Machine Learning & Deep Learning – Học Máy & Học Sâu

Nguyễn Quản Bá Hồng*

Ngày 21 tháng 1 năm 2025

Tóm tắt nội dung

This text is a part of the series *Some Topics in Advanced STEM & Beyond*:
URL: https://nqbh.github.io/advanced_STEM/.
Latest version:

- *Machine Learning & Deep Learning – Học Máy & Học Sâu.*
  PDF: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/machine_learning/NQBH_machine_learning.pdf.
  TEX: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/machine_learning/NQBH_machine_learning.tex.

Mục lục

*A Scientist & Creative Artist Wannabe. E-mail: nguyenquanbahong@gmail.com. Bến Tre City, Việt Nam.

Question 1. *Compare recurrent neural network with recursive neural network.*

Community – Cộng đồng.

1. Francis Bach.
   - Website: https://francisbach.com/.
   - ENS Homepage: https://www.di.ens.fr/~fbach/.
2. Phạm Hy Hiếu.
3. Phạm Tuấn Huy.
4. Võ Văn Huy.
5. Yan LeCun.
6. Nguyễn Phan Minh.

1 Journal

1. **Journal of Machine Learning Research [JMLR]**
   - Website: https://www.jmlr.org/.
     Journal of Machine Learning Research [JMLR], established in 2000, provides an international forum for electronic & paper publication of high-quality scholarly articles in all areas of ML. All published papers are freely available online.
     JMLR has a commitment to rigorous yet rapid reviewing. Final versions are published electronically (ISSN 1533-7928) immediately upon receipt.
2. **Machine Learning.** Netherlands. Subject area & category: Computer Science [AI, Software]. SJR 2023: 1.72.
   Overview. Machine Learning is an international forum focusing on computational approaches to learning.
   - Reports substantive results on a wide range of learning methods applied to various learning problems.
   - Provides robust support through empirical studies, theoretical analysis, or comparison to psychological phenomena.
   - Demonstrates how to apply learning methods to solve significant application problems.

- Improves how ML research is conducted.
- Prioritizes verifiable & replicable supporting evidence in all published papers.

Aims & scope. *Machine Learning* is an international forum for research on computational approaches to learning. Journal publishes articles reporting substantive results on a wide range of learning methods applied to a variety of learning problems, including but not limited to:

- **Learning Problems.** Classification, regression, recognition, & prediction; Problem solving & planning; Reasoning & inference; Data mining; Web mining; Scientific discovery; Information retrieval; Natural language processing; Design & diagnosis; Vision & speech perception; Robotics & control; Combinatorial optimization; Game playing; Industrial, financial, & scientific applications of all kinds.
- **Learning Methods.** Supervised & unsupervised learning methods (including learning decision & regression trees, rules, connectionist networks, probabilistic networks & other statistical models, inductive logic programming, case-based methods, ensemble methods, clustering, etc.); Reinforcement learning; Evolution-based methods; Explanation-based learning; Analogical learning methods; Automated knowledge acquisition; Learning from instruction; Visualization of patterns in data; Learning in integrated architectures; Multistrategy learning; Multi-agent learning.

Papers describe research on problems & methods, applications research, & issues of research methodology. Papers making claims about learning problems (e.g., inherent complexity) or methods (e.g., relative performance of alternative algorithms) provide solid support via empirical studies, theoretical analysis, or comparison to psychological phenomena. Applications papers show how to apply learning methods to solve important applications problems. Research methodology papers improve how ML research is conducted. All papers must state their contributions clearly & describe how contributions are supported. All papers must describe supporting evidence in ways that can be verified or replicated by other researchers. All papers must describe learning component clearly, & must discuss assumptions regarding knowledge representation & performance task. All papers must place their contribution clearly in context of existing work in ML. Variations from these prototypes, e.g. comprehensive surveys of active research areas, critical reviews of existing work, & books reviews, will be considered provided they make a clear contribution to the field.

2 Machine Learning

Resources – Tài nguyên.

2.1 Wenbo Cao, Weiwei Zhang. *ML of PDEs from Noise Data*. Theoretical & Applied Mechanics Letters

[12 citations] Keywords. PDE, ML, Sparse regression, Noise data.

Abstract. ML of PDEs from data is a potential breakthrough for addressing lack of physical equations in complex dynamic systems. Recently, sparse regression has emerged as an attractive approach. However, noise presents biggest challenge in sparse regression for identifying equations, as it relies on local derivative evaluations of noisy data. This study proposes a simple & general approach that significantly improves noises robustness by projecting evaluated time derivative & partial differential term into a subspace with less noise. This method enables accurate reconstruction of PDEs involving high-order derivatives, even from data with considerable noise. Additionally, discuss & compare effects of proposed method based on Fourier subspace & POD (proper orthogonal decomposition) subspace. Generally, the latter yields better results since it preserves maximum amount of information.

- 1. Introduction. PDEs are increasingly important in modern science. PDEs are used to describe mathematical laws behind physical systems & play a vital role in analysis, prediction, & control of many systems. In past, PDEs were derived via basic conservation laws, which resulted in many canonical models in physics, engineering, & other fields. However, in modern applications, mechanisms of many complex systems remain unclear, making it difficult to derive PDEs. This is particularly true in fields e.g. multiphase flow, neuroscience, finance, bioscience, & others. In past decade, with rapid development of sensors, computing power & data storage, cost of data collection & computing has been greatly reduced, resulting in a large amount of experimental data. Meanwhile rapid development of ML [1] has also provided a reliable tool to discover potential laws of system from large datasets. Nowadays, ML of differential equations has become a promising new technology to discover physical laws in complex systems.

  Among all methods investigated for model discovery [2–9], sparse regression [4–6] has gained most attention in recent studies because of its ability to discover interpretable & generalizable models with a balance of accuracy & efficiency. This approach provides an important model discovery framework, relying on sparse linear regression to select sparse terms to match data from a predefined function library which containing many nonlinear & partial derivative terms. Sparse regression has been applied to many challenging problems to identify potential ODEs or PDEs, including fluid dynamics [10–12], turbulence closures [13–15], vortex-induced vibration [16], subsurface flow equations [17,18], & others. More details on important applications & extensions of sparse regression can be found in [19].

  Nowadays, biggest challenge with sparse regression: identify equations from noisy data. Sensitivity of sparse regression to noise seriously damages reliability of results because sparse regression relies on accurate evaluation of derivatives, which is

especially challenging for PDEs where noise can be strongly amplified when computing higher-order spatial derivatives. Various approaches have been proposed to improve noise robustness of sparse regression, which can be roughly divided into 3 types. 1st: obtain smooth data by using smooth function to approximate noisy data globally, including use of neural network [8,20–23] or filter program. This approach can filter out part of noise, but usually introduces new approximation errors, which may result in data no longer strictly satisfying original governing equation. A typical demonstration: performing such preprocessing on clean data usually leads to loss of accuracy. 2nd: use accurate derivation approximation methods [5,21], including spline smoothing, polynomial fitting, Gaussian kernel smoothing & Tikhonov differentiation. These methods approximate noisy data locally, & they also introduce approximation errors, but usually small than 1st type of methods. 3rd: avoid numerical differentiation by explicitly or implicitly introducing numerical integration [20,24–26] for ODE discovery or reduce order of derivative by using weak form [27–31] for PDE discovery. These methods can greatly improve robustness of sparse regression against noise. Because they are derived by mathematical methods, they will not lose accuracy or even improve accuracy when used for clean data. However, those methods are complex to use & often contain many hyperparameters.

Present paper significantly improves noise robustness of sparse regression by simply projecting time derivative & library functions into a low noise subspace. Below introduce our method & discuss its similarities & differences with other methods, & then give some representation examples.

- Methods. Here, 1st describe general framework of sparse regression, then propose our improved method

  - 2.1. Sparse regression. Consider general form of nonlinear PDE $\boxed{u_t = N(u, u_x, u_{xx}, \ldots, x, \mu)}$, where, subscripts denote partial differentiation in either time or space, & $N(\cdot)$: an uknown nonlinear function of state $u(t, x)$. Method builds an over-completed library that contains all terms that may appear in PDE, e.g.

$$\theta(u) = [u, u^2, u_x, uu_x, u^2u_x, u_{xx}, uu_{xx}, u^2u_{xx}, u_{xxx}, uu_{xxx}, u^2u_{xxx}]. \tag{1}$$

  Then, time derivative u_t can be expressed as a linear combination of library terms (2)

$$u_t = \sum_{i=1}^{K} \xi_i f_i(u) = \theta(u)\boldsymbol{\xi}. \tag{2}$$

  For given experimental data of a physical field, values of u_t & $f_k(u)$ at many spatiotemporal points are evaluated, which lead to a system of linear equations (3)

$$(\mathbf{U}_t)_c = (\boldsymbol{\Theta}(u))_c\boldsymbol{\xi}, \tag{3}$$

  where, $(\cdot)_c$: operator that arranges data into a column vector, $(\boldsymbol{\Theta})_c$ denotes $[(\mathbf{F}_1)_c, (\mathbf{F}_2)_c, \ldots, (\mathbf{F}_K)_c]$, $\mathbf{U}_t, \mathbf{F}_k$: matrices whose i, j element holds function values of $u_t, f_k(u)$, resp., at ith time at jth space position, & $\boldsymbol{\xi}$: a vector of unknown coefficients. Each element in $\boldsymbol{\xi}$ is a coefficient corresponding to a term in PDE. Many dynamical systems have relatively few active terms in governing equations. Thus, may employ sparse regression to identify sparse vector $\boldsymbol{\xi}$, which signifies fewest active terms from library that result in a good model fit. This can be represented as (4)

$$\boldsymbol{\xi} = \arg\min_{\boldsymbol{\xi}} \|(\mathbf{U}_t)_c - (\boldsymbol{\Theta}(u))_c\boldsymbol{\xi}\|_2^2 + R(\boldsymbol{\xi}). \tag{4}$$

  Regularizer $R(\boldsymbol{\xi})$ is chosen to promote sparsity of $\boldsymbol{\xi}$. E.g., sequentially thresholded least-squares (STLS) [4] uses $R(\boldsymbol{\xi}) = \lambda\|\boldsymbol{\xi}\|_0$, whereas sequentially thresholded ridge regression (STRidge) [5] uses $R(\boldsymbol{\xi}) = \lambda_1\|\boldsymbol{\xi}\|_0 + \lambda_2\|\boldsymbol{\xi}\|_2$. Solution of sparse vector $\boldsymbol{\xi}$ reveals hidden PDE of given system.

  - 2.2. Subspace projection denoising. Sparse regression is simple & effective, but difficult to identify PDE from noisy data because local derivative evaluation strongly amplifies noise. In this paper, propose subspace projection denoising (SPD) method to solve thorny problem.

    As a basic assumption of sparse regression, any linear or nonlinear PDE can be expressed as a linear combination of library terms (2). Therefore, performing a same linear transformation on u_t & f_k, have

$$L(u_t) = L(\theta)\boldsymbol{\xi}, \tag{5}$$

    where, L: a linear operator, $L(\boldsymbol{\theta})$ denotes $[L(f_1), L(f_2), \ldots, L(f_K)]$. To filter noise as much as possible, project both time derivative $\mathbf{U}_t$ & library function $\mathbf{F}_k$ into a subspace with less noise

$$\widetilde{\mathbf{U}}_t = \boldsymbol{\Theta}_t^\top \mathbf{U}_t \boldsymbol{\Theta}_x, \ \widetilde{\mathbf{F}}_k = \boldsymbol{\Theta}_t^\top \mathbf{F}_k \boldsymbol{\Theta}_x, \tag{6}$$

    where, $\boldsymbol{\Theta}_t$: a matrix whose columns represent basis functions related to time, & $\boldsymbol{\Theta}_x$: a matrix whose columns represent basis functions related to space. Since projection is a linear operator, vector $\boldsymbol{\xi}$ can be obtained by solving

$$(\widetilde{\mathbf{U}}_t)_c = (\widetilde{\boldsymbol{\Theta}})_c\boldsymbol{\xi}, \tag{7}$$

    where $(\widetilde{\boldsymbol{\Theta}})_c \coloneqq [(\widetilde{\mathbf{F}}_1)_c, \ldots, (\widetilde{\mathbf{F}}_K)_c]$. Fourier basis can be used as basis function, in which case projection (6) can be easily implemented by fast Fourier Transformation (FFT) on each spatiotemporal axis, then low frequency components are chosen to solve vector $\boldsymbol{\xi}$. On other hand, proper orthogonal decomposition (POD) is a linear method for establishing an optimal

basis, or modal decomposition, of an ensemble of continuous or discrete functions. Therefore, POD basis can also be selected to better characterize current data & PDE, in which case $\mathbf{\Theta}_t$ & $\mathbf{\Theta}_x$ are given by singular value decomposition of $\mathbf{U}$:

$$\mathbf{U} \cong \mathbf{W}_r \mathbf{\Sigma}_r \mathbf{V}_r^* = \mathbf{\Theta}_t \mathbf{\Sigma}_r \mathbf{\Theta}_x^*, \tag{8}$$

where $\mathbf{\Theta}_x^*$: transpose of $\mathbf{\Theta}_x$, r: a user-specified hyperparameter, which indicates that only 1st r bases with larger energy are retained, & $\mathbf{\Sigma}_r = \text{diag}\lambda_i$ is a diagonal matrix with singular values of $\mathbf{U}$, which represents energy of corresponding POD basis function. Obviously, proposed subspace projection denoising (SPD) method is independent of evaluation method of derivatives. It only projects evaluated time derivatives & library functions into a new space. Therefore, it can still be effectively combined with other robust derivation approximation methods. SPD method is quite simple & requires only minimal additional processing (6), but it significantly enhances robustness to noisy data, which will be verified later. Furthermore, method is suitable for high dimensional spatiotemporal data & PDE, requiring only additional rounds of FFT or projection.

Before presenting results, further illustrate proposed method by comparing it with low-pass filtering. When using Fourier basis, proposed method is similar to low-pass filtering of state $u(t, x)$, but their purposes & effects are different. Low-pass filtering of state $u(t, x)$ may change its distribution so that filtered data no longer satisfies original governing equation. Therefore, it can only filter out little noise to preserve real signal as much as possible. In contrast, SPD method performs low-pass filtering on evaluated time derivative & library functions. It can filter out part of real signal to filter noise as much as possible, & can use lowest frequency component with low-noise data to identify PDE, because (5) is always automatically satisfied for any linear operator L.

- 3. Parameter identification of PDEs. For simplicity, this subsect assumes: equation structure is known, & only parameters in equation are calculated from noisy data to evaluate effectiveness of proposed method. Obviously, accuracy of calculated parameter errors directly affects likelihood of identifying equation from library.

  Adopt def of noise given by Rudy [5]

$$u_n = u + \sigma \times \text{std}(u) \times \text{randn}, \tag{9}$$

  where u: numerical solution, σ: noise level, std(u): standard deviation of u, randn: a random variable with standard normal distribution, & u_n represents data with noise level σ. Accuracy of model reconstruction quantified by relative errors

$$\text{error}_i = \left| \frac{\widehat{\xi}_i - \xi_i}{\xi_i} \right|, \tag{10}$$

  where ξ_i: correct coefficients & $\widehat{\xi}_i$: coefficient estimated from noisy data. In all cases, hyperparameter r is set to 5. Therefore, total number of equations in system defined by (7) is 5^2 for 2D PDE & 5^3 for 3D PDE.

  - 3.1. Burgers' equation. Burgers' equation arises in many technological contexts, including fluid mechanics, nonlinear acoustics, gas dynamics, & traffic flow. Take form $u_t = -uu_x + \nu u_{xx}$, where $\nu > 0$: diffusion coefficient. Burgers' equation is a nonlinear 2nd-order PDE. Test SPD method on Burgers' equation $u_t = \xi_1 uu_x + \xi_2 u_{xx}$ with unknown coefficients $\xi_1 = -1, \xi_2 = 0.05$. Data set used to identify coefficients is shown in Fig. 1: Solution of Burgers' equation with 0% noise & 20% noise.

    1stly, cf relative error distributions of $\mathbf{U}_{xx}$ evaluated from data with 20% noise in physical space, Fourier space & POD space. Fig. 2: Relative error $\log_{10}\left|\frac{\hat{U}_{xx} - U_{xx}}{U_{xx}}\right|$ evaluated from data with 20% noise in (a) physical space, (b) Fourier space, & (c) POD space, k_x, k_t represent corresponding coordinates in Fourier space, which are frequency indexes. n_x, n_t represent corresponding coordinates in POD space, & energy (i.e., singular value) of corresponding basis function decreases as they increase. Number of basis functions for both time & space are limited to 40 to make picture clearer. Fig. 2a shows: error is almost evenly distributed throughout original physical space. In Fourier space, error decreases as frequency decreases (Fig. 2b), while in POD space, error decreases as singular value increases (Fig. 2c). Therefore, projecting each term of library into Fourier subspace with lower frequencies or POD subspace with higher energy can significantly reduce error, as shown in lower left corner of Fig. 2b & c.

    After that, PDE-FIND is used as baseline, in which derivative is evaluated by finite difference, & SPD method is used to improve noise robustness. Results for different noise levels are shown in Fig. 3: Coefficient identification errors of (a) uu_x. (b) u_{xx} in Burgers' equation against different noise levels, where local derivatives are evaluated by finite difference., where "initial" refers to identification results of projecting time derivative & library functions into Fourier subspace & POD subspace resp. (7).

    As shown in Fig. 3, since noise is strongly amplified when computing derivative by finite difference, identified coefficients error of initial method is $> 10\%$ for only 1% noise, & coefficient error of u_{xx} is even close to 100%, which is actually maximum error because identified coefficient tends to 0 as noise level increases. Compared with initial method, SPD method with either Fourier bases or POD bases has a great improvement, which can decrease error by 2–4 orders of magnitude. Even for clean data, SPD method has less error than baseline. In addition, since time derivative & library functions have larger components in truncated POD subspace than Fourier subspace, former has smaller errors.

    In additions, present a result of applying low-pass filtering on $u(t, x)$ with 0% noise, preserving same frequencies as SPD-Fourier method. As shown in Fig. 4: (a) Low-pass filtered data. (b) Error between filtered data & original data., low-pass filtering introduces considerable error because it retains only very few frequencies. As a result, coefficient identification error for clean data is as high as 20%, $\hat{\xi}_1 = -0.7973, \hat{\xi}_2 = 0.0602$). This supports the point mentioned in Sect. 2.2: low-pass

filtering of $u(t,x)$ can cause data to no longer satisfy equation, whereas low-pass filtering of evaluated time derivative & function libraries does not.

Furthermore, polynomial fitting is used to evaluate local derivatives due to its robustness to noisy data. Fig. 5: Coefficient identification errors of (a) uu_x. (b) u_{xx} in Burgers' equation against different noise levels, where local derivatives are evaluated by polynomial fitting. shows: polynomial fitting reduces coefficient errors for noisy data compared with finite difference. But at same time, errors for clean data are also increased due to approximation error of polynomial fitting. Therefore, although polynomial fitting can improve robustness to noise, it also inevitably introduces new approximation errors. In contrast, SPD method filters noise without introducing any new errors because (5) is always true for any linear operator L. Although polynomial fitting reduces derivative evaluation error for noisy data, coefficient error of initial method is still $> 40\%$ for 20% noise, while SPD method can reduce errors by 2 orders of magnitude.

- 3.2. Kuramoto–Sivashinsky equation. Kuramoto–Sivashinsky describes chaotic dynamics of laminar frame fronts, reaction-diffusion systems, & coating flows. It takes form $u_t = -uu_x - u_{xx} - u_{xxxx}$. This is a notable example of a nonlinear PDE that involves high-order partial derivatives, which has made it difficult to identify from noisy data accurately. Test SPD method on equation $u_t = \xi_1 uu_x + \xi_2 u_{xx} + \xi_3 u_{xxxx}$ with unknown coefficients $\xi_1 = \xi_2 = \xi_3 = -1$. Data set used to identify coefficients is shown in Fig. 6: Solution of Kuramoto–Sivashinsky equation with (a) 0% noise. (b) 20% noise.

  Use PDE-FIND as baseline & polynomial fitting is used to evaluate derivatives. Fig. 7: Maximum coefficient error of Kuramoto–Sivashinsky equation against different noise levels. shows maximum coefficient error for different noise levels, which corresponds to 1 of terms in PDE. It shows: even for high-order derivative with 20% noise, SPD method can still reduce coefficient error to about 5%. This result is roughly equivalent to that of weak form [29,30], which is most robust method for noisy data reported. Moreover, SPD method is much simpler than weak form. In addition, identified coefficients have small error for clean data, which is caused by approximation error of polynomial fitting.

- 4. Sparse regression. Finally, as an example of how proposed method could be used in context of sparse regression, consider a numerical example in [29], which applies weak form to discover λ-ω reaction-diffusion system

$$u_t = D\nabla^2 u + \lambda u - \omega v, \tag{11}$$

$$v_t = D\nabla^2 v + \omega u + \lambda u, \tag{12}$$

where $\omega = -(u^2+v^2), \lambda = 1 - u^2 - v^2$, & $D = 0.1$ is constant. Weak form is 1 of current state-of-art methods for identifying equations from noisy data. It attempts to reduce order of derivatives by multiplying basis functions over terms in library & integrating result by parts over a spatiotemporal domain. In order to compare with weak form, use same data set (as shown in Fig. 8: A typical snapshot of solution of λ-ω reaction-diffusion system with (a) 0% noise. (b) 100% noise.) & library functions (as listed in (12)) as [29]. In total, generalized model involves a total of 20 different terms (2 diffusion terms & 18 polynomial terms). Correspondingly, 20 unknown coefficients need to be determined:

$$\boldsymbol{\theta}_{u_t} = [\nabla^2 u, u, u^2, u^3, v, v^2, v^3, uv, u^2 v, uv^2],\ \boldsymbol{\theta}_{v_t} = [\nabla^2 v, u, u^2, u^3, v, v^2, v^3, uv, u^2 v, uv^2]. \tag{13}$$

1stly, evaluate accuracy of identified parameter under different noise levels & compare it with weak form [29]. Initial method & SPD method uses finite difference instead of polynomial fitting to evaluate derivative to avoid high computational cost. Fig. 9: Maximum coefficient error of λ-ω system against different noise levels shows: under all noise levels, errors of weak form & SPD method are dramatically reduced compared with baseline. When noise level is $> 3\%$, weak form & SPD method have about same accuracy; when noise level is smaller, SPD method has higher accuracy. In addition, emphasize: implementation of SPD is much simpler than weak form. In this case, error of SPD using Fourier bases is lower than that of POD bases. This can be explained by data's approximate periodicity along each space-time axis.

After that, use SINDy [4] algorithm to determine parsimonious model (mô hình tiết kiệm). Find: for noise levels of up to 10%, PDE was identified correctly for 200 cases with different random noises by SPD method with Fourier bases or POD bases. With 30% noise, model is identified correctly is about 6% of cases, with remaining cases featuring spurious terms that are not present in λ-ω system. For ref, PDE-FIND failed to correctly identify this PDE for as little as 1% noise & weak form correctly identify this PDE in about 95% of cases for 10% noise. Therefore, SPD method has roughly same effect as weak form, with a dramatic improvement over baseline.

- 5. Discussion. This work has developed a simple & effective method that greatly improves robustness & accuracy for model discovery, reducing data requirements & increasing noise tolerance. Proposed subspace projection denoising method projects evaluated time derivative & library terms into low frequency subspace with low noise or POD subspace with high energy so as to greatly reduce influence of noise. Moreover, method can be used in combination with many other methods, including polynomial fitting, neural network smoothing, to further improve robustness to noisy data. Several typical examples show: compared with baseline, SPD method can reduce error by several orders of magnitude, achieving same effect as weak form, while SPD method is simpler & has only 1 hyperparameter. More importantly, this study points out: original PDE automatically satisfied when any linear operator applied to evaluated time derivative & library functions, which provides a general framework for denoising, thus allowing more denoising methods with different linear operators.

2.2 [DFO23]. Marc Peter Deisenroth, A. Aldo Faisal, Cheng Soon Ong. *Mathematics for Machine Learning*. 2023

[849 Amazon ratings]

Amazon review. Fundamental mathematical tools needed to understand ML include linear algebra, analytic geometry, matrix decompositions, vector calculus, optimization, probability & statistics. These topics are traditionally taught in disparate courses, making it hard for DS or CS students, or professionals, to efficiently learn mathematics. This self-contained textbook bridges gap between mathematical & ML texts, introducing mathematical concepts with a minimum of prerequisites. It uses these concepts to derive 4 central ML methods: linear regression, principal component analysis, Gaussion mixture models & support vector machines. For students & others with a mathematical background, these derivations provide a starting point to ML texts. For those learning mathematics for 1st time, methods help build intuition & practical experience with applying mathematical concepts. Every chap includes worked examples & exercises to test understanding. Programming tutorials are offered on book's web site.

Editorial Reviews.

- "This book provides great coverage of all basic mathematical concepts for ML. I'm looking forward to sharing it with students, colleagues, & anyone interested in building a solid understanding of fundamentals." – Joelle Pineau, McGill University, Montreal

- "The field of ML has grown dramatically in recent years, with an increasingly impressive spectrum of successful applications. This comprehensive text covers key mathematical concepts that underpin modern ML, with a focus on linear algebra, calculus, & probability theory. It will prove valuable both as a tutorial for newcomers to field, & as a reference text for ML researchers & engineers." – Christopher Bishop, Microsoft Research Cambridge

- "This book provides a beautiful exposition of mathematics underpinning modern ML. Highly recommended for anyone wanting a 1-stop-shop to acquire a deep understanding of ML foundations." – Pieter Abbeel, University of California, Berkeley

- "Really successful are numerous explanatory illustrations, which help to explain even difficult concepts in a catch way. Each chap concludes with many instructive exercises. An outstanding feature of this book is additional material presented on website ..." – Volker H. Schulz, SIAM Review

Book Description. Distills key concepts from linear algebra, geometry, matrices, calculus, optimization, probability & statistics that are used in ML.

About the Author.

Foreword. ML is latest in a long line of attempts to distill human knowledge & reasoning into a form that is suitable for constructing machines & engineering automated systems. As ML becomes more ubiquitous & its software packages become easier to use, natural & desirable: low-level technical details are abstracted away & hidden from practitioner. However, this brings with it danger that a practitioner becomes unaware of design decisions &, hence, limits of ML algorithms.

Enthusiastic practitioner who is interested to learn more about magic behind successful ML algorithms currently faces a daunting set of pre-requisite knowledge:

- Programming languages & data analysis tools

- Large-scale computation & associated frameworks

- Mathematics & statistics & how ML builds on it

At universities, introductory courses on ML tend to spend early parts of course covering some of these pre-requisites. For historical reasons, courses in ML tend to be taught in CS department, where students are often trained n 1st 2 areas of knowledge, but not so much in mathematics & statistics.

Current ML textbooks primarily focus on ML algorithms & methodologies & assume: reader is competent in mathematics & statistics. Therefore, these books only spend 1 or 2 chaps on background mathematics, either at beginning of book or as appendices. Have found many people who want to delve into foundations of basic ML methods who struggle with mathematical knowledge required to read a ML textbook. Having taught undergraduate & graduate courses at universities, find: gap between high school mathematics & mathematics level required to read a standard ML textbook is too big for many people.

This book brings mathematical foundations of basic ML concepts to fore & collects information in a single place so that this skills gap is narrowed or even closed.

- Why Another Book on ML? ML builds upon language of mathematics to express concepts that seem intuitively obvious but that are surprisingly difficult to formalize. Once formalized properly, can gain insights into task we want to solve. 1 common complain of students of mathematics around globe: topics covered seem to have little relevance to practical problems. Believe: ML is an obvious & direct motivation for people to learn mathematics.

  "Math is linked in popular mind with phobia & anxiety. You'd think we're discussing spiders." Strogatz, 2014, p. 281

  This book is intended to be a guidebook to vast mathematical literature that forms foundations of modern ML. Motivate need for mathematical concepts by directly pointing out their usefulness in context of fundamental ML problems. In interest of keeping book short, many details & more advanced concepts have been left out. Equipped with basic concepts presented here, & how they fit into larger context of ML, reader can find numerous resources for further study, provided at end of respective chaps. For readers with a mathematical background, this book provides a brief but precisely stated glimpse o ML. In contrast to other books that focus on methods & models of ML (MacKay, 2003; Bishop, 2006; Alpaydin, 2010; Barber, 2012; Murphy, 2012; Shalev-Shwartz & Ben-David, 2014; Rogers & Girolami, 2016) or programmatic aspects of ML (Müller & Guido, 2016;

Raschka & Mirjalili, 2017; Chollet & Allaire, 2018), provide only 4 representative examples of ML algorithms. Instead, focus on mathematical concepts behind models themselves. Hope readers will be able to gain a deeper understanding of basic questions in ML & correct practical questions arising from use of ML with fundamental choices in mathematical model.

Do not aim to write a classical ML book. Instead, intention: provide mathematical background, applied to 4 central ML problems, to make it easier to read other ML textbooks.

- Who Is Target Audience? As applications of ML become widespread in society, believe: everybody should have some understanding of its underlying principles. This book is written in an academic mathematical style, which enables us to be precise about concepts behind ML. Encourage readers unfamiliar with this seemingly terse style to persevere & to keep goals of each topic in mind. Sprinkle comments & remarks throughout text, in hope: it provides useful guidance w.r.t. big picture.

  Book assumes reader to have mathematical knowledge commonly covered in high school mathematics & physics. E.g., reader should have seen derivatives & integrals before, & geometric vectors in 2D or 3D. Starting from there, generalize these concepts. Therefore, target audience of book includes undergraduate university students, evening learners & learners participating in online ML courses.

  In analogy to music, there are 3 types of interaction that people have with ML:

  - **Astute Listener.** (Người nghe tinh ý): Democratization of ML by provision of open-source software, online tutorials & cloud-based tools allows users to not worry about specifics of pipelines. Users can focus on extracting insights from data using off-the-shelf tools. This enables non-tech-savvy domain experts to benefit from ML. This is similar to listening to music; user is able to choose & discern between different types of ML, & benefits from it. More experienced users are like music critics, asking important questions about application of ML in society e.g. ethics, fairness, & privacy of individual. Hope: this book provides a foundation for thinking about certification & risk management of ML systems, & allows them to use their domain expertise to build better ML systems.

    – **Astute Listener.** (Người nghe tinh ý): Dân chủ hóa ML bằng cách cung cấp phần mềm nguồn mở, hướng dẫn trực tuyến & các công cụ dựa trên đám mây cho phép người dùng không phải lo lắng về các chi tiết cụ thể của đường ống. Người dùng có thể tập trung vào việc trích xuất thông tin chi tiết từ dữ liệu bằng các công cụ có sẵn. Điều này cho phép các chuyên gia trong lĩnh vực không am hiểu công nghệ được hưởng lợi từ ML. Điều này tương tự như việc nghe nhạc; người dùng có thể lựa chọn & phân biệt giữa các loại ML khác nhau, & hưởng lợi từ nó. Những người dùng có kinh nghiệm hơn giống như các nhà phê bình âm nhạc, đặt ra những câu hỏi quan trọng về ứng dụng ML trong xã hội, ví dụ như đạo đức, công bằng, & quyền riêng tư của cá nhân. Hy vọng: cuốn sách này cung cấp nền tảng để suy nghĩ về chứng nhận & quản lý rủi ro của các hệ thống ML, & cho phép họ sử dụng chuyên môn trong lĩnh vực của mình để xây dựng các hệ thống ML tốt hơn.

  - **Experienced Artist.** (Nghệ sĩ giàu kinh nghiệm): Skilled practitioners of ML can plug & play different tools & libraries into an analysis pipeline. Stereotypical practitioner would be a data scientist or engineer who understands ML interfaces & their use cases, & is able to perform wonderful feats of prediction from data. This is similar to a virtuoso playing music, where highly skilled practitioners can bring existing instruments to life & bring enjoyment to their audience. Using mathematics presented here as a primer, practitioners would be able to understand benefits & limits of their favorite method, & to extend & generalize existing ML algorithms. Hope this book provides impetus for more rigorous & principled development of ML methods.

    – **Nghệ sĩ giàu kinh nghiệm.** (Nghệ sĩ giàu kinh nghiệm): Những người hành nghề ML có kỹ năng có thể cắm & chạy các công cụ & thư viện khác nhau vào 1 đường ống phân tích. Người hành nghề theo khuôn mẫu sẽ là 1 nhà khoa học dữ liệu hoặc kỹ sư hiểu các giao diện ML & các trường hợp sử dụng của chúng, & có thể thực hiện những kỳ công dự đoán tuyệt vời từ dữ liệu. Điều này tương tự như 1 nghệ sĩ chơi nhạc điêu luyện, nơi những người hành nghề có kỹ năng cao có thể thổi hồn vào các nhạc cụ hiện có & mang lại niềm vui cho khán giả của họ. Sử dụng toán học được trình bày ở đây như 1 tài liệu tham khảo, những người hành nghề sẽ có thể hiểu được lợi ích & giới hạn của phương pháp yêu thích của họ, & mở rộng & khái quát hóa các thuật toán ML hiện có. Hy vọng cuốn sách này sẽ cung cấp động lực cho sự phát triển & có nguyên tắc chặt chẽ hơn của các phương pháp ML.

  - **Fledgling Composer.** (Nhà soạn nhạc trẻ): As ML is applied to new domains, developers of ML need to develop new methods & extend existing algorithms. They are often researchers who need to understand mathematical basis of ML & uncover relationships between different tasks. This is similar to composers of music who, within rules & structure of musical theory, create new & amazing pieces. Hope this book provides a high-level overview of other technical books for people who want to become composers of ML. There is a great need in society for new researchers who are able to propose & explore novel approaches for attacking many challenges of learning from data.

    – **Fledgling Composer.** (Nhà soạn nhạc trẻ): Khi ML được áp dụng vào các lĩnh vực mới, các nhà phát triển ML cần phát triển các phương pháp mới & mở rộng các thuật toán hiện có. Họ thường là các nhà nghiên cứu cần hiểu cơ sở toán học của ML & khám phá mối quan hệ giữa các nhiệm vụ khác nhau. Điều này tương tự như các nhà soạn nhạc, những người, trong các quy tắc & cấu trúc của lý thuyết âm nhạc, tạo ra các tác phẩm mới & tuyệt vời. Hy vọng cuốn sách này cung cấp 1 cái nhìn tổng quan cấp cao về các cuốn sách kỹ thuật khác cho những người muốn trở thành nhà soạn nhạc của ML. Xã hội có nhu cầu lớn đối với các nhà nghiên cứu mới có khả năng đề xuất & khám phá các phương pháp tiếp cận mới để giải quyết nhiều thách thức của việc học từ dữ liệu.

- Acknowledgments. Grateful to many people who looked at early drafts of book & suffered through painful expositions of concepts. Tried to implement their ideas that we did not vehemently disagree with. Have been lucky to benefit from generosity

of online community, who have suggested improvements via GitHub, which greatly improved book. Following people have found bugs, proposed clarifications & suggested relevant literature, either via GitHub or personal communication.

PART I: MATHEMATICAL FOUNDATIONS.

- 1. Introduction & Motivation. ML is about designing algorithms that automatically extract valuable information from data. Emphasis here is on "automatic", i.e., ML is concerned about general-purpose methodologies that can be applied to many datasets, while producing sth that is meaningful. There are 3 concepts that are at core of ML: data, a model, & learning.

  Since ML is inherently data driven, *data* is at core of ML. Goal of ML: design general-purpose methodologies to extract valuable patterns from data, ideally without much domain-specific expertise. E.g., given a large corpus of documents (e.g., books in many libraries), ML methods can be used to automatically find relevant topics that are shared across documents (Hoffman et al., 2010). To achieve this goal, design *models* that are typically related to process that generates data, similar to dataset given. E.g., in a regression setting, model would describe a function that maps inputs to real-valued outputs. To paraphrase Mitchell (1997): A model is said to learn from data if its performance on a given task improves after data is taken into account. Goal: find good models that generalize well to yet unseen data, which we may care about in future. *Learning* can be understood as a way to automatically find patterns & structure in data by optimizing parameters of model.

  While ML has seen many success stories, & software is readily available to design & train rich & flexible ML systems, believe: mathematical foundations of ML are important in order to understand fundamental principles upon which more complicated ML systems are built. Understanding these principles can facilitate creating new ML solutions, understanding & debugging existing approaches, & learning about inherent assumptions & limitations of methodologies we are working with.

  - 1.1. Find Words for Intuitions. A challenge we face regularly in ML: concepts & words are slippery, & a particular component of ML system can be abstracted to different mathematical concepts. E.g., word "algorithm" is used in ≥ 2 different senses in context of ML. In 1st sense, use phrase "ML algorithm" to mean a system that makes predictions based on input data. Refer to these algorithms as *predictors*. In 2nd sense, use exact same phrase "ML algorithm" to mean a system that adapts some internal parameters of predictor so that it performs well on future unseen input data. Here refer to this adaptation as *training* a system.

    – Một thách thức mà chúng ta thường xuyên gặp phải trong ML: các khái niệm & từ ngữ rất khó nắm bắt, & 1 thành phần cụ thể của hệ thống ML có thể được trừu tượng hóa thành các khái niệm toán học khác nhau. Ví dụ, từ "thuật toán" được sử dụng theo ≥ 2 nghĩa khác nhau trong bối cảnh của ML. Theo nghĩa thứ nhất, sử dụng cụm từ "thuật toán ML" để chỉ 1 hệ thống đưa ra dự đoán dựa trên dữ liệu đầu vào. Tham khảo các thuật toán này là *predictors*. Theo nghĩa thứ hai, sử dụng chính xác cụm từ "thuật toán ML" để chỉ 1 hệ thống điều chỉnh 1 số tham số nội bộ của bộ dự đoán để nó hoạt động tốt trên dữ liệu đầu vào chưa từng thấy trong tương lai. Ở đây, hãy gọi sự điều chỉnh này là *training* 1 hệ thống.

    This book will not resolve issue of ambiguity, but want to highlight upfront that, depending on context, same expressions can mean different things. However, attempt to make context sufficiently clear to reduce level of ambiguity.

    1st part of this book introduces mathematical concepts & foundations needed to talk about 3 main components of a ML system: data, models, & learning. Briefly outline these components here & revisit them in Chap. 8 once have discussed necessary mathematical concepts.

    While not all data is numerical, often useful to consider data in a number format. In this book, assume: *data* has already been appropriately converted into a numerical representation suitable for reading into a computer program. Therefore, think of data as vectors. As another illustration of how subtle words are, there are (at least) 3 different ways to think about vectors: a vector as an array of numbers (a CS view), a vector as an arrow with a direction & magnitude (a physics view), & a vector as an object that obeys addition & scaling (a mathematical view).

    A *model* is typically used to describe a process for generating data, similar to dataset at hand. Therefore, good models can also be thought of as simplified versions of real (unknown) data-generating process, capturing aspects that are relevant for modeling data & extracting hidden patterns from it. A good model can then be used to predict what would happen in real world without performing real-world experiments.

    Now come to crux of matter, *learning* component of ML. Assume given a dataset & a suitable model. *Training* model means to use data available to optimize some parameters of model w.r.t. a utility function that evaluates how well model predicts training data. Most training methods can be thought of as an approach analogous to climbing a hill to reach its peak. In this analogy, peak of hill corresponds to a maximum of some desired performance measure. However, in practice, interested in model to perform well on unseen data. Performing well on data that we have already seen (training data) may only mean that we found a good way to memorize data. However, this may not generalize well to unseen data, & in practical applications, often need to expose ML system to situations that it has not encountered before.

    Summarize main concepts of ML covered in this book:

    * Represent data as vectors.
    * Choose an appropriate model, either using probabilistic or optimization view.
    * Learn from available data by using numerical optimization methods with aim: model performs well on data not used for training.

  - 1.2. 2 Ways to Read This Book. Can consider 2 strategies for understanding mathematics for ML:

* **Bottom-up.** Building up concepts from foundational to more advanced: Often preferred approach in more technical fields, e.g. mathematics. This strategy has advantage that reader at all times is able to rely on their previously learned concepts. Unfortunately, for a practitioners many of foundational concepts are not particularly interesting by themselves, & lack of motivation means: most foundational defs are quickly forgotten.
* **Top-down.** Drilling down from practical needs to more basic requirements. This goal-driven approach has advantage that readers know at all times why they need to work on a particular concept, & there is a clear path of required knowledge. Downside of this strategy: knowledge is built on potentially shaky foundations, & readers have to remember a set of words that they do not have any way of understanding.

Decided to write this book in a modular way to separate foundational (mathematical) concepts from applications so that this book can be read in both ways. Book is split into 2 parts, where Part I lays mathematical foundations & Part II applies concepts from Part I to a set of fundamental ML problems, which form 4 pillars of ML as illustrated in Fig. 1.1: Foundations & 4 pillars of ML: regression, dimensionality reduction, density estimation, & classification. Chaps in Part I mostly build upon previous ones, but possible to skip a chap & work backward if necessary. Chaps in Part II are only loosely coupled & can be read in any order. There are many pointers forward & backward between 2 parts of book to link mathematical concepts with ML algorithms.

Of course there are > 2 *ways to read this book.* Most readers learn using a combination of top-down & bottom-up approaches, sometimes building up basic mathematical skills before attempting more complex concepts, but also choosing topics based on applications of ML.

Part I Is About Mathematics. 4 pillars of ML covered in this book require a solid mathematical foundation, which is laid out in Part I.

* Chap. 2: Represent numerical data as vectors & represent a table of such data as a matrix. Study of vectors & matrices is called *linear algebra*. Describe collection of vectors as a matrix.
* Chap. 3: Given 2 vectors representing 2 objects in real world, want to make statements about their similarity. Idea: vectors that are similar should be predicted to have similar outputs by ML algorithm (our predictor). To formalize idea of similarity between vectors, need to introduce operations that take 2 vectors as input & return a numerical value representing their similarity. Construction of similarity & distances is central to *analytic geometry*.
* Chap. 4: Introduce some fundamental concepts about matrices & *matrix decomposition*. Some operations on matrices are extremely useful in ML, & they allow for an intuitive interpretation of data & more efficient learning.
  Often consider data to be noisy observations of some true underlying signal. Hope: by applying ML, can identify signal from noise. This requires us to have a language for quantifying what "noise" means. Often would also like to have predictors that allows us to express some sort of uncertainty, e.g., to quantify confidence we have about value of prediction at a particular test data point. Quantification of uncertainty is realm of *probability theory* & is covered in Chap. 6.
  To train ML models, typically find parameters that maximize some performance measure. Many optimization techniques require concept of a gradient, which tells us direction in which to search for a solution. Chap. 5 is about *vector calculus* & details concept of gradients, which subsequently use in Chap. 7, where talk about *optimization* to find maxima/minima of functions.

Part II is about ML. 2nd part of book introduces *4 pillars of ML*. Illustrate how mathematical concepts introduced in 1st part of book are foundation for each pillar. Broadly speaking, chaps are ordered by difficulty (in ascending order).

* Chap. 8: restate 3 components of ML (data, models, & parameter estimation) in a mathematical fashion. In addition, provide some guidelines for building experimental set-ups that guard against overly optimistic evaluations of ML systems. Recall: goal: build a predictor that performs well on unseen data.
* Chap. 9: Have a close look at *linear regression*, where objective: find functions that map inputs $\mathbf{x} \in \mathbb{R}^d$ to corresponding observed function values $y \in \mathbb{R}$, which can interpret as labels of their respective inputs. Discuss classical model fitting (parameter estimation) via maximum likelihood & maximum a posteriori estimation, as well as Bayesan linear regression, where integrate parameters out instead of optimizing them.
* Chap. 10 focuses on *dimensionality reduction*, 2nd pillar in Fig. 1.1, using principal component analysis. Key objective of dimensionality reduction: find a compact, lower-dimensional representation of high-dimensional data $\mathbf{x} \in \mathbb{R}^d$, often easier to analyze than original data. Unlike regression, dimensionality reduction is only concerned about modeling data – there are no labels associated with a data point $\mathbf{x}$.
* Chap. 11: Move to 3rd pillar: *density estimation*. Objective of density estimation: find a probability distribution that describes a given dataset. Focus on Gaussian mixture models for this purpose, & discuss an iterative scheme to find parameters of this model. As in dimensionality reduction, there are no labels associated with data points $\mathbf{x} \in \mathbb{R}^d$. However, do not seek a low-dimensional representation of data. Instead, interested in a density model that describes data.
* Chap. 12 concludes book with an in-depth discussion of 4th pillar: *classification*. Discuss classification in context of support vector machines. Similar to regression (Chap. 9), have inputs $\mathbf{x}$ & corresponding labels y. However, unlike regression, where labels were real-valued, labels in classification are integers, which requires special care.

- 1.3. Exercises & Feedback. Provide some exercises in Part I, which can be done mostly by pen & paper. For Part II, provide programming tutorials (jupyter notebooks) to explore some properties of ML algorithms discussed.

• 2. Linear Algebra.

- 2.1. Systems of Linear Equations.

- 2.2. Matrices.
- 2.3. Solving Systems of Linear Equations.
- 2.4. Vector Spaces.
- 2.5. Linear Independence.
- 2.6. Basis & Rank.
- 2.7. Linear Mappings.
- 2.8. Affine Spaces.
- 2.9. Further Reading.

- 3. Analytic Geometry.
  - 3.1. Norms.
  - 3.2. Inner Products.
  - 3.3. Lengths & Distances.
  - 3.4. Angles & Orthogonality.
  - 3.5. Orthonormal Basis.
  - 3.6. Orthogonal Complement.
  - 3.7. Inner Product of Functions.
  - 3.8. Orthogonal Projections.
  - 3.9. Rotations.
  - 3.10. Further Reading.
- 4. Matrix Decompositions.
  - 4.1. Determinant & Trace.
  - 4.2. Eigenvalues & Eigenvectors.
  - 4.3. Cholesky Decomposition.
  - 4.4. Eigendecomposition & Diagonalization.
  - 4.5. Singular Value Decomposition.
  - 4.6. Matrix Approximation.
  - 4.7. Matrix Phylogeny.
  - 4.8. Further Reading.
- 5. Vector Calculus.
  - 5.1. Differentiation of Univariate Functions.
  - 5.2. Partial Differentiation & Gradients.
  - 5.3. Gradients of Vector-Valued Functions.
  - 5.4. Gradients of Matrices.
  - 5.5. Useful Identities for Computing Gradients.
  - 5.6. Backpropagating & Automatic Differentiation. A good discussion about backpropagation & chain rule is available at a blog by Tim Vieira at http://timvieira.github.io/blog/post/2017/08/18/backprop-is-not-just-the-chain-rule/. In many ML applications, find good model parameters by performing gradient descent (Sect. 7.1), which relies on fact that can compute gradient of learning objective w.r.t. parameters of model. For a given objective function, can obtain gradient w.r.t. model parameters using calculus & applying chain rule, see Sect. 5.2.2. Already had a taste in Sect. 5.3 when looked at gradient of a squared loss w.r.t. parameters of a linear regression model.

    Consider function

$$f(x) = \sqrt{x^2 + e^{x^2}} + \cos(x^2 + e^{x^2}). \tag{14}$$

    By application of chain rule, & noting that differentiation is linear, compute gradient:

$$\frac{df}{dx} = 2x \left(\frac{1}{2\sqrt{x^2 + e^{x^2}}} - \sin(x^2 + e^{x^2}) \right) (1 + e^{x^2}). \tag{15}$$

    Writing out gradient in this explicit way is often impractical since it often results in a very lengthy expression for a derivative. In practice, it means: if not careful, implementation of gradient could be significantly more expensive than computing function, which imposes unnecessary overhead. For training deep neural network models, *backpropagation* algorithm (Kelley, 1960; Bryson, 1961; Dreyfus, 1962; Rumelhart et al., 1986) is an efficient way to compute gradient of an error function w.r.t. parameters of model.

∗ 5.6.1. Gradients in a Deep Network. An area where chain rule is used to an extreme is DL, where function value $\mathbf{y}$ is computed as a many-level function composition

$$\mathbf{y} = (f_K \circ f_{K-1} \circ \cdots \circ f_1)(\mathbf{x}) = f_K(f_{K-1}(\cdots(f_1(\mathbf{x}))\cdots)), \tag{16}$$

where $\mathbf{x}$: inputs (e.g., images), $\mathbf{y}$: observations (e.g., class labels), & every function $f_i, i = 1, \ldots, K$, possesses its own parameters.

In neural networks with multiple layers, have functions $f_i(\mathbf{x}_{i-1}) = \sigma(\mathbf{A}_{i-1}\mathbf{x}_{i-1} + \mathbf{b}_{i-1})$ in ith layer. Here $\mathbf{x}_{i-1}$ is output of layer $i-1$ & σ an activation function, e.g. logistic sigmoid $\frac{1}{1+e^{-x}}$, tanh or a rectified linear unit (ReLU). In order to train these models, require gradient of a loss function L w.r.t. all model parameters $\mathbf{A}_i, \mathbf{b}_i$ for $i = 1, \ldots, K$. This also requires us to compute gradient of L w.r.t. inputs of each layer. E.g., if have inputs $\mathbf{x}$ & observations $\mathbf{y}$ & a network structure defined by

$$\mathbf{f}_0 = \mathbf{x}, \tag{17}$$

$$\mathbf{f}_i = \sigma_i(\mathbf{A}_{i-1}\mathbf{f}_{i-1} + \mathbf{b}_{i-1}),\ i = 1, \ldots, K, \tag{18}$$

see Fig. 5.8: Forward pass in a multi-layer neural network to compute loss L as a function of inputs $\mathbf{x}$ & parameters $\mathbf{A}_i, \mathbf{b}_i$. for a visualization, may be interested in finding $\mathbf{A}_i, \mathbf{b}_i$ for $i = 0, \ldots, K-1$ s.t. squared loss

$$L(\boldsymbol{\theta}) = \|\mathbf{y} - \mathbf{f}_K(\boldsymbol{\theta}, \mathbf{x})\|^2 \tag{19}$$

is minimized, where $\boldsymbol{\theta} = \{\mathbf{A}_0, \mathbf{b}_0, \ldots, \mathbf{A}_{K-1}, \mathbf{b}_{K-1}\}$.

To obtain gradients w.r.t. parameter set $\boldsymbol{\theta}$, require partial derivatives of L w.r.t. parameters $\boldsymbol{\theta}_i = \{\mathbf{A}_i, \mathbf{b}_i\}$ of each layer $i = 0, \ldots, K-1$. Chain rule allows us to determine partial derivatives as

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{K-1}} = \frac{\partial L}{\partial \mathbf{f}_K}\frac{\partial \mathbf{f}_K}{\partial \boldsymbol{\theta}_{K-1}}, \tag{20}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{K-2}} = \frac{\partial L}{\partial \mathbf{f}_K}\frac{\partial \mathbf{f}_K}{\partial \mathbf{f}_{K-1}}\frac{\partial \mathbf{f}_{K-1}}{\partial \boldsymbol{\theta}_{K-2}}, \tag{21}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_i} = \frac{\partial L}{\partial \mathbf{f}_K}\frac{\partial \mathbf{f}_K}{\partial \mathbf{f}_{K-1}}\cdots\frac{\partial \mathbf{f}_{i+2}}{\partial \mathbf{f}_{i+1}}\frac{\partial \mathbf{f}_{i+1}}{\partial \boldsymbol{\theta}_i}, \tag{22}$$

$$\tag{23}$$

Orange terms are partial derivatives of output of a layer w.r.t. its inputs, whereas blue terms are partial derivatives of output of a layer w.r.t. its parameters. Assuming, have already computed partial derivatives $\frac{\partial L}{\partial \boldsymbol{\theta}_{i+1}}$, then most of computation can be reused to compute $\frac{\partial L}{\partial \boldsymbol{\theta}_i}$. Additional terms that we need to compute are indicated by boxes. Fig. 5.9: Backward pass in a multi-layer neural network to compute gradients of loss function visualizes: gradients are passed backward through network.

∗ 5.6.2. Automatic Differentiation. Turn out: Backpropagation is a special case of a general technique in numerical analysis called *automatic differentiation*. Can think of automatic differentiation as a set of techniques to numerically (in contrast to symbolically) evaluate exact (up to machine precision) gradient of a function by working with intermediate variables & applying chain rule. Automatic differentiation applies a series of elementary arithmetic operations, e.g., addition & multiplication & elementary functions, e.g., $\sin, \cos, \exp, \log$. By applying chain rule to these operations, gradient of quite complicated functions can be computed automatically. Automatic differentiation applies to general computer programs & has forward & reverse modes. Baydin et al. (2018) give a great overview of automatic differentiation in ML.

Fig. 5.10: Simple graph illustrating flow of data from x to y via some intermediate variables a, b shows a simple graph representing data flow from inputs x to outputs y via some intermediate variables a, b. If were to compute derivative $\frac{dy}{dx}$, would apply chain rule & obtain

$$\frac{dy}{dx} = \frac{dy}{db}\frac{db}{da}\frac{da}{dx}. \tag{24}$$

Intuitively, forward & reverse mode differ in order of multiplication. Due to associativity of matrix multiplication, can choose between

$$\frac{dy}{dx} = \left(\frac{dy}{db}\frac{db}{da}\right)\frac{da}{dx} = \frac{dy}{db}\left(\frac{db}{da}\frac{da}{dx}\right). \tag{25}$$

1st eqn would be *reverse mode* because gradients are propagated backward through graph, i.e., reverse to data flow. 2nd eqn would be *forward mode*, where gradients flow with data from left to right through graph.

In general case, work with Jacobians, which can be vectors, matrices, or tensors.

Automatic differentiation is different from symbolic differentiation & numerical approximations of gradient, e.g., by using finite differences.

In following, focus on reverse mode automatic differentiation, which is backpropagation. In context of neural networks, where input dimensionality is often much higher than dimensionality of labels, reverse mode is computationally significantly cheaper than forward mode. Start with an instructive example.

Example 1. *Consider function*

$$f(x) = \sqrt{x^2 + e^{x^2}} + \cos\left(x^2 + e^{x^2}\right). \tag{26}$$

If were to implement a function f on a computer, would be able to save some computation by using intermediate variables:

$$a = x^2, b = \exp a, c = a + b, d = \sqrt{c}, e = \cos c, f = d + e. \tag{27}$$

This is same kind of thinking process that occurs when applying chain rule. Note: preceding set of equations requires fewer operations than a direct implementation of function $f(x)$. Corresponding computation graph in Fig. 5.11: Computation graph with inputs x, function values f, & intermediate variables a, b, c, d, e. *shows flow of data & computations required to obtain function value f.*
Set of equations that include intermediate variables can be thought of as a computation graph, a representation that is widely used in implementations of neural network software libraries. Can directly compute derivatives of intermediate variables w.r.t. their corresponding inputs by recalling definition of derivative of elementary functions. Obtain:

$$\frac{\partial a}{\partial x} = 2x, \frac{\partial b}{\partial a} = e^a, \frac{\partial c}{\partial a} = 1 = \frac{\partial c}{\partial b}, \frac{\partial d}{\partial c} = \frac{1}{2\sqrt{c}}, \frac{\partial e}{\partial c} = -\sin c, \frac{\partial f}{\partial d} = 1 = \frac{\partial f}{\partial e}. \tag{28}$$

By looking at computation graph in Fig. 5.11, can compute $\partial_x f$ by working backward from output & obtain

$$\frac{\partial f}{\partial c} = \frac{\partial f}{\partial d}\frac{\partial d}{\partial c} + \frac{\partial f}{\partial e}\frac{\partial e}{\partial c}, \frac{\partial f}{\partial b} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b}, \frac{\partial f}{\partial a} = \frac{\partial f}{\partial b}\frac{\partial b}{\partial a} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial a}, \frac{\partial f}{\partial x} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial x}. \tag{29}$$

Note: implicitly applied chain rule to obtain $\frac{\partial f}{\partial x}$. By substituting results of derivatives of elementary functions, get

$$\frac{\partial f}{\partial c} = 1 \cdot \frac{1}{2\sqrt{c}} + 1 \cdot (-\sin c), \frac{\partial f}{\partial b} = \frac{\partial f}{\partial c} \cdot 1, \frac{\partial f}{\partial a} = \frac{\partial f}{\partial b}e^a + \frac{\partial f}{\partial c} \cdot 1, \frac{\partial f}{\partial x} = \frac{\partial f}{\partial a} \cdot 2x. \tag{30}$$

By thinking of each of derivatives above as a variable, observe: computation required for calculating derivative is of similar complexity as computation of function itself. This is quite counterintuitive since mathematical expression for derivative $\frac{\partial f}{\partial x}$ is significantly more complicated than mathematical expression of function $f(x)$.
Automatic differentiation is a formalization of Example 5.14. Let $x_1, \ldots, x_d$: input variables to function $x_{d+1}, \ldots, x_{D-1}$ be intermediate variables, & x_D: output variable. Then computation graph can be expressed as follows: (5.143)

$$\text{For } i = d+1, \ldots, D: \; x_i = g_i(x_{\text{Pa}(x_i)}), \tag{31}$$

where $g_i(\cdot)$: elementary functions & $x_{\text{Pa}(x_i)}$: parent nodes of variable x_i in graph. Given a function defined in this way, can use chain rule to compute derivative of function in a step-by-step fashion. Recall by def $f = x_D$ & hence

$$\frac{\partial f}{\partial x_D} = 1. \tag{32}$$

For other variables x_i, apply chain rule (5.145)

$$\frac{\partial f}{\partial x_i} = \sum_{x_j : x_i \in \text{Pa}(x_i)} \frac{\partial f}{\partial x_j}\frac{\partial x_j}{\partial x_i} = \sum_{x_j : x_i \in \text{Pa}(x_i)} \frac{\partial f}{\partial x_j}\frac{\partial g_j}{\partial x_i}, \tag{33}$$

where $\text{Pa}(x_j)$: set of parent nodes of x_j in computation graph. Equation (5.143) is forward propagation of a function, whereas (5.145) is backpropagation of gradient through computation graph. For neural network training, backpropagate error of prediction w.r.t. label.

Auto-differentiation in reverse mode requires a parse tree.

Automatic differentiation approach above works whenever have a function that can be expressed as a computation graph, where elementary functions are differentiable. In fact, function may not even be a mathematical function but a computer program. However, not all computer programs can be automatically differentiated, e.g., if cannot find differential elementary functions. Programming structures, e.g. `for` loops & `if` statements, require more care as well.

- 5.7. Higher-Order Derivatives.
- 5.8. Linearization & Multivariate Taylor Series.
- 5.9. Further Reading. Further details of matrix differentials, along with a short review of required linear algebra, can be found in Magnus & Neudecker (2007). Automatic differentiation has had a long history, & refer to Griewank & Walther (2003), Griewank & Walther (2008), & Elliott (2009) & the references therein.

  In ML (& other disciplines), often need to compute expectations, i.e., need to solve integrals of form (5.181)

$$\mathbb{E}_{\mathbf{x}}[f(\mathbf{x})] = \int f(\mathbf{x})p(\mathbf{x})\,\mathrm{d}\mathbf{x}. \tag{34}$$

  Even if $p(\mathbf{x})$ is in convenient form (e.g., Gaussian), this integral generally cannot be solved analytically. Taylor series expansion of f is 1 way of finding an approximate solution: Assuming $p(\mathbf{x}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is Gaussian, then 1st-order Taylor

series expansion around $\boldsymbol{\mu}$ locally linearizes nonlinear function f. For linear functions, can compute mean (& covariance) exactly if $p(\mathbf{x})$ is Gaussian distributed (Sect. 6.5). This property is heavily exploited by *extended Kalman filter* (Maybeck, 1979) for online state estimation in nonlinear dynamical systems (also called "state-space models"). Other deterministic ways to approximate integral in (5.181) are *unscented transform* (biến đổi không mùi) (Julier & Uhlmann, 1997), which does not require any gradients, or *Laplace approximation* (MacKay, 2003; Bishop, 2006; Murphy, 2012), which uses a 2nd-order Taylor series expansion (requiring Hessian) for a local Gaussian approximation of $p(\mathbf{x})$ around its mode.

- 6. Probability & Distributions. Probability, loosely speaking, concerns study of uncertainty. Probability can be thought of as fraction of times an event occurs, or as a degree of belief about an event. Then would like to use this probability to measure chance of sth occurring in an experiment. Often quantify uncertainty in data, uncertainty in ML model, & uncertainty in predictions produced by model. Quantifying uncertainty requires idea of a *random variable*, which is a function that maps outcomes of random experiments to a set of properties that we are interested in. Associated with random variable is a function that measures probability that a particular outcome (or set of outcomes) will occur; this called *probability distribution*.

  Probability distributions are used as a building block for other concepts, e.g. probabilistic modeling (Sect. 8.4), graphical models (Sect. 8.5), & model selection (Sect. 8.6). In next sect, present 3 concepts that define a probability space (sample space, events, & probability of an event) & how they are related to a 4th concept called random variable. Presentation is deliberately slightly hand wavy since a rigorous presentation may occlude intuition behind concepts. An outline of concepts presented in this chap are shown in Fig. 6.1.

  - 6.1. Construction of a Probability Space. Theory of probability aims at defining a mathematical structure to describe random outcomes of experiments. E.g., when tossing a single coin, cannot determine outcome, but by doing a large number of coin tosses, can observe a regularity in average outcome. Using this mathematical structure of probability, goal: perform automated reasoning, & in this sense, probability generalizes logical reasoning (Jaynes, 2003).
  - 6.2. Discrete & Continuous Probabilities.
  - 6.3. Sum Rule, Product Rule, & Bayes' Theorem.
  - 6.4. Summary Statistics & Independence.
  - 6.5. Gaussian Distribution.
  - 6.6. Conjugacy & Exponential Family.
  - 6.7. Change of Variables/Inverse Transform.
  - 6.8. Further Reading.

- 7. Continuous Optimization. Since ML algorithms are implemented on a computer, mathematical formulations are expressed as numerical optimization methods. This chap describes basic numerical methods for training ML models. Training a ML model often boils down to finding a good set of parameters. Notion of "good" is determined by objective function or probabilistic model, which we will see examples of in 2nd part of this book. Given an objective function, finding best value is done using optimization algorithms.

  > Since consider data & models in $\mathbb{R}^D$, optimization problems we face are *continuous* optimization problems, as opposed to *combinatorial* optimization problems for discrete variables.

  This chap covers 2 main branches of continuous optimization (Fig. 7.1: A mind map of concepts related to optimization, as presented in this chap. There are 2 main ideas: gradient descent & convex optimization.): unconstrained & constrained optimization. Assume in this chap: objective function is differentiable (Chap. 5), hence have access to a gradient at each location in space to help us find optimum value. By convention, most objective functions in ML are intended to be minimized, i.e., best value is minimum value. Intuitively finding best value is like finding values of objective function, & gradients point us uphill. Idea: move downhill (opposite to gradient) & hope to find deepest point. For unconstrained optimization, this is only concept we need, but there are several design choices, discussed in Sect. 7.1. For constrained optimization, need to introduce other concepts to manage constraints (Sect. 7.2). Will also introduce a special class of problems (convex optimization problems in Sect. 7.3) where we can make statements about reaching global optimum.

  Consider function in Fig. 7.2: Example objective function. Negative gradients are indicated by arrows, & global minimum is indicated by dashed blue line. Function has a *global minimum* around $x = -4.5$, with a function value of approximately -47. Since function is "smooth," gradients can be used to help find minimum by indicating whether should take a step to right or left. This assumes that are in correct bowl, as there exists another *local minimum* around $x = 0.7$. Recall: can solve $\forall$ stationary points of a function by calculating its derivative & setting it to 0. For $l(x) = x^4 + 7x^3 + 5x^2 - 17x + 3$, obtain corresponding gradient as $\frac{dl(x)}{dx} = 4x^3 + 21x^2 + 10x - 17$.

  > Stationary points are real roots of derivative, i.e., points that have zero gradient.

  Since this is a cubic equation, it has in general 3 solutions when set to 0. In example, 2 of them are minimums & one is a maximum (around $x = -1.4$). To check whether a stationary point is a minimum or maximum, need to take derivative a 2nd time & check whether 2nd derivative is positive or negative at stationary point. In our case, 2nd derivative is $\frac{d^2l(x)}{dx^2} = 12x^2 + 42x + 10$. By substituting our visually estimated values of $x = -4.5, -1.4, 0.7$, observe: as expected middle point is a maximum $\frac{d^2l(x)}{dx^2} < 0$ & the other 2 stationary points are minimums.

Note: have avoided analytically solving for values of x in previous discussion, although for low-order polynomials e.g. preceding, could do so. In general, unable to find analytic solutions, & hence need to start at some value, say $x_0 = -6$, & follow negative gradient. Negative gradient indicates: should go right, but not how far (this is called *step-size*). Furthermore, if had started at right side (e.g., $x_0 = 0$) negative gradient would have led us to wrong minimum. Fig. 7.2 illustrates fact: for $x > -1$, negative gradient points toward minimum on right of figure, which has a larger objective value.

> According to Abel–Ruffini theorem, there is in general no algebraic solution for polynomials of degree ≥ 5 (Abel, 1826).

In Sect. 7.3, will learn about a class of functions called *convex functions*, that do not exhibit this tricky dependency on starting point of optimization algorithm. For convex functions, all local minimums are global minimum. Turn out: many ML objective functions are designed s.t. they are convex, & see an example in Chap. 12.

> For convex functions all local minima are global minimum.

Discussion in this chap so far was about a 1D function, where able to visualize ideas of gradients, descent directions, & optimal values. In rest of this chap, develop same ideas in high dimensions. Unfortunately, can only visualize concepts in 1D, but some concepts do not generalize directly to higher dimensions, therefore some care needs to be taken when reading.

- 7.1. Optimization Using Gradient Descent.
- 7.2. Constrained Optimization & Lagrange Multipliers.
- 7.3. Convex Optimization.
- 7.4. Further Reading.

PART II: CENTRAL MACHINE LEARNING PROBLEMS.

- 8. When Models Meet Data. In 1st part of book, introduced mathematics that form foundations of many ML methods. Hope: a reader would be able to learn rudimentary forms (hình thức thô sơ) of language of mathematics from 1st part, which we will now use to describe & discuss ML. 2nd part of book introduces 4 pillars of ML:
  - Chap. 9: Regression
  - Chap. 10: Dimensionality reduction
  - Chap. 11: Density estimation
  - Chap. 12: Classification

  Main aim of this part of book: illustrate how mathematical concepts introduced in 1st part of book can be used to design ML algorithms that can be used to solve tasks within remit of 4 pillars (nhiệm vụ của 4 trụ cột). Do not intend to introduce advanced ML concepts, but instead to provide a set of practical methods that allow reader to apply knowledge they gained from 1st part of book. It also provides a gateway to wider ML literature for readers already familiar with mathematics.

  - 8.1. Data, Models, & Learning. Worth at this point, to pause & consider problem that a ML algorithm is designed to solve. There are 3 major components of a ML system: data, models, & learning. Main question of ML: "What do we mean by good models?". Word *model* has many subtleties, & revisit it multiple times in this chap. Also not entirely obvious how to objectively define word "good". 1 of guiding principles of ML: good models should perform well on unseen data. This requires us to define some performance metrics, e.g. accuracy or distance from ground truth, as well as figuring out ways to do well under these performance metrics. This chap covers a few necessary bits & pieces of mathematical & statistical language that are commonly used to talk about ML models. By doing so, briefly outline current best practices for training a model s.t. resulting predictor does well on data that we have not yet seen.

    There are 2 different senses in which use phrase "ML algorithm": training & prediction. Describe these ideas in this chap, as well as idea of selecting among different models. Introduce framework of empirical risk minimization in Sect. 8.2, principle of maximum likelihood in Sect. 8.3 & idea of probabilistic models in Sect. 8.4. Briefly outline a graphical language for specifying probabilistic models in Sect. 8.5 & finally discuss model selection in Sect. 8.6. Rest of this sect expands upon 3 main components of ML: data, models, & learning.

    * 8.1.1. Data as Vectors. Assume: our data can be read by a computer, & represented adequately in a numerical format. Data is assumed to be tabular Fig. 8.1: Examples data from a fictitious human resource database that is not in a numerical format, where think of each row of table as representing a particular instance or example, & each row of table to be a particular feature. In recent years, ML has been applied to many types of data that do not obviously come in tabular numerical format, e.g. genomic sequences, text, & image contents of a webpage, & social media graphs. Do not discuss important & challenging aspects of identifying good features. Many of these aspects depend on domain expertise & require careful engineering, &, in recent years, they have been put under umbrella of DS (Stray, 2016; Adhikari & DeNero, 2018).

      > Data is assumed to be in a tidy format (Wickham, 2014; Codd, 1990).

      Even when have data in tabular format, there are still choices to be made to obtain a numerical representation. E.g., in Table 8.1: Example data from a fictitious human resource database that is not in a numerical format., gender column (a categorical variable) may be converted into numbers 0 representing "Male" & 1 representing "Female". Alternatively, gender could be represented by numbers ± 1, resp., as shown in Table 8.2: Example data from a fictitious human resource

database, converted to a numerical format. Furthermore, often important to use domain knowledge when constructing representation, e.g. knowing that university degrees progress from bachelor's to master's to PhD or realizing: postcode provided is not just a string of characters but actually encodes an area in London. In Table 8.2, converted data from Table 8.1 to a numerical format, & each postcode is represented as 2 numbers, a latitude & longitude. Even numerical data that could potentially be directly read into a ML algorithm should be carefully considered for units, scaling, & constraints. Without additional information, one should shift & scale all columns of dataset s.t. they have an empirical mean of 0 & an empirical variance of 1. For purposes of this book, assume: a domain expert already converted data appropriately, i.e., each input $\mathbf{x}_n$ is a d-dimensional vector of real numbers, which are called *features, attributes*, or *covariates* (các tính năng, thuộc tính hoặc biến phụ thuộc). Consider a dataset to be of form as illustrated by Table 8.2. Observe: have dropped Nam column of Table 8.1 in new numerical representation. There are 2 main reasons why this is desirable:

1. Do not expect identifier (Name) to be informative for a ML task;
2. May wish to anonymize data to help protect privacy of employees.

In this part of book, use N to denote number of examples in a dataset & index examples with lowercase $n = 1, \ldots, N$. Assume: are given a set of numerical data, represented as an array of vectors. Each row is a particular individual $\mathbf{x}_n$, often referred to as an *example* or *data point* in ML. Subscript n refers to fact: this is nth example out of a total of N examples in dataset. Each column represents a particular feature of interest about example, & index features as $d = 1, \ldots, D$. Recall data is represented as vectors, i.e., each example (each data point) is a D-dimensional vector. Orientation of table originates from database community, but for some ML algorithms, more convenient to represent examples as column vectors.

Consider problem of predicting annual salary from age, based on data in Table 8.2. This is called a *supervised learning problem* where have a label y_n (salary) associated with each example $\mathbf{x}_n$ (age). Label y_n has various other names, including *target, response variable, & annotation.* A dataset is written as a set of example-label pairs $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n), \ldots, (\mathbf{x}_N, y_N)\}$. Table of examples $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ is often concatenated, & written as $\mathbf{X} \in \mathbb{R}^{N \times D}$. Fig. 8.1: Toy data for linear regression. Training data in (x_n, y_n) pairs from rightmost 2 columns of Table 8.2. Interested in salary of a person aged 60 ($x = 60$) illustrated as a vertical dashed red line, which is not part of training data. illustrates dataset consisting of 2 rightmost columns of Table 8.2, where $x =$ age & $y =$ salary.

Use concepts introduced in 1st part of book to formalize ML problems e.g. that in previous paragraph. Representing data as vectors $\mathbf{x}_n$ allows us to use concepts from linear algebra. In many ML algorithms, need to additionally be able to compare 2 vectors. As see in Chaps. 9 & 12, computing similarity or distance between 2 examples allows us to formalize intuition that examples with similar features should have similar labels. Comparison of 2 vectors requires: construct a geometry & allows us to optimize resulting learning problem using techniques from Chap. 7.

Since have vector representations of data, can manipulate data to find potentially better representations of it. Discuss finding good representations in 2 ways: finding lower-dimensional approximations of original feature vector, & using nonlinear higher-dimensional combinations of original feature vector. In Chap. 10, see an example of finding a low-dimensional approximation of original data space by finding principal components. Finding principal components is closely related to concepts of eigenvalue & singular value decomposition. For high-dimensional representation, see an explicit *feature map* $\phi(\cdot)$ that allows us to represent inputs $\mathbf{x}_n$ using a higher-dimensional representation $\phi(\mathbf{x}_n)$. Main motivation for higher-dimensional representations: can construct new features as nonlinear combinations of original features, which in turn may make learning problem easier. Discuss feature map in Sect. 9.2 & show how this feature map leads to a *kernel* in Sect. 12.4. In recent years, DL methods (Goodfellow et al., 2016) have shown promise in using data itself to learn new good features & have been very successful in areas, e.g. computer vision, speech recognition, & natural language processing. Will not cover neural networks in this part of book, but reader is referred to Sect. 5.6 for mathematical description of backpropagation, a key concept for training neural networks.

∗ 8.1.2. Models as Functions. Once have data in an appropriate vector representation, can get to business of constructing a predictive function (known as a *predictor*). In Chap. 1, did not yet have language to be precise about models. Using concepts from 1st part of book, can now introduce what "model" means. Present 2 major approaches in this book: a predictor as a function, & a predictor as a probabilistic model. Describe former here & latter in next subsection.

A *predictor* is a function that, when given a particular input example (in our case, a vector of features), produces an output. For now, consider output to be a single number, i.e., a real-valued scalar output. This can be written as $f : \mathbb{R}^D \to \mathbb{R}$, where input vector $\mathbf{x}$: D-dimensional (has D features), & function f then applied to it (written as $f(\mathbf{x})$) returned a real number. Fig. 8.2: Example function (black solid diagonal line) & its prediction at $x = 60$, i.e., $f(60) = 100$. illustrates a possible function that can be used to compute value of prediction for input values x.

In this book, do not consider general case of all functions, which would involve need for functional analysis. Instead, consider special case of linear functions

$$\boxed{f(\mathbf{x}) = \boldsymbol{\theta}^\top \mathbf{x} + \theta_0} \tag{35}$$

for unknown $\boldsymbol{\theta}, \theta_0$. This restriction means: contents of Chaps. 2–3 suffice for precisely stating notion of a predictor for non-probabilistic (in contrast to probabilistic view described next) view of ML. Linear functions strike a good balance between generality of problems that can be solved & amount of background mathematics that is needed.

∗ 8.1.3. Models as Probability Distributions. Often consider data to be noisy observations of some true underlying effect, & hope: by applying ML, can identify signal from noise. This requires us to have a language for quantifying effect of noise. Often would also like to have predictors that express some sort of uncertainty, e.g., to quantify confidence we have about value of prediction for a particular test data point. As seen in Chap. 6, probability theory provides a language for

quantifying uncertainty. Fig. 8.3: Example function (black solid diagonal line) & its predictive uncertainty at $x = 60$ (drawn as a Gaussian) illustrates predictive uncertainty of function as a Gaussian distribution.

Instead of considering a predictor as a single function, could consider predictors to be probabilistic models, i.e., models describing distribution of possible functions. Limit ourselves in this book to special case of distributions with finite-dimensional parameters, which allows us to describe probabilistic models without needing stochastic processes & random measures. For this special case, can think about probabilistic models as multivariate probability distributions, which already allow for a rich class of models.

Introduce how to use concepts from probability (Chap. 6) to define ML models in Sect. 8.4, & introduce a graphical language for describing probabilistic models in a compact way in Sect. 8.5.

∗ 8.1.4. Learning is Finding Parameters. Goal of learning: find a model & its corresponding parameters s.t. resulting predictor will perform well on unseen data. There are conceptually 3 distinct algorithmic phases when discussing ML algorithms:

1. Prediction or inference – Dự đoán hoặc suy luận
2. Training or parameter estimation
3. Hyperparameter tuning or model selection – Điều chỉnh siêu tham số hoặc lựa chọn mô hình

Prediction phase is when use a trained predictor on previously unseen test data. I.e., parameters & model choice is already fixed & predictor is applied to new vectors representing new input data points. As outlined in Chap. 1 & previous subsection, will consider 2 schools of ML in this book, corresponding to whether predictor is a function or a probabilistic model. When have a probabilistic model (discussed further in Sect. 8.4) prediction phase is called *inference*.

Remark 1. *Unfortunately, there is no agreed upon naming for different algorithmic phases. Word "inference" is sometimes also used to mean parameter estimation of a probabilistic model, & less often may be also used to mean prediction for non-probabilistic models.*

Training or parameter estimation phase is when adjust our predictive model based on training data. Would like to find good predictors given training data, & there are 2 main strategies for doing so: finding best predictor based on some measure of quality (sometimes called *finding a point estimate*), or using Bayesian inference. Finding a point estimate can be applied to both types of predictors, but Bayesian inference requires probabilistic models.

For non-probabilistic model, follow principle of *empirical risk minimization*, described in Sect. 8.2. Empirical risk minimization directly provides an optimization problem for finding good parameters. With a statistical model, principle of *maximum likelihood* is used to find a good set of parameters (Sect. 8.3). Can additionally model uncertainty of parameters using a probabilistic model (Sect. 8.4).

Use numerical methods to find good parameters that "fit" data, & most training methods can be thought of as hill-climbing approaches to find maximum of an objective, e.g. maximum of a likelihood. To apply hill-climbing approaches us gradients described in Chap. 5 & implement numerical optimization approaches from Chap. 7.

Convention in optimization: minimize objectives. Hence, there is often an extra minus sign in ML objectives.

