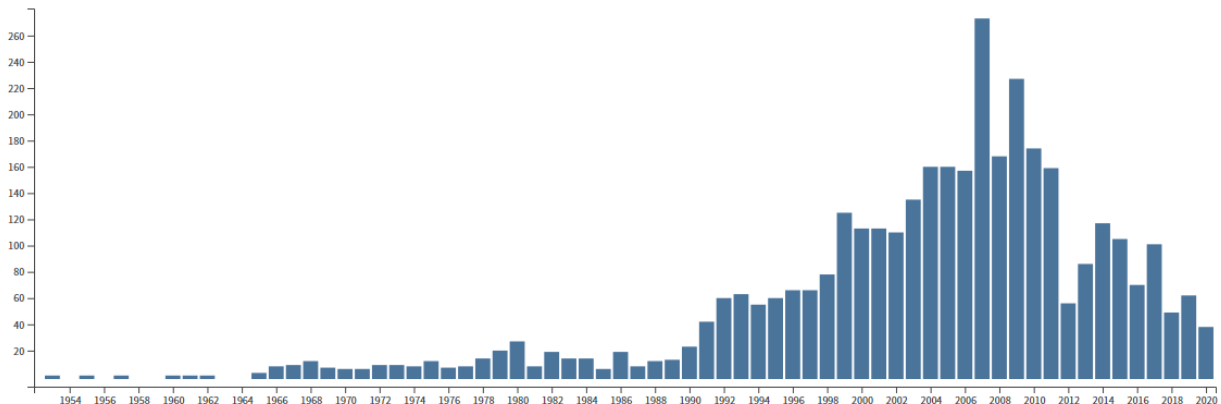**Total Publications**

3,555 Analyze

Figure 155: *Cybernetics papers*, (Appendix 4). Web of Science search on 2020.04.17, ALL Computer-Science categories (3,555 papers): Cybernetics (2,666), Artificial Intelligence (602), Information Systems (432), Theory Methods (300), Interdisciplinary Applications (293), Software Engineering (163). The wave crest was in 2007, with a tiny bump in 1980.

"... (feedback) control and communication theory pertinent to the description, analysis, or construction of systems that involve (1) mechanisms (receptors) for the reception of messages or stimuli, (2) means (circuits) for communication of these to (3) a central control unit that responds by feeding back through the system (4) instructions that (will or tend to) produce specific actions on the part of (5) particular elements (effectors) of the system. ... The central concept in cybernetics is a feedback mechanism that, in response to information (stimuli, messages) received through the system, feeds back to the system instructions that modify or otherwise alter the performance of the system."

Even though [432] did not use the word "control", the definition is similar:

"The core concepts involved natural and artificial systems organized to attain internal stability (homeostasis), to adjust internal structure and behavior in light of experience (adaptive, self-organizing systems), and to pursue autonomous goal-directed (purposeful, purposive) behavior." [432]

and is succinctly summarized by [434]:

"If "cybernetics" means "control and communication," what does it not mean? It would be difficult to think of any process in which nothing is either controlled or communicated."

which is the reason why cybernetics is found in a large number of different fields. [431], p.4, offered a similar, more detailed explanation of cybernetics as encompassing all fields of knowledge:

"Wiener and Shannon defined the amount of information transmitted in communications systems with a formula mathematically equivalent to entropy (a measure of the degradation of energy). Defining information in terms of one of the pillars of physics convinced many researchers that information theory could bridge the physical, biological, and social sciences.