Interested in learning a model based on data s.t. it performs well on future data. Not enough for model to only fit training data well, predictor needs to perform well on unseen data. Simulate behavior of our predictor on future unseen data using *cross-validation* (Sect. 8.2.4). To achieve goal of performing well on unseen data, need to balance between fitting well on training data & finding "simple" explanations of phenomenon. This trade-off is achieved using regularization (Sect. 8.2.3) or by adding a prior (Sect. 8.3.2). In philosophy, this is considered to be neither induction nor deduction, but is called *abduction* (bắt cóc, dụ dỗ). According to *Stanford Encyclopedia of Philosophy*, abduction is process of inference to best explanation (Douven, 2017).

Often need to make high-level modeling decisions about structure of predictor, e.g. number of components to use or class of probability distributions to consider. Choice of number of components is an example of a *hyperparameter*, & this choice can affect performance of model significantly. Problem of choosing among different models is called *model selection*, described in Sect. 8.6. For non-probabilistic models, model selection is often done using *nested cross-validation*, described in Sect. 8.6.1. Also use model selection to choose hyperparameters of our model.

Remark 2. *Distinction between parameters & hyperparameters is somewhat arbitrary, & is mostly driven by distinction between what can be numerically optimized vs. what needs to use search techniques. Another way to consider distinction: consider parameters as explicit parameters of a probabilistic model, & to consider hyperparameters (higher-level parameters) as parameters that control distribution of these explicit parameters.*

In following sects, look at 3 flavors of ML: empirical risk minimization (Sect. 8.2), principle of maximum likelihood (Sect. 8.3), & probabilistic modeling (Sect. 8.4).

◦ 8.2. Empirical Risk Minimization. After having all mathematics under our belt, now in a position to introduce what it means to learn. "Learning" part of ML boils down to estimating parameters based on training data.

In this sect, consider case of a predictor that is a function, & consider case of probabilistic models in Sect. 8.3. Describe idea of empirical risk minimization, which was originally popularized by proposal of support vector machine, described in Chap. 12. However, its general principles are widely applicable & allow us to ask question of what is learning without explicitly constructing probabilistic models. There are 4 main design choices:

1. Sect. 8.2.1: What is set of functions we allow predictor to take?
2. Sect. 8.2.2: How do we measure how well predictor performs on training data?
3. Sect. 8.2.3: How do we construct predictors from only training data that performs well on unseen test data?
4. Sect. 8.2.4: What is procedure for searching over space of models?

* 8.2.1. Hypothesis Class of Functions. Assume given N examples $\mathbf{x}_n \in \mathbb{R}^D$ & corresponding scalar labels $y_n \in \mathbb{R}$. Consider supervised learning setting, where obtain pairs $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$. Given this data, would like to estimate a predictor $f(\cdot, \boldsymbol{\theta}) : \mathbb{R}^D \to \mathbb{R}$, parametrized by $\boldsymbol{\theta}$. Hope to be able to find a good parameter $\boldsymbol{\theta}^\star$ s.t. fit data well, i.e.,

$$f(\mathbf{x}_n, \boldsymbol{\theta}^\star) \approx y_n, \ \forall n = 1, \ldots, N. \tag{36}$$

Use notation $\hat{y}_n \coloneqq f(\mathbf{x}_n, \boldsymbol{\theta}^\star)$ to represent output of predictor.

Remark 3. *For ease of presentation, describe empirical risk minimization in terms of supervised learning (where have labels). This simplifies definition of hypothesis class & loss function. Also common in ML to choose a parameterized class of functions, e.g. affine functions.*

Affine functions are often referred to as linear functions in ML.

Example 2. *Introduce problem of ordinary least-squares regression to illustrate empirical risk minimization. A more comprehensive account of regression is given in Chap. 9. When label y_n is real-valued, a popular choice of function class for predictors is set of affine functions. Choose a more compact notation for an affine function by concatenating an additional unit feature $x^{(0)} = 1$ to $\mathbf{x}_n$, i.e., $\mathbf{x}_n = [1, x_n^{(1)}, x_n^{(2)}, \ldots, x_n^{(d)}]^\top$. Parameter vector is correspondingly $\boldsymbol{\theta} = [\theta_0, \theta_1, \ldots, \theta_D]^\top$, allowing us to write predictor as a linear function*

$$f(\mathbf{x}_n, \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \mathbf{x}_n. \tag{37}$$

This linear predictor is equivalent to affine model

$$f(\mathbf{x}_n, \boldsymbol{\theta}) = \theta_0 + \sum_{d=1}^{D} \theta_d x_n^{(d)}. \tag{38}$$

Predictor takes vector of features representing a single example $\mathbf{x}_n$ as input & produces a real-valued output, i.e., $f : \mathbb{R}^{D+1} \to \mathbb{R}$. Previous figures in this chap had a straight line as a predictor, i.e., have assumed an affine function.
Instead of a linear function, may wish to consider nonlinear functions as predictors. Recent advances in neural networks allow for efficient computation of more complex nonlinear function classes.

Given class of functions, want to search for a good predictor. Now move on to 2nd ingredient of empirical risk minimization: how to measure how well predictor fits training data.

* 8.2.2. Loss Function for Training. Consider label y_n for a particular example; & corresponding prediction $\hat{y}_n$ that we make based on $\mathbf{x}_n$. To define what it means to fit data well, need to specify a *loss function* $l(y_n, \hat{y}_n)$ that takes ground truth label & prediction as input & produces a nonnegative number (referred to as loss) representing how much error we have made on this particular prediction. Goal for finding a good parameter vector $\boldsymbol{\theta}^*$: minimize average loss on set of N training examples.

Expression "error" is often used to mean loss.

1 assumption commonly made in ML: set of example $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$ is *independent & identically distributed*. Word independent (Sect. 6.4.5) means: 2 data points $(\mathbf{x}_i, y_i), (\mathbf{x}_j, y_j)$ do not statistically depend on each other, meaning: empirical mean is a good estimate of population mean (Sect. 6.4.1). This implies: can use empirical mean of loss on training data. For a given *training set* $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$, introduce notation of an example matrix $\mathbf{X} \coloneqq [\mathbf{x}_1, \ldots, \mathbf{x}_N]^\top \in \mathbb{R}^{N \times D}$ & a label vector $\mathbf{y} \coloneqq [y_1, \ldots, y_N]^\top \in \mathbb{R}^N$. Using this matrix notation, average loss is given by (8.6)

$$\mathbf{R}_{\text{emp}}(f, \mathbf{X}, \mathbf{y}) = \frac{1}{N} \sum_{n=1}^{N} l(y_n, \hat{y}_n), \tag{39}$$

where $\hat{y}_n = f(\mathbf{x}_n, \boldsymbol{\theta})$. (8.6) is called *empirical risk* & depends on 3 arguments, predictor f & data $\mathbf{X}, \mathbf{y}$. This general strategy for learning is called *empirical risk minimization*.

Example 3 (Least-Square Loss)**.** *Continuing example of least-squares regression, specify: measure cost of making an error during training using squared loss $l(y_n, \hat{y}_n) = (y_n - \hat{y}_n)^2$. Wish to minimize empirical risk (8.6), which is average of losses over data*

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^D} \frac{1}{N} \sum_{n=1}^{N} (y_n - f(\mathbf{x}_n, \boldsymbol{\theta}))^2, \tag{40}$$

where substituted predictor $\hat{y}_n = f(\mathbf{x}_n, \boldsymbol{\theta})$. By using our choice of a linear predictor $f(\mathbf{x}_n, \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \mathbf{x}_n$, obtain optimization problem

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^D} \frac{1}{N} \sum_{n=1}^{N} (y_n - \boldsymbol{\theta}^\top \mathbf{x}_n)^2. \tag{41}$$

This equation can be equivalently expressed in matrix form

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^D} \frac{1}{N} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2. \tag{42}$$

This is known as least-squares problem. *There exists a closed-form analytic solution for this by solving normal equations, discussed in Sect. 9.2.*

Not interested in a predictor that only performs well on training data. Instead, seek a predictor that performs well (has low risk) on unseen test data. More formally, interested in finding a predictor f (with parameters fixed) that minimizes *expected risk*

$$\mathbf{R}_{\text{true}}(f) = \mathbb{E}_{\mathbf{x},y}[l(y, f(\mathbf{x}))], \tag{43}$$

where y: label, $f(\mathbf{x})$: prediction based on example $\mathbf{x}$. Notation $\mathbf{R}_{\text{true}}(f)$ indicates: this is the true risk if had access to an infinite amount of data (NQBH: still not sure? may be all data). Expectation is over (infinite) set of all possible data & labels. There are 2 practical questions that arise from our desire to minimize expected risk, which address in following 2 subsects:

1. How should we change our training procedure to generalize well?
2. How do we estimate expected risk from (finite) data?

Another phrase commonly used for expected risk is "population risk".

Remark 4. *Many ML tasks are specified with an associated performance measure, e.g., accuracy of prediction or root mean squared error. Performance measure could be more complex, be cost sensitive, & capture details about particular application. In principle, design of loss function for empirical risk minimization should correspond directly to performance measure specified by ML task. In practice, there is often a mismatch between design of loss function & performance measure. This could be due to issues e.g. ease of implementation or efficiency of optimization.*

∗ 8.2.3. Regularization to Reduce Overfitting. This sect describes an addition to empirical risk minimization that allows it to generalize well (approximately minimizing expected risk). Recall: aim of training a ML predictor is so that we can perform well on unseen data, i.e., predictor generalizes well. Simulate this unseen data by holding out a proportion of whole dataset. This hold out set is referred to as *test set*. Given a sufficiently rich class of functions for predictor f, can essentially memorize training data to obtain zero empirical risk. While this is great to minimize loss (& therefore risk) on training data, would not expect predictor to generalize well to unseen data. In practice, have only a finite set of data, & hence split our data into a training & a test set. Training set is used to fit model, & test set (not seen by ML algorithm during training) is used to evaluate generalization performance. Important for user to not cycle back to a new round of training after having observed test set. Use subscripts $_{\text{train}, \text{test}}$ to denote training & test sets, resp. Revisit this idea of using a finite dataset to evaluate expected risk in Sect. 8.2.4.

Even knowing only performance of predictor on test set leaks information (Blum & Hardt, 2015).

Turn out: empirical risk minimization can lead to *overfitting*, i.e., predictor fits too closely to training data & does not generalize well to new data (Mitchell, 1997). This general phenomenon of having very small average loss on training set but large average loss on test set tends to occur when have little data & a complex hypothesis class. For a particular predictor f (with parameters fixed), phenomenon of overfitting occurs when risk estimate from training data $\mathbf{R}_{\text{emp}}(f, \mathbf{X}_{\text{train}}, \mathbf{y}_{\text{train}})$ underestimates expected risk $\mathbf{R}_{\text{true}}(f)$. Since estimate expected risk $\mathbf{R}_{\text{true}}(f)$ by using empirical risk on test set $\mathbf{R}_{\text{emp}}(f, \mathbf{X}_{\text{test}}, \mathbf{y}_{\text{test}})$ if test risk is much larger than training risk, this is an indication of overfitting. Revisit idea of overfitting in Sect. 8.3.3.

Therefore, need to somehow bias search for minimizer of empirical risk by introducing a penalty term, which makes it harder for optimizer to return an overly flexible predictor. In ML, penalty term is referred to as *regularization*. Regularization is a way to compromise between accurate solution of empirical risk minimization & size or complexity of solution.

Example 4 (Regularized Least Squares)**.** *Regularization is an approach that discourages complex or extreme solutions to an optimization problem. Simplest regularization: replace least-squares problem*

$$\min_{\boldsymbol{\theta}} \frac{1}{N} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2, \tag{44}$$

in previous example with "regularized" problem by adding a penalty term involving only $\boldsymbol{\theta}$:

$$\min_{\boldsymbol{\theta}} \frac{1}{N} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2 + \lambda \|\boldsymbol{\theta}\|^2. \tag{45}$$

Additional term $\|\boldsymbol{\theta}\|^2$ is called regularizer, & parameter λ: regularization parameter. Regularization parameter trades off minimizing loss on training set & magnitude of parameters $\boldsymbol{\theta}$. Often happens: magnitude of parameter values become relatively large if run into overfitting (Bishop, 2006).

Regularization term is sometimes called *penalty term*, which biases vector $\boldsymbol{\theta}$ to be closer to origin. Idea of regularization also appears in probabilistic models as prior probability of parameters. Recall from Sect. 6.6: for posterior distribution to be of same form as prior distribution, prior & likelihood need to be conjugate. Revisit this idea in Sect. 8.3.2. See in Chap. 12: idea of regularizer is equivalent to idea of a large margin.

∗ 8.2.4. Cross-Validation to Access Generalization Performance. Mentioned in previous sect: measure generalization error by estimating it by applying predictor on test data. This data is also sometimes referred to as *validation set*. Validation set is a subset of available training data that we keep aside. A practical issue with this approach: amount of data is limited, & ideally we would use as much of data available to train model. This would require us to keep our validation set $\mathcal{V}$ small, which then would lead to a noisy estimate (with high variance) of predictive performance. 1 solution to these contradictory objectives (large training set, large validation set): use *cross-validation*. K-fold cross-validation effectively partitions data into K chunks, $K-1$ of which form training set $\mathcal{R}$, & last chunk serves as validation set $\mathcal{V}$ (similar to idea outlined previously). Cross-validation iterates through (ideally) all combinations of assignments of chunks to $\mathcal{R}, \mathcal{V}$;

see Fig. 8.4: K-fold cross-validation. Dataset is divided into $K = 5$ chunks, $K - 1$ of which serve as training set (blue) & 1 as validation set (orange hatch). This procedure is repeated $\forall K$ choices for validation set, & performance of model from K runs is averaged.

Partition our dataset into 2 sets $\mathcal{D} = \mathcal{R} \cup \mathcal{V}$, s.t. they do not overlap $\mathcal{R} \cap \mathcal{V} = \emptyset$, where $\mathcal{V}$ is validation set, & train our model on $\mathcal{R}$. After training, assess performance of predictor f on validation set $\mathcal{V}$ (e.g., by computing root mean square error (RMSE) of trained model on validation set). More precisely, for each partition k training data $\mathcal{R}^{(k)}$ produces a predictor $f^{(k)}$, which is then applied to validation set $\mathcal{V}^{(k)}$ to compute empirical risk $R(f^{(k)}, \mathcal{V}^{(k)})$. Cycle through all possible partitionings of validation & training sets & compute average generalization error of predictor. Cross-validation approximates expected generalization error

$$\mathbb{E}_{\mathcal{V}}[R(f, \mathcal{V})] \approx \frac{1}{K} \sum_{k=1}^{K} R(f^{(k)}, \mathcal{V}^{(k)}), \tag{46}$$

where $R(f^{(k)}, \mathcal{V}^{(k)})$ is risk (e.g., RMSE) on validation set $\mathcal{V}^{(k)}$ for predictor $f^{(k)}$. Approximation has 2 sources: 1st, due to finite training set, which results in not best possible $f^{(k)}$; & 2nd, due to finite validation set, which results in an inaccurate estimation of risk $R(f^{(k)}, \mathcal{V}^{(k)})$. A potential disadvantage of K-fold cross-validation is computational cost of training model K times, which can be burdensome if training cost is computationally expensive. In practice, often not sufficient to look at direct parameters alone. E.g., need to explore multiple complexity parameters (e.g., multiple regularization parameters), which may not be direct parameters of model. Evaluating quality of model, depending on these hyperparameters, may result in a number of training runs that is exponential in number of model parameters. One can use nested cross-validation (Sect. 8.6.1) to search for good hyperparameters.

However, cross-validation is an *embarrassingly parallel* problem, i.e., little effort is needed to separate problem into a number of parallel tasks. Given sufficient computing resources (e.g., cloud computing, server farms), cross-validation does not require longer than a single performance assessment.

In this sect, saw: empirical risk minimization is based on following concepts: hypothesis class of functions, loss function & regularization. In Sect. 8.3, see effect of using a probability distribution to replace idea of loss functions & regularization.

- 8.2.5. Further Reading. Due to fact: original development of empirical risk minimization (Vapnik, 1998) was couched in heavily theoretical language, many of subsequent developments have been theoretical. Area of study is called *statistical learning theory* (Vapnik, 1999; Evgeniou et al., 2000; Hastie et al., 2001; von Luxburg & Schölkopf, 2011). A recent ML textbook that builds on theoretical foundations & develops efficient learning algorithms is Shalev-Shwartz & Ben-David (2014).

  Concept of regularization has its roots in solution of ill-posed inverse problems (Neumaier, 1998). Approach presented here is called *Tikhonov regularization*, & there is a closely related constrained version called *Ivanov regularization*. Tikhonov regularization has deep relationships to bias-variance trade-off & feature selection (Bühlmann & Van De Geer, 2011). An alternative to cross-validation: bootstrap & jackknife (Efron & Tibshirani, 1993; Davidson & Hinkley, 1997; Hall, 1992).

  Thinking about empirical risk minimization (Sect. 8.2) as "probability free" is incorrect. There is an underlying unknown probability distribution $p(\mathbf{x}, y)$ that governs data generation. However, approach of empirical risk minimization is agnostic to that choice of distribution. This is in contrast to standard statistical approaches that explicitly require knowledge of $p(\mathbf{x}, y)$. Furthermore, since distribution is a joint distribution on both examples $\mathbf{x}$ & labels y, labels can be non-deterministic. In contrast to standard statistics, do not need to specify noise distribution for labels y.

- 8.3. Parameter Estimation. See also [ABT18]. In Sect. 8.2, did not explicitly model our problem using probability distributions. In this sect, see how to use probability distributions to model our uncertainty due to observation process & our uncertainty in parameters of our predictors. In Sect. 8.3.1, introduce likelihood, which is analogous to concept of loss functions (Sect. 8.2.2) in empirical risk minimization. Concept of priors (Sect. 8.3.2) is analogous to concept of regularization (Sect. 8.2.3).

  - 8.3.1. Maximum Likelihood Estimation. Idea behind *maximum likelihood estimation* (MLE): define a function of parameters that enables us to find a model that fits data well. Estimation problem is focused on *likelihood* function, or more precisely its negative logarithm. For data represented by a random variable $\mathbf{x}$ & for a family of probability densities $p(\mathbf{x}|\boldsymbol{\theta})$ parameterized by $\boldsymbol{\theta}$, *negative log-likelihood* is given by

$$\mathcal{L}_{\mathbf{x}}(\boldsymbol{\theta}) = -\log p(\mathbf{x}|\boldsymbol{\theta}). \tag{47}$$

    Notation $\mathcal{L}_{\mathbf{x}}(\boldsymbol{\theta})$ emphasizes fact: parameter $\boldsymbol{\theta}$ is varying & data $\mathbf{x}$ is fixed. Very often drop reference to $\mathbf{x}$ when writing negative log-likelihood, as it is really a function of $\boldsymbol{\theta}$, & write it as $\mathcal{L}(\boldsymbol{\theta})$ when random variable representing uncertainty in data is clear from context.

    Interpret what probability density $p(\mathbf{x}|\boldsymbol{\theta})$ is modeling for a fixed value of $\boldsymbol{\theta}$. It is a distribution that models uncertainty of data. I.e., once have chosen type of function we want as a predictor, likelihood provides probability of observing data $\mathbf{x}$. In a complementary view, if consider data to be fixed (because it has been observed), & vary parameters $\boldsymbol{\theta}$, what does $\mathcal{L}(\boldsymbol{\theta})$ tell us? It tells us how likely a particular setting of $\boldsymbol{\theta}$ is for observations $\mathbf{x}$. Based on this 2nd view, maximum likelihood estimator gives us most likely parameter $\boldsymbol{\theta}$ for set of data.

    Consider supervised learning setting, where obtain pairs $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$ with $\mathbf{x}_n \in \mathbb{R}^D$ & labels $y_n \in \mathbb{R}$. Interested in constructing a predictor that takes a feature vector $\mathbf{x}_n$ as input & produces a prediction y_n (or sth close to it), i.e., given a vector $\mathbf{x}_n$ we want probability distribution of label y_n. I.e., specify conditional probability distribution of labels given examples for particular parameter settings $\boldsymbol{\theta}$.

Example 5. *1st example often used: specify: conditional probability of labels given examples is a Gaussian distribution. I.e., assume: can explain our observation uncertainty by independent Gaussian noise (refer to Sect. 6.5) with zero mean, $\varepsilon_n \sim \mathcal{N}(0, \sigma^2)$. Further assume: linear model $\mathbf{x}_n^\top \boldsymbol{\theta}$ is used for prediction. I.e., specify a Gaussian likelihood for each example label pair $\mathbf{x}_n, y_n$,*

$$p(y_n|\mathbf{x}_n, \boldsymbol{\theta}) = \mathcal{N}(y_n|\mathbf{x}_n^\top \boldsymbol{\theta}, \sigma^2). \tag{48}$$

An illustration of a Gaussian likelihood for a given parameter $\boldsymbol{\theta}$ is shown in Fig. 8.3. *Sect. 9.2: how to explicitly expand preceding expression out in terms of Gaussian distribution.*

Assume: set of examples $(x_1, y_1), \ldots, (x_N, y_N)$ are *independent & identically distributed* (i.i.d.). Word "independent" (Sect. 6.4.5) implies: likelihood of whole dataset $\mathcal{Y} = \{y_1, \ldots, y_N\}$ & $\mathcal{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ factorizes into a product of likelihoods of each individual example (8.16)

$$p(\mathcal{Y}|\mathcal{X}, \boldsymbol{\theta}) = \prod_{n=1}^{N} p(y_n|\mathbf{x}_n, \boldsymbol{\theta}), \tag{49}$$

where $p(y_n|\mathbf{x}_n, \boldsymbol{\theta})$ is a particular distribution (which was Gaussian in Example 8.4). Expression "identically distributed" means: each term in product (8.16) is of same distribution, & all of them share same parameters. Often easier from an optimization viewpoint to compute functions that can be decomposed into sums of simpler functions. Hence, in ML, often consider negative log-likelihood (8.17)

$$\mathcal{L}(\boldsymbol{\theta}) = -\log p(\mathcal{Y}|\mathcal{X}, \boldsymbol{\theta}) = -\sum_{n=1}^{N} \log p(y_n|\mathbf{x}_n, \boldsymbol{\theta}). \tag{50}$$

While it is tempting to interpret fact: $\boldsymbol{\theta}$ is on right of conditioning in $p(y_n|\mathbf{x}_n, \boldsymbol{\theta})$ (8.15), & hence should be interpreted as observed & fixed, this interpretation is incorrect. Negative log-likelihood $\mathcal{L}(\boldsymbol{\theta})$ is a function of $\boldsymbol{\theta}$. Therefore, to find a good parameter vector $\boldsymbol{\theta}$ that explains data $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)$ well, minimize negative log-likelihood $\mathcal{L}(\boldsymbol{\theta})$ w.r.t. $\boldsymbol{\theta}$.

Remark 5. *Negative sign in (8.17) is a historical artifact that is due to convention that we want to maximize likelihood, but numerical optimization literature tends to study minimization of functions.*

Example 6. *Continuing on our example of Gaussian likelihoods (8.15), negative log-likelihood can be rewritten as*

$$\mathcal{L}(\boldsymbol{\theta}) = \cdots = \frac{1}{2\sigma^2} \sum_{n=1}^{N} (y_n - \mathbf{x}_n^\top \boldsymbol{\theta})^2 - \sum_{n=1}^{N} \log \frac{1}{\sqrt{2\pi\sigma^2}}. \tag{51}$$

As σ is given, 2nd term in the last formula is constant, & minimizing $\mathcal{L}(\boldsymbol{\theta})$ corresponds to solving least-squares problem (cf. (8.8)) expressed in 1st term.

Turn out: for Gaussian likelihoods resulting optimization problem corresponding to maximum likelihood estimation has a closed-form solution. See more details on this in Chap. 9. Fig. 8.5: For given data, maximum likelihood estimate of parameters results in black diagonal line. Orange square shows value of maximum likelihood prediction at $x = 60$. shows a regression dataset & function that is induced by maximum-likelihood parameters. Maximum likelihood estimation may suffer from overfitting (Sect. 8.3.3), analogous to unregularized empirical risk minimization (Sect. 9.2.3). For other likelihood functions, i..e, if model our noise with non-Gaussian distributions, maximum likelihood estimation may not have a closed-form analytic solution. In this case, resort to numerical optimization methods discussed in Chap. 7.

∗ 8.3.2. Maximum A Posteriori Estimation. If have prior knowledge about distribution of parameters $\boldsymbol{\theta}$, can multiply an additional term to likelihood. This additional term is a prior probability distribution on parameters $p(\boldsymbol{\theta})$. For a given prior, after observing some data $\mathbf{x}$, how should we update distribution of $\boldsymbol{\theta}$? I.e., how should we represent fact: have more specific knowledge of $\boldsymbol{\theta}$ after observing data $\mathbf{x}$? Bayes' theorem (Sect. 6.3) gives us a principled tool to update our probability distributions of random variables. It allows us to compute a *posterior* distribution $p(\boldsymbol{\theta}|\mathbf{x})$ (more specific knowledge) on parameters $\boldsymbol{\theta}$ from general *prior* statements (prior distribution) $p(\boldsymbol{\theta})$ & function $p(\mathbf{x}|\boldsymbol{\theta})$ that links parameters $\boldsymbol{\theta}$ & observed data $\mathbf{x}$ (called *likelihood*):

$$p(\boldsymbol{\theta}|\mathbf{x}) = \frac{p(\mathbf{x}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathbf{x})}. \tag{52}$$

Recall: interested in finding parameter $\boldsymbol{\theta}$ that maximizes posterior. Since distribution $p(\mathbf{x})$ does not depend on $\boldsymbol{\theta}$, can ignore value of denominator for optimization & obtain

$$p(\boldsymbol{\theta}|\mathbf{x}) \asymp p(\mathbf{x}|\boldsymbol{\theta})p(\boldsymbol{\theta}). \tag{53}$$

Preceding proportion relation hides density of data $p(\mathbf{x})$, which may be difficult to estimate. Instead of estimating minimum of negative log-likelihood, now estimate minimum of negative log-posterior, which is referred to as *maximum a posteriori estimation (MAP estimation)*. An illustration of effect of adding a zero-mean Gaussian prior is shown in Fig. 8.6: Comparing predictions with maximum likelihood estimate & MAP estimate at $x = 60$. Prior biases slope to be less steep & intercept to be closer to 0. In this example, bias that moves intercept closer to 0 actually increases slope.

Example 7. *In addition to assumption of Gaussian likelihood in previous example, assume: parameter vector is distributed as a multivariate Gaussian with zero mean, i.e., $p(\boldsymbol{\theta}) = \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma})$, where $\boldsymbol{\Sigma}$: covariance matrix (Sect. 6.5). Note: conjugate prior of a Gaussian is also a Gaussian (Sect. 6.6.1), & therefore expect posterior distribution to also be a Gaussian. See details of maximum a posteriori estimation in Chap. 9.*

Idea of including prior knowledge about where good parameters lie is widespread in ML. An alternative view, which saw in Sect. 8.2.3, is idea of regularization, which introduces an additional term that biases resulting parameters to be close to origin. Maximum a posteriori estimation can be considered to bridge non-probabilistic & probabilistic worlds as it explicitly acknowledges need for a prior distribution but it still only produces a point estimate of parameters.

Remark 6. *Maximum likelihood estimate $\boldsymbol{\theta}_{\mathrm{ML}}$ possesses following properties (Lehmann & Casella, 1998; Efron & Hastie, 2016):*

- *Asymptotic consistency: MLE converges to true value in limit of infinitely many observations, plus a random error that is approximately normal.*
- *Size of samples necessary to achieve these properties can be quite large.*
- *Error's variance decays in $\frac{1}{N}$, where N: number of data points.*
- *Especially, in "small" data regime, maximum likelihood estimation can lead to* overfitting.

Principle of maximum likelihood estimation (& maximum a posteriori estimation) uses probabilistic modeling to reason abut uncertainty in data & model parameters. However, have not yet taken probabilistic modeling to its full extent. In this sect, resulting training procedure still produces a point estimate of predictor, i.e., training returns 1 single set of parameter values that represent best predictor. In Sect. 8.4, will take view: parameter values should also be treated as random variables, & instead of estimating "best" values of that distribution, use full parameter distribution when making predictions.

∗ 8.3.3. Model Fitting. Consider setting where given a dataset, & interested in fitting a parameterized model to data. When talk about "fitting", typically mean optimizing/learning model parameters so that they minimize some loss function, e.g., negative log-likelihood. With maximum likelihood (Sect. 8.3.1) & maximum a posteriori estimation (Sect. 8.3.2), already discussed 2 commonly used algorithms for model fitting.

Parametrization of model defines a model class $M_{\boldsymbol{\theta}}$ with which we can operate. E.g., in a linear regression setting, may define relationship between inputs x & (noise-free) observations y to be $y = ax + b$, where $\boldsymbol{\theta} \coloneqq \{a, b\}$: model parameters. In this case, model parameters $\boldsymbol{\theta}$ describe family of affine functions, i.e., straight lines with slope a, which are offset from 0 by b. Assume: data comes from a model M^*, which is unknown to us. For a given training dataset, optimize $\boldsymbol{\theta}$ so that $M_{\boldsymbol{\theta}}$ is as close as possible to M^*, where "closeness" is defined by objective function we optimize (e.g., squared loss on training data). Fig. 8.7: Model fitting. In a parametrized class $M_{\boldsymbol{\theta}}$ of models, optimize model parameters $\boldsymbol{\theta}$ to minimize distance to true (unknown) model M^*. illustrates a setting where have a small model class (indicated by circle $M_{\boldsymbol{\theta}}$), & data generation model M^* lies outside set of considered models. Begin our parameter search at $M_{\boldsymbol{\theta}_0}$. After optimization, i.e., when obtain best possible parameters $\boldsymbol{\theta}^*$, distinguish 3 different cases: (i) overfitting, (ii) underfitting, & (iii) fitting well. Give a high-level intuition of what these 3 concepts mean.

> 1 way to detect overfitting in practice: observe: model has low training risk but high test risk during cross validation (Sect. 8.2.4).

Roughly speaking, *overfitting* refers to situation where parametrized model class is too rich to model dataset generated by M^*, i.e., $M_{\boldsymbol{\theta}}$ could model much more complicated datasets. E.g., if dataset was generated by a linear function, & define $M_{\boldsymbol{\theta}}$ to be class of 7th-order polynomials, could model not only linear functions, but also polynomials of degree 2, 3, etc. Models that overfit typically have a large number of parameters. An observation often make: overly flexible model class $M_{\boldsymbol{\theta}}$ uses all its modeling power to reduce training error. If training data is noisy, it will therefore find some useful signal in noise itself. This will cause enormous problems when predict away from training data. Fig. 8.8(a): Fitting (by maximum likelihood) of different model classes to a regression dataset: (a) Overfitting. (b) Underfitting. (c) Fitting well. gives an example of overfitting in context of regression where model parameters are learned by means of maximum likelihood (Sect. 8.3.1). Discuss overfitting in regression more in Sect. 9.2.2.

When run into *underfitting*, encounter opposite problem where model class $M_{\boldsymbol{\theta}}$ is not rich enough. E.g., if our dataset was generated by a sinusoidal function, but $\boldsymbol{\theta}$ only parametrizes straight lines, best optimization produce will not get us close to true model. However, still optimize parameters & find best straight line that models dataset. Fig. 8.8(b) shows an example of a model that underfits because it is insufficiently flexible. Models that underfit typically have few parameters. 3rd case is when parametrized model class is about right. Then, our model fits well, i.e., it neither overfits nor underfits. I.e., our mean class is just rich enough to describe dataset given. Fig. 8.8(c) shows a model that fits given dataset fairly well. Ideally, this is model we would want to work with since it has good generalization properties.

In practice, often define very rich model classes $M_{\boldsymbol{\theta}}$ with many parameters, e.g. deep neural networks. To mitigate problem of overfitting, can use regularization (Sect. 8.2.3) or priors (Sect. 8.3.2). Discuss how to choose model class in Sect. 8.6.

∗ 8.3.4. Further Reading. When considering probabilistic models, principle of maximum likelihood estimation generalizes idea of least-squares regression for linear models, discussed in detail in Chap. 9. When restricting predictor to have linear form with an additional nonlinear function φ applied to output, i.e.,

$$p(y_n|\mathbf{x}_n, \boldsymbol{\theta}) = \varphi(\boldsymbol{\theta}^\top \mathbf{x}_n), \tag{54}$$

can consider other models for other prediction tasks, e.g. binary classification or modeling count data (McCullagh & Nelder, 1989). An alternative view of this: consider likelihoods that are from exponential family (Sect. 6.6). Class of models, which have linear dependence between parameters & data, & have potentially nonlinear transformation φ (called a *link function*), is referred to as *generalized linear models* (Agresti, 2002, Chap. 4).

Maximum likelihood estimation has a rich history, & was originally proposed by Sir Ronald Fisher in 1930s. Will expand upon idea of a probabilistic model in Sect. 8.4. 1 debate among researchers who use probabilistic models: discussion between

Bayesian & frequentist statistics. As mentioned in Sect. 6.1.1, it boils down to def of probability. Recall from Sect. 6.1: one can consider probability to be a generalization (by allowing uncertainty) of local reasoning (Cheeseman, 1985; Jaynes, 2003). Method of maximum likelihood estimation is frequentist in nature, & interested reader is pointed to Efron & Hastie (2016) for a balanced view of both Bayesian & frequentist statistics.

There are some probabilistic models where maximum likelihood estimation may not be possible. Reader is referred to more advanced statistical textbooks, e.g., Casella and Berger (2002), for approaches, e.g. method of moments, M-estimation, & estimating equations.

◦ 8.4. Probabilistic Modeling & Inference. In ML, frequently concerned with interpretation & analysis of data, e.g., for prediction of future events & decision making. To make this task more tractable (dễ uốn nắn/làm/sai khiến), often build models that describe *generative process* that generates observed data.

E.g., can describe outcome of a coin-flip experiment ("heads" or "tails") in 2 steps. 1st, define a parameter μ, which describes probability of "heads" as parameter of a Bernoulli distribution (Chap. 6); 2nd, can sample an outcome $x \in \{\text{head}, \text{tail}\}$ from Bernoulli distribution $p(x|\mu) = \text{Ber}(\mu)$. Parameter μ gives rise to a specific dataset $\mathcal{X}$ & depends on coin used. Since μ is unknown in advance & can never be observed directly, need mechanisms to learn sth about μ given observed outcomes of coin-flip experiments. In following, discuss how probabilistic modeling can be used for this purpose.

∗ 8.4.1. Probabilistic models.

A probabilistic model is specified by joint distribution of all random variables.

Probabilistic models represent uncertain aspects of an experiment as probability distributions. Benefit of using probabilistic models: they offer a unified & consistent set of tools from probability theory (Chap. 6) for modeling, inference, prediction, & model selection.

In probabilistic modeling, joint distribution $p(\mathbf{x}, \boldsymbol{\theta})$ of observed variables $\mathbf{x}$ & hidden parameters $\boldsymbol{\theta}$ of observed variables $\mathbf{x}$ & hidden parameters $\boldsymbol{\theta}$ is of central importance: It encapsulates information from following:

· Prior & likelihood (product rule, Sect. 6.3).

· Marginal likelihood $p(\mathbf{x})$, which will play an important role in model selection (Sect. 8.6), can be computed by taking joint distribution & integrating out parameters (sum rule, Sect. 6.3).

· Posterior, which can be obtained by dividing joint by marginal likelihood.

Only joint distribution has this property. Therefore, a probabilistic model is specified by joint distribution of all its random variables.

∗ 8.4.2. Bayesian Inference.

Parameter estimation can be phrased as an optimization problem.

A key task in ML: take a model & data to uncover values of model's hidden variables $\boldsymbol{\theta}$ given observed variables $\mathbf{x}$. In Sect. 8.3.1, already discussed 2 ways for estimating model parameters $\boldsymbol{\theta}$ using maximum likelihood or maximum a posteriori estimation. In both cases, obtain a single-best value for $\boldsymbol{\theta}$ so that key algorithmic problem of parameter estimation is solving an optimization problem. Once these point estimates $\boldsymbol{\theta}^*$ are known, use them to make predictions. More specifically, predictive distribution will be $p(\mathbf{x}|\boldsymbol{\theta}^*)$, where use $\boldsymbol{\theta}^*$ in likelihood function.

As discussed in Sect. 6.3, focusing solely on some statistics of posterior distribution (e.g. parameter $\boldsymbol{\theta}^*$ that maximizes posterior) leads to loss of information, which can be critical in a system that uses prediction $p(\mathbf{x}|\boldsymbol{\theta}^*)$ to make decisions. These decision-making systems typically have different objective functions than likelihood, a squared-error loss or a mis-classification error. Therefore, having full posterior distribution around can be extremely useful & leads to more robust decisions. *Bayesian inference* is about finding this posterior distribution (Gelman et al., 2004). For a dataset $\mathcal{X}$, a parameter prior $p(\boldsymbol{\theta})$ & a likelihood function, posterior

$$p(\boldsymbol{\theta}|\mathcal{X}) = \frac{p(\mathcal{X}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathcal{X})}, \; p(\mathcal{X}) = \int p(\mathcal{X}|\boldsymbol{\theta})p(\boldsymbol{\theta})\,\mathrm{d}\boldsymbol{\theta}, \tag{55}$$

is obtained by applying Bayes' theorem. Key idea: exploit Bayes' theorem to invert relationship between parameters $\boldsymbol{\theta}$ & data $\mathcal{X}$ (given by likelihood) to obtain posterior distribution $p(\boldsymbol{\theta}|\mathcal{X})$.

Bayesian inference is about learning distribution of random variables.

Bayesian inference inverts relationship between parameters & data.

Implication of having a posterior distribution on parameters: it can be used to propagate uncertainty from parameters to data. More specifically, with a distribution $p(\boldsymbol{\theta})$ on parameters, our predictions will be (8.23)

$$p(\mathbf{x}) = \int p(\mathbf{x}|\boldsymbol{\theta})p(\boldsymbol{\theta})\,\mathrm{d}\boldsymbol{\theta} = \mathbb{E}_{\boldsymbol{\theta}}[p(\mathbf{x}|\boldsymbol{\theta})], \tag{56}$$

& they no longer depend on model parameters $\boldsymbol{\theta}$, which have been marginalized/integrated out. (8.23) reveals: prediction is an average over all plausible parameter values $\boldsymbol{\theta}$, where plausibility is encapsulated by parameter distribution $p(\boldsymbol{\theta})$.

Having discussed parameter estimation in Sect. 8.3 & Bayesian inference here, compare these 2 approaches to learning. Parameter estimation via maximum likelihood or MAP estimation yields a consistent point estimate $\boldsymbol{\theta}^*$ of parameters, & key computational problem to be solved is optimization. In contrast, Bayesian inference yields a (posterior) distribution, & key computational problem to be solved is integration. Predictions with point estimates are straightforward, whereas predictions in Bayesian framework require solving another integration problem; see (8.23). However, Bayesian inference

gives us a principled way to incorporate prior knowledge, account for side information, & incorporate structural knowledge, all of which is not easily done in context of parameter estimation. Moreover, propagation of parameter uncertainty to prediction can be valuable in decision-making systems for risk assessment & exploration in context of data-efficient learning (Deisenroth et al., 2015; Kamthe and Deisenroth, 2018).

While Bayesian inference is a mathematically principled framework for learning about parameters & making predictions, there are some practical challenges that come with it because of integration problems need to solve; see (8.22) & (8.23). More specifically, if do not choose a conjugate prior on parameters (Sect. 6.6.1), integrals in (8.22) & (8.23) are not analytically tractable, & cannot compute posterior, predictions, or marginal likelihood in closed form. In these cases, need to resort to approximations. Here, can use stochastic approximations, e.g. Markov chain Monte Carlo (MCMC) (Gilks et al., 1996), or deterministic approximations, e.g. Laplace approximation (Bishop, 2006; Barber, 2012; Murphy, 2012), variational inference (Jordan et al., 1999; Blei et al., 2017), or expectation propagation (Minka, 2001a).

Despite these challenges, Bayesian inference has been successfully applied to a variety of problems, including large-scale topic modeling(Hoffman et al., 2013), click-through-rate prediction (Graepel et al., 2010), data-efficient reinforcement learning in control systems (Deisenroth et al., 2015), online ranking systems (Herbrich et al., 2007), & large-scale recommender systems. There are generic tools, e.g. Bayesian optimization (Brochu et al., 2009; Snoek et al., 2012; Shahriari et al., 2016), that are very useful ingredients for an efficient search of meta parameters of models or algorithms.

Remark 7. *In ML literature, there can be a somewhat arbitrary separation between (random) "variables" & "parameters". While parameters are estimated (e.g., via maximum likelihood), variables are usually marginalized out. In this book, not so strict with this separation because, in principle, can replace a prior on any parameter & integrate it out, which would then turn parameter into a random variable according to aforementioned separation.*

∗ 8.4.3. Latent-Variable Models. In practice, sometimes useful to have additional *latent variables* $\mathbf{z}$ (besides model parameters $\boldsymbol{\theta}$) as part of model (Moustaki et al., 2015). These latent variables are different from model parameters $\boldsymbol{\theta}$ as they do not parameterize model explicitly. Latent variables may describe data-generating process, thereby contributing to interpretability of model. They also often simplify structure of model & allow us to define simper & richer model structures. Simplification of model structure often goes hand in hand with a smaller number of model parameters (Paquet, 2008; Murphy, 2012). Learning in latent-variable models (at least via maximum likelihood) can be done in a principled way using expectation maximization (EM) algorithm (Dempster et al., 1977; Bishop, 2006). Examples, where such latent variables (biến tiềm ẩn) are helpful, are principle component analysis for dimensionality reduction (Chap. 10), Gaussian mixture models for density estimation (Chap. 11), hidden Markov models (Maybeck, 1979) or dynamical systems (Ghahramani and Roweis, 1999; Ljung, 1999) for time-series modeling, & meta learning & task generalization (Hausman et al., 2018; Sæmundsson et al., 2018). Although introduction of these latent variables may make model structure & generative process easier, learning in latent-variable models is generally hard, as see in Chap. 11.

Since latent-variable models also allow us to define process that generates data from parameters, have a look at this generative process. Denoting data by $\mathbf{x}$, model parameters by $\boldsymbol{\theta}$ & latent variables by $\mathbf{z}$, obtain conditional distribution $p(\mathbf{x}|\mathbf{z},\boldsymbol{\theta})$ that allows us to generate data for any model parameters & latent variables. Given that $\mathbf{z}$ are latent variables, place a prior $p(\underline{\mathrm{z}})$ on them.

As models discussed previously, models with latent variables can be used for parameter learning & inference within frameworks discussed in Sects. 8.3 & 8.4.2. To facilitate learning (e.g., by means of maximum likelihood estimation or Bayesian inference), follow a 2-step procedure. 1st, compute likelihood $p(\mathbf{x}|\boldsymbol{\theta})$ of model, which does not depend on latent variables. 2nd, use this likelihood for parameter estimation or Bayesian inference, where use exactly same expressions as in Sects. 8.3 & 8.4.2, resp.

Since likelihood function $p(\mathbf{x}|\boldsymbol{\theta})$ is predictive distribution of data given model parameters, need to marginalize out latent variables so that (8.25)

$$p(\mathbf{x}|\boldsymbol{\theta}) = \int p(\mathbf{x}|\mathbf{z},\boldsymbol{\theta})p(\mathbf{z})\,\mathrm{d}\mathbf{z}, \tag{57}$$

where $p(\mathbf{x}|\mathbf{z},\boldsymbol{\theta})$ is given in (8.24) & $p(\mathbf{z})$ is prior on latent variables. Note: likelihood must not depend on latent variables $\mathbf{z}$, but it is only a function of data $\mathbf{x}$ & model parameters $\boldsymbol{\theta}$.

Likelihood is a function of data & model parameters, but is independent of latent variables.

Likelihood in (8.25) directly allows for parameter estimation via maximum likelihood. MAP estimation is also straightforward with an additional prior on model parameters $\boldsymbol{\theta}$ as discussed in Sect. 8.3.2. Moreover, with likelihood (8.25) Bayesian inference (Sect. 8.4.2) in a latent-variable model works in usual way: Place a prior $p(\boldsymbol{\theta})$ on model parameters & use Bayes' theorem to obtain a posterior distribution (8.26)

$$p(\boldsymbol{\theta}|\mathcal{X}) = \frac{p(\mathcal{X}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathcal{X})} \tag{58}$$

over model parameters given a dataset $\mathcal{X}$. Posterior in (8.26) can be used for predictions within a Bayesian inference framework; see (8.23).

1 challenge have in this latent-variable model: likelihood $p(\mathcal{X}|\boldsymbol{\theta})$ requires marginalization of latent variables according to (8.25). Except when choose a conjugate prior $p(\mathbf{z})$ for $p(\mathbf{x}|\mathbf{z},\boldsymbol{\theta})$, marginalization in (8.25) is not analytically tractable, & need to resort to approximations (Bishop, 2006; Paquet, 2008; Murphy, 2012; Moustaki et al., 2015).

Similar to parameter posterior (8.26), can compute a posterior on latent variables according to

$$p(\mathbf{z}|\mathcal{X}) = \frac{p(\mathcal{X}|\mathbf{z})p(\mathbf{z})}{p(\mathcal{X})},\ p(\mathcal{X}|\mathbf{z}) = \int p(\mathcal{X}|\mathbf{z},\boldsymbol{\theta})p(\boldsymbol{\theta})\,\mathrm{d}\boldsymbol{\theta}, \tag{59}$$

where $p(\mathbf{z})$: prior on latent variables & $p(\mathcal{X}|\mathbf{z})$ requires us to integrate out model parameters $\boldsymbol{\theta}$.
Given difficulty of solving integrals analytically, clear: marginalizing out both latent variables & model parameters at same time is not possible in general (Bishop, 2006; Murphy, 2012). A quantity that is easier to compute is posterior distribution on latent variables, but conditioned on model parameters, i.e.,

$$p(\mathbf{z}|\mathcal{X}, \boldsymbol{\theta}) = \frac{p(\mathcal{X}|\mathbf{z}, \boldsymbol{\theta})p(\mathbf{z})}{p(\mathcal{X}|\boldsymbol{\theta})}, \tag{60}$$

where $p(\mathbf{z})$: prior on latent variables & $p(\mathcal{X}|\mathbf{z}, \boldsymbol{\theta})$ is given i (8.24).
In Chaps. 10–11, derive likelihood functions for PCA & Gaussian mixture models, resp. Moreover, compute posterior distributions (8.28) on latent variables for both PCA & Gaussian mixture models.

Remark 8. *In following chaps, may not be drawing such a clear distinction between latent variables* $\mathbf{z}$ *& uncertain model parameters* $\boldsymbol{\theta}$ *& call model parameters "latent" or "hidden" as well because they are unobserved. In Chaps. 10–11, where use latent variables* $\mathbf{z}$*, will pay attention to difference as will have 2 different types of hidden variables: model parameters* $\boldsymbol{\theta}$ *& latent variables* $\mathbf{z}$*.*

Can exploit fact: all elements of a probabilistic model are random variables to define a unified language for representing them. In Sect. 8.5, see a concise graphical language for representing structure of probabilistic models. Use this graphical language to describe probabilistic models in subsequent chaps.

* 8.4.4. Further Reading. Probabilistic models in ML (Bishop, 2006; Barber, 2012; Murphy, 2012) provide a way for users to capture uncertainty about data & predictive models in a principled fashion. Ghahramani (2015) presents a short review of probabilistic models in ML. Given a probabilistic model, may be lucky enough to be able to compute parameters of interest analytically. However, in general, analytic solutions are rare, & computational methods e.g. sampling (Gilks et al., 1996; Brooks et al., 2011) & variational inference (Jordan et al., 1999; Blei et al., 2017) are used. Moustaki et al. (2015) and Paquet (2008) provide a good overview of Bayesian inference in latent-variable models.
  In recent years, several programming languages have been proposed that aim to treat variables defined in software as random variables corresponding to probability distributions. Objective: be able to write complex functions of probability distributions, while under hood compiler automatically takes care of rules of Bayesian inference. This rapidly changing field is called *probabilistic programming*.

◦ 8.5. Directed Graphical Models. In this sect, introduce a graphical language for specifying a probabilistic model, called *directed graphical model*. It provides a compact & succinct way to specify probabilistic models, & allows reader to visually parse dependencies between random variables. A graphical model visually captures way in which joint distribution over all random variables can be decomposed into a product of factors depending only on a subset of these variables. In Sect. 8.4, we identified joint distribution of a probabilistic model as key quantity of interest because it comprises information about prior, likelihood, & posterior. However, joint distribution by itself can be quite complicated, & it does not tell us anything about structural properties of probabilistic model. E..g, joint distribution $p(a, b, c)$ does not tell us anything abut independence relations. This is point where graphical models come into play. This sect relies on concepts of independence & conditional independence, as described in Sect. 6.4.5.

Directed graphical models are also known as Bayesian networks.

It a *graphical model*, nodes are random variables. In Fig. 8.9: Examples of directed graphical models: (a) Fully connected. (b) Not fully connected(a), nodes represent random variables a, b, c. Edges represent probabilistic relations between variables, e.g., conditional probabilities.

Remark 9. *Not every distribution can be represented in a particular choice of graphical model. A discussion of this can be found in Bishop (2006).*

Probabilistic graphical models have some convenient properties:

* they are a simple way to visualize structure of a probabilistic model.
* Inspection of graph alone gives us insight into properties, e.g., conditional independence.
* Complex computations for inference & learning in statistical models can be expressed in terms of graphical manipulations.

* 8.5.1. Graph Semantics. (Ngữ nghĩa đồ thị) *Directed graphical models/Bayesian networks* are a method for representing conditional dependencies in a probabilistic model. They provide a visual description of conditional probabilities, hence, providing a simple language for describing complex interdependence. Modular description also entails computational simplification. Directed links (arrows) between 2 nodes (random variables) indicate conditional probabilities. E.g., arrow between a, b in Fig. 8.9(a) gives conditional probability $p(b|a)$ of b given a.

  With additional assumptions, arrows can be used to indicate causal relationships (Pearl, 2009).

  Directed graphical models can be derived from joint distributions if know sth about their factorization.

  Example 8. *Consider joint distribution* $p(a, b, c) = p(c|a, b)p(b|a)p(a)$ *(8.29) of 3 random variables* a, b, c*. Factorization of joint distribution in (8.29) tells us sth about relationship between random variables: c depends directly on a & b, b depends directly on a, a depends neither on b nor on c.*
  For factorization in (8.29), obtain directed graphical model in Fig. 8.9(a).

  In general, can construct corresponding directed graphical model from a factorized joint distribution as follows:

  1. Create a node $\forall$ random variables.

2. For each conditional distribution, add a directed link (arrow) to graph from nodes corresponding to variables on which distribution is conditioned.

Graph layout depends on factorization of joint distribution. Graph layout depends on choice of factorization of joint distribution.

Discussed how to get from a known factorization of joint distribution to corresponding directed graphical model. Now, will do exactly opposite & describe how to extract joint distribution of a set of random variables from a given graphical model.

Example 9. *Looking at graphical model in* Fig. 8.9(b)*, exploit 2 properties: (i) Joint distribution* $p(x_1, \ldots, x_5)$ *we seek is product of a set of conditionals, one for each node in graph. In this particular example, need 5 conditionals. (ii) Each conditional depends only on parents of corresponding node in graph. E.g.,* x_4 *will be conditioned on* x_2*.*
These 2 properties yield desired factorization of joint distribution $p(x_1, x_2, x_3, x_4, x_5) = p(x_1)p(x_5)p(x_2|x_5)p(x_3|x_1, x_2)p(x_4|x_2)$*.*

In general, joint distribution $p(\mathbf{x}) = p(x_1, \ldots, x_K)$ is given as

$$p(\mathbf{x}) = \prod_{k=1}^{K} p(x_k|\mathsf{Pa}_k), \tag{61}$$

where Pa_k means "parent nodes of x_k". Parent nodes of x_k are nodes that have arrows pointing to x_k.
Conclude this subsection with a concrete example of coin-flip experiment. Consider a Bernoulli experiment (Example 6.8) where probability that outcome x of this experiment is "heads" is

$$p(x|\mu) = \text{Ber}(\mu). \tag{62}$$

Now repeat this experiment N times & observe outcomes $x_1, \ldots, x_N$ so that obtain joint distribution

$$p(x_1, \ldots, x_N|\mu) = \prod_{n=1}^{N} p(x_n|\mu). \tag{63}$$

Expression on RHS is a product of Bernoulli distributions on each individual outcome because experiments are independent. Recall from Sect. 6.4.5: statistical independence means: distribution factorizes. To write graphical model down for this setting, make distinction between unobserved/latent variables & observed variables. Graphically, observed variables are denoted by shaded nodes so that obtain graphical model in Fig. 8.10(a): Graphical models for a repeated Bernoulli experiment: (a) Versions with x_n explicit. (b) Version with plate notation. (c) Hyperparameters α, β on latent μ. See: single parameter μ is same $\forall x_n$, $n = 1, \ldots, N$ as outcomes x_n are identically distributed. A more compact, but equivalent, graphical model for this setting is given in Fig. 8.10(b), where use *plate* notation. Plate (box) repeats everything inside (in this case, observations x_n) N times. Therefore, both graphical models are equivalent, but plate notation is more compact. Graphical models immediately allow us to place a hyperprior on μ. A *hyperprior* is a 2nd layer of prior distributions on parameters of 1st layer of priors. Fig. 8.10(c) places a Beta(α, β) prior on latent variable μ. If treat α, β as deterministic parameters, i.e., not random variables, omit circle around it.

∗ 8.5.2. Conditional Independence & d-Separation. Directed graphical models allow us to find conditional independence (Sect. 6.4.5) relationship properties of joint distribution only by looking at graph. A concept called *d-separation* (Pearl, 1988) is key to this.
Consider a general directed graph in which $\mathcal{A}, \mathcal{B}, \mathcal{C}$ are arbitrary nonintersecting sets of nodes (whose union may be smaller than complete set of nodes in graph). Wish to ascertain whether a particular conditional independence statement, "$\mathcal{A}$ is conditionally independent of $\mathcal{B}$ given $\mathcal{C}$", denoted by $\mathcal{A} \perp\!\!\!\perp \mathcal{B}|\mathcal{C}$, is implied by a given directed acyclic graph. To do so, consider all possible trails (paths that ignore direction of arrows) from any node in $\mathcal{A}$ to any nodes in $\mathcal{B}$. Any such path is said to be *blocked* if it includes any node s.t. either of following are true:

· Arrows on path meet either head to tail or tail to tail at node, & node is in set $\mathcal{C}$.

· Arrows meet head to head at node, & neither node nor any of its descendants is in set $\mathcal{C}$.

If all paths are blocked, then $\mathcal{A}$ is said to be *d-separated* from $\mathcal{B}$ by $\mathcal{C}$, & joint distribution over all of variables in graph will satisfy $\mathcal{A} \perp\!\!\!\perp \mathcal{B}|\mathcal{C}$.

Example 10 (Conditional Independence)**.** *Consider graphical model in* Fig. 8.11: D-separation example*. Visual inspection gives us* $b \perp\!\!\!\perp d|a, c$*,* $a \perp\!\!\!\perp c|b$*,* $b \not\perp\!\!\!\perp d|c$*,* $a \not\perp\!\!\!\perp c|b, e$*.*

Directed graphical models allow a compact representation of probabilistic models, & will see examples of directed graphical models in Chaps. 9–11. Representation, along with concept of conditional independence, allows us to factorize respective probabilistic models into expressions that are easier to optimize.
Graphical representation of probabilistic model allows us to visually see impact of design choices we have made on structure of model. Often need to make high-level assumptions about structure of model. These modeling assumptions (hyperparameters) affect prediction performance, but cannot be selected directly using approaches have seen so far. Discuss different ways to choose structure in Sect. 8.6.

∗ 8.5.3. Further Reading. An introduction to probabilistic graphical models can be found in Bishop (2006, Chap. 8), & an extensive description of different applications & corresponding algorithmic implications can be found in book by Koller and Friedman (2009). There are 3 main types of probabilistic graphical models

- *Directed graphical models (Bayesian networks)*; see Fig. 8.12(a): 3 types of graphical models: (a) Directed graphical models (Bayesian networks); (b) Undirected graphical models (Markov random fields); (c) Factor graphs.
- *Undirected graphical models (Markov random fields)*; see Fig. 8.12(a)
- *Factor graphs*; see Fig. 8.12(c)

  Graphical models allow for graph-based algorithms for inference & learning, e.g., via local message passing. Application range from ranking in online games (Herbrich et al., 2007) & computer vision (e.g., image segmentation, semantic labeling, image denoising, image restoration (Kittler and Föglein, 1984; Sucar and Gillies, 1994; Shotton et al., 2006; Szeliski et al., 2008)) to coding theory (McEliece et al., 1998), solving linear equation systems (Shental et al., 2008), & iterative Bayesian state estimation in signal processing (Bickson et al., 2007; Deisenroth and Mohamed, 2012).
  1 topic particularly important in real applications that we do not discuss in this book: idea of structured prediction (Bakir et al., 2007; Nowozin et al., 2014), which allows ML models to tackle predictions that are structured, e.g. sequences, trees, & graphs. Popularity of neural network models has allowed more flexible probabilistic models to be used, resulting in many useful applications of structured models (Goodfellow et al., 2016, Chap. 16). In recent years, there has been a renewed interest in graphical models due to their applications to causal inference (Pearl, 2009; Imbens and Rubin, 2015; Peters et al., 2017; Rosenbaum, 2017).

- 8.6. Model Selection.

- 9. Linear Regression.
  - 9.1. Problem Formulation.
  - 9.2. Parameter Estimation.
  - 9.3. Bayesian Linear Regression.
  - 9.4. Maximum Likelihood as Orthogonal Projection.
  - 9.5. Further Reading.
- 10. Dimensionality Reduction with Principal Component Analysis.
  - 10.1. Problem Setting.
  - 10.2. Maximum Variance Perspective.
  - 10.3. Projection Perspective.
  - 10.4. Eigenvector Computation & Low-Rank Approximations.
  - 10.5. PCA in High Dimensions.
  - 10.6. Key Steps of PCA in Practice.
  - 10.7. Latent Variable Perspective.
  - 10.8. Further Reading.
- 11. Density Estimation with Gaussian Mixture Models.
  - 11.1. Gaussian Mixture Model.
  - 11.2. Parameter Learning via Maximum Likelihood.
  - 11.3. EM Algorithm.
  - 11.4. Latent-Variable Perspective.
  - 11.5. Further Reading.
- 12. Classification with Support Vector Machines.
  - 12.1. Separating Hyperplanes.
  - 12.2. Primal Support Vector Machine.
  - 12.3. Dual Support Vector Machine.
  - 12.4. Kernels.
  - 12.5. Numerical Solution.
  - 12.6. Further Reading.

2.3 Thang Nguyen, Dung Nguyen, Kha Pham, Truyen Tran. MP-PINN: A Multi-Phase Physics-Informed Neural Network for Epidemic Forecasting

Abstract. Forecasting temporal processes e.g. virus spreading in epidemics often requires more than just observed time-series data, especially at beginning of a wave when data is limited. Traditional methods employ mechanistic models like SIR family, which make strong assumptions about underlying spreading process, often represented as a small set of compact differential equations. Data-driven methods e.g. deep neural networks make no such assumptions & can capture generative process in more detail, but fail in long-term forecasting due to data limitations. Propose a new hybrid method called MP-PINN (Multi-Phase Physics-Informed Neural Network) to overcome limitations of these 2 major approaches. MP-PINN instills spreading mechanism into a neural network, enabling mechanism to update in phases over time, reflecting dynamics of epidemics due to policy interventions. Experiments on COVID-19 waves demonstrate: MP-PINN achieves superior performance over pure data-driven or model-driven approaches for both short-term & long-term forecasting.

Keywords. Time-series forecasting; Physics-Informed Neural Network; Epidemiological Models; COVID-19.

- 1. Introduction. COVID-19 pandemic has claimed > 7 million lives for just 4 years. Pause for a moment & consider a hard *counterfactual* question: Could majority of these lives have been saved if we had predicted spread better at onset of pandemic & acted more effectively? World did all it could: modeling, forecasting, implementing lockdowns, developing vaccines, & much more. In absence of proper understanding of viruses' nature & with limited data available when a wave just started, epidemiologists had to make assumptions in *model-driven* methods, e.g. those in mechanistic SIR family [10] or in detailed agent-based simulations [11]. When some data became available, e.g., after a month, *data-driven* methods, as preferred by DS community, could be employed to detect trends in time-series [25,21]. A key open challenge: complex interplay of evolving interventions, human factors & technological advances driving epidemic waves [1].

  If anything, extremely high death toll has profoundly demonstrated 1 thing: Have failed to predict spread of COVID-19 virus variants. Fig. 1: A representative case of forecasting COVID-19 at 35 days of wave. Our hybrid multiphase method MP-PINN strikes a balance between model-driven & data-driven approaches, & hence is more accurate in both short/long-term forecasting. clearly illustrates this failure. As seen, model-driven methods e.g. mechanistic SIR models capture overall shape of wave but are inadequate in reflecting current data & changing reality on a daily basis. This might be due to rigid & strong assumptions made at modeling time. Data-driven methods, e.g. those using deep neural networks, fit new evidence better but fail to capture long-term underlying mechanisms. Clearly, a better approach is needed to (a) capture both short-term & long-term processes [22,18], & (b) dynamically calibrate models in face of new evidence [19].

  To this end, propose MP-PINN (which stands for Multi-Phase Physics-Informed Neural Network) to overcome these limitations. MP-PINN employs a recent powerful approach known as Physics-Informed Neural Network (PINN), which trains a neural network to agree with both empirical data & epidemiological models. However, PINNs alone are not sufficient to reflect reality: Must account for complex interplay of evolving factors driving epidemic waves, e.g. changing regulations, emerging information, & shifting public sentiment, all of which influence pandemic's trajectory. This is where MP-PINN comes in: Instead of assuming a single set of parameters fore entire wave, allow model to vary across *multiple distinct phases*, each represented by a set of SIR parameters. This brings adaptibility into MP-PINN.

  Demonstrate MP-PINN on COVID-19 data sets collected from COVID-19 data from 21 regions in Italy in 1st half of 2020. Results show: MP-PINN achieves superior performance in both short-term & long-term forecasting of COVID-19 spread. In particular, MP-PINN outperforms traditional SIR models, pure data-driven approaches (MLP), & single-phase PINNs. See Fig. 1 for a representative case demonstrating efficacy of MP-PINN.

- 2. Related Works.
  - **Epidemic Forecasting.** Briefly review literature in epidemic forecasting most relevant to our work as literature has exploded since COVID-19 outbreak in late 2019:
    - *Model-driven approach.* Compartmental models (Mô hình ngăn), e.g. Susceptible-Infectious-Recovered (SIR) model, are foundational in epidemic modeling due to their simplicity & reliance on mechanistic understanding of disease spread. They are particularly effective for long-term forecasting because they incorporate known epidemiological dynamics [10]. However, their fixed parameters often lead to less accurate short-term predictions, as they cannot easily adapt to rapid changes in transmission dynamics. Have been a plethora of SIR extensions with sophisticated assumptions e.g. SIRD [13], SEIR [4], & SEIRM [24].
    - Data-driven approach. ML models, particularly DL techniques, have gained prominence for short-term forecasting due to their ability to capture complex patterns in large datasets [3]. Techniques e.g. Long Short-Term Memory (LSTM) networks excel in identifying trends & making predictions over short periods. However, their performance deteriorates over longer horizons due to their lack of incorporation of epidemiological knowledge, making them less reliable for long-term predictions [28]. In [12], authors studied forecasting influenza outbreaks, e.g. training model with data in 4 years to predict in future outbreaks, hence training data is much larger than ours which consists of only 1 outbreak.

  Most existing data-driven approaches make short-term predictions (< 1 month). E.g., [9] used 2 months to train & predict next months with prediction windows are 3/7/14 days. Likewise, model in [30] trained on almost 10 months & test on 1 month with prediction window is 7/14/21/28 days. In [14,26], models were trained on 377 days, but window of forecasting in both works is next 7 days (observed previous 21 days). In contrast, train model only on data collected for 35 days, & forecast rest of outbreak (97 days). Thus out setting is much more challenging & most impossible to achieve without utilizing epidemic models & prior knowledge.

- **Physics-Informed Neural Networks (PINNs).** Recently, PINns have emerged as a framework to incorporate known physical rules to train deep neural networks [23]. It has been demonstrated to be effective in solving forwards & inverse problems involving PDEs. Since then, numerous studies have explored application of PINNs in various domains, e.g. fluid dynamics [2], material science [20], & epidemic modelling [24].
  PINNs for epidemic modeling. PINNs have been used to build hybrid forecasting models, integrating model-driven & data-driven methods [7,16,24,27]. [24] proposed to regularize embeddings from both time-dependent model which can be regularized via physical law & exogenous features extractor which obtain information from multiple sources for making better predictions. Wang et al. [31] proposed a physics-informed neural network (PINN) framework for learning parameters of a COVID-19 compartment model from observed data. [5] used epidemic model to compute ahead unobserved data points then augment training process directly with prediction loss. In [5,29], although parameters of epidemic model, e.g., infection rate & recovery rate, are generated by a trainable NNs-based module, potential impact of data instability on learned epidemic parameters is not explicitly addressed. E.g., if case counts change significantly & differ from historical patterns, it can be challenging for [5,29] models to learn valid & stable transmission & recovery rates. In contrast, our work introduces distinct phases within SIR model to capture long-term dynamics of an outbreak, which may not be apparent in short-term or noisy data.

Building on strengths & limitations of these approaches, our proposed MP-PINN framework aims to address gaps in both model-driven & data-driven methods.

- 3. Preliminaries.
  - 3.1. Mechanistic Compartment Models. In epidemiology & other fields where behavior of large populations is studied, compartment models are models in which population are divided into a discrete set of qualitatively-distinct states/types/classes/groups, so-called *compartments* (ngăn). These models also define transition between compartments. Focus on SIR (Susceptible-Infected-Recovered), most popular compartments model used to model spread of infectious diseases in epidemiology [10]. SIR contains 3 compartments: (1) Susceptible (Dễ bị tổn thương), (2) Infected (Bị lây nhiễm), (3) Recovered/removed populations (Dân số được phục hồi/loại bỏ) which are denoted as $S(t), I(t), R(t)$ as a function of time t, resp. At time t, an individual in population is classified into 1 of these compartments, typically transiting from being susceptible to infected & finally recovered (or removed). Hence, size of population N is sum of number of susceptible, infectious, & recovered persons, i.e., $N = S(t) + I(t) + R(t)$. Here, considered a *single outbreak* SIR model comprises a set of ODEs that describe transitions between 3 compartments:

$$\frac{dS(t)}{dt} = -\frac{\beta}{N} S(t) I(t), \tag{64}$$

$$\frac{dI(t)}{dt} = \frac{\beta}{N} S(t) I(t) - \gamma I(t), \tag{65}$$

$$\frac{dR(t)}{dt} = \gamma I(t), \tag{66}$$

  where parameters of model $\beta > 0, \gamma \in (0, 1)$: *infection rate & recovery rate*, resp. In real-world & more complex models, these parameters can also vary over time or depend on different factors e.g. policy or other properties of population. Initial condition of ODEs are $S(0) > 0, I(0) > 0, R(0) \geq 0$.
  An important assumption of SIR model: all recovered individuals (in R group) are completely immune & cannot return to susceptible S or infected I groups, & total population N remains constant. However, N does not always represent entire population, especially in real-world scenarios. E.g., 1 assumption of SIR model [8] : population mixes homogeneously, meaning everyone has same level of interaction with others. However, during early stages of COVID-19 outbreak in 2020, it was impractical to consider entire population as susceptible. Instead, N might only represent a fraction of population. Moreover, population could therefore be divided into 2 distinct groups [32]: those with inherited immunity & those without. These distinctions imply: total number of susceptible individuals $S(t)$ may not always correspond to entire population N, but rather to a specific portion of it, depending on factors e.g. inherited immunity or other epidemiological considerations.
  - 3.2. Physics-Informed Neural Networks (PINNs). PINNs [23] are neural networks equipped with physical constraints of domain, either through network architecture or as a regularization term during training process. These physical laws often involve parameters that need to be estimated. During training of a PINN, both neural network weights & physical parameters are optimized to achieve best fit to observed data while simultaneously satisfying physical constraints.
    More formally, given a training dataset $\mathcal{D}$, learning searches for a neural network $f \in \mathcal{H}$ by solving an optimization problem:

$$f^* = \min_{f \in \mathcal{H}} \mathcal{L}(f; \mathcal{D}) + \lambda \Omega(f), \tag{67}$$

    where $\mathcal{L}(f; \mathcal{D})$: usual data loss function, $\Omega(f)$: a regularization term that introduces physical prior knowledge into learning process, & $\lambda > 0$: a balancing weight. When prior is specified as PDEs, regularization typically takes form of PDE residual loss:

$$\Omega(f) = \frac{1}{L} \sum_{i=1}^{L} (\partial f_{\mathrm{NN}}(x_i) - \partial f_{\mathrm{PDE}}(x_i))^2, \tag{68}$$

    where $\partial f_{\mathrm{NN}}(x_i)$ denotes partial derivative of neural network evaluated at x_i & $\partial f_{\mathrm{PDE}}(x_i)$ denotes corresponding partial derivative specified by PDEs. Evaluation points $i = 1, 2, \ldots, L$ are sampled so that function f & its derivative are well supported.

- 4. Methods. Present main contributions of developing PINNs for epidemic forecasting. Overall framework is depicted in Fig. 2: Multi-phase Physics-Informed Neural Network (MP-PINN) Framework for Epidemic Forecasting. Framework illustrates integration of expert knowledge & data-driven approaches to estimate parameters in a multi-phase scenario, where key parameters e.g. infection rate β & recovery rate γ vary across phases. Start from a single-phase assumption, then advance into multi-phase model. Both single-phase & multi-phase models integrate expert knowledge & a data-driven parameter estimation process. In multi-phase model, however, parameters e.g. infection rate β & recovery rate γ vary between phases, enhancing model's ability to adapt to complex real-world scenarios. Single-phase approach is described in Sect. 4.1 & multi-phase approach is detailed in Sect. 4.2.

  More concretely, using SIR model described in Sect. 3.1 as physics prior, build a PINN framework with separate neural networks $f^S_{\psi_1}(t), f^S_{\psi_2}(t)$ where f_{ψ_i} that takes t as input to predict Susceptible $S(t)$ & Infected $I(t)$, resp. For clarity, present case where input is only time t, but any other features, static or dynamic can be applicable. Note: do not need to model Recovered $R(t)$ because of constraint $S(t) + I(t) + R(t) = N$.

  - 4.1. Single-Phase PINN (SP-PINN).
  - 4.2. Multi-Phase PINN (MP-PINN).

- 5. Experiments.

  - 5.1. Settings.
  - 5.2. Results.

- 6. Conclusions. Introduced MP-PINN (Multi-Phase Physics-Informed Neural Network), a novel approach to epidemic forecasting that addresses limitations of both traditional model-driven & data-driven methods. By integrating mechanistic understanding of SIR models with flexibility of neural networks & allowing for multiple SIR parameters across phases, MP-PINN achieves superior performance in both short-term & long-term forecasting of COVID-19 spread. Our experiments on COVID-19 data from 21 regions in Italy demonstrate: MP-PINN outperforms traditional SIR models, pure data-driven approaches (MLP), & single-phase PINNs. Ability to capture evolving dynamics through multiple phases proves crucial in reflecting impact of changing interventions & public behaviors throughout course of epidemic. MP-PINN's success highlights potential of hybrid approaches that combine domain knowledge with data-driven learning. By allowing for incorporation of expert insights & prior knowledge about parameter ranges, our method provides a flexible framework that can adapt to complex, evolving nature of real-world epidemics. Future work could explore automatic detection of phase transition points & incorporation of additional epidemiological factors. Finally, emphasize: MP-PINN framework is generally applicable to any outbreaks where underlying dynamics may shift over time.

2.4 [RHP21]. Rishikesh Ranade, Chris Hill, Jay Pathak. *DiscretizationNet: A ML Based Solver for NSEs using FV Discretization*

Journal of Computer Methods in Applied Mechanics & Engineering. [139 citations]

Abstract. Over last few decades, existing PDE solvers have demonstrated a tremendous success in solving complex, nonlinear PDEs. Although accurate, these PDE solvers are computationally costly. With advances in ML technologies, there has been a significant increase in research of using ML to solve PDEs. Goal: develop an ML-based PDE solver, that couples' important characteristics of existing PDE solvers with ML technologies. 2 solver characteristics that have been adopted in this work are:

1. use of discretization-based schemes to approximate spatio-temporal partial derivatives &

2. use of iterative algorithms to solve linearized PDEs in their discrete form.

In presence of highly nonlinear, coupled PDE solutions, these strategies can be very important in achieving good accuracy, better stability & faster convergence. Our ML-solver, DiscretizationNet, employs a generative CNN-based encoder-decoder model with PDE variables as both input & output features. During training, discretization schemes are implemented inside computational graph to enable faster GPU computation of PDE residuals, which are used to update network weights that result into converged solutions. A novel iterative capability is implemented during network training to improve stability & convergence of ML-solver. ML-Solver is demonstrated to solve steady, incompressible NSEs in 3D for several cases e.g., lid-driven cavity, flow past a cylinder & conjugate heat transfer.

Keywords. PDEs; ML; Discretization Methods; Physics-Informed Learning

- 1. Introduction. Coupling of physics & DL to solve problems in engineering simulation space has drawn interest in recent years. In context of neural networks, this has been achieved by constraining network optimization by embedding physical constraints in loss formulation. These physics-based constraints ensure: solution space is bounded & obeys physical laws. Although this idea was proposed back in 90s [1,2], it has started to have a big impact in recent times due to rapid advances in computational sciences & DL.

  Within context of physics-based DL there are 2 types of methods used to approximate PDEs governing physical processes: data-driven & data-free. Data-driven methods use simulation or experimental data to construct models while enforcing physical laws. These models heavily depend on fidelity of data & hence are limited in accuracy & generalizability. Data-free methods use neural networks to generate solutions by rigorously constraining partial differential governing equations through loss formulation. In this respect, Raissi et al. [3,4] recently introduced Physics Informed Neural Network (PINN) framework which

approximates partial derivatives of solution variables w.r.t. space & time using automatic differentiation (AD) [5]. Partial derivatives are used to estimate PDE losses which are back-propagated to neural network for weight updates. In addition to constraints on PDE residual, boundary & initial conditions are specified on computational domain & constrained in loss formulation to ensure that a unique solution is obtained at convergence. Based on this work, PINN framework has been extended in different directions to improve effectiveness of learning PDEs & to develop a theoretical understanding of PINNs. Recently, Kharzami et al. [6] developed Variational PINNs to solve PDE in its weak form. Similarly, Khodayi-Mehr et al. [7] proposed VarNet, where loss formulation was based on integral form of PDE as opposed to differential form. In recent effects a distributed version of PINN has been explored, where learning problem is decomposed into smaller regions of computational domain & a physical compatibility condition is enforced in between neighboring domains [8–11]. Other variations of PINN as well as convergence & stability of PINN-based algorithms has been studied in [12–18].

PINn methodology has been demonstrated to solve a number of canonical PDEs as well as to different applications involving vastly different physics [19–34]. A number of these studies have used PINN framework to solve more complex PDEs e.g. NSEs, which is focus of this work. Dwivedi et al. [35] developed a Distributed-PINN to address some of issues with PINN & demonstrated it to solve NSE in a lid-driven cavity at low Reynolds numbers. Sun et al. [36] demonstrated PINN methodology for surrogate modeling of fluid flow at low Reynolds numbers. Zhu et al. [37] implemented physical constraints on an encoder–decoder network in conjunction with flow based conditional generative models for stochastic modeling of fluid flows. Rao et al. [38] demonstrated PINN approach to solve NSEs at low Reynolds numbers for a 2D flow over a cylinder. Very recently, Jin et al. [39] proposed PIN approach for solving NSEs in both laminar & turbulent regimes. Gao et al. [40] proposed PhysGeoNet to solve NSEs using finite difference discretizations of PDE residuals in neural network loss formulation. Methodology was demonstrated in irregular domains using elliptic coordinate mappings to transform spatial coordinates.

Computation of PDE loss & choice of network architecture used for network optimization become crucial when solving for highly nonlinear, multidimensional, stiff, coupled PDEs e.g. system of NSEs. Highly nonlinear solution space accessed by NSEs may be challenging to resolve due to presence of sharp local gradients in a broad computational domain. As a result, methodology used for computing gradients as well as approach of network training can be very important in achieving accurate solutions, better stability & faster convergence in training process.

Traditional PDE solver technologies developed over last decades have primarily relied on solving discretized formulations of PDEs using methods e.g. FD, FE, or FV. Exact or approximate forms of linearized discrete equations are used, in combination with linear equation solvers, to improve solutions iteratively. Discretization method allows access to higher order & advanced numerical approximations for partial derivatives which can be useful in resolving highly nonlinear parts of PDE solution & also add artificial dissipation to improve solver stability. Additionally, these schemes coupled with iterative solution strategy have proved to be robust in terms of solver stability & convergence. Recently, ML based models have been developed to either learn new discretization schemes from solution data [41,42] or to mimic these schemes through novelties in neural network architectures [43,44]. On other hand, Ansys suite of software already has access to a large number of advanced discretization schemes that can capture complexities over a wide range of physics. Coupling of these discretization schemes with ML algorithms, along with iterative solution algorithm, can provide same benefits in ML-based solvers as observed with traditional solvers.

Main goal: introduce a new ML-solver, DiscretizationNet, which is a framework that couples solver characteristics with generative networks to solve highly nonlinear, multidimensional, stiff, coupled PDEs. Solver does not require any training data but generates PDE solutions & simultaneously learns them during training process. Different FV based numerical schemes are implemented inside computational graph to enable fast, vectorized operations on GPU & a modified encoder–decoder network architecture is proposed to solve PDEs in an iterative manner. Finally, discretization based iterative ML solver is used to solve steady, incompressible, NSE in a 3D for a several cases e.g., lid-driven cavity flow at a high Reynolds number, flow past a cylinder in laminar regime & conjugate heat transfer. Remainder of paper is organized as follows. Sect. 2: introduce solution methodology adopted in DiscretizationNet. Sect. 3: discuss numerical results followed with conclusions. Sect. 4: future work.

- 2. Solution methodology. Discuss methodology for solving system of steady, incompressible NSEs using DiscretizationNet. System of NSEs consists of continuity equation & momentum equation for each directional velocity component. Non-dimensional equations described in a vectorized form:

$$\nabla \cdot \mathbf{v} = 0, \tag{69}$$

$$(\mathbf{v} \cdot \nabla)\mathbf{v} + \nabla p - \frac{1}{\mathrm{Re}}\Delta \mathbf{v} = 0, \tag{70}$$

where $\mathbf{v}$: non-dimensional velocity vector, $\mathbf{v} = (u, v, w)$, p: non-dimensional pressure, ∇: divergence operator, Re: Reynolds number.

When solving using neural networks, PDE described in (1) are used to compute residuals in loss formulation. Some of reasons that can result in a stiff formulation of PDE residual-based loss term include, complicated geometries, strong coupling of PDE variables in large system of coupled PDEs, presence of multiple domains with different PDE formulations & material properties (e.g. conjugate heat transfer between fluid & solid domains) & reasonably large Reynolds numbers, where nonlinear, convective component, $(\mathbf{v}\cdot\nabla)\mathbf{v}$, is dominant. Automatic differentiation (AD) [5] allows computation of partial derivatives within computational graph using back-propagation, but stiffness of computed gradients may affect accuracy, stability & convergence of neural network training, & may require a large number of training epochs, use of regularization techniques, as well as deep network architectures, which have their own set of problem, e.g. vanishing gradients. Considering these challenges, it becomes

increasingly important to resolve such PDEs using advanced numerical schemes & develop novel strategies of neural network training. Existing solver methodologies have solved some of these problems & in this work, draw from vast pool of knowledge to develop our ML-based solver. Next, introduce network architecture used in work followed by loss formulation & training mechanics.

- 2.1. DiscretizationNet architecture. Network proposed in this work is a generative Convolutional Neural Network (CNN) based encoder–decoder whose input features are flow variables, $\mathbf{v} = (u, v, w, p)$, initialized with random uniform solution fields, boundary condition encoding b, & level set of geometry h. Level sets are real valued functions which depict geometry s.t. regions inside geometry are flagged with -1, region outside it with 1 & regions representing surface to 0 [45]. Objective of this network, as shown in Fig. 1: DiscretizationNet for Navier–Stokes solution, is to compress input features $(\underline{u}, \underline{v}, \underline{w}, \underline{p}, \underline{b}, \underline{h})$ into a lower dimensional space, (η), using a convolutional encoder & to decode latent vector encoding to new solutions, $(\hat{u}, \hat{v}, \hat{w}, \hat{p})$, which are closer to actual Navier–Stokes solutions. It may be observed: boundary conditions as well as geometry level sets are used to enrich encoded latent space & also in computation of coupled PDE loss terms, (R_c, R_u, R_v, R_w), corresponding to continuity & momentum equations in each spatial direction. Enrichment of latent space with geometry & boundary information is crucial as it ensures that network outputs are conditioned upon them. Geometry & boundary encoder networks are pre-trained on similar samples & their weights are used to perform encoding in this network. Encoding geometry & boundary is crucial in this approach, because, in their original form, these representations can be very sparse. This may have an adverse effect on learning & generalizability of DiscretizationNet. Additionally, network proposed here can be used to compute a large batch of solutions at different boundary & geometry conditions in a single training session. As a result, conditioning of solutions with boundary & geometry conditions enables generalization of PDE solutions for a large set of problems. Purpose of designing DiscretizationNet as an encoder–decoder network: obtain a legitimate lower-dimensional encoding of PDE solution space. Encoded solution space can be useful in developing reduced-order models (ROM) on top of DiscretizationNet.
- 2.2. Training mechanics. It was suggested previously: input vectors to encoder–decoder network are randomly initialized solution fields of velocity & pressure. This has 2 implications:
  1. PDE solution encoding (η) does not have a physical meaning &
  2. decoding solutions, which are functions of random noise, is a difficult task & may slow down convergence of network & provide poor stability.

  In order to tackle these challenges, an iterative approach is followed, where inputs to network are replaced with newly generated solutions, every time PDE residuals reduce by an order of magnitude. At any given point during training, solutions generated are dependent on solutions from a previous iteration & not initial random solutions used in beginning. This allows network to converge from partially converged solutions to fully converged solutions in an iterative fashion & improves stability & convergence as compared to other ML methods in this space. At convergence, when L^2 norm of PDE residuals have dropped to a reasonably low level, input & output solutions are very similar & effectively turns network into a conditional autoencoder. It is a conditional autoencoder, because decoder network is conditioned on solution encoding as well as encoding of geometry & boundary. This iterative strategy provides a physical meaning to reduced dimensional solution latent space, which can now describe flow solutions at given boundary, geometry or flow conditions. Moreover, PDE solutions are independent of spatial dimension & depend only on solutions at previous iterations. This is analogous to how traditional solvers function & additionally provides an opportunity to operate this network under transient conditions. Important to note: training is completely data free & goal: generate solutions by minimizing PDE residuals & simultaneously learn them into an encoded latent space.
- 2.3. Geometry & boundary encoder. In this sect, elaborate on geometry & boundary encoders used in network architecture in Fig. 1. In this work, use a modified level set approach to represent geometry, s.t. voxels inside geometry are represented by 0 & outside by 1. Gradient of level set are used to track voxels activated by surfaces of geometry. Level sets for primitive geometries of different shape, size, & orientation are learned using a generative encoder-decoder network & represented in a lower-dimensional space η_h. A schematic of geometry autoencoder can be seen in Fig. 2A: Geometry & boundary autoencoders. In this work, encoder & decoder networks are CNN-based & a binary cross-entropy loss function is employed to update weights of networks. Level sets of different geometries can be generated by parsing through latent space vector of a trained geometry encoder & used to parameterize ML-solver.

  On other hand, a separate boundary autoencoder is used to represent different boundary conditions. Boundary encoder is only required if boundary conditions are spatially or temporally varying. Here, propose a generative encoder–decoder network to learn boundary condition encoding but leave choice of network architecture open. In scenarios where boundary condition is constant along different surfaces, as is in all of test cases demonstrated in this work, a neural network based boundary condition encoder is not required. Instead, a custom encoding can be constructed & used as latent vector, η_b. Flow conditions, e.g. Reynolds number or Prandtl number can also be perceived as boundary conditions & be added to this latent vector. E.g., an encoding of $\eta_b = [1, 1, 2, 1, 0.3, 1.2, 0, 3, 40]$ can be perceived as Dirichlet inlet boundary conditions (1) on left, right, & bottom surfaces with specified values of $0.3, 1.2, 3.0$, resp., & a Neumann boundary condition (2) on top surface, where flux of variable equal 0. Reynolds number (40) or other flow conditions can also be specified in encoding. A similar choice of boundary condition encoding is employed in this work.
- 2.4. Loss formulation. Loss formulation of network comprises of PDE residual from all governing equations. Each PDE has its own loss formulation given as follows: (2)

$$\lambda(\mathbf{W}, \mathbf{b}) = \|\lambda_c\|_\Omega + \|\lambda_u\|_\Omega + \|\lambda_v\|_\Omega + \|\lambda_w\|_\Omega, \tag{71}$$

where $\|\lambda_c\|_\Omega$: L^2 norm of continuity residual, $\|\lambda_u\|_\Omega$: L^2 norm of x-momentum residual, $\|\lambda_v\|_\Omega$: L^2 norm of y-momentum residual, $\|\lambda_w\|_\Omega$: L^2 norm of z-momentum residual & Ω: solution space. Moreover, this network can generate & learn a large set of solutions at different Reynolds numbers. Different solutions simply form a part of training samples of encoder–decoder network.

Computation of PDE residuals involves approximation of 1st & 2nd order spatial gradients. Employ traditional FV discretization technique to compute PDE residual loss & moreover, all numerical schemes are implemented inside computational graph to enable fast, vectorized GPU computation. However, this does not limit use of other discretization schemes e.g. FEM, Discontinuous Galerkin (DG) etc., if higher order elements are required.

In FV discretization here, each voxel of PDE solution is considered as cell center of an imaginary, finite control volume (CV) as shown in Fig. 3: FV stencil on interior & boundary pixels in a computational graph. Volume integrals on CV are expressed as surface integrals using Green–Gauss divergence theorem shown below.

$$\int (\nabla \cdot \mathbf{v}) \, \mathrm{d}\mathbf{x} = \sum_{i}^{M} v_i n_i A_i \tag{72}$$

where v: a solution variable, M: number of faces on CV, n_i: normal along each face on CV & A_i: area of each face on CV. Hence, a face-based approach is used to compute gradients across all interior & boundary faces of each control volume in computation graph. 2nd order gradients are also computed using (3), with only difference: solution variable is replaced by its gradient. Convective fluxes are discretized using 1st or 2nd-order upwind scheme & diffusive fluxes are evaluated using a central difference approximation. A 2nd order approximation is used for computing gradients of pressure field. Since dealing with an incompressible formulation of Navier–Stokes & discretization is analogous to FV discretization on collocated grids, pressure field is prone to checker-boarding due to a lack of explicit coupling between pressure & velocity & use of 2nd order numerical schemes. In this work, pressure-velocity coupling is achieved using Rhie–Chow interpolation [46], which essentially adds a 4th order dissipation of pressure to continuity equation. Addition of Rhie–Chow flux suffices & more sophisticated schemes e.g. SIMPLE [47] are not required but remain an option. In Rhie–Chow formulation, velocity at faces is interpolated as shown in (4):

$$u_e = \frac{u_i + u_{i+1}}{2} + \frac{1}{a_p}(p_i + \nabla P_i \cdot \bar{r}_i - p_{i+1} - \nabla P_{i+1} \cdot \bar{r}_{i+1}), \tag{73}$$

where u_e: velocity approximation at *east* face of control volume, p: pressure, ∇P: gradient of pressure, a_p: matrix coefficients from momentum equations.

Boundary condition treatment is incorporated through discretization of boundary voxels by enforcing boundary constraints in flux computation & enforces order of accuracy at boundary, as opposed to using 1-sided finite difference schemes. E.g., x-velocity gradients along a boundary as shown in Fig. 3 is represented as follows:

$$\left(\frac{du}{dx}\right)_G = \frac{u_B + u_G}{2\Delta x_B} - \frac{u_B + u_I}{2\Delta x_I}, \tag{74}$$

$$u_G = 2u_b - u_B, \tag{75}$$

where u_b: specified boundary condition, u_G: an imaginary ghost pixel, u_B: boundary voxel, & u_I: interior voxel adjacent to boundary in direction of gradient. In case of unstructured boundaries, e.g., cells adjacent to walls of a cylinder, a stair-step discretization [48] or a cut-cell discretization [49] can be implemented. In this work, have implemented stair-step discretization, where boundary conditions at unstructured boundaries are implemented similarly as (5).

Numerical schemes at both interior & boundary voxels are implemented together in computational graph using vectorized operation for fast computation on GPU. Discretization schemes are implemented through custom hidden layers in Keras [50], where solution variable tensors as well as boundary & geometry condition tensors are used to compute PDE residuals of coupled PDE system at each voxel. As a result, loss formulation in (2) implicitly contains boundary information & a separate loss term is not needed to model it, as in previous studies [3,4], thereby avoiding need to use strategies for multi-objective optimization. Moreover, use of discretization techniques to compute loss, allows access to higher order approximations for higher accuracy & advanced numerical schemes e.g. Rhie–Chow flux [46] etc. that can enhance stability & convergence of solution & neural network training by providing additional physics-based regularization.

- 2.5. Inference for other geometry & boundary conditions. Network architecture described in Fig. 1 is a generative encoder–decoder network, where, at convergence, input & output samples are essentially actual solutions of given PDEs. It may be understood: model obtained at convergence has learned to encode actual PDE solutions & decode them from solution latent space combined with geometry & boundary encoding. As a result, model in its current form cannot be directly used for inferencing solutions at other geometry & boundary conditions, since actual PDE solutions may be required as inputs to network & these are obviously not available.

  Here, propose a novel algorithm that enables inferencing for other geometry & boundary conditions. A schematic diagram of algorithm is shown in Fig. 4: Algorithm for inferencing & important steps are outlined:

  1. On a given geometry & boundary condition initialized for inference, geometry & boundary encoding, η_h, η_b are computed using their respective encoder networks.
  2. Since, solution encoding η is unknown, it is initialized with a random field drawn from a uniform distribution.

3. Initial solution encoding combined with geometry & boundary encoding is passed through trained weights of CNN decoder to generate a solution field $\hat{u}, \hat{v}, \hat{w}, \hat{p}$.
4. Solution field is encoded to a new solution encoding using trained weights of CNN encoder $\hat{\eta}$.
5. New solution encoding $\hat{\eta}$ replaces solution encoding of previous iteration η & steps (iii) & (iv) are repeated until $\|\eta - \hat{\eta}\|_{L^2} < 1e^{-6}$. Geometry & boundary encoding are fixed during entire process.
6. At convergence, PDE solutions at a given geometry & boundary condition are decoded using most recent η.

May be observed: solution inference happens in encoded latent vector space & goal of iterative procedure: steer solution latent vector to a space that is in close proximity to latent vectors observed in network training. Since geometry & boundary condition encoding are fixed for a given problem, they provide necessary constraints for solution latent vector to iteratively improve itself & generate an accurate PDE solution. Outcome of this algorithm, in terms of generalization, improves with number of different variations of geometry & boundary conditions adopted during training. Generally, starting from weights obtained from a well trained model, inference algorithm converges in < 10 iterations. Although not a scope of our current work, functioning in space of latent vectors may provide an opportunity to explore new solution spaces of a given PDE & construct computationally inexpensive reduced order models.

- 3. Results. Provide detailed numerical experiments to demonstrate ML-solver for several cases of fluid flow e.g. lid-driven cavity ,flow past a cylinder & conjugate heat transfer. Proposed ML-solver is validated against ANSYS Fluent 19.3 CFD [51] solver for solving incompressible, steady NSE for these different cases.
  - 3.1. Lid-driven cavity flow.
  - 3.2. Laminar flow past a cylinder.
  - 3.3. Conjugate heat transfer.
- 4. Conclusion. Have presented a novel ML-Solver, which uses important characteristics from existing PDE solvers for solving system of steady, incompressible NSE. ML-solver does not require any training data & instead, generates & learns PDE solutions simultaneously, during training process. It uses discretization techniques to approximate PDE residual at each voxel of a given computational domain & uses L^2 norm of residual to update network weights. Discretization schemes are implemented inside computational graph to enable vectorization on GPU & provide access to numerous higher order & advanced based regularization. In this work, have extended discretizations to unstructured domains by employing stair-step discretizations to provide flexibility in modeling different types of geometries as well as widely varying boundary conditions.

  From network architecture perspective, introduce DiscretizationNet, which is a generative CNN-based encoder–decoder network conditioned on geometry & boundary conditions. Separate autoencoders are constructed to learn lower-dimensional vectors (or encodings) for different geometry & boundary conditions. These encodings are used to enrich & parameterize solution latent vector space of generative network & thus allow for simultaneously generating & learning a wide range of solutions at different conditions in same training session. Moreover, employ a novel iterative capability in network to mimic existing PDE solvers. In this implementation, inputs to generative model are replaced with outputs during network training, as network learns to generate better solutions. This strategy is unique & have observed: it provides better stability & faster convergence in comparison to other ML strategies, especially in cases when ground truth solutions are not known. Additionally, have proposed an algorithm for interencing using DiscretizationNet. Algorithm functions in latent space to iteratively infer solutions using trained model weights.

  Have validated ML-solver by solving 3D steady, incompressible NSEs on 3 different cases, (i) lid-driven cavity, (ii) laminar flow past a cylinder, & (iii) conjugate heat transfer. Contour & line plot comparisons made with ANSYS Fluent R19.3 [51] in all 3 cases show a good agreement. Additionally, it has been observed: training for a large number of PDE solutions results in a stable convergence within 3×10^4 training epochs.

  ML-solver proposed here can be extended to solve unsteady problems using LSTM-type networks [55]. Deficiencies in stair-step discretization, in computing accurate solutions near boundaries can be mitigated by using a cut-cell of unstructured grid discretization. Moreover, ML-solver can be applied to other PDEs with complex physics as well as to develop computationally inexpensive, low-dimensional models.

3 Learning Theory

Resources – Tài nguyên.

1. [Bac24]. Francis Bach. *Learning Theory from 1st Principles.*

   Amazon review. A comprehensive & cutting-edge introduction to foundations & modern applications of learning theory.

   Research has exploded in field of machine learning resulting in complex mathematical arguments that are hard to grasp for new comers. In this accessible textbook, Francis Bach presents foundations & latest advances of learning theory for graduate students as well as researchers who want to acquire a basic mathematical understanding of most widely used machine learning architectures. Taking position that learning theory does not exist outside of algorithms that can be run in practice, this book focuses on theoretical analysis of learning algorithms as it relates to their practical performance. Bach provides simplest formulations that can be derived from 1st principles, constructing mathematically rigorous results & proofs without overwhelming students.

- Provides a balanced & unified treatment of most prevalent machine learning methods
- Emphasizes practical application & features only commonly used algorithmic frameworks
- Covers modern topics not found in existing texts, e.g. overparametrized models & structured prediction
- Integrates coverage of statistical theory, optimization theory, & approximation theory
- Focuses on adaptivity, allowing distinctions between various learning techniques
- Hands-on experiments, illustrative examples, & accompanying code link theoretical guarantees to practical behaviors

About the Author. Francis Bach is a researcher at Inria where he leads the machine learning team which is part of the Computer Science department at Ecole Normale Supérieure. His research focuses on machine learning & optimization.

Preface. **Why study learning theory?** Data have become ubiquitous in science, engineering, industry, & personal life, leading to need for automated processing. Machine learning is concerned with making predictions from training examples & is used in all of these areas, in small & large problems, with a variety of learning models, ranging from simple linear models to deep neural networks. It has now become an important part of algorithmic toolbox.

How can we make sense of these practical successes? Can we extract a few principles to understand current learning methods & guide design of new techniques for new applications or to adapt to new computational environments? This is precisely goal of learning theory. Beyond being already mathematically rich & interesting (as it imports from many mathematical fields), most behaviors seen in practice can, in principle, be understood with sufficient effort & idealizations. In return, once understood, appropriate modifications can be made to obtain even greater success.

Why read this book? Goal of this textbook: to present old & recent results in learning theory for most widely used learning architectures. Doing so, a few principles are laid out to understand overfitting & underfitting phenomena, as well as a systematic exposition of 3 types of components in their analysis, estimation, approximation, & optimization errors. Moreover, goal: not only to show: learning methods can learn given sufficient amounts of data but also to understand how quickly (or slowly) they learn, with a particular eye toward adaptivity to specific structures that make learning faster (e.g. smoothness of prediction functions or dependence on low-dimensional subspaces).

This book is geared toward theory-oriented students, as well as students who want to acquire a basic mathematical understanding of algorithms used throughout machine learning & associated fields that are significant users of learning methods (e.g. computer vision & natural language processing). Moreover, it is well suited to students & researchers coming from other areas of applied mathematics or computer science who want to learn about theory behind machine learning. Finally, since many simple proofs have been put together, it can serve as a reference for researchers in theoretical machine learning.

A particular effort will be made to prove *many results from 1st principles* while keeping exposition as simple as possible. This will naturally lead to a choice of key results showcasing essential concepts in learning theory in simple but relevant instances. A few general results will also be presented without proof. Of course, concept of 1st principles is subjective, & I will assume readers have a good knowledge of linear algebra, probability theory, & differential calculus.

Moreover, focus on part of learning theory that deals with algorithms that can be run in practice, & thus, all algorithmic frameworks described in this book are routinely used. Since many modern learning methods are based on optimization, Chap. 5 is dedicated to that topic. For most learning methods, present some simple *illustrative experiments* with accompanying code (MATLAB & Python for moment, & Julia in future) so students can see for themselves that algorithms are simple & effective in synthetic experiments. Exercises currently come with no solutions & are meant to help students understand related material.

Finally, 3rd part of book provides an in-depth discussion of *modern special topics* e.g. online learning, ensemble learning, structured prediction, & overparametrized models.

Note: this is not an introductory textbook on machine learning. There are already several good ones in several languages (see, e.g., Alpaydin, 2020; Lindholm et al., 2022; Azencott, 2019; Alpaydin, 2022). This textbook focuses on learning theory – i.e., deriving mathematical guarantees for most widely used learning algorithms & characterizing what makes a particular algorithmic framework successful. In particular, given that many modern methods are based on optimization algorithms, put a significant emphasis on gradient-based methods & their relation with machine learning.

A key goal: to look at simplest results to make them easier to understand, rather than focusing on material that is more advanced but potentially too hard at 1st & provides only marginally better understanding. Throughout book, propose references to more modern work that goes deeper.

Book organization. Book comprises 3 main parts: an introduction, a core part, & special topics. Readers are encouraged to read 1st 2 parts to understand main concepts fully & can pick & choose among special topic chapters in a 2nd reading or if used in a 2-semester class.

All chapters start with a summary of main concepts & results that will be covered. All simulation experiments are available at https://www.di.ens.fr/~fbach/ltfp/ as MATLAB & Python code. Many exercises are proposed & are embedded in text with dedicated paragraphs, with a few mentioned within text (e.g., as "proof left as an exercise"). These exercises are meant to deepen understanding of nearby material, by proposing extensions or applications.

Many topics are not covered at all, & many others are not covered in depth. There are many good textbooks on learning theory that go deeper or wider (e.g., Christmann & Steinwart, 2008; Koltchinskii, 2011; Mohri et al., 2018; Shalev-Shwartz & Ben-David, 2014). See also the nice notes from Alexander Rakhlin & Karthik Sridharan[1], as well as from Michael Wolf.

[1] http://www.mit.edu/~rakhlin/notes.html.

In particular, book focuses primarily on real-valued prediction functions, as it has become the de factor standard for modern machine learning techniques, even when predicting discrete-valued outputs. Thus, although its historical importance & influence are crucial, choose not to present Vapnik-Chervonenkis dimension (see, e.g., Vapnik & Chervonenkis, 2015), & instead based my generic bounds on Rademacher complexities. This focus on real-valued prediction functions makes least-squares regression a central part of theory, which is well appreciated by students. Moreover, this allows for drawing links with related statistical literature.

Some areas, e.g. online learning or probabilistic methods, are described in a single chapter to draw links with classical theory & encourage readers to learn more about them through dedicated books. Have also included Chap. 12 on overparametrized models & Chap. 13 on structured prediction, which present modern topics in machine learning. More generally, goal in 3rd part of book (special topics) was, for each chapter, to introduce new concepts, while remaining a few steps away from core material & using unified notations.

A book is always a work in process. In particular, there are still typos & almost surely places where more details are needed. Convinced: more straightforward mathematical arguments are possible in many places in book. Let me know if you have any elegant & simple ideas I have overlooked.

Mathematical notations. Throughout textbook, provide unified notations:

- Random variables: given a set $\mathcal{X}$, will use lowercase notation for a random variable with values in $\mathcal{X}$, as well as for its observations. Probability distributions will be denoted μ or p & expectations as $\mathbb{E}[f(x)] = \int_{\mathcal{X}} f(x)\,\mathrm{d}p(x)$: slightly ambiguous but will not cause major problems (& is standard in research papers). In this book, following most of learning theory literature, will gloss over measurability issues to avoid overformalizations. For a detailed treatment, see Devroye et al. (1996) & Christmann & Steinwart (2008).
- Norms on $\mathbb{R}^d$: will consider usual l_p-norms on $\mathbb{R}^d$, defined through $\|x\|_p^p = \sum_{i=1}^d |x_i|^p$ for $p \in [1,\infty)$, with $\|x\|_\infty = \max_{i\in\{1,\dots,d\}} |x_i|$.
- For a symmetric matrix $A \in \mathbb{R}^{n\times n}$, $A \succeq 0$ means A is positive semidefinite (i.e., all of its eigenvalues are nonnegative), & for 2 symmetric matrices A, B, $A \succeq B$ means $A - B \succeq 0$. For a vector $\lambda \in \mathbb{R}^n$, $\mathrm{Diag}(\boldsymbol{\lambda})$: diagonal matrix with diagonal vector $\boldsymbol{\lambda}$.
- For a differentiable function $f : \mathbb{R}^d \to \mathbb{R}$, its gradient at $\mathbf{x}$ is denoted $f'(\mathbf{x}) \in \mathbb{R}^d$, & if it is twice differentiable, its Hessian is denoted as $f''(\mathbf{x}) \in \mathbb{R}^{d\times d}$.

How to use this book? 1st 9 chapters (in sequence, without diamond parts) are adapted for a 1-semester upper-undergraduate or graduate class, if possible, after an introductory course on machine learning. Following 6 chapters can be read mostly in any order & are here to deepen understanding of some special topics; they can be read as homework assignments (using exercises) or taught within a longer (e.g., 2-semester) class. Book is intended to be adapted to self-study, with 1st 9 chapters being read in sequence & last 6 in random order. In all situations, Chap. 1, on mathematical preliminaries, can be read quickly & studied in more detail when relevant notions are needed in subsequent chapters.

- 1. Mathematical Preliminaries. Chapter Summary: *Linear algebra*: A bag of tricks to avoid lengthy & faulty computations. *Concentration inequalities*: For n independent random variables, derivation between empirical average & expectation is of order $O(\frac{1}{\sqrt{n}})$. What is in big O, & how does it depend explicitly on problem parameters?

  Mathematical analysis & design of ML algorithms require specialized tools beyond classic linear algebra, differential calculus, & probability. In this chapter, review these nonelementary mathematical tools used throughout book: 1st, linear algebra tricks, & then concentration inequalities. Chapter can be safely skipped for readers familiar with linear algebra & concentration inequalities since relevant results will be referenced when needed.

  - Linear Algebra & Differential Calculus. Review basic linear algebra & differential calculus results that will be used throughout book. Using these usually greatly simplifies computations. Matrix notations will be used as much as possible.
    - Minimization of Quadratic Forms. Given a positive-definite (& hence invertible) symmetric matrix $A \in \mathbb{R}^{n\times n}$ & vector $\mathbf{b} \in \mathbb{R}^n$, minimization of quadratic forms with linear terms can be done in closed form:

$$\inf_{\mathbf{x}\in\mathbb{R}^n} \frac{1}{2}\mathbf{x}^\top A\mathbf{x} - \mathbf{b}^\top\mathbf{x} = -\frac{1}{2}\mathbf{b}^\top A^{-1}\mathbf{b}, \tag{76}$$

      with minimizer $x_\star = A^{-1}\mathbf{b}$ obtained by zeroing gradient $f'(\mathbf{x}) = A\mathbf{x} - \mathbf{b}$ of function $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top A\mathbf{x} - \mathbf{b}^\top\mathbf{x}$. Moreover, have $\frac{1}{2}\mathbf{x}^\top A\mathbf{x} - \mathbf{b}^\top\mathbf{x} = \frac{1}{2}(\mathbf{x}-\mathbf{x}_\star)^\top A(\mathbf{x}-\mathbf{x}_\star) - \frac{1}{2}\mathbf{b}^\top A^{-1}\mathbf{b}$. If A were not invertible (simply positive semidefinite) & $\mathbf{b}$ were not in column space of A, then infimum would be $-\infty$.
      Note this result is often used in various forms, e.g.

$$\mathbf{b}^\top\mathbf{x} \le \frac{1}{2}\mathbf{b}^\top A^{-1}\mathbf{b} + \frac{1}{2}\mathbf{x}^\top A\mathbf{x},\ \mathbf{b}^\top\mathbf{x} = \frac{1}{2}\mathbf{b}^\top A^{-1}\mathbf{b} + \frac{1}{2}\mathbf{x}^\top A\mathbf{x} \Leftrightarrow \mathbf{b} = A\mathbf{x}. \tag{77}$$

      This form is exactly Fenchel–Young inequality (see Wikipedia/convex conjugate) for quadratic forms & often used in 1D in form $ab \le \frac{a^2}{2\eta} + \frac{\eta b^2}{2}$, $\forall \eta \ge 0$ & equality iff $\eta = \frac{a}{b}$.

* Inverting a 2×2 Matrix. Solving small systems happens frequently, as well as inverting small matrices. This can be easily done in 2D. Let $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ be a 2×2 matrix. If $ad - bc \neq 0$, then may invert it as

$$M^{-1} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}. \tag{78}$$

This can be checked by multiplying 2 matrices or using Cramer's rule, & it can be generalized to matrices defined by blocks.

* Inverting Matrices Defined by Blocks, Matrix Inversion Lemma.

Above example may be generalized to matrices of form $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$ with blocks of consistent sizes (note: A, D have to be square matrices). Inverse of M may be obtained by applying directly Gaussian elimination in block form. Given 2 matrices $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, N = \begin{pmatrix} I & 0 \\ 0 & I \end{pmatrix}$, may linearly combine lines (with same coefficients for 2 matrices). Once M has been transformed into identity matrix, N has been transformed to inverse of M.
Make simplifying assumption that A is invertible, use notation $M/A = D - DA^{-1}B$ for Schur complement of block A & also assume that M/A is invertible. Thus get by Gaussian eliminination, referring to $L_i, i = 1, 2$ as 2 lines of blocks, so for 1st matrix $M = \begin{pmatrix} L_1 \\ L_2 \end{pmatrix}$: ***

- 2. Introduction to Supervised Learning. Chapter Summary: *Decision theory (loss, risk, optimal predictors)*: What is optimal prediction & performance given infinite data & infinite computational resources? *Statistical learning theory*: When is an algorithm "consistent"? *"No free lunch" theorems*: Learning is impossible without making assumptions.
  Present supervised learning problem: main object of study in this book. After a short introduction highlighting main motivating practical examples in Sect. 2.1, decision-theoretic probabilistic framework set forth in Sect. 2.2 provides traditional mathematical formalization, with notion of loss, risk, & optimal predictor. This will precisely define goals & evaluation standards of machine learning that will be applied to learning algorithms presented throughout this book. Sect. 2.3 presents 2 main classes of learning algorithms: local averaging techniques, & methods based on empirical risk minimization. Notions of statistical consistencies are described in Sect. 2.4; studying consistency of learning methods: main objective in this book: as shown in Sect. 2.5 on "no free lunch" theorems, no method can perform uniformly well, & assumptions have to be made to obtain meaningful quantitative results, as shown in Sect. 2.6. Sect. 2.7: present classical extensions to basic supervised supervised learning frameworks, & in Sect. 2.8: a summary & an outline of subsequent chapters of this book.
  - 2.1: From Training Data to Predictions. **Main goal.** Give some observations $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}, i = 1, \ldots, n$, of inputs/outputs, features/labels, covariates/responses (which are referred to as "training data"), main goal of supervised learning is to predict a new $y \in \mathcal{Y}$ given a new previously unseen $x \in \mathcal{X}$. Unobserved data are usually referred to as "testing data."
    – **Mục tiêu chính.** Đưa ra 1 số quan sát $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}, i = 1, \ldots, n$, của các đầu vào/đầu ra, các nhãn/tính năng, các phản hồi của biến phụ thuộc/(được gọi là "dữ liệu đào tạo"), mục tiêu chính của học có giám sát là dự đoán 1 $y \in \mathcal{Y}$ mới khi biết trước 1 $x \in \mathcal{X}$ mới chưa từng thấy. Dữ liệu chưa quan sát thường được gọi là "dữ liệu thử nghiệm."
    There are few fundamental differences between ML & branch of statistics dealing with regression & its various extensions, particularly when providing theoretical guarantees. Focus on algorithms & computational scalability is arguably stronger within ML (but also exists in statistics). At same time, emphasis on models & their interpretability beyond their predictive performance is more prominent within statistics (but also exists in ML).
    Examples. Supervised learning is used in many areas of science, engineering, & industry. There are thus many examples where $\mathcal{X}, \mathcal{Y}$ can be very diverse:
    * **Inputs** $x \in \mathcal{X}$: They can be images, sounds, videos, text documents, proteins, sequences of DNA bases, web pages, social network activities, sensors from industry, financial time series, etc. Set $\mathcal{X}$ may thus have a variety of structures that can be leveraged. All learning methods presented in this textbook will use at 1 point a vector space representation of inputs, either by building an explicit mapping from $\mathcal{X}$ to a vector space, e.g., $\mathbb{R}^d$, or implicitly by using a notion of pairwise dissimilarity or similarity between pairs of inputs. Choice of these representations is highly domain-dependent. However, note:
      · common topologies are encountered in many diverse areas (e.g. sequences or 2D or 3D objects), & thus common tools are used, &
      · learning these representations is an active area of research (Chaps. 7 & 9).

      In this textbook, will primarily consider that inputs are d-dimensional vectors, with d potentially large, up to 10^6 or 10^9.
    * **Outputs** $y \in \mathcal{Y}$**.** Most classical examples are binary labels $\mathcal{Y} = \{0, 1\}$ or $\mathcal{Y} = \{\pm 1\}$, multicategory classification problems with $\mathcal{Y} = \{1, \ldots, k\}$, & classical regression with real responses/outputs $\mathcal{Y} = \mathbb{R}$. These will be main examples examined in most of book. Note, however: most of concepts extend to more general *structured prediction* setup, where more general structured outputs (e.g., graph prediction, visual scene analysis, source separation, ranking) can be considered (Chap. 13).

    Why difficult? Supervised learning is difficult (& thus interesting) for a variety of reasons:

* Label y may not be a deterministic function of x: Given $x \in \mathcal{X}$, outputs are noisy, i.e., y is a random function of x. When $y \in \mathbb{R}$, will often make simplifying "additive noise" assumption: $y = f(x) + \varepsilon$ with some zero-mean noise ε, but in general, only assume: there is a conditional distribution of y given x. This stochasticity is typically due to diverging views between labelers or dependence on random external unobserved quantities (i.e., $y = f(x, z)$, with z random & not observed, which is common, e.g., in medical applications, where need to predict a future occurrence of a disease based on limited information about patients).
* Prediction function f may be quite complex, highly nonlinear when $\mathcal{X}$ is a vector space, & even hard to define when $\mathcal{X}$ is not a vector space.
* Only a few x's are observed: thus need interpolation & potentially extrapolation (diagram for an illustration for $\mathcal{X} = \mathcal{Y} = \mathbb{R}$), & therefore overfitting (predicting well on training data but not as well on testing data) is always a possibility. Moreover, training observations may not be uniformly distributed in $\mathcal{X}$. In this book, they will be assumed to be random, but some analyzes will rely on deterministically located inputs to simplify some theoretical arguments.
* Input space $\mathcal{X}$ may be very large (i.e., with high dimension when this is a vector space). This leads to both computational issues (scalability) & statistical issues (generalization to unseen data). One usually refers to this problem as *curse of dimensionality*.
* There may be a weak link between training & testing distributions. I.e., data at training time can have different characteristics than data at testing time.
* Criterion for performance is not always well defined.

Main formalization. Most modern theoretical analyzes of supervised learning rely on a probabilistic formulation, i.e., see (x_i, y_i) as a realization of random variables. Criterion: to maximize expectation of some performance measure w.r.t. distribution of test data (in this book, *maximizing* performance will be obtained by *minimizing* a loss function). Main assumption: random variables (x_i, y_i) are independent & identically distributed (i.i.d.) with same distribution as testing distribution. In this book, ignore potential mismatch between train & test distributions (although this is an important research topic, as in most applications, training data are not i.i.d. from same distribution as test data).
A ML algorithm $\mathcal{A}$ is then a function that goes from a dataset (i.e., an element of $(\mathcal{X} \times \mathcal{Y})^n$) to a function from $\mathcal{X}$ to $\mathcal{Y}$. I.e., output of a ML algorithm is itself an algorithm.
Practical performance evaluation. In practice, do not have access to test distribution but samples from it. In most cases, data given to ML user are split into 3 parts:

* *Training set*, on which learning models will be estimated.
* *Validation set*, to estimate hyperparameters (all learning techniques have some) to optimize performance measure.
* *Testing set*, to evaluate performance of final chosen model.

In theory, test set can be used only once. In practice, this is unfortunately only sometimes the case. If test data are seen multiple times, estimation of performance on unseen data is overestimated.
Cross-validation is often preferred, to use a maximal amount of training data & reduce variability of validation procedure: available data are divided into k folds (typically $k = 5$ or 10), & all models are estimated k times, each time choosing a different fold as validation data, & averaging k obtained error measures. Cross-validation can be applied to any learning method, & its detailed theoretical analysis is an active area of research (see Arlot & Celisse, 2010, & many references therein).
"Debugging" a ML implementation is often an art: on top of commonly found bugs, learning method may not predict well enough with testing data. This is where theory can be useful to understand when a method is supposed to work or not: primary goal of this book.
Model selection. Most ML models have hyperparameters (e.g., regularization weight, size of model, number of parameters). To estimate them from data, common practical approach is to use validation approaches like those highlighted thus far. Also possible to use penalization techniques based on generalization bounds. These 2 approaches are analyzed in Sect. 4.6.
Random design vs. fixed design. What have described is often referred to as "random design" setup in statistics, where both x, y are assumed to be random & sample i.i.d. Common to simplify analysis by considering: input data $x_1, \ldots, x_n$ are deterministic, either because they are actually deterministic (e.g., equally spaced in input space $\mathcal{X}$) or by conditioning on them if they are actually random. This will be referred to as "fixed design" setting & studied precisely in context of least-squares regression in Chap. 3.
In context of fixed design analysis, error is evaluated "within-sample" (i.e., for same input points $x_1, \ldots, x_n$, but over new associated outputs). This explicitly removes difficulty of extrapolating to new inputs, hence a simplification in mathematical analysis.

◦ 2.2: Decision Theory. **Main question.** Tackle question: What is optimal performance, regardless of finiteness of training data? I.e., what should be done if we have a perfect knowledge of underlying probability distribution of data? Will thus introduce concepts of *loss function, risk, & Bayes predictor*.
Consider a fixed (testing) distribution $p_{(x,y)}$ on $\mathcal{X} \times \mathcal{Y}$, with marginal distribution $p_{(x)}$ on $\mathcal{X}$. Note: make no assumptions at this point on input space $\mathcal{X}$.
Will almost always use overload notation p, to denote $p_{(x,y)}$ & $p_{(x)}$, where context can always make definition unambiguous. E.g., when $f : \mathcal{X} \to \mathbb{R}, g : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$, have $\mathbb{E}[f(x)] = \int_{\mathcal{X}} f(x) \, \mathrm{d}p(x), \mathbb{E}[g(x,y)] = \int_{\mathcal{X}\times\mathcal{Y}} g(x,y) \, \mathrm{d}p(x,y)$.
Ignore measurability issues on purpose. Instead reader can look at Christmann & Steinwart (2008): *Support Vector Machines* for a more formal presentation.

* Supervised Learning Problems & Loss Functions. Consider a loss function $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ (often $\mathbb{R}_+$), where $l(y, z)$: loss of predicting z while true label is y.
Some authors swap y, z in def of loss. Some related research communities (e.g., economics) use concept of "utility," which is then maximized.
Loss function only concerns output space $\mathcal{Y}$ independent of input space $\mathcal{X}$. Main examples: each corresponding to a particular supervised learning problem (note: for each problem, different losses may be considered):
  - **Binary classification.** $\mathcal{Y} = \{0, 1\}$ (or often $\mathcal{Y} = \{\pm 1\}$, or, less, often, when seen as a subcase of multicategory situation below, $\mathcal{Y} = \{1, 2\}$); "0–1 loss" defined as $l(y, z) = 1_{y \neq z}$ is most commonly used, i.e., 0 if $y = z$ (no mistake), & 1 otherwise (mistake).
    Very common to mix 2 conventions $\mathcal{Y} = \{0, 1\}$ & $\mathcal{Y} = \{\pm 1\}$: double-check which convention is used when using toolboxes.
  - **Multicategory classification.** $\mathcal{Y} = \{1, \ldots, k\}, l(y, z) = 1_{y \neq z}$ (0–1 loss).
  - **Regression**: $\mathcal{Y} = \mathbb{R}, l(y, z) = (y - z)^2$ (square loss). Absolute loss $l(y, z) = |y - z|$ is often used for robust estimation (since penalty for large errors is smaller).
  - **Structured prediction.** while this textbook focuses primarily on 3 examples above, there are many practical problems where $\mathcal{Y}$ is more complicated, with associated algorithms & theoretical results. E.g., when $\mathcal{Y}\{0, 1\}^k$ (leading to multilabel classification), Hamming loss $l(y, z) = \sum_{j=1}^{k} 1_{y_j \neq z_j}$ is commonly used; also, ranking problems involve losses on permutations, see Chap. 13 for a detailed treatment.

  Throughout this textbook, will assume: loss function is given to us. Note: in practice, final user imposes loss function, as this is how models will be evaluated. Clearly, a single real number may not be enough to characterize entire prediction behavior. E.g., in binary classification, there are 2 types of errors, false positives & false negatives, which can be considered simultaneously. Since now have 2 performance measures, typically need a curve to characterize performance of a prediction function. This is precisely what receiver operating characteristic (ROC) curves are achieving (see, e.g., Bach et al., 2006, & references therein). For simplicity, stick to a single loss function l in this book.
  While loss function l will be used to define generalization performance in Sect. 2.2.2, for computational reasons, learning algorithms may explicitly minimize a different (but related) loss function, with better computational properties. This loss function used in training is often called a "surrogate." This will be studied in context of binary classification in Sect. 4.1, & more generally for structured prediction in Chap. 13.
* Risks. Given loss function $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, can define *expected risk* (also referred to as *generalization error*, or *testing error*) of a function $f : \mathcal{X} \to \mathcal{Y}$, as expectation of loss function between output y & prediction $f(x)$.

  Definition 1 (Expected risk)**.** *Given a prediction function $f : \mathcal{X} \to \mathcal{Y}$, a loss function $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, & a probability distribution p on $\mathcal{X} \times \mathcal{Y}$,* expected risk *of f is defined as*

$$\mathcal{R}(f) = \mathbb{E}[l(y, f(x))] = \int_{\mathcal{X} \times \mathcal{Y}} l(y, f(x)) \, \mathrm{d}p(x, y). \tag{79}$$

  Risk depend on distribution p on (x, y). Sometimes use notation $\mathcal{R}_p(f)$ to make it explicit. Expected risk is our main performance criterion in this textbook.

  Be careful with randomness, or lack thereof, of f: when performing learning from data, f will depend on random training data, not on testing data, & thus $\mathcal{R}(f)$ is typically random because of dependence on training data. However, as a function on functions, expected risk $\mathcal{R}$ is deterministic.
  Note: sometimes consider random predictions, i.e., for any x, output a distribution on y, & then risk is taken as expectation over randomness of outputs.
  Averaging loss on training data defines *empirical risk* or *training error*.

  Definition 2 (Empirical risk)**.** *Given a prediction function $f : \mathcal{X} \to \mathcal{Y}$, a loss function $l : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$, & data $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}, i = 1, \ldots, n,$* empirical risk *of f is defined as*

$$\widehat{\mathcal{R}}(f) = \frac{1}{n} \sum_{i=1}^{n} l(y_i, f(x_i)). \tag{80}$$

  Note: $\widehat{\mathcal{R}}$ is a random function on functions (& is often applied to random functions, with dependent randomness as both will depend on training data).
  Special cases. For classical losses defined earlier, expected & empirical risks have specific formulations:
  - **Binary classifications.** $\mathcal{Y} = \{0, 1\}$ (or often $\mathcal{Y} = \{\pm 1\}$), & $l(y, z) = 1_{y \neq z}$ (0–1 loss). Can express risk as $\mathcal{R}(f) = \mathbb{P}(f(x) \neq y)$. This is simply probability of making a mistake on testing data (error rate), while empirical risk is proportion of mistakes on training data.
    In practice, *accuracy*, which is 1 minus error rate, is often reported.
  - **Multicategory classification.** $\mathcal{Y} = \{1, \ldots, k\}$, & $l(y, z) = 1_{y \neq z}$ (0–1 loss). Can also express risk as $\mathcal{R}(f) = \mathbb{P}(f(x) \neq y)$. This is also probability of making a mistake (error rate).
  - **Regression.** $\mathcal{Y} = \mathbb{R}, l(y, z) = (y - z)^2$ (square loss). Risk is then equal to $\mathcal{R}(f) = \mathbb{E}[(y - f(x))^2]$, often referred to as "mean squared error."

∗ Bayes Risk & Bayes Predictor. Now have defined performance criterion for supervised learning (expected risk), main question tackle here: What is best prediction function f (regardless of training data)?
Using conditional expectation & its associated law of total expectation, have

$$\mathcal{R}(f) = \mathbb{E}[l(y, f(x))] = \mathbb{E}[\mathbb{E}[l(y, f(x))|x]], \tag{81}$$

which can be rewritten, for a fixed $x' \in \mathcal{X}$:

$$\mathcal{R}(f) = \mathbb{E}_{x' \sim p}[\mathbb{E}[l(y, f(x'))|x = x']] = \int_{\mathcal{X}} \mathbb{E}[l(y, f(x'))|x = x'] \, \mathrm{d}p(x'). \tag{82}$$

To distinguish between random variable x & a value it may take, use notation x'.
From conditional distribution given any $x' \in \mathcal{X}$ (i.e., $y|x = x'$), can define *conditional risk* for any $z \in \mathcal{Y}$ (it is a deterministic function of z & x'):

$$r(z|x') = \mathbb{E}[l(y, z)|x = x'], \tag{83}$$

which leads to

$$\mathcal{R}(f) = \int_{\mathcal{X}} r(f(x')|x') \, \mathrm{d}p(x'). \tag{84}$$

To find a minimizing function $f : \mathcal{X} \to \mathbb{R}$, 1st assume: set $\mathcal{X}$ is finite: in this situation, risk can be expressed as a sum of functions that depends on a *single* value of f, i.e., $\mathcal{R}(f) = \sum_{x' \in \mathcal{X}} r(f(x')|x')\mathbb{P}(x = x')$. Therefore, can minimize w.r.t. each $f(x')$ *independently*. Therefore, a minimizer of $\mathcal{R}(f)$ can be obtained by considering for any $x' \in \mathcal{X}$, function value $f(x')$ to be equal to a minimizer $z \in \mathcal{Y}$ of $r(z|x') = \mathbb{E}[l(y, z)|x = x']$. This extends beyond finite sets.
Minimizing expected risk w.r.t. a function f in a restricted set does not lead to such decoupling.

Proposition 1 (Bayes predictor & Bayes risk)**.** *Expected risk is minimized at a Bayes predictor $f_\star : \mathcal{X} \to \mathcal{Y}$, satisfying $\forall x' \in \mathcal{X}$, (2.1)*

$$f_\star(x') \in \arg\min_{z \in \mathcal{Y}} \mathbb{E}[l(y, z)|x = x'] = \arg\min_{z \in \mathcal{Y}} r(z|x'). \tag{85}$$

Bayes risk $\mathcal{R}^\star$ is risk fo all Bayes predictors & is equal to

$$\mathcal{R}^\star = \mathbb{E}_{x' \sim p}\left[\inf_{z \in \mathcal{Y}} \mathbb{E}[l(y, z)|x = x']\right]. \tag{86}$$

Chứng minh. Have $\mathcal{R}(f) - \mathcal{R}^\star = \mathcal{R}(f) - \mathcal{R}(f_\star) = \int_{\mathcal{X}} [r(f(x')|x') - \min_{z \in \mathcal{Y}} r(z|x')] \, \mathrm{d}p(x')$. □

Note:

(a) Bayes predictor is not always unique, but that all lead to same Bayes risk (e.g., in binary classification when $\mathbb{P}(y = 1|x) = \frac{1}{2}$)
(b) Bayes risk is usually nonzero (unless dependence between x, y is deterministic). Given a supervised learning problem, Bayes risk is optimal performance; define excess risk as deviation w.r.t. optimal risk.

Definition 3 (Excess risk)**.** Excess risk *of a function $f : \mathcal{X} \to \mathcal{Y}$ is equal to $\mathcal{R}(f) - \mathcal{R}^\star$ (always nonnegative).*

⇒ ML could be seen trivial: *given* distribution $y|x$ for any x, optimal predictor is known & given by (2.1). Difficulty: this distribution is unknown.

Special cases. For usual set of losses, can compute Bayes predictors in closed forms as follows:

· **Binary classification.** Bayes predictor for $\mathcal{Y} = \{\pm 1\}$ & $l(y, z) = 1_{y \neq z}$ is s.t.

$$f_*(x') \in \arg\min_{z \in \{\pm 1\}} \mathbb{P}(y \neq z|x = x') = \arg\min_{z \in \{\pm 1\}} 1 - \mathbb{P}(y = z|x = x') = \arg\min_{z \in \{\pm 1\}} \mathbb{P}(y = z|x = x'). \tag{87}$$

Optimal classifier will select most likely class given x'. Using notation $\eta(x') = \mathbb{P}(y = 1|x = x')$, then, if $\eta(x') > \frac{1}{2}, f_*(x') = 1$, while if $\eta(x') < \frac{1}{2}, f_*(x') = -1$. What happens for $\eta(x') = \frac{1}{2}$ is irrelevant, as expected error is same for 2 potential predictions.
Bayes risk is then equal to $\mathcal{R}^* = \mathbb{E}[\min\{\eta(x), 1 - \eta(x)\}]$, which in general is strictly positive (unless $\eta(x) \in \{0, 1\}$ almost surely – i.e., y is a deterministic function of x).
This extends directly to multiple categories $\mathcal{Y} = \{1, \ldots, k\}$, for $k \geq 2$, where have $f_*(x') \in \arg\max_{i \in \{1,\ldots,k\}} \mathbb{P}(y = i|x = x')$.
These Bayes predictors & risks are valid only for 0–1 loss. Less symmetric losses are common in applications (e.g., for spam detection) & would lead to different formulas (Exercise 2.1 & Chap. 13).

· **Regression.** Bayes predictor for $\mathcal{Y} = \mathbb{R}$ & $l(y, z) = (y - z)^2$ is s.t.[2]

$$f_*(x') \in \arg\min_{z \in \mathbb{R}} \mathbb{E}[(y - z)^2|x = x'] = \arg\min_{z \in \mathbb{R}} \left\{\mathbb{E}[(y - \mathbb{E}[y|x = x'])^2|x = x'] + (z - \mathbb{E}[y|x = x'])^2\right\}. \tag{88}$$

This leads to conditional expectation $f_\star(x') = \mathbb{E}[y|x = x']$, with a Bayes risk equal to expected conditional variance.

Problem 1. *Consider binary classification with $\mathcal{Y} = \{\pm 1\}$ with loss function $l(-1, -1) = l(1, 1) = 0$ & $l(-1, 1) = c_- > 0$ (cost of a false positive), $l(1, -1) = c_+ > 0$ (cost of a false negative). Compute a Bayes predictor at x as a function of $\mathbb{E}[y|x]$.*

[2] Use law of total variance: $\mathbb{E}[(y - a)^2] = \mathrm{var}(y) + (\mathbb{E}[y] - a)^2$ for any random variable y & constant $a \in \mathbb{R}$, which can be shown by expanding square.

Problem 2. *Consider a learning problem on $\mathcal{X} \times \mathcal{Y}$, with $\mathcal{Y} = \mathbb{R}$ & absolute loss defined as $l(y,z) = |y-z|$. Compute a Bayes predictor $f_* : \mathcal{X} \to \mathbb{R}$.*

Problem 3. *Consider a learning problem $\mathcal{X} \times \mathcal{Y}$, with $\mathcal{Y} = \mathbb{R}$ & "pinball" loss $l(y,z) = \alpha(y-z)_+ + (1-\alpha)(z-y)_+$, for $\alpha \in (0,1)$. Compute a Bayes predictor $f_* : \mathcal{X} \to \mathbb{R}$. Provide an interpretation in terms of quantiles.*

Problem 4. *Characterize Bayes predictors for regression with "ε-insensitive" loss defined as $l(y,z) = \max\{0, |y-z|-\varepsilon\}$. If for each x, y is supported in an interval of length $< 2\varepsilon$, what are Bayes predictors?*

Problem 5 (Inverting predictions)**.** *Consider binary classification problem with $\mathcal{Y} = \{\pm 1\}$ & 0–1 loss. Relate risk of a prediction f & to that of its opposite $-f$.*

Problem 6 ("Chance" predictions)**.** *Consider binary classification problems with 0–1 loss. What is risk of a random prediction rule where predict 2 classes with equal probabilities independent of input x? Address same question with multiple categories.*

Problem 7. *Consider a random prediction rule where predict from probability distribution of y given x. When is this achieving Bayes risk?*

- 2.3. Learning from Data. Decision theory framework outlined in Sect. 2.2, with notations summarized in Table 2.1: Summary of notions & notations presented in this chapter & used throughout book [Bac24]:
  - $\mathcal{X}$: Input space
  - $\mathcal{Y}$: Output space
  - p: Joint distribution on $\mathcal{X} \times \mathcal{Y}$
  - $(x_1, y_1, \ldots, x_n, y_n)$: Training data
  - $f : \mathcal{X} \to \mathcal{Y}$: Prediction function
  - $l(y,z)$: Loss function between output y & prediction z
  - $\mathcal{R}(f) = \mathbb{E}[l(y, f(x))]$: Expected risk of prediction function f
  - $\widehat{R}(f) = \frac{1}{n}\sum_{i=1}^n l(y_i, f(x_i))$: Empirical risk of prediction function f
  - $f_*(x')\arg\min_{z\in\mathcal{Y}}\mathbb{E}[l(y,z)|x=x']$: Bayes prediction at x'
  - $\mathcal{R}^* = \mathbb{E}_{x'\sim p}\inf_{z\in\mathcal{Y}}\mathbb{E}[l(y,z)|x=x']$: Bayes risk

  gives a test performance criterion & optimal predictors, but it depends on full knowledge of test distribution p. Now briefly review how we can obtain good prediction functions from training data, i.e., data sampled i.i.d. from same distribution. 2 main classes of prediction algorithms will be studied in this textbook: (1) Local averaging (Chap. 6). (2) Empirical risk minimization (Chaps. 3, 4, 7–9, 11–13).

  Note: there are prediction algorithms that do not fit precisely into 1 of these 2 categories, e.g. boosting or ensemble classifiers (which perform several empirical risk minimizations, in series or parallel, see Chap. 10). Moreover, some situations do not fit classical i.i.d. framework, e.g. in online learning (see Chap. 11). Finally, consider probabilisitic methods in Chap. 14, which rely on a different principle.
  - 2.3.1. Local Averaging. Goal: to approximate/emulate (bắt chước) Bayes predictor (e.g., $f_*(x') = \mathbb{E}[y|x=x']$ for least-squares regression, or $f_*(x') = \arg\max_{z\in\mathcal{Y}}\mathbb{P}(y=z|x=x')$ for classification with 0–1 loss) from empirical data. This is often done by explicit or implicit estimation of conditional distribution by *local averaging* (k-nearest neighbors, which is used as primary example for this chapter; Nadaraya–Watson estimators; or decision trees). Briefly outline here main properties for 1 instance of these algorithms, see Chap. 6 for details.

    k-nearest-neighbor classifier. Given n observations $(x_1, y_1), \ldots, (x_n, y_n)$ where $\mathcal{X}$ is a metric space, $\mathcal{Y} \in \{\pm 1\}$, a new point x^{test} is classified by a majority vote among k-nearest neighbors of x^{test}.

    Consider 3-nearest-neighbor classifier on a particular testing point (which will be predicted as 1):
    - Pros: (1) no optimization or training, (2) often easy to implement, & (3) can get very good performance in low dimensions (in particular for nonlinear dependences between x, y).
    - Cons: (1) slow at query time: must pass through all training data at each testing point (there are algorithmic tools to reduce complexity; see Chap. 6); (2) bad for high-dimensional data (because of curse of dimensionality; more on this in Chap. 6); (3) choice of local distance function is crucial; (4) choice of width hyperparameters (or k) has to be performed.
    - Plot of training errors & testing errors as functions of k for a typical problem. When k is too large, there is *underfitting* (learned function is too close to a constant, which is too simple), while for k too small, there is *overfitting* (there is a strong discrepancy between testing & training errors).

    Problem 8. *How would curve move when n increases (assuming same balance between classes)?*
  - 2.3.2. Empirical Risk Minimization. Consider a parametrized family of prediction functions (often referred to as *models*) $f_\theta : \mathcal{X} \to \mathcal{Y}$ for $\theta \in \Theta$ (typically a subset of a vector space). This class of learning methods aims at minimizing empirical risk w.r.t. $\theta \in \Theta$:

$$\widehat{\mathcal{R}}(f_\theta) = \frac{1}{n}\sum_{i=1}^n l(y_i, f_\theta(x_i)). \tag{89}$$

    This defines an estimator $\hat{\theta} \in \arg\min_{\theta\in\Theta}\widehat{\mathcal{R}}(f_\theta)$, & thus a prediction function $f_{\hat{\theta}} : \mathcal{X} \to \mathcal{Y}$.

    Most classic example: linear least-squares regression (studied thoroughly in Chap. 3), where minimize $\frac{1}{n}\sum_{i=1}^n (y_i - \theta^\top\varphi(x_i))^2$, & f is linear in some feature vector $\varphi(x) \in \mathbb{R}^d$ (there is no need for $\mathcal{X}$ to be a vector space). Vector $\varphi(x)$ can

be quite large (or even implicit, like in kernel methods, see Chap. 7). Other examples include neural networks (Chap. 9).

· Pros: (1) can be relatively easy to optimize (e.g., least-squares with its simple derivation & numerical algebra; see Chap. 3), many algorithms are available (primarily based on gradient descent; see Chap. 5); & (2) can be applied in any dimension (if a suitable feature vector is available).

· Cons: (1) can be relatively hard to optimize when optimization formulation is not convex (e.g., neural networks); (2) need a suitable feature vector for linear methods; (3) dependence on parameters can be complex (e.g., neural networks); (4) need some capacity control to avoid overfitting; & (5) require to parameterize functions with values in $\{0,1\}$ (see Chap. 4 for use of convex surrogates).

Risk decomposition. Material in this section will be studied further in more detail in Chap. 4.

· Risk decomposition in estimation error + approximation error: given any $\hat{\theta} \in \Theta$, can write excess risk of $f_{\hat{\theta}}$ as

$$\mathcal{R}(f_{\hat{\theta}}) - \mathcal{R}^\star = \left\{ \mathcal{R}(f_{\hat{\theta}}) - \inf_{\theta' \in \Theta} \mathcal{R}(f_{\theta'}) \right\} + \left\{ \inf_{\theta' \in \Theta} \mathcal{R}(f_{\theta'}) - \mathcal{R}^* \right\} = \text{estimation error} + \text{approximation error}. \tag{90}$$

Approximation error $\{\inf_{\theta' \in \Theta} \mathcal{R}(f_{\theta'}) - \mathcal{R}^*\}$ is always nonnegative, does not depend on chosen $f_{\hat{\theta}}$, & depends only on class of functions parametrized by $\theta \in \Theta$. It is thus always a deterministic quantity, which characterizes modeling assumptions made by chosen class of functions. When Θ grows, approximation error goes down to 0 if arbitrary functions can be approximated arbitrarily well by functions f_θ. It is also independent of number n of observations.
Estimation error $\left\{\mathcal{R}(f_{\hat{\theta}}) - \inf_{\theta' \in \Theta} \mathcal{R}(f_{\theta'})\right\}$ is also always nonnegative & is typically random because function $f_{\hat{\theta}}$ is random. It typically decreases in n & increases when Θ grows.
Overall, typical error curves look like this Fig: Size of Θ–Errors plot.

· Typically, see in later chaps: estimation error is often decomposed as follows, for θ' a minimizer on Θ of expected risk $\mathcal{R}(f_{\theta'})$:

$$\mathcal{R}(f_{\hat{\theta}}) - \mathcal{R}(f_{\theta'}) = \{\mathcal{R}(f_{\hat{\theta}}) - \widehat{\mathcal{R}}(f_{\hat{\theta}})\} + \{\widehat{\mathcal{R}}(f_{\hat{\theta}}) - \widehat{\mathcal{R}}(f_{\theta'})\} + \{\widehat{\mathcal{R}}(f_{\theta'} - \mathcal{R}(f_{\theta'})\} \leq 2 \sup_{\theta \in \Theta} |\widehat{\mathcal{R}}(f_\theta) - \mathcal{R}(f_\theta)| + \text{empirical optimization error} \tag{91}$$

where empirical optimization error is $\sup_{\theta \in \Theta} \{\widehat{\mathcal{R}}(f_{\hat{\theta}}) - \widehat{\mathcal{R}}(f_\theta)\}$ (it is equal to 0 for exact empirical risk minimizers, but it is not when using optimization algorithms from Chap. 5 in practice). Uniform deviation defined as $\sup_{\theta \in \Theta} |\widehat{\mathcal{R}}(f_\theta) - \mathcal{R}(f_\theta)|$ grows with "size" of Θ (e.g., number or norm of parameters), & usually decays with n. See more details in Chap. 4.

Capacity control. To avoid overfitting, need to make sure: set of allowed functions is not too large by typically reducing number of parameters or by restricting norm of predictors (thus by lowering "size" of Θ): this leads to constrained optimization & still allows for risk decompositions as done previously.
Capacity control can also be done by *regularization*, i.e., by minimizing

$$\widehat{\mathcal{R}}(f_\theta) + \lambda \Omega(\theta) = \frac{1}{n} \sum_{i=1}^n l(y_i, f_\theta(x_i)) + \lambda \Omega(\theta), \tag{92}$$

where $\Omega(\theta)$ controls complexity of f_θ. Main example: ridge regression

$$\min_{\theta \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^n (y_i - \theta^\top \varphi(x_i))^2 + \lambda \|\theta\|_2^2. \tag{93}$$

Regularization is often easier for optimization but harder to analyze (see Chaps. 4–5).
There is a difference between parameters (e.g., θ) learned on training data & hyperparameters (e.g., λ) estimated on validation data.

Examples of approximations by polynomials in 1D regression. Consider $(x, y) \in \mathbb{R}^2$, with prediction functions that are polynomials of order k, from $k = 0$ (constant functions) to $k = 14$ (this corresponds to linear regression with $f_\theta(x)$of form $\theta^\top \varphi(x)$, where $\varphi(x) = (1, x, \ldots, x^k)^\top \in \mathbb{R}^{k+1}$). For each k, model has $k + 1$ parameters. Training error (using square loss) is minimized with $n = 20$ observations. Data were generated with inputs uniformly distributed on $[-1, 1]$ & outputs as quadratic function $f(x) = x^2 - \frac{1}{2}$ of inputs plus some independent additive noise (Gaussian with standard deviation $\frac{1}{4}$). As shown in Fig. 2.1: Polynomial regression with increasing orders k. Plots of estimated functions in red, with training & testing errors. Bayes prediction function $f_*(x) = \mathbb{E}[y|x]$ is plotted in blue (same for all plots). Fig. 2.2: Polynomial regression with increasing orders. Plots of training & testing errors with error bars (computed as standard deviations obtained from 32 replications), together with Bayes error. Note: variance is increasing with order k., training error monotonically decreases in k while testing error goes down & then up. Note: strong overfitting when k is large (3d row in Fig. 2.1).

◦ 2.4. Statistical Learning Theory. Goal of learning theory: to provide some guarantees of performance on unseen data given some properties of learning problem. A common assumption: data $\mathcal{D}_n(p) = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ are obtained as i.i.d. observations from some unknown distribution p from some family $\mathcal{P}$. Family $\mathcal{P}$ of probability distributions on (x, y) encapsulates properties of learning problem & may consider conditions on distributions of inputs or on conditional distributions of outputs given inputs.

As seen earlier, algorithm $\mathcal{A}$ is a mapping from $\mathcal{D}_n(p)$ (for any n) to a function from $\mathcal{X} \to \mathcal{Y}$. Expected risk depends on probability distribution $p \in \mathcal{P}$, as $\mathcal{R}_p(f)$. Goal: to find $\mathcal{A}$ s.t. excess expected risk $\mathcal{R}_p(\mathcal{A}(\mathcal{D}_n(p))) - \mathcal{R}_p^*$ is small, where $\mathcal{R}_p^*$: Bayes risk (which depends on joint distribution p), assuming: $\mathcal{D}_n(p)$ is sampled from p, but without knowing which $p \in \mathcal{P}$ is considered. Moreover, risk is random because $\mathcal{D}_n(p)$ is random.

* 2.4.1. Measures of Performance. There are several ways of dealing with randomness of expected risk of estimator to obtain a criterion:
  - *Expected error*: measure performance as $\mathbb{E}[\mathcal{R}_p(\mathcal{A}(\mathcal{D}_n(p)))]$, where expectation is w.r.t. training data. Algorithm $\mathcal{A}$ is called *consistent in expectation* for distribution p, if $\mathbb{E}[\mathcal{R}_p(\mathcal{A}(\mathcal{D}_n(p)))] - \mathcal{R}_p^* \to 0$ when $n \to \infty$. In this book, primarily use this notion of consistency.
  - *Probably approximately correct (PAC) learning*: for a given $\delta \in (0,1), \varepsilon > 0$: $\mathbb{P}(\mathcal{P}_p(\mathcal{A}(\mathcal{D}_n(p))) - \mathcal{R}_p^* \leq \varepsilon) \geq 1 - \delta$. Goal of learning theory in this framework: to find an ε as small as possible (typically as a function of δ, n). Notion of PAC consistency corresponds, for any $\varepsilon > 0$, to have such an inequality for each n & a sequence $\delta_n \to 0$.
* 2.4.2. Notions of Consistency over Classes of Problems. An algorithm is called *universally consistent* (in expectation) if for all probability distributions $p = p_{(x,y)}$ on (x, y), algorithm $\mathcal{A}$ is consistent in expectation for distribution p.
  Be careful with order of quantifiers: convergence speed of excess risk toward 0 will depend on p. See "no free lunch" theorem in Sect. 2.5 that highlights: a uniform rate over all distributions is hopeless.
  Most often, want to study uniform consistency within a class $\mathcal{P}$ of distributions satisfying some regularity properties (e.g., inputs live in a compact space or dependence between y, x has at most some complexity, e.g., linear in some feature vector or with a certain number of bounded derivatives).
  Thus aim at finding algorithm $\mathcal{A}$ s.t. $\sup_{p \in \mathcal{P}} \left\{\mathbb{E}[\mathcal{R}_p(\mathcal{A}(\mathcal{D}_n(p)))] - \mathcal{R}_p^*\right\}$ is as small as possible. So-called *minimax risk* is equal to $\inf_{\mathcal{A}} \sup_{p \in \mathcal{P}} \left\{\mathbb{E}[\mathcal{R}_p(\mathcal{A}(\mathrm{D}_n(p)))] - \mathcal{R}_p^*\right\}$. This is typically a function of sample size n & parameters that are characteristic of $\mathcal{X}, \mathcal{Y}$ & allowed set of problems $\mathcal{P}$ (e.g., dimension of $\mathcal{X}$, model size). To compute estimates of minimax risk, several techniques exist:
  - Upper-bounding optimal excess risk: 1 given algorithm with a convergence proof provides an upper bound: Main focus of this book.
  - Lower-bounding optimal excess risk: in some setups, possible to show: infimum over all algorithms is $>$ a certain quantity. See Chap. 15 for a description of techniques to obtain such lower bounds. Machine learners are happy when upper bounds & lower bounds match (up to constant factors).

  Nonasymptotic vs. asymptotic analysis. Theoretical results in learning theory can be *nonasymptotic*, with an upper bound with explicit dependence on all quantities; bound is then valid for all n, even if it is sometimes vacuous (e.g., a bound > 1 for a loss uniformly bounded by 1).
  Analysis can also be *asymptotic*, where, e.g., $n \to \infty$ & limits are taken. Alternatively, several quantities can be made to grow simultaneously, which is common in random matrix theory, where dimension d of features & number n of observations both $\to \infty$, with a ratio tending to a constant (see, e.g., Potters & Bouchaud, 2020). See also discussion in Sect. 4.7.
  Key aspect here is (arguably) how these rates depend on problem. Specifically, choice of in expectation vs. in high probability, or asymptotic vs. nonasymptotic, does not really matter as long as problem parameters explicitly appear.

◦ 2.5. "No Free Lunch" Theorem. Although it may be tempting to define optimal learning algorithm that works optimally for all distributions, this is impossible. I.e., learning is only possible with assumptions. See Chap. 7 of Devroye et al. (1996) for more details.
Prop. 2.2 shows: for any algorithm, for a fixed n, there is a data distribution that makes algorithm useless (with a risk that is the same as chance level).

Proposition 2 (No free lunch–fixed n)**.** *Consider binary classification with* 0–1 *loss & $\mathcal{X}$ infinite. Let $\mathcal{P}$ denote set of all probability distributions on $\mathcal{X} \times \{0,1\}$. For any $n > 0$ & any learning algorithm $\mathcal{A}$,* $\sup_{p \in \mathcal{P}} \left\{\mathbb{E}[\mathcal{R}_p(\mathcal{A}(\mathcal{D}_n(p)))] - \mathcal{R}_p^*\right\} \geq \frac{1}{2}$.

Main ideas of proof: (1) to construct a probability distribution supported on k elements in $\mathbb{N}$, where k is large compared to n (which is fixed), & to show: knowledge of n labels does not imply doing well on all k elements, & (2) to choose parameters of this distribution (binary vector r defined next) with largest possible expected risk & compare this worst performance to performance obtained by a random choice of parameters.
A caveat (cảnh báo) of Prop. 2.2: hard distribution used in proof above may depend on n (from proof, it takes k values, with $k \to \infty$ fast enough compared with n). Following Prop. (Thm. 7.2 from Devroye et al., 1996) is much "stronger," as it more convincingly shows: learning can be arbitrarily slow without assumption (note: earlier one is not a corollary of later one).

Proposition 3 (No free lunch–sequence of errors)**.** *Consider a binary classification problem with* 0–1 *loss, with $\mathcal{X}$ infinite. Let $\mathcal{P}$ denote set of all probability distributions on $\mathcal{X} \times \{0,1\}$. For any decreasing sequence $a_n \to 0$ & s.t. $a_1 \leq \frac{1}{16}$, for any learning algorithm $\mathcal{A}$, there exists $p \in \mathcal{P}$ s.t. $\mathbb{E}[\mathcal{R}_p(\mathcal{A}(\mathcal{D}_n(p)))] - \mathcal{R}_p^* \geq a_n$, $\forall n \geq 1$.*

◦ 2.6. Quest for Adaptivity. As seen in Sect. 2.5, no method can be universal & achieve a good convergence rate on all problems. However, such negative results consider classes of problems that are arbitrarily large. In this textbook, consider reduced sets of learning problems by considering $\mathcal{X} = \mathbb{R}^d$ & putting restrictions on target function f_* based on smoothness &/or dependence on an unknown low-dimensional projection. I.e., most general set of functions will be set of Lipschitz-continuous functions, for which optimal rate will be essentially proportional to $O(n^{-\frac{1}{d}})$, typical of curse of dimensionality (as required number n of observations to reach a given precision is exponential in d). No method can beat this–not k-nearest-neighbors, not kernel methods, & not even neural networks (see lower bounds on performance in Chap. 15).

When target function is smoother (i.e., with all derivatives up to order m bounded), then will see: kernel methods (Chap. 7) & neural networks (Chap. 9), with proper choice of regularization parameter, will lead to optimal rate of $O(n^{-\frac{m}{d}})$.
When target function moreover depends only on a r-dimensional linear projection, neural networks (if optimization problem is solved correctly) will have extra ability to lead to rates of form $O(n^{-\frac{m}{r}})$ instead of $O(n^{-\frac{m}{d}})$. This is not the case for kernel methods (see Chap. 9).
Note: another form of adaptivity, which is often considered, may apply in situations where input data lie on a submanifold of $\mathbb{R}^d$ (e.g., an affine subspace), where for most methods presented in this textbook, adaptivity is obtained. In convergence rate, d can be replaced by dimension of subspace (or submanifold) where data live. For more, see Kpotufe (2011) for k-nearest neighbors, & Hamm & Steinwart (2021) for kernel methods. See more details in https://francisbach.com/quest-for-adaptivity/, as well as Chaps. 7 & 9 for detailed results regarding adaptivity for kernel methods & neural networks.

- 2.7. Beyond Supervised Learning. This textbook focuses primarily on traditional supervised learning paradigm, with i.i.d. data & where training & testing distributions match. Many applications require extensions to this basic framework, which also lead to many interesting theoretical developments that are out of scope. Next, present briefly some of these extensions, with references for further reading.
Unsupervised learning. While in supervised learning, both inputs & outputs (e.g., labels) are observed, & main goal: to model how output depends on input, in unsupervised learning only inputs are given. Goal: to find some structure within data – e.g., an affine subspace around which data live for principal component analysis (PCA, studied in Sect. 3.9), separation of data in several groups (for clustering), or identification of an explicit latent variable model (e.g. with matrix factorization). New representation of data is typically either used for visualization (then, with 2D or 3D), or for reducing dimension before applying a supervised learning algorithm.
While supervised learning relied on an explicit decision-theoretic framework, not always clear how to characterize performance & perform evaluation in unsupervised learning; each method typically has an ad hoc empirical criterion, e.g. reconstruction of data, full or partial (like in self-supervised learning); or log-likelihood when probabilistic models are used (see Chap. 14), in particular graphical models (Bishop, 2006; Murphy, 2012). Often, immediate representations are used for subsequent processing (see, e.g., Goodfellow et al., 2016).
Theoretical guarantees can be obtained for sampling behavior & recovery of specific structures when assumed (e.g., for clustering or dimension reduction), with a variety of results in manifold learning, matrix factorization methods e.g. K-means, PCA, or sparse dictionary learning (Mairal et al., 2014), outlier/novelty detection (Pimentel et al., 2014), or independent component analysis (Hyvärinen et al., 2001).
Semisupervised learning. Intermediate situation between supervised & unsupervised, with typically a few labeled examples & typically many unlabeled examples. Several frameworks exist based on various assumptions (Chapelle et al., 2010; van Engelen & Hoos, 2020).
Active learning. A similar setting as semisupervised learning, but user can choose which unlabeled point to label to maximize performance over new labels are obtained. Selection of samples to label is often done by computing some form of uncertainty estimation on unlabeled data points (see, e.g., Settles, 2009).
Online learning. Mostly in a supervised setting, this framework allows us to go beyond training/testing splits, where data are acquired & predictions are made on fly, with a criterion that takes into account sequential nature of learning. See Cesa-Bianchi & Lugosi (2006), Hazan (2022), & Chap. 11.
Reinforcement learning. On top of sequential nature of learning already present in online learning, predictions may influence future sampling distributions; e.g., in situations where some agents interact with an environment (Sutton & Barto, 2018), with algorithms relying on similar concepts than optimal control (Liberzon, 2011).
Generative modeling. A key task in computer vision or natural language processing is to generate images or text documents based on simple "prompts." Goal: often to given an output that minimizes some loss, but rather to sample from a distribution that reflects natural variability of images & text, given prompt. Sampling from such high-dimensional distributions is a practical & theoretical challenge, where diffusion models prove particularly useful (see, e.g., Chan, 2024, & references therein).
- 2.8. Summary – Book Outline. Introduced main concepts, can give an outline of chapters of this book, separated into 3 parts.
Part I: Preliminaries contains Chap. 1 on mathematical preliminaries, this introductory chapter, & Chap. 3, on linear least-squares regression. Start with least-squares, as it allows introduction of main concepts of book, e.g. underfitting, overfitting, regularization, using only simple linear algebra, without need for more advanced analytic or probabilistic tools.
Part II: Generalization bounds for learning algorithms is dedicated to core concepts in learning theory & should be studied sequentially.
  - **Empirical risk minimization.** Chap. 4 is dedicated to methods based on minimization of potentially regularized or constrained regularized risk, with introduction of key concept of Rademacher complexity, which analyzes estimation errors efficiently. Convex surrogates for binary classification are also introduced to allow use of only real-valued prediction functions.
  - **Optimization.** Chap. 5 shows how gradient-based techniques can be used to approximately minimize empirical risk &, through stochastic gradient descent (SGD), obtain generalization bounds for finitely-parameterized linear models (which are linear in their parameters), leading to convex objective functions.
  - **Local averaging methods.** Chap. 6 is 1st chapter dealing with so-called "nonparametric" methods that can potentially

adapt to complex prediction functions. This class of methods explicitly builds a prediction function mimicking Bayes predictor (without any optimization algorithm), e.g., k-nearest-neighbor methods. These methods are classically subject to curse of dimensionality.

* **Kernel methods.** Chap. 7 presents most general class of linear models that can be infinite-dimensional & adapt to complex prediction functions. They are made computationally feasible using "kernel trick," & they still rely on convex optimization, so they lead to strong theoretical guarantees, particularly by adapting to smoothness of target prediction function.
* **Sparse methods.** While Chap. 7 focused on Euclidean or Hilbertian regularization techniques for linear models, Chap. 8 considers regularization by sparsity-inducing penalties e.g. l_1-norm or l_0-penalty, leading to high-dimensional phenomenon that learning is possible even with potentially exponentially many irrelevant variables.
* **Neural networks.** Chap. 9 presents a class of prediction functions that are not linearly parameterized, leading to nonconvex optimization problems, where obtaining a global optimum is not certain. Chap studies approximation & estimation errors, showing adaptivity of neural networks to smoothness & linear latent variables (in particular for nonlinear variable selection).