The allure of cybernetics rested on its promise to model mathematically the purposeful behavior of all organisms, as well as inanimate systems. Because cybernetics included information theory in its purview, its proponents thought it was more universal than Shannon's theory, that it applied to all fields of knowledge."
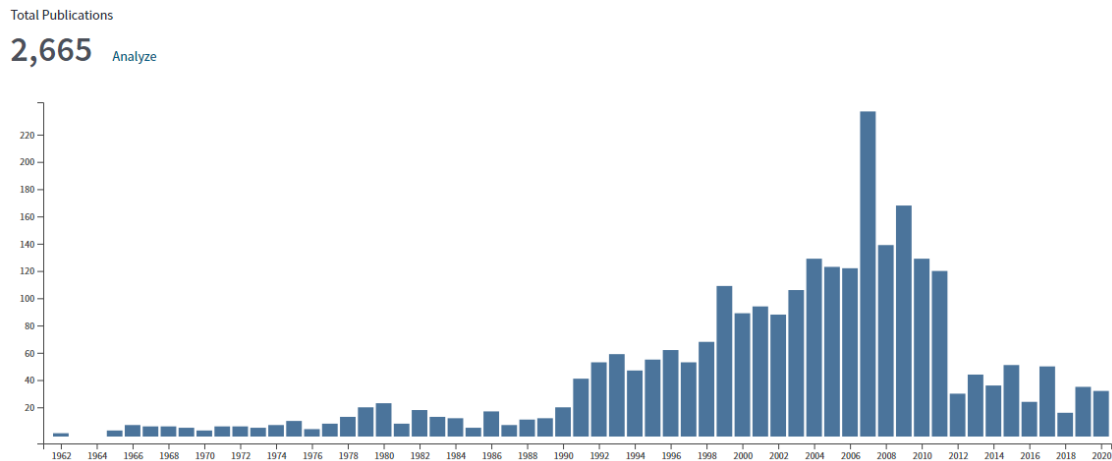

Figure 156: *Cybernetics papers*, (Appendix 4). Web of Science search on 2020.04.15 (two days before Figure 155), category Computer Science Cybernetics (2,665 papers). The wave crest was in 2007, with a tiny bump in 1980.
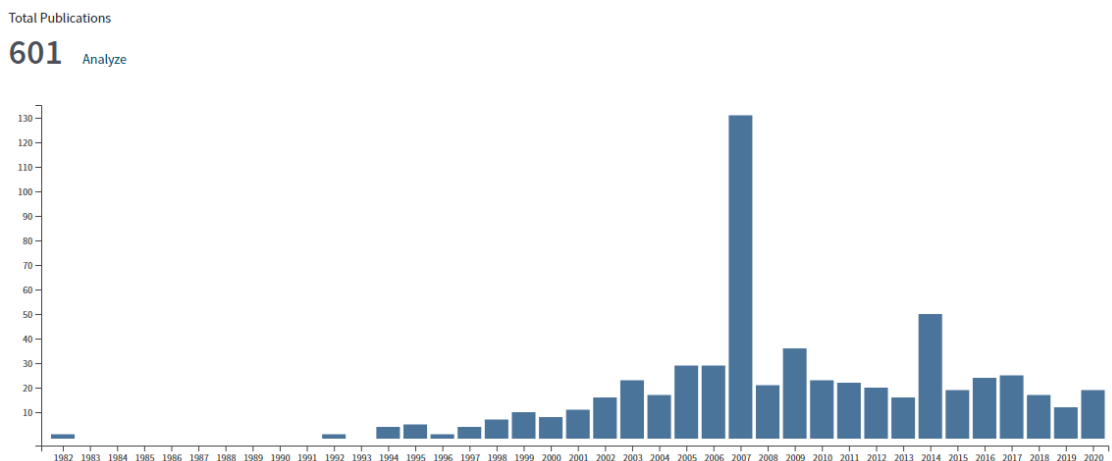

Figure 157: *Cybernetics papers*, (Appendix 4). Web of Science search on 2020.04.15 (two days before Figure 155), category Computer Science Artificial Intelligence (601 papers). Similar to Figure 156, the wave crest was in 2007, but with no tiny bump in 1980, since the first paper was in 1982.