Part III: Special topics presents a series of chapters on special topics that can be read in essentially any order.

* **Ensemble learning.** Chap. 10 presents a class of techniques aiming at combining several predictors obtained from same model class but learned on slightly modified datasets. This can be done in parallel, e.g. in bagging techniques, or sequentially, e.g. in boosting methods.
* **From online learning to bandits.** (bọn cướp) Chap. 11 considers sequential decision problems within regret framework, focusing 1st on online convex optimization, then on 0th-order optimization (without access to gradients), & finally multiarmed bandits.
* **Overparameterized models.** Chap. 12 presents a series of results related to models with a large number of parameters (enough to fit training data perfectly) & trained with gradient descent (GD). Present implicit bias of GD in linear models toward minimum Euclidean norm solutions & then double descent phenomenon, before looking at implicit biases & global convergence for nonconvex optimization problems.
* **Structured prediction.** Chap. 13 goes beyond traditional regression & binary classification frameworks by 1st considering multicategory classification & then general framework of structured prediction, where output spaces can be arbitrarily complex.
* **Probabilistic methods.** Chap. 14 presents a collection of results related to probabilistic modeling, highlighting: probabilistic interpretations can sometimes be misleading but also naturally lead to model selection frameworks through Bayesian inference & PAC–Bayesian analysis.
* **Lower bounds on generalization & optimization errors.** While most of book is dedicated to obtaining upper bounds on generalization or optimization errors of our algorithms, Chap. 15 considers lower bounds on such errors, showing how many algorithms presented in this book are, in fact, optimal for a specific class of learning or optimization problems.

- 3. Linear Least-Squares Regression. Chapter Summary: *Ordinary least-squares estimator*: Least-squares regression with linearly parameterized predictors leads to a linear system of size d (number of predictors). *Guarantees in fixed design setting with no regularization*: When inputs are assumed deterministic & $d < n$, excess risk $= \frac{\sigma^2 d}{n}$, where σ^2: prediction noise variance. *Ridge regression*: With l_2-regularization, excess risk bounds become dimension independent & allow high-dimensional feature vectors where $d > n$. *Guarantees in random design setting*: Although they are harder to show, they have a similar form. *Lower bound of generalization error*: Under well-specification, rate $\frac{\sigma^2 d}{n}$ cannot be improved.
  - 3.1. Introduction. Introduce & analyze linear least-squares regression, a tool that can be traced to Legendre (1805) & Gauss (1809). See https://en.wikipedia.org/wiki/Least_squares for an interesting discussion & claim: Gauss had known about it already in 1795. *Why should we study linear least-squares regression? Has there not been any progress since 1805?* A few reasons:
    * It already captures many of concepts in learning theory, e.g. bias-variance trade-off, as well as dependence of generalization performance on underlying dimension of problem with no regularization, or on dimensionless quantities when regularization is added.
    * Because of its simplicity, many results can be easily derived without need for complicated mathematics, both in terms of algorithms & statistical analysis (simple linear algebra for simplest linear algebra for simplest results in fixed design setting).
    * Using nonlinear features, it can lead to arbitrary nonlinear predictions (see discussion of kernel methods in Chap. 7).

    In subsequent chapters, will extend many of these results beyond least-squares regression with proper additional mathematical tools.
  - 3.2. Least-Squares Framework. Recall goal of supervised ML from Chap. 2: given some training data composed of observations $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}, i = 1, \dots, n$, which are pairs of inputs/outputs, sometimes referred to as features/responses. Given $x \in \mathcal{X}$, goal: to predict $y \in \mathcal{Y}$ (testing data) with a *regression* function f s.t. $y \approx f(x)$. Assume $\mathcal{Y} = \mathbb{R}$ & use square loss $l(y, z) = (y - z)^2$, known from Chap. 2: optimal predictor is $f_*(x) = \mathbb{E}[y|x]$ (see Sect. 2.2.3).
    In Chap. 3, consider empirical risk minimization for regression problems. Choose a parameterized family of prediction functions (often referred to as "models") $f_\theta : \mathcal{X} \to \mathcal{Y} = \mathbb{R}$ for some parameter $\theta \in \Theta$ & minimize empirical risk

$\frac{1}{n}\sum_{i=1}^n (y_i - f_\theta(x_i))^2$, leading to estimator $\hat{\theta} \in \arg\min_{\theta\in\Theta} \frac{1}{n}\sum_{i=1}^n (y_i - f_\theta(x_i))^2$. Note: in most cases, Bayes predictor f_* does not belong to class of functions $\{f_\theta,\ \theta \in \Theta\}$, i.e., model is said to be *misspecified*.

Least-squares regression can be carried out with parameterizations of function f_θ that may be nonlinear in parameter θ (e.g. for neural networks in Chap. 9). In this chapter, will consider only situations where $f_\theta(x)$ is linear in θ, which is thus assumed to live in a vector space, taken to be $\mathbb{R}^d$ for simplicity.

Being linear in x or linear in θ is different!

While assume linearity in parameter θ, nothing forces $f_\theta(x)$ to be linear in input x. In fact, even concept of linearity may be meaningless if $\mathcal{X}$ is not a vector space. If $f_\theta(x)$ is linear in $\theta \in \mathbb{R}^d$, then it has to be a linear combination of form $f_\theta(x) = \sum_{i=1}^d \alpha_i(x)\theta_i$, where $\alpha_i : \mathcal{X} \to \mathbb{R}, i = 1, \dots, d$, are d functions. By concatenating them in a vector $\varphi(x) \in \mathbb{R}^d$ where $\varphi(x)_i = \alpha_i(x)$, get representation $f_\theta(x) = \varphi(x)^\top\theta$. Vector $\varphi(x) \in \mathbb{R}^d$ is typically called *feature vector*, which assume to be known (i.e., given to us & can be computed explicitly when needed). Thus consider minimizing empirical risk:

$$\widehat{\mathcal{R}}(\theta) = \frac{1}{n}\sum_{i=1}^n (y_i - \varphi(x_i)^\top\theta)^2. \tag{94}$$

When $\mathcal{X} \subset \mathbb{R}^d$, can make extra assumptions: f_θ is an affine function in x, which can be obtained through $\varphi(x) = \begin{pmatrix} x \\ 1 \end{pmatrix} = (x^\top, 1)^\top \in \mathbb{R}^{d+1}$. Another classical assumption is to consider vectors $\varphi(x)$ composed of monomials (so that prediction functions are polynomials, as done in experiments in Sect. 3.5.2). See in Chap. 7: *Kernel methods*: can consider infinite-dimensional features.

Matrix notation. Cost function (153) can be rewritten in matrix notation. Let $y = (y_1, \dots, y_n)^\top \in \mathbb{R}^n$: vector of outputs (sometimes called *response vector*), & $\Phi \in \mathbb{R}^{n\times n}$: matrix of inputs, whose rows are $\varphi(x_i)^\top$, called *design matrix* or *data matrix*. In this notation, empirical risk is:

$$\widehat{\mathcal{R}}(\theta) = \frac{1}{n}\|y - \Phi\theta\|_2^2, \tag{95}$$

where $\|\alpha\|_2^2 = \sum_{j=1}^d \alpha_j^2$: squared l_2-norm of α.

Sometimes tempting at 1st to avoid matrix notation. Strongly advise against it, as it leads to lengthy & error-prone formulas.

- 3.3. Ordinary Least-Squares Estimator. Assume: matrix $\Phi \in \mathbb{R}^{n\times d}$ has full column rank (i.e., rank of Φ is d). In particular, problem is said to be "overdetermined," & must have $d \le n$, i.e., more observations than feature dimension. Equivalently, assume: $\Phi^\top\Phi \in \mathbb{R}^{d\times d}$ is invertible.

  Definition 4 (OLS)**.** *When Φ has full column rank, minimizer of (95) is unique & called* ordinary least-squares (OLS) estimator*.*

  - 3.3.1. Closed-Form Solution. Since objective function is quadratic, gradient will be linear, & zeroing it will lead to a closed-form solution through a linear system.

    Proposition 4. *When Φ has full column rank, OLS estimator exists & is unique, & is given by $\hat{\theta} = (\Phi^\top\Phi)^{-1}\Phi^\top y$. Denote noncentered*[3] *empirical covariance matrix as $\widehat{\Sigma} = \frac{1}{n}\Phi^\top\Phi \in \mathbb{R}^{d\times d}$; have $\hat{\theta} = \frac{1}{n}\widehat{\Sigma}^{-1}\Phi^\top y$.*

    Coercive = going to ∞ at ∞. Condition $\widehat{\mathcal{R}}'(\hat{\theta}) = 0$ gives *normal equation* $\Phi^\top\Phi\hat{\theta} = \Phi^\top y$. Multidimensional linear normal equations has a unique solution: $\hat{\theta} = (\Phi^\top\Phi)^{-1}\Phi^\top y$, which shows uniqueness of minimizer of $\widehat{\mathcal{R}}$, as well as its closed-form expression. Another way to show uniqueness of minimizer is by showing: $\widehat{\mathcal{R}}$ is strongly convex since Hessian $\widehat{\mathcal{R}}''(\theta) = 2\widehat{\Sigma}$ is invertible $\forall\theta \in \mathbb{R}^d$ (convexity is studied in Chap. 5). For readers worried about carrying a factor of 2 in gradients, will sue an additional factor $\frac{1}{2}$ in chaps on optimization.

  - 3.3.2. Geometric Interpretation. OLS estimator has a natural geometric interpretation.

    Proposition 5. *Vector predictions $\Phi\hat{\theta} = \Phi(\Phi^\top\Phi)^{-1}\Phi^\top y$ is orthogonal projection of $y \in \mathbb{R}^n$ onto $\mathrm{im}\Phi \subset \mathbb{R}^n$, column space of Φ.*

    Can thus interpret OLS estimation as doing following: 1. Compute projection $\bar{y}$ of y onto image of Φ. 2. Solve linear system $\Phi\theta = \bar{y}$, which has a unique solution.

  - 3.3.3. Numerical Resolution. While closed-form $\hat{\theta} = (\Phi^\top\Phi)^{-1}\Phi^\top y$ is convenient for analysis, inverting $\Phi^\top\Phi$ is sometimes unstable & has a large computational cost when d is large. Following methods are usually preferred.

    QR **factorization.** QR decomposition factorizes matrix Φ as $\Phi = QR$, where $Q \in \mathbb{R}^{n\times n}$ has orthonormal columns, i.e., $Q^\top Q = I, R \in \mathbb{R}^{d\times d}$ is upper triangular (see Golub & Loan, 1996). Computing a QR decomposition is faster & more stable than inverting a matrix. Then have $\Phi^\top\Phi = R^\top Q^\top QR = R^\top R$, & R: Cholesky factor of positive semidefinite matrix $\Phi^\top\Phi \in \mathbb{R}^d$. Since R is invertible, one then has

    $$(\Phi^\top\Phi)\hat{\theta} = \Phi^\top y \Leftrightarrow R^\top Q^\top QR\hat{\theta} = R^\top Q^\top y \Leftrightarrow R^\top R\hat{\theta} = R^\top Q^\top y \Leftrightarrow R\hat{\theta} = Q^\top y.$$

    Only remains to solve a triangular linear system, which is easy. Overall running time complexity remains $O(d^3)$. Conjugate gradient algorithm can also be used (see Golub & Loan, 1996, for details).

[3] *Centered* covariance matrix would be $\frac{1}{n}\sum_{i=1}^n [\varphi(x_i) - \hat{\mu}][\varphi(x_i) - \hat{\mu}]^\top$, where $\hat{\mu} = \frac{1}{n}\sum_{i=1}^n \varphi(x_i) \in \mathbb{R}^d$ is empirical mean, while consider $\widehat{\Sigma} = \frac{1}{n}\sum_{i=1}^n \varphi(x_i)\varphi(x_i)^\top$.

Gradient descent. Can bypass need for matrix inversion or factorization using gradient descent (GD). It consists in approximately minimizing $\widehat{\mathcal{R}}$ by taking an initial point $\theta_0 \in \mathbb{R}^d$ & iteratively going toward minimizer by following opposite of gradient: $\theta_t = \theta_{t-1} - \gamma \widehat{\mathcal{R}}'(\theta_{t-1})$ for $t \geq 1$, where $\gamma > 0$: step size. When these iterates converge, it is toward OLS estimator since a fixed-point θ satisfies $\widehat{\mathcal{R}}'(\theta) = 0$. Study such algorithms in Chap. 5, with running-time complexities going down to linear in d, e.g., $O(nd)$.

- 3.4. Statistical Analysis of Ordinary Least-Squares.

- 4. Empirical Risk Minimization.
- 5. Optimization for Machine Learning.
- 6. Local Averaging Models.
- 7. Kernel Methods.
- 8. Sparse Methods.
  - 8.6. Conclusion. In this chapter, considered sparse methods based on penalization by l_0- or l_1-penalties of weight vector of a linear model. For square loss, l_0-penalties led to an excess risk proportional to $\frac{\sigma^2 k \log d}{n}$, with a price of adaptivity of $\log d$, with few conditions on problem but no provably computationally efficient procedures. On contrary, l_1-norm penalization can be solved efficiently with appropriate convex optimization algorithms (e.g. proximal methods), but it only obtained a slow rate proportional to $\sqrt{\frac{\log d}{n}}$, exhibiting a high-dimensional phenomenon, but a worse dependence in n. Fast rates can be obtained only with stronger assumptions on covariance matrix of features.
    This chapter was limited to linear models. In Chap. 9, on neural networks, will see how models that are nonlinear in their parameters can lead to nonlinear variable selection, still exhibiting a high-dimensional phenomenon but at expense of harder optimization. This will be obtained by an l_1-norm on an infinite-dimensional space, & studied further in context of gradient boosting in Sect. 10.3.
- 9. Neural Networks. Chapter Summary:
  - Neural networks are flexible models for nonlinear predictions. They can be studied in terms of 3 errors usually related to empirical risk minimization: optimization, estimation, & approximation errors. In this chapter, focus primarily on single hidden-layer neural networks, which are linear combinations of simple affine functions with additional nonlinearities.
  - *Optimization error*: As prediction functions are nonlinearly dependent on their parameters, obtain nonconvex optimization problems with guaranteed convergence only to stationary points.
  - *Estimation error*: Number of parameters is not driver of estimation error, as norms of various weights play an important role, with explicit rates in $O(\frac{1}{\sqrt{n}})$ obtained from Rademacher complexity tools.
  - *Approximation error*: For rectified linear unit (ReLU) activation function, universal approximation properties can be characterized & are superior to those of kernel methods because they are adaptive to linear latent variables. In particular, neural networks can efficiently perform nonlinear variable selection.
  - 9.1. Introduction. In supervised learning, main focus has been put on methods to learn from n observations $(x_i, y_i), i = 1, \ldots, n$, with $x_i \in \mathcal{X}$ (input space) & $y_i \in \mathcal{Y}$ (output/label space). As presented in Chap. 4, a large class of methods relies on minimizing a regularized empirical risk w.r.t. a function $f : \mathcal{X} \to \mathbb{R}$, where following cost function is minimized:

$$\frac{1}{n} \sum_{i=1}^{n} l(y_i, f(x_)) + \Omega(f), \tag{96}$$

    where $l : \mathcal{Y} \times \mathbb{R} \to \mathbb{R}$: a loss function, $\Omega(f)$: a regularization term. Typical examples were:
    * **Regression.** $\mathcal{Y} = \mathbb{R}, l(y_i, f(x_i)) = \frac{1}{2}(y_i - f(x_i))^2$.
    * **Classification.** $\mathcal{Y} = \{\pm 1\}, l(y_i, f(x_i)) = \Phi(y_i f(x_i))$, where Φ is convex, e.g., $\Phi(u) = \max\{1 - u, 0\}$ (hinge loss leading to support vector machine) or $\Phi(u) = \log(1 + e^{-u})$ (leading to logistic regression). See more examples in Sect. 4.1.1.

    Class of prediction functions considered so far were as follows, with their pros & cons:
    * **Linear functions in some explicit features.** Given a feature map $\varphi : \mathcal{X} \to \mathbb{R}^d$, consider $f(x) = \theta^\top \varphi(x)$, with parameters $\theta \in \mathbb{R}^d$, as analyzed in Chap. 3 (for least-squares regression) & Chap. 4 (for Lipschitz-continuous losses).
      · Pros: Simple to implement, as they lead to convex optimization with gradient descent (GD) algorithms, with running time complexity in $O(nd)$, as shown in Chap. 5. They come with theoretical guarantees that are not necessarily scaling badly with dimension d if regularizers are used (l_2- or l_1-norm).
      · Cons: They only apply to linear functions on explicit (& fixed feature spaces), so they can underfit data. Moreover, feature vector φ is not learned from data.
    * **Linear functions in some implicit features through kernel methods.** Feature map can have arbitrarily large dimension, i.e., $\varphi(x) \in \mathcal{H}$ where $\mathcal{H}$: a Hilbert space, accessed through kernel function $k(x, x') = \langle \varphi(x), \varphi(x') \rangle_{\mathcal{H}}$, as presented in Chap. 7.
      · Pros: Nonlinear flexible predictions, simple to implement, & can be used with convex optimization algorithms with strong guarantees. They provide adaptivity to regularity of target function, allowing higher-dimensional applications than local averaging methods from Chap. 6.

· Cons: Running-time complexity goes up to $O(n^2)$ with algorithms from Sect. 7.4 (but this scaling can be improved with appropriate techniques discussed in same section, e.g. column sampling or random features). Method may still suffer from curse of dimensionality for target functions that are not smooth enough.

Aim: to explore another class of functions for nonlinear predictions – namely, neural networks, which come with additional benefits, e.g. more adaptivity to linear linear latent variables, but also have some potential drawbacks, e.g. a harder optimization problem.

◦ 9.2. Single Hidden-Layer Neural Network. Consider $\mathcal{X} = \mathbb{R}^d$ & set of prediction functions that can be written as

$$f(\mathbf{x}) = \sum_{j=1}^{m} \eta_j \sigma(\mathbf{w}_j^\top \mathbf{x} + b_j), \tag{97}$$

where $\mathbf{w}_j \in \mathbb{R}^d, b_j \in \mathbb{R}, j = 1, \dots, m$: *input weights*, $\eta_j \in \mathbb{R}, j = 1, \dots, m$: *output weights*, & σ: an *activation function*. Often represented as graph. Same architecture can also be considered with $\boldsymbol{\eta}_j \in \mathbb{R}^k$, for $k > 1$ to deal with multicategory classification (see Sect. 13.1).

Activation function is typically chosen from 1 of following examples (see plot):

∗ Sigmoid $\sigma(u) = \frac{1}{1+e^{-u}}$.

∗ Step function $\sigma(u) = 1_{u>0}$, which is not continuous & with zero derivative everywhere (& thus not amenable to gradient-based optimization).

∗ Rectified linear unit (ReLU) $\sigma(u) = (u)_+ = \max\{u, 0\}$, which will be main focus of this chapter.

∗ Hyperbolic tangent $\sigma(u) = \tanh u = \frac{e^u - e^{-u}}{e^u + e^{-u}}$.

Function f is defined as linear combination of m functions $\mathbf{x} \mapsto \sigma(\mathbf{w}_j^\top \mathbf{x} + b_j)$, which are hidden neurons. See https://playground.tensorflow.org/ for a nice interactive illustrative of this architecture. If input weights are fixed, obtain a linear model with m hidden neurons as features. A key benefit of neural networks: they perform feature learning by optimizing w.r.t. input weights.

Constant terms b_j are sometimes referred to as "biases," which is unfortunate in a statistical context, as that word already has a precise meaning within bias/variance trade-off (see Chap. 3 & Sect. 7.3).

Do not be confused by name "neural network" & its biological inspiration. This inspiration is not a proper justification for its behavior on ML problems.

Cross-entropy loss & sigmoid activation function for last layer. Following standard practice, we are not adding a nonlinearity to last layer; note: if were to use an additional sigmoid activation & consider cross-entry loss for binary classification, would exactly be using logistic loss on output without an extra activation function.

Indeed, if consider $g(x) = \frac{1}{1+e^{-f(x)}} \in [0, 1]$, & given an output variable $y \in \{\pm 1\}$, so-called "cross-entropy loss," an instance of maximum likelihood (see more details in Chap. 14), is equal to

$$-1_{y=1} \log g(x) - 1_{y=-1} \log(1 - g(x)) = 1_{y=1} \log(1 + e^{-f(x)}) + 1_{y=-1} \log(1 + e^{f(x)}) \tag{98}$$

which is exactly logistic loss $\log(1 + e^{-yf(x)})$ defined in Sect. 4.1.1 applied to prediction function $f(x)$. Practitioners sometimes refer to cross-entropy loss without mentioning: a sigmoid is applied beforehand (they, in fact, mean logistic loss). Such a discussion applies as well as multicategory classification & softmax loss (see Sect. 13.1.1).

Theoretical analysis of neural networks. As with any method based on empirical risk minimization, have to study 3 classical aspects:

(a) optimization error (convergence properties of algorithms for minimizing risk),

(b) estimation error (effect of having a finite amount of data on prediction performance),

(c) approximation error (effect of having a finite number of parameters or a constraint on norm of these parameters).

∗ 9.2.1. Optimization. To find parameters $\theta = \{(\eta_j), (\mathbf{w}_j), (b_j)\} \in \mathbb{R}^{m(d+2)}$, empirical risk minimization can be applied & following optimization problem has to be solved: (9.2)

$$\min_{\theta \in \mathbb{R}^{m(d+2)}} \frac{1}{n} \sum_{i=1}^{n} l\left(y_i, \sum_{j=1}^{m} \eta_j \sigma(\mathbf{w}_j^\top \mathbf{x}_i + b_j)\right), \tag{99}$$

with potentially additional regularization (often squared l_2-norm of all weights).

Note (as discussed in Chap. 5): true objective is to perform on unseen data, & optimization problem in (9.2) is just a means to an end.

This is a nonconvex optimization problem where GD algorithms from Chap. 5 can be applied without a strong guarantee beyond obtaining a vector with a small gradient norm (Sect. 5.2.6). See following discussion for recent results when providing qualitative global convergence guarantees when m is large.

While stochastic gradient descent (SGD) remains an algorithm of choice (also with a good generalization behavior, as discussed in Sect. 5.4), several algorithmic improvements have been observed to lead to better stability & performance: specific step-size decay schedules, preconditioning as presented in Sect. 5.4.2 (Duchi et al., 2011), momentum (Kingma & Ba, 2014), batch normalization (Ioffe & Szegedy, 2015), & layer normalization (Ba et al., 2016) to make optimization better behaved. However, overall, objective function is nonconex, & it remains challenging to understand precisely why

gradient-based methods perform well in practice, particularly with deeper networks (some elements are presented next & in Chap. 12). See also boosting procedures in Sect. 10.3 & Chap. 12, which learn neuron weights incrementally.

Global convergence of GD for infinite widths. Turn out: global convergence can be shown for this nonconvex optimization problem (Chizat & Bach, 2018; Bach & Chizat, 2022), with tools that go beyond the scope of this book & are partially described in Chap. 12.[4]

Simply show some experimental evidence for a simple 1D setup, where compare several runs of SGD when observations are seen only once (so no overfitting is possible) & with random initializations, on a regression problem with deterministic outputs, thus with optimal testing error (Bayes rate) equal to 0. Show in Fig. 9.1: Comparison of optimization behavior for different numbers m of neurons for ReLU activations $m = 5, 20, 100$. To generate data, also used a neural network with ReLU activations & 3 hidden neurons. Top: examples of final prediction functions at convergence; bottom: plot of test errors vs. number of iterations. estimated predictors & corresponding testing errors with 20 different initializations. Can observe: small errors are never achieved when $m = 5$ (sufficient to have zero testing errors). With $m = 20$ neurons, SGD finds optimal predictor for most restarts. When $m = 100$, all restarts have desired behaviors, highlighting benefits of overparametrization (see more details in Sect. 12.3).

∗ 9.2.2. Rectified Linear Units & Homogeneity. From now on, will mostly focus on ReLU activation $\sigma(u) = u_+$. Main property: will employ is its "positive homogeneity", i.e., for $\alpha > 0$, $(\alpha u)_+ = \alpha u_+$. This implies: in def of prediction function as sum of terms $\eta_j(\mathbf{w}_j^\top \mathbf{x} + b_j)_+$, can freely multiply $\eta_j \in \mathbb{R}$ by a positive scalar α_j & divide $(\mathbf{w}_j, b_j) \in \mathbb{R}^{d+1}$ by same α_j without changing prediction function, since then $\eta_j(\mathbf{w}_j^\top \mathbf{x} + b_j)_+ = (\alpha_j \eta_j)\left(\left(\frac{\mathbf{w}_j}{\alpha_j}\right)^\top \mathbf{x} + \frac{b_j}{\alpha_j}\right)_+$.

This has a particular effect when using a squared l_2-regularizer on all weights, which is standard, either explicitly (by adding a penalty to cost function) or implicitly (see Sect. 12.1). Indeed, consider penalizing $\eta_j^2 + \|\mathbf{w}_j\|_2^2 + \frac{b_j^2}{R^2}$ for each $j \in \{1, \dots, m\}$, where have added factor R^2 to constant term for unit homogeneity reasons between slop $\mathbf{w}_j$ & constant term b_j (R will be a bound on l_2-norm of input data). Dealing with unit homogeneity between η_j & $(\mathbf{w}_j, \frac{b_j}{R})$ does not matter because of invariance by rescaling described next.

Optimizing w.r.t. a scaling factor α_j (which affects only regularizer), have to minimize $\alpha_j^2 \eta_j^2 + \frac{\|\mathbf{w}_j\|_2^2 + \frac{b_j^2}{R^2}}{\alpha_j^2}$ with $\alpha_j^2 = \frac{(\|\mathbf{w}_j\|_2^2 + \frac{b_j^2}{R^2})^{\frac{1}{2}}}{|\eta_j|}$ as a minimizer & with optimal value of penalty equal to $2|\eta_j|(\|\mathbf{w}_j\|_2^2 + \frac{b_j^2}{R^2})^{\frac{1}{2}}$ (note: this leads to an l_1-norm penalty, thus with potentially sparsifying effects) (setting some of output weights η_j to 0), & robustness to large number of neurons (as shown in Sect. 9.2.3); for other relationship between l_2-regularization in neural networks & sparse estimation, see Sect. 12.1.3.

Therefore, for theoretical analysis (study of approximation & estimation errors), because of homogeneity, can choose to normalize each $(\mathbf{w}_j, b_j)$ to have unit norm $\|\mathbf{w}_j\|_2^2 + \frac{b_j^2}{R^2} = 1$, & use penalty $|\eta_j|$ for each $j \in \{1, \dots, m\}$, & thus use an overall l_1-norm penalty on η, i.e., $\|\eta\|_1$ (will consider other normalizations for input weights, either to ease exposition or to induce another behavior; e.g., by using l_1-norms on $\mathbf{w}_j$'s). Focus on this choice of regularization in following sections.

In this chapter, R denotes an almost sure upper bound on x directly, not on a feature map $\varphi(x)$ (as done in earlier chapters).

∗ 9.2.3. Estimation Error. To study estimation error, will consider: parameters of network are constrained, i.e., $\|\mathbf{w}_j\|_2^2 + \frac{b_j^2}{R^2} = 1$ for each $j \in \{1, \dots, m\}$ & $\|\eta\|_1 \leq D$. This defines a set Φ of allowed parameters $\theta = \{(\eta_j), (\mathbf{w}_j), (b_j)\}$.

Defining class $\mathcal{F}$ of neural network models f_θ with parameters $\theta \in \Theta$, can compute its Rademacher complexity using tools from Chap. 4 (Sect. 4.5). Assume: almost surely, $\|\mathbf{x}\|_2 \leq R$, i.e., input data are bounded in l_2-norm by R.

Following developments of Sect. 4.5 on Rademacher averages, denote by $\mathcal{G} = \{(x, y) \mapsto l(y, f(x)),\ f \in \mathcal{F}\}$ set of loss functions for a prediction function $f \in \mathcal{F}$. Note: following Sect. 4.5.3, consider a constraint on $\|\eta\|_1$, but could also penalize, which is more common to practice & can be tackled with tools from Sect. 4.5.5.

Have, by def of Rademacher complexity $R_n(\mathcal{G})$ of $\mathcal{G}$, & taking expectations w.r.t. data $(x_i, y_i), i = 1, \dots, n$, which are assumed to be independent & identically distributed (i.i.d.), & independent Rademacher random variables $\varepsilon_i \in \{\pm 1\}, i = 1, \dots, n$:

$$R_n(\mathcal{G}) = \mathbb{E}\left[\sup_{\theta \in \Theta} \frac{1}{n} \sum_{i=1}^n \varepsilon_i l(y_i, f_\theta(x_i))\right]. \tag{100}$$

This quantity is known to provide an upper bound on estimation error, as, using symmetrization from Prop. 4.2 & (4.10) from Sect. 4.4, when $\hat{f}$ is a minimizer of empirical risk over $\mathcal{F}$, have

$$\mathbb{E}\left[\mathcal{R}(\hat{f}) - \inf_{f \in \mathcal{F}} \mathcal{R}(f)\right] \leq 4 R_n(\mathcal{G}). \tag{101}$$

Can now use properties of Rademacher complexities presented in Sect. 4.5, particularly their nice handling of nonlinearities. Assuming: loss is G-Lipschitz-continuous w.r.t. 2nd variable, using Prop. 4.3 from Chap. 4, which allows getting rid of loss, get bound: `[skipped complicated estimates & techniques]`.

[4] See also https://francisbach.com/gradient-descent-neural-networks-global-convergence/ for more details.

Since ReLU activation function is 1-Lipschitz continuous & satisfies $(0)_+ = 0$, get, this time using extension of Prop. 4.3 from Chap. 4 to Rademacher complexities defined with an absolute value (i.e., Prop. 4.4), which adds an extra factor of 2 `[skipped complicated estimates & techniques]`.
Thus, get Prop. 9.1, with a bound proportional to $\frac{1}{\sqrt{n}}$ *with no explicit dependence in number of parameters.*

Proposition 6 (Estimation error)**.** *Let $\mathcal{F}$ be class of neural networks defined in (9.1), with constraint that $\|\eta\|_1 \leq D$ & $\|\mathbf{w}_j\|_2^2 + \frac{b_j^2}{R^2} = 1$, $\forall j \in \{1, \dots, m\}$, with ReLU activation function. If loss function is G-Lipschitz-continuous, then, for $\hat{f}$ a minimizer of empirical risk over $\mathcal{F}$,*

$$\mathbb{E}\left[\mathcal{R}(\hat{f}) - \inf_{f \in \mathcal{F}} \mathcal{R}(f)\right] \leq \frac{16GDR}{\sqrt{n}}. \tag{102}$$

Prop. 9.1 will be combined with a study of approximation properties in Sect. 9.3, with a summary provided in Sect. 9.4. Will see in Chap. 12 some recent results showing how optimization algorithms add an implicit regularization that leads to provable generalization in overparameterized neural networks (i.e., networks with many hidden units).
For estimation error, number of parameters is irrelevant! What counts is overall norm of weights.

Problem 9. *Provide a bound similar to Prop. 9.1 for alternative constraint $\|\mathbf{w}_j\|_1 + \frac{|b_j|}{R} = 1$, where R denotes supremum of $\|\mathbf{x}\|_\infty$ over all $\mathbf{x}$ in support of its distribution.*

Before moving on to approximation properties of neural networks, note: reasoning given here for computing Rademacher complexity can be extended by recursion to deeper networks & other activation functions, as Exercise 9.2 shows (see, e.g., Neyshabur et al., 2015, for further results).

Problem 10. *Consider a 1-Lipschitz-continuous activation function σ s.t. $\sigma(0) = 0$, & classes of functions defined recursively as $\mathcal{F}_0 = \{\mathbf{x} \mapsto \boldsymbol{\theta}^\top \mathbf{x},\ \|\boldsymbol{\theta}\|_2 \leq D_0\}$, &, for $i = 1, \dots, M$, $\mathcal{F}_i = \{\mathbf{x} \mapsto \sum_{j=1}^{m_i} \theta_j \sigma(f_j(\mathbf{x})), f_j \in \mathcal{F}_{i-1}, \|\theta\|_1 \leq D_i\}$, corresponding to a neural network with M layers. Assuming $\|\mathbf{x}\|_2 \leq R$ almost surely, show by recursion: Rademacher complexity satisfies $R_n(\mathcal{F}_M) \leq 2^M \frac{R}{\sqrt{n}} \prod_{i=0}^M D_i$.*

- 9.3. Approximation Properties. As seen in Sect. 9.2.3, estimation error for constrained output weights grows as $\frac{\|\boldsymbol{\eta}\|_1}{\sqrt{n}}$, where $\boldsymbol{\eta}$: vector of output weights & is independent of number m of neurons. Several important questions will be tackled in following sects:
  - *Universality*: Can we approximate any prediction function with a sufficiently large number of neurons?
  - *Bound on approximation error*: What is associated approximation error so that we can derive generalization bounds? How can we use control of l_1-norm $\|\boldsymbol{\eta}\|_1$, particularly when number of neurons m is allowed to tend to ∞?
  - *Finite number of neurons*: What is number of neurons required to reach such a behavior?

  To do this, need to understand space of functions that neural networks span & how they relate to smoothness properties of function (as did for kernel methods in Chap. 7).
  Focus on ReLU activation function, note: universal approximation results exist as soon as activation function is not a polynomial (Leshno et al., 1993). Start with a simple nonquantitative argument to show universality in 1D (& then in all dimensions) before formalizing function space obtained by letting number of neurons go to ∞.
  - 9.3.1. Universal Approximation Property in 1D. Start with a number of simple, nonquantitative arguments.
    Approximation of continuous piece affine functions. Since each individual function $x \mapsto \eta_j(w_j x + b_j)_+$ is continuous piecewise affine, output of a neural network has to be continuous piecewise affine as well. Turn out: all continuous piecewise affine functions with $m-2$ kinks in open interval $(-R, R)$ can be represented by m neurons on $[-R, R]$. Indeed, as illustrated here with $m = 8$, if assume: function f is s.t. $f(-R) = 0$, with kinks $a_1 < \dots < a_{m-2}$ on $(-R, R)$, can approximate it on $[-R, a_1]$ by function $v_1(x + R)_+$ where v_1: slop of f on $[-R, a_1]$. Approximation is tight on $[-R, a_1]$. To have a tight approximation on $[a_1, a_2]$ without perturbing approximation on $[-R, a_1]$, can add to approximation $v_2(x - a_1)_+$, where v_2 is exactly what is needed to compensate for change in slope of f. By pursuing this reasoning, can present function on $[-R, R]$ exactly with $m-1$ neurons Fig.
    To remove constraint $f(-R) = 0$, can simply notice: $\frac{1}{2R}(x + R)_+ + \frac{1}{2R}(-x + R)_+$ is equal to 1 on $[-R, R]$. Thus, with 1 additional neuron (only 1 since $(x + R)_+$ has already been used), can represent any piecewise-affine function with $m-2$ kinks using m neurons. This argument will be made more quantitative in Sect. 9.3.3 by looking at slopes of piecewise affine function.
    Universal approximation properties. Now can represent precisely all continuous piecewise affine functions on $[-R, R]$, can use classical approximation theorems for functions on $[-R, R]$. They come in different flavors depending on norm used to characterize approximation. E.g., continuous functions can be approximated by piecewise affine functions with arbitrary precision in L_∞-norm (defined as maximal value of $|f(x)|$ for $x \in [-R, R]$) by simply taking piecewise interpolant from a grid (see quantitative arguments in Sect. 9.3.3). With a weaker criterion e.g. L^2-norm (w.r.t. Lebesgue measure), can approximate any function in L^2 (see, e.g., [Rud87]). This can be extended to any dimension d by using Fourier transform representation as $f(x) = \frac{1}{(2\pi)^2} \int_{\mathbb{R}^d} \hat{f}(\boldsymbol{\omega}) e^{\mathrm{i}\boldsymbol{\omega}^\top \mathbf{x}} \mathrm{d}\boldsymbol{\omega}$ & approximating 1D functions sine & cosine as linear superpositions of ReLUs. See a more formal quantitative argument in Sect. 9.3.4.
    To obtain precise bounds in all dimensions in terms of number of kinks or l_1-norm of output weights, 1st need to define limit when number of neurons diverges.
  - 9.3.2. Infinitely Many Neurons & Variation Norm. In this section, consider neural networks of form $f(x) = \sum_{j=1}^m \eta_j(\mathbf{w}_j^\top \mathbf{x} + b_j)_+$, where input weights are constrained, i.e., $(\mathbf{w}_j, \frac{b_j}{R}) \in K$, for K a compact subset of $\mathbb{R}^{d+1}$, e.g. unit l_2-sphere (but

will consider a slightly different set at end of this sect). Goal: to define set of functions that can be approximated by neural networks, while defining a norm on them that extends l_1-norm of output weights. Consider $\mathcal{X}$ d-dimensional l_2-ball of radius R & center 0 (but construction applies to any compact subset of $\mathbb{R}^d$).

Formulation through measures. Can write a neural network with finitely many neurons $f(x) = \sum_{j=1}^m \eta_j (\mathbf{w}_j^\top \mathbf{x} + b_j)_+$ as integral (9.4)

$$f(x) = \int_K (\mathbf{w}^\top \mathbf{x} + b)_+ \, \mathrm{d}\nu(\mathbf{w}, b), \tag{103}$$

for ν being signed measure $\nu = \sum_{j=1}^m \eta_j \delta_{(\mathbf{w}_j, b_j)}$, where $\delta_{(\mathbf{w}_j, b_j)}$: Diract measure at $(\mathbf{w}_j, b_j)$. Penalty can be written as $\|\boldsymbol{\eta}\|_1 = \int_K |\mathrm{d}\nu(\mathbf{w}, b)|$, which is total variation of ν.[5]

Since want to have a norm $\|\boldsymbol{\eta}\|_1$ which is as small as possible, among all representations of f as in (9.4), look for the one for which $\int_K |d\nu(\mathbf{w}, b)|$ is smallest, i.e., for $f \in \widetilde{\mathcal{F}}_1$ set of neural networks with arbitrary (finite) width, define

$$\tilde{\gamma}_1(f) = \int_{\nu \in \widetilde{\mathcal{M}}(K)} \int_K |d\nu(\mathbf{w}, b)| \text{ s.t. } \forall x \in \mathcal{X}, \; f(x) = \int_K (\mathbf{w}^\top \mathbf{x} + b)_+ \, \mathrm{d}\nu(\mathbf{w}, b), \tag{104}$$

where $\widetilde{\mathcal{M}}(K)$: set of signed measures on K with *finite* support. This happens to define a norm on $\widetilde{\mathcal{F}}_1$. In order to extend beyond set $\widetilde{\mathcal{F}}_1$ (which is equal to set of continuous piecewise affine functions for $d = 1$), simply relax constraint of finite support for measure ν. I.e., for $f : \mathcal{X} \to \mathbb{R}$, define (9.5)

$$\gamma_1(f) = \inf_{\nu \in \mathcal{M}(K)} \int_K |d\nu(\mathbf{w}, b)| \text{ s.t. } \forall x \in \mathcal{X}, \; f(x) = \int_K (\mathbf{w}^\top \mathbf{x} + b)_+ \, \mathrm{d}\nu(\mathbf{w}, b), \tag{105}$$

where $\mathcal{M}(K)$: set of signed measures on K with finite total variation, with convention: if no measure can be found to represent f, then $\gamma_1(f) = +\infty$. Prop. 9.2 shows: γ_1 defines a norm on set $\mathcal{F}_1$ of functions s.t. $\gamma_1(f)$ is finite.

Proposition 7. *Assume $K \subset \mathbb{R}^{d+1}$ & $\mathcal{X} \subset \mathbb{R}^d$ are compact sets. Set $\mathcal{F}_1$ of functions s.t. $\gamma_1(f)$ defined in (9.5) is finite is a vector space, a subset of set of Lipschitz-continuous functions on $\mathcal{X}$. Moreover, γ_1 is a norm on $\mathcal{F}_1$.*

Obtain a Banach space $\mathcal{F}_1$ of functions (proof of completeness is left as a technical exercise), with a norm γ_1 that is often referred to as "variation norm" (Kurková & Sanguineti, 2001). This characterizes set of functions that can be asymptotically reached by neural networks with a bounded l_1-norm of output weights, regardless of number of neurons. Index 1 in γ_1 will become natural when we compare with positive-definite kernels in Sect. 9.5. Note: although defined it for ReLU activation, same argument applies to all continuous activation functions. Finally, in order to obtain upper bounds on $\gamma_1(f)$, it suffices to represent f as an integral of neurons as in (9.5), & compute corresponding total variation, e.e.g, for a single neuron $f(\mathbf{x}) = (\mathbf{w}^\top \mathbf{x} + b)_+$ for $(\mathbf{w}, b) \in K, \gamma_1(f) \leq 1$, a property that will be used several times in Sect. 9.3.3.

Note: due to positive homogeneity of ReLU activation function, norm γ_1 does not change if replace compact set K with $\bigcup_{c \in [0,1]} cK$ (i.e., union of all segments $[0, v]$ for $v \in K$), with a proof left as an exercise. Therefore, choosing unit l_2-sphere or unit l_2-ball for K gives same results. (Will make a slightly different choice below.)

Studying approximation properties of $\mathcal{F}_1$. Have characterized function space $\mathcal{F}_1$ through (9.5), need to describe set of functions with finite norm & relate this norm to classical smoothness properties (as done for kernel methods in Chap. 7). To do so, as illustrated below, consider a smaller set K than unit l_2-ball, i.e., set K of $(\mathbf{w}, \frac{b}{R})$ s.t. $\|\mathbf{w}\|_2 = \frac{1}{\sqrt{2}}, |b| \leq \frac{R}{\sqrt{2}}$, which is enough to obtain upper bounds on approximation errors. For simplicity, & losing a factor of $\sqrt{2}$, consider normalization $K = \{\mathbf{w}, \frac{b}{R} \in \mathbb{R}^{d+1}, \|\mathbf{w}\|_2 = 1, |b| \leq R\}$ & norm γ_1 defined in (9.5) with this set K. Note: for $d = 1$, have $K = \{(\mathbf{w}, \frac{b}{R}) \in \mathbb{R}^d, w \in \{\pm 1\}, |b| \leq R\}$, as illustrated below for $d = 1$ (with new set $\bigcup_{c \in [0,1]} cK$ in dark gray, & old one in light gray). Could stick to l_2-sphere, but our particular choice of K leads to simpler formulas.

* 9.3.3. Variation Norm in 1D. ReLU activation function is specific & leads to simple approximation properties in interval $[-R, R]$. As already qualitatively described in Sect. 9.3.1, start with continuous piecewise affine functions, which, given shape of ReLU activation, should be easy to approximate (& immediately lead to universal approximation results as all reasonable functions can be approximated by piecewise affine functions). See more details by Breiman (1993) & Barron & Klusowski (2018).

Continuous piecewise affine functions. Can make reasoning in Sect. 9.3.1 quantitative. Consider a continuous piecewise affine function on $[-R, R]$ with specific knots at each $-R = a_0 < a_1 < \cdots < a_{m-2} < a_{m-1} = R$, so on $[a_j, a_{j+1}]$, f is affine with slope v_j, for $j \in \{0, \ldots, m-2\}$. `[Technical details]`

$$\gamma_1(f) \leq \frac{1}{2} \left| \frac{1}{2R} [f(-R) + f(R)] + v_0 \right| + \frac{1}{2} \left| \frac{1}{2R} [f(-R) + f(R)] - v_{m-2} \right| + \sum_{j=1}^{m-2} |v_j - v_{j-1}|. \tag{106}$$

Norm is thus upper-bounded by values of f & its derivatives at boundaries of interval & sums of changes in slope.

Twice continuously differentiable functions. Now consider a twice continuously differentiable function f on $[-R, R]$, & would like to express it as a continuous linear combination of functions $x \mapsto (\pm x + b)_+$. Will consider

[5] When ν has density $\frac{d\nu}{d\tau}$ w.r.t. a base measure τ with full support in K, then total variation is defined as integral $\int_K |\frac{d\nu}{d\tau(\mathbf{w},b)}| d\tau(\mathbf{w}, b)$ & is independent of choice of τ. See https://en.wikipedia.org/wiki/Total_variation for more details.

2 arguments: one through approximation by piecewise affine functions & one through Taylor's formula with integral remainder.

Piecewise-affine approximation. Consider equally spaced knots $a_j = -R + \frac{j}{s}R$ for $j \in \{0, \dots, 2s\}$, & piecewise affine interpolation $\hat{f}$ from values $a_j, f(a_j)$ (& slopes v_j on $[a_j, a_{j+1}]$), with $j \in \{0, \dots, 2s\}$, for s that will tend to ∞ (see following illustration, where have $m - 1 = 2s$) `[Technical details]`

Thus, approximant $\hat{f}$ has a γ_1-norm $\gamma_1(\hat{f})$ upper-bounded asymptotically by

$$\frac{1}{2}|\frac{1}{2R}[f(-R) + f(R)] + f'(-R)| + \frac{1}{2}|\frac{1}{2R}[f(-R) + f(R)] - f'(R)| + \frac{R}{s}\sum_{j=1}^{2s-1}|f''\left(-R + \frac{j}{s}R\right)|. \tag{107}$$

Last term $\frac{R}{s}\sum_{j=1}^{2s-1}|f''\left(-R + \frac{j}{s}R\right)| \to \int_{-R}^{R}|f''(x)|\,\mathrm{d}x$. Thus, letting $s \to \infty$, get (informally, as reasoning given next will make it more formal)

$$\gamma_1(f) \leq \frac{1}{2}|\frac{1}{2R}[f(-R) + f(R)] + f'(-R)| + \frac{1}{2}|\frac{1}{2R}[f(-R) + f(R)] - f'(R)| + \int_{-R}^{R}|f''(x)|\,\mathrm{d}x. \tag{108}$$

This notably shows: although number of neurons is allowed to grow, l_1-norm of weights remains bounded by quantity in (9.8).

Direct proof through Taylor's formula. (9.8) can be extended to continuously differentiable functions, which are only twice differentiable a.e. with integrable 2nd-order derivatives. In this sect, assume: function f is twice continuously differentiable but could extend to only integrable 2nd derivatives by a density argument (see, e.g., [Rud87]). For such a function, using Taylor's formula with integral remainder, have, for $x \in [-R, R]$, using fact $(x - b)_+ = 0$ as soon as $b \geq x$ `[Technical details]`

Will also use a simpler upper bound, obtained from triangle inequality:

$$\gamma_1(f) \leq \frac{1}{2R}|f(-R) + f(R)| + \frac{1}{2}|f'(R)| + \frac{1}{2}|f'(-R)| + \int_{-R}^{R}|f''(x)|\,\mathrm{d}x. \tag{109}$$

Problem 11. *Assume $-R = x_1 < \cdots < x_n = R, y_1, \dots, y_n \in \mathbb{R}$, show: piecewise-affine interpolant on $[-R, R]$ is a minimum norm interpolant.*

* 9.3.4. Variation Norm in an Arbitrary Dimension. In order to extend to larger dimensions than $d = 1$, will use Fourier transforms. This requires to consider functions on $\mathcal{X}$ ball with center 0 & radius R as restrictions of functions defined on $\mathbb{R}^d$ with compact support (so that they belong to $L^2(\mathbb{R}^d)$, space of square-integrable functions for Lebesgue measure, & $L^1(\mathbb{R}^d)$ space of integrable functions); this can be done in a number of ways (see [Rud87] & end of Sect. 7.5.2). `[Technical details]`

  Obtain (9.14)

$$\gamma_1(f) \leq \frac{1}{(2\pi)^d}\int_{\mathbb{R}^d}|\hat{f}(\boldsymbol{\omega})|\gamma_1(\mathbf{x} \mapsto e^{\mathrm{i}\boldsymbol{\omega}^\top \mathbf{x}})\,\mathrm{d}\boldsymbol{\omega} \leq \frac{2}{(2\pi)^d R}\int_{\mathbb{R}^d}|\hat{f}(\boldsymbol{\omega})|(1 + 2R^2\|\boldsymbol{\omega}\|_2^2)\,\mathrm{d}\boldsymbol{\omega}. \tag{110}$$

  Given function $g : \mathbb{R}^d \to \mathbb{R}$, $\int_{\mathbb{R}^d}|\hat{g}(\boldsymbol{\omega})|\,\mathrm{d}\boldsymbol{\omega}$ is a measure of smoothness of g, so $\gamma_1(f)$ being finite imposes: f & all 2nd-order derivatives of f have this form of smoothness. RHS of (9.14) is often referred to as "Barron norm," which is named after Barron (1993, 1994). See Klusowski & Barron (2018) for more details.

  To relate norm γ_1 to other function spaces e.g. Sobolev spaces, will consider further upper bounds (& relate them to another norm γ_2, described in Sect. 9.5).

  Problem 12 (Step activation function)**.** *Consider step activation function defined as $\sigma(u) = 1_{u>0}$. Show: corresponding variation norm can be upper-bounded by a constant times $\int_{\mathbb{R}^d}|\hat{f}(\boldsymbol{\omega})|(1 + R\|\boldsymbol{\omega}\|_2)\,\mathrm{d}\boldsymbol{\omega}$.*

* 9.3.5. Precise Approximation Properties.
* 9.3.6. From Variation Norm to a Finite Number of Neurons.

○ 9.4. Generalization Performance for Neural Networks.

○ 9.5. Relationship with Kernel Methods.

* 9.5.1. From a Banach Space $\mathcal{F}_1$ to a Hilbert Space $\mathcal{F}_2$.
* 9.5.2. Kernel Function.
* 9.5.3. Upper Bound on RKHS Norm.

○ 9.6. Experiments. Consider same experimental setup as Sect. 7.7, i.e., 1D problems to highlight adaptivity of neural methods to regularity of target function, with smooth targets & nonsmooth targets. Consider several values for number m of hidden neurons & a neural network with ReLU activation functions & an additional global constant term. Training is done by SGD with a small constant step size & random initialization.

Note: for small m, while a neural network with same number of hidden neurons could fit data better, optimization is unsuccessful (SGD gets trapped in a bad local minimum). Moreover, between $m = 32$ & $m = 100$, do not see any overfitting, highlighting potential underfitting behavior of neural networks. See also Stewart et al. (2023) for a formulation of regression through classification that alleviates some of these issues, as well as https://francisbach.com/quest-for-adaptivity/.

- 9.7. Extensions. Fully connected, single-hidden-layer neural networks are far from what is used in practice, particularly in computer vision, & natural language processing. Indeed, state-of-the-art performance is typically achieved with following extensions:
  - **Going deep with multiple layers.** Most simple form of deep neural network is a multilayer, fully connected neural network. Ignoring constant terms for simplicity, it is of form $f(\mathbf{x}^{(0)}) = \mathbf{y}^{(L)}$, with input $x^{(0)}$ & output $y^{(L)}$ given by

$$\mathbf{y}^{(k)} = (W^{(k)})^\top \mathbf{x}^{(k-1)}, \tag{111}$$
$$\mathbf{x}^{(k)} = \sigma(\mathbf{y}^{(k)}), \tag{112}$$

    where $W^{(l)}$: matrix of weights for layer l. For these models, obtaining simple & powerful theoretical results is still an active area of research in terms of approximation, estimation, & optimization errors. See, e.g., Lu et al. (2021), Ma et al. (2020), & Yang & Hu (2021). Among these results, so-called "neural tangent kernel" provides another link between neural networks & kernel methods beyond the one described in Sect. 9.5, & that applies more generally (see Sect. 12.4 &, e.g., Jacot et al., 2018; Chizat et al., 2019).
  - **Residual networks.** An alternative to stacking layers 1 after the other as before is to introduce a different architecture of form:

$$\mathbf{y}^{(k)} = (W^{(k)})^\top \mathbf{x}^{(k-1)}, \tag{113}$$
$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + \sigma(\mathbf{y}^{(k)}). \tag{114}$$

    Direct modeling of $\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}$ instead of $\mathbf{x}^{(k)}$ through an extra nonlinearity, originating from He et al. (2016), can be seen as a discretization of an ODE (see Chen et al., 2018).
  - **Convolutional neural networks.** To tackle large data & improve performances, important to use prior knowledge about typical data structure to process. E.g., for signals, images, & videos, important to take into account translation invariance (up to boundary issues) of domain. Done by constraining linear operators involved in linear part of neural networks to respect some form of translation invariance, & thus to use convolutions. See Goodfellow et al. (2016) for details. This can be extended beyond grids to topologies expressed in terms of graphs, leading to graph neural networks (see, e.g., Bronstein et al., 2021).
  - **Transformers.** 1 approach to capture long-range dependencies in sequential data $X = (x_1, \ldots, x_L) \in \mathbb{R}^{L \times d}$, is to learn query $Q = W^{(Q)}X$, key $K = W^{(K)}X$, & value $V = W^{(V)}X$ matrices obtained by linear operators on X of compatible sizes, which are combined together to form an attention mapping (Bahdanau et al., 2014):

$$\text{attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right) V. \tag{115}$$

    Such a mapping is capable of capturing a variety of semantic relationships over sequences of data (e.g., grammatical relationships between query & key tokens within a corpus of text). Transformer (Vaswani et al., 2017) is an architecture that consists of stacked blocks made up of attention mappings, fully-connected layers & residual connections. Transformer architecture & its variants have a multitude of applications in fields e.g. natural language processing, audio, & computer vision.
- 9.8. Conclusions. In this chapter, have focused primarily on neural networks with 1 hidden layer & provided guarantees on approximation & estimation errors, which show: this class of models, if empirical risk minimization can be performed, leads to a predictive performance that improves on kernel methods from Chap. 7 by being adaptive to linear latent variables (e.g., dependence on an unknown linear projection of data). In particular, highlight: having a number of neurons in order of number of observations is not detrimental to good generalization performance, so long as norm of weights is controlled.
  – Việc có số lượng nơ-ron theo thứ tự số lượng quan sát không gây bất lợi cho hiệu suất tổng quát tốt, miễn là chuẩn mực của trọng số được kiểm soát.
  Pursue study of overparameterized models in Chap. 12, where show how optimization algorithms both globally converge & lead to implicit biases.

- 10. Ensemble Learning.
- 11. From Online Learning to Bandits.
- 12. Overparametrized Models.
- 13. Structured Prediction.
- 14. Probabilistic Methods.
- 15. Lower Bounds.
- Conclusion.

4 Deep Learning

Resources – Tài nguyên.

1. [BB24]. Christopher M. Bishop, Hugh Bishop. *Deep Learning: Foundations & Concepts*. [130 Amazon ratings]

Amazon review. This book offers a comprehensive introduction to central ideas that underpin DL. Intended both for newcomers to ML & for those already experienced in field. Covering key concepts relating to contemporary architectures & techniques, this essential book equips readers with a robust foundation for potential future specialization. Field of DL is undergoing rapid evolution ⇒ this book focuses on ideas that are likely to ensure test of time.

Book is organized into numerous bite-sized chaps, each exploring a distinct topic, & narrative follows a linear progression, with each chap building upon content from its predecessors. This structure is well-suited to teaching a 2-semester undergraduate or postgraduate ML course, while remaining equally relevant to those engaged in active research or in self-study.

A full understanding of ML requires some mathematical background & so book includes a self-contained introduction to probability theory. However, focus of book is on conveying a clear understanding of ideas, with emphasis on real-world practical value of techniques rather than on abstract theory. Complex concepts are therefore presented from multiple complementary perspectives including textual descriptions, diagrams, mathematical formulae, & pseudo-code.

- "Chris Bishop wrote a terrific textbook on neural networks in 1996 & has a deep knowledge of field & its core ideas. His many years of experience in explaining neural networks have made him extremely skillful at presenting complicated ideas in simplest possible way & it is a delight to see these skills applied to revolutionary new developments in field." – Geoffrey Hinton
- "With recent explosion of DL & AI as a research topic, & quickly growing importance of AI applications, a modern textbook on topic was badly needed. "New Bishop" masterfully fills gap, covering algorithms for supervised & unsupervised learning, modern DL architecture families, as well as how to apply all of this to various application areas." – Yann LeCun
- "This excellent & very educational book will bring reader up to date with main concepts & advances in DL with a solid anchoring in probability. These concepts are powering current industrial AI systems & are likely to form basis of further advances towards artificial general intelligence." – Yoshua Bengio

  About the Author. Chris Bishop is a Technical Fellow at Microsoft & is Director of Microsoft Research AI4Science. He is a Fellow of Darwin College, Cambridge, a Fellow of Royal Academy of Engineering, a Fellow of Royal Society of Edinburgh, & a Fellow of Royal Society of London. He is a keen advocate of public engagement in science, & in 2008 he delivered prestigious Royal Institution Christmas Lectures, established in 1825 by Michael Faraday, & broadcast on prime-time national television. Chris was a founding member of UK AI Council & was also appointed to Prime Minister's Council for Science & Technology. Christopher Michael Bishop (born Apr 7, 1959) FREng, FRSE, is Laboratory Director at Microsoft Research Cambridge & professor of Computer Science at University of Edinburgh & a Fellow of Darwin College, Cambridge. Chris obtained a Bachelor of Arts degree in Physics from St Catherine's College, Oxford, & a PhD in Theoretical Physics from University of Edinburgh, with a thesis on quantum field theory.

  Huge Bishop is an Applied Scientist at Wayve, & end-to-end DL based autonomous driving company in London, where he designs & trains deep neural networks. Before working at Wayve, he completed his MPhil in ML & Machine Intelligence in engineering department at Cambridge University. Hugh also holds an MEng in Computer Science from University of Durham, where he focused his projects on DL. During his studies, he also worked as an intern at FiveAI, another autonomous driving company in UK, & as a Research Assistant, producing educational interactive iPython notebooks for ML courses at Cambridge University.

Preface. DL uses multilayered neural networks trained with large data sets to solve complex information processing tasks & has emerged as most successful paradigm in field of ML. Over last decade, DL has revolutionized many domains including computer vision, speech recognition, & natural language processing, & it is being used in a growing multitude of applications across healthcare, manufacturing, commerce, finance, scientific discovery, & many other sectors. Recently, massive neural networks, known as large language models & comprising of order of a trillion learnable parameters, have been found to exhibit 1st indications of general AI & are now driving 1 of biggest disruptions in history of technology.

- Goals of Book. This expanding impact has been accompanied by an explosion in number & breadth of research publications in ML, & pace of innovation continues to accelerate. For newcomers to field, challenge of getting to grips with key ideas, let alone catching up to research frontier, can seem daunting (đáng sợ). Against this backdrop, *Deep Learning: Foundations & Concepts* aims to provide newcomers to ML, as well as those already experienced in field, with a thorough understanding of both foundational ideas that underpin DL as well as key concepts of modern DL architectures & techniques. This material will equip reader with a strong basis for future specialization. Due to breadth & pace of change in field, have deliberately avoided trying to create a comprehensive survey of latest research. Instead, much of value of book derives from a distillation of key ideas, & although field itself can be expected to continue its rapid advance, these foundations & concepts are likely to stand test of time. E.g., large language models (LLMs) have been evolving very rapidly at time of writing, yet underlying transformer architecture & attention mechanism have remained largely unchanged for last 5 years, while many core principles of ML have been known for decades.
- Responsible use of technology. DL is a powerful technology with broad applicability that has potential to create huge value for world & address some of society's most pressing challenges. However, these same attributes mean: DL also has potential for deliberate misuse & to cause unintended harms. Have chosen not to discuss ethical or societal aspects of use of DL, as these topics are of such importance & complexity that they warrant a more thorough treatment than is possible in a technical textbook such as this. Such considerations should, however, be informed by a solid grounding in underlying technology & how it works, & so hope this book will make a valuable contribution towards these important discussions. Reader is,

nevertheless, strongly encouraged to be mindful about broader implications of their work & to learn about responsible use of DL & AI alongside their studies of technology itself.

- Structure of book. Book is structured into a relatively large number of smaller bite-sized chaps, each of which explores a specific topic. Book has a linear structure in sense: each chap depends only on material covered in earlier chaps. Well suited to teaching a 2-semester undergraduate or postgraduate course on ML but is equally relevant to those engaged in active research or in self-study.

  A clear understanding of ML can be achieved only through use of some level of mathematics. Specifically, 3 areas of mathematics lie at heart of ML: probability theory, linear algebra, & multivariate calculus. Book provides a self-contained introduction to required concepts in probability theory & includes an appendix that summarizes some useful results in linear algebra. Assumed: reader already has some familiarity with basic concepts of multivariate calculus although there are appendices that provide introductions to calculus of variations & to Lagrange multipliers. Focus of book, however, is on conveying a clear understanding of ideas, & emphasis is on techniques that have real-world practical value rather than on abstract theory. Where possible try to present more complex concepts from multiple complementary perspectives including textual description, diagrams, & mathematical formulae. In addition, many of key algorithms discussed in text are summarized in separate boxes. These do not address issues of computational efficiency, but are provided as a complement to mathematical explanations given in text. Therefore hope: material in this book will be accessible to readers from a variety of backgrounds.

  Conceptually, this book is perhaps most naturally viewed as a successor (người kế nhiệm/, người nối nghiệp) to *Neural Networks for Pattern Recognition* (Bishop, 1995b), which provided 1st comprehensive treatment of neural networks from a statistical perspective. It can also be considered as a companion volume to *Pattern Recgonition & ML* (Bishop, 2006), which covered a broader range of topics in ML although it predated DL revolution. However, to ensure that this new book is self-contained, to appropriate material has been carried over from Bishop (2006) & refactored to focus on those foundational ideas that are needed for DL. I.e., there are many interesting topics in ML discussed in Bishop (2006) that remain of interest today but which have been omitted from this new book. E.g., Bishop (2006) discussed Bayesian methods in some depth, whereas this book is almost entirely non-Bayesian.

  Book is accompanied by a web site that provides supporting material, including a free-to-use digital version of book as well as solutions to exercises & downloadable versions of figures in PDF & JPEG formats: https://www.bishopbook.com.

- References. In spirit of focusing on core ideas, make no attempt to provide a comprehensive literature review, which in any case would be impossible given scale & pace of change of field. Do, however, provide refs to some of key research papers as well as review articles & other refs to some of key research papers as well as review articles & other sources of further reading. In many cases, there also provide important implementation details that we gloss over in text in order to distract reader from central concepts being discussed.

  Many books have been written on subject of ML in general & on DL in particular. Those which are closest in level & style to this book include Bishop (2006), Goodfellow, Bengio, & Courville (2016), Murphy (2022), Murphy (2023), & Prince (2023).

  Over last decade, nature of ML scholarship has changed significantly, with many papers being posted online on archival sites ahead of, or even instead of, submission to peer-reviewed conferences & journals. Most popular of these sites is *arXiv* https://arXiv.org. The site allows papers to be updated, often leading to multiple versions associated with different calendar years, which can result in some ambiguity as to which version should be cited & for which year. Also provides free access to a PDF of each paper. Have therefore adopted a simple approach of referencing paper according to year of 1st upload, although recommend reading most recent version. Papers on arXiv are indexed using a notation `arXiv:YYMM.XXXXX` where `YY, MM` denote year & month of 1st upload, resp. Subsequent versions are denoted by appending a version number `N` in form `arXiv:YYMM.XXXXXvN`.

- Exercises. Each chap concludes with a set of exercises designed to reinforce key ideas explained in text or to develop & generalize them in significant ways. These exercises form an important part of text & each is graded according to difficulty ranging from $\star$, which denotes a simple exercise taking a few moments to complete, through to $\star\star\star$, which denotes a significantly more complex exercise. Reader is strongly encouraged to attempt exercises since active participation with material greatly increases effectiveness of learning. Worked solutions to all of exercises are available as a downloadable PDF file from book website.

- Mathematical notation. Follow same notation as Bishop (2006). For an overview of mathematics in context of ML, see Deisenroth, Faisal, & Ong (2020).

  Vectors are denoted by lower case bold roman letters e.g. $\mathbf{x}$, whereas matrices are denoted by uppercase bold roman letters, e.g. $\mathbf{M}$. All vectors are assumed to be column vectors unless otherwise stated. A superscript $\top$ denotes transpose of a matrix or vector, so that $\mathbf{x}^\top$ will be a row vector. Notation $(w_1, \ldots, w_M)$ denotes a row vector with M elements, & corresponding column vector is written as $\mathbf{w} = (w_1, \ldots, w_M)^\top$. $M \times M$ identity matrix (also known as unit matrix) is denoted $\mathbf{I}_M$, abbreviated to $\mathbf{I}$ if there is no ambiguity about its dimensionality. It has elements $I_{ij} = \delta_{ij}$. Elements of a unit matrix are sometimes denoted by δ_{ij}. Notation $\mathbf{1}$ denotes a column vector in which all elements have value 1. $\mathbf{a} \oplus \mathbf{b}$ denotes concatenation of vectors $\mathbf{a}, \mathbf{b}$, so that if $\mathbf{a} = (a_1, \ldots, a_N), \mathbf{b} = (b_1, \ldots, b_M)$ then $\mathbf{a} \oplus \mathbf{b} = (a_1, \ldots, a_N, b_1, \ldots, b_M)$. $|x|$ denotes modulus (positive part) of a scalar x, also known as *absolute value*. Use $\det \mathbf{A}$ to denote determinant of a matrix $\mathbf{A}$.

  Notation $x \sim p(x)$ signifies: x is sampled from distribution $p(x)$. Where there is ambiguity, use subscripts as in $p_x(\cdot)$ to denote which density is referred to. Expectation of a function $f(x, y)$ w.r.t. a random variable x is denoted by $\mathbb{E}_x[f(x, y)]$.

In situations where there is no ambiguity as to which variable is being averaged over, this will be simplified by omitting suffice, e.g. $\mathbb{E}[x]$. If distribution of x is conditioned on another variable z, then corresponding conditional expectation will be written $\mathbb{E}_x[f(x)|z]$. Similarly, variance of $f(x)$ is denoted $\text{var}[f(x)]$, & for vector variables, covariance is written $\text{cov}[\mathbf{x}, \mathbf{y}]$. Will also use $\text{cov}[\mathbf{x}]$ as a shorthand notation for $\text{cov}[\mathbf{x}, \mathbf{x}]$.

On a graph, set of neighbors of node i is denoted $\mathcal{N}(i)$, which should not be confused with Gaussian or normal distribution $\mathcal{N}(x|\mu, \sigma^2)$. A functional is denoted $f[y]$ where $y(x)$ is some function. Concept of a functional is discussed in Appendix B. Curly braces $\{\}$ denote a set. Notation $g(x) = O(f(x))$ denotes $\left|\frac{f(x)}{g(x)}\right|$ is bounded as $x \to \inf$. E.g., if $g(x) = 3x^2 + 2$, then $g(x) = O(x^2)$.

If have N independent & identically distributed (i.i.d.) values $\mathbf{x}_1, \ldots, \mathbf{x}_N$ of a D-dimensional vector $\mathbf{x} = (x_1, \ldots, x_D)^\top$, can combine observations into a data matrix $\mathbf{X}$ of dimension $N \times D$ in which nth row of $\mathbf{X}$ corresponds to ith element of nth observation $\mathbf{x}_n$ & is written x_{ni}. For 1D variables, denote such a matrix by $\mathbf{X}$, which is a column vector whose nth element is x_n. Note $\mathbf{x}$ (which has dimensionality N) uses a different typeface to distinguish it from $\mathbf{x}$ (which has dimensionality D).

- 1. DL Revolution. ML today is 1 of most important, & fastest growing, fields of technology. Applications of ML are becoming ubiquitous, & solutions learned from data are increasingly displacing traditional hand-crafted algorithms. This has not only led to improved performance for existing technologies but has opened door to a vast range of new capabilities that would be inconceivable if new algorithms had to be designed explicitly by hand.

  1 particular branch of ML, known as `deep learning`, has emerged as an exceptionally powerful & general-purpose framework for learning from data. DL is based on computational models called *neural networks* which were originally inspired by mechanisms of learning & information processing in human brain. Field of *artificial intelligence*, or AI, seeks to recreate powerful capabilities of brain in machines, & today terms ML & AI are often used interchangeably. Many of AI systems in current use represent applications of ML which are designed to solve very specific & focused problems, & while these are extremely useful they fall far short of tremendous breadth of capabilities of human brain. This had led to introduction of term *artificial general intelligence*, or AGI, to describe aspiration of building machines with this much greater flexibility. After many decades of steady progress, ML has now entered a phase of very rapid development. Recently, massive DL systems called large language models have started to exhibit remarkable capabilities that have been described as 1st indications of artificial general intelligence (Bubeck et al., 2023).

  * 1.1. Impact of DL. Begin discussion of ML by considering 4 examples drawn from diverse fields to illustrate huge breadth of applicability of this technology & to introduce some basic concepts & terminology. What is particularly remarkable about these & many other examples is that they have all been addressed using variants of same fundamental framework of DL. This is in sharp contrast to conventional approaches in which different applications are tackled using widely differing & specialist techniques. Emphasize: examples chosen represent only a tiny fraction of breath of applicability for deep neural networks & almost every domain where computation has a role is amenable to transformational impact of DL.

    · 1.1.1. Medical diagnosis. Consider 1st application of ML to problem of diagnosing skin cancer. Melanoma (U hắc tố) is most dangerous kind of skin cancer but is curable if detected early. Fig. 1.1: Examples of skin lesions corresponding to dangerous malignant melanomas on top row & benign nevi on bottom row. Difficult for untrained eye to distinguish between these 2 classes. shows example images of skin lesions, with malignant melanomas on top row & benign nevi on bottom row. Distinguish between these 2 classes of image is clearly very challenging, & virtually impossible to write an algorithm by hand that could successfully classify such images with any reasonable level of accuracy.

    – Xem xét ứng dụng đầu tiên của ML vào vấn đề chẩn đoán ung thư da. U hắc tố (U hắc tố) là loại ung thư da nguy hiểm nhất nhưng có thể chữa khỏi nếu phát hiện sớm. Hình 1.1: Ví dụ về các tổn thương da tương ứng với các khối u ác tính nguy hiểm ở hàng trên cùng & nốt ruồi lành tính ở hàng dưới cùng. Khó để mắt thường có thể phân biệt giữa 2 loại này. hiển thị hình ảnh ví dụ về các tổn thương da, với khối u ác tính ở hàng trên cùng & nốt ruồi lành tính ở hàng dưới cùng. Việc phân biệt giữa 2 loại hình ảnh này rõ ràng là rất khó khăn, & hầu như không thể viết thủ công 1 thuật toán có thể phân loại thành công các hình ảnh như vậy với bất kỳ mức độ chính xác hợp lý nào.

    This problem has been successfully addressed using DL (Esteva et al., 2017). Solution was created using a large set of lesion images, known as a *training set*, each of which is labeled as either malignant or benign, where labels are obtained from biopsy test that is considered to provide true class of lesion. Training set is used to determine values of some 25 million adjustable parameters, known as *weights*, in a deep neural network. This process of setting parameter values from data is known as *learning* or *training*. goal: for trained network to predict correct label for a new lesion just from image alone without needing time-consuming step of taking a biopsy. This is an example of a *supervised learning* problem because, for each training example, network is told correct label. Also an example of a *classification* problem because each input must be assigned to a discrete set of classes (benign or malignant in this case). Applications in which output consist of 1 or more continuous variables are called *regression* problems. An example of a regression problem would be prediction of yield in a chemical manufacturing process in which inputs consist of temperature, pressure, & concentrations of reactants.

    An interesting aspect of this application: number of labeled training images available, roughly 129000, is considered relatively small, & so deep neural network was 1st trained on a much larger data set of 1.28 million images of everyday objects (e.g. dogs, buildings, & mushrooms) & then *fine-tuned* on data set of lesion images. An example of *transfer learning* in which network learns general properties of natural images from large data set of everyday objects & is then specialized to specific problem of lesion classification. Through use of DL, classification of skin lesion images has reached a level of accuracy that exceeds that of professional dermatologists (Brinker et al., 2019).

– 1 khía cạnh thú vị của ứng dụng này: số lượng hình ảnh đào tạo được gắn nhãn có sẵn, khoảng 129000, được coi là tương đối nhỏ, & do đó, mạng nơ-ron sâu đầu tiên được đào tạo trên 1 tập dữ liệu lớn hơn nhiều gồm 1,28 triệu hình ảnh về các vật thể hàng ngày (ví dụ: chó, tòa nhà, & nấm) & sau đó *được tinh chỉnh* trên tập dữ liệu hình ảnh tổn thương. Một ví dụ về *học chuyển giao* trong đó mạng học các thuộc tính chung của hình ảnh tự nhiên từ tập dữ liệu lớn về các vật thể hàng ngày & sau đó được chuyên môn hóa cho vấn đề cụ thể về phân loại tổn thương. Thông qua việc sử dụng DL, việc phân loại hình ảnh tổn thương da đã đạt đến mức độ chính xác vượt xa các bác sĩ da liễu chuyên nghiệp (Brinker và cộng sự, 2019).

· 1.1.2. Protein structure. Proteins are sometimes called *building blocks of living organisms*. They are biological molecules that consist of 1 or more long chains of units called *amino acids*, of which there are 22 different types, & protein is specified by sequence of amino acids. Once a protein has been synthesized inside a living cell, it folds into a complex 3D structure whose behavior & interactions are strongly determined by its shape. Calculating this 3D structure, given amino acid sequence, has been a fundamental open problem in biology for half a century that had seen relatively little progress until advent of DL.

3D structure can be measured experimentally using techniques e.g. X-ray crystallography, cryogenic electron microscopy, or nuclear magnetic resonance spectroscopy. However, this can be extremely time-consuming & for some proteins can prove to be challenging, e.g. due to difficulty of obtaining a pure sample or because structure is dependent on context. In contrast, amino acid sequence of a protein can be determined experimentally at lower cost & higher throughput (thông lượng). Consequently, there is considerable interest in being able to predict 3D structures of proteins directly from their amino acid sequences in order to better understand biological processes or for practical applications e.g. drug discovery. A DL model can be trained to take an amino acid sequence as input & generate 3D structure as output, in which training data consist of a set of proteins for which amino acid sequence & 3D structure are both known. Protein structure prediction is therefore another example of supervised learning. Once system is trained it can take a new amino acid sequence as input & can predict associated 3D structure (Jumper et al., 2021). Fig. 1.2: Illustration of 3D shape of a protein called T1044/6VR4. Green structure shows ground truth as determined by X-ray crystallography, whereas superimposed blue structure hows prediction obtained by a DL model called AlphaFold. compares predicted 3D structure of a protein & ground truth obtained by X-ray crystallography.

· 1.1.3. Image synthesis. In 2 applications discussed so far, a neural network learned to transform an input (a skin image or an amino acid sequence) into an output (a lesion classification or a 3D protein structure, resp.). Turn now to an example where training data consist simply of a set of sample images & goal of trained network: create new images of same kind. An example of *unsupervised learning* because images are unlabeled, in contrast to lesion classification & protein structure examples. Fig. 1.3: Synthetic face images generated by a deep neural network trained using unsupervised learning. shows examples of synthetic images generated by a deep neural network trained on a set of images of human faces taken in a studio against a plain background. Such synthetic images are of exceptionally high quality & it can be difficult tell them apart from photographs of real people.

An example of a *generative model* because it can generate new output examples that differ from those used to train model but which share same statistical properties. A variant of this approach allows images to be generated that depend on an input text string known, as a *prompt*, so that image content reflects semantics of text input. Term *generative AI* is used to describe DL learning models that generate outputs in form of images, video, audio, text, candidate drug molecules, or other modalities.

· 1.1.4. Large language models. 1 of most important advances in ML in recent years has been development of powerful models for processing natural language & other forms of sequential data e.g. source code. A *large language model*, or LLM, uses DL to build rich internal representations that capture semantic properties of language. An important class of large language models, called *autoregressive* language models, can generate language as output, & therefore, they are a form of generative AI. Such models take a sequence of words as input & for output, generate a single word that represents next word in sequence. Augmented sequence, with new word appended at end, can then be fed through model again to generate subsequent word, & this process can be repeated to generate a long sequence of words. Such models can also output a special 'stop' word that signals end of text generation, thereby allowing them to output text of finite length & then halt. At that point, a user could append their own series of words to sequence before feeding complete sequence back through model to trigger further word generation. In this way, possible for a human to have a conversation with neural network.

Such models can be trained on large data sets of text by extracting training pairs each consisting of a randomly selected sequence of words as input with known next word as target output. An example of *self-supervised learning* in which a function from inputs to outputs is learned but where labeled outputs are obtained automatically from input training data without needing separate human-derived labels. Since large volumes of text are available from multiple sources, this approach allows for scaling to very large training sets & associated very large neural networks. Large language models can exhibit extraordinary capabilities that have been described as 1st indications of emerging artificial general intelligence (Bubeck et al., 2023), & discuss such models at length later in book. Give an illustration of language generation, based on a model called GPT-4 (OpenAI, 2023), in response to an input prompt *'Write a proof of fact that there are infinitely many primes; do it in style of a Shakespeare play through a dialogue between 2 parties arguing over proof.'*

∗ 1.2. A Tutorial Example. For newcomer to field of ML, many of basic concepts & much of terminology can be introduced in context of a simple example involving fitting of a polynomial to a small synthetic data set (Bishop, 2006). This is a form of supervised learning problem in which would like to make a prediction for a target variable, given value of an

input variable.
– Đối với người mới tham gia lĩnh vực ML, nhiều khái niệm cơ bản & nhiều thuật ngữ có thể được giới thiệu trong bối cảnh của 1 ví dụ đơn giản liên quan đến việc khớp 1 đa thức với 1 tập dữ liệu tổng hợp nhỏ (Bishop, 2006). Đây là 1 dạng bài toán học có giám sát trong đó muốn đưa ra dự đoán cho 1 biến mục tiêu, với giá trị cho trước của 1 biến đầu vào.

· 1.2.1. Synthetic data. Denote input variable by x & target variable by t, & assume both variables take continuous values on real axis. Suppose: given a training set comprising N observations of x, written $x_1, \ldots, x_N$, together with corresponding observations of values of t, denoted $t_1, \ldots, t_N$. Goal: predict value of t for some new value of x. Ability to make accurate predictions on previously unseen inputs is a key goal in ML & is known as *generalization*.
Can illustrate this using a synthetic data set generated by sampling from a sinusoidal function. Fig. 1.4: Plot of a training data set of $N = 10$ points, each comprising an observation of input variable x along with corresponding target variable t. Green curve shows function $\sin 2\pi x$ used to generate data. Goal: predict value of t for some new value of x, without knowledge of green curve. shows a plot of a training set comprising $N = 10$ data points in which input values were generated by choosing values of x_n, for $n = 1, \ldots, N$, spaced uniformly in range $[0, 1]$. Associated target data values were obtained by 1st computing values of function $\sin 2\pi x$ for each value of x & then adding a small level of random noise (governed by a Gaussian distribution) to each such point to obtain corresponding target value t_n. By generating data in this way, we are capturing an important property of many real-world data sets, namely: they possess an underlying regularity, which wish to learn, but that individual observations are corrupted by random noise. This noise might arise from intrinsically *stochastic* (i.e., random) processes e.g. radioactive decay but more typically is due to there being sources of variability that are themselves unobserved.
In this tutorial example, know true process that generated data, namely sinusoidal function. In a practical application of ML, goal: discover underlying trends in data given finite training set. Knowing process that generated data, however, allows us to illustrate important concepts in ML.

· 1.2.2. Linear models. Goal: exploit this training set to predict value $\hat{t}$ of target variable for some new value $\hat{x}$ of input variable. This involves implicitly trying to discover underlying function $\sin 2\pi x$. This is intrinsically a difficult problem as we have to generalize from a finite data set to an entire function. Furthermore, observed data is corrupted with noise, & so for a given $\hat{x}$ there is uncertainty as to appropriate value for $\hat{t}$. *Probability theory* provides a framework for expressing such uncertainty as to appropriate value for $\hat{t}$. *Probability theory* provides a framework for expressing such uncertainty in a precise & quantitative manner, whereas *decision theory* allows us to exploit this probabilistic representation to make predictions that are optimal according to appropriate criteria. Learning probabilities from data lies at heart of ML & will be explored in great detail in this book.
To start with, however, will proceed rather informally & consider a simple approach based on curve fitting. In particular, will fit data using a polynomial function of form

$$y(x, \mathbf{w}) = \sum_{i=0}^{M} w_i x^i = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M, \tag{116}$$

where M: *order* of polynomial. Polynomial coefficients $w_0, \ldots, w_M$ are collectively denoted by vector $\mathbf{w}$. Note: although polynomial function $y(x, \mathbf{x})$ is a nonlinear function of x, it is a linear function of coefficients $\mathbf{w}$. Functions, e.g. this polynomial, that are linear in unknown parameters have important properties, as well as significant limitations, & are called *linear models*.

· 1.2.3. Error function. Values of coefficients will be determined by fitting polynomial to training data. This can be done by minimizing an *error function* that measures misfit between function $y(x, \mathbf{w})$, for any given value of $\mathbf{w}$, & training set data points. 1 simple choice of error function, which is widely used, is sum of squares of differences between predictions $y(x_n, \mathbf{w})$ for each data point x_n & corresponding target value t_n, given by

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} (y(x_n, \mathbf{w}) - t_n)^2, \tag{117}$$

where factor of $\frac{1}{2}$ is included for later convenience. Will derive this error function starting from probability theory. Here simply note: it is a nonnegative quantity that would be 0 iff function $y(x, \mathbf{w})$ were to pass exactly through each training data point. Geometrical interpretation of sum-of-squares error function is illustrated in Fig. 1.5: Error function $E(\mathbf{w})$ corresponds to $\frac{1}{2}$ sum of squares of displacements of each data point from function $y(x, \mathbf{w})$.
Can solve curve fitting problem by choosing value of $\mathbf{w}$ for which $E(\mathbf{w})$ is as small as possible. Because error function is a quadratic function of coefficients $\mathbf{w}$, its derivatives w.r.t. coefficients will be linear in elements $\mathbf{w}$, & so minimization of error function has a unique solution, denoted by $\mathbf{w}^\star$, which can be found in closed form. Resulting polynomial is given by function $y(x, \mathbf{w}^\star)$.

· 1.2.4. Model complexity. There remains problem of choosing order M of polynomial, & this will turn out to be an example of an important concept called *model comparison* or *model selection*. In Fig. 1.6: Plots of polynomials having various orders M, fitted to data set shown in Fig. 1.4 by minimizing error function $E(\mathbf{w})$, show 4 examples of results of fitting polynomials having orders $M = 0, 1, 3, 9$ to data set shown in Fig. 1.4.
Notice: constant ($M = 0$) & 1st-order ($M = 1$) polynomials give poor fits to data & consequently poor representations of function $\sin 2\pi x$. 3rd-order ($M = 3$) polynomial seems to give best fit to function $\sin 2\pi x$ of examples shown in

Fig. 1.6. When go to a much higher order polynomial ($M = 9$), obtain an excellent fit to training data. In fact, polynomial passes exactly through each data point & $E(\mathbf{w}^\star) = 0$. However, fitted curve oscillates wildly & gives a very poor representation of function $\sin 2\pi x$. This latter behavior is known as *over-fitting*.

Goal: achieve good generalization by making accurate predictions for new data. Can obtain some quantitative insight into dependence of generalization performance on M by considering a separate set of data known as a *test set*, comprising 100 data points generated using same procedure as used to generate training set points. For each value of M, can evaluate residual value of $E(\mathbf{w}^\star)$ for training data, & can also evaluate $E(\mathbf{w}^\star)$ for test data set. Instead of evaluating error function $E(\mathbf{w})$, sometimes more convenient to use root-mean-square (RMS) error defined by

$$E_{\mathrm{RMS}} = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(y(x_n, \mathbf{w}) - t_n)^2} \tag{118}$$

in which division by N allows us to compare different sizes of data sets on an equal footing, & square root ensures: E_{RMS} is measured on same scale (& in same units) as target variable t. Graphs of training-set & test-set RMS errors are shown, for various values of M, in Fig. 1.7: Graphs of root-mean-square error evaluated on training set, & on an independent test set, for various values of M. Test set error is a measure of how well we are doing in predicting values of t for new data observations of x. Note from Fig. 1.7: small values of M give relatively large values of test set error, & this can be attributed to fact: corresponding polynomials are rather inflexible & are incapable of capturing oscillations in function $\sin 2\pi x$. Values of M in range $3 \leq M \leq 8$ give small values for test set error, & there also give reasonable representations for generating function $\sin 2\pi x$, as can be seen for $M = 3$ in Fig. 1.6.

For $M = 9$, training set error $\to 0$, as might expect because this polynomial contains 10 degrees of freedom corresponding to 10 coefficients $w_0, \ldots, w_9$, & so can be tuned exactly to 10 data points in training set. However, test set error has become very large &, as saw in Fig. 1.6, corresponding function $y(x, \mathbf{w}^\star)$ exhibits wild oscillations.

This may seem paradoxical because a polynomial of a given order contains all lower-order polynomials as special cases. $M = 9$ polynomial is therefore capable to generating results at least as good as $M = 3$ polynomial. Furthermore, might suppose: best predictor of new data would be function $\sin 2\pi x$ from which data was generated (see later this is indeed the case). Know: a power series expansion of function $\sin 2\pi x$ contains terms of all orders, so might expect: results should improve monotonically as increase M.

Can gain some insight into problem by examining values of coefficients $\mathbf{w}^\star$ obtained from polynomials of various orders, as shown in Table 1.1: Table of coefficients $\mathbf{w}^\star$ for polynomials of various order. Observe how typical magnitude of coefficients increases dramatically as order of polynomial increases. As M increases, magnitude of coefficients typically gets larger. In particular, for $M = 9$ polynomial, coefficients have become finely tuned to data. They have large positive & negative values so that corresponding polynomial function matches each of data points exactly, but between data points (particularly near ends of range) function exhibits large oscillations observed in Fig. 1.6. Intuitively, what is happening: more flexible polynomials with larger values of M are increasingly tuned to random noise on target values.

Further insight into this phenomenon can be gained by examining behavior of learned model as size of data set is varied, as shown in Fig. 1.8: Plots of solutions obtained by minimizing sum-of-squares error function using $M = 9$ polynomial for $N = 15$ data points (left plot) & $N = 100$ data points (right plot). See: increasing size of data set reduces over-fitting problem. See: for a given model complexity, over-fitting problem become less severe as size of data set increases. Another way to say this: with a larger data set, can afford to fit a more complex (i.e., more flexible) model to data. 1 rough heuristic that is sometimes advocated in classical statistics: number of data points should be no less than some multiple (say 5 or 10) of number of learnable parameters in model. However, when discuss DL later, excellent results can be obtained using models that have significantly more parameters than number of training data points.

· 1.2.5. Regularization. There is something rather unsatisfying about having to limit number of parameters in a model according to size of available training set. It would seem more reasonable to choose complexity of model according to complexity of problem being solved. 1 technique often used to control overfitting phenomenon, as an alternative to limiting number of parameters, is that of *regularization*, which involves adding a penalty term to error function (1.2) to discourage coefficients from having large magnitudes. Simplest such penalty term takes form of sum of squares of all of coefficients, leading to a modified error function of form (1.4)

$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y(x_n, \mathbf{w}) - t_n)^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2, \tag{119}$$

where $\|\mathbf{w}\|^2 \coloneqq \mathbf{w}^\top\mathbf{w} = \sum_{i=0}^{M} w_i^2 = w_0^2 + w_1^2 + \cdots + w_M^2$, & coefficient λ governs relative importance of regularization term compared with sum-of-squares error term. Note: often coefficient w_0 is omitted from regularizer because its inclusion causes results to depend on choice of origin for target variable (Hastie, Tibshirani, & Friedman, 2009), or it may be included but with its own regularization coefficient. Again, error function in (1.4) can be minimized exactly in closed form. Techniques e.g. this are known in statistics literature as *shrinkage* (sự co rút) methods because they reduce value of coefficients. In context of neural networks, this approach is known as *weight decay* because parameters in a neural network are called weights & this regularizer encourages them to decay towards 0.

Fig. 1.9: Plots of $M = 9$ polynomials fitted to data set shown in Fig. 1.4 using regularized error function (1.4) for 2 values of regularization parameter λ corresponding to $\ln \lambda = -18$ & $\ln \lambda = 0$. Case of no regularizer, i.e., $\lambda = 0$, corresponding to $\ln \lambda = -\inf$, is shown at bottom right of Fig. 1.6. shows results of fitting polynomial of order $M = 9$ to same data set as before but now using regularized error function given by (1.4). See: for a value of $\ln \lambda = -18$, overfitting has been suppressed & now obtain a much closer representation of underlying function $\sin 2\pi x$. If, however, use too large a value for λ then again obtain a poor fit, as shown in Fig. 1.9 for $\ln \lambda = 0$. Corresponding coefficients from fitted polynomials are given in Table 1.2: Table of coefficients $\mathbf{w}^\star$ for $M = 9$ polynomials with various values for regularization parameter λ. Note $\ln \lambda = -\inf$ corresponds to a model with no regularization, i.e., to graph at bottom right in Fig. 1.6. See: as value of λ increases, magnitude of a typical coefficient gets smaller., showing: regularization has desired effect of reducing magnitude of coefficients.

Impact of regularization term on generalization error can be seen by plotting value of RMS error (1.3) for both training & test sets against $\ln \lambda$, as shown in Fig. 1.10: Graph of root-mean-square error (1.3) vs. $\ln \lambda$ for $M = 9$ polynomial. See: λ now controls effective complexity of model & hence determines degree of over-fitting.

- 1.2.6. Model selection. Quantity λ is an example of a *hyperparameter* whose values are fixed during minimization of error function to determine model parameters $\mathbf{w}$.

- 2. Probabilities.
- 3. Standard Distributions.
- 4. Single-layer Networks: Regression.
- 5. Single-layer Networks: Classification.
- 6. Deep Neural Networks.
- 7. Gradient Descent.
- 8. Backpropagation.
- 9. Regularization.
- 10. Convolutional Networks.
- 11. Structured Distributions.
- 12. Transformers.
- 13. Graph Neural Networks.
- 14. Sampling.
- 15. Discrete Latent Variables.
- 16. Continuous Latent Variables.
- 17. Generative Adversarial Networks.
- 18. Normalizing Flows.
- 19. Autoencoders.
- 20. Diffusion Models.
- Appendix A: Linear Algebra.
- Appendix B: Calculus of Variations.
- Appendix C: Lagrange Multipliers.

2. Arnulf Jentzen, Benno Kuckuck, Philippe von Wurstemberger. *Mathematical Introduction to Deep Learning: Methods, Implementations, & Theory.*

Keywords. DL, ANN, SGD, optimization.

Mathematics Subject Classification (2020): 68T07

All Python source codes in this book can be downloaded from https://github.com/introdeeplearning/book or from the arXiv page of this book (by clicking on "Other formats" & then "Download source").

Preface. Aim: provide an introduction to topic of DL algorithms. Very roughly speaking, when speak of a *deep learning algorithm*, think of a computational scheme which aims to approximate certain relations, functions, or quantities by means of so-called deep *artificial neural networks* (ANNs) & iterated use of some kind of data. ANNs, in turn, can be thought of as classes of functions that consist of multiple compositions of certain nonlinear functions, which are referred to as *activation functions*, & certain affine functions. Loosely speaking, depth of such ANNs corresponds to number of involved iterated compositions in ANN & one starts to speak of *deep* ANNs when number of involved compositions of nonlinear & affine functions > 2.

Hope this book will be useful for students & scientists who do not yet have any background in DL at all & would like to gain a solid foundations as well as for practitioners who would like to obtain a firmer mathematical understanding of objects & methods considered in DL.

After a brief intro, this book is divided into 6 parts.

- Part I
  - Chap. 1: introduce different types of ANNs including *fully-connected feedforward ANNs, convolutional ANNs (CNNs), recurrent ANNs (RNNs), & residual ANNs (ResNets)* in all mathematical details.

- Chap. 2: present a certain calculus for fully-connected feedforward ANNs.

- Part II: present several mathematical results that analyze how well ANNs can approximate given functions.
  - Chap. 3: to make this part more accessible, 1st restrict to 1D functions $\mathbb{R} \to \mathbb{R}$, thereafter
  - Chap. 4: study ANN approximation results for multivariate functions.
- Part III: A key aspect of DL algorithms is usually to model or reformulate problem under consideration as a suitable optimization problem involving deep ANNs. Subject of Part III: study such & related optimization problems & corresponding optimization algorithms to approximately solve such problems in detail. In particular, in context of DL methods such optimization problems – typically given in form of a minimization problem – are usually solved by means of appropriate *gradient based* optimization methods. Roughly speaking, think of a gradient based optimization method as a computational scheme which aims to solve considered optimization problem by performing successive steps based on direction of (negative) gradient of function which one wants to optimize.
  - Chap. 5: GD-type & SGD-type optimization methods can, roughly speaking, be viewed as time-discrete approximations of solutions of suitable *gradient flow (GF) ODEs*. To develop intuitions for GD-type & SGD-type optimization methods & for some of tolls which we employ to analyze such methods, study such GF ODEs. In particular, show in Chap. 5 how such GF ODEs can be used to approximately solve appropriate optimization problems.
  - Chap. 6: Review & study deterministic variants of such gradient based optimization methods e.g. *gradient descent* (GD) optimization method.
  - Chap. 7: Review & study stochastic variants of such gradient based optimization methods e.g. *stochastic gradient descent* (SGD) optimization method.
    Implementations of gradient based methods discussed in Chaps. 6–7 require efficient computations of gradients.
  - Chap. 8: Derive & present in detail the most popular & in some sense most natural method to explicitly compute such gradients in case of training of ANNs: *backpropagation* method.
    Mathematical analyses for gradient based optimization methods presented in Chaps. 5–7 are in almost all cases too restrictive to cover optimization problems associated to training of ANNs.
  - Chap. 9: However, such optimization problems can be covered by *Kurdyka–Łojasiewicz (KL)* approach.
  - Chap. 10: rigorously review *batch normalization (BN)* methods, which are popular methods that aim to accelerate ANN training procedures in data-driven learning problems.
  - Chap. 11: review & study approach to optimize an objective function through different random initializations.
- Part IV: Mathematical analysis of DL algorithms does not only consist of error estimates for approximation capacities of ANNs (cf. Part II) & of error estimates for involved optimization methods (cf. Part III) but also requires estimates for *generalization error* which, roughly speaking, arises when probability distribution associated to learning problem cannot be accessed explicitly but is approximated by a finite number of realizations/data. Precisely subject of Part IV to study generalization error.
  - Chap. 12: review suitable probabilistic generalization error estimates
  - Chap. 13: review suitable strong L^p-type generalization error estimates.
- Part V: illustrate how to combine parts of *approximation error* estimates from Part II, parts of *optimization error* estimates from Part III, & parts of *generalization error* estimates from Part IV to establish estimates for overall error in exemplary situation of training of ANNs based on SGD-type optimization methods with many independent random initializations.
  - Chap. 14: present a suitable overall error decomposition for supervised learning problems, which we employ in
  - Chap. 15 together with some of findings of Parts II–IV to establish aforementioned illustrative overall error analysis.
- Part VI: DL methods have not only become very popular for data-driven learning problems, but are nowadays also heavily used for approximately solving PDEs. In Part VI review & implement 3 popular variants of such DL methods for PDEs.
  - Chap. 16: treat *physics-informed neural networks* (PINNs) & *deep Galerkin methods* (DGMs).
  - Chap. 17: treat *deep Kolmogorov methods* (DKMs).

  This book contains a number of Python source codes, which can be downloaded from two sources, namely from the public GitHub repository at https://github.com/introdeeplearning/book & from arXiv page of this book (by clicking on link "Other formats" & then on "Download source"). For ease of reference, caption of each source listing in this book contains the filename of the corresponding source file.

3. Introduction. Very roughly speaking, field *deep learning* can be divided into 3 subfields, deep *supervised learning*, deep *unsupervised learning*, & deep *reinforcement learning*. Algorithms in deep supervised learning often seem to be most accessible for a mathematical analysis. Briefly sketch in a simplified situation some ideas of deep supervised learning.

   Let $d, M \in \mathbb{N}^\star, \mathcal{E} \in C(\mathbb{R}^d, \mathbb{R}), \mathbf{x}_1, \ldots, \mathbf{x}_{M+1} \in \mathbb{R}^d, y_1, \ldots, y_M \in \mathbb{R}$ satisfy $\forall m = 1, \ldots, M$ that (1)

$$y_m = \mathcal{E}(\mathbf{x}_m). \tag{120}$$

   In framework described in previous sentence, think of $M \in \mathbb{N}^\star$ as number of available known input-output data pairs, think of $d \in \mathbb{N}^\star$ as dimension of input data, think of $\mathcal{E} : \mathbb{R}^d \to \mathbb{R}$ as an unknown function which relates input & output data through (1), think of $\mathbf{x}_1, \ldots, \mathbf{x}_{M+1} \in \mathbb{R}^d$ as available known input data, & think of $y_1, \ldots, y_M \in \mathbb{R}$ as available known output data.

In context of a learning problem of type (1) objective: approximately compute output $\mathcal{E}(\mathbf{x}_{M+1})$ of $(M+1)$-th input data $\mathbf{x}_{M+1}$ without using explicit knowledge of function $\mathcal{E}:\mathbb{R}^d\to\mathbb{R}$ but instead by using knowledge of M input-output data pairs

$$(\mathbf{x}_1,y_1)=(\mathbf{x}_1,\mathcal{E}(\mathbf{x}_1)),\dots,(\mathbf{x}_M,y_M)=(\mathbf{x}_M,\mathcal{E}(\mathbf{x}_M))\in\mathbb{R}^d\times\mathbb{R}. \tag{121}$$

To accomplish this, one considers optimization problem of computing approximate minimizers of function $\mathcal{L}:C(\mathbb{R}^d,\mathbb{R})\to[0,\infty)$ which satisfies

$$\mathfrak{L}(\phi)=\frac{1}{M}\left(\sum_{i=1}^{M}|\phi(\mathbf{x}_i)-y_m|^2\right),\ \forall\phi\in C(\mathbb{R}^d,\mathbb{R}). \tag{122}$$

Observe: (1) ensures $\mathcal{L}(\mathcal{E})=0$ &, in particular, have: unknown function $\mathcal{E}:\mathbb{R}^d\to\mathbb{R}$ in (1) is a minimizer of function

$$\mathfrak{L}:C(\mathbb{R}^d,\mathbb{R})\to[0,\infty). \tag{123}$$

Optimization problem of computing approximate minimizers of function $\mathcal{L}$ is not suitable for discrete numerical computations on a computer as function $\mathcal{L}$ is defined on infinite dimensional vector space $C(\mathbb{R}^d,\mathbb{R})$.

To overcome this, introduce a spatially discretized version of this optimization problem. More specifically, let $\mathfrak{d}\in\mathbb{N}$, let $\psi=(\psi_\theta)_{\theta\in\mathbb{R}^\mathfrak{d}}:\mathbb{R}^\mathfrak{d}\to C(\mathbb{R}^d,\mathbb{R})$ be a function, & $\mathcal{L}:\mathbb{R}^\mathfrak{d}\to[0,\infty)$ satisfy

$$\mathcal{L}=\mathfrak{L}\circ\psi. \tag{124}$$

Think of set (6)

$$\{\psi_\theta:\theta\in\mathbb{R}^\mathfrak{d}\}\subseteq C(\mathbb{R}^d,\mathbb{R}) \tag{125}$$

as a parameterized set of functions which employ to approximate infinite dimensional vector space $C(\mathbb{R}^d,\mathbb{R})$ & think of function (7)

$$\mathbb{R}^\mathfrak{d}\ni\theta\mapsto\psi_\theta\in C(\mathbb{R}^d,\mathbb{R}) \tag{126}$$

as parameterization function associated to this set. E.g., in case $d=1$ one could think of (7) as parametrization function associated to polynomials in sense: $\forall\theta=(\theta_1,\dots,\theta_\mathfrak{d})\in\mathbb{R}^\mathfrak{d},x\in\mathbb{R}$ it holds: (8)

$$\psi_\theta(x)=\sum_{k=0}^{\mathfrak{d}-1}\theta_{k+1}x^k \tag{127}$$

or one could think of (7) as parametrization associated to trigonometric polynomials. However, in context of *deep supervised learning* one neither choose (7) as parametrization of polynomials nor as parametrization of trigonometric polynomials, but instead one chooses (7) as a parametrization associated to *deep* ANNs. In Chap. 1 in Part I, present different types of such deep ANN parametrization functions in all mathematical details.

Taking set in (6) & its parametrization function in (7) into account, then intend to compute approximate minimizers of function $\mathcal{L}$ restricted to set $\{\psi_\theta:\theta\in\mathbb{R}^\mathfrak{d}\}$, i.e., consider optimization problem of computing approximate minimizers of function (9)

$$\{\psi_\theta:\theta\in\mathbb{R}^\mathfrak{d}\}\ni\phi\mapsto\mathfrak{L}(\phi)=\frac{1}{M}\left(\sum_{m=1}^{M}|\phi(\mathbf{x}_m)-y_m|^2\right)\in[0,\infty). \tag{128}$$

Employing parametrization function in (7), one can also reformulate optimization problem in (9) as optimization problem of computing approximate minimizers of function (10)

$$\mathbb{R}^\mathfrak{d}\ni\theta\mapsto\mathcal{L}(\theta)=\mathfrak{L}(\psi_\theta)=\frac{1}{M}\left(\sum_{m=1}^{M}|\psi_\theta(\mathbf{x}_m)-y_m|^2\right)\in[0,\infty), \tag{129}$$

& this optimization problem now has potential to be amenable for discrete numerical computations. In context of deep supervised learning, where one chooses parametrization function in (7) as deep ANN parametrizations, one would apply an SGD-type optimization algorithm to optimization problem in (10) to compute approximate minimizers of (10). In Chap. 7 in Part III, present most common variants of such SGD-type optimization algorithms. If $\vartheta\in\mathbb{R}^\mathfrak{d}$ is an approximate minimizer of (10) in sense: $\mathcal{L}(\vartheta)\approx\inf_{\theta\in\mathbb{R}^\mathfrak{d}}\mathcal{L}(\theta)$, which is, however, typically not a minimizer of (10) in sense: $\mathcal{L}(\vartheta)\approx\inf_{\theta\in\mathbb{R}^\mathfrak{d}}\mathcal{L}(\theta)$, one then considers $\psi_\vartheta(\mathbf{x}_{M+1})$ as an approximation (11)

$$\psi_\vartheta(\mathbf{x}_{M+1})\approx\mathcal{E}(\mathbf{x}_{M+1}) \tag{130}$$

of unknown output $\mathcal{E}(\mathbf{x}_{M+1})$ of $(M+1)$th input data $\mathbf{x}_{M+1}$. Note: in deep supervised learning algorithms one typically aims to compute an approximate minimizer $\vartheta\in\mathbb{R}^\mathfrak{d}$ of (10) in sense: $\mathcal{L}(\vartheta)\approx\inf_{\theta\in\mathbb{R}^\mathfrak{d}}\mathcal{L}(\theta)$, which is, however, typically not a minimizer of (10) in sense that $\mathcal{L}(\vartheta)=\inf_{\theta\in\mathbb{R}^\mathfrak{d}}\mathcal{L}(\theta)$ (cf. Sect. 9.14).

In (3) above, have set up an optimization problem for learning problem by using standard mean squared error function to measure loss. This *mean squared error loss function* is just 1 possible example in formulation of DL optimization problems. In particular, in image classification problems other loss functions e.g. *cross-entropy loss function* are often used & refer to Chap. 5 of Part III for a survey of commonly used loss function in DL algorithms (see Sect. 5.4.2). Also refer to Chap. 9 for convergence results in above framework where parametrization function in (7) corresponds to *fully-connected feedforward* ANNs (see Sect. 9.14).

PART I. ARTIFICIAL NEURAL NETWORKS (ANNs).

- 1. Basics on ANNs. Review different types of architectures of ANNs e.g. fully-connected feedforward ANNs (Sects. 1.1 & 1.3), CNNs (Sect. 1.4), ResNets (Sect. 1.5), & RNNs (Sect. 1.6), review different types of popular activation functions used in applications e.g. *rectified linear unit* (ReLU) activation (Sect. 1.2.3), *Gaussian error linear unit* (GELU) activation (Sect. 1.2.6), & standard logistic activation (Sect. 1.2.7) among others, & review different procedures for how ANNs can be formulated in rigorous mathematical terms (see. Sect. 1.1 for a vectorized description & Sect. 1.3 for a structure description).
  In literature different types of ANN architectures & activation functions have been reviewed in several excellent works; cf., e.g., [4, 9, 39, 60, 63, 97, 164, 182, 189, 367, 373, 389, 431] & the references therein. The specific presentation of Sections 1.1 & 1.3 is based on [19, 20, 25, 159, 180].
  - 1.1. Fully-connected feedforward ANNs (vectorized description). Start mathematical content of this book with a review of fully-connected feedforward ANNs, most basic type of ANNs. Roughly speaking, fully-connected feedforward ANNs can be thought of as parametric functions resulting from successive compositions of affine functions followed by nonlinear functions, where parameters of a fully-connected feedforward ANN correspond to all entries of linear transformation matrices & translation vectors of involved affine functions (cf. Def. 1.1.3 below for a precise def of fully-connected feedforward ANNs & Fig. 1.2: Graphical illustration of an ANN. ANN has 2 hidden layers & length $L = 3$ with 3 neurons in input layer (corresponding to $l_0 = 3$), 6 neurons in 1st hidden layer (corresponding to $l_1 = 6$), 3 neurons in 2nd hidden layer (corresponding to $l_2 = 3$), & 1 neuron in output layer (corresponding to $l_3 = 1$). In this situation, have an ANN with 39 weight parameters & 10 bias parameters adding up to 49 parameters overall. Realization of this ANN is a function from $\mathbb{R}^3 \to \mathbb{R}$. for a graphical illustration of fully-connected feedforward ANNs). Linear transformaition matrices & translation vectors are sometimes called *weight matrices & bias vectors*, resp., & can be thought of as *trainable parameters* of fully-connected feedforward ANNs.
    Introduce in Def. 1.1.3 a *vectorized description* of fully-connected feedforward ANNs in sense: all trainable parameters of a fully-connected feedforward ANN are represented by components of a single Euclidean vector. Sect. 1.3: discuss an alternative way to describe fully-connected feedforward ANNs in which trainable parameters of a fully-connected feedforward ANN are represented by a tuple of matrix-vector pairs corresponding to weight matrices & bias vectors of fully-connected feedforward ANNs (cf. Defs. 1.3.1 & 1.3.4).
    Fig. 1.1: Graphical illustration of a fully-connected feedforward ANN consisting of $L \in \mathbb{N}$ affine transformations (i.e., consisting of $L + 1$ layers: 1 input layer, $L - 1$ hidden layers, & 1 output layer) with $l_0 \in \mathbb{N}$ neurons on input layer (i.e., with l_0-dimensional input layer), with $l_1 \in \mathbb{N}$ neurons on 1st hidden layer (i.e., with l_1-dimensional 1st hidden layer), with $l_2 \in \mathbb{N}$ neurons on 2nd hidden layer (i.e., with l_2-dimensional 2nd hidden layer),..., with l_{L-1} neurons on $(L-1)$-th hidden layer (i.e., with (l_{L-1})-dimensional $(L-1)$-th hidden layer), & with l_L neurons in output layer (i.e., with l_L-dimensional output layer).
    - 1.1.1. Affine functions.
      Definition 5 (Affine functions)**.** *Let $\mathfrak{d}, m, n \in \mathbb{N}, s \in \mathbb{N}_0, \theta = (\theta_1, \ldots, \theta_{\mathfrak{d}}) \in \mathbb{R}^{\mathfrak{d}}$ satisfy $\mathfrak{d} \geq s + mn + m$. Then denote by $\mathcal{A}^{\theta,s}_{m,n} : \mathbb{R}^n \to \mathbb{R}^m$ function which satisfies $\forall \mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$:*

$$\mathcal{A}^{\theta,s}_{m,n}(\mathbf{x}) = \left([\sum_{k=1}^{n} x_k \theta_{s+k}] + \theta_{s+mn+1}, [\sum_{k=1}^{n} x_k \theta_{s+n+k}] + \theta_{s+mn+2}, \ldots, [\sum_{k=1}^{n} x_k \theta_{s+(m-1)n+k}] + \theta_{s+mn+m} \right), \tag{131}$$

      & call $\mathcal{A}^{\theta,s}_{m,n}$ affine function *from $\mathbb{R}^n$ to $\mathbb{R}^m$ associated to (θ, s).*
    - 1.1.2. Vectorized description of fully-connected feedforward ANNs.
      Definition 6 (Vectorized description of fully-connected feedforward ANNs)**.** *Let $\mathfrak{d}, L \in \mathbb{N}, l_0, l_1, \ldots, l_L \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ satisfy*

$$\mathfrak{d} \geq \sum_{k=1}^{L} l_k (l_{k-1} + 1) \tag{132}$$

      & $\forall k \in \{1, 2, \ldots, L\}$ let $\Psi_k : \mathbb{R}^{l_k} \to \mathbb{R}^{l_k}$ be a function. Denote by $\mathcal{N}^{\theta,l_0}_{\Psi_1,\ldots,\Psi_L} : \mathbb{R}^{l_0} \to \mathbb{R}^{l_L}$ function which satisfies $\forall \mathbf{x} \in \mathbb{R}^{l_0}$:

$$(\mathcal{N}^{\theta,l_0}_{\Psi_1,\ldots,\Psi_L})(\mathbf{x}) = (\Psi_L \circ \mathcal{A}^{\theta,\sum_{k=1}^{L-1} l_k(l_{k-1}+1)}_{l_L,l_{L-1}} \circ \Psi_{L-1} \circ \mathcal{A}^{\theta,\sum_{k=1}^{L-2} l_k(l_{k-1}+1)}_{l_{L-1},l_{L-2}} \circ \Psi_2 \circ \mathcal{A}^{\theta,l_1(l_0+1)}_{l_2,l_1} \circ \Psi_1 \circ \mathcal{A}^{\theta,0}_{l_1,l_0})(\mathbf{x}), \tag{133}$$

      & call $\mathcal{N}^{\theta,l_0}_{\Psi_1,\ldots,\Psi_L}$ realization function *or* realization *of fully-connected feedforward ANN associated to θ with $L+1$ layers with dimensions $(l_0, l_1, \ldots, l_L)$ & activation functions $(\Psi_1, \ldots, \Psi_L)$.*
    - 1.1.3. Weight & bias parameters of fully-connected feedforward ANNs.
      Remark 10 (Weights & biases for fully-connected feedforward ANNs)**.** *Let $L \in \{2, 3, \ldots\}, v_0, v_1, \ldots, v_{L-1} \in \mathbb{N}_0, l_0, l_1, \ldots, l_L, \mathfrak{d} \in \mathbb{N}, \theta = (\theta_1, \ldots, \theta_{\mathfrak{d}}) \in \mathbb{R}^{\mathfrak{d}}$ satisfy $\forall k \in \{0, 1, \ldots, L-1\}$:*

$$\mathfrak{d} \geq \sum_{i=1}^{L} l_i (l_i + 1), \ v_k = \sum_{i=1}^{k} l_i (l_{i-1} + 1), \tag{134}$$

      let $W_k \in \mathbb{R}^{l_k \times l_{k-1}}, k \in \{1, \ldots, L\}, b_k \in \mathbb{R}^{l_k}, k \in \{1, \ldots, L\}$, satisfy $\forall k = 1, \ldots, L$:

$$W_k = \textit{weight parameters}, \ b_k = \textit{bias parameters}, \tag{135}$$

      & let $\Psi_k : \mathbb{R}^{l_k} \to \mathbb{R}^{l_k}, k \in \{1, \ldots, L\}$, be functions. Then

· *(i) it holds*

$$\mathcal{N}^{\theta,l_0}_{\Psi_1,\dots,\Psi_L} = \Psi_L \circ \mathcal{A}^{\theta,v_{L-1}}_{l_L,l_{L-1}} \circ \Psi_{L-1} \circ \mathcal{A}^{\theta,v_{L-2}}_{l_{L-1},l_{L-2}} \circ \Psi_{L-2} \circ \cdots \circ \mathcal{A}^{\theta,v_1}_{l_2,l_1} \circ \Psi_1 \circ \mathcal{A}^{\theta,v_0}_{l_1,l_0}, \tag{136}$$

· *(ii) it holds* $\forall k \in \{1,\dots,L\}, \mathbf{x} \in \mathbb{R}^{l_{k-1}}$ *that* $\mathcal{A}^{\theta,v_{k-1}}_{l_k,l_{k-1}}(\mathbf{x}) = W_k\mathbf{x} + b_k$.

○ 1.2. Activation functions. Review a few popular activation functions from literature (cf. Def. 1.1.2 & Def. 1.3.4 for use of activation functions in context of fully-connected feedforward ANNs, cf. Def. 1.4.5 below for use of activation functions in context of CNNs, cf. Def. 1.5.4 for use of activation functions in context of ResNets, & cf. Defs. 1.6.3 & 1.6.4 for use of activation functions in context of RNNs).

∗ 1.2.1. Multidimensional versions. To describe multidimensional activation functions, frequently employ concept of multidimensional version of a function.

Definition 7 (Multidimensional versions of 1D functions)**.** *Let* $T \in \mathbb{N}, d_1,\dots,d_T \in \mathbb{N}$ *& let* $\psi : \mathbb{R} \to \mathbb{R}$ *be a function. Then denote by*

$$\mathfrak{M}_{\psi,d_1,\dots,d_T} : \mathbb{R}^{d_1\times\cdots\times d_T} \to \mathbb{R}^{d_1\times\cdots d_T} \tag{137}$$

function which satisfies $\forall \mathbf{x} = (x_{k_1,\dots,k_T})_{k(1,\dots,k_T)\in(\times_{t=1}^{T}\{1,2,\dots,d_t\})} \in \mathbb{R}^{d_1\times\cdots\times d_T}, \mathbf{y} = (y_{k_1,\dots,k_T})_{k(1,\dots,k_T)\in(\times_{t=1}^{T}\{1,2,\dots,d_t\})} \in \mathbb{R}^{d_1\times\cdots\times d_T}$ *with* $\forall k_1 \in \{1,\dots,d_1\}, k_2 \in \{1,\dots,d_2\},\dots,k_T \in \{1,\dots,d_T\}$*:* $y_{k_1,\dots,k_T} = \psi(x_{k_1,\dots,k_T})$ *that*

$$\mathfrak{M}_{\psi,d_1,\dots,d_T}(\mathbf{x}) = \mathbf{y}, \tag{138}$$

& call $\mathfrak{M}_{\psi,d_1,\dots,d_T}$ $d_1 \times d_2 \times \cdots \times d_T$-dimensional version *of* ψ.

∗ 1.2.2. Single hidden layer fully-connected feedforward ANNs. Fig. 1.3: Graphical illustration of a fully-connected feedforward ANN consisting 2 affine transformations (i.e., consisting of 3 layers: 1 input layer, 1 hidden layer, & 1 output layer) with $\mathcal{I} \in \mathbb{N}$ neurons on input layer (i.e., with $\mathcal{I}$-dimensional input layer), with $\mathcal{H} \in \mathbb{N}$ neurons on hidden layer (i.e., with $\mathcal{H}$-dimensional hidden layer), & with 1 neuron in output layer (i.e., with 1D output layer).

Lemma 1 (Fully-connected feedforward ANN with 1 hidden layer)**.** *Let* $\mathcal{I},\mathcal{H} \in \mathbb{N}, \theta = (\theta_1,\dots,\theta_{\mathcal{H}\mathcal{I}+2\mathcal{H}+1}) \in \mathbb{R}^{\mathcal{H}\mathcal{I}+2\mathcal{H}+1}, \mathbf{x} = (x_1,\dots,x_{\mathcal{I}}) \in \mathbb{R}^{\mathcal{I}}$ *& let* $\psi : \mathbb{R} \to \mathbb{R}$ *be a function. Then*

$$\mathcal{N}_{\mathfrak{M}_\psi,\mathcal{H},\mathrm{id}_{\mathbb{R}}}(\mathbf{x}) = \left[\sum_{k=1}^{\mathcal{H}} \theta_{\mathcal{H}\mathcal{I}+\mathcal{H}+k}\,\psi\left(\left[\sum_{i=1}^{\mathcal{I}} x_i\theta_{(k-1)\mathcal{I}+i}\right] + \theta_{\mathcal{H}\mathcal{I}+k}\right)\right] + \theta_{\mathcal{H}\mathcal{I}+2\mathcal{H}+1}. \tag{139}$$

∗ 1.2.3. Rectified linear unit (ReLU) activation. Formulate ReLu functions which is 1 of most frequently used activation functions in DL applications (cf., e.g., [LBH15]).

Definition 8 (ReLU activation function)**.** *Denote by* $\mathfrak{r} : \mathbb{R} \to \mathbb{R}$ *the function which satisfies* $\forall x \in \mathbb{R}$*:* $\mathfrak{r}(x) = \max\{x, 0\}$ *& call* $\mathfrak{r}$ ReLU activation function *(call* $\mathfrak{r}$ rectifier function*).*

Definition 9 (Multidimensional ReLU activation functions)**.** *Let* $d \in \mathbb{N}$*. Then denote by* $\mathfrak{R}_d : \mathbb{R}^d \to \mathbb{R}^d$ *function given by* $\mathfrak{R}^d = \mathfrak{M}_{\mathfrak{r},d}$ *& call* $\mathfrak{R}^d$ d-dimensional ReLU activation function *(call* $\mathfrak{R}^d$ d-dimensional rectifier function*).*

Lemma 2 (An ANN with ReLU activation function as activation function)**.** *Let* $W_1 = w_1 = 1, W_2 = w_2 = -1, b_1 = b_2 = B = 0$*. Then it holds* $\forall x \in \mathbb{R}$*:*

$$x = W_1 \max\{w_1x + b_1, 0\} + W_2 \max\{w_2x + b_2, 0\} + B. \tag{140}$$

Problem 13 (Real identity)**.** *Prove or disprove following statement: There exist* $\mathfrak{d}, H \in \mathbb{N}, l_1,\dots,l_H \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ *with* $\mathfrak{d} \geq 2l_1 + [\sum_{k=2}^{H} l_k(l_{k-1}+1)] + l_H + 1$ *s.t.* $\forall x \in \mathbb{R}$*:* $(\mathcal{N}^{\theta,1}_{\mathfrak{R}_{l_1},\dots,\mathfrak{R}_{l_H},\mathrm{id}_{\mathbb{R}}})(x) = x$.

A partial answer:

Lemma 3 (Real identity)**.** *Let* $\theta = (1,-1,0,0,1,-1,0) \in \mathbb{R}^7$*. Then* $(\mathcal{N}^{\theta,1}_{\mathfrak{R}_2,\mathrm{id}_{\mathbb{R}}})(x) = x$.

Problem 14 (Absolute value)**.** *Prove or disprove: There exist* $\mathfrak{d}, H \in \mathbb{N}, l_1,\dots,l_H \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ *with* $\mathfrak{d} \geq 2l_1 + [\sum_{k=2}^{H} l_k(l_{k-1}+1)] + l_H + 1$ *s.t.* $\forall x \in \mathbb{R}$*,* $(\mathcal{N}^{\theta,1}_{\mathfrak{R}_{l_1},\dots,\mathfrak{R}_{l_H},\mathrm{id}_{\mathbb{R}}})(x) = |x|$.

Problem 15 (Exponential)**.** *Prove or disprove: There exist* $\mathfrak{d}, H \in \mathbb{N}, l_1,\dots,l_H \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ *with* $\mathfrak{d} \geq 2l_1 + [\sum_{k=2}^{H} l_k(l_{k-1}+1)] + l_H + 1$ *s.t.* $\forall x \in \mathbb{R}$*,* $(\mathcal{N}^{\theta,1}_{\mathfrak{R}_{l_1},\dots,\mathfrak{R}_{l_H},\mathrm{id}_{\mathbb{R}}})(x) = e^x$.

Problem 16 (2D maximum)**.** *Prove or disprove: There exist* $\mathfrak{d}, H \in \mathbb{N}, l_1,\dots,l_H \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ *with* $\mathfrak{d} \geq 3l_1 + [\sum_{k=2}^{H} l_k(l_{k-1}+1)] + l_H + 1$ *s.t.* $\forall x,y \in \mathbb{R}$*,* $(\mathcal{N}^{\theta,2}_{\mathfrak{R}_{l_1},\dots,\mathfrak{R}_{l_H},\mathrm{id}_{\mathbb{R}}})(x,y) = \max\{x,y\}$.

Problem 17 (Real identity with 2 hidden layers)**.** *Prove or disprove: There exist* $\mathfrak{d}, H \in \mathbb{N}, l_1,\dots,l_H \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ *with* $\mathfrak{d} \geq 2l_1 + l_1l_2 + 2l_2 + 1$ *s.t.* $\forall x \in \mathbb{R}$*,* $(\mathcal{N}^{\theta,1}_{\mathfrak{R}_{l_1},\mathfrak{R}_{l_2},\mathrm{id}_{\mathbb{R}}})(x) = x$.

A partial answer:

Lemma 4 (Real identity with 2 hidden layers)**.** *Let* $\theta = (1,-1,0,0,1,-1,-1,1,0,0,1,-1,0) \in \mathbb{R}^{13}$*. Then* $\forall x \in \mathbb{R}$*,* $(\mathcal{N}^{\theta,1}_{\mathfrak{R}_{l_1},\mathfrak{R}_{l_2},\mathrm{id}_{\mathbb{R}}})(x) = x$.

Problem 18 (3D maximum)**.** *Prove or disprove: There exist* $\mathfrak{d}, H \in \mathbb{N}, l_1,\dots,l_H \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ *with* $\mathfrak{d} \geq 4l_1 + [\sum_{k=2}^{H} l_k(l_{k-1}+1)] + l_H + 1$ *s.t.* $\forall x,y,z \in \mathbb{R}$*,* $(\mathcal{N}^{\theta,3}_{\mathfrak{R}_{l_1},\dots,\mathfrak{R}_{l_H},\mathrm{id}_{\mathbb{R}}})(x,y,z) = \max\{x,y,z\}$.

Problem 19 (Multidimensional maxima)**.** *Prove or disprove: For every* $k \in \mathbb{N}$*, there exists* $\mathfrak{d}, H \in \mathbb{N}, l_1,\dots,l_H \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ *with* $\mathfrak{d} \geq (k+1)l_1 + [\sum_{k=2}^{H} l_k(l_{k-1}+1)] + l_H + 1$ *s.t.* $\forall x,y,z \in \mathbb{R}$*,* $(\mathcal{N}^{\theta,k}_{\mathfrak{R}_{l_1},\dots,\mathfrak{R}_{l_H},\mathrm{id}_{\mathbb{R}}})(x_1,\dots,x_k) = \max\{x_1,\dots,x_k\}$.

Problem 20. *Prove or disprove: There exist* $\mathfrak{d}, H \in \mathbb{N}, l_1, \ldots, l_H \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ *with* $\mathfrak{d} \geq 2l_1 + [\sum_{k=2}^{H} l_k(l_{k-1}+1)] + l_H + 1$ *s.t.* $\forall x \in \mathbb{R}$, $(\mathcal{N}^{\theta,1}_{\mathfrak{R}_{l_1},\ldots,\mathfrak{R}_{l_H},\mathrm{id}_{\mathbb{R}}})(x) = \max\{x, \frac{x}{2}\}$.

Problem 21 (Hat function)**.** *Prove or disprove: There exist* $\mathfrak{d}, l \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ *with* $\mathfrak{d} \geq 3l+1$ *s.t.* $\forall x \in \mathbb{R}$*:* $(\mathcal{N}^{\theta,1}_{\mathfrak{R}_l,\mathrm{id}_{\mathbb{R}}})(x) = \mathbf{1}_{(-\infty,2]} + (x-1)\mathbf{1}_{(2,3]} + (5-x)\mathbf{1}_{(3,4]} + \mathbf{1}_{(4,\infty)}$.

[MANY PROBLEMS]

∗ 1.2.4. Clipping activation.

Definition 10 (Clipping (Cắt xén) activation function)**.** *Let* $u \in [-\infty, \infty), v \in (u, \infty]$*. Then denote by* $\mathfrak{c}_{u,v} : \mathbb{R} \to \mathbb{R}$ *function which satisfies* $\forall x \in \mathbb{R}$, $\mathfrak{c}_{u,v}(x) = \max\{u, \min\{x, v\}\}$ *& call* $\mathfrak{c}_{u,v}$ (u,v)-clipping activation function*.*

Fig. 1.5: A plot of $(0,1)$-clipping activation function & ReLU activation function.

- ◦ 1.3. Fully-connected feedforward ANNs (structured description).
- ◦ 1.4. Convolutional ANNs (CNNs).
- ◦ 1.5. Residual ANNs (ResNets).
- ◦ 1.6. Recurrent ANNs (RNNs).
- ◦ 1.7. Further types of ANNs.

- 2. ANN calculus.
  - ◦ 2.1. Compositions of fully-connected feedforward ANNs.
  - ◦ 2.2. Parallelizations of fully-connected feedforward ANNs.
  - ◦ 2.3. Scalar multiplications of fully-connected feedforward ANNs.
  - ◦ 2.4. Sums of fully-connected feedforward ANNs with same length.

PART II. APPROXIMATION.

- 3. 1D ANN approximation results.
- 3. Multi-dimensional ANN approximation results.

PART III. OPTIMIZATION.

- 5. Optimization through gradient flow (GF) trajectories.
- 6. Deterministic gradient descent (GD) optimization methods.
- 7. Stochastic gradient descent (SGD) optimization methods.
- 8. Backpropagation.
- 9. Kurdyka–Łojasiewicz (KL) inequalities.
- 10. ANNs with batch normalization.
- 11. Optimization through random initializations.

PART IV. GENERALIZATION.

- 12. Probabilistic generalization error estimates.
- 13. Strong generalization error estimates.

PART V. COMPOSED ERROR ANALYSIS.

- 14. Overall error decomposition.
- 15. Composed error estimates.

PART VI. DL FOR PDES.

- 16. Physics-informed neural networks (PINNs). DL methods have not only become very popular for data-driven learning problems, but are nowadays also heavily used for solving mathematical equations e.g. ODEs & PDEs (cf., e.g., [119, 187, 347, 379]). In particular, refer to overview articles [24, 56, 88, 145, 237, 355] & refs therein for numerical simulations & theoretical investigations for DL methods for PDEs.

  Often DL methods for PDEs are obtained, 1st, by reformulating PDE problem under consideration as an infinite dimensional stochastic optimization problem, then, by approximating infinite dimensional stochastic optimization problem through finite dimensional stochastic optimization problems involving deep ANNs as approximations for PDE solution &/or its derivatives, & therefore, by approximately solving resulting finite dimensional stochastic optimization problems through SGD-type optimization methods.

  Among most basic schemes of such DL learning methods for PDEs are PINNs & DGMs; see [347, 379]. In this chapter present in Thm. 16.1.1 in Sect. 16.1 a reformulation of PDE problems as stochastic optimization problems, use theoretical considerations from Sect. 16.1 to briefly sketch in Sect. 16.2 a possible derivation of PINNs & DGMs, & present in Sects. 16.3–16.4 numerical simulations for PINNs & DGMs. For simplicity & concreteness, restrict in this chap to case of semilinear heat PDEs. Specific presentation of this chap is based on Beck et al. [24].

- 16.1. Reformulation of PDE problems as stochastic optimization problems. Both PINNs & DGMs are based on reformulations of considered PDEs as suitable infinite dimensional stochastic optimization problems. Present theoretical result behind this reformulation in special case of semilinear heat PDEs.

  Theorem 1. *Let $T \in (0,\infty), d \in \mathbb{N}, g \in C^2(\mathbb{R}^d,\mathbb{R}), u \in C^{1,2}([0,T]\times\mathbb{R}^d,\mathbb{R}), \mathfrak{t} \in C([0,T],(0,\infty)), \mathfrak{x} \in C(\mathbb{R}^d,(0,\infty))$, assume that g has at most polynomially growing partial derivatives, let $(\Omega,\mathcal{F},\mathbb{P})$ be a probability space, let $\mathcal{T}:\Omega\to[0,T]$ & $\mathcal{X}:\Omega\to\mathbb{R}^d$ be independent random variables, assume $\forall A \in \mathcal{B}([0,T]), B \in \mathcal{B}(\mathbb{R}^d)$ that*

  $$\mathbb{P}(\mathcal{T}\in A) = \int_A \mathfrak{t}(t)\,\mathrm{d}t,\ \mathbb{P}(\mathcal{X}\in B) = \int_B \mathfrak{x}(\mathbf{x})\,\mathrm{d}\mathbf{x}, \tag{141}$$

  let $f:\mathbb{R}\to\mathbb{R}$ be Lipschitz continuous, & let $\mathfrak{L}: C^{1,2}([0,T]\times\mathbb{R}^d,\mathbb{R})\to[0,\infty]$ satisfy $\forall v = (v(t,\mathbf{x}))_{(t,\mathbf{x})\in[0,T]\times\mathbb{R}^d} \in C^{1,2}([0,T]\times\mathbb{R}^d,\mathbb{R})$ that

  $$\mathfrak{L}(v) = \mathbb{E}[|v(0,\mathcal{X}) - g(\mathcal{X})|^2 + |(\partial_t v)(\mathcal{T},\mathcal{X}) - (\Delta_{\mathbf{x}} v)(\mathcal{T},\mathcal{X}) - f(v(\mathcal{T},\mathcal{X}))|^2]. \tag{142}$$

  Then 2 statements are equivalent:
  (i) It holds that $\mathfrak{L}(u) = \inf_{v\in C^{1,2}([0,T]\times\mathbb{R}^d,\mathbb{R})}\mathfrak{L}(v)$.
  (ii) It holds $\forall t\in[0,T], \mathbf{x}\in\mathbb{R}^d$ that $u(0,\mathbf{x}) = g(\mathbf{x})$ &

  $$\partial_t u(t,\mathbf{x}) = (\Delta_{\mathbf{x}} u)(t,\mathbf{x}) + f(u(t,\mathbf{x})). \tag{143}$$

- 16.2. Derivation of PINNs & deep Galerkin methods (DGMs). Employ reformulation of semilinear PDEs as optimization problems from Thm. 16.1.1 to sketch an informal derivation of DL schemes to approximate solutions of semilinear heat PDEs. For this let $T\in(0,\infty), d\in\mathbb{N}, u\in C^{1,2}([0,T]\times\mathbb{R}^d,\mathbb{R}), g\in C^2(\mathbb{R}^d,\mathbb{R})$ satisfy: g has at most polynomial growing partial derivatives, let $f:\mathbb{R}\to\mathbb{R}$ be Lipschitz continuous, & assume $\forall t\in[0,T], \mathbf{x}\in\mathbb{R}^d$ that $u(0,\mathbf{x}) = g(\mathbf{x})$ &

  $$\partial_t u(t,\mathbf{x}) = (\Delta_{\mathbf{x}} u)(t,\mathbf{x}) + f(u(t,\mathbf{x})). \tag{144}$$

  In framework described in previous sentence, think of u as unknown PDE solution. Objective of this derivation: develop DL methods which aim to approximate unknown function u.
  In 1st step employ Thm. 16.1.1 to reformulate PDE problem associated to (16.10) as an infinite dimensional stochastic optimization problem over a function space. For this let $\mathfrak{t}\in C([0,T](0,\infty)), \mathfrak{x}\in C(\mathbb{R}^d,(0,\infty))$, let $(\Omega,\mathcal{F},\mathbb{P})$ be a probability space, let $\mathcal{T}:\Omega\to[0,T], \mathcal{X}:\Omega\to\mathbb{R}^d$ be independent random variables, assume $\forall A\in\mathcal{B}([0,T]), B\in\mathcal{B}(\mathbb{R}^d)$ that

  $$\mathbb{P}(\mathcal{T}\in A) = \int_A \mathfrak{t}(t)\,\mathrm{d}t,\ \mathbb{P}(\mathcal{X}\in B) = \int_B \mathfrak{x}(\mathbf{x})\,\mathrm{d}\mathbf{x}, \tag{145}$$

  & let $\mathfrak{L}: C^{1,2}([0,T]\times\mathbb{R}^d,\mathbb{R})\to[0,\infty]$ satisfy $\forall v = (v(t,\mathbf{x}))_{(t,\mathbf{x})\in[0,T]\times\mathbb{R}^d}\in C^{1,2}([0,T]\times\mathbb{R}^d,\mathbb{R})$:

  $$\mathfrak{L}(v) = \mathbb{E}[|v(0,\mathcal{X}) - g(\mathcal{X})|^2 + |\partial_t v(\mathcal{T},\mathcal{X}) - (\Delta_{\mathbf{x}} v)(\mathcal{T},\mathcal{X}) - f(v(\mathcal{T},\mathcal{X}))|^2]. \tag{146}$$

  Observe: Thm. 16.1.1 assures: unknown function u satisfies $\mathfrak{L}(u) = 0$ & is thus a minimizer of optimization problem associated to (16.12). Motivated by this, consider aim to find approximations of u by computing approximate minimizers of function $\mathfrak{L}: C^{1,2}([0,T]\times\mathbb{R}^d,\mathbb{R})\to[0,\infty]$. Due to its infinite dimensionality this optimization problem is however not yet amenable to numerical computations.
  For this reason, in 2nd step, reduce this infinite dimensional stochastic optimization problem to a finite dimensional stochastic optimization problem involving ANNs. Specifically, let $a:\mathbb{R}\to\mathbb{R}$ be differentiable, let $h\in\mathbb{N}, l_1,\dots,l_h,\mathfrak{d}\in\mathbb{N}$ satisfy $\mathfrak{d} = l_1(d+2) + [\sum_{k=2}^{h} l_k(l_{k-1}+1)] + l_h + 1$, & let $\mathcal{L}:\mathbb{R}^{\mathfrak{d}}\to[0,\infty)$ satisfy $\forall\theta\in\mathbb{R}^{\mathfrak{d}}$:

  $$\mathcal{L}(\theta) = \dots \tag{147}$$

  (cf. Defs. 1.1.3 & 1.2.1). Can now compute an approximate minimizer of function $\mathcal{L}$ by computing an approximate minimizer $\vartheta\in\mathbb{R}^{\mathfrak{d}}$ of function $\mathcal{L}$ & employing realization $\mathcal{N}_{\mathfrak{M}_{a,l_1},\mathfrak{M}_{a,l_2},\dots,\mathfrak{M}_{a,l_h},\mathrm{id}_{\mathbb{R}}}$ of ANN associated to this approximate minimizer as an approximate minimizer of $\mathcal{L}$.
  3rd & last step of this derivation is to approximately compute such an approximate minimizer of $\mathcal{L}$ by means of SGD-type optimization methods. Now sketch this in case of plain-vanilla SGD optimization method (cf. Def. 7.2.1). Let $\xi\in\mathbb{R}^{\mathfrak{d}}, J\in\mathbb{N}, (\gamma_n)_{n\in\mathbb{N}}\subseteq[0,\infty), \forall n\in\mathbb{N}, j\in\{1,2,\dots,J\}$ let $\mathfrak{T}_{n,j}:\Omega\to[0,T]$ & $\mathfrak{X}_{n,j}:\Omega\to\mathbb{R}^d$ be random variables, assume $\forall n\in\mathbb{N}$, $j\in\{1,\dots,J\}$, $A\in\mathcal{B}([0,T]), B\in\mathcal{B}(\mathbb{R}^d)$:

  $$\mathbb{P}(\mathcal{T}\in A) = \mathbb{P}(\mathfrak{T}_{n,j}\in A),\ \mathbb{P}(\mathcal{X}\in B) = \mathbb{P}(\mathfrak{X}_{n,j}\in B), \tag{148}$$

  [DIFFICULT!!!]
- 16.3. Implementation of PINNs.
- 16.4. Implementation of DGMs.

- 17. Deep Kolmogorov methods (DKMs).
  - 17.1. Stochastic optimization problems for expectations of random variables.

- 17.2. Stochastic optimization problems for expectations of random fields.
- 17.3. Feymann–Kac formulas.
  - 17.3.1. Feynman–Kac formulas providing existence of solutions.
  - 17.3.1. Feynman–Kac formulas providing uniqueness of solutions.
- 17.4. Reformulation of PDE problems as stochastic optimization problems.
- 17.5. Derivation of DKMs.
- 17.6. Implementation of DKMs.

• 18. Further DL methods for PDEs.
- 18.1. DL methods based on strong formulations of PDEs.
- 18.2. DL methods based on weak formulations of PDEs.
- 18.3. DL methods based on stochastic representations of PDEs.
- 18.4. Error analyzes for DL methods for PDEs.

4. Phillip Petersen, Jakob Zech. *Mathematical Theory of Deep Learning*. Oct 14, 2023.

Preface. This book serves as an introduction to key ideas in mathematical analysis of DL. Designed to help students & researchers to quickly familiarize themselves with area & to provide a foundation for development of university courses on mathematics of DL. Main goal in composition of this book was to present various rigorous, but easy to grasp, results that help to build an understanding of fundamental mathematical concepts in DL. To achieve this, prioritize simplicity over generality.

As a mathematical introduction to DL, this book does not aim to give an exhaustive survey of entire (& rapidly growing) field, & some important research directions are missing. In particular, have favored mathematical results over empirical research, even though an accurate account of theory of DL requires both.

Book is intended for students & researchers in mathematics & related areas. While believe: every diligent (siêng năng) researcher or student will be able to work through this manuscript, emphasize: a familiarity with analysis, linear algebra, probability theory, & basic functional analysis is recommended for an optimal reading experience. To assist readers, a review of key concepts in probability theory & functional analysis is provided in appendix.

Material is structured around 3 main pillars of DL theory: Approximation theory, Optimization theory, & Statistical Learning theory. Chap. 1 provides an overview & outlines key questions for understanding DL. Chaps. 2–9 explore results in approximation theory, Chaps. 10–13 discuss optimization theory for DL, & remaining Chaps. 14–16 address statistical aspects of DL.

This book is result of a series of lectures given by authors. Parts of material were presented by P.P. in a lecture titled "Neural Network Theory" at University of Vienna, & by J.Z. in a lecture titled "Theory of Deep Learning" at Heidelberg University. Lecture notes of these courses formed basis of book.

• 1. Introduction.
- 1.1. Mathematics of DL. In 2012, a DL architecture revolutionized field of computer vision by achieving unprecedented performance in ImageNet Large Scale Visual Recognition Challenge (ILSVRC). DL architecture, known as AlexNet, significantly outperformed all competing technologies. A few years later, in Mar 2015, a DL-based architecture called AlphaGo defeated best Go player at time, Lee Sedol, in a 5-game match. Go is a highly complex board game with a vast number of possible moves, making it a challenging problme for AI. Because of this complexity, many researchers believed: defeating a top human Go player was a feat that would only be achieved decades later.
  These breakthroughs, along with many others including DeepMind's AlphaFold, which revolutionized protein structure prediction in 2020, unprecedented language capabilities of large language models like GPT-3 (& later versions), & emergence of generative AI models like Stable Diffusion, Midjourney, & DALL-E, have sparked interest among scientists across (almost) all disciplines. Likewise, while mathematical research on neural networks has a long history, these groundbreaking developments revived interest in theoretical underpinnings of DL among mathematicians. However, initially, there was a clear consensus in mathematics community: *We do not understand why this technology works so well! In fact, there are many mathematical reasons that, at least superficially (ít nhất là bề ngoài), should prevent observed success.*
  Over past decade field has matured, & mathematicians have gained a more profound understanding of DL, although many open questions remain. Recent years have brought various new explanations & insights into inner workings of DL models. Before discussing these in detail in following chaps, 1st give a high-level introduction to DL, with a focus on supervised learning framework – central theme of this book.
- 1.2. High-level overview of DL. DL refers to application of deep neural networks trained by gradient-based methods, to identify unknown input-output relationships. This approach has 3 key ingredients: *deep neural networks, gradient-based training, & prediction*. Now explain each of these ingredients separately.
  * Deep Neural Networks. Deep neural networks are formed by a combination of neurons. A *neuron* is a function of form

$$\mathbb{R}^d \ni \mathbf{x} \mapsto \nu(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b), \tag{149}$$

  where $\mathbf{w} \in \mathbb{R}^d$: a *weight vector*, $b \in \mathbb{R}$ is called *bias*, & function σ is referred to as an *activation function*. This concept is due to McCulloch & Pitts [142] & is a mathematical model for biological neurons. If consider σ to be Heaviside function $\sigma = \mathbf{1}_{\mathbb{R}_+}$ with $\mathbb{R}_+ \coloneqq [0, \infty)$, then neuron "fires" if weighted sum of inputs $\mathbf{x}$ surpasses threshold $-b$. Depict

a neuron in Fig. 1.1: Illustration of a single neuron ν. Neuron receives 6 inputs $(x_1, \dots, x_6) = \mathbf{x}$ computes their weighted sum $\sum_{i=1}^{6} x_i w_i$, adds a bias b, & finally applies activation function σ to produce output $\nu(\mathbf{x})$. Note: if fix d & σ, then set of neurons can be naturally parameterized by $d+1$ real values $w_1, \dots, w_d, b \in \mathbb{R}$.

Neural networks are functions formed by connecting neurons, where output of 1 neuron becomes input to another. 1 simple but very common type of neural network is so-called feedforward neural network. This structure distinguishes itself by having neurons grouped in layers, & inputs to neurons in $(l+1)$-st layer are exclusively neurons from lth layer. Start by defining a *shallow feedforward neural network* as an affine transformation applied to output of a set of neurons that share same input & same activation function. Here, an *affine transformation* is a map $T : \mathbb{R}^p \to \mathbb{R}^q$ s.t. $T(\mathbf{x}) = \mathbf{W}\mathbf{x} + \mathbf{b}$ for some $\mathbf{W} \in \mathbb{R}^{q \times p}, \mathbf{b} \in \mathbb{R}^q$ where $p, q \in \mathbb{N}$.

Formally, a shallow feedforward neural network is, therefore, a map Φ of form

$$\mathbb{R}^d \ni \mathbf{x} \mapsto \Phi(\mathbf{x}) = T_1 \circ \sigma \circ T_0(\mathbf{x}), \tag{150}$$

where T_0, T_1: affine transformations & application of σ is understood to be in each component of $T_1(\mathbf{x})$. A visualization of a shallow neural network: Fig. 1.2: Illustration of a shallow neural network. Affine transformation T_0 of form $(x_1, \dots, x_6) = \mathbf{x} \mapsto \mathbf{W}\mathbf{x} + \mathbf{b}$, where rows of $\mathbf{W}$: weight vectors $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$ for each respective neuron.

A *deep feedforward neural network* is constructed by compositions of shallow neural networks. This yields a map of type

$$\mathbb{R}^d \ni \mathbf{x} \mapsto \Phi(\mathbf{x}) = T_{L+1} \circ \sigma \circ \cdots T_1 \circ \sigma \circ T_0(\mathbf{x}), \tag{151}$$

where $L \in \mathbb{N}$ & $(T_i)_{i=0}^{L+1}$: affine transformations. Number of compositions L is referred to as *number of layers* of deep neural network. Similar to a single neuron, (deep) neural networks can be viewed as a parameterized function class, with *parameters* being entries of matrices & vectors determining affine transformations $(T_i)_{i=0}^{L+1}$.

∗ Gradient-based training. After defining structure or *architecture* of neural network, e.g., activation function & number of layers, 2nd step of DL consists of determining optimal values for its parameters. This optimization is carried out by minimizing an objective function. In *suppervised learning* – our focus – this objective depends on a collection of input-output pairs known as a *sample*. Concretely, let $S = (\mathbf{x}_i, \mathbf{y}_i)_{i=1}^m$ be a sample, where $\mathbf{x}_i \in \mathbb{R}^d$ represents inputs & $\mathbf{y}_i \in \mathbb{R}^k$ corresponding outputs with $d, k \in \mathbb{N}$. Goal: find a deep neural network Φ s.t. (1.2.2)

$$\Phi(\mathbf{x}_i) \approx \mathbf{y}_i, \ \forall i = 1, \dots, m, \tag{152}$$

in a meaningful sense. E.g., could interpret "≈" to mean closeness w.r.t. Euclidean norm, or more generally, $\mathcal{L}(\Phi(\mathbf{x}_i), \mathbf{y}_i)$ is small for a function $\mathcal{L}$ measuring dissimilarity between its inputs. Such a function $\mathcal{L}$ is called a *loss function*. A standard way of achieving (1.2.2) is by minimizing so-called *empirical risk of* Φ *w.r.t. sample* S defined as

$$\widehat{\mathcal{R}}_S(\Phi) \coloneqq \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\Phi(\mathbf{x}_i), \mathbf{y}_i). \tag{153}$$

if $\mathcal{L}$ is differentiable, & $\forall \mathbf{x}_i$, output $\Phi(\mathbf{x}_i)$ depends differentiably on parameters of neural network, then gradient of empirical risk $\widehat{\mathcal{R}}_S(\Phi)$ w.r.t. parameters is well-defined. This gradient can be efficiently computed using a technique called *backpropagation*. This allows to minimize (1.2.3) by optimization algorithms e.g. (stochastic) gradient descent. They produce a sequence of neural networks parameters, & corresponding neural network function $\Phi_1, \Phi_2, \dots$, for which empirical risk is expected to decrease. Fig. 1.3: A sequence of 1D neural networks $\Phi_1, \dots, \Phi_4$ that successfully minimizes empirical risk for sample $S = (x_i, y_i)_{i=1}^6$ illustrates a possible behavior of this sequence.

∗ Prediction. Final part of DL concerns question of whether we have actually learned something by procedure above. Suppose: our optimization routine has either converged or has been terminated, yielding a neural network Φ_*. While optimization aimed to minimize empirical risk on training sample S, our ultimate interest is not in how well Φ_* performs on S. Rather, interested in its performance on new, unseen data points $(\mathbf{x}_{\text{new}}, \mathbf{y}_{\text{new}})$. To make meaningful statements about this performance, need to assume a relationship between training sample S & other data points.

Standard approach: assume existence of a *data distribution* $\mathcal{D}$ on input-output space – in our case: $\mathbb{R}^d \times \mathbb{R}^k$ – s.t. both elements of S & all other considered data points are drawn from this distribution. I.e., treat S as an i.i.d. draw from $\mathcal{D}$, & $(\mathbf{x}_{\text{new}}, \mathbf{y}_{\text{new}})$ also sampled independently from $\mathcal{D}$. If want Φ_* to perform well on average, then this amounts to controlling expression

$$\mathcal{R}(\Phi_*) = \mathbb{E}_{(\mathbf{x}_{\text{new}}, \mathbf{y}_{\text{new}}) \sim \mathcal{D}}[\mathcal{L}(\Phi_*(\mathbf{x}_{\text{new}}), \mathbf{y}_{\text{new}})], \tag{154}$$

which is called *risk* of Φ_*. If risk is not much larger than empirical risk, then say: neural network Φ_* has a small *generalization error*. On other hand, if risk is much larger than empirical risk, then say: Φ_* *overfits* training data, meaning: Φ_* has memorized training samples, but does not generalize well to new data.

◦ 1.3. Why does it work? Natural to wonder why DL pipeline, ultimately succeeds in learning, i.e., achieving a small risk. True?: for a given sample $(\mathbf{x}_i, \mathbf{y}_i)_{i=1}^m$ there exist a neural network s.t. $\Phi(\mathbf{x}_i) \approx \mathbf{y}_i$, $\forall i = 1, \dots, m$. Does optimization routine produce a meaningful result? Can we control risk, knowing only: empirical risk is small?

While most of these questions can be answered affirmatively under certain assumptions, these assumptions often do not apply to DL in practice. Next explore some potential explanations & explanations & show that they lead to even more questions.

* Approximation. A fundamental result in study of neural networks is so-called universal approximation theorem, discussed in Chap. 3. This result states: every continuous function on a compact domain can be approximated arbitrary well (in a uniform sense) by a shallow neural network.
  This result, however, does not answer questions that are more specific of DL, e.g. question of efficiency. E.g., if aim for computational efficiency, then might be interested in smallest neural network that fits data. This raises question: *What is role of architecture for expensive capabilities of neural networks?* Furthermore, if consider reducing empirical risk an approximation problem, are confronted with 1 of main issues of approximation theory, which is *curse of dimensionality*. Function approximation in high dimensions is notoriously difficult & gets exponentially harder with increasing dimension. In practice, many successful DL architectures operate in this high-dimensional regime. *Why do these neural networks not seem to suffer from curse of dimensionality?*
* Optimization. While gradient descent can sometimes be proven to converge to a global minimum discussed in Chap. 10, this typically requires objective function to be at least convex. However, there is no reason to believe: e.g., empirical risk is a convex function of network parameters. In fact, due to repeatedly occurring compositions with nonlinear activation function in network, empirical risk is typically *highly nonlinear & not convex*. Therefore, there is generally no guarantee: optimization routine will converge to a global minimum, & may get stuck in a local (& non-global) minimum or a saddle point. *Why is output of optimization nonetheless often meaningful in practice?*
* Generalization. In traditional statistical learning theory, reviewed in Chap. 14, extent to which risk exceeds empirical risk, can be bounded a priori; such bounds are often expressed in terms of a notion of complexity of set of admissible functions (class of neural networks) divided by number of training samples. For class of neural networks of a fixed architecture, complexity roughly amounts to number of neural network parameters. In practice, typically neural networks with *more* parameters than training samples are used. This is dubbed *overparameterized regime* (chế độ). In this regime, classical estimates described above are void.
  Why is it that, nonetheless, *deep overparameterized architectures are capable of making accurate predictions* on unseen data? Furthermore, while deep architectures often generalize well, they sometimes fail spectacularly on specific, carefully crafted examples. In image classification tasks, these examples may differ only slightly from correctly classified images in a way that is not perceptible to human eye. Such examples are known as *adversarial example* (ví dụ đối nghịch), & their existence poses a great challenge for applications of DL.

◦ 1.4. Outline & philosophy. This book addresses questions raised in previous sect, providing answers that are mathematically rigorous & accessible. Our focus will be on provable statements, presented in a manner that prioritizes simplicity & clarity over generality. Will sometimes illustrate key ideas only in special cases, or under strong assumptions, both to avoid an overly technical exposition, & because definitive answers are often not yet available. In following, summarize content of each chapter & highlight parts pertaining to questions stated in previous sect.

* **Chap. 2. Feedforward neural networks.** Introduce main object study of this book: feedforward neural network.
* **Chap. 3: Universal approximation.** Present classical view of function approximation by neural networks, & give 2 instances of so-called universal approximation results. Such statements describe ability of neural networks to approximate every function of a given class to arbitrary accuracy, given that network size is sufficiently large. 1st result, which holds under very broad assumptions on activation function, is on uniform approximation of continuous functions on compact domains. 2nd result shows: for a very specific activation function, network size can be chosen independent of desired accuracy, highlighting: universal approximation needs to be interpreted with caution.
* **Chap. 4: Splines.** Going beyond universal approximation, this chap starts to explore approximate rates of neural networks. Specifically, examine how well certain functions can be approximated relative to number of parameters in network. For so-called sigmoidal activation functions, establish a link between neural-network- & spline-approximation. This reveals: smoother functions require fewer network parameters. However, achieving this increased efficiency necessitates use of deep neural networks. This observation offers a 1st glimpse into *importance of depth in DL*.
* **Chap. 5: ReLU neural networks.** Focus on 1 of most popular activation functions in practice – ReLU. Prove: class of ReLU networks is equal to set of continuous piecewise linear functions, thus providing a theoretical foundation for their expressive power. Furthermore, given a continuous piecewise linear function, investigate necessary width & depth of a ReLU network to represent it. Finally, leverage approximation theory for piecewise linear functions to derive convergence rates for approximating Hölder continuous functions.
* **Chap. 6: Affine pieces for ReLU neural networks.** Having gained some intuition about ReLU neural networks, address some potential limitations. Analyze ReLU neural networks by counting number of affine regions that they generate. Key insight of this chap: deep neural networks can generate exponentially more regions than shallow ones. This observation provides *further evidence for potential advantages of depth* in neural network architectures.
* **Chap. 7: Deep ReLU neural networks.** Having identified ability of deep ReLU neural networks to generate a large number of affine regions, investigate whether this translates into an actual advantage in function approximation. Indeed, for approximating smooth functions, prove substantially better approximation rates than obtained for shallow neural networks. This adds again to our *understanding of depth & its connections to expressive power* of neural network architectures.
* **Chap. 8: High-dimensional approximation.** Convergence rates established in previous chaps deteriorate significantly in high-dimensional settings. This chap examines 3 scenarios under which neural networks can provably *overcome curse of dimensionality*.

- **Chap. 9: Interpolation.** Shift our perspective from approximation to exact interpolation of training data. Analyze conditions under which exact interpolation is possible, & discuss implications for empirical risk minimization. Furthermore, present a constructive proof showing: ReLU networks can express an optimal interpolant of data (in a specific sense).
- **Chap. 10: Training of neural networks.** Start to examine training process of DL. 1st, study fundamentals of (stochastic) gradient descent & convex optimization. Then, discuss how backpropagation algorithm can be used to implement these optimization algorithms for training neural networks. Finally, examine accelerated methods & highlight key principles behind popular & more advanced training algorithms e.g. Adam.
- **Chap. 11: Wide neural networks & neural tangent kernel.** Introduce neural tangent kernel as a tool for analyzing training behavior of neural networks. Begin by revisiting linear & kernel regression for approximation of functions based on data. Afterwards, demonstrate in an abstract setting that under certain assumptions, training dynamics of gradient descent for neural networks resemble those of kernel regression, converging to a global minimum. Using standard initialization schemes, then show: assumptions for such a statement to hold are satisfied with high probability, if network is sufficiently wide (overparameterized). This analysis provides insights into why, under certain conditions, can train neural networks *without getting stuck in (bad) local minima*, despite non-convexity of objective function. Additionally, discuss a well-known link between neural networks & Gaussian processes, giving some indication why overparameterized networks *do not necessarily overfit* in practice.
- **Chap. 12: Loss landscape analysis.** Present an alternative view on optimization problem, by analyzing loss landscape – empirical risk as a function of neural network parameters. Give theoretical arguments showing: increasing overparameterization leads to greater connectivity between valleys & basins of loss landscape. Consequently, overparamterized architectures make it easier to reach a region where all minima are global minima. Additionally, observe: most stationary points associated with non-global minima are saddle points. This sheds further light on empirically observed fact: deep architectures can often be optimized *without getting stuck in non-global minima*.
- **Chap. 13: Shape of neural network spaces.** While Chaps. 11–12 highlight potential reasons for success of neural network training, in this chap, show: set of neural networks of a fixed architecture has some undesirable properties from an optimization perspective. Specifically, show: this set is typically non-convex. Moreover, in general it does not possess best-approximation property, meaning: there might not exist a neural network within set yielding best approximation for a given function.
- **Chap. 14: Generalization properties of deep neural networks.** To understand why deep neural networks successfully generalize to unseen data points (outside of training set), study classical statistical learning theory, with a focus on neural network functions as hypothesis class. Then show how to establish generalization bounds for DL, providing theoretical insights into *performance on unseen data*.
- **Chap. 15: Generalization in overparameterized regime.** Generalization bounds of previous chap are not meaningful when number of parameters of a neural network surpasses number of training samples. However, this overparameterized regime is where many successful network architectures operate. To gain a deeper understanding of generalization in this regime, describe phenomenon of double descent & present a potential explanation. This addresses question of why deep neural networks *perform well despite being highly overparameterized*.
- **Chap. 16: Robustness & adversarial examples.** In final chap, explore existence of adversarial examples – inputs designed to deceive neural networks. Provide some *theoretical explanations of why adversarial examples arise*, & discuss potential strategies to prevent them.