In 1969, the then president of the International Association of Cybernetics asked "But after all what is cybernetics? Or rather what is it not, for paradoxically the more people talk about cybernetics the less they seem to agree on a definition," then identified several meanings: A mathematical control theory, automation, computerization, communication theory, study of human-machine analogies, philosophy explaining the mysteries of life! [431], p.5.
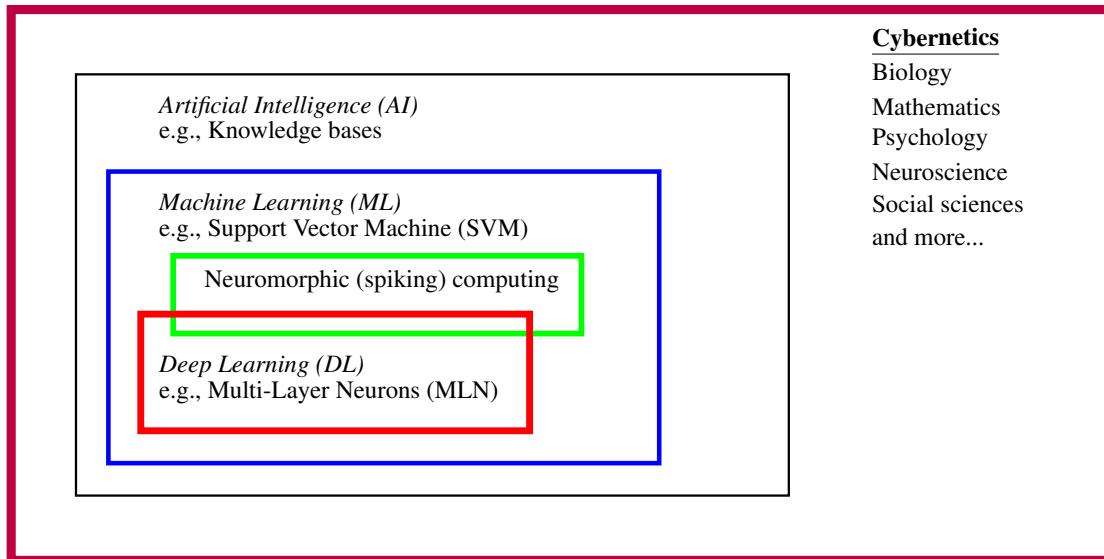
Figure 158: *Artificial Intelligence* (AI), *Machine Learning* (ML), and *Deep Learning* (DL). *Cybernetics* is broad and encompasses many fields, including AI. See also Figure 6.

So was there a first wave in AI called "cybernetics" ? Back in Oct 2018, we conveyed our search result at that time—which was similar to Figure 154, but clearly did not support the existence of the cybernetics wave shown in Figure 153—to Y. Bengio of [78], who then replied:

> "Ian [Goodfellow] did those figures, but my take on your observations is that the later surge in 'new cybernetics' does not have much more to do with artificial neural networks. I'm not sure why the Google Books search did not catch that usage, though."

We then selected only the categories that had the words "Computer Science" in their names; there were only six such categories among more than 100 categories, as shown in Figure 155. A similar figure obtained in Oct 2018 was also shared with Bengio, who had no further comment. The wave crest in Figure 155 occurred in 2007, with a tiny bump in 1980, but not before 1970 as in Figure 153.

Figure 156 is the histogram for the largest single category Computer Science Cybernetics with 2,665 papers. In this figure, similar to Figure 155, the wave crest here also occurred in 2007, with a tiny bump in 1980.

Figure 157 is the histogram for the category Computer Science Artificial Intelligence with 601 papers. Again here, similar to Figure 155 and Figure 156, the wave crest here also occurred in 2007, with no bump in 1980. The first document, a 5-year plan report of Latvia, appeared in 1982. There is a large "impulse" of number of papers in 2007, and a smaller "impulse" in 2014, but no smooth bump. There were no papers for 9 years between 1982 and 1992, in which a single paper appeared in the series "Lecture Notes in Artificial Intelligence" on cooperative agents.

Cybernetics, including the original cybernetics moment, as described in [431], encompassed many fields and involved many researchers not working on neural nets, such as Wiener, John von Neumann, Margaret Mead (anthropologist), etc., whereas the physiologist McCulloch co-authored the first "foundational article of cybernetics on the brain's neural network". So it is not easy to attribute even the original cybernetic moment to research on neural nets alone. Moreover, many topics of interest to researchers at the time involve natural systems (including [430]), and thus natural intelligence, instead of artificial intelligence.