- 1.5. Material not covered in this book. This book studies some central topics of DL but leaves out even more. Interesting questions associated with field that were omitted, as well as some pointers to related works:
  - **Advanced architectures.** (Deep) Forward neural network is far from only type of neural network. In practice, architectures must be adapted to type of data. E.g., images exhibit strong spatial dependencies in sense that adjacent pixels often have similar values. Convolutional neural networks are particularly well suited for this type of input, as they employ convolutional filters that aggregate information from neighboring pixels, thus capturing data structure better than a fully connected feedforward network. Similarly, graph neural networks are a natural choice for graph-based data. For sequential data, e.g. natural language, architectures with some form of memory component are used, including Long Short-Term Memory (LSTM) networks & attention-based architectures like transformers.
  - **Interpretability/Explainability & Fairness.** Use of deep neural networks in critical decision-making processes, e.g. allocating scarce resource (e.g., organ transplants in medicine, financial credit approval, hiring decisions) or engineering (e.g., optimizing bridge structures, autonomous vehicle navigation, predictive maintenance), necessitates an understanding of their decision-making process. This is crucial for both practical & ethical reasons.
    Practically, understanding how a model arrives at a decision can help us improve its performance & mitigate problems. It allows us to ensure: model performs according to our intentions & does not produce undesirable outcomes. E.g., in bridge design, understanding why a model suggests or rejects a particular configuration can help engineers identify potential vulnerabilities, ultimately leading to safer & more efficient designs. Ethically, transparent decision-making is crucial, especially when outcomes have significant consequences for individuals or society; biases present in data or model design can lead to discriminatory outcomes, making explainability essential.
    However, explaining predictions of deep neural networks is not straightforward. Despite knowledge of network weights & biases, repeated & complex interplay of linear transformations & nonlinear activation functions often readers these models black boxes. A comprehensive overview of various techniques for interpretability, not only for deep neural networks, can be found in C. Molnar. *Interpretable machine learning*. Regarding the topic of fairness, see refs.

* **Unsupervised & Reinforcement Learning.** While this book focuses on supervised learning, where each data point x_i has a label y_i, there is a vast field of ML called *unsupervised learning*, where labels are absent. Classical unsupervised learning problems include clustering & dimensionality reduction.
  A popular area in DL, where no labels are used, is physics-informed neural networks [M. Raissi, P. Perdikaris, & G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward & inverse problems involving nonlinear partial differential equations. Journal of Computational physics, 378:686–707, 2019.]. Here, a neural network is trained to satisfy a PDE, with loss function quantifying deviation from this PDE.
  Finally, reinforcement learning is a technique where an agent can interact with an environment & receives feedback based on its actions. Actions are guided by a so-called *policy*, which is to be learned, [148, Chapter 17]. In deep reinforcement learning, this policy is modeled by a deep neural network. Reinforcement learning is basis of aforementioned AlphaGo.
* **Implementation.** While this book focuses on provable theoretical results, field of DL is strongly driven by applications, & a thorough understanding of DL cannot be achieved without practical experience. For this, there exist numerous resources with excellent explanations. Recommend [67, 38, 182] as well as the countless online tutorials that are just a Google (or alternative) search away.
* **Many more.** Field is evolving rapidly, & new ideas are constantly being generated & tested. This book cannot give a complete overview. However, hope: provide reader with a solid foundation in fundamental knowledge & principles to quickly grasp & understand new developments in field.

Bibliography & further reading. In this introductory chap, highlight several other recent textbooks & works on DL. For a historical survey on neural networks see [J. Schmidhuber. Deep learning in neural networks: An overview. Neural Networks, 61:85–117, 2015.] & [LBH15]. For general textbooks on neural networks & DL, refer to [84, 72, 182] for more recent monographs. A more mathematical introduction to topic is given, e.g., in 3, 107, 29]. For the implementation of neural networks [67, 38].

- 2. Feedforward neural networks. Feedforward neural networks, henceforth simply referred to as neural networks (NNs), constitute central object of study of this book. In this chap, provide a formal def of neural networks, discuss *size* of a neural network, & give a brief overview of common activation functions.

  - 2.1. Formal def. Defined a single neuron ν in (149) & Fig. 1.1. A neural network is constructed by connecting multiple neurons. Make precise this connection procedure:

  Definition 11 (Neural network)**.** *Let $L \in \mathbb{N}, d_0, \dots, d_{L+1} \in \mathbb{N}$, & let $\sigma : \mathbb{R} \to \mathbb{R}$. A function $\Phi : \mathbb{R}^{d_0} \to \mathbb{R}^{d_{L+1}}$ is called a* neural network *if there exist matrices $\mathbf{W}^{(l)} \in \mathbb{R}^{d_{l+1} \times d_l}$ & vectors $\mathbf{b}^{(l)} \in \mathbb{R}^{d_{l+1}}$, $l = 0, \dots, L$, s.t. with (2.1.1)*

$$\mathbf{x}^{(0)} \coloneqq \mathbf{x}, \tag{155}$$

$$\mathbf{x}^{(l)} \coloneqq \sigma(\mathbf{W}^{(l-1)}\mathbf{x}^{(l-1)} + b^{(l-1)}), \ \forall l \in \{1, \dots, L\}, \tag{156}$$

$$\mathbf{x}^{(L+1)} \coloneqq \mathbf{W}^{(L)}\mathbf{x}^{(L)} + \mathbf{b}^{(L)} \tag{157}$$

holds

$$\Phi(\mathbf{x}) = \mathbf{x}^{(L+1)}, \ \forall \mathbf{x} \in \mathbb{R}^{d_0}. \tag{158}$$

Call L depth*, $d_{\max} = \max_{l=1,\dots,L} d_l$* width*, σ:* activation function*, & $(\sigma; d_0, \dots, d_{L+1})$* architecture *of neural network Φ. Moreover, $\mathbf{W}^{(l)} \in \mathbb{R}^{d_{l+1} \times d_l}$:* weight matrices *& $\mathbf{b}^{(l)} \in \mathbb{R}^{d_{l+1}}$:* bias vectors *of Φ for $l = 0, \dots, L$.*

Remark 11. *Typically, there exist different choices of architectures, weights, & biases yielding same function $\Phi : \mathbb{R}^{d_0} \to \mathbb{R}^{d_{L+1}}$. For this reason, cannot associate a unique meaning to these notions solely based on* function *realized by Φ. In following, when refer to properties of a neural network Φ, always understood to mean: there exists at least 1 construction as in Def. 2.1, which realizes function Φ & uses parameters that satisfy those properties.*

Architecture of a neural network is often depicted as a connected graph, as illustrated in Fig. 2.1: Sketch of a neural network with 3 hidden layers, & $d_0 = 3, d_1 = 4, d_2 = 3, d_3 = 4, d_4 = 2$. Neural network has depth 3 & width 4. *Nodes* in such graphs represent (output of) neurons. They are arranged in *layers*, with $\mathbf{x}^{(l)}$ in Def. 2.1 corresponding to neurons in layer l. Also refer to $\mathbf{x}^{(0)}$ in (2.1.1a) as *input layer* & to $\mathbf{x}^{(L+1)}$ in (2.1.1c) as *output layer*. All layers in between are referred to as *hidden layers* & their output is given by (2.1.1b). Number of hidden layers corresponds to depth. For correct interpretation of such graphs, note: by our conventions in Def. 2.1, activation function is applied after each affine transformation, except in final layer.
Neural networks of depth 1 are called *shallow*, if depth is larger than 1 they are called *deep*. Notion of deep neural networks is not used entirely consistently in literature, & some authors use word deep only in case depth is much larger than 1, where precise meaning of "much larger" depends on application.
Throughout, only consider neural networks in sense of Def. 2.1. Emphasize however: this is just 1 (simple but very common) type of neural network. Many adjustments to this construction are possible & also widely used. E.g.:

* May use *different activation functions* σ_l in each layer l or may even use a different activation function for each node.
* *Residual* neural networks allow "skip connections". I.e., information is allowed to skip layers in sense: nodes in layer l may have $\mathbf{x}^{(0)}, \dots, \mathbf{x}^{(l-1)}$ as their input (& not just $\mathbf{x}^{(l-1)}$).
* In contrast to feedforward neural networks, *recurrent* neural networks allow information to flow backward, in sense: $\mathbf{x}^{(l-1)}, \dots, \mathbf{x}^{(L+1)}$ may serve as input for nodes in layer l (& not just $\mathbf{x}^{(l-1)}$). This creates loops in flow of information, & one has to introduce a time index $t \in \mathbb{N}$, as output of a node in time step t might be different from output in time step $t + 1$.

Clarify some further common terminology used in context of neural network:

* **parameters.** Parameters of a neural network refer to set of all entries of weight matrices & bias vectors. These are often collected in a single vector
$$\mathbf{w} = ((\mathbf{W}^{(0)}, \mathbf{b}^{(0)}), \dots, (\mathbf{W}^{(L)}, \mathbf{b}^{(L)})). \tag{159}$$
These parameters are adjustable & are learned during training process, determining specific function realized by network.
* **hyperparameters.** Hyperparameters are settings that define network's architecture (& training process), but are not directly learned during training. Examples include depth, number of neurons in each layer, & choice of activation function. They are typically set before training begins.
* **weights.** Term "weights" is often used broadly to refer to *all* parameters of a neural network, including both weight matrices & bias vectors.
* **model.** For a fixed architecture, every choice of network parameters $\mathbf{w}$ as in (159) defines a specific function $\mathbf{x} \mapsto \Phi_{\mathbf{w}}(\mathbf{x})$. In DL this function is often referred to as a model. More generally, "model" can be used to describe any function parameterization by a set of parameters $\mathbf{w} \in \mathbb{R}^n, n \in \mathbb{N}$.
* 2.1.1. Basic operations on neural networks. There are various ways how neural networks can be combined with 1 another. Next proposition addresses this for linear combinations, compositions, & parallelization. Formal proof, which is a good exercise to familiarize oneself with neural networks.

Proposition 8. *For 2 neural networks Φ_1, Φ_2, with architectures $(\sigma; d_0^1, d_1^1, \dots, d_{L_1+1}^1), (\sigma; d_0^2, d_1^2, \dots, d_{L_2+1}^2)$ resp., holds:*

(i) $\forall \alpha \in \mathbb{R}$ exists a neural network Φ_α with architecture $(\sigma; d_0^1, d_1^1, \dots, d_{L_1+1}^1)$ s.t. $\Phi_\alpha(\mathbf{x}) = \alpha\Phi_1(\mathbf{x})$, $\forall \mathbf{x} \in \mathbb{R}^{d_0^1}$,

(ii) if $d_0^1 = d_0^2 =: d_0, L_1 = L_2 =: L$, then there exists a neural network Φ_{parallel} with architecture $(\sigma; d_0, d_1^1 + d_1^2, \dots, d_{L+1}^1 + d_{L+1}^2)$ s.t. $\Phi_{\text{parallel}}(\mathbf{x}) = (\Phi_1(\mathbf{x}), \Phi_2(\mathbf{x}))$, $\forall \mathbf{x} \in \mathbb{R}^{d_0}$,

(iii) if $d_0^1 = d_0^2 =: d_0$, $L_1 = L_2 =: L$, & $d_{L+1}^1 = d_{L+1}^2 =: d_{L+1}$, then there exists a neural network Φ_{sum} with architecture $(\sigma; d_0, d_1^1 + d_1^2, \dots, d_L^1 + d_L^2, d_{L+1})$ s.t. $\Phi_{\text{sum}}(\mathbf{x}) = \Phi_1(\mathbf{x}) + \Phi_2(\mathbf{x})$, $\forall \mathbf{x} \in \mathbb{R}^{d_0}$,

(iv) if $d_{L_1+1}^1 = d_0^2$, then there exists a neural network Φ_{comp} with architecture $(\sigma; d_0^1, d_1^1, \dots, d_{L_1}^1, d_1^2, \dots, d_{L_2+1}^2)$ s.t. $\Phi_{\text{comp}}(\mathbf{x}) = \Phi_2 \circ \Phi_1(\mathbf{x})$, $\forall \mathbf{x} \in \mathbb{R}^{d_0^1}$.

◦ 2.2. Notion of size. Neural networks provide a framework to parameterize functions. Ultimately, goal: find a neural network that fits some underlying input-output relation. Architecture (depth, width, & activation function) is typically chosen a priori & considered fixed. During training of neural network, its parameters (weights & biases) are suitably adapted by some algorithm. Depending on application, on top of stated architecture choices, further restrictions on weights & biases can be desirable. E.g., following 2 appear frequently:

* **weight sharing.** a technique where specific entries of weight matrices (or bias vectors) are constrained to be equal. Formally, this means imposing conditions of form $W_{k,l}^{(i)} = W_{s,t}^{(j)}$, i.e., entry (k,l) of ith weight matrix is equal to entry at position (s,t) of weight matrix j. Denote this assumption by $(i,k,l) \sim (j,s,t)$, paying tribute to trivial fact: "$\sim$" is an equivalence relation. During training, shared weights are updated jointly, meaning: any change to 1 weight is simultaneously applied to all other weights of this class. Weight sharing can also be applied to entries of bias vectors.
* **sparsity.** This refers to imposing a sparsity structure on weight matrices (or bias vectors). Specifically, apriorily set $W_{k,l}^{(i)} = 0$ for certain (k,l,i), i.e., impose entry (k,l) of ith weight matrix to be 0. These zero-valued entries are considered fixed, & are not adjusted during training. Condition $W_{k,l}^{(i)} = 0$ corresponds to node l of layer $i-1$ *not* serving as an input to node k in layer i. If represent neural network as a graph, this is indicated by not connecting corresponding nodes. Sparsity can also be imposed on bias vectors.

Both of these restrictions decrease number of learnable parameters in neural network. Number of parameters can be seen as a measure of complexity of represented function class. For this reason, introduce size(Φ) as a notion for number of learnable parameters. Formally (with $|S|$ denoting cardinality of a set S):

Definition 12 (Size of neural network)**.** *Let Φ be as in Def. 2.1. Then* size *of Φ is*
$$\text{size}(\Phi) := \left| \left(\{(i,k,l) | W_{k,l}^{(i)} \neq 0\} \cup \{(i,k) | b_k^{(i)} \neq 0\} \right) / \sim \right|. \tag{160}$$

◦ 2.3. Activation functions. Activation functions are a crucial part of neural networks, as they introduce nonlinearity into model. If an affine activation function were used, resulting neural network function would also be affine & hence very restricted in what it can represent.

Choice of activation function can have a significant impact on performance, but there does not seem to be a universally optimal one. Discuss a few important activation functions & highlight some common issues associated with them.

* **Sigmoid.** Sigmoid activation function is given by
$$\sigma_{\text{sig}}(x) = \frac{1}{1+e^{-x}}, \ \forall x \in \mathbb{R}. \tag{161}$$
Its output ranges between 0 & 1, making it interpretable as a a probability. Sigmoid is a smooth function, which allows application of gradient-based training.

It has disadvantage: its derivative $\frac{d}{dx}\frac{1}{1+e^{-x}} = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{e^x}{(e^x+1)^2} = \frac{1}{e^x+1} - \frac{1}{(e^x+1)^2} \in (0, \frac{1}{4}]$ becomes very small if $|x| \to \infty$. This can affect learning due to so-called *vanishing gradient problem*. Consider simple neural network $\Phi_n(x) = \sigma \circ \cdots \sigma(x+b)$ defined with $n \in \mathbb{N}$ compositions of σ, & where $b \in \mathbb{R}$ is a bias. Its derivative w.r.t. b is

$$\frac{d}{db}\Phi_n(x) = \sigma'(\Phi_{n-1}(x))\frac{d}{db}\Phi_{n-1}(x). \tag{162}$$

If $\sup_{x\in\mathbb{R}} |\sigma'(x)| \leq 1-\delta$, then by induction, $|\frac{d}{db}\Phi_n(x)| \leq (1-\delta)^n$. Opposite effect happens for activation functions with derivatives uniformly > 1. This argument shows: derivative of $\Phi_n(x,b)$ w.r.t. b can become exponentially small or exponentially large when propagated through layers. This effect, known as *vanishing- or exploding gradient effect*, also occurs for activation functions which do not admit uniform bounds assumed above. However, since sigmoid activation function exhibits areas with extremely small gradients, vanishing gradient effect can be strongly exacerbated. – Tuy nhiên, vì hàm kích hoạt sigmoid thể hiện các vùng có độ dốc cực kỳ nhỏ nên hiệu ứng độ dốc biến mất có thể bị trầm trọng hơn nhiều.

* **ReLU (Rectified Linear Unit).** ReLU is defined as $\sigma_{\mathrm{ReLU}}(x) = \max\{x, 0\}$, for $x \in \mathbb{R}$. It is piecewise linear, & due to its simplicity its evaluation is computationally very efficient. It is 1 of most popular activation functions in practice. Since its derivative is always 0 or 1, it does not suffer from vanishing gradient problem to same extent as sigmoid function. However, ReLU can suffer from so-called *dead neurons* problem. Consider neural network

$$\Phi(x) = \sigma_{\mathrm{ReLU}}(b - \sigma_{\mathrm{ReLU}}(x)) \ \forall x \in \mathbb{R} \tag{163}$$

depending on bias $b \in \mathbb{R}$. If $b < 0$, then $\Phi(x) = 0$, $\forall x \in \mathbb{R}$. Neuron corresponding to 2nd application of σ_{ReLU} thus produces a constant signal. Moreover, if $b < 0$, $\frac{d}{db}\Phi(x) = 0$, $\forall x \in \mathbb{R}$. As a result, every negative value of b yields a stationary point of empirical risk. A gradient-based method will not be able to further train parameter b. Thus refer to this neuron as a dead neuron.

* **SiLU (Sigmoid Linear Unit).** An important difference between ReLU & Sigmoid: ReLU is not differentiable at 0. SiLU activation function (also referred to as "swish" (quẹt)) can be interpreted as a smooth approximation to ReLU. It is defined as

$$\sigma_{\mathrm{SiLU}}(x) \coloneqq x\sigma_{\mathrm{sig}}(x) = \frac{x}{1+e^{-x}}, \ \forall x \in \mathbb{R}. \tag{164}$$

There exists various other smooth activation functions that mimic ReLU, including Softplus $x \mapsto \log(1+e^x)$, GELU (Gaussian Error Linear Unit) $x \mapsto xF(x)$ where $F(x)$ denotes cumulative distribution function of standard normal distribution, & Mish $x \mapsto x\tanh(\log(1+e^x))$.

* **Parametric ReLU or Leaky ReLU.** This variant of ReLU addresses dead neuron problem. For some $a \in (0,1)$, parametric ReLU is defined as

$$\sigma_a(x) = \max\{x, ax\}, \ \forall x \in \mathbb{R}, \tag{165}$$

depicted in Fig. 2.2c for 3 different values of a. Since output of σ does not have flat regions like ReLU, dying ReLU problem is mitigated. If a is not chosen too small, then there is less of a vanishing gradient problem than for Sigmoid. In practice, additional parameter a has to be fine-tuned depending on application. Like ReLU, parametric ReLU is not differentiable at 0.

Bibliography & further reading. Concept of neural networks was 1st introduced by McCulloch & Pitts in [142]. Later Rosenblatt [192] introduced perceptron (a fully connected feedforward neural network). Vanishing gradient problem shortly addressed in Sect. 2.3 was discussed by Hochreiter in his diploma thesis [91] & later in [17, 93].

Problem 22. *Show ReLU & parametric ReLu create similar sets of neural network functions. Fix $a > 0$. (i) Find a set of weight matrices & biases vectors, s.t. associated neural network Φ_1, with ReLU activation function σ_{ReLU} satisfies $\Phi_1(x) = \sigma_a(x)$, $\forall x \in \mathbb{R}$. (ii) Find a set of weight matrices & biases vectors, s.t. associated neural network Φ_2 with parametric ReLU activation function σ_a satisfies $\Phi_2(x) = \sigma_{\mathrm{ReLU}}(x)$, $\forall x \in \mathbb{R}$. (iii) Conclude: every ReLU neural network can be expressed as a leaky ReLU neural network & vice versa.*

Problem 23. *Show: for sigmoid activation functions, dead-neuron-like behavior is very rare. Let Φ be a neural network with sigmoid activation function. Assume: Φ is a constant function. Show: $\forall \varepsilon > 0$, there is a non-constant neural network $\widetilde{\Phi}$ with same architecture as Φ s.t. $\forall l = 0, \ldots, L$, $\|\mathbf{W}^{(l)} - \widetilde{\mathbf{W}}^{(l)}\| \leq \varepsilon$, $\|\mathbf{b}^{(l)} - \widetilde{\mathbf{b}}^{(l)}\| \leq \varepsilon$ where $\mathbf{W}^{(l)}, \mathbf{b}^{(l)}$: weights & biases of Φ & $\widetilde{\mathbf{W}}^{(l)}, \widetilde{\mathbf{b}}^{(l)}$: biases of $\widehat{\Phi}$. Show: such a statement does not hold for ReLU neural networks. What about leaky ReLU?*

- 3. Universal approximation. After introducing neural networks in Chap. 2, natural to inquire about their capabilities. Specifically, might wonder if there exist inherent limitations to type of functions a neural network can represent. Could there be a class of functions that neural networks cannot approximate? If so, it would suggest: neural networks are specialized tools, similar to how linear regression is suited for linear relationships, but not for data with nonlinear relationships.

  – Sau khi giới thiệu mạng nơ-ron trong Chương 2, tự nhiên là phải tìm hiểu về khả năng của chúng. Cụ thể, có thể tự hỏi liệu có tồn tại những hạn chế cố hữu đối với loại hàm mà mạng nơ-ron có thể biểu diễn không. Có thể có 1 lớp hàm mà mạng nơ-ron không thể xấp xỉ được không? Nếu có, điều đó sẽ gợi ý: mạng nơ-ron là các công cụ chuyên biệt, tương tự như cách hồi quy tuyến tính phù hợp với các mối quan hệ tuyến tính, nhưng không phù hợp với dữ liệu có các mối quan hệ phi tuyến tính.

  In this chap, show: this is not the case, & neural networks are indeed a *universal* tool. More precisely, given sufficiently large & complex architectures, they can approximate almost every sensible input-output relationship. Formalize & prove this claim in subsequent sects.

◦ 3.1. A universal approximation theorem. To analyze what kind of functions can be approximated with neural networks, start by considering uniform approximation of continuous functions $f : \mathbb{R}^d \to \mathbb{R}$ on compact sets. To this end, 1st introduce notion of compact convergence.

Definition 13. *Let $d \in \mathbb{N}$. A sequence of functions $f_n : \mathbb{R}^d \to \mathbb{R}, n \in \mathbb{N}$, is said to* converge compactly *to a function $f : \mathbb{R}^d \to \mathbb{R}$, if for every compact $K \subseteq \mathbb{R}^d$, $\lim_{n\to\infty} \sup_{\mathbf{x}\in K} |f_n(\mathbf{x}) - f(\mathbf{x})| = 0$. In this case, write $f_n \xrightarrow{cc} f$.*

Throughout what follows, always consider $C^0(\mathbb{R}^d)$ equipped with topology of Def. 3.1, & every subset e.g. $C^0(D)$ with subspace topology: e.g., if $D \subseteq \mathbb{R}^d$ is bounded, then convergence in $C^0(D)$ refers to uniform convergence $\lim_{n\to\infty} \sup_{\mathbf{x}\in D} |f_n(\mathbf{x}) - f(\mathbf{x})| = 0$.

∗ 3.1.1. Universal approximators. Want to show: deep neural networks can approximate every continuous function in sense of Def. 3.1. Call sets of functions that satisfy this property *universal approximators*.

Definition 14. *Let $d \in \mathbb{N}$. A set of functions $\mathcal{H}$ from $\mathbb{R}^d$ to $\mathbb{R}$ is a* universal approximator *(of $C^0(\mathbb{R}^d)$), if $\forall \varepsilon > 0$, every compact $K \subseteq \mathbb{R}^d$, & every $f \in C^0(\mathbb{R}^d)$, there exists $g \in \mathcal{H}$ s.t. $\sup_{\mathbf{x}\in K} |f(\mathbf{x}) - g(\mathbf{x})| < \varepsilon$.*

For a set of (not necessarily continuous) functions $\mathcal{H}$ mapping between $\mathbb{R}^d$ & $\mathbb{R}$, denote by $\overline{\mathcal{H}}^{\text{cc}}$ its closure w.r.t. compact convergence.

Relationship between a universal approximator & closure w.r.t. compact convergence is established in:

Proposition 9. *Let $d \in \mathbb{N}$ & $\mathcal{H}$ be a set of functions from $\mathbb{R}^d$ to $\mathbb{R}$. Then, $\mathcal{H}$ is a universal approximator of $C^0(\mathbb{R}^d)$ iff $C^0(\mathbb{R}^d) \subseteq \overline{\mathcal{H}}^{\text{cc}}$.*

A key tool to show that a set is a universal approximator is Stone–Weierstrass theorem, see, e.g., [Rud91, Sect. 5.7].

Theorem 2 (Stone–Weierstrass)**.** *Let $d \in \mathbb{N}$, let $K \subseteq \mathbb{R}^d$ be compact, & let $\mathcal{H} \subseteq C^0(K, \mathbb{R})$ satisfy that*

(a) $\forall \mathbf{x} \in K$, there exists $f \in \mathcal{H}$ s.t. $f(\mathbf{x}) \neq 0$,

(b) $\forall \mathbf{x} \neq \mathbf{y} \in K$ there exists $f \in \mathcal{H}$ s.t. $f(\mathbf{x}) \neq f(\mathbf{y})$,

(c) $\mathcal{H}$ is an algebra of functions, i.e., $\mathcal{H}$ is closed under addition, multiplication, & scalar multiplication.

Then $\mathcal{H}$ is dense in $C^0(K)$.

Example 11 (Polynomials are dense in $C^0(\mathbb{R}^d)$)**.** *For a multiindex $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_d) \in \mathbb{N}_0^d$ & a vector $\mathbf{x} = (x_1, \dots, x_d) \in \mathbb{R}^d$ denote $\mathbf{x}^{\boldsymbol{\alpha}} \coloneqq \prod_{j=1}^d x_j^{\alpha_j}$. In the following, with $|\boldsymbol{\alpha}| \coloneqq \sum_{i=1}^d \alpha_i$, write $\mathcal{P}_n \coloneqq \operatorname{span}\{\mathbf{x}^{\boldsymbol{\alpha}} | \boldsymbol{\alpha} \in \mathbb{N}_0^d,\ |\boldsymbol{\alpha}| \leq n\}$, i.e., $\mathcal{P}_n$ is space of polynomials of degree $\leq n$ (with real coefficients). Easy to check: $\mathcal{P} \coloneqq \bigcup_{n\in\mathbb{N}} \mathcal{P}_n(\mathbb{R}^d)$ satisfies assumptions of Stone–Weierstrass on every compact set $K \subseteq \mathbb{R}^d$. Thus space of polynomials* P *is a universal approximator of $C^0(\mathbb{R}^d)$, & by Prop. 3.3, $\mathcal{P}$ is dense in $C^0(\mathbb{R}^d)$. In case we wish to emphasize dimension of underlying space, in following, will also write $\mathcal{P}_n(\mathbb{R}^d)$ or $\mathcal{P}(\mathbb{R}^d)$ to denote $\mathcal{P}_n, \mathcal{P}$ resp.*

∗ 3.1.2. Shallow neural networks. With necessary formalism established, can now show: shallow neural networks of arbitrary width form a universal approximator under certain (mild) conditions on activation function. Results in this sect are based on [132], & for proofs follow arguments in that paper.

1st introduce notation for set of all functions realized by certain architectures.

Definition 15. *Let $d, m, L, n \in \mathbb{N}$ & $\sigma : \mathbb{R} \to \mathbb{R}$. Set of all functions realized by neural networks with d-dimensional input, m-dimensional output, depth at most L, width at most n, & activation function σ is denoted by*

$$\mathcal{N}_d^m(\sigma; L, n) \coloneqq \{\Phi : \mathbb{R}^d \to \mathbb{R}^m | \Phi \text{ as in Def. 2.1},\ \operatorname{depth}(\Phi) \leq L,\ \operatorname{width}(\Phi) \leq n\}. \tag{166}$$

Furthermore,

$$\mathcal{N}_d^m(\sigma; L) \coloneqq \bigcup_{n\in\mathbb{N}} \mathcal{N}_d^m(\sigma; L, n). \tag{167}$$

In sequel, require activation function σ to belong to set of piecewise continuous & locally bounded functions

$$\mathcal{M} \coloneqq \{\sigma \in L^\infty_{\text{loc}}(\mathbb{R}) | \text{there exists intervals } I_1, \dots, I_M \text{ partitioning } \mathbb{R},\ \text{s.t. } \sigma \in C^0(I_i),\ \forall i = 1, \dots, M\}. \tag{168}$$

Here, $M \in \mathbb{N}$ is finite, & intervals I_i are understood to have positive (possibly infinite) Lebesgue measure, i.e., I_i is e.g. not allowed to be empty or a single point. Hence, σ is a piecewise continuous function, & it has discontinuities at most finitely many points.

Example 12. *Activation functions belonging to $\mathcal{M}$ include, in particular, all continuous non-polynomial functions, which in turn includes all practically relevant activation functions e.g. ReLU, SiLU, & Sigmoid discussed in Sect. 2.3. In these cases, can choose $M = 1$ & $I_1 = \mathbb{R}$. Discontinuous functions include e.g. Heaviside function $x \mapsto \mathbf{1}_{\mathbf{x>0}}$ (also called a "perceptron" in this context) but also $x \mapsto \mathbf{1}_{x>0} \sin \frac{1}{x}$: Both belong to $\mathcal{M}$ with $M = 2$, $I_1 = (-\infty, 0], I_2 = (0, \infty)$. Exclude e.g. function $x \mapsto \frac{1}{x}$, which is not locally bounded.*

Rest of this subsect is dedicated to proving following theorem that has now already been announced repeatedly.

Theorem 3. *Let $d \in \mathbb{N}$ & $\sigma \in \mathcal{M}$. Then $\mathcal{N}_d^1(\sigma, 1)$ is a universal approximator of $C^0(\mathbb{R}^d)$ iff σ is not a polynomial.*

Remark 12. *Exercise 3.26 & Corollary 3.18: neural networks can also arbitrarily well approximate non-continuous functions w.r.t. suitable norms.*

Universal approximation theorem by Leshno, Lin, Pinkus & Schocken [132] – of which Thm. 3.8 is a special case – is even formulated for a much larger set $\mathcal{M}$, which allows for activation functions that have discontinuities at a (possibly non-finite) set of Lebesgue measure 0. Instead of proving theorem in this generality, resort to simpler case stated above. This allows to avoid some technicalities, but main ideas remain same. Proof strategy: verify 3 claims:

(i) if $C^0(\mathbb{R}) \subseteq \overline{\mathcal{N}_1^1(\sigma;1)}^{\mathrm{cc}}$ then $C^0(\mathbb{R}^d) \subseteq \overline{\mathcal{N}_d^1(\sigma;1)}^{\mathrm{cc}}$,

(ii) if $\sigma \in C^\infty(\mathbb{R})$ is not a polynomial then $C^0(\mathbb{R}) \subseteq \overline{\mathcal{N}_1^1(\sigma;1)}^{\mathrm{cc}}$,

(iii) if $\sigma \in \mathcal{M}$ is not a polynomial then there exists $\widetilde{\sigma} \in C^\infty(\mathbb{R}) \cap \overline{\mathcal{N}_1^1(\sigma;1)}^{\mathrm{cc}}$ which is not a polynomial.

- 3.2. Superexpressive activations & Kolmogorov's supersolution theorem.

- 4. Splines.
  - 4.1. B-splines & smooth functions.
  - 4.2. Reaproximation of B-splines with sigmoidal activations.
- 5. ReLU neural networks.
  - 5.1. Basic ReLU calculus.
  - 5.2. Continuous piecewise linear functions.
  - 5.3. Simplicial pieces.
  - 5.4. Convergence rates for Hölder continuous functions.
- 6. Affine pieces for ReLU neural networks.
  - 6.1. Upper bounds.
  - 6.2. Tightness of upper bounds.
  - 6.3. Depth separation.
  - 6.4. Number of pieces in practice.
- 7. Deep ReLU neural networks.
  - 7.1. Square function.
  - 7.2. Multiplication.
  - 7.3. $C^{k,s}$ functions.
- 8. High-dimensional approximation.
  - 8.1. Barron class.
  - 8.2. Functions with compositionality structure.
  - 8.3. Functions on manifolds.
- 9. Interpolation.
  - 9.1. Universal interpolation.
  - 9.2. Optimal interpolation & reconstruction.
- 10. Training of neural networks.
  - 10.1. Gradient descent.
  - 10.2. Stochastic gradient descent (SGD).
  - 10.3. Backpropagation.
  - 10.4. Acceleration.
  - 10.5. Other methods.
- 11. Wide neural networks & neural tangent kernel.
  - 11.1. Linear least-squares.
  - 11.2. Kernel least-squares.
  - 11.3. Tangent kernel.
  - 11.4. Convergence to global minimizers.
  - 11.5. Training dynamics for LeCun initialization.
  - 11.6. Normalized initialization.
- 12. Loss landscape analysis.
  - 12.1. Visualization of loss landscapes.
  - 12.2. Spurious valleys.
  - 12.3. Saddle points.
- 13. Shape of neural network spaces.
  - 13.1. Lipschitz parameterizations.
  - 13.2. Convexity of neural network spaces.
  - 13.3. Closedness & best-approximation property.
- 14. Generalization properties of deep neural networks.
  - 14.1. Learning setup.
  - 14.2. Empirical risk minimization.

- 14.3. Generalization bounds.
- 14.4. Generalization bounds from covering numbers.
- 14.5. Covering numbers of deep neural networks.
- 14.6. Approximate-complexity trade-off.
- 14.7. PAC learning from VC dimension.
- 14.8. Lower bounds on achievable approximation rates.

- 15. Generalization in overparameterized regime.
  - 15.1. Double descent phenomenon.
  - 15.2. Size of weights.
  - 15.3. Theoretical justification.
  - 15.4. Double descent for neural network learning.
- 16. Robustness & adversarial examples.
  - 16.1. Adversarial examples.
  - 16.2. Bayes classifier.
  - 16.3. Affine classifiers.
  - 16.4. ReLU neural networks.
  - 16.5. Robustness.
- A. Probability theory.
  - A.1. Sigma-algebras, topologies, & measures.
  - A.2. Random variables.
  - A.3. Conditionals, marginals, & independence.
  - A.4. Concentration inequalities.
- B. Functional analysis.
  - B.1. Vector spaces.
  - B.2. Fourier transform.

5. [Zha+23]. Aston Zhang, Zachary C. Lipton, Mu Li, Alexander J. Smola. *Dive into Deep Learning*.

- Preface. Just a few years ago, there were no legions of deep learning scientists developing intelligent products & services at major companies & startups. When entered field, ML did not command headlines in daily newspapers. Parents had no idea what ML was, let alone why might prefer it to a career in medicine or law. ML was a blue skies academic discipline whose industrial significance was limited to a narrow set of real-world applications, including speech recognition & computer vision. Moreover, many of these applications required so much domain knowledge that they were often regarded as entirely separate areas for which ML was 1 small component. At that time, neural networks – predecessors of deep learning methods – were generally regarded as outmoded.

  Yet in just few years, deep learning has taken the world by surprise, driving rapid progress in such diverse fields as computer vision, natural language processing, automatic speech recognition, reinforcement learning, & biomedical informatics. Moreover, success of deep learning in so many tasks of practical interest has even catalyzed developments in theoretical machine learning & statistics. With these advances in hand, can now build cars that drive themselves with more autonomy than ever before (though less autonomy than some companies might have you believe), dialogue systems that debug code by asking clarifying questions, & software agents beating best human players in world at board games e.g. Go, a feat once thought to be decades away. Already, these tools exert ever-wider influence on industry & society, changing way movies are made, diseases are diagnosed, & playing a growing role in basic sciences – from astrophysics, to climate modeling, to weather prediction, to biomedicine.

  About this Book. This book represents attempt to make DL approachable, teaching you *concepts, context, & code*.

  - One Medium Combining Code, Math, & HTML. For any computing technology to reach its full impact, it must be well understood, well documented, & supported by mature, well-maintained tools. Key ideas should be clearly distilled, minimizing onboarding time needed to bring new practitioners up to date. Mature libraries should automate common tasks, & exemplar code should make it easy for practitioners to modify, apply, & extend common applications to suit their needs.

    E.g., take dynamic web applications. Despite a large number of companies, e.g. Amazon, developing successful database-driven web applications in 1990s, potential of this technology to aid creative entrepreneurs was realized to a far greater degree only in past 10 years, owing in part to development of powerful, well-documented frameworks.

    Testing potential of deep learning presents unique challenges because any single application brings together various disciplines. Applying deep learning requires simultaneously understanding:

    (i) motivations for casting a problem in a particular way;

    (ii) mathematical form of a given model;

    * (iii) optimization algorithms for fitting models to data;

(iv) statistical principles that tell us when we should expect our models to generalize to unseen data & practical methods for certifying that they have, in fact, generalized;

(v) engineering techniques required to train models efficiently, navigating pitfalls of numerical computing & getting most out of available hardware.

Teaching critical thinking skills required to formulate problems, mathematics to solve them, & software tools to implement those solutions all in 1 place presents formidable challenges. Goal of this book: to present a unified resource to bring would-be practitioners up to speed.

When started this book project, there were no resources that simultaneously

(i) remained up to date;

(ii) covered breadth of modern machine learning practices with sufficient technical depth;

(iii) interleaved exposition of quality one expects of a textbook with clean runnable code that one expects of a hands-on tutorial.

Found plenty of code examples illustrating how to use a given deep learning framework (e.g., how to do basic numerical computing with matrices in TensorFlow) or for implementing particular techniques (e.g., code snippets for LeNet, AlexNet, ResNet, etc.) scattered across various blog posts & GitHub repositories. However, these examples typically focused on *how to* implement a given approach, but left out discussion of *why* certain algorithmic decisions are made. While some interactive resources have popped up sporadically to address a particular topic, e.g., engaging blog posts published on website Distill https://distill.pub/, or personal blogs, they only covered selected topics in deep learning, & often lacked associated code. On other hand, while several deep learning textbooks have emerged – e.g., Goodfellow et al. (2016), which offers a comprehensive survey on basics of deep learning – these resources do not marry descriptions to realizations of concepts in code, sometimes leaving readers clueless as to how to implement them. Moreover, too many resources are hidden behind the paywalls of commercial course providers.

Set out to create a resource that could

(i) be freely available for everyone;

(ii) offer sufficient technical depth to provide a starting point on path to actually becoming an applied machine learning scientist;

(iii) include runnable code, showing readers *how* to solve problems in practice;

(iv) allow for rapid updates, both by us & also by community at large;

(v) be complemented by a foru https://discuss.d2l.ai/c/english-version/5 for interactive discussion of technical details & to answer questions.

These goals were often in conflict. Equations, theorems, & citations are best managed & laid out in LaTeX. Code is best described in Python. & webpages are native in HTML & JavaScript. Furthermore, want content to be accessible both as executable code, as a physical book, as a downloadable PDF, & on Internet as a website. No workflows seemed suited to these demands, so decided to assemble our own (Sect. B.6). Settled on GitHub to share source & to facilitate community contributions; Jupyter notebooks for mixing code, equations & text; Sphinx as a rendering engine; & Discourse as a discussion platform. While our system is not perfect, these choices strike a compromise among competing concerns. Believe: *Dive into Deep Learning* might be 1st book published using such an integrated workflow.

- Learning by Doing. Many textbooks present concepts in succession, covering each in exhaustive detail. E.g., excellent textbook of Bishop (2006), teaches each topic so thoroughly that getting to chapter on linear regression requires a nontrivial amount of work. While experts love this book precisely for its thoroughness, for true beginners, this property limits its usefulness as an introductory text.

  In this book, teach most concepts *just in time*. I.e., will learn concepts at very moment that they are needed to accomplish some practical end. While take some time at outset to teach fundamental preliminaries, like linear algebra & probability, want you to taste satisfaction of training your 1st model before worrying about more esoteric concepts.

  Aside from a few preliminary notebooks that provide a crash course in basic mathematical background, each subsequent chapter both introduces a reasonable number of new concepts & provides several self-contained working examples, using real datasets. This presented an organizational challenge. Some models might logically be grouped together single notebook. & some ideas might be best taught by executing several models in succession. By contrast, there is a big advantage to adhering to a policy of *1 working example, 1 notebook*: This makes it as easy as possible for you to start your own research projects by leveraging our code. Just copy a notebook & start modifying it.

  Throughout, interleave runnable code with background material as needed. In general, err on side of making tools available before explaining them fully (often filling in background later). E.g., might use *stochastic gradient descent* before explaining why it is useful or offering some intuition for why it works. This helps to give practitioners necessary ammunition to solve problems quickly, at expense of requiring reader to trust us with some curatorial decisions.

  This book teaches deep learning concepts from scratch. Sometimes, delve into fine details about models that would typically be hidden from users by modern deep learning frameworks. This comes up especially in basic tutorials, where want you to understand everything that happens in a given layer or optimizer. In these cases, often present 2 versions of example: 1 where implement everything from scratch, relying only NumPy-like functionality & automatic differentiation, & a more practical example, where write succinct code using high-level APIs of deep learning frameworks. After explaining how some component works, rely on high-level API in subsequent tutorials.

- Content & Structure. Book can be divided into roughly 3 parts, dealing with preliminaries, deep learning techniques, & advanced topics focused on real systems & applications. Book structure:

* **Part 1: Basics & Preliminaries.**
  - Chap. 1 is an introduction to deep learning.
  - Chap. 2: quickly bring up to speed on prerequisites required for hands-on deep learning, e.g. how to store & manipulate data, & how to apply various numerical operations based on elementary concepts from linear algebra, calculus, & probability. Chaps. 3 & 5 cover most fundamental concepts & techniques in deep learning, including regression & classification; linear models; multilayer perceptrons; & overfitting & regularization.
* **Part 2: Modern Deep Learning Techniques.**
  - Chap. 6 describes key computational components of deep learning systems & lays groundwork for subsequent implementations of more complex models.
  - Chaps. 7 & 8 present convolutional neural networks (CNNs), powerful tools that form backbone of most modern computer vision systems.
  - Similarly, Chaps. 9–10 introduce recurrent neural networks (RNNs), models that exploit sequential (e.g., temporal) structure in data & are commonly used for natural language processing & time series prediction.
  - Chap. 11: describe a relatively new class of models, based on so-called *attention mechanisms*, that has displaced RNNs as dominant architecture for most natural language processing tasks. These sects will bring up to speed on most powerful & general tools that are widely used by deep learning practitioners.
* **Part 3: Scalability, Efficiency, & Applications** available online https://d2l.ai/.
  - Chap. 12: discuss several common optimization algorithms used to train deep learning models.
  - Chap. 13: examine several key factors that influence computational performance of deep learning code.
  - Chap. 14: illustrate major applications of deep learning in computer vision.
  - Chaps. 15–16: demonstrate how to pretrain language representation models & apply them to natural language processing tasks.
* Code. Most sects of this book feature executable code. Believe: some intuitions are best developed via trial & error, tweaking code in small ways & observing results. Ideally, an elegant mathematical theory might tell us precisely how to tweak our code to achieve a desired result. However, deep learning practitioners today must often tread where no solid theory provides guidance. Despite best attempts, formal explanations for efficacy of various techniques are still lacking, for a variety of reasons: mathematics to characterize these models can be so difficult; explanation likely depends on properties of data that currently lack clear defs; & serious inquiry on these topics has only recently kicked into high gear. Hopeful: As theory of deep learning progresses, each future edition of this book will provide insights that eclipse those presently available.
  To avoid unnecessary repetition, capture some of most frequently imported & used functions & classes in `d2l` package. Throughout, mark blocks of code (e.g. functions, classes, or collection of import statements) with `#@save` to indicate: they will be accessed later via `d2l` package. Offer a detailed overview of these classes & functions in Sect. B.8. `d2l` package is lightweight & only requires following dependencies:

```
#@save
import collections
import hashlib
import inspect
import math
import os
import random
import re
import shutil
import sys
import tarfile
import time
import zipfile
from collections import defaultdict
import pandas as pd
import requests
from IPython import display
from matplotlib import pyplot as plt
from matplotlib_inline import backend_inline
d2l = sys.modules[__name__]
```

  Most of code in this book is based on PyTorch, a popular open-source framework that has been enthusiastically embraced by deep learning research community. All of code in this book has passed tests under latest stable version of PyTorch. However, due to rapid development of deep learning, some code *in print edition* may not work properly in future versions of PyTorch. Plan to keep online version up to date. In case encounter any problems, consult *Installation* to update your code & runtime environment. Below lists dependencies in our PyTorch implementation.

```
#@save
```

```
import numpy as np
import torch
import torchvision
from PIL import Image
from scipy.spatial import distance_matrix
from torch import nn
from torch.nn import functional as F
from torchvision import transforms
```

* Target Audience. This book is for students (undergraduate or graduate), engineers, & researchers, who seek a solid grasp of practical techniques of deep learning. Because explain every concept from scratch, no previous background in deep learning or ML is required. Fully explaining methods of deep learning requires some mathematics & programming, but will only assume that you enter with some basics, including modest amounts of linear algebra, calculus, probability, & Python programming. Just in case you have forgotten anything, online Appendix https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/index.html provides a refresher on most of mathematics you will find in this book. Usually, will prioritize intuition & ideas over mathematical rigor. If would like to extend these foundations beyond prerequisites to understand book, happily recommend some other terrific resources: *Linear Analysis* by Bollobás (1999) covers linear algebra & functional analysis in great depth. *All of Statistics* (Wasserman, 2013) provides a marvelous introduction to statistics. Joe Blitzstein's books *Introduction to Probability* & courses https://projects.iq.harvard.edu/stat110/home on probability & inference are pedagogical gems. & if you have not used Python before, may want to peruse this Python tutorial http://learnpython.org/.
* Notebooks, Website, GitHub, & Forum. All of notebooks are available for download on https://d2l.ai & on GitHub https://github.com/d2l-ai/d2l-en. Associated with this book, have launched a discussion forum, located at https://discuss.d2l.ai/c/5. Whenever you have questions on any sect of book, can find a link to associated discussion page at end of each notebook.
* Acknowledgments. This book was originally implemented with MXNet as primary framework. Adapt a majority part of earlier MXNet code into PyTorch & TensorFlow implementations, resp. Since Jul 2021, have redesigned & reimplemented this book in PyTorch, MXNet, & TensorFlow, choosing PyTorch as primary framework. Adapt a majority part of more recent PyTorch code into JAX implementations. From Baidu for adapting a majority part of more recent PyTorch code into PaddlePaddle implementations in Chinese draft.
* Summary. Deep Learning has revolutionized pattern recognition, introducing technology that now powers a wide range of technologies, in such diverse fields as computer vision, natural language processing, & automatic speech recognition. To successfully apply deep learning, must understand how to cast a problem, basic mathematics of modeling, algorithms for fitting models to data, & engineering techniques to implement it all. This book presents a comprehensive resource, including prose, figures, mathematics, & code, all in 1 place.

- Installation. In order to get up & running, need an environment for running Python, Jupyter Notebook, relevant libraries, & code needed to run book itself.
  - Installing Miniconda. Simplest option: to install Miniconda. Note: Require Python 3.x version. Visit Miniconda website & determine appropriate version for your system based on your Python 3.x version & machine architecture. A Linux user would download file whose name contains strings "Linux" & execute following at download location:

```
# The file name is subject to changes
sh Miniconda3-py39_4.12.0-Linux-x86_64.sh -b
```

Next, initialize shell so can run conda directly.

```
~/miniconda3/bin/conda init
```

Then close & reopen current shell. Should be able to create a new environment as follows:

```
(base) nqbh@nqbh-dell:~$ python --version
Python 3.12.7
(base) nqbh@nqbh-dell:~$ conda create --name d2l python=3.12.7 -y
Channels:
- defaults
Platform: linux-64
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

environment location: /home/nqbh/anaconda3/envs/d2l

added / updated specs:
- python=3.12.7
```

```
The following packages will be downloaded:

package                    |            build
---------------------------|-----------------
expat-2.6.4                |       h6a678d5_0         180 KB
------------------------------------------------------------
Total:         180 KB

The following NEW packages will be INSTALLED:

_libgcc_mutex      pkgs/main/linux-64::_libgcc_mutex-0.1-main
_openmp_mutex      pkgs/main/linux-64::_openmp_mutex-5.1-1_gnu
bzip2              pkgs/main/linux-64::bzip2-1.0.8-h5eee18b_6
ca-certificates    pkgs/main/linux-64::ca-certificates-2024.11.26-h06a4308_0
expat              pkgs/main/linux-64::expat-2.6.4-h6a678d5_0
ld_impl_linux-64   pkgs/main/linux-64::ld_impl_linux-64-2.40-h12ee557_0
libffi             pkgs/main/linux-64::libffi-3.4.4-h6a678d5_1
libgcc-ng          pkgs/main/linux-64::libgcc-ng-11.2.0-h1234567_1
libgomp            pkgs/main/linux-64::libgomp-11.2.0-h1234567_1
libstdcxx-ng       pkgs/main/linux-64::libstdcxx-ng-11.2.0-h1234567_1
libuuid            pkgs/main/linux-64::libuuid-1.41.5-h5eee18b_0
ncurses            pkgs/main/linux-64::ncurses-6.4-h6a678d5_0
openssl            pkgs/main/linux-64::openssl-3.0.15-h5eee18b_0
pip                pkgs/main/linux-64::pip-24.2-py312h06a4308_0
python             pkgs/main/linux-64::python-3.12.7-h5148396_0
readline           pkgs/main/linux-64::readline-8.2-h5eee18b_0
setuptools         pkgs/main/linux-64::setuptools-75.1.0-py312h06a4308_0
sqlite             pkgs/main/linux-64::sqlite-3.45.3-h5eee18b_0
tk                 pkgs/main/linux-64::tk-8.6.14-h39e8969_0
tzdata             pkgs/main/noarch::tzdata-2024b-h04d1e81_0
wheel              pkgs/main/linux-64::wheel-0.44.0-py312h06a4308_0
xz                 pkgs/main/linux-64::xz-5.4.6-h5eee18b_1
zlib               pkgs/main/linux-64::zlib-1.2.13-h5eee18b_1



Downloading \& Extracting Packages:

Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate d2l
#
# To deactivate an active environment, use
#
#     $ conda deactivate

(base) nqbh@nqbh-dell:~$ conda activate d2l
(d2l) nqbh@nqbh-dell:~$
```

Now can activate `d2l` environment:

```
conda activate d2l
```

- Installing Deep Learning Framework & `d2l` Package. Before installing any DL framework, 1st check whether or not you have proper GPUs on machine (GPUs power display on a standard laptop are not relevant for our purposes). E.g., if computer has NVIDIA GPUs & has installed CUDA https://developer.nvidia.com/cuda-downloads, then you are all set. If your machine does not house any GPU, there is no need to worry just yet. Your CPU provides more than enough horsepower to get through 1st few chaps. Just remember: will want to access GPUs before running larger models.

Can install PyTorch (specified versions are tested at time of writing) with either CPU or GPU support as follows:

```
pip install torch==2.0.0 torchvision==0.15.1
```

Next step: to install `d2l` package developed in order to encapsulate frequently used functions & classes found throughout this book:

```
pip install d2l==1.0.3
```

- **Downloading & Running Code.** Download notebooks so that can run each of book's code blocks. Simply click on "Notebooks" tab at top of any HTML page on https://d2l.ai/ to download code & then unzip it. Alternatively, can fetch notebooks from command line as follows:

```
mkdir d2l-en && cd d2l-en
curl https://d2l.ai/d2l-en-1.0.3.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
cd pytorch
```

`SKIP INSTALLATION STEPS`

- 1. Introduction. Until recently, nearly every computer program that you might have interacted with during an ordinary day was coded up as a rigid set of rules specifying precisely how it should behave. Say: wanted to write an application to manage an e-commerce platform. After huddling around a whiteboard for a few hours to ponder problem, might settle on broad strokes of a working solution, e.g.:

(i) users interact with application through an interface running in a web browser or mobile application;

(ii) application interacts with a commercial-grade database engine to keep track of each user's state & maintain records of historical transactions;

(iii) at heart of application, *business logic* (you might say, *brains*) of application spells out a set of rules that map every conceivable circumstances to corresponding action that our program should take.

To build brains of application, might enumerate all common events that program should handle. E.g., whenever a customer clicks to add an item to their shopping cart, program should add an entry to shopping cart database table, associating that user's ID with requested product's ID. Might then attempt to step through every possible corner case, testing appropriateness of our rules & making any necessary modifications. What happens if a user initiates a purchase with an empty cart? While few developers ever get it completely right 1st time (it might take some test runs to work out kinks), for most part, can write such programs & confidently launch them *before* ever seeing a real customer. Ability to manually design automated systems that drive functioning products & systems, often in novel situations, is a remarkable cognitive feat. & when you are able to devise solutions (đưa ra giải pháp) that work 100% of time, typically should not be worrying about ML.

Fortunately for growing community of ML scientists, many tasks that we would like to automate do not bend so easily to human ingenuity. Imagine huddling around whiteboard with smartest minds you know, but this time you are tackling 1 of following problems:

- Write a program that predicts tomorrow's weather given geographic information, satellite images, & a trailing window of past weather.
- Write a program that takes in a factoid question, expressed in free-form text, & answers it correctly.
- Write a program that, given an image, identifies every person depicted in it & draws outlines around each.
- Write a program that presents users with products that they are likely to enjoy but unlikely, in natural course of browsing, to encounter.

For these problems, even elite programmers would struggle to code up solutions from scratch. The reasons can vary. Sometimes program that we are looking for follows a pattern that changes over time, so there is no fixed right answer! In such cases, any successful solution must adapt gracefully to a changing world. At other times, relationship (say between pixels, & abstract categories) may be too complicated, requiring thousands or millions of computations & following unknown principles. In case of image recognition, precise steps required to perform task lie beyond our conscious understanding, even though our subconscious cognitive processes execute task effortlessly.

ML is study of algorithms that can learn from experience. As a ML algorithm accumulates more experience, typically in form of observational data or interactions with an environment, its performance improves. Contrast this with our deterministic e-commerce platform, which follows same business logic, no matter how much experience accrues, until developers themselves learn & decide that it is time to update software. In this book, teach fundamentals of ML, focusing in particular on *deep learning*, a powerful set of techniques driving innovations in areas as diverse as computer vision, natural language processing, healthcare, & genomics.

- 1.1. A Motivating Example. Before beginning writing, authors of this book, like much of work force, had to become caffeinated. Hopped in car & started driving. Using an iPhone, Alex called out "Hey Siri", awakening phone's voice recognition system. Then Mu commanded "directions to Blue Bottle coffee shop". Phone quickly displayed transcription of his command. Also recognized: were asking for directions & launched Maps application (app) to fulfill our request. Once launched, Maps app identified a number of routes. Next to each route, phone displayed a predicted transit time.

While this story was fabricated for pedagogical convenience, it demonstrates: in span of just a few secs, our everyday interactions with a smart phone can engage several ML models.

Imagine just writing a program to respond to a *wake word* e.g. "Alexa", "OK Google", & "Hey Siri". Try coding it up in a room by yourself with nothing but a computer & a code editor. *How would write such a program from 1st principles?* Think about it ... problem is hard. Every sec, microphone will collect roughly 44000 samples. Each sample is a measurement of amplitude of sound wave. What rule could map reliably from a snippet of raw audio to confident predictions {yes, no} about whether snippet contains wake word? If stuck, do not worry. Do not know how to write such a program from scratch either. That is why use ML.

Here is trick. Often, even when we do not know how to tell a computer explicitly how to map from inputs to outputs, we are nonetheless capable of performing cognitive feat ourselves. I.e., even if do not know how to program a computer to recognize word "Alexa", you yourself are able to recognize it. Armed with this ability, can collect a huge *dataset* containing examples of audio snippets & associated labels, indicating which snippets contain wake word. In currently dominant approach to ML, do not attempt to design a system *explicitly* to recognize wake words. Instead, define a flexible program whose behavior is determined by a number of *parameters*. Then use dataset to determine best possible parameter values, i.e., those that improve performance of our program w.r.t. a chosen performance measure.

Can think of parameters as knobs (núm vặn) that we can turn, manipulating behavior of program. Once parameters are fixed, called program a *model*. Set of all distinct programs (input–output mappings) that we can produce just by manipulating parameters is called a *family* of models. & "meta-program" that uses our dataset to choose parameters is called a *learning algorithm*.

Before can go ahead & engage learning algorithm, have to define problem precisely, pinning down exact nature of inputs & outputs, & choosing an appropriate model family. In this case, our model receives a snippet of audio as *input*, & model generates a selection among {yes, no} as *output*. If all goes according to plan model's guesses will typically be correct as to whether snippet contains wake word.

If choose right family of models, there should exist 1 setting of knobs s.t. model fires "yes" every time it hears word "Alexa". Because exact choice of wake word is arbitrary, will probably need a model family sufficiently rich that, via another setting of knobs, it could fire "yes" only upon hearing word "Apricot" (quả mơ). Expect: same model family should be suitable for "Alexa" recognition & "Apricot" recognition because they seem, intuitively, to be similar tasks. However, might need a different family of models entirely if want to deal with fundamentally different inputs or outputs, say if wanted to map from images to captions, or from English sentences to Chinese sentences.

As you might guess, if just set all of knobs randomly, unlikely: our model will recognize "Alexa", "Apricot", or any other English word. In ML, *learning* is process by which discover right setting of knobs for coercing desired behavior from our model. I.e., we *train* our model with data. As shown in Fig. 1.1.2: A typical training process, training process usually looks like following:

(a) Start off with a randomly initialized model that cannot do anything useful.
(b) Grab some of your data (e.g., audio snippets & corresponding {yes, no} labels).
(c) Tweak knobs to make model perform better as assessed on those examples.
(d) Repeat Steps 2 & 3 until model is awesome.

To summarize, rather than code up a weak word recognizer, code up a program that can *learn* to recognize wake words, if presented with a large labeled dataset. You can think of this act of determining a program's behavior by presenting it with a dataset as *programming with data*. I.e., can "program" a cat detector by providing our ML system with many examples of cats & dogs. This way detector will eventually learn to emit a large positive number if it is a cat, a very large negative number if it is a dog, & something closer to 0 if not sure. This barely scratches surface of what ML can do. DL is just 1 among many popular methods for solving ML problems.

- 1.2. Key Components. In wake word example, described a dataset consisting of audio snippets & binary labels, & gave a hand-wavy sense of how might train a model to approximate a mapping from snippets to classifications. This sort of problem, where try to predict a designated unknown label based on known inputs given a dataset consisting of examples for which labels are known, is called *supervised learning*. This is just 1 among many kinds of ML problems. Before explore other varieties, would like to shed more light on some core components that will follow us around, no matter what kind of ML problem we tackle:
  (a) *Data* that we can learn from.
  (b) A *model* of how to transform data.
  (c) An *objective function* that quantifies how well (or badly) model is doing.
  (d) An *algorithm* to adjust model's parameters to optimize objective function.
  - 1.2.1. Data. Cannot do data science without data. Could lose hundreds of pages pondering what precisely data *is*, but for now, focus on key properties of datasets that we will be concerned with. Generally, concerned with a collection of examples. In order to work with data usefully, typically need to come up with a suitable numerical representation. Each *example* (or *data point, data instance, sample*) typically consists of a set of attributes called *features* (sometimes called *covariates* or *inputs*), based on which model must make its predictions. In supervised learning problems, goal: to predict value of a special attribute, called *label* (or *target*), that is not part of model's input.

    If were working with image data, each example might consist of an individual photograph (features) & a number indicating category to which photograph belongs (label). Photograph would be represented numerically as 3 grids of numerical values representing brightness of red, green, & blue light at each pixel location. E.g., a 200×200 pixel color photograph would consist of $200 \times 200 \times 3 = 120000$ numerical values.

Alternatively, might work with electronic health record data & tackle task of predicting likelihood (khả năng xảy ra) that a given patient will survive next 30 days. Here, features might consist of a collection of readily available attributes & frequently recorded measurements, including age, vital signs, comorbidities, current medications, & recent procedures. Label available for training would be a binary value indicating whether each patient in historical data survived within 30-day window.

In such cases, when every example is characterized by same number of numerical features, say: inputs are fixed-length vectors & call (constant) length of vectors *dimensionality* of data. As you might imagine, fixed-length inputs can be convenient, giving us 1 less complication to worry about. However, not all data can easily be represented as *fixed-length* vectors. While might expect microscope images to come from standard equipment, cannot expect expect images mined from Internet all to have same resolution or shape. For images, might consider cropping them to a standard size, but that strategy only gets us so far. Risk losing information in cropped-out portions. Moreover, text data resists fixed-length representations even more stubbornly. Consider customer reviews left on e-commerce sites e.g. Amazon, IMDb, & TripAdvisor. Some are short: "it stinks!". Others ramble for pages. 1 major advantage of DL over traditional methods is comparative grace with which modern models can handle *varying-length* data.

Generally, more data we have, easier our job becomes.[6] When have more data, can train more powerful models & rely less heavily on preconceived assumptions. Regime (chế độ) change from (comparatively) small to big data is a major contributor to success of modern DL. To drive point home, many of most exciting models in DL do not work without large datasets. Some others might work in small data regime, but are no better than traditional approaches.

Finally, not enough to have lots of data & to process it cleverly. Need *right* data. If data is full of mistakes, or if chosen features are not predictive of target quantity of interest, learning is going to fail. Situation is captured well by cliché: *garbage in, garbage out*. Moreover, poor predictive performance is not only potential consequence. In sensitive applications of ML, like predictive policing, resume screening, & risk models used for lending, must be especially alert to consequences of garbage data. 1 commonly occurring failure mode concerns datasets where some groups of people are unrepresented in training data. Imagine applying a skin cancer recognition system that had never been black skin before. Failure can also occur when data does not only under-represent some groups but reflects societal prejudices. E.g., if past hiring decisions are used to train a predictive model that will be used to screen resumes then ML models could inadvertently capture & automate historical injustices. Note: this can all happen without data scientist actively conspiring, or even being aware.

* 1.2.2. Models. Most ML involves transforming data in some sense. Might want to build a system that ingests photos & predicts smiley-ness. Alternatively, might want to ingest a set of sensor readings & predict how normal vs. anomalous (bất thường) readings are. By *model*, denote computational machinery for ingesting data of 1 type, & spitting out predictions of a possibly different type. In particular, interested in *statistical models* that can be estimated from data. While simple models are perfectly capable of addressing appropriately simple problems, problems that we focus on in this book stretch limits of classical methods. DL is differentiated from classical approaches principally by set of powerful models that it focuses on. These models consist of many successive transformations of data chained together top to bottom, thus name *deep learning*. On our way to discussing deep models, also discuss some more traditional methods.

* 1.2.3. Objective Functions. Earlier, introduced ML as learning from experience. By *learning* here, mean improving at some task over time. But who is to say what constitutes an improvement? You might imagine: could propose updating our model, & some people might disagree on whether our proposal constituted an improvement or not.

  In order to develop a formal mathematical system of learning machines, need to have formal measures of how good (or bad) models are. In ML, & optimization more generally, call these *objective functions*. By convention, usually define objective functions so that lower is better. This is merely a convention. Can take any function for which higher is better, & turn it into a new function that is qualitatively identical but for which lower is better by flipping sign. Because choose lower to be better, these functions are sometimes called *loss functions*.

  When trying to predict numerical values, most common loss function is *squared error*, i.e., square of difference between prediction & ground truth target. For classification, most common objective is to minimize error rate, i.e., fraction of examples on which our predictions disagree with ground truth. Some objectives (e.g., squared error) are easy to optimize, while others (e.g., error rate) are difficult to optimize directly, owning to non-differentiability or other complications. In these cases, common instead to optimize a *surrogate objective* (mục tiêu thay thế).

  During optimization, think of loss as a function of model's parameters, & treat training dataset as a constant. Learn best values of our model's parameters by minimizing loss incurred on a set consisting of some number of examples collected for training. However, doing well on training data does not guarantee that we will do well on unseen data. So will typically want to split available data into 2 partitions: *training dataset* (or *training set*), for learning model parameters; & *test dataset* (or *test set*), which is held out for evaluation. At end of day, typically report how our models perform on both partitions. Could think of training performance as analogous to scores that a student achieves on practice exams used to prepare for some real final exam. Even if results are encouraging, that does not guarantee success on final exam. Over course of studying, student might begin to memorize practice questions, appearing to master topic but faltering when faced with previously unseen questions on actual final exam. When a model performs well on training set but fails to generalize to unseen data, say: it is *overfitting* (quá phù hợp) to training data.

* 1.2.4. Optimization Algorithms. Once have got some data source & representation, a model, & a well-defined objective function, need an algorithm capable of searching for best possible parameters for minimizing loss function. Popular

[6] NQBH: Really? Or more illusion, delusion?

optimization algorithms for DL are based on an approach called *gradient descent*. In brief, at each step, this method checks to see, for each parameter, how that training set loss would change if you perturbed that parameter by just a small amount. It would then update parameter in direction that lowers loss.

- 1.3. Kinds of ML Problems. Wake word problem in motivating example is just 1 among many ML can tackle. To motivate reader further & provide us with some common language that will follow us throughout book, provide a broad overview of landscape of ML problems.
  - 1.3.1. Supervised Learning. Supervised learning describes tasks where we are given a dataset containing both features & labels & asked to produce a model that predicts labels when given input features. Each feature-label pair is called an *example*. Sometimes, when context is clear, may use term *examples* to refer to a collection of inputs, even when corresponding labels are unknown. Supervision comes into play because, for choosing parameters, we (supervisors) provide model with a dataset consisting of labeled examples. In probabilistic terms, typically are interested in estimating conditional probability of a label given input features. While being just 1 among several paradigms, supervised learning accounts for majority of successful applications of ML in industry. Partly because many important tasks can be described crisply as estimating probability of something unknown given a particular set of available data:
    - Predict cancer vs. not cancer, given a computer tomography image.
    - Predict correct translation in French, given a sentence in English.
    - Predict price of a stock next month based on this month's financial reporting data.

    While all supervised learning problems are captured by simple description "predicting labels given input features", supervised learning itself can take diverse forms & require tons of modeling decisions, depending on (among other considerations) type, size, & quantity of inputs & outputs. E.g., use different models for processing sequences of arbitrary lengths & fixed-length vector representations. Visit many of these problems in depth throughout this book. Informally, learning process looks something like following. 1st, grab a big collection of examples for which features are known & select from them a random subset, acquiring ground truth labels for each. Sometimes these labels might be available data that have already been collected (e.g., did a patient die within following year?) & other times we might need to employ human annotators to label data, (e.g., assigning images to categories). Together, these inputs & corresponding labels comprise training set. Feel training dataset into a supervised learning algorithm, a function that takes as input a dataset & outputs another function: learned model. Finally, can feed previously unseen inputs to learned model, using its outputs as predictions of corresponding label. Full process is drawn in Fig. 1.3.1: Supervised learning.
    - **Regression.** Perhaps simplest supervised learning task to wrap your head around is *regression*. Consider, e.g., a set of data harvested from a database of home sales. Might construct a table, in which each row corresponds to a different house, & each column corresponds to some relevant attribute, e.g. square footage of a house, number of bedrooms, number of bathrooms, & number of minutes (walking) to center of town. In this dataset, each example would be a specific house, & corresponding feature vector would be 1 row in table. If live in New York or San Francisco, & you are not CEO of Amazon, Google, Microsoft, or Facebook, (sq. footage, no. of bedrooms, no. of bathrooms, walking distance) feature vector for your home might look something like: $[600, 1, 1, 60]$. However, if live in Pittsburg, it might look more like $[3000, 4, 3, 10]$. Fixed-length feature vectors like this are essential for most classic ML algorithms.
      What makes a problem a regression is actually form of target. Say : in market for a new home. Might want to estimate fair market value of a house, given some features e.g. above. Data here might consist of historical home listings & labels might be observed sales prices. When labels take on arbitrary numerical values (even within some interval), call this a *regression* problem. Goal: to produce a model whose predictions closely approximate actual label values.
      Lots of practical problems are easily described as regression problems. Predicting rating a user will assign to a movie can be thought of as a regression problem & if you designed a great algorithm to accomplish this feat in 2009, might have won 1 million-dollar Netflix prize https://en.wikipedia.org/wiki/Netflix_Prize. Predicting length of stay for patients in hospital is also a regression problem. A good rule of thumb: any *how much?* or *how many?* problem is likely to be regression. E.g.: How many hours will this surgery take? How much rainfall will this town have in next 6 hours? Even if have never worked with ML before, have probably worked through a regression problem informally. Imagine, e.g., had your drains repaired & your contractor spent 3 hours removing gunk from sewage pipes. Then they sent you a bill of 350$. Imagine: your friend hired same contractor for 2 hours & received a bill of 250$. If someone then asked: how much to expect on their upcoming gunk-removal invoice you might make some reasonable assumptions, e.g. more hours worked costs more $. Might also assume there is some base charge & contractor then charges per hour. If these assumptions held true, then given these 2 data examples, could already identify contractor's pricing structure: 100$ per hour + 50$ to show up at your house. If you followed that much, then you already understand high-level idea behind *linear* regression.
      In this case, could produce parameters exactly matched contractor's prices. Sometimes this is not possible, e.g., if some of variation arises from factors beyond your 2 features. In these cases, will try to learn models that minimize distance between our predictions & observed values. In most of chaps, will focus on minimizing squared error loss function. This loss corresponds to assumption: our data were corrupted by Gaussian noise.
    - Classification. While regression models are great for addressing *how many?* questions, lots of problems do not fit comfortably in this template. Consider, e.g., a bank that wants to develop a check scanning feature for its mobile app. Ideally, customer would simply snap a photo of a check & app would automatically recognize text from image. Assume: had some ability to segment out image patches corresponding to each handwritten character, then primary remaining task would be to determine which character among some known set is depicted in each image patch. These

kinds of *which one?* problems are called *classification* & require a different set of tools from those used for regression, although many techniques will carry over.

In *classification*, want our model to look at features, e.g., pixel values in an image, & then predict to which *category* (sometimes called a *class*) among some discrete set of options, an example belongs. For handwritten digits, might have 10 classes, corresponding to digits 9 through 9. Simplest form of classification is when there are only 2 classes, a problem which we call *binary classification*. E.g., our dataset could consist of images of animals & our labels might be classes {cat, dog}. Whereas in regression, sought a regressor to output a numerical value, in classification seek a classifier, whose output is predicted class assignment.

Can be difficult to optimize a model that can only output a *firm* categorical assignment, e.g., either "cat" or "dog". In these cases, usually much easier to express our model in language of probabilities. Given features of an example, our model assigns a probability to each possible class. Return to animal classification example where classes are {cat, dog}, a classifier might see an image & output probability: image is a cat as 0.9. Can interpret this number by saying: classifier is 90% sure: image depicts a cat. Magnitude of probability for predicted class conveys a notion of uncertainty. Not only one available & discuss others in chaps dealing with more advanced topics.

When have > 2 possible classes, call problem *multiclass classification*. Common examples include handwritten character recognition $\{0, 1, \ldots, 9, a, b, c, \ldots\}$. While attacked regression problems by trying to minimize squared error loss function, common loss function for classification problems is called *cross-entropy*, whose name will be demystified when introduce information theory.

Note: most likely class is not necessarily one that you are going to use for your decision. Assume: find a beautiful mushroom in your backyard as shown in Fig. 1.3.2: Death cap - do not eat!

Now, assume: built a classifier & trained it to predict whether a mushroom is poisonous based on a photograph. Say poison-detection classifier outputs: probability Fig. 1.3.2 shows a death cap is 0.2. I.e., classifier is 80% sure: our mushroom is not a death cap. Still, would have to be a fool to eat it. Because certain benefit of a delicious dinner is not worth a 20% risk of dying from it. I.e., effect of uncertain risk outweighs benefit by far. Thus, in order to make a decision about whether to eat mushroom, need to compute expected detriment associated with each action which depends both on likely outcomes & benefits or harms associated with each. In this case, detriment incurred by eating mushroom might be $0.2 \cdot \infty + 0.8 \times 0 = \infty$, whereas loss of discarding it is $0.2 \cdot 0 + 0.8 \cdot 1 = 0.8$. Caution was justified: as any mycologist would tell us, this mushroom is actually a death cap.

Classification can get much more complicated than just binary or multiclass classification. E.g., there are some variants of classification addressing hierarchically structured classes. In such cases not all errors are equal – if we must err, might prefer to misclassify to a related class rather than a distant class. Usually, this is referred to as *hierarchical classification*. For inspiration, might think of Linnaeus https://en.wikipedia.org/wiki/Carl_Linnaeus, who organized fauna in a hierarchy.

In case of animal classification, it might not be so bad to mistake a poodle for a schnauzer, but our model would pay a huge penalty if it confused a poodle with a dinosaur. Which hierarchy is relevant might depend on how you plan to use model. E.g., rattlesnakes & garter snakes might be close on phylogenetic tree, but mistaking a rattler for a garter could have fatal consequences.

- **Tagging.** Some classification problems fit neatly into binary or multiclass classification setups. E.g., could train a normal binary classifier to distinguish cats from dogs. Given current state of computer vision, can do this easily, with off-the-shelf tools. Nonetheless, no matter how accurate model gets, might find ourselves in trouble when classifier encounters an image of *town Musicians of Bremen*, a popular German fairy tale featuring 4 animals Fig. 1.3.3: A donkey, a dog, a cat, & a rooster.

  Photo features a cat, a rooster, a dog, & a donkey, with some trees in background. If anticipate encountering such images, multiclass classification might not be right problem formulation. Instead, might want to give model option of saying image depicts a cat, a dog, a donkey, *&* a rooster.

  Problem of learning to predict classes that are not mutually exclusive is called *multi-label classification*. Auto-tagging problems are typically best described in terms of multi-label classification. Think of tags people might apply to posts on a technical blog, e.g., "machine learning", "technology", "gadgets", "programming languages", "Linux", "cloud computing", "AWS". A typical article might have 5–10 tags applied. Typically, tags will exhibit some correlation structure. Posts about "cloud computing" are likely to mention "AWS" & posts about "ML" are likely to mention "GPUs".

  Sometimes such tagging problems draw on enormous label sets. National Library of Medicine employs many professional annotators who associate each article to be indexed in PubMed with a set of tags drawn from Medical Subject Headings (MeSH) ontology, a collection of roughly 28000 tags. Correctly tagging articles is important because it allows researchers to conduct exhaustive reviews of literature. This is a time-consuming process & typically there is a 1-year lag between archiving & tagging. ML can provide provisional tags until each article has a proper manual review. Indeed, for several years, BioASQ organization has hosted competitions http://bioasq.org/ for this task.

- **Search.** In field of information retrieval, often impose ranks on sets of items. Take web e.g. Goal is less to determine *whether* a particular page is relevant for a query, but rather which, among a set of relevant results, should be shown most prominently to a particular user. 1 way of doing this might be to 1st assign a score to every element in set & then to retrieve top-rated elements. PageRank https://en.wikipedia.org/wiki/PageRank, original secret sauce behind Google search engine, was an early example of such a scoring system. Weirdly, scoring provided by PageRank did not depend on actual query. Instead, they relied on a simple relevance filter to identify set of relevant candidates

& then used PageRank to prioritize more authoritative pages. Nowadays, search engines use ML & behavioral models to obtain query-dependent relevance scores. There are entire academic conferences devoted to this subject.

- **Recommender System.** Recommender systems are another problem setting that is related to search & ranking. Problems are similar insofar as goal is to display a set of items relevant to user. Main difference: emphasis on *personalization* to specific users in context of recommender systems. E.g., for movie recommendations, results page for a science fiction fan & results page for a connoisseur of Peter Sellers comedies might differ significantly. Similar problems pop up in other recommendation settings, e.g., for retail products, music, & news recommendation.
  In some cases, customers provide explicit feedback, communicating how much they liked a particular product (e.g., product ratings & reviews on Amazon, IMDb, or Goodreads). In other cases, they provide implicit feedback, e.g., by skipping titles on a playlist, which might indicate dissatisfaction or maybe just indicate: song was inappropriate in context. In simplest formulations, these systems are trained to estimate some score, e.g. an expected star rating or probability that a given user will purchase a particular item.
  Given such a model, for any given user, could retrieve set of objects with largest scores, which could then be recommended to user. Production systems are considerably more advanced & take detailed user activity & item characteristics into account when computing such scores. Fig. 1.3.4: DL books recommended by Amazon displays DL books recommended by Amazon based on personalization algorithms tuned to capture Aston's preferences.
  Despite their tremendous economic value, recommender systems naively built on top of predictive models suffer some serious conceptual flaws. to start, only observe *censored feedback*: users preferentially rate movies that they feel strongly about. E.g., on a 5-point scale, might notice: items receive many 1- & 5-star ratings but that there are conspicuously few 3-star ratings. Moreover, current purchase habits are often a result of recommendation algorithm currently in place, but learning algorithms do no always take this detail into account. Thus possible for feedback loops to form where a recommender system preferentially pushes an item that is then taken to be better (due to greater purchases) & in turn is recommended even more frequently. Many of these problems – about how to deal with censoring, incentives, & feedback loops – are important open research questions.
- **Sequence Learning.** So far, have looked at problems where have some fixed number of inputs & produced a fixed number of outputs. E.g., considered predicting house prices given a fixed-set of features: square footage, number of bedrooms, number of bathrooms, & transit time to downtown. Also discussed mapping from an image (of fixed dimension) to predicted probabilities that it belongs to each among a fixed number of classes & predicting star ratings associated with purchases based on user ID & product ID alone. In these cases, once our model is trained, after each test example is fed into our model, it is immediately forgotten. Assumed: successive observations were independent & thus there was no need to hold on to this context.
  But how should we deal with video snippets? In this case, each snippet might consist of a different number of frames. & our guess of what is going on in each frame might be much stronger if we take into account previous or succeeding frames. Same goes for language. E.g., 1 popular DL problem is machine translation: task of ingesting sentences in some source language & predicting their translations in another language.
  Such problems also occur in medicine. Might want a model to monitor patients in intensive care unit & to fire off alerts whenever their risk of dying in next 24 hours exceeds some threshold. Here, would not throw away everything that we know about patient history every hour, because might not want o make predictions based only on most recent measurements.
  Questions like these are among most exciting applications of ML & they are instances of *sequence learning*. They require a model either to ingest sequences of inputs or to emit sequences of outputs (or both). Specifically, *sequence-to-sequence learning* considers problems where both inputs & outputs consist of variable-length sequences. Examples include machine translation & speech-to-text transcription. While impossible to consider all types of sequence transformations, following special cases are worth mentioning.
  (a) **Tagging & Parsing.** This involves annotating a text sequence with attributes. Here, inputs & outputs are *aligned*, i.e., they are of same number & occur in a corresponding order. E.g., in *part-of-speech (PoS) tagging*, annotate every word in a sentence with corresponding part of speech, i.e., "noun" or "direct object". Alternatively, might want to know which groups of contiguous words refer to named entities, like *people, places*, or *organizations*. In cartoonishly simple example below, might just want to indicate whether or not any word in sentence is part of a named entity (tagged as "Ent").
  (b) **Automatic Speech Recognition.** With speech recognition, input sequence is an audio recording of a speaker Fig. 1.3.5: -D-e-e-p- L-ea-r-ni-ng- in an audio recording, & output is a transcript of what speaker said. Challenge: there are many more audio frames (sound is typically sampled at 8kHz or 16kHz) than text, i.e., there is no 1:1 correspondence between audio & text, since thousands of samples may correspond to a single spoken word. These are sequence-to-sequence learning problems, where output is much shorter than input. While humans are remarkably good at recognizing speech, even from low-quality audio, getting computers to perform same feat is a formidable challenge.
  (c) **Text to Speech.** Inverse of automatic speech recognition. Here, input is text & output is an audio file. In this case, output is much longer than input.
  (d) **Machine Translation.** Unlike case of speech recognition, where corresponding inputs & outputs occur in same order, in machine translation, unaligned data poses a new challenge. Here input & output sequences can have different lengths, & corresponding regions of respective sequences may appear in a different order. Consider following illustrative example of peculiar tendency of Germans to place verbs at end of sentences:

```
German: Haben Sie sich schon dieses grossartige Lehrwerk angeschaut?
English: Have you already looked at this excellent textbook?
Wrong alignment: Have you yourself already this excellent textbook looked at?
```

Many related problems pop up in other learning tasks. E.g., determining order in which a user reads a webpage is a 2D layout analysis problem. Dialogue problems exhibit all kinds of additional complications, where determining what to say next requires taking into account real-world knowledge & prior state of conversation across long temporal distances. Such topics are active areas of research.

∗ 1.3.2. Unsupervised & Self-Supervised Learning. Previous examples focused on supervised learning, where we feed model a giant dataset containing both features & corresponding label values. Could think of supervised learner as having an extremely specialized job & an extremely dictatorial boss. Boss stands over learner's shoulder & tells them exactly what to do in every situation until they learn to map from situations to actions. Working for such a boss sounds pretty lame. On other hand, pleasing such a boss is pretty easy. Just recognize pattern as quickly as possible & imitate boss's actions.

Considering opposite situation, it could be frustrating to work for a boss who has no idea what they want you to do. However, if you plan to be a data scientist, you had better get used to it. Boss might just hand you a giant dump of data & tell you to *do some data science with it!* This sounds vague because it is vague. Call this class of problems *unsupervised learning*, & type & number of questions we can ask is limited only by our creativity. Will address unsupervised learning techniques in later chaps. To whet your appetite for now, describe a few of following questions you might ask.

- Can we find a small number of prototypes that accurately summarize data? Given a set of photos, can we group them into landscape photos, pictures of dogs, babies, cats, & mountain peaks? Likewise, given a collection of users' browsing activities, can we group them into users with similar behavior? This problem is typically known as *clustering* (phân cụm).
- Can we find a small number of parameters that accurately capture relevant properties of data? Trajectories of a ball are well described by velocity, diameter, & mass of ball. Tailors have developed a small number of parameters that describe human body shape fairly accurately for purpose of fitting clothes. These problems are referred to as *subspace estimation*. If dependence is linear, called *principal component analysis*.
- Is there a representation of (arbitrarily structured) objects in Euclidean space s.t. symbolic properties can be well matched? This can be used to describe entities & their relations, e.g. "Rome" − "Italy" + "France" = "Paris".
- Is there a description of root causes of much of data that we observe. E.g., if have demographic data about house prices, pollution, crime, location, education, & salaries, can discover how they are related simply based on empirical data? Fields concerned with *causality & probabilistic graphical models* tackle such questions.
- Another important & exciting recent development in unsupervised learning: advent of *deep generative models* (sự ra đời của các mô hình sinh sản sâu sắc). These models estimate density of data, either explicitly or *implicitly*. Once trained, can use a generative model either to score examples according to how likely they are, or to sample synthetic examples from learned distribution. Early deep learning breakthroughs in generative modeling came with invention with *variational autoencoders* (Kingma & Welling, 2014, Rezende et al., 2014) & continued with development of *generative adversarial networks* (Goodfellow et al., 2014). More recent advances include normalizing flows (Dinh et al., 2014, Dinh et al., 2017) & diffusion models (Ho et al., 2020, Sohl-Dickstein et al., 2015, Song & Ermon, 2019, Song et al., 2021).

A further development in unsupervised learning has been rise of *self-supervised learning*, techniques that leverage some aspect of unlabeled data to provide supervision. For text, can train models to "fill in the blanks" by predicting randomly masked words using their surrounding words (contexts) in big corpora without any labeling effort (Devlin et al., 2018)! For images, may train models to tell relative position between 2 cropped regions of same image (Doersch et al., 2015), to predict an occluded (bị che khuất) part of an image based on remaining portions of image, or to predict whether 2 examples are perturbed versions of same underlying image. Self-supervised models often learn representations that are subsequently leveraged by fine-tuning resulting models on some downstream task of interest.

∗ 1.3.3. Interacting with an Environment. So far, have no discussed where data actually comes from, or what actually happens when a ML model generates an output. Because supervised learning & unsupervised learning do not address these issues in a very sophisticated way. In each case, grab a big pile of data upfront, then set our pattern recognition machines in motion without ever interacting with environment again. Because all learning takes place after algorithm is disconnected from environment, this is sometimes called *offline learning*. E.g., supervised learning assumes simple interaction pattern depicted in Fig. 1.3.6: Collecting data for supervised learning from an environment.

This simplicity of offline learning has its charms. Upside: we can worry about pattern recognition in isolation, with no concern about complications arising from interactions with a dynamic environment. But this problem formulation is limiting. If grew up reading Asimov's Robot novels, then probably picture artificially intelligent agents capable not only of making predictions, but also of taking actions in world. Want to think about intelligent *agents*, not just predictive models. I.e., need to think about choosing *actions*, not just making predictions. In contrast to mere predictions, actions actually impact environment. If want to train an intelligent agent, must account for way its actions might impact future observations of agent, & so offline learning is inappropriate.

Considering interaction with an environment opens a whole set of new modeling questions. Just a few examples:

- Does environment remember what we did previously?
- Does environment want to help us, e.g., a user reading text into a speech recognizer?
- Does environment want to beat us, e.g., spammers adapting their emails to evade spam filters?
- Does environment have shifting dynamics? E.g., would future data always resemble past or would patterns change over time, either naturally or in response to our automated tools?

These questions raise problem of *distribution shift*, where training & test data are different. An example of this: many of us may have met, is when taking exams written by a lecturer, while homework was composed by their teaching assistants. Next, briefly describe reinforcement learning, a rich framework for posing learning problems in which an agent interacts with an environment.

∗ 1.3.4. Reinforcement Learning. If interested in using ML to develop an agent that interacts with an environment & takes actions, then you are probably going to wind up focusing on *reinforcement learning*. This might include applications to robotics, to dialogue systems, & even to developing AI for video games. *Deep reinforcement learning*, which applies DL to reinforcement learning problems, has surged in popularity. Breakthrough deep Q-network, that beat humans at Atari games using only visual input (Mnih et al., 2015), & AlphaGo program, which dethroned world champion at board game Go (Silver et al., 2016), are 2 prominent examples.

Reinforcement learning gives a very general statement of a problem in which an agent interacts with an environment over a series of time steps. At each time step, agent receives some *observation* from environment & must choose an *action* that is subsequently transmitted back to environment via some mechanism (sometimes called an *actuator*), when, after each loop, agent receives a reward from environment. This process is illustrated in Fig. 1.3.7: Interaction between reinforcement learning & an environment. Agent then receives a subsequent observation, & chooses a subsequent action, & so on. Behavior of a reinforcement learning agent is governed by a *policy*. In brief, a *policy* is just a function that maps from observations of environment to actions. Goal of reinforcement learning: to produce good policies.

Hard to overstate generality of reinforcement learning framework. E.g., supervised learning can be recast as reinforcement learning. Say we had a classification problem. Could create a reinforcement learning agent with 1 action corresponding to each class. Could then create an environment which gave a reward that was exactly = loss function from original supervised learning problem.

Further, reinforcement learning can also address many problems that supervised learning cannot. E.g., in supervised learning, always expect: training input comes associated with correct label. But in reinforcement learning, do not assume that, for each observation environment tells us optimal action. In general, just get some reward. Moreover, environment may not even tell us which actions led to reward.

Consider game of chess. only real reward signal comes at end of game when we either win, earning a reward of, say, 1, or when we lose, receiving a reward of, say, -1. So reinforcement learners must deal with *credit assignment* problem: determining which actions to credit or blame for an outcome. Same goes for an employee who gets a promotion on Oct 11. That promotion likely reflects a number of well-chosen actions over previous year. Getting promoted in future requires figuring out which actions along way led to earlier promotions.

Reinforcement learners may also have to deal with problem of partial observability. I.e., current observation might not tell you everything about your current state. Say your cleaning robot found itself trapped in 1 of many identical closets in your house. Rescuing robot involves inferring its precise location which might require considering earlier observations prior to it entering closet.

Finally, at any given point, reinforcement learners might know of 1 good policy, but there might be many other better policies that agent has never tried. Reinforcement learner must constantly choose whether to *exploit* best (currently) known strategy as a policy, or to *explore* space of strategies, potentially giving up some short-term reward in exchange for knowledge.

General reinforcement learning problem has a very general setting. Actions affect subsequent observations. Rewards are only observed when they correspond to chosen actions. Environment may be either fully or partially observed. Accounting for all this complexity at once may be asking too much. Moreover, not every practical problem exhibits all this complexity. As a result, researchers have studied a number of special cases of reinforcement learning problems.

When environment is fully observed, call reinforcement learning problem a *Markov decision process*. When state does not depend on previous actions, call it a *contextual bandit problem* (vấn đề cướp bối cảnh). When there is no state, just a set of available actions with initially unknown rewards, have classic *multi-armed bandit problem* (bài toán máy đánh bạc nhiều tay).

◦ 1.4. Roots. Have just reviewed a small subset of problems that ML can address. For a diverse set of ML problems, DL provides powerful tools for their solution. Although many DL methods are recent inventions, core ideas behind learning from data have been studied for centuries. In fact, humans have held desire to analyze data & to predict future outcomes for ages, & it is this desire that is at root of much of natural science & mathematics. 2 examples: Bernoulli distribution, named after Jacob Bernoulli (1655–1705) & Gaussian distribution discovered by Carl Friedrich Gauss (1777–1855). Gauss invented, e.g., least mean squares algorithm, still used today for a multitude of problems from insurance calculations to medical diagnostics. Such tools enhanced experimental approach in natural sciences – e.g., Ohm's law relating current & voltage in a resistor is perfectly described by a linear model.

Even in middle ages, mathematicians had a keen intuition of estimates. E.g., geometry book of Jacob Köbel (1460–1533) https://www.maa.org/press/periodicals/convergence/mathematical-treasures-jacob-kobels-geometry illustrates averaging length of 16 adult men's feet to estimate typical foot length in population Fig. 1.4.1: Estimating length of a foot.

As a group of individuals exited a church, 16 adult men were asked to line up in a row & have their feet measured. Sum of these measurements was then divided by 16 to obtain an estimate for what now is called 1 foot. This "algorithm" was later improved to deal with misshapen feet; 2 men with shortest & longest feet were sent away, averaging only over remainder. This is among earliest examples of a trimmed mean estimate.

Statistics really took off with availability & collection of data. 1 of its pioneers, Ronald Fisher (1890–1962), contributed significantly to its theory & also its applications in genetics. Many of his algorithms (e.g. linear discriminant analysis) & concepts (e.g. Fisher information matrix) still hold a prominent place in foundations of modern statistics. Even his data resources had a lasting impact. Iris dataset that Fisher released in 1936 is still sometimes used to demonstrate ML algorithms. Fisher was also a proponent of eugenics, which should remind us: morally dubious use of DS has as long & enduring a history as its productive use in industry & natural sciences.

Other influences for ML came from information theory of Claude Shannon (1916–2001) & theory of computation proposed by Alan Turing (1912–1954). Turing posed question "can machines think?" in his famous paper *Computing Machinery & Intelligence* (Turing, 1950). Describing what is now known as *Turing test*, he proposed that a machine can be considered *intelligent* if difficult for a human evaluator to distinguish between replies from a machine & those of a human, based purely on textual interactions.

Further influences came from neuroscience & psychology. After all, humans clearly exhibit intelligent behavior. Many scholars have asked whether one could explain & possibly reverse engineer this capacity. 1 of 1st biologically inspired algorithms was formulated by Donald Hebb (1904–1985). In his groundbreaking book *The Organization of Behavior* (Hebb, 1949), he posited: neurons learn by positive reinforcement. This became known as *Hebbian learning rule*. These ideas inspired later work, e.g. Rosenblatt's perceptron learning algorithm, & laid foundations of many stochastic gradient descent algorithms that underpin deep learning today: reinforce desirable behavior & diminish undesirable behavior to obtain good settings of parameters in a neural network.

Biological inspiration is what gave *neural networks* their name. For over a century (dating back to models of Alexander Bain, 1873, & ßJames Sherrington, 1890), researchers have tried to assemble computational circuits that resemble networks of interacting neurons. Over time, interpretation of biology has become less literal, but name stuck. At its heart lie a few key principles that can be found in most networks today:

* Alternation of linear & nonlinear processing units, often referred to as *layers*.
* Use of chain rule (also known as *backpropagation*) for adjusting parameters in entire network at once.

After initial rapid progress, research in neural networks languished from around 1995 until 2005. This was mainly due to 2 reasons. 1st, training a network is computationally very expensive. While random-access memory was plentiful at end of past century, computational power was scarce. 2nd, datasets were relatively small. In fact, Fisher's Iris dataset from 1936 was still a popular tool for testing efficacy of algorithms. MNIST dataset with its 60000 handwritten digits was considered huge.

Given scarcity (sự khan hiếm) of data & computation, strong statistical tools e.g. kernel methods, decision trees, & graphical models proved empirically superior in many applications. Moreover, unlike neural networks, they did not require weeks to train & provided predictable results with strong theoretical guarantees.

- 1.5. Road to Deep Learning. Much of this changed with availability of massive amounts of data, thanks to World Wide Web, advent of companies serving hundreds of millions of users online, a dissemination of low-cost, high-quality sensors, inexpensive data storage (Kryder's law), & cheap computaiton (Moore's law). In particular, landscape of computation in DL was revolutionized by advances in GPUs that were originally engineered for computer gaming. Suddenly algorithms & models that seemed computationally infeasible were within reach. This is illustrated in `tab_intro_decade` dataset vs. computer memory & computational power.

  Note: random-access memory has not kept pace with growth in data. At same time, increases in computational power have outpaced growth in datasets. I.e., statistical models need to become more memory efficient, & so they are free to spend more computer cycles optimizing parameters, thanks to increased compute budget. Consequently, sweet spot in ML & statistics moved from (generalized) linear models & kernel methods to deep neural networks. Also 1 of reasons why many of mainstays of DL, e.g. multilayer perceptrons (McCulloch & Pitts, 1943), convolutional neural networks (LeCun et al., 1998), long short-term memory (Hochreiter & Schmidhuber, 1997), & Q-Learning (Watkins & Dayan, 1992), were essentially "rediscovered" in past decade, after lying comparatively dormant for considerable time (sau khi nằm im trong 1 thời gian khá dài).

  Recent progress in statistical models, applications, & algorithms has sometimes been likened (giống như) to Cambrian explosion: a moment of rapid progress in evolution of species. Indeed, state of art is not just a mere consequence of available resources applied to decades-old algorithms. Note: list of ideas below barely scratches surface of what has helped researchers achieve tremendous progress over past decade.

  * Novel methods for capacity control, e.g. *dropout* (Srivastava et al., 2014), have helped to mitigate overfitting. Here, noise is injected (Bishop, 1995) throughout neural network during training.
  * *Attention mechanisms* solved a 2nd problem that had plagued (quấy rầy) statistics for over a century: how to increase memory & complexity of a system without increasing number of learnable parameters. Researchers found an elegant solution by using what can only be viewed as a *learnable pointer structure* (Bahdanau et al., 2014). Rather than having to remember an entire text sequence, e.g., for machine translation in a fixed-dimensional representation, all that needed to be stored was a pointer to intermediate state of translation process. This allowed for significantly increased accuracy for long sequences, since model no longer needed to remember entire sequence before commencing generation of a new one.

- Built solely on attention mechanisms, *Transformer* architecture (Vaswani et al., 2017) has demonstrated superior *scaling* behavior: it performs better with an increase in dataset size, model size, & amount of training compute (Kaplan et al., 2020). This architecture has demonstrated compelling success in a wide range of areas, e.g. natural language processing (Brown et al., 2020, Devlin et al., 2018), computer vision (Dosovitskiy et al., 2021, Liu et al., 2021), speech recognition (Gulati et al., 2020), reinforcement learning (Chen et al., 2021), & graph neural networks (Dwivedi & Bresson, 2020). E.g., a single Transformer pretrained on modalities as diverse as text, images, joint torques, & button presses can play Atari, caption images, chat, & control a robot (Reed et al., 2022).
- Modeling probabilities of text sequences, *language models* can predict text given other text. Scaling up data, model, & compute has unlocked a growing number of capabilities of language models to perform desired tasks via human-like text generation based on input text (Anil et al., 2023, Brown et al., 2020, Chowdhery et al., 2022, Hoffmann et al., 2022, OpenAI, 2023, Rae et al., 2021, Touvron et al., 2023a, Touvron et al., 2023b). E.g., aligning language models with human intent (Ouyang et al., 2022), OpenAIs ChatGPT https://chat.openai.com/ allows users to interact with it in a conversational way to solve problems, e.g. code debugging & creative writing.
- Multi-stage designs, e.g., via memory networks (Sukhbaatar et al., 2015) & neural programmer-interpreter (Reed & De Freitas, 2015) permitted statistical modelers to describe iterative approaches to reasoning. These tools allow for an internal state of deep neural network to be modified repeatedly, thus carrying out subsequent steps in a chain of reasoning, just as a processor can modify memory for a computation.
- A key development in *deep generative modeling* was invention of *generative adversarial networks* (Goodfellow et al., 2014). Traditionally, statistical methods for density estimation & generative models focused on finding proper probability distributions & (often approximate) algorithms for sampling from them. As a result, these algorithms were largely limited by lack of flexibility inherent in statistical models. Crucial innovation in generative adversarial networks was to replace sampler by an arbitrary algorithm with differentiable parameters. These are then adjusted in such a way that discriminator (effectively a 2-sample test) cannot distinguish fake from real data. Through ability to use arbitrary algorithms to generate data, density estimation was opened up to a wide variety of techniques. Examples of galloping zebras (Zhu et al., 2017) & of fake celebrity faces (Karras et al., 2017) are each testimony to this progress. Even amateur doodlers can produce photorealistic images just based on sketches describing layout of a scene (Park et al., 2019).
- Furthermore, while diffusion process gradually adds random noise to data samples, *diffusion models* (Ho et al., 2020, Sohl-Dickstein et al., 2015) learn denoising process to gradually construct data samples from random noise, reversing diffusion process. They have started to replace generative adversarial networks in more recent deep generative models, e.g. in DALL-E 2 (Ramesh et al., 2022) & Imagen (Saharia et al., 2022) for creative art & image generation based on text descriptions.
- In many cases, a single GPU is insufficient for processing large amounts of data available for training. Over past decade ability to build parallel & distributed training algorithms has improved significantly. 1 of key challenges in designing scalable algorithms: workhorse (ngựa thồ) of DL optimization, stochastic gradient descent, relies on relatively small minibatches of data to be processed. At same time, small batches limit efficiency of GPUs. Hence, training on 1024 GPUs with a minibatch size of, say, 32 images per batch amounts to an aggregate minibatch of about 32000 images. Work, 1st by Li (2017) & subsequently by You et al. (2017) & Jia et al. (2018) pushed size up to 64,000 observations, reducing training time for ResNet-50 model on ImageNet dataset to < 7 minutes. By comparison, training times were initially of order of days.
- Ability to parallelize computation has also contributed to progress in *reinforcement learning*. This has led to significant progress in computers achieving superhuman performance on tasks like Go, Atari games, Starcraft, & in physics simulations (e.g., using MuJoCo) where environment simulators are available. See, e.g., Silver et al. (2016) for a description of such achievements in AlphaGo. In a nutshell, reinforcement learning works best if plenty of (state, action, reward) tuples are available. Simulation provides such an avenue.
- DL frameworks have played a crucial role in disseminating ideas (truyền bá ý tưởng). 1st generation of open-source frameworks for neural network modeling consisted of Caffe https://github.com/BVLC/caffe, Torch https://github.com/torch, & Theano https://github.com/Theano/Theano. Many seminal papers were written using these tools. These have now been superseded by TensorFlow https://github.com/tensorflow/tensorflow (often used via its high-level API Keras https://github.com/keras-team/keras), CNTK https://github.com/Microsoft/CNTK, Caffe 2 https://github.com/caffe2/caffe2, & Apache MXNet https://github.com/apache/incubator-mxnet. 3rd generation of frameworks consists of so-called *imperative* tools for deep learning, a trend that was arguably ignited by Chainer https://github.com/chainer/chainer, which used a syntax similar to Python NumPy to describe models. This idea was adopted by both PyTorch https://github.com/pytorch/pytorch, Gluon API https://github.com/apache/incubator-mxnet of MXNet, & JAX https://github.com/google/jax.

Division of labor between system researchers building better tools & statistical modelers building better neural networks has greatly simplified things. E.g., training a linear logistic regression model used to be a nontrivial homework problem, worthy to give to new ML Ph.D. students at Carnegie Mellon University in 2014. By now, this task can be accomplished with < 10 lines of code, putting it firmly within reach of any programmer.

- 1.6. Success Stories. AI has a long history of delivering results that would be difficult to accomplish otherwise. E.g., mail sorting systems using optical character recognition have been deployed since 1990s. This is, after all, source of famous MNIST dataset of handwritten digits. Same applies to reading checks for bank deposits & scoring creditworthiness of applicants. Financial transactions are checked for fraud automatically. This forms backbone of many e-commerce

payment systems, e.g. Paypal, Stripe, AliPay, WeChat, Apple, Visa, & MasterCard. Computer programs for chess have been competitive for decades. ML feeds search, recommendation, personalization, & ranking on Internet. I.e., ML is pervasive, albeit often hidden from sight – Học máy rất phổ biến, mặc dù thường bị ẩn khỏi tầm nhìn.

Only recently: AI has been in limelight, mostly due to solutions to problems that were considered intractable previously & that are directly related to consumers. Many of such advances are attributed to DL.

* Intelligent assistants, e.g. Apple's Siri, Amazon's Alexa, & Google's assistant, are able to respond to spoken requests with a reasonable degree of accuracy. This includes menial jobs, like turning on light switches, & more complex tasks, e.g. arranging barber's appointments & offering phone support dialog. This is likely most noticeable sign that AI is affecting our lives.
* A key ingredient in digital assistants is their ability to recognize speech accurately. Accuracy of such systems has gradually increased to point of achieving parity with humans for certain applications (Xiong et al., 2018).
* Object recognition has likewise come a long way. Identifying object in a picture was a fairly challenging task in 2010. On ImageNet benchmark researchers from NEC Labs & University of Illinois at Urbana-Champaign achieved a top-5 error rate of 28% (Lin et al., 2010). By 2017, this error rate was reduced to 2.25% (Hu et al., 2018). Similarly, stunning results have been achieved for identifying birdsong & for diagnosing skin cancer.
* Prowess in games (Tài năng trong trò chơi) used to provide a measuring stick for human ability. Starting from TD-Gammon, a program for playing backgammon using temporal difference reinforcement learning, algorithmic & computational progress has led to algorithms for a wide range of applications. Compared with backgammon, chess has a much more complex state space & set of actions. DeepBlue beat Garry Kasparov using massive parallelism, special-purpose hardware & efficient search through game tree (Campbell et al., 2002). Go is more difficult still, due to its huge state space. AlphaGo reached human parity (sự bình đẳng của con người) in 2015, using DL combined with Monte Carlo tree sampling (Silver et al., 2016). Challenge in Poker was: state space is large & only partially observed (do not know opponents' cards). Libratus exceeded human performance in Poker using efficiently structured strategies (Brown & Sandholm, 2017).
* Another indication of progress in AI: advent of self-driving vehicles (sự ra đời của xe tự lái). While full autonomy is not yet within reach, excellent progress has been made in this direction, with companies e.g. Tesla, NVIDIA, & Waymo shipping products that enable partial autonomy. What makes full autonomy so challenging: proper driving requires ability to perceive, to reason & to incorporate rules into a system. At present, DL is used primarily in visual aspect of these problems. The rest is heavily tuned by engineers.

This barely scratches surface of significant applications of ML. E.g., robotics, logistics, computational biology, particle physics, & astronomy owe some of their most impressive recent advances at least in parts to ML, which is thus becoming a ubiquitous tool for engineers & scientists.

Frequently, questions about a coming AI apocalypse & plausibility of a *singularity* have been raised in non-technical articles. Thông thường, các câu hỏi về ngày tận thế sắp tới của AI & khả năng xảy ra *điểm kỳ dị* đã được nêu ra trong các bài viết không mang tính kỹ thuật. Fear: somehow ML systems will become sentient & make decisions, independently of their programmers, that directly impact lives of humans. To some extent, AI already affects livelihood of humans in direct ways: creditworthiness is assessed automatically, autopilots mostly navigate vehicles, decisions about whether to grant bail use statistical data as input. More frivolously, can ask Alexa to switch on coffee machine.

Fortunately, we are far from a sentient AI system that could deliberately manipulate its human creators. 1st, AI systems are engineered, trained, & deployed in a specific, goal-oriented manner. While their behavior might give illusion of general intelligence, it is a combination of rules, heuristics & statistical models that underlie design. 2nd, at present, there are simply no tools for *artificial general intelligence* that are able to improve themselves, reason about themselves, & that are able to modify, extend, & improve their own architecture while trying to solve general tasks.

A much more pressing concern is how AI is being used in our daily lives. Likely: many routine tasks, currently fulfilled by humans, can & will be automated. Farm robots will likely reduce costs for organic farmers but they will also automate harvesting operations. This phase of industrial revolution may have profound consequences for large swaths of society, since menial jobs provide much employment in many countries. Furthermore, statistical models, when applied without care, can lead to racial, gender, or age bias & raise reasonable concerns about procedural fairness if automated to drive consequential decisions. Important to ensure: these algorithms are used with care. With what we know today, this strikes us as a much more pressing concern than potential of malevolent superintelligence for destroying humanity.

- 1.7. Essence of Deep Learning. Thus far, have talked in broad terms about ML. DL is subset of ML concerned with models based on many-layered neural networks. *Deep* in precisely sense that its models learn many *layers* of transformations. While this might sound narrow, DL has given rise to a dizzying (chóng mặt) array of models, techniques, problem formulations, & applications. Many intuitions have been developed to explain benefits of depth. Arguably, all ML has many layers of computation, 1st computing of feature processing steps. What differentiates DL: operations learned at each of many layers of representations are learned jointly from data.

  Problems that have discussed so far, e.g. learning from raw audio signal, raw pixel values of images, or mapping between sentences of arbitrary lengths & their counterparts in foreign languages, are those where DL excels & traditional methods falter (những nơi mà DL vượt trội & các phương pháp truyền thống không hiệu quả). Turn out: these many-layered models are capable of addressing low-level perceptual data in a way that previous tools could not – Thực tế: các mô hình nhiều lớp này có khả năng giải quyết dữ liệu nhận thức cấp thấp theo cách mà các công cụ trước đây không thể làm được. Arguably most significant commonality in DL methods is *end-to-end training*. I.e., rather than assembling a system based on components that are individually tuned, one builds system & then tunes their performance jointly.

E.g., in computer vision scientists used to separate process of *feature engineering* from process of building ML models. Canny edge detector (Canny, 1987) & Lowe's SIFT feature extractor (Lowe, 2004) reigned supreme for over a decade as algorithms for mapping images into feature vectors. In bygone days, crucial part of applying Ml to these problems consisted of coming up with manually-engineered ways of transforming data into some form amenable to shallow models. Unfortunately, there is only so much that humans can accomplish by ingenuity in comparison with a consistent evaluation over millions of choices carried out automatically by an algorithm. When DL took over, these feature extractors were replaced by automatically tuned filters that yielded superior accuracy.

Thus, 1 key advantage of DL: DL replaces not only shallow models at end of traditional learning pipelines, but also labor-intensive process of feature engineering. Moreover, by replacing much of domain-specific preprocessing, DL has eliminated many of boundaries that previously separated computer vision, speech recognition, natural language processing, medical informatics, & other application areas, thereby offering a unified set of tools for tackling diverse problems.

Beyond end-to-end training, are experiencing a transition from parametric statistical descriptions to fully nonparametric models. When data is scarse, one needs to rely on simplifying assumptions about reality in order to obtain useful models. When data is abundant, these can be replaced by nonparametric models that better fit data. To some extent, this mirrors progress that physics experienced in middle of previous century with availability of computers. Rather than solving by hand parametric approximations of how electrons behave, one can now resort to numerical simulations of associated PDEs. This has led to much more accurate models, albeit often at expense of interpretation.

Another difference from previous work is acceptance of suboptimal solutions, dealing with nonconvex nonlinear optimization problems, & willingness to try things before proving them. This new-found empiricism in dealing with statistical problems, combined with a rapid influx of talent has led to rapid progress in development of practical algorithms, albeit in many cases at expense of modifying & re-inventing tools that existed for decades.

In the end, DL community prides itself on sharing tools across academic & corporate boundaries, releasing many excellent libraries, statistical models, & trained networks as open source. In this spirit: notebooks forming this book are freely available for distribution & use. Have worked hard to lower barriers of access for anyone wishing to learn about DL & hope: readers will benefit from this.

- 1.8. Summary. ML studies how computer systems can leverage experience (có thể tận dụng kinh nghiệm) (often data) to improve performance at specific tasks. ML combines ideas from statistics, data mining, & optimization. Often, ML is used as a means of implementing AI solutions. As a class of ML, representational learning focuses on how to automatically find appropriate way to represent data. Considered as multi-level representation learning through learning many layers of transformations, DL replaces not only shallow models at end of traditional ML pipelines, but also labor-intensive process of feature engineering. Much of recent progress in DL has been triggered by an abundance of data rising from cheap sensors & Internet-scale applications, & by significant progress in computation, mostly through GPUs. Furthermore, availability of efficient DL frameworks has made design & implementation of whole system optimization significantly easier & this is a key component in obtaining high performance.
- Exercises.
  (a) Which parts of code that you are currently writing could be "learned", i.e., improved by learning & automatically determining design choices that are made in your code? Does your code include heuristic design choices? What data might you need to learn desired behavior?
  (b) Which problems that you encounter have many examples for their solution, yet no specific way for automating them? These may be prime candidates for using DL.
  (c) Describe relationships between algorithms, data, & computation. How do characteristics of data & current available computational resources influence appropriateness of various algorithms?
  (d) Name some settings where end-to-end training is not currently default approach but where it might be useful.

- 2. Preliminaries. To prepare for your dive into DL, need a few survival skills:
  (a) techniques for storing & manipulating data
  (b) libraries for ingesting & preprocessing data from a variety of sources
  (c) knowledge of basic linear algebraic operations that we apply to high-dimensional data elements
  (d) just enough calculus to determine which direction to adjust each parameter in order to decrease loss function
  (e) ability to automatically compute derivatives so that you can forget much of calculus you just learned
  (f) some basic fluency in probability, our primary language for reasoning under uncertainty
  (g) some aptitude for finding answers in official documentation when you get stuck.

  In short, this chap provides a rapid introduction to basics that you will need to follow *most* of technical content in this book.

  - 2.1. Data Manipulation. In order to get anything done, need some way to store & manipulate data. Generally, there are 2 important things we need to do with data: (i) acquire them (thu thập dữ liệu); (ii) process them once they are inside computer. There is no point in acquiring data without some way to store it, so to start, get our hands dirty with n-dimensional arrays, also called *tensors*. If already know NumPy scientific computing package, this will be a breeze. For all modern DL framework, *tensor class* (`ndarray` in MXNet, `Tensor` in PyTorch & TensorFlow) resembles NumPy's `ndarray`, with a few killer features added. 1st, tensor class supports automatic differentiation (AD). 2nd, it leverages GPUs to accelerate numerical computation, whereas NumPy only runs on CPUs. These properties make neural networks both easy to code & fast to run.

* 2.1.1. Getting Started. To start, import PyTorch library. Note: package name is `torch`.

```
import torch
```

- 3. Linear Neural Networks for Regression.
- 4. Linear Neural Networks for Classification.
- 5. Multilayer Perceptrons.
- 6. Builder's Guide.
- 7. Convolutional Neural Networks.
- 8. Modern Convolutional Neural Networks.
- 9. Recurrent Neural Networks.
- 10. Recurrent Neural Networks.
- 11. Attention Mechanisms & Transformers.
- 12. Optimization Algorithms.
- 13. Computational Performance.
- 14. Computer Vision.
- 15. Natural Language Processing: Pretraining.
- 16. Natural Language Processing: Applications.
- 17. Reinforcement Learning.
- 18. Gaussian Processes.
- 19. Hyperparameter Optimization.
- 20. Generative Adversarial Networks.
- 21. Recommender Systems.
- Appendix A: Mathematics for Deep Learning.
- Appendix B: Tools for Deep Learning.

4.1 [RPK19]. M. Raissi, P. Perdikaris, G.E. Karniadakis. Physics-informed neural networks: A DL Framework for Solving Forward & Inverse Problems Involving Nonlinear PDEs

Journal of Computational Physics. [12432 citations]

Keywords. Data-driven scientific computing; ML; Predictive modeling; Runge–Kutta methods; Nonlinear dynamics

Abstract. Introduce *physics-informed neural networks* – neural networks that are trained to solve supervised learning tasks while respecting any given laws of physics described by general nonlinear PDEs. In this work, present our developments in context of solving 2 main classes of problems: data-driven solution & data-driven discovery of PDEs. Depending on nature & arrangement of available data, devise 2 distinct types of algorithms, namely continuous time & discrete time models. 1st type of models forms a new family of *data-efficient* spatio-temporal function approximators, while the latter type allows use of arbitrarily accurate implicit Runge–Kutta time stepping schemes with unlimited number of stages. Effectiveness of proposed framework is demonstrated through a collection of classical problems in fluids, quantum mechanics, reaction–diffusion systems, & propagation of nonlinear shallow-water waves.

- 1. Introduction.
- 2. Problem setup.
- 3. Data-driven solutions of PDEs.
- 4. Data-driven discovery of PDEs.
- 5. Conclusions. Have introduced *physics-informed neural networks*, a new class of universal function approximators that is capable of encoding any underlying physical laws that govern a given data-set, & can be described by PDEs. In this work, design data-driven algorithms for inferring solutions to general nonlinear PDEs, & constructing computationally efficient physics-informed surrogate models. Resulting methods showcase a series of promising results for a diverse collection of problems in computational science, & open path for endowing DL with powerful capacity of mathematical physics to model world around us. As DL technology is continuing to grow rapidly both in terms of methodological & algorithmic developments, believe: this is a timely contribution that can benefit practitioners across a wide range of scientific domains. Specific applications that can readily enjoy these benefits include, but are not limited to, data-driven forecasting of physical processes, model predictive control, multi-physics/multi-scale modeling & simulation.

  Must note however: proposed methods should not be viewed as replacements of classical numerical methods for solving PDEs (e.g., finite elements, spectral methods, etc.). Such methods have matured over last 50 years &, in many cases, meet robustness

& computational efficiency standards required in practice. Message: as advocated in Sect. 3.2: classical methods e.g. Runge–Kutta time-stepping schemes can coexist in harmony with deep neural networks, & offer invaluable intuition in constructing structured predictive algorithms. Moreover, implementation simplicity of the latter greatly favors rapid development & testing of new ideas, potentially opening path for a new era in data-driven scientific computing.

Although a series of promising results was presented, reader may perhaps agree this work creates more questions than it answers. How deep/wide should neural network be? How much data is really needed? Why does algorithm converge to unique values for parameters of differential operators, i.e., why is algorithm not suffering from local optima for parameters of differential operator? Does network suffer from vanishing gradients for deeper architectures & higher order differential operators? Could this be mitigated by using different activation functions? Can we improve on initializing network weights or normalizing data? Are mean square error & sum of squared errors appropriate loss functions? Why are these methods seemingly so robust to noise in data? How can we quantify uncertainty associated with our predictions? Throughout this work, have attempted to answer some of these questions, but have observed: specific settings that yielded impressive results for 1 equation could fail for another. Admittedly, more work is needed collectively to set foundations in this field.

In a broader context, & along way of seeking answers to those questions, believe: this work advocates a fruitful synergy between ML & classical computational physics that has potential to enrich both fields & lead to high-impact developments.

- Appendix A. Data-driven solutions of PDEs.
- Appendix B. Data-driven discovery of PDEs.

5 Neural Network

Resources – Tài nguyên.

1. Amal Alphonse, Michael Hintermüller, Alexander Kister, Chin Hang Lun. *A neural network approach to learning solutions of a class of elliptic variational inequalities.*

   Abstract. Develop a weak adversarial approach to solving obstacle problems using neural networks. By employing (generalized) regularized gap functions & their properties, rewrite obstacle problem (which is an elliptic variational inequality) as a minmax problem, providing a natural formulation amenable to learning. Our approach, in contrast to much of literature, does not require elliptic operator to be symmetric. Provide an error analysis for suitable discretizations of continuous problem, estimating in particular approximation & statistical errors. Parametrizing solution & test function as neural networks, apply a modified gradient descent ascent algorithm to treat problem & conclude paper with various examples & experiments. Our solution algorithm is in particular able to easily handle obstacle problem that feature biactivity (or lack of strict complementarity), a situation that poses difficulty for traditional numerical methods.

   - 1. Introduction. Use neural networks to find solutions of variational inequalities (VIs) of type:

   $$\text{find } u \in K : \langle Au - f, u - v\rangle_{V^*,V} \leq 0, \ \forall v \in K, \tag{169}$$

   where $V \coloneqq H^1(\Omega)$: usual Sobolev space on a bounded Lipschitz domain $\Omega \subset \mathbb{R}^n$, $\langle\cdot,\cdot\rangle_{V^*,V}$ denotes duality pairing between V & its topological dual $V^\star$, constraint se $K \coloneqq \{u \in H^1(\Omega) | u \geq \psi \text{ in } \Omega,\ u = h \text{ on } \partial\Omega\}$, $h \in H^{1/2}(\partial\Omega)$: given boundary data, $\psi \in H^1(\Omega)$: a given obstacle that satisfies $\psi \leq h$ on $\partial\Omega$, & $f \in L^2(\Omega)$: a given source term. Operator $A : K \subset V \to V^*$ appearing in (169) is assumed to be Lipschitz & coercive, i.e., there exist constant $C_a, C_b > 0$ s.t. (3)

   $$\|Au - Av\|_{V^*} \leq C_b \|u - v\|_V, \ \forall u, v \in K, \tag{170}$$
   $$\langle Au - Av, u - v\rangle_{V^*,V} \geq C_a \|u - v\|_V^2, \ \forall u, v \in K, \tag{171}$$

   & for simplicity, focus attention on linear differential operators of form

   $$\langle Au, u - v\rangle_{V^*,V} = \int_\Omega \nabla u \cdot \nabla(u - v) + \sum_{i=1}^n b_i \partial_{x_i} u(u - v) + ku(u - v), \tag{172}$$

   where $k, b_i \geq 0, i = 1, \ldots, n$, are constants (which of course have to be s.t. (3) is satisfied), $\partial_{x_i} u$: weak partial derivative of u w.r.t. ith coordinate & ∇u: weak gradient of u. In operator form, have $A = -\Delta + \sum_{i=1}^n b_i \partial_{x_i} + k\mathrm{Id}$ with Δ representing weak Laplacian & Id: identity map. Setting $b_{=}0$ for $i = 1, \ldots, n, k = 0, h \equiv 0$: prototypical example of an elliptic VI & is commonly referred to as obstacle problem [39].

   Variational inequalities of type (169) have numerous applications in diverse scientific areas; mention in particular contact mechanics, processes in biological cells, ecology, fluid flow, & finance, see e.g. [52, 39, 55]. They are also fundamental objects of study in applied analysis due to their interesting structure. Indeed, VIs are examples of free boundary problems. Obstacle problems sometimes are stated in form (5)

   $$0 \leq (Au - f) \perp (u - \psi) \geq 0 \text{ a.e. on } \Omega, \tag{173}$$
   $$u = h \text{ a.e. on } \partial\Omega, \tag{174}$$

where $a \perp b$ stands for $ab = 0$. This formulation $\Leftrightarrow$ (169) under sufficient regularity (see Prop. 2.2). Classical methods for solving obstacle problems or VIs include projection methods, multilevel & multigrid methods [40, 30], primal dual active set strategies & path following schemes [27], semismooth Newton schemes [25, 33], shape & topological sensitivity based methods [28, 29], level set methods & discontinuous Galerkin schemes [58]; see also [20, 9, 37, 19] & references therein.

Aim: to formulate, analyze, & implement a deep network approach to compute solutions of obstacle problems like (169) or (5). More specifically, rephrase VI as minmax optimization problem involving minimization over feasible set & maximization over all feasible test functions, both of which are parametrized by neural networks, & use a modified gradient descent ascent scheme to numerically solve for solution. Our motivation stems from fact: **neural networks can efficiently represent nonlinear, nonsmooth functions, & have added advantage of not being intrinsically reliant on a mesh**[7]: they provide a naturally global & meshless representation of solution, offering an advantage over other methods e.g. finite elements. Furthermore, they are able to handle complicated geometries & high-dimensional problems without great cost. Our work can also be considered as a 1st step in studying more complicated problems involving e.g. operator learning.

Related papers in the literature addressing solving elliptic variational inequalities or hemivariational inequalities via neural networks include [13, 64, 53, 31, 7]. A typical path taken by many works entails rewriting (169) as a minimization problem. Indeed, if operator A generates a bilinear form which is symmetric, then, as explained, e.g., in [52, §4:3, Remark 3.5, p. 97], VI (169) $\Leftrightarrow$ to minimization problem $\min_{u\in K} \frac{1}{2}\langle Au, u\rangle - \langle f, u\rangle$. This formulation gives rise to a natural loss function that can be tackled via neural networks, as done in [13, 31, 64]. If A is nonsymmetric, this equivalence is not available & one cannot in general pose an associated minimization problem. However, (169) $\Leftrightarrow$ minmax problem

$$\min_{u\in K}\max_{v\in K}\langle Au - f, u - v\rangle - \frac{1}{2\gamma}\|u - v\|_V^2 \tag{175}$$

for a given parameter $\gamma > 0$, regardless of whether A is symmetric or not. This problem formulation appears very natural since it resembles notion of weak formulations in PDEs, which are well understood.

In order to approximate (175), express u & test function v by deep neural networks. This technique falls into class of weak adversarial network (WAN) problems in spirit of [62]; maximization for test function acts as an adversarial approaches for PDEs & related theory. Moreover, our approach is related to Physics Informed Neural Networks (PINNs); see, e.g., [41, 50]. More specifically it can be viewed as a weak PINNs-type approach with a hard constrained boundary condition.

In this work, provide a theoretical justification for minmax problem, a discretized formulation of problem amenable to computation, an error analysis as well as comprehensive numerical algorithms. A special highlight of our work: can also tackle non-symmetric problems, e.g., A given as in (4) with $b_i \neq 0$ for at least 1 $i = 1, \ldots, n$.

- 2. Analysis of continuous problem. Some theoretical results concerning VI (169).
  - 2.1. Basic properties & saddle points. Address existence & uniqueness for (169). Since A is coercive & Lipschitz on $H^1(\Omega)$ & K is nonempty[8] closed & convex, well posedness follows from classical Lions–Stampacchia theorem [52, §4:3, Thm. 3.1] (see also [52, §4:2, Thm. 2.3] for the case where $A = -\Delta$).

    Proposition 10 (H^2-regularity)**.** *Let $A := -\Delta + \sum_{i=1}^n b_i\partial_{x_i} + c\mathrm{Id}$ with coefficients $b_i, c \in L^\infty(\Omega), c \geq c_0 \geq 0$ for a constant c_0, s.t. coercivity condition (3b) is satisfied, $f \in L^2(\Omega), h \in H^{3/2}(\partial\Omega), \psi \in H^2(\Omega)$ with $\psi|_{\partial\Omega} \leq h$, & let Ω be convex or $C^{1,1}$. Then solution of (169) has regularity $u \in H^2(\Omega)$. Furthermore, we have a priori estimate $\|u\|_{H^2(\Omega)} \leq C^*$, where C^* is a constant (that depends in particular on f, ψ, h).*
  - 2.2. Minma approach via regularized gap function.
  - 2.3. Relaxations of problem.
- 3. Neural network approach. Wish to compute (approximate) solutions of VI (169) using neural networks. Architecture we use is essentially residual neural network considered in [16] consisting of usual affine transformations & activations combined with skip connections, inspired by original work [22]. Residual networks or ResNets have been empirically observed to be better at training deep networks & they avoid vanishing gradient problem, see e.g. [23, 60] for some analysis.

  Describe this special ResNet architecture precisely. Let $\mathfrak{b}, \mathfrak{w} \in \mathbb{N}$ be given positive integers (representing depth & width of network resp.).

  TROUBLE IN COMPLICATED NOTATIONS!

6 Recurrent Neural Network

Resources – Tài nguyên.

1. Zachary C. Lipton, John Berkowitz, Charles Elkan. *A Critical Review of Recurrent Neural Networks for Sequence Learning.* [3525 citations]

   Abstract. Countless learning tasks require dealing with sequential data. Image captioning, speech synthesis, & music generation all require: a model produce outputs that are sequences. In other domains, e.g. time series prediction, video analysis, & musical information retrieval, a model must learn from inputs that are sequences. Interactive tasks, e.g. translating natural

[7] có thêm lợi thế là không phụ thuộc hoàn toàn vào lưới.

[8] By properties of trace operator [59, Theorem 8.8, Chapter 1], there exists a function $w \in H^1(\Omega)$ with $w|_{\partial\Omega} = h$, & function $\max\{w, \psi\} \in H^1(\Omega)$ belongs to K.

language, engaging in dialogue, & controlling a robot, often demand both capabilities. Recurrent neural networks (RNNs) are connectionist models that capture dynamics of sequences via cycles in network of nodes. Unlike standard feedforward neural networks, recurrent networks retain a state that can represent information from an arbitrarily long context window. Although recurrent neural networks have traditionally been difficult to train, & often contain millions of parameters, recent advances in network architectures, optimization techniques, & parallel computation have enabled successful large-scale learning with them. In recent years, systems based on long short-term memory (LSTM) & bidirectional (BRNN) architectures have demonstrated ground-breaking performance on tasks as varied as image captioning, language translation, & handwriting recognition. In this survey, review & synthesize research that over past 3 decades 1st yielded & then made practical these powerful learning models. When appropriate, reconcile conflicting notation & nomenclature. When appropriate, reconcile conflicting notation & nomenclature. Goal: to provide a self-contained explication of state of art together with a historical perspective & refs to primary research.

- 1. Introduction. Neural networks are powerful learning models that achieve state-of-art results in a wide range of supervised & unsupervised machine learning tasks. They are suited especially well for machine perception tasks, where raw underlying features are not individually interpretable. This success is attributed to their ability to learn hierarchical representations, unlike traditional methods that rely upon hand-engineered features [Farabet et al., 2013]. Over past several years, storage has become more affordable, datasets have grown far larger, & field of parallel computing has advanced considerably. In setting of large datasets, simple linear models tend to under-fit, & often under-utilize computing resources. Deep learning methods, in particular those based on deep belief networks (DNNs), which are greedily built by stacking restricted Boltzmann machines, & convolutional neural networks, which exploit local dependency of visual information, have demonstrated record-setting results on many important applications.

  However, despite their power, standard neural networks have limitations. Most notably, they rely on assumption of independence among training & test examples. After each example (data point) is processed, entire state of network is lost. If each example is generated independently, this presents no problem. But if data points are related in time or space, this is unacceptable. Frames from video, snippets of audio, & words pulled from sentences, represent settings where independence assumption fails. Additionally, standard networks generally rely on examples being vectors of fixed length. Thus desirable to extend these powerful learning tools to model data with temporal or sequential structure & varying length inputs & outputs, especially in many domains where neural networks are already state of art. Recurrent neural networks (RNNs) are connectionist models with ability to selectively pass information across sequence steps, while processing sequential data 1 element at a time. Thus they can model input &/or output consisting of sequences of elements that are not independent. Further, recurrent neural networks can simultaneously model sequential & time dependencies on multiple scales.

  In following subsections, explain fundamental reasons why recurrent netral networks are worth investigating. To be clear, motivated by a desire to achieve empirical results. This motivation warrants clarification because recurrent networks have roots in both cognitive modeling & supervised machine learning. Owing to this difference of perspectives, many published papers have different aims & priorities. In many foundational papers, generally published in cognitive science & computational neuroscience journals, e.g. [Hopfield, 1982, Jordan, 1986, Elman, 1990], biologically plausible mechanisms are emphasized. In [Schuster & Paliwal, 1997, Socher et al., 2014, Karpathy & Fei-Fei, 2014], biological inspiration is downplayed ion favor of achieving empirical results on important tasks & datasets. This review is motivated by practical results rather than biological plausibility, but where appropriate, draw connections to relevant concepts in neuroscience. Given empirical aim, now address 3 significant questions that one might reasonably want answered before reading further.

  - 1.1. Why model sequentiality explicitly? In light of practical success & economic value of sequence-agnostic models, this is a fair question. Support vector machines, logistic regression, & feedforward networks have proved immensely useful without explicitly modeling time. Arguably, precisely assumption of independence that has led to much recent progress in machine learning. Further, many models implicitly capture time by concatenating each input with some number of its immediate predecessors & successors, presenting machine learning model with a sliding window of context about each point of interest. This approach has been used with deep belief nets for speech modeling by Maas et al. [2012].
    Unfortunately, despite usefulness of independence assumption, it precludes modeling long-range dependencies. E.g., a model trained using a finite-length context window of length 5 could never be trained to answer simple question, *"what was data point seen 6 time steps ago?"* For a practical application e.g. call center automation, such a limited system might learn to route calls, but could never participate with complete success in an extended dialogue. Since earliest conception of artificial intelligence, researchers have sought to build systems that interact with humans in time. In Alan Turing's groundbreaking paper *Computing Machinery & Intelligence*, he proposes an "imitation game" which judges a machine's intelligence by its ability to convincingly engage in dialogue [Turing, 1950]. Besides dialogue systems, modern interactive systems of economic importance include self-driving cars & robotic surgery, among others. Without an explicit model of sequentiality or time, it seems unlikely that any combination of classifiers or regressors can be cobbled together to provide this functionality.
  - 1.2. Why not use Markov models? Recurrent neural networks are not only models capable of representing time dependencies. Markov chains, which model transitions between states in an observed sequence, were 1st described by mathematician Andrey Markov in 1906. Hidden Markov models (HMMs), which model an observed sequence as probabilistically dependent upon a sequence of unobserved states, were described in 1950s & have been widely studied since 1960s Stratonovich, 1960]. However, traditional Markov model approaches are limited because their states must be drawn from a modestly sized discrete state space S. Dynamic programming algorithm that is used to perform efficient inference with hidden Markov models scales in time $O(|S|^2)$ [Viterbi, 1967]. Further, transition table capturing probability of moving

between any 2 time-adjacent states is of size $|S|^2$. Thus, standard operations become infeasible with an HMM when set of possible hidden states grows large. Further, each hidden state can depend only on immediately previous state. While possible to extend a Markov model to account for a larger context window by creating a new state space equal to cross product of possible states at each time in window, this procedure grows state space exponentially with size of window, rendering Markov models computationally impractical for modeling long-range dependencies [Graves et al., 2014].
Given limitations of Markov models, ought to explain why reasonable that connectionist models, i.e., artificial neural networks, should fare better. 1st, recurrent neural networks can capture long-range time dependencies, overcoming chief limitation of Markov models. This point requires a careful explanation. As in Markov models, any state in a traditional RNN depends only on current input as well as on state of network at previous time step.[9] However, hidden state at any time step can contain information from a nearly arbitrarily long context window. This is possible because number of distinct states that can be represented in a hidden layer of nodes grows exponentially with number of nodes in layer. Even if each node took only binary values, network could represent 2^N states where N is number of nodes in hidden layer. When value of each node is a real number, a network can represent even more distinct states. While potential expressive power of a network grows exponentially with number of nodes, complexity of both inference & training grows at most quadratically.

- 1.3. Are RNNs too expensive? Finite-sized RNNs with nonlinear activations are a rich family of models, capable of nearly arbitrary computation. A well-known result: a finite-sized recurrent neural network with sigmoidal activation functions can simulate a universal Turing machine [Siegelmann & Sontag, 1991]. Capability of RNNs to perform arbitrary computation demonstrates their expressive power, but one could argue: C programming language is equally capable of expressing arbitrary programs. & yet there are no papers claiming: invention of C represents a panacea (thuốc chữa bách bệnh) for ML. A fundamental reason: there is no simple way of efficiently exploring space of C programs. In particular, there is no general way to calculate gradient of an arbitrary C program to minimize a chosen loss function. Moreover, given any finite dataset, there exist countless programs which overfit dataset, generating desired training output but failing to generalize to test examples.
  Why then should RNNs suffer less from similar problem? 1st, given any fixed architecture (set of nodes, edges, & activation functions), recurrent neural networks with this architecture are differentiable end to end. Derivative of loss function can be calculated w.r.t. each of parameters (weights) in model. Thus, RNNs are amenable to gradient-based training. 2nd, while Turing-completeness of RNNs is an impressive property, given a fixed-size RNN with a specific architecture, not actually possible to reproduce any arbitrary program. Further, unlike a program composed in C, a recurrent neural network can be regularized via standard techniques that help prevent overfitting, e.g. weight decay, dropout, & limiting degrees of freedom.
- 1.4. Comparison to prior literature. Literature on recurrent neural networks can seem impenetrable to uninitiated. Shorter papers assume familiarity with a large body of background literature, while diagrams are frequently underspecified, failing to indicate which edges span time steps & which do not. Jargon abounds, & notation is inconsistent across papers or overloaded within 1 paper. Readers are frequently in unenviable position of having to synthesize conflicting information across many papers in order to understand just one. E.g., in many papers subscripts index both nodes & time steps. In others, h simultaneously stands for a link function & a layer of hidden nodes. Variable t simultaneously stands for both time indices & targets, sometimes in same equation. Many excellent research papers have appeared recently, but clear reviews of recurrent neural network literature are rare.
  Among most useful resources are a recent book on supervised sequence labeling with recurrent neural network [Graves, 2012], & an earlier doctoral thesis [Gers, 2001]. A recent survey covers recurrent neural nets for language modeling [De Mulder et al., 2015]. Various authors focus on specific technical aspects; e.g. Pearlmutter [1995] surveys gradient calculations in continuous time recurrent neural networks. Aim: to provide a readable, intuitive, consistently notated, & reasonably comprehensive but selective survey of research on recurrent neural networks for learning with sequences. Emphasize architectures, algorithms, & results, but aim also to distill intuitions that have guided this largely heuristic & empirical field. In addition to concrete modeling details, offer qualitative arguments, a historical perspective, & comparisons to alternate methodologies where appropriate.

- 2. Background. This sect introduces formal notation & provides a brief background on neural networks in general.
  - 2.1. Sequences. Input to an RNN is a sequence, &/or its target is a sequence.
    An input sequence can be denoted $(\mathbf{x}^{(1)}\mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T)})$ where each data point $\mathbf{x}^{(t)}$: a real-valued vector. Similarly, a target sequence can be denoted $(\mathbf{y}^{(1)}\mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(T)})$. A training set typically is a set of examples where each example is an (input sequence, target sequence) pair, although commonly either input or output may be a single data point. Sequences may be of finite or countably infinite length. When they are finite, maximum time index of sequence is called T. RNNs are not limited to time-based sequences. They have been used successfully on non-temporal sequence data, including genetic data Baldi & Pollastri, 2003]. However, in many important applications of RNNs, sequences have an explicit or implicit temporal aspect. While often refer to time in this survey, methods described here are applicable to non-temporal as well as to temporal tasks.
    Using temporal terminology, an input sequence consists of data points $\mathbf{x}^{(t)}$ that arrive in a discrete sequence of *time steps* indexed by t. A target sequence consists of data points $\mathbf{y}^{(t)}$. Use superscripts with parentheses for time, & not subscripts, to prevent confusion between sequence steps & indices of nodes in a network. When a model produces predicted data points, these are labeled $\hat{\mathbf{y}}^{(t)}$.

[9] While traditional RNNs only model dependence of current state on previous state, bidirectional recurrent neural networks (BRNNs) [Schuster & Paliwal, 1997] extend RNNs to model dependence on both past states & future states.

Time-indexed data points may be equally spaced samples from a continuous real-world process. Examples include still images that comprise frames of videos or discrete amplitudes sampled at fixed intervals that comprise audio recordings. Time steps may also be ordinal, with no exact correspondence to durations. In fact, RNNs are frequently applied to domains where sequences have a defined order but no explicit notion of time. This is case with natural language. In word sequence "John Coltrane plays saxophone", $\mathbf{x}^{(1)}$ = John, $\mathbf{x}^{(2)}$ = Coltrane, etc.

- 2.2. Neural networks. Neural networks are biologically inspired models of computation. Generally, a neural network consists of a set of *artificial neurons*, commonly referred to as *nodes* or *units*, & a set of directed edges between them, which intuitively represent *synapses* in a biological neural network. Associated with each neuron j is an activation function $l_j(\cdot)$, which is sometimes called a *link function*. Use notation l_j & not h_j, unlike some other papers, to distinguish activation function from values of hidden nodes in a network, which, as a vector, is commonly notated $\mathbf{h}$ in literature.
Associated with each edge from node j' to j is a weight $w_{jj'}$. Following convention adopted in several foundational papers. [Hochreiter & Schmidhuber, 1997, Gers et al., 2000, Gers, 2001, Sutskever et al., 2011], index neurons with j & j', & $w_{jj'}$ denotes "to-form" weight corresponding to directed edge to node j from node j'. Important to note: in many refs indices are flipped & $w_{j'j} \neq w_{jj'}$ denotes "from-to" weight on directed edge from node j' to node j, as in lecture notes by Elkan [2015] & in Wikipedia/backpropagation.
Value v_j of each neuron j is calculated by applying its activation function to a weighted sum of values of its input nodes (Fig. 1: An artificial neuron computes a nonlinear function of a weighted sum of its inputs):

$$v_j = l_j\left(\sum_{j'} w_{jj'} \cdot v_{j'}\right). \tag{176}$$

For convenience, term weighted sum inside parentheses *incoming activation* & notate it as a_j. Represent this computation in diagrams by depicting neurons as circles & edges as arrows connecting them. When appropriate, indicate exact activation function with a symbol, e.g., σ for sigmoid.
Common choices for activation function include sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$ & tanh function $\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. The latter was become common in feedforward neural nets & was applied to recurrent nets by Sutskever et al. [2011]. Another activation function which has become prominent in deep learning research is rectified linear unit (ReLU) whose formula is $l_j(z) = \max\{0, z\}$. This type of unit has been demonstrated to improve performance of many deep neural networks [Nair & Hinton, 2010, Maas et al., 2012, Zeiler et al., 2013] on tasks as varied as speech processing & object recognition, & has been used in recurrent neural networks by Bengio et al. [2013].
Activation function at output nodes depends upon task. For multiclass classification with K alternative classes, apply a softmax nonlinearity in an output layer of K nodes. Softmax function calculates

$$\hat{y}_k = \frac{e^{a_k}}{\sum_{k'=1}^{K} e^{a_{k'}}}, \ \forall k = 1, \ldots, K. \tag{177}$$

Denominator is a normalizing term consisting of sum of numerators, ensuring: outputs of all nodes sum to 1. For multilabel classification, activation function is simply a pointwise sigmoid, & for regression typically have linear output.

- 2.3. Feedforward networks & backpropagation. With a neural model of computation, one must determine order in which computation should proceed. Should nodes be sampled 1 at a time & updated, or should value of all nodes be calculated at once & then all updates applied simultaneously? Feedforward networks (Fig. 2: A feedforward neural network. An example is presented to network by setting values of blue (bottom) nodes. Values of nodes in each layer are computed successively as a function of prior layers until output is produced at topmost layer.) are a restricted class of networks which deal with this problem by forbidding cycles in directed graph of nodes. Given absence of cycles, all nodes can be arranged into layers, & outputs in each layer can be calculated given outputs from lower layers.
Input $\mathbf{x}$ to a feedforward network is provided by setting values of lowest layer. Each higher layer is then successively computed until output is generated at topmost layer $\hat{\mathbf{y}}$. Feedforward networks are frequently used for supervised learning tasks e.g. classification & regression. Learning is accomplished by iteratively updating each of weights to minimize a loss function, $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y})$, which penalizes distance between output $\hat{\mathbf{y}}$ & target $\mathbf{y}$.
Most successful algorithm for training neural networks is backpropagation, introduced for this purpose by Rumelhart et al. [1985]. Backpropagation uses chain rule to calculate derivative of loss function $\mathcal{L}$ w.r.t. each parameter in network. Weights are then adjusted by gradient descent. Because loss surface is non-convex, there is no assurance that backpropagation will reach a global minimum. Moreover, exact optimization is known to be an NP-hard problem. However, a large body of work on heuristic pre-training & optimization techniques has led to impressive empirical success on many supervised learning tasks. In particular, convolutional neural networks, popularized by Le Cun et al. [1990], are a variant of feedforward neural network that holds records since 2012 in many computer vision tasks e.g. object detection [Krizhevsky et al., 2012].
Nowadays, neural networks are usually trained with stochastic gradient descent (SGD) using mini-batches. With batch size = 1, stochastic gradient update equation is

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} F_i, \tag{178}$$

where η: learning rate, $\nabla_{\mathbf{w}} F_i$: gradient of objective function w.r.t. parameters $\mathbf{w}$ as calculated on a single example (x_i, y_i). Many variants of SGD are used to accelerate learning. Some popular heuristics, e.g. AdaGrad [Duchi et al.,

2011], AdaDelta [Zeiler, 2012], & RMSprop [Tieleman & Hinton, 2012], tune learning rate adaptively for each feature. AdaGrad, arguably most popular, adapts learning rate by caching sum of squared gradients w.r.t. each parameter at each time step. Step size for each feature is multiplied by inverse of square root of this cached value. AdaGrad leads to fast convergence on convex error surfaces, but because cached sum is monotonically increasing, method has a monotonically decreasing learning rate, which may be undesirable on highly non-convex loss surfaces. RMSprop modifies AdaGrad by introducing a decay factor in cache, changing monotonically growing value into a moving average. Momentum methods are another common SGD variant used to train neural networks. These methods add to each update a decaying sum of previous updates. When momentum parameter is tuned well & network is initialized well, momentum methods can train deep nets & recurrent nets competitively with more computationally expensive methods like Hessian-free optimizer of Sutskever et al. [2013].

To calculate gradient in a feedforward neural network, backpropagation proceeds as follows. 1st, an example is propagated forward through network to produce a value v_j at each node & outputs $\hat{\mathbf{y}}$ at topmost layer. Then, a loss function value $\mathcal{L}(\hat{y}_k, y_k)$ is computed at each output node k. Subsequently, for each output node k, calculate

$$\delta_k = \frac{\partial \mathcal{L}(\hat{y}_k, y_k)}{\partial \hat{y}_k} \cdot l'_k(a_k). \tag{179}$$

Given these values δ_k, for each node in immediately prior layer, calculate

$$\delta_j = l'(a_j) \sum_k \delta_k \cdot w_{kj}. \tag{180}$$

This calculation is performed successively for each lower layer to yield δ_j for every node j given δ values for each node connected to j by an outgoing edge. Each value δ_j represents derivative $\partial_{a_j} \mathcal{L}$ of total loss function w.r.t. that node's incoming activation. Given values v_j calculated during forward pass, & values δ_j calculated during backward pass, derivative of loss $\mathcal{L}$ w.r.t. a given parameter $w_{jj'}$ is

$$\partial_{w_{jj'}} \mathcal{L} = \delta_j v'_j. \tag{181}$$

Other methods have been explored for learning weights in a neural network. A number of papers from 1990s [Belew et al., 1990, Gruau et al., 1994] championed idea of learning neural networks with genetic algorithms, with some even claiming: achieving success on real-world problems only by applying many small changes to weights of a network was impossible. Despite subsequent success of backpropagation, interest in genetic algorithms continues. Several recent papers explore genetic algorithms for neural networks, especially as a means of learning architecture of neural networks, a problem not addressed by backpropagation [Bayer et al., 2009, Harp & Samad, 2013]. By *architecture*, mean number of layers, number of nodes in each, connectivity pattern among layers, choice of activation functions, etc.

1 open question in neural network research is how to exploit sparsity in training. In a neural network with sigmoidal or tanh activation functions, nodes in each layer never take value exactly 0. Thus, even if inputs are sparse, nodes at each hidden layer are not. However, rectified linear units (ReLUs) introduce sparsity to hidden layers [Glorot et al., 2011]. In this setting, a promising path may be to store sparsity pattern when computing each layer's values & use it to speed up computation of next layer in network. Some recent work shows: given sparse inputs to a linear model with a standard regularizer, sparsity can be fully exploited even if regularization makes gradient be not sparse [Carpenter, 2008, Langford et al., 2009, Singer & Duchi, 2009, Lipton & Elkan, 2015].

- 3. Recurrent neural networks. Recurrent neural networks are feedforward neural networks augmented by inclusion of edges that span adjacent time steps, introducing a notion of time to model. Like feedforward networks, RNNs may not have cycles among conventional edges. However, edges that connect adjacent time steps, called *recurrent edges*, may form cycles, including cycles of length 1 that are self-connections from a node to itself across time. At time t, nodes with recurrent edges receive input from current data point $\mathbf{x}^{(t)}$ & also from hidden node values $\mathbf{h}^{(t-1)}$ in network's previous state. Output $\hat{\mathbf{y}}^{(t)}$ at each time t is calculated given hidden node values $\mathbf{h}^{(t)}$ at time t. Input $\mathbf{x}^{(t-1)}$ at time $t-1$ can influence output $\hat{\mathbf{y}}^{(t)}$ at time t & later by way of recurrent connections.

  2 equations specify all calculations necessary for computations at each time step on forward pass in a simple recurrent neural network as in Fig. 3: A simple recurrent network. At each time step t, activation is passed along solid edges as in a feedforward network. Dashed edges connect a source node at each time t to a target node at each following time $t+1$.:

$$\mathbf{h}^{(t)} = \sigma(W_{\mathrm{hx}} \mathbf{x}^{(t)} + W_{\mathrm{hh}} \mathbf{h}^{(t-1)} + \mathbf{b}_h), \tag{182}$$

$$\hat{\mathbf{y}}^{(t)} = \mathrm{softmax}(W_{\mathrm{yh}} \mathbf{h}^{(t)} + \mathbf{b}_y). \tag{183}$$

  Here W_{hx}: matrix of conventional weights between input & hidden layer & W_{hh}: matrix of recurrent weights between hidden layer & itself at adjacent time steps. Vectors $\mathbf{b}_h, \mathbf{b}_y$: bias parameters which allow each node to learn an offset.

  Dynamics of network depicted in Fig. 3 across time steps can be visualized by *unfolding* it as in Fig. 4: Recurrent network of Fig. 3 unfolded across time steps. Given this picture, network can be interpreted not as cyclic, but rather as a deep network with 1 layer per time step & shared weights across time steps. Then clear: unfolded network can be trained across many time steps using backpropagation. This algorithm, called *backpropagation through time* (BPTT), was introduced by Werbos [1990]. All recurrent networks in common current use apply it.

- 3.1. Early recurrent network designs. Foundational research on recurrent networks took place in 1980s. In 1982, Hopfield introduced a family of recurrent neural networks that have pattern recognition capabilities Hopfield, 1982]. They are defined by values of weights between nodes & link functions are simple thresholding at 0. In these nets, a pattern is placed in network by setting values of nodes. Network then runs for some time according to its update rules, & eventually another pattern is read out. Hopfield networks are useful for recovering a stored pattern from a corrupted version & are forerunners of Boltzmann machines & auto-encoders.
  An early architecture for supervised learning on sequences was introduced by Jordan [1986]. Such a network (Fig. 5: A recurrent neural network as proposed by Jordan [1986]. Output units are connected to special units that at next time step feed into themselves & into hidden units.) is a feedforward network with a single hidden layer that is extended with special units.[10] Output node values are fed to special units, which then feed these values to hidden nodes at following time step. If output values are actions, special units allow network to remember actions taken at previous time steps. Several modern architectures use a related form of direct transfer from output nodes; Sutskever et al. [2014] translates sentences between natural languages, & when generating a text sequence, word chosen at each time step is fed into network as input at following time step. Additionally, special units in a Jordan network are self-connected. Intuitively, these edges allow sending information across multiple time steps without perturbing output at each intermediate time step.
  Architecture introduced by Elman [1990] is simpler than earlier Jordan architecture. Associated with each unit in hidden layer is a context unit. Each such unit j' takes as input state of corresponding hidden node j at previous time step, along an edge of fixed weight $w_{j'j} = 1$. This value then feeds back into same hidden node j along a standard edge. This architecture is equivalent to a simple RNN in which each hidden node has a single self-connected recurrent edge. Idea of fixed-weight recurrent edges that make hidden nodes self-connected is fundamental in subsequent work on LSTM networks [Hochreiter & Schmidhuber, 1997].
  Elman [1990] trains network using backpropagation & demonstrates: network can learn time dependencies. Paper features 2 sets of experiments. The 1st extends logical operation *eclusive or* (XOR) to time domain by concatenating sequences of 3 tokens. For each 3-token segment, e.g., "011", 1st 2 tokens ("01") are chosen randomly & 3rd ("1') is set by performing xor on 1st 2. Random guessing should achieve accuracy of 50%. A perfect system should perform same as random for 1st 2 tokens, but guess 3rd token perfectly, achieving accuracy of 66.7%. Simple network of Elman [1990] does in fact approach this maximum achievable score.
- 3.2. Training recurrent network designs. Learning with recurrent networks has long been considered to be difficult. Even for standard feedforward networks, optimization task is NP-complete Blum & Rivest [1993]. But learning with recurrent networks can be especially challenging due to difficulty of learning long-range dependencies, as described by Bengio et al. [1994] & expanded upon by Hochreiter et al. [2001]. Problems of *vanishing* & *exploding* gradients occur when backpropagating errors across many time steps. As a toy example, consider a network with a single input node, a single output node, & a single recurrent hidden note (Fig. 7: A simple recurrent net with 1 input unit, 1 output unit, & 1 recurrent hidden unit.). Now consider an input passed to network at time τ & an error calculated at time t, assuming input of 0 in intervening time steps. Typing of weights across time steps means: recurrent edge at hidden node j always has same weight. Therefore, contribution of input at time τ to output at time t will either explode or approach 0, exponentially fast, as $t - \tau$ grows large. Hence derivative of error w.r.t. input will either explode or vanish.
  Fig. 6: A recurrent neural network as described by Elman [1990]. Hidden units are connected to context units, which feed back into hidden units at next time step.
  Which of 2 phenomena occurs depends on whether weight of recurrent edge $|w_{jj}| > 1$ or $|w_{jj}| < 1$ & on activation function in hidden node (Fig. 8: A visualization of vanishing gradient problem, using network depicted in Fig. 7, adapted from Graves [2012]. If weight along recurrent edge < 1, contribution of input at 1st time step to output at final time step will decrease exponentially fast as a function of length of time interval in between.). Given a sigmoid activation function, vanishing gradient problem is more pressing, but with a rectified linear unit $\max\{0, x\}$, easier to imagine exploding gradient. Pascanu et al. [2012] give a thorough mathematical treatment of vanishing & exploding gradient problems, characterizing exact conditions under which these problems may occur. Given these conditions, they suggest an approach to training via a regularization term that forces weights to values where gradient neither vanishes nor explodes.
  Truncated backpropagation through time (TBPTT) is 1 solution to exploding gradient problem for continuously running networks [Williams & Zipser, 1989]. With TBPTT, some maximum number of time steps is set along which error can be propagated. While TBPTT with a small cutoff can be used to alleviate exploding gradient problem, it requires: one sacrifice ability to learn long-range dependencies. LSTM architecture described below uses carefully designed nodes with recurrent edges with fixed unit weight as a solution to vanishing gradient problem.
  Issue of local optima is an obstacle to effective training that cannot be dealt with simply by modifying network architecture. Optimizing even a single hidden-layer feedforward network is an NP-complete problem [Blum & Rivest, 1993]. However, recent empirical & theoretical studies suggest: in practice, issue may not be as important as once thought. Dauphin et al. [2014] show: while many critical points exist on error surfaces of large neural networks, ratio of saddle points to true local minima increases exponentially with size of network, & algorithms can be designed to escape from saddle points.
  Overall, along with improved architectures explained below, fast implementations & better gradient-following heuristics have rendered RNN training feasible. Implementations of forward & backward propagation using GPUs, e.g. Theano [Bergstra et al., 2010] & Torch [Collobert et al., 2011] packages, have made it straightforward to implement fast training

[10] [Jordan 1986] calls special units "state units" while [Elman 1990] calls a corresponding structure "context units." In this paper simplify terminology by using only "context units".

algorithms. In 1996, prior to introduction of LSTM, attempts to train recurrent nets to bridge long time gaps were shown to perform no better than random guessing [Hochreiter & Schmidhuber, 1996]. However, RNNs are now frequently trained successfully.

For some tasks, freely available software can be run on a single GPU & produce compelling results in hours [Karpathy, 2015]. Martens & Sutskever [2011] reported success training recurrent neural networks with a Hessian-free truncated Newton approach, & applied method to a network which learns to generate text 1 character at a time in [Sutskever et al., 2011]. In paper that describes abundance of saddle points on error surfaces of neural networks [Dauphin et al., 2014], authors present a saddle-free version of Newton's method. Unlike Newton's method, which is attracted to critical points, including saddle points, this variant is specially designed to escape from them. Experimental results include a demonstration of improved performance on recurrent networks. Newton's method requires computing Hessian, which is prohibitively expensive for large networks, scaling quadratically with number of parameters. While their algorithm only approximates Hessian, still computationally expensive compared to SGD. Thus authors describe a hybrid approach in which saddle-free Newton method is applied only in places where SGD appears to be stuck.

- 4. Modern RNN architectures. Most successful RNN architectures for sequence learning stem from 2 papers published in 1997. 1st paper, *Long Short-Term Memory* by Hochreiter & Schmidhuber [1997], introduces *memory cell*, a unit of computation that replaces traditional nodes in hidden layer of a network. With these memory cells, networks are able to overcome difficulties with training encountered by earlier recurrent networks. 2nd paper, *Bidirectional Recurrent Neural Networks* by Schuster & Paliwal [1997], introduces an architecture in which information from both future & past are used to determine output at any point in sequence. This is in contrast to previous networks, in which only past input can affect output, & has been used successfully for sequence labeling tasks in natural language processing, among others. Fortunately, 2 innovations are not mutually exclusive, & have been successfully combined for phoneme classification [Graves & Schmidhuber, 2005] & handwriting recognition [Graves et al., 2009]. In this section, explain LSTM & BRNN & describe *neural Turing machine* (NTM), which extends RNNs with an addressable external memory [Graves et al., 2014].
  - 4.1. Long short-term memory (LSTM). Hochreiter & Schmidhuber [1997] introduced LSTM model primarily in order to overcome problem of vanishing gradients. This model resembles a standard recurrent neural network with a hidden layer, but each ordinary node (Fig. 1) in hidden layer is replaced by a *memory cell* (Fig. 9: 1 LSTM memory cell as proposed by Hochreiter & Schmidhuber [1997]. Self-connected node is internal state s. Diagonal line indicates: it is linear, i.e., identity link function is applied. Blue dashed line is recurrent edge, which has fixed unit weight. Nodes marked Π output product of their inputs. All edges into & from Π nodes also have fixed unit weight.). Each memory cell contains a node with a self-connected recurrent edge of fixed weight 1, ensuring: gradient can pass across many time steps without vanishing or exploding. To distinguish refs to a memory cell & not an ordinary node, use subscript c.

    Term "long short-term memory" comes from following intuition. Simple recurrent neural networks have *long-term memory* in form of weights. Weights change slowly during training, encoding general knowledge about data. They also have *short-term memory* in form of ephemeral activations, which pass from each node to successive nodes. LSTM model introduces an intermediate type of storage via memory cell. A memory cell is a composite unit, built from simpler nodes in a specific connectivity pattern, with novel inclusion of multiplicative nodes, represented in diagrams by letter Π. All elements of LSTM cell are enumerated & described below. Note: when use vector notation, we are referring to values of nodes in an entire layer of cells. E.g., $\mathbf{s}$: a vector containing value of s_c at each memory cell c in a layer. When subscript c is used, it is to index an individual memory cell.
    - *Input node*: This unit, labeled g_c, is a node that takes activation in standard way from input layer $\mathbf{x}^{(t)}$ at current time step & (along recurrent edges) from hidden layer at previous time step $\mathbf{h}^{(t-1)}$. Typically, summed weighted input is run through a tanh activation function, although in original LSTM paper, activation function is a *sigmoid*.
    - *Input gate*: Gates are a distinctive feature of LSTM approach. A gate is a sigmoidal unit that, like input node, takes activation from current data point $\mathbf{x}^{(t)}$ as well as from hidden layer at previous time step. A gate is so-called because its value is used to multiply value of another node. It is a *gate* in sense that if its value is 0, then flow from other node is cut off. If value of gate is 1, all flow is passed through. Value of *input gate* i_c multiples value of *input node*.
    - *Internal state*: At heart of each memory cell is a node s_c with linear activation, which is referred to in original paper as "internal state" of cell. Internal state s_c has a self-connected recurrent edge with fixed unit weight. Because this edge spans adjacent time steps with constant weight, error can flow across time steps without vanishing or exploding. This edge is often called *constant error carousel*. In vector notation, update for internal state is $\mathbf{s}^{(t)} = \mathbf{g}^{(t)} \odot \mathbf{i}^{(t)} + \mathbf{s}^{(t-1)}$ where $\odot$ is pointwise multiplication.
    - *Forget gate*: These gates f_c were introduced by Gers et al. [2000]. They provide a method by which network can learn to flush contents of internal state. This is especially useful in continuously running networks. With forget gates, equation to calculate internal state on forward pass is:

$$\mathbf{s}^{(t)} = \mathbf{g}^{(t)} \odot \mathbf{i}^{(t)} + \mathbf{f}^{(t)} \odot \mathbf{s}^{(t-1)}. \tag{184}$$

    - *Output gate*: Value v_c ultimately produced by a memory cell is value of internal state s_c multiplied by value of *output gate* o_c. Customary: internal state 1st be run through a tanh activation function, as this gives output of each cell same dynamic range as an ordinary tanh hidden unit. However, in other neural network research, rectified linear units, which have a greater dynamic range, are easier to train. Thus it seems plausible: nonlinear function on internal state might be omitted.

    In original paper & in most subsequent work, input node is labeled g. Adhere to this convention but note: may be confusing as g does not stand for *gate*. In original paper, gates are called $y_{\text{in}}, y_{\text{out}}$ but this is confusing because y generally

stands for output in ML literature. Seeking comprehensibility, break with this convention & use i, f, o to refer to input, forget, & output gates resp., as in Sutskever et al. [2014].
Since original LSTM was introduced, several variations have been proposed. Forget gates described above were proposed in 2000 & were not part of original LSTM design. However, they have proven effective & are standard in most modern implementations. That same year, Gers & Schmidhuber [2000] proposed peephole connections that pass from internal state directly to input & output gates of that same node without 1st having to be modulated by output gate. They report: these connections improve performance on timing tasks where network must learn to measure precise intervals between events. Intuition of peephole connection can be captured by following example. Consider a network which must learn to count objects & emit some desired output when n objects have been seen. Network might learn to let some fixed amount of activation into internal state after each object is seen. This activation is trapped in internal state s_c by constant error carousel, & is incremented iteratively each time another object is seen. When nth object is seen, network needs to know to let out content from internal state so that it can affect output. To accomplish this, output gate o_c must know content of internal state s_c. Thus s_c should be an input to o_c.
Put formally, computation in LSTM model proceeds according to following calculations, which are performed at each time step. These equations give full algorithm for a modern LSTM with forget gates: ***

2. [MC01]. Danilo Mandic, Jonathon Chambers. *Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures & Stability.*

Preface. New technologies in engineering, physics, & biomedicine are creating problems in which nonstationarity, nonlinearity, uncertainty, & complexity play a major role. Solutions to many of these problems require the use of nonlinear processors, among which neural networks are 1 of the most powerful. Neural networks are appealing because they learn by example & are strongly supported by statistical & optimization theories. They not only complement conventional signal processing techniques, but also emerge as a convenient alternative to expand signal processing horizons.

The use of recurrent neural networks as identifiers & predictors in nonlinear dynamical systems has increased significantly. They can exhibit a wide range of dynamics, due to feedback, & are also tractable nonlinear maps.

Neural network models are considered as massively interconnected nonlinear adaptive filters. The emphasis is on dynamics, stability, & spatio-temporal behavior of recurrent architectures & algorithms for prediction. However, wherever possible the material has been presented starting from feedforward networks & building up to the recurrent case.

Objective: to offer an accessible self-contained research monograph which can also be used as a graduate text. The material presented in the book is of interest to a wide population of researchers working in engineering, computing, science, finance, & biosciences. So that the topics are self-contained, assume familiarity with basic concepts of analysis & linear algebra. The material presented in Chaps. 1–6 can serve as an advanced text for courses on neural adaptive systems. The book encompasses traditional & advanced learning algorithms & architectures for recurrent neural networks. Although we emphasis the problem of time series prediction, the results are applicable to a wide range of problems, including other signal processing configurations e.g. system identification, noise cancellation, & inverse system modeling. Harmonize concepts of learning algorithms, embedded systems, representation of memory, neural network architectures & causal-noncausal dealing with time. A special emphasis is given to stability of algorithms – a key issue in real-time applications of adaptive systems.

- Introduction. Artificial neural network (ANN) models have been extensively studied with the aim of achieving human-like performance, especially in the field of pattern recognition. These networks are composed of a number of nonlinear computational elements which operate in parallel & are arranged in a manner reminiscent of biological neural interconnections. ANNs are known by many names e.g. connectionist models, parallel distributed processing models & neurmorphic systems (Lippmann 1987). The origin of connectionist ideas can be traced back to the Greek philosopher, Aristotle, & his ideas of mental associations. He proposed some of the basic concepts e.g. memory is composed of simple elements connected to each other via a number of different mechanisms (Medler 1998).

  While early work in ANNs used anthropomorphic arguments to introduce the methods & models used, today neural networks used in engineering are related to algorithms & computation & do not question how brains might work (Hunt et al. 1992). E.g., recurrent neural networks have been attractive to physicists due to their isomorphism to spin glass systems (Ermentrout 1998). The following properties of neural networks make them important in signal processing (Hunt et al. 1992): they are nonlinear systems; they enable parallel distributed processing; they can be implemented in VLSI technology; they provide learning, adaptation & data fusion of both qualitative (symbolic data from artificial intelligence) & quantitative (from engineering) data; they realize multivariable systems.

  The area of neural networks is nowadays considered from 2 main perspectives. 1st perspective: cognitive science, which is an interdisciplinary study of the mind. 2nd perspective is connectionism, which is a theory of information processing (Medler 1998). The neural networks in this work are approached from an engineering perspective, i.e., to make networks efficient in terms of topology, learning algorithms, ability to approximate functions & capture dynamics of time-varying systems. From the perspective of connection patterns, neural networks can be grouped into 2 categories: feedforward networks, in which graphs have no loops, & recurrent networks, where loops occur because of feedback connections. Feedforward networks are static, i.e., a given input can produce only 1 set of outputs, & hence carry no memory. In contrast, recurrent network architectures enable the information to be temporally memorized in the networks (Kung & Hwang 1998). Based on training by example, with strong support of statistical & optimization theories (Cichocki & Unbehauen 1993; Zhang & Constantinides 1992), neural networks are becoming 1 of the most powerful & appealing nonlinear signal processors for a variety of signal processing applications. As such, neural networks expand signal processing horizons (Chen 1997; Haykin

1996b), & can be considered as massively interconnected nonlinear adaptive filters. Our emphasis will be on dynamics of recurrent architectures & algorithms for prediction.

- Some Important Dates in History of Connectionism. In early 1940s pioneers of the field, McCulloch & Pitts, studied potential of interconnection of a model of a neuron. They proposed a computational model based on a simple neuron-like element (McCulloch & Pitts 1943). Others, like Hebb were concerned with the adaptation laws involved in neural systems. In 1949 Donald Hebb devised a learning rule for adapting the connections within artificial neurons (Hebb 1949). A period of early activity extends up to the 1960s with work of Rosenblatt (1962) & Widrow & Hoff (1960). In 1958, Rosenblatt coined the name 'perceptron'[11]. Based upon perceptron (Rosenblatt 1958), he developed the theory of statistical separability. Next major development: new formulation of learning rules by Widrow & Hoff in their Adaline (Widrow & Hoff 1960). In 1969, Minsky & Papert (1969) provided a rigorous analysis of perceptron. Work of Grossberg in 1976 was based on biological & psychological evidence. He proposed several new architectures of nonlinear dynamical systems (Grossberg 1974) & introduced adaptive resonance theory (ART), which is a real-time ANN that performs supervised & unsupervised learning of categories, pattern classification & prediction. In 1982 Hopfield pointed out that neural networks with certain symmetries are analogues to spin glasses.
  A seminal book on ANNs is by Rumelhart et al. (1986). Fukushima explored competitive learning in his biologically inspired Cognitron & Neocognitron (Fukushima 1975; Widrow & Lehr 1990). In 1971 Werbos developed a backpropagation learning algorithm which he published in his doctoral thesis (Werbos 1974). Rumelhart et al . rediscovered this technique in 1986 (Rumelhart et al. 1986). Kohonen (1982), introduced *self-organized maps* for pattern recognition (Burr 1993).
- Structure of Neural Networks. In neural networks, computational models or nodes are connected through weights that are adapted during use to improve performance. Main idea: to achieve good performance via dense interconnection of simple computational elements. The simplest node provides a linear combination of N weights $w_1, \ldots, w_N$, & N inputs $x_1, \ldots, x_N$, & passes the result through a nonlinearity Φ.
  Models of neural networks are specified by the net topology, node characteristics & training or learning rules. From perspective of connection patterns, neural networks can be grouped into 2 categories: feedforward networks, in which graphs have no loops, & recurrent networks, where loops occur because of feedback connections. Neural networks are specified by (Tsoi & Back 1997). Connections within a node $y = \Phi\left(\sum_i w_i x_i + w_0\right)$.
  * Node: typically a sigmoid function
  * Layer: a set of nodes at the same hierarchical level
  * Connection: constant weights or weights as a linear dynamical system, feedforward or recurrent
  * Architecture: an arrangement of interconnected neurons
  * Mode of operation: analogue or digital.

  Massively interconnected neural nets provide a greater degree of robustness or fault tolerance than sequential machines. By robustness we mean that small perturbations in parameters will also result in small deviations of the values of the signals from their nominal values.
  In our work, hence, the term *neuron* will refer to an operator which performs the mapping: Neuron : $\mathbb{R}^{N+1} \to \mathbb{R}$. The equation $y = \Phi\left(\sum_{i=1}^{N} w_i x_i + w_0\right)$ represents a mathematical description of a neuron. The input vector is given by $\mathbf{x} = [x_1, \ldots, x_N, 1]^\top$, whereas $\mathbf{w} = [w_1, \ldots, w_N, w_0]^\top$ is referred to as the weight vector of a neuron. The weight w_0 is the weight which corresponds to the bias input, which is typically set to unity. The function $\Phi : \mathbb{R} \to (0, 1)$ is monotone & continuous, most commonly of a sigmoid shape. A set of interconnected neurons is a neural network (NN). If there are N input elements to an NN & M output elements of an NN, then an NN defines a continuous mapping NN : $\mathbb{R}^N \to \mathbb{R}^M$.
- Perspective. Before 1920s, prediction was undertaken by simply extrapolating the time series through a global fit procedure. The beginning of modern time series prediction was in 1927 when Yule introduced the autoregressive model in order to predict the annual number of sunspots. For the next half century the models considered were linear, typically driven by white noise. In 1980s, state-space representation & machine learning, typically by neural networks, emerged as new potential models for prediction of highly complex, nonlinear, & nonstationary phenomena. This was the shift from rule-based models to data-driven methods (Gershenfeld & Weigend 1993).
  Time series prediction has traditionally been performed by use of linear parametric autoregressive (AR), moving-average (MA) or autoregressive moving-average (ARMA) models (Box & Jenkins 1976; Ljung & Soderstrom 1983; Makhoul 1975), the parameters of which are estimated either in a block or a sequential manner with least mean square (LMS) or recursive least-squares (RLS) algorithms (Haykin 1994). An obvious problem is that these processors are linear & are not able to cope with certain nonstationary signals, & signals whose mathematical model is not linear. On the other hand, neural networks are powerful when applied to problems whose solutions require knowledge which is difficult to specify, but for which there is an abundance of examples (Dillon & Manikopoulos 1991; Gent & Sheppard 1992; Townshend 1991). As time series prediction is conventionally performed entirely by inference of future behavior from examples of past behavior, it is a suitable application for a neural network predictor. The neural network approach to time series prediction is non-parametric in the sense that it does not need to know any information regarding the process that generates the signal. E.g., the order & parameters of an AR or ARMA process are not needed in order to carry out the prediction. This task is carried out by a process of learning from examples presented to the network & changing network weights in response to the output error.

[11] **perceptron** [n] (*computing*) an artificial network which is intended to copy the brain's ability to recognize things & see the differences between things.

Li (1992) has shown that the recurrent neural network (RNN) with a sufficiently large number of neurons is a realization of the nonlinear ARMA (NARMA) process. RNNs performing NARMA prediction have traditionally been trained by the real-time recurrent learning (RTRL) algorithm (Williams & Zipser 1989a) which provides the training process of the RNN 'on the run'. However, for a complex physical process, some difficulties encountered by RNNs e.g. high degree of approximation involved in RTRL algorithm for a higher-order MA part of underlying NARMA process, high computational complexity of $O(N^4)$, with N being the number of neurons in RNN, insufficient degree of nonlinearity involved, & relatively low robustness, induced a search for some other, more suitable schemes for RNN-based predictors.

In addition, in time series prediction of nonlinear & nonstationary signals, there is a need to learn long-time temporal dependencies. This is rather difficult with conventional RNNs because of the problem of vanishing gradient (Bengio et al. 1994). A solution to that problem might be NARMA models & nonlinear autoregressive moving average models with exogenous inputs (NARMAX) (Siegelmann et al. 1997) realized by recurrent neural networks. However, the quality of performance is highly dependent on the order of AR & MA parts in NARMAX model.

Main reasons for using neural networks for prediction rather than classical time series analysis are (Wu 1995)

* they are computationally at least as fast, if not faster, than most available statistical techniques
* they are self-monitoring (i.e. they learn how to make accurate predictions)
* they are as accurate if not more accurate than most of the available statistical techniques
* they provide iterative forecasts
* they are able to cope with nonlinearity & nonstationarity of input processes
* they offer both parametric & nonparametric prediction.

- Neural Networks for Prediction: Perspective. Many signals are generated from an inherently nonlinear physical mechanism & have statistically non-stationary properties, a classic example of which is speech. Linear structure adaptive filters are suitable for the nonstationary characteristics of such signals, but they do not account for nonlinearity & associated higher-order statistics (Shynk 1989). Adaptive techniques which recognize the nonlinear nature of the signal should therefore outperform traditional linear adaptive filtering techniques (Haykin 1996a; Kay 1993). The classic approach to time series prediction is to undertake an analysis of the time series data, which includes modeling, identification of the model & model parameter estimation phases (Makhoul 1975). The design may be iterated by measuring the closeness of the model to the real data. This can be a long process, often involving the derivation, implementation & refinement of a number of models before one with appropriate characteristics is found.

  In particular, the most difficult systems to predict are

  * those with non-stationary dynamics, where the underlying behavior varies with time, a typical example of which is speech production
  * those which deal with physical data which are subject to noise & experimentation error, e.g. biomedical signals
  * those which deal with short time series, providing few data points on which to conduct the analysis, e.g. heart rate signals, chaotic signals & meteorological signals.

  In all these situations, traditional techniques are severely limited & alternative techniques must be found (Bengio 1995; Haykin & Li 1995; Li & Haykin 1993; Niranjan & Kadirkamanathan 1991).

  On the other hand, neural networks are powerful when applied to problems whose solutions require knowledge which is difficult to specify, but for which there is an abundance of examples (Dillon & Manikopoulos 1991; Gent & Sheppard 1992; Townshend 1991). From a system theoretic point of view, neural networks can be considered as a conveniently parametrized class of nonlinear maps (Narendra 1996).

  There has been a recent resurgence in the field of ANNs caused by new net topologies, VLSI computational algorithms & introduction of massive paralelism into neural networks. As such, they are both universal function approximators (Cybenko 1989; Hornik et al. 1989) & arbitrary pattern classifiers. From the Weierstrass theorem, polynomials, & many other approximation schemes, can approximate arbitrarily well a continuous function. Kolmogorov's theorem (a negative solution of Hilbert's 13th problem (Lorentz 1976)) states that any continuous function can be approximated using only linear summations & nonlinear but continuously increasingly functions of only 1 variable. This makes neural networks suitable for universal approximation, & hence prediction. Although sometimes computationally demanding (Williams & Zipser 1995), neural networks have found their place in the area of nonlinear autoregressive moving average (NARMA) (Bailer-Jones et al. 1998; Connor et al. 1992; Lin et al. 1996) prediction applications. Comprehensive survey papers on the use & role of ANNs can be found in Widrow & Lehr (1990), Lippmann (1987), Medler (1998), Ermentrout (1998), Hunt et al. (1992) & Billings (1980).

  Only recently, neural networks have been considered for prediction. A recent competition by the Santa Fe Institute for Studies in the Science of Complexity (1991–1993) (Weigend & Gershenfeld 1994) showed that neural networks can outperform conventional linear predictors in a number of applications (Waibel et al. 1989). In journals, there has been an ever increasing interest in applying neural networks. A most comprehensive issue on recurrent neural networks is the issue of the *IEEE Transactions of Neural Networks*, vol. 5, no. 2, Mar 1994. In the signal processing community, there has been a recent special issue 'Neural Networks for Signal Processing' of *IEEE Transactions on Signal Processing*, vol. 45, no. 11, Nov 1997, & also the issue 'Intelligent Signal Processig' of the *Proceedings of IEEE*, vol. 86, no. 11, Nov 1998, both dedicated to the use of neural networks in signal processing applications.

  Frequency of appearance of articles on recurrent neural networks in common citation index databases. Number of journal & conference articles on recurrent neural networks in IEE/IEEE publications between 1988 & 1999. The data were gathered using IEL Online service, & these publications are mainly periodicals & conferences in electronics engineering. Frequency

of appearance for BIDS/ATHENS database, between 1988 & 2000, which also includes non-engineering publications. There is a clear growing trend in frequency of appearance of articles on recurrent neural networks. Therefore, felt that there was a need for a research monograph that would cover a part of the area with up to data ideas & results.

- Structure of Book. Divide book into 12 chapters & 10 appendices.
  - Chap. 1: An introduction to connectionism & notion of neural networks for prediction
  - Chap. 2: Detail fundamentals of adaptive signal processing & learning theory
  - Chap. 3: an initial overview of network architectures for prediction.
  - Chap. 4: a detailed discussion of activation functions & new insights are provided by consideration of neural networks within framework of modular groups from number theory.
  - Chap. 5: build material in Chap. 3 & provide more comprehensive coverage of recurrent neural network architectures together with concepts from nonlinear system modeling.
  - Chap. 6: Consider neural networks as nonlinear adaptive filters whereby develop necessary learning strategies for recurrent neural networks.
  - Chap. 7: Consider stability issues for certain recurrent neural network architectures through exploitation of fixed point theory & derive bounds for global asymptotic stability.
  - Chap. 8: Introduce a posteriori adaptive learning algorithms & highlight synergy with data-reusing algorithms.
  - Chap. 9: Derive a new class of normalized algorithms for online training of recurrent neural networks.
  - Chap. 10: Address convergence of online learning algorithms for neural networks.
  - Chap. 11: Present experimental results for prediction of nonlinear & nonstationary signals with recurrent neural networks.
  - Chap. 12: Describe exploitation of inherent relationships between parameters within recurrent neural networks.
  - Appendices A–J provide background to main chapters & cover key concepts from linear algebra, approximation theory, complex sigmoid activation functions, a precedent learning algorithm for recurrent neural networks, terminology in neural networks, *a posteriori* techniques in science & engineering, contraction mapping theory, linear relaxation & stability, stability of general nonlinear systems & deseasonalizing of time series. A comprehensive bibliography.
- Readership. This book is targeted at graduate students & research engineers active in the areas of communications, neural networks, nonlinear control, signal processing & time series analysis. It will also be useful for engineers & scientists working in diverse application areas, e.g., AI, biomedicine, earth sciences, finance & physics.

- Fundamentals.
  - Perspective. Adaptive systems are at the very core of modern digital signal processing. There are many reasons for this, foremost amongst these is that adaptive filtering, prediction or identification do not require explicit *a priori* statistical knowledge of the input data. Adaptive systems are employed in numerous areas e.g. biomedicine, communications, control, radar, sonar, & video processing (Haykin 1996a).
    Chap Summary. Introduce fundamentals of adaptive systems. Emphasis is 1st placed upon various structures available for adaptive signal processing, & includes predictor structure which is focus of this book. Detail basic learning algorithms & concepts in context of linear & nonlinear structure filters & networks. Discuss issue of modularity.
  - Adaptive Systems. Adaptability, in essence, is ability to react in sympathy with disturbances to environment. A system that exhibits adaptability is said to be *adaptive*. Biological systems are adaptive systems; animals, e.g., can adapt to changes in their environment through a learning process (Haykin 1999a). Block diagram of an adaptive system: A generic adaptive system employed in engineering. It consists of:
    - a set of adjustable parameters (weights) within some filter structure
    - an error calculation block (difference between desired response & output of filter structure)
    - a control (learning) algorithm for adaptation of weights.

    Type of learning represented is so-called *supervised learning*, since the learning is directed by the desired response of the system. Goal: to adjust iteratively free parameters (weights) of adaptive system so as to minimize a prescribed cost function in some predetermined sense.[12] The filter structure within adaptive system may be linear, e.g. a finite impulse response (FIR) or infinite impulse response (IIR) filter, or nonlinear, e.g. a Volterra filter or a neural network.
    - Configurations of Adaptive Systems Used in Signal Processing. 4 typical configurations of adaptive systems used in engineering: (Jenkins et al. 1996)
      · System identification configuration
      · Noise canceling configuration
      · Prediction configuration
      · Inverse system configuration

      Use notions of an adaptive filter & adaptive system interchangeably here. For the system identification configuration, both the adaptive filter & the unknown system are fed with the same input signal $x(k)$. Error signal $e(k)$ is formed at output as $e(k) \coloneqq d(k) - y(k)$, & parameters of adaptive system are adjusted using this error information. An attractive point of this configuration is that desired response signal $d(k)$, also known as a *teaching/training signal*, is readily

[12] Aim: to minimize some function of error e. If $E[e^2]$ is minimized, consider minimum mean squared error (MSE) adaptation, the *statistical expectation operator* $E[\cdot]$ is due to random nature of inputs to adaptive system.

available from unknown system (plant). Applications of this scheme are in acoustic & electrical echo cancellation, control, & regulation of real-time industrial & other processes (plants). Knowledge about system is stored in set of converged weights of adaptive system. If dynamics of plant are not time-varying, possible to identify parameters (weights) of the plant to an arbitrary accuracy.

If desire to form a system which inter-relates noise components in input & desired response signals, noise canceling configuration can be implemented. The only requirement: noise in primary input & reference noise are correlated. This configuration subtracts an estimate of noise from received signal. Applications of this configuration include noise cancellation in acoustic environments & estimation of total ECG from mixture of maternal & foetal ECG (Widrow & Stearns 1985).

In adaptive prediction configuration, desired signal is input signal advanced relative to input of adaptive filter. This configuration has numerous applications in various areas of engineering, science, & technology & most of material in this book is dedicated to prediction. Prediction may be considered as a basis for any adaptation process, since adaptive filter is trying to predict desired response.

Inverse system configuration has an adaptive system cascaded with unknown system. A typical application is adaptive channel equalization in telecommunications, whereby an adaptive system tries to compensate for possibly time-varying communication channel, so that transfer function from input to output approximates a pure delay.

In most adaptive signal processing applications, parametric methods are applied which require *a priori* knowledge (or postulation) of a specific model in form of differential or difference equations. Thus, necessary to determine appropriate model order for successful operation, which will underpin data length requirements. On the other hand, nonparametric methods employ general model forms of integral equations or functional expansions valid for a broad class of dynamic nonlinearities. Most widely used nonparametric methods are referred to as Volterra–Wiener approach & are based on functional expansions.

- ∗ Blind Adaptive Techniques. Presence of an explicit desired response signal $d(k)$ in all structures shown in Block diagram of a blind equalization structure implies that conventional, supervised, adaptive signal processing techniques may be applied for purpose of learning. When no such signal is available, may still be possible to perform learning by exploiting so-called *blind*, or *unsupervised*, methods. These methods exploit certain *a priori* statistical knowledge of input data. For a single signal, this knowledge may be in form of its constant modulus property, or, for multiple signals, their mutual statistical independence (Haykin 2000). Structure of a blind equalizer is shown, notice desired response is generated from output of a zero-memory nonlinearity. This nonlinearity is implicitly being used to test higher-order (i.e. greater than 2nd-order) statistical properties of output of adaptive equalizer. When ideal convergence of adaptive filter is achieved, zero-memory nonlinearity has no effect upon signal $y(k)$ & therefore $y(k)$ has identical statistical properties to that of channel input $s(k)$.

- ◦ Gradient-Based Learning Algorithms. A brief introduction to notion of gradient-based learning. Aim: to update iteratively weight vector $\mathbf{w}$ of an adaptive system so that a nonnegative error measure $\mathcal{J}(\cdot)$ is reduced at each time step k, $\mathcal{J}(\mathbf{w}+\Delta\mathbf{w}) < \mathcal{J}(\mathbf{w})$, where $\Delta\mathbf{w}$ represents change in $\mathbf{w}$ from 1 iteration to the next. This will generally ensure that after training, an adaptive system has captured relevant properties of unknown system that we are trying to model. Using a Taylor series expansion to approximate error measure, obtain $\mathcal{J}(\mathbf{w}) + \Delta\mathbf{w}\partial_{\mathbf{w}}\mathcal{J}(\mathbf{w}) + O(\mathbf{w}^2) < \mathcal{J}(\mathbf{w})$. This way, with the assumption that the higher-order terms in LHS can be neglected, $\mathcal{J}(\mathbf{w}+\Delta\mathbf{w}) < \mathcal{J}(\mathbf{w})$ can be rewritten as $\Delta\mathbf{w}\partial_{\mathbf{w}}\mathcal{J}(\mathbf{w}) < 0$. From this, an algorithm that would continuously reduce error measure on the run, should change the weights in opposite direction of gradient $\partial_{\mathbf{w}}\mathcal{J}(\mathbf{w})$, i.e., $\Delta\mathbf{w} = -\eta\partial_{\mathbf{w}}\mathcal{J}$, where η is a small positive scalar called the *learning rate, step size or adaptation parameter*.

  Examining $\Delta\mathbf{w} = -\eta\partial_{\mathbf{w}}\mathcal{J}$, if gradient of error measure $\mathcal{J}(\mathbf{w})$ is steep, large changes will be made to weights, & conversely, if gradient of error measure $\mathcal{J}(\mathbf{w})$ is small, namely a flat error surface, a larger step size η may be used. Gradient descent algorithms cannot, however, provide a sense of importance or hierarchy to weights (Agarwal & Mammone 1994). E.g., value of weight w_1 in Fig. 2.4 is 10 times greater than w_2 & 1000 times greater than w_4. Hence, component of output of filter within adaptive system due to w_1 will, on average, be larger than that due to other weights. For a conventional gradient algorithm, however, change in w_1 will not depend upon relative sizes of coefficients, but relative sizes of input data. This deficiency provides motivation for certain partial update gradient-based algorithms (Douglas 1997).

  Important to notice: *gradient-descent-based algorithms inherently forget old data*, which leads to a problem called *vanishing gradient* & has particular importance for learning in filters with recursive structures.

- ◦ A General Class of Learning Algorithms. To introduce a general class of learning algorithms & explain in very crude terms relationships between them, follow approach from Guo & Ljung (1995). Start from *linear regression equation* $\boxed{y(k) = \mathbf{x}^\top(k)\mathbf{w}(k) + \nu(k)}$, where $y(k)$: output signal, $\mathbf{x}(k)$: a vector comprising input signals, $\nu(k)$: a disturbance or noise sequence, & $\mathbf{w}(k)$: an unknown time-varying vector of weights (parameters) of adaptive system. Variation of weights at time k is denoted by $\mathbf{n}(k)$, & weight change equation becomes $\mathbf{w}(k) = \mathbf{w}(k-1) + \mathbf{n}(k)$. Adaptive algorithms can track weights only approximately, hence for the following analysis use symbol $\hat{\mathbf{w}}$. A general expression for weight update in an adaptive algorithm:

$$\hat{\mathbf{w}}(k+1) = \hat{\mathbf{w}}(k) + \eta\Gamma(k)(y(k) - \mathbf{x}^\top(k)\hat{\mathbf{w}}(k)), \tag{185}$$

  where $\Gamma(k)$: adaptation gain vector, & η: step size. To assess how far an adaptive algorithm is from optimal solution, introduce *weight error vector* $\breve{\mathbf{w}}(k)$, & a sample input matrix $\Sigma(k)$ as $\breve{\mathbf{w}}(k) \coloneqq \mathbf{w}(k) - \hat{\mathbf{w}}(k), \Sigma(k) \coloneqq \Gamma(k)\mathbf{x}^\top(k)$. Yield *weight error equation*:

$$\breve{\mathbf{w}}(k+1) = (I - \eta\Sigma(k))\breve{\mathbf{w}}(k) - \eta\Gamma(k)\nu(k) + \mathbf{n}(k+1). \tag{186}$$

For different gains $\Gamma(k)$, 3 well-known algorithms can be obtained from (185). Notice: role of η in RLS & KF algorithm is different to that in LMS algorithm. For RLS & KF may put $\eta = 1$ & introduce a forgetting factor instead.

(a) Least mean square (LMS) algorithm: $\Gamma(k) = \mathbf{x}(k)$.

(b) Recursive least-squares (RLS) algorithm:

$$\Gamma(k) = P(k)\mathbf{x}(k), \tag{187}$$

$$P(k) = \frac{1}{1-\eta}\left[P(k-1) - \eta\frac{P(k-1)\mathbf{x}(k)\mathbf{x}^\top(k)P(k-1)}{1-\eta+\eta\mathbf{x}^\top(k)P(k-1)\mathbf{x}(k)}\right]. \tag{188}$$

(c) Kalman filter (KF) algorithm (Guo & Ljung 1995; Kay 1993):

$$\Gamma(k) = \frac{P(k-1)\mathbf{x}(k)}{R+\eta\mathbf{x}^\top(k)P(k-1)\mathbf{x}(k)}, \tag{189}$$

$$P(k) = P(k-1) - \frac{\eta P(k-1)\mathbf{x}(k)\mathbf{x}^\top(k)P(k-1)}{R+\eta\mathbf{x}^\top(k)P(k-1)\mathbf{x}(k)} + \eta Q. \tag{190}$$

The KF algorithm is the optimal algorithm in this setting if elements of $\mathbf{n}(k)$ & $\nu(k)$ in (2.5) & (2.6) are Gaussian noises with a covariance matrix $Q > 0$ & a scalar value $R > 0$, resp. (Kay 1993). All of these adaptive algorithms can be referred to as sequential estimators, since they refine their estimate as each new sample arrives. On the other hand, block-based estimators require all measurements to be acquired before the estimate is formed.

Although the most important measure of quality of an adaptive algorithm is generally covariance matrix of weight tracking error $E[\breve{\mathbf{w}}(k)\breve{\mathbf{w}}^\top(k)]$, due to statistical dependence between $\mathbf{x}(k), \nu(k), \mathbf{n}(k)$, precise expressions for this covariance matrix are extremely difficult to obtain.

To undertake statistical analysis of an adaptive learning algorithm, classical approach: assume $\mathbf{x}(k), \nu(k), \mathbf{n}(k)$ are statistically independent. Another assumption: homogeneous part of (2.9) $\breve{\mathbf{w}}(k+1) = (I - \eta\Sigma(k))\breve{\mathbf{w}}(k)$ & its averaged version $E[\breve{\mathbf{w}}(k+1)] = (I - \eta E[\Sigma(k)])E[\breve{\mathbf{w}}(k)]$ are exponentially stable in stochastic & deterministic senses (Guo & Ljung 1995).

∗ Quasi-Newton Learning Algorithm. Quasi-Newton learning algorithm utilizes 2nd-order derivative of objective function to adapt weights. If change in objective function between iterations in a learning algorithm is modeled with a Taylor series expansion, have

$$\Delta E(\mathbf{w}) = E(\mathbf{w}+\Delta\mathbf{w}) - E(\mathbf{w}) \approx (\nabla_{\mathbf{w}}E(\mathbf{w}))^\top\Delta\mathbf{w} + \frac{1}{2}\Delta\mathbf{w}^\top\mathbf{H}\Delta\mathbf{w}. \tag{191}$$

After setting differential w.r.t. $\Delta\mathbf{w}$ to 0, weight update equation becomes $\Delta\mathbf{w} = -\mathbf{H}^{-1}\nabla_{\mathbf{w}}E(\mathbf{w})$. The Hessian $\mathbf{H}$ in this equation determines not only the direction but also step size of gradient descent.

Conclude: adaptive algorithms mainly differ in their form of adaptation gains. The gains can be roughly divided into 2 classes: gradient-based gains (e.g. LMS, quasi-Newton) & Riccati equation-based gains (e.g. KF & RLS).

◦ A Step-by-Step Derivation of Least Mean Square (LMS) Algorithm. Consider a set of input-output pairs of data described by a mapping function f: $d(k) = f(\mathbf{x}(k))$, $k = 1, \ldots, N$. Function $f(\cdot)$ is assumed to be unknown. Using concept of adaptive systems explained, aim: to approximate unknown function $f(\cdot)$ by a function $F(\cdot, \mathbf{w})$ with adjustable parameters $\mathbf{w}$, in some prescribed sense. Function F is defined on a system with a known architecture or structure. Convenient to define an instantaneous performance index,

$$J(\mathbf{w}(k)) = [d(k) - F(\mathbf{x}(k), \mathbf{w}(k))]^2, \tag{192}$$

which represents an energy measure. In that case, function F is most often just inner product $F = \mathbf{x}^\top(k)\mathbf{w}(k)$ & corresponds to operation of a linear FIR filter structure. Goal: to find an optimization algorithm that minimizes cost function $J(\mathbf{w})$. Common choice of algorithm is motivated by method of steepest descent, & generates a sequence of weight vectors $\mathbf{w}(1), \mathbf{w}(2), \ldots$ as $\boxed{\mathbf{w}(k+1) = \mathbf{w}(k) - \eta\mathbf{g}(k)}$, $k = 0, 1, 2, \ldots$ (2.21), where $\mathbf{g}(k)$ is gradient vector of cost function $J(\mathbf{w})$ at point $\mathbf{w}(k)$: $\mathbf{g}(k) = \partial_{\mathbf{w}}J(\mathbf{w})|_{\mathbf{w}=\mathbf{w}(k)}$. Parameter η in $\mathbf{w}(k+1) = \mathbf{w}(k) - \eta\mathbf{g}(k)$ determines behavior of algorithm:

∗ for η small, algorithm (2.21) converges towards global minimum of error performance surface;

∗ if value of η approaches some critical value η_c, trajectory of convergence on error performance surface is either oscilltory or overdamped;

∗ if value of η is $> \eta_c$, system is unstable & does not converge.

These observations can only be visualized in 2D, i.e. for only 2 parameter values $w_1(k), w_2(k)$, & can be found in Widrow & Stearns (1985). If approximation function F in gradient descent algorithm (2.21) is linear, call such an adaptive system a *linear adaptive system*. Otherwise, describe it as a nonlinear adaptive system. Neural networks belong to this latter class.

∗ Wiener Filter. Suppose system shown in Fig. 2.1 is modeled as a linear FIR filter: Fig. 2.5: Structure of a finite impulse response filter, have $F(\mathbf{x}, \mathbf{w}) = \mathbf{x}^\top\mathbf{w}$, dropping k index for convenience. Consequently, instantaneous cost function $J(\mathbf{w}(k))$ is a quadratic function of weight vector. Wiener filter is based upon minimizing ensemble average of this instantaneous cost function, i.e.,

$$J_{\text{Wiener}}(\mathbf{w}(k)) = E[[d(k) - \mathbf{x}^\top(k)\mathbf{w}(k)]^2], \tag{193}$$

& assuming $d(k), x(k)$ are zero mean & jointly wide sense stationary. To find minimum of cost function, differentiate w.r.t. $\mathbf{w}$ & obtain

$$\partial_{\mathbf{w}} J_{\text{Wiener}} = -2E[e(k)\mathbf{x}(k)], \tag{194}$$

where $e(k) = d(k) - \mathbf{x}^\top(k)\mathbf{w}(k)$.
At Wiener solution, this gradient equals null vector $\mathbf{0}$. Solving (194) for this condition yields Wiener solution, $\mathbf{w} = \mathbf{R}_{\mathbf{x},\mathbf{x}}^{-1}\mathbf{r}_{\mathbf{x},d}$, where $\mathbf{R}_{\mathbf{x},\mathbf{x}} = E[\mathbf{x}(k)\mathbf{x}^\top(k)]$ is the autocorrelation matrix of zero mean input data $\mathbf{x}(k)$ & $\mathbf{r}_{\mathbf{x},d} = E[\mathbf{x}(k)d(k)]$ is crosscorrelation between input vector & desired signal $d(k)$. Wiener formula has same general form as block least-squares (LS) solution, when exact statistics are replaced by temporal averages.
RLS algorithm, as in (2.12), with assumption that input & desired response signals are jointly ergodic, approximates Wiener solution & asymptotically matches Wiener solution. More details about derivation of Wiener filter can be found in Haykin (1996a, 1999a).

* Further Perspective on Least Mean Square (LMS) Algorithm. To reduce computational complexity of Wiener solution, which is a block solution, can use method of steepest descent for a recursive, or sequential, computation of weight vector $\mathbf{w}$. Derive LMS algorithm for an adaptive FIR filter, structure of which is shown in Fig. 2.5. In view of a general adaptive system, this FIR filter becomes filter structure within Fig. 2.1. Output of this filter: $y(k) = \mathbf{x}^\top(k)\mathbf{w}(k)$. Widrow & Hoff (1960) utilized this structure for adaptive processing & proposed instantaneous values of autocorrelation & crosscorrelation matrices to calculate gradient term within steepest descent algorithm. Cost function they proposed was $J(k) = \frac{1}{2}e^2(k)$, which is again based upon instantaneous output error $e(k) = d(k) - y(k)$. In order to derive weight update equation, start from instantaneous gradient $\partial_{\mathbf{w}(k)} J(k) = e(k)\partial_{\mathbf{w}(k)} e(k)$. Following same procedure as for general gradient descent algorithm, obtain

$$\partial_{\mathbf{w}(k)} e(k) = -\mathbf{x}(k), \;\; \partial_{\mathbf{w}(k)} J(k) = -e(k)\mathbf{x}(k). \tag{195}$$

Set of equations that describes LMS algorithm is given by

$$\begin{cases} y(k) = \sum_{i=1}^{N} x_i(k) w_i(k) = \mathbf{x}^\top(k)\mathbf{w}(k), \\ e(k) = d(k) - y(k), \\ \mathbf{w}(k+1) = \mathbf{w}(k) + \eta e(k)\mathbf{x}(k). \end{cases} \tag{196}$$

LMS algorithm is a very simple yet extremely popular algorithm for adaptive filtering. Also optimal in H^∞ sense which justifies its practically utility (Hassibi et al. 1996).

- On Gradient Descent for Nonlinear Structures. Adaptive filters & neural networks are formally equivalent, in fact, structures of neural networks are generalizations of linear filters (Maass & Sontag 2000; Nerrand et al. 1991). Depending on architecture of a neural network & whether it is used online or offline, 2 broad classes of learning algorithms are available:
  * techniques that use a direct computation of gradient, which is typical for linear & nonlinear adaptive filters
  * techniques that involve backpropagation, which is commonplace for most offline applications of neural networks.

  Backpropagation is a computational procedure to obtain gradients necessary for adaptation of weights of a neural network contained within its hidden layers & is not radically different from a general gradient algorithm.
  As interested in neural networks for real-time signal processing, will analyze online algorithms that involve direct gradient computation. In this sect, introduce a learning algorithm for a nonlinear FIR filter, whereas learning algorithms for online training of recurrent neural networks will be introduced later. Start from a simple nonlinear FIR filter, which consists of standard FIR filter cascaded with a memoryless nonlinearity Φ as shown in Fig. 2.6: Structure of a nonlinear adaptive filter. This structure can be seen as a single neuron with a dynamical FIR synapse. This FIR synapse provides memory to neuron. Output of this filter is given by $y(k) = \Phi(\mathbf{x}^\top(k)\mathbf{w}(k))$. Nonlinearity $\Phi(\cdot)$ after tap-delay line is typically a sigmoid. Using ideas from LMS algorithm, if cost function is given by $J(k) = \frac{1}{2}e^2(k)$, have

$$e(k) = d(k) - \Phi(\mathbf{x}^\top(k)\mathbf{w}(k)), \tag{197}$$

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \eta\nabla_{\mathbf{w}} J(k), \tag{198}$$

where $e(k)$ is the *instantaneous error* at output neuron, $d(k)$ is some *teaching (desired) signal*, $\mathbf{w}(k) = [w_1(k), \ldots, w_N(k)]^\top$: *weight vector* & $\mathbf{x}(k) = [x_1(k), \ldots, x_N(k)]^\top$: *input vector*.
Gradient $\nabla_{\mathbf{w}} J(k)$ can be calculated as

$$\partial_{\mathbf{w}(k)} J(k) = e(k)\partial_{\mathbf{w}(k)} e(k) = -e(k)\Phi'(\mathbf{x}^\top(k)\mathbf{w}(k))\mathbf{x}(k), \tag{199}$$

where $\Phi'(\cdot)$ represents 1st derivative of nonlinearity $\Phi(\cdot)$ & weight update equation (198) can be rewritten as

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \eta\Phi'(\mathbf{x}^\top(k)\mathbf{w}(k))e(k)\mathbf{x}(k). \tag{200}$$

This is the *weight update equation* for a direct gradient algorithm for a nonlinear FIR filter.
Extension to a General Neural Network. When deriving a direct gradient algorithm for a general neural network, network architecture should be taken into account. For large networks for offline processing, classical backpropagation is the most convenient algorithm. However, for online learning, extensions of previous algorithm should be considered.

- On Some Important Notions From Learning Theory. Discuss in more detail interrelations between error, error function, & objective function in learning theory.
  - Relationship Between Error & Error Function. Error at output of an adaptive system is defined as difference between output value of network & target (desired output) value. E.g., *instantaneous error* $e(k)$ is defined as $e(k) \coloneqq d(k) - y(k)$. Instantaneous error can be positive, negative or zero, & is therefore not a good candidate for criterion function for training adaptive systems. Here look for another function, called *error function*, i.e., a function of instantaneous error, but is suitable as a criterion function for learning. Error functions are also called *loss functions*. They are defined so that an increase in the error function corresponds to a reduction in quality of learning, & they are nonnegative. An error function can be defined as $E(N) = \sum_{i=0}^{N} e^2(i)$ or as an average value $\bar{E}(N) = \frac{1}{N+1} \sum_{i=0}^{N} e^2(i)$.
  - Objective Function. *Objective function* is a function that we want to minimize during training. It can be equal to an error function, but often it may include other terms to introduce constraints. E.g. in generalization, too large a network might lead to overfitting. Hence objective function can consist of 2 parts, one for error minimization & the other which is either a penalty for a large network or a penalty term for excessive increase in weights of adaptive system or some other chosen function (Tikhonov et al. 1998). An example of such an objective function for online learning is:

$$J(k) = \frac{1}{N} \sum_{i=1}^{N} (e^2(k-i+1) + G(\|\mathbf{w}(k-i+1)\|_2^2)), \tag{201}$$

    where G is some linear or nonlinear function. Often use symbols E, J interchangeably to denote cost function.
  - Types of Learning w.r.t. Training Set & Objective Function. *Batch learning* is also known as *epochwise*, or *offline learning*, & is a common strategy for offline training. Idea: to adapt weights once whole training set has been presented to an adaptive system. It can be described by following steps.
    (a) Initialize weights
    (b) Repeat
      · Pass all training data through network
      · Sum errors after each particular pattern
      · Update weights based upon total error
      · Stop if some prescribed error performance is reached

    Counterpart of batch learning is so-called *incremental learning, online, or pattern training*. The procedure for this type of learning is as follows.
    (a) Initialize weights
    (b) Repeat
      · Pass 1 pattern through network
      · Update weights based upon instantaneous error
      · Stop if some prescribed error performance is reached

    Choice of type of learning is very much dependent upon application. Quite often, for networks that need initialization, perform 1 type of learning in initialization procedure, which is by its nature an offline procedure, & then use some other learning strategy while network is running. Such is the case with recurrent neural networks for online signal processing (Mandic & Chambers 1999f).
  - Deterministic, Stochastic, & Adaptive Learning. *Deterministic learning* is an optimization technique based on an objective function which always produces same result, no matter how many times we recompute it. Deterministic learning is always offline.
    Stochastic learning is useful when objective function is affected by noise & local minima. It can be employed within context of a gradient descent learning algorithm. Idea: learning rate gradually decreases during training & hence steps on error performance surface in beginning of training are large which speeds up training when far from optimal solution. Learning rate is small when approaching optimal solution, hence reducing misadjustment. This gradual reduction of learning rate can be achieved by e.g. annealing (Kirkpatrick et al. 1983; Rose 1998; Szu & Hartley 1987).
    The idea behind concept of adaptive learning is to forget the past when it is no longer relevant & adapt to changes in environment. The terms 'adaptive learning' or 'gear-shifting' are sometimes used for gradient methods in which learning rate is changed during training.
  - Constructive Learning. Constructive learning deals with change of architecture or interconnections in network during training. Neural networks for which topology can change over time are called *ontogenic* neural networks (Fiesler & Beale 1997). 2 basic classes of constructive learning are network growing & network pruning. In network growing approach, learning begins with a network with no hidden units, & if error is too big, new hidden units are added to network, training resumes, & so on. Most used algorithm based upon network growing is so-called *cascade-correlation algorithm* (Hoehfeld & Fahlman 1992). Network pruning starts from a large network & if error in learning is smaller than allowed, network size is reduced until desired ratio between accuracy & network size is reached (Reed 1993; Sum et al. 1999).
  - Transformation of Input Data, Learning, & Dimensionality. A natural question is whether to linearly/nonlinearly transform data before feeding them to an adaptive processor. This is particularly important for neural networks, which are nonlinear processors. If consider each neuron as a basic component of a neural network, then can refer to a general

neural network as a system with componentwise nonlinearities. To express this formally, consider a scalar function $\sigma : \mathbb{R} \to \mathbb{R}$ & systems of form

$$\mathbf{y}(k) = \boldsymbol{\sigma}(\mathbf{A}\mathbf{x}(k)), \tag{202}$$

where matrix $\mathbf{A}$ is an $N \times N$ matrix & $\boldsymbol{\sigma}$ is applied componentwise $\boldsymbol{\sigma}(x_1(k), \ldots, x_N(k)) = (\sigma(x_1(k)), \ldots, \sigma(x_N(k)))$. Systems of this type arise in a wide variety of situations. For a linear σ, have a linear system. If range of σ is finite, state vector of (202) takes values from a finite set, & dynamical properties can be analyzed in time which is polynomial in number of possible states. Throughout this book, interested in functions σ & combination matrices $\mathbf{A}$ which would guarantee a fixed point of this mapping. Neural networks are commonly of form (202). In such a context call σ *activation function*. Results of Siegelmann & Sontag (1995) show: saturated linear systems (piecewise linear) can represent Turing machines, which is achieved by encoding transition rules of Turing machine in matrix $\mathbf{A}$.

Curse of dimensionality. Curse of dimensionality (Bellman 1961) refers to exponential growth of computation needed for a specific task as a function of dimensionality of input space. In neural networks, a network quite often has to deal with many irrelevant inputs which, in turn, increase dimensionality of input space. In such a case, network uses much of its resources to represent & compute irrelevant information, which hampers processing of desired information. A remedy for this problem is preprocessing of input data, e.g. feature extraction, & to introduce some importance function to input samples. Curse of dimensionality is particularly prominent in unsupervised learning algorithms. Radial basis functions are also prone to this problem. Selection of a neural network model must therefore be suited for a particular task. Some a priori information about data & scaling of inputs can help to reduce severity of problem.

Transformations on input data. Activation functions used in neural networks are centered around a certain value in their output space. E.g., mean of logistic function is 0.5, whereas the tanh function is centered around zero. Therefore, in order to perform efficient prediction, should match range of input data, their mean & variance, with range of chosen activation function. There are several operations that we could perform on input data, e.g. following.

(a) Normalization, which in this context means dividing each element of input vector $\mathbf{x}(k)$ by its squared norm, i.e. $x_i(k) \in \mathbf{x}(k) \to \frac{x_i(k)}{\|\mathbf{x}(k)\|_2^2}$.

(b) Rescaling, which means transforming input data in manner that we multiply/divide them by a constant & also add/subtract a constant from data.[13]

(c) Standardization, which is borrowed from statistics, where, e.g., a random Gaussian vector is standardized if its mean is subtracted from it, & vector is then divided by its standard deviation. Resulting random variable is called a 'standard normal' random variable with zero mean & unity standard deviation. Some examples of data standardization:

· Standardization to zero mean & unity standard deviation can be performed as

$$\text{mean} = \frac{\sum_i X_i}{N}, \ \text{std} = \sqrt{\frac{\sum_i (X_i - \text{mean}^2)}{N-1}}. \tag{203}$$

Standardized quantity becomes $S_i = \frac{X_i - \text{mean}}{\text{std}}$.

· Standardize X to midrange 0 & range 2. This can be achieved by

$$\text{midrange} = \frac{1}{2}(\max_i X_i + \min_i X_i), \ \text{range} = \max_i X_i - \min_i X_i, \ S_i = \frac{X_i - \text{midrange}}{\text{range}/2}. \tag{204}$$

(d) Principal component analysis (PCA) represents data by a set of unit norm vectors called *normalized eigenvectors*. Eigenvectors are positioned along directions of greatest data variance. Eigenvectors are found from covariance matrix $\mathbf{R}$ of input dataset. An eigenvalue λ_i, $i = 1, \ldots, N$, is associated with each eigenvector. Every input data vector is then represented by a linear combination of eigenvectors.

As pointed out earlier, standardizing input variables has an effect on training, since steepest descent algorithms are sensitive to scaling due to change in weights being proportional to value of gradient & input data.

Nonlinear transformations of data. This method to transform data can help when dynamic range of data is too high. In the case, e.g., typically apply log function to input data. Log function is often applied in error & objective functions for same purposes.

◦ Learning Strategies. To construct an optimal neural approximating model, have to determine an appropriate training set containing all relevant information of process & define a suitable topology that matches complexity & performance requirements. Training set construction issue requires 4 entities to be considered (Alippi & Piuri 1996; Bengio 1995; Haykin & Li 1995; Shadafan & Niranjan 1993):

∗ number of training data samples N_{D}

∗ number of patterns N_{P} constituting a batch

∗ number of batches N_{B} to be extracted from training set

∗ number of times generic batch is presented to network during learning.

Assumption is that training set is sufficiently rich so that it contains all the relevant information necessary for learning. Requirement coincides with hypothesis that training data have been generated by a fully exciting input signal, e.g. white noise, which is able to excite all process dynamics. White noise is a persistently exciting input signal & is used for driving component of moving average (MA), autoregressive (AR), & autoregressive moving average (ARMA) models.

[13] In real life a typical rescaling is transforming temperature from Celcius into Fahrenheit scale.

- General Framework for Training of Recurrent Networks by Gradient-Descent-Based Algorithms. Summarize some of important concepts mentioned earlier.
  - Adaptive vs. Nonadaptive Training. Training of a network makes use of 2 sequences, sequence of inputs & sequence of corresponding desired outputs. If network is 1st trained (with a training sequence of finite length) & subsequently used (with fixed weights obtained from training), this mode of operation is referred to as *non-adaptive* (Nerrand et al. 1994). Conversely, the term *adaptive* refers to mode of operation whereby network is trained permanently throughout its application (with a training sequence of infinite length). Therefore, adaptive network is suitable for input processes which exhibit statistically non-stationary behavior, a situation which is normal in the fields of adaptive control & signal processing (Bengio 1995; Haykin 1996a; Haykin & Li 1995; Khotanzad & Lu 1990; Narendra & Parthasarathy 1990; Nerrand et al. 1994).
  - Performance Criterion, Cost Function, Training Function. Computation of coefficients during training aims at finding a system whose operation is optimal w.r.t. some performance criterion which may be either qualitative, e.g. (subjective) quality of speech reconstruction, or quantitative, e.g. maximizing signal to noise ratio for spatial filtering. Goal: to define a positive *training function* which is s.t. a decrease of this function through modifications of coefficients of network leads to an improvement of performance of system (Bengio 1995; Haykin & Li 1995; Nerrand et al. 1994; Qin et al. 1992). In the case of non-adaptive training, training function is defined as a function of all data of training set (in such a case, usually termed as a *cost function*). The minimum of cost function corresponds to optimal performance of system. Training is an optimization procedure, conventionally using gradient-based methods.
    In case of adaptive training, impossible, in most instances, to define a time-independent cost function whose minimization leads to a system that is optimal w.r.t. performance criterion. Therefore, training function is time dependent. Modification of coefficients is computed continually from gradient of training function. Latter involves data pertaining to a time window of finite length, which shifts in time (sliding window) & coefficients are updated at each sampling time.
  - Recursive vs. Nonrecursive Algorithms. A nonrecursive algorithm employs a cost function (i.e. a training function defined on a fixed window), whereas a recursive algorithm makes use of a training function defined on a sliding window of data. An adaptive system must be trained by a recursive algorithm, whereas a non-adaptive system may be trained either by a nonrecursive or by a recursive algorithm (Nerrand et al. 1994).
  - Iterative vs. Noniterative Algorithms. An iterative algorithm performs coefficient modifications several times from a set of data pertaining to a given data window, a non-iterative algorithm makes only 1 (Nerrand et al. 1994). E.g., conventional LMS algorithm (2.31) is thus a recursive, non-iterative algorithm operating on a sliding window.
  - Supervised vs. Unsupervised Algorithms. A supervised learning algorithm performs learning by using a *teaching signal*, i.e. the desired output signal, while an unsupervised learning algorithm, as in blind signal processing, has no reference signal as a teaching input signal. An example of a supervised learning algorithm is the *delta rule*, while unsupervised learning algorithms are, e.g., the *reinforcement learning algorithm* & the *competitive rule ('winner takes all') algorithm*, whereby there is some sense of concurrency between elements of network structure (Bengio 1995; Haykin & Li 1995).
  - Pattern vs. Batch Learning. Updating network weights by *pattern learning* means that weights of network are updated immediately after each pattern is fed in. Other approach is to take all data as a whole batch, & network is not updated until entire batch of data is processed. This approach is referred to as *batch learning* (Haykin & Li 1995; Qin et al. 1992).
    It can be shown (Qin et al. 1992) that while considering feedforward networks (FFN), after 1 training sweep through all data, pattern learning is a 1st-order approximation of batch learning w.r.t. learning rate η. Therefore, FFN pattern learning approximately implements FFN batch learning after 1 batch interval. After multiple sweeps through training data, difference between FFN pattern learning & FFN batch learning is of order[14] $O(\eta^2)$.Therefore, for small training rates, FFN pattern learning approximately implements FFN batch learning after multiple sweeps through training data. For recurrent networks, weight updating slops for pattern learning & batch learning are different[15] (Qin et al. 1992). However, difference could also be controlled by learning rate η. Difference will converge to 0 as quickly as $\eta \to 0$[16] (Qin et al. 1992).
- Modularity Within Neural Networks. Hierarchical levels in neural network architectures are synapses, neurons, layers, & neural networks, & will be discussed in Chap. 5. Next step would be combinations of neural networks. In this case, consider modular neural networks. Modular neural networks are composed of a set of smaller subnetworks (modules), each performing a subtask of complete problem. To depict this problem, resource to case of linear adaptive filters described by a transfer function in z-domain $H(z)$ as

$$H(z) = \frac{\sum_{k=0}^{M} b(k) z^{-k}}{1 + \sum_{k=1}^{N} a(k) z^{-k}}. \tag{205}$$

[14] In fact, if data being processed exhibit highly stationary behavior, then average error calculated after FFN batch learning is very close to instantaneous error calculated after FFN pattern learning, e.g. speech data can be considered as being stationary within an observed frame. That forms the basis for use of various real-time & recursive learning algorithms, e.g. RTRL.

[15] (Qin et al. 1992) showed for feedforward networks, updated weights for both pattern learning & batch learning adapt at same slope (derivative $d_\eta w$) w.r.t. learning rate η. For recurrent networks, this is not the case.

[16] In which case, have a very slow learning process.

Can rearrange this function either in a cascaded manner as

$$H(z) = A \prod_{k=1}^{\max\{M,N\}} \frac{1 - \beta_k z^{-1}}{1 - \alpha_k z^{-1}}, \tag{206}$$

or in a parallel manner as

$$H(z) = \sum_{k=1}^{N} \frac{A_k}{1 - \alpha_k z^{-1}}, \tag{207}$$

where for simplicity, have assumed 1st-order poles & zeros of $H(z)$. A cascade realization of a general system is shown in Fig. 2.7: A cascaded realization of a general system, whereas a parallel realization of a general system is shown in Fig. 2.8: A parallel realization of a general system. Can also combine neural networks in these 2 configurations. An example of cascaded neural network is the so-called *pipelined recurrent neural network*, whereas an example of a parallel realization of a neural network is associative Gaussian mixture model, or winner takes all network. Taking into account that neural networks are nonlinear systems, talk about nested modular architectures instead of cascaded architectures. Nested neural scheme can be written as

$$F(W, X) = \Phi\left(\sum_n w_n \Phi\left(\sum_i v_i \Phi\left(\cdots \Phi\left(\sum_j u_j X_j\right)\cdots\right)\right)\right), \tag{208}$$

where Φ is a sigmoidal function. It corresponds to a multilayer network of units that sum their inputs with 'weights' $W = \{w_n, v_i, u_j, \ldots\}$ & then perform a sigmoidal transformation of this sum. Its motivation is: function

$$F(W, X) = \Phi\left(\sum_n w_n \Phi\left(\sum_j u_j X_j\right)\right) \tag{209}$$

can approximate arbitrarily well any continuous multivariate function (Funahashi 1989; Poggio & Girosi 1990).
Since use sigmoid 'squashing' activation functions, modular structures contribute to a general stability issue. Effects of a simple scheme of nested sigmoids are shown in Fig. 2.9: Effects of nesting sigmoid nonlinearities: 1st, 2nd, 3rd, & 4th pass. Pure nesting successively reduces range of output signal, bringing this composition of nonlinear functions to fixed point of employed nonlinearity for sufficiently many nested sigmoids.
Modular networks possess some advantages over over classical networks, since overall complex function is simplified & modules possibly do not have hidden units which speeds up training. Also, input data might be decomposable into subsets which can be fed to separate modules. Utilizing modular neural networks has not only computational advantages but also development advantages, improved efficiency, improved interpretability & easier hardware implementation. Also, there are strong suggestions from biology that modular structures are exploited in cognitive mechanisms (Fiesler & Beale 1997).

- Summary. Configurations of general adaptive systems have been provided, & prediction configuration has been introduced within this framework. Gradient-descent-based learning algorithms have then been developed for these configurations, with an emphasis on LMS algorithm. A thorough discussion of learning modes & learning parameters is given. Finally, modularity within neural networks has been addressed.

- Network Architectures for Prediction.
  - Perspective. Architecture, or structure, of a predictor underpins its capacity to represent dynamic properties of a statistically nonstationary discrete time input signal & hence its ability to predict or forecast some future value ⇒ this chapter provides an overview of available structures for prediction of discrete time signals.
  - Introduction. Basic building blocks of all discrete time predictors are adders, delayers, multipliers & for nonlinear case zero-memory nonlinearities. Manner in which these elements are interconnected describes architecture of a predictor. Foundations of linear predictors for statistically stationary signals are found in work of Yule (1927), Kolmogorov (1941), Wiener (1949). Later studies of Box & Jenkins (1970) & Makhoul (1975) were built upon these fundamentals. Such linear structures are very well established in digital signal processing & are classified either as finite impulse response (FIR) or infinite impulse response (IIR) digital filters (Oppenheim et al. 1999). FIR filters are generally realized without feedback, whereas IIR filters[17] utilize feedback to limit number of parameters necessary for their realization. Presence of feedback implies that consideration of stability underpins design of IIR filters. In statistical signal modeling, FIR filters are better known as moving average (MA) structures & IIR filters are named autoregressive (AR) or autoregressive moving average (ARMA) structures. Most straightforward version of nonlinear filter structures can easily be formulated by including a nonlinear operation in output stage of an FIR or an IIR filter. These represent simple examples of nonlinear autoregressive (NAR), nonlinear moving average (NMA) or nonlinear autoregressive moving average (NARMA) structures (Nerrand et al. 1993). Such filters have immediate application in prediction of discrete time random signals that arise from some nonlinear physical system, as for certain speech utterances. These filters, moreover, are strongly linked to single neuron neural networks.

[17] FIR filters can be represented by IIR filters, however, in practice it is not possible to represent an arbitrary IIR filter with an FIR filter of finite length.

Neuron, or node, is basic processing element within a neural network. Structure of a neuron is composed of multipliers, termed synaptic weights, or simply weights, which scale inputs, a linear combiner to form activation potential, & a certain zero-memory nonlinearity to model activation function. Different neural network architectures are formulated by combination of multiple neurons with various interconnections, hence term *connectionist modeling* (Rumelhart et al. 1986). Feedforward neural networks, as for FIR/MA/NMA filters, have no feedback within their structure. Recurrent neural networks, on the other hand, similarly to IIR/AR/NAR/NARMA filters, exploit feedback & hence have much more potential structural richness. Such feedback can either be local to neurons or global to network (Haykin 1999b; Tsoi & Back 1997). When inputs to a neural network are delayed versions of a discrete time random input signal correspondence between architectures of nonlinear filters & neural networks is evident.

From a biological perspective (Marmarelis 1989), *prototypical* neuron is composed of a cell body (soma), a tree-like element of fibres (dendrites) & a long fibre (axon) with sparse branches (collaterals). Axon is attached to soma at the *axon hillock*, &, together with its collaterals, ends at synaptic terminals (boutons), which are employed to pass information onto their neurons through *synaptic junctions*. Soma contains nucleus & is attached to trunk of dendritic tree from which it receives incoming information. Dendrites are conductors of input information to soma, i.e. input ports, & usually exhibit a high degree of arborisation.

Possible architectures for nonlinear filters or neural networks are manifold. State-space representation from system theory is established for linear systems (Kailath 1980; Kailath et al. 2000) & provides a mechanism for representation of structural variants. An insightful canonical form for neural networks is provided by Nerrand et al. (1993), by exploitation of state-space representation which facilitates a unified treatment of architectures of neural networks.[18]

- Overview. An explanation of concept of prediction of a statistically stationary discrete time random signal. Building blocks for realization of linear & nonlinear predictors are then discussed. These same building blocks are also shown to be basic elements necessary for realization of a neuron. Emphasis is placed upon particular zero-memory nonlinearities used in output of nonlinear filters & activation functions of neurons.

  An aim: to highlight correspondence between structures in nonlinear filtering & neural networks, so as to remove apparent boundaries between work of practitioners in control, signal processing, & neural engineering. Conventional linear filter models for discrete time random signals are introduced &, with aid of statistical modeling, motivate structures for linear predictors, their nonlinear counterparts are then developed.

  A feedforward neural network is next introduced in which nonlinear elements are distributed throughout structure. To employ such a network as a predictor, shown: short-term memory is necessary, either at input or integrated within network. Recurrent networks follow naturally from feedforward neural networks by connecting output of network to its input. Implications of local & global feedback in neural networks are also discussed.

  Role of state-space representation in architectures for neural networks is described & this leads to a canonical representation.

- Prediction. A real discrete time random signal $\{y(k)\}$, where k: *discrete time index*, is most commonly obtained by sampling some analogue measurement. Voice of an individual, e.g., is translated from pressure variation in air into a continuous time electrical signal by means of a microphone & then converted into a digital representation by an analogue-to-digital converter. Such discrete time random signals have statistics that are time-varying, but on a short-term basics, statistics may be assumed to be time invariant.

  Principle of prediction of a discrete time signal is represented in Fig. 3.1: Basic concept of linear prediction & forms basis of linear predictive coding (LPC) which underlies many compression techniques. Value of signal $y(k)$ is predicted on basis of a sum of p past values, i.e., $y(k-1), y(k-2), \ldots, y(k-p)$, weighted, by coefficients a_i, $i = 1, \ldots, p$, to form a prediction, $\hat{y}(k)$. Prediction error $e(k)$ thus become

$$e(k) = y(k) - \hat{y}(k) = y(k) - \sum_{i=1}^{p} a_i y(k-i). \tag{210}$$

  Estimation of parameters a_i is based upon minimizing some function fo error, most convenient form being mean square error $E[e^2(k)]$, where $E[\cdot]$ denotes *statistical expectation operator*, & $\{y(k)\}$ is assumed to be statistically wide sense stationary,[19] with zero mean (Papoulis 1984). A fundamental advantage of mean square error criterion is so-called *orthogonality condition*, which implies

$$E[e(k), y(k-j)] = 0, \;\; j = 1, 2, \ldots, p, \tag{211}$$

  is satisfied only when a_i, $i = 1, \ldots, p$, take on their optimal values. As a consequence of of (211) & linear structure of predictor, optimal weight parameters may be found from a set of linear equations, named the *Yule–Walker equations* (Box & Jenkins 1970), $\mathbf{R}_{yy}\mathbf{a} = \mathbf{r}_{yy}$ where $\mathbf{R}_{yy} = (r_{yy}(|i-j|))_{i,j=1}^{p}$, $r_{yy}(\tau) = E[y(k)y(k+\tau)]$ is value of autocorrelation function of $\{y(k)\}$ at lag τ. These equations may be equivalently written in matrix form as $\mathbf{R}_{yy}\mathbf{a} = \mathbf{r}_{yy}$ where $\mathbf{R}_{yy} \in \mathbb{R}^{p\times p}$: *autocorrelation matrix*, $\mathbf{a}, \mathbf{r}_{yy} \in \mathbb{R}^p$ are, resp., parameter vector of predictor & crosscorrelation vector. Toeplitz symmetric structure of $\mathbf{R}_{yy}$ is exploited in Levinson–Durbin algorithm (Hayes 1997) to solve for optimal parameters in $O(p^2)$ operations. Quality of prediction is judged by minimum mean square error (MMSE), which is calculated from $E[e^2(k)]$ when weight parameters of predictor take on their optimal values. The MMSE is calculated from $r_{yy}(0) - \sum_{i=1}^{p} a_i r_{yy}(i)$.

[18] ARMA models also have a canonical (up to an invariant) representation.

[19] Wide sense stationarity implies that mean is constant, autocorrelation function is only a function of time lag & variance is finite.

Real measurements can only be assumed to be locally wide sense stationary & therefore, in practice, autocorrelation function values must be estimated from some finite length measurement in order to employ (3.3). A commonly used, but statistically biased & low variance (Kay 1993), autocorrelation estimator for application to a finite length N measurement, $\{y(0), y(1), \ldots, y(N-1)\}$, is given by

$$\hat{r}_{yy}(\tau) = \frac{1}{N} \sum_{k=0}^{N-\tau-1} y(k)y(k+\tau), \ \tau = 0, 1, \ldots, p. \tag{212}$$

These estimates would then replace exact values in (3.3) from which weight parameters of predictor are calculated. This procedure, however, needs to be repeated for each new length N measurement, & underlies operation of a block-based predictor.

A 2nd approach to estimation of weight parameters $\mathbf{a}(k)$ of a predictor is sequential, adaptive or learning approach. Estimates of weight parameters are refined at each sample number k on basis of new sample $y(k)$ & prediction error $e(k)$. This yields an update equation of form $\hat{\mathbf{a}}(k+1) = \hat{a}(k) + \eta f(e(k), \mathbf{y}(k))$, $k \geq 0$, where η is termed adaptation gain, $f(\cdot)$ is some function dependent upon particular learning algorithm, whereas $\hat{\mathbf{a}}(k), \mathbf{y}(k)$ are, resp., estimated weight vector & predictor input vector. Without additional prior knowledge, zero or random values are chosen for initial values of weight parameters in (3.6), i.e. $\hat{a}_i(0) = 0$, or n_i, $i = 1, \ldots, p$, where n_i: a random variable drawn from a suitable distribution. Sequential approach to estimation of weight parameters is particularly suitable for operation of predictors in statistically nonstationary environments. Both block & sequential approach to estimation of weight parameters of predictors can be applied to linear & nonlinear structure predictors.

- Building Blocks. In Fig. 3.2: Building blocks of predictors: (a) delayer, (b) adder, (c) multiplier the basic building blocks of discrete time predictors are shown. A simple delayer has input $y(k)$ & output $y(k-1)$, note: sampling period is normalized to unity. From linear discrete time system theory, delay operation can also be conveniently represented in $\mathcal{Z}$-domain notation as the z^{-1} operator[20] (Oppenheim et al. 1999). An adder, or sumer, simply produces an output which is the sum of all the components at its input. A multiplier, or scaler, used in a predictor generally has 2 inputs & yields an output which is product of 2 inputs. Manner in which delayers, adders, & multipliers are interconnected determines architecture of linear predictors. These architectures, or structures, are shown in block diagram form in the ensuing sections.

  To realize nonlinear filters & neural networks, zero-memory nonlinearities are required. 3 zero-memory nonlinearities, as given in Haykin (1999b), with inputs $v(k)$ & outputs $\Phi(k)$ are described by following operations:

  * Threshold:

$$\Phi(v(k)) = \begin{cases} 0 & v(k) < 0, \\ 1 & v(k) \geq 0, \end{cases} \tag{213}$$

  * Piecewise-linear:

$$\Phi(v(k)) = \begin{cases} 0 & v(k) \leq -\frac{1}{2}, \\ v(k) & -\frac{1}{2} < v(k) < \frac{1}{2}, \\ 1 & v(k) \geq \frac{1}{2}, \end{cases} \tag{214}$$

  * Logistic:

$$\Phi(v(k)) = \frac{1}{1 + e^{-\beta v(k)}}, \ \beta \geq 0. \tag{215}$$

  The most commonly used nonlinearity is logistic function since it is continuously differentiable & hence facilities analysis of operation of neural networks. This property is crucial in development of 1st- & 2nd-order learning algorithms. When $\beta \to \infty$, moreover, logistic function becomes unipolar threshold function. Logistic function is a strictly nondecreasing function which provides for a gradual transition from linear to nonlinear operation. Inclusion of such a zero-memory nonlinearity in output stage of structure of a linear predictor facilitates design of nonlinear predictors.

  Threshold nonlinearity is well-established in neural network community as it was proposed in seminal work of McCulloch & Pitts (1943), however, it has a discontinuity at the origin. Piecewise-linear model, on the other hand, operates in a linear manner for $|v(k)| < \frac{1}{2}$ & otherwise saturates at zero or unity. Although easy to implement, neither of these zero-memory nonlinearities facilitates analysis of operation of nonlinear structures, because of badly behaved derivatives. Neural networks are composed of basic processing units named neurons, or nodes, in analogy with biological elements present within human brain (Haykin 1999b). Basic building blocks of such artificial neurons are identical to those for nonlinear predictors. Block diagram of an artificial neuron[21] is shown in Fig. 3.3: Structure of a neuron for prediction. In context fo prediction, inputs are assumed to be delayed versions of $y(k)$, i.e., $y(k-i)$, $i = 1, \ldots, p$. There is also a constant bias input with unity value. These inputs are then passed through $(p+1)$ multipliers for scaling. In neural network parlance, this operation in scaling inputs corresponds to role of synapses in physiological neurons. A sumer then linearly combines (in fact this is an affine transformation) these scaled inputs to form an output $v(k)$ which is termed induced local field or activation potential of neuron. Save for presence of bias input, this output is identical to output of a

[20] z^{-1} operator is a delay operator s.t. $\mathcal{Z}(y(k-1)) = z^{-1}\mathcal{Z}(y(k))$.

[21] Term 'artificial neuron' will be replaced by 'neuron' in sequel.

linear predictor. This component of neuron, from a biological perspective, is termed synaptic part (Rao & Gupta 1993). Finally, $v(k)$ is passed through a zero-memory nonlinearity to form output $\hat{y}(k)$. This zero-memory nonlinearity is called (nonlinear) activation function of a neuron & can be referred to as somatic part (Rao & Gupta 1993). Such a neuron is a static mapping between its input & output (Hertz et al. 1991) & is very different from dynamic form of a biological neuron. Synergy between nonlinear predictors & neurons is therefore evident. Structural power of neural networks in prediction results, however, from interconnection of many such neurons to achieve overall predictor structure in order to distribute underlying nonlinearity.

- Linear Filters. In digital signal processing & linear time series modeling, linear filters are well-established (Hayes 1997; Oppenheim et al. 1999) & have been exploited for structures of predictors. Essentially, there are 2 families of filters: those without feedback, for which their output depends only upon current & past input values; & those with feedback, for which their output depends both upon input values & past outputs. Such filters are best described by a constant coefficient difference equation, most general form of which is given by

$$y(k) = \sum_{i=1}^{p} a_i y(k-i) + \sum_{j=0}^{q} b_j e(k-j), \tag{216}$$

where $y(k)$: output, $e(k)$: input,[22] $a_i, i = 1, \ldots, p$, are (AR) feedback coefficients & $b_j, j = 0, 1, \ldots, q$, are (MA) feedforward coefficients. In causal systems, (216) is satisfied for $k \geq 0$ & initial conditions $y(i), i = -1, -2, \ldots, -p$, are generally assumed to be zero. Block diagram for filter represented by (216) is shown in Fig. 3.4: Structure of an autoregressive moving average filter ARMA(p, q). Such a filter is termed an autoregressive moving average ARMA(p, q) filter, where p is order of autoregressive, or feedback, part of structure, & q: order of moving average, or feedforward, element of structure. Due to feedback present within this filter, impulse response, namely values of $y(k), k \geq 0$, when $e(k)$ is a discrete time impulse, is infinite in duration $\Rightarrow$ such a filter is termed an infinite impulse response (IIR) filter within field of digital signal processing.

General form of (216) is simplified by removing feedback terms to yield

$$y(k) = \sum_{j=0}^{q} b_j e(k-j). \tag{217}$$

Such a filter is termed moving average MA(q) & has a finite impulse response, which is identical to parameters $b_j, j = 0, 1, \ldots, q$. In digital signal processing $\Rightarrow$ such a filter is named a finite impulse response (FIR) filter. Similarly, (217) is simplified to yield an autoregressive AR(p) filter

$$y(k) = \sum_{i=1}^{p} a_i y(k-i) + e(k), \tag{218}$$

which is also termed an IIR filter. Filter described by (3.12) is basis for modeling speech production process (Makhoul 1975). Presence of feedback within AR(p) & ARMA(p, q) filters implies that selection of $a_i, i = 1, \ldots, p$, coefficients must be s.t. filters are BIBO stable, i.e. a bounded output will result from a bounded input (Oppenheim et al. 1999).[23] Most straightforward way to test stability is to exploit $\mathcal{Z}$-domain representation of transfer function of filter represented by (3.10):

$$H(z) = \frac{Y(z)}{E(z)} = \frac{b_0 + b_1 z^{-1} + \cdots + b_q z^{-q}}{1 - a_1 z^{-1} - \cdots - a_p z^{-p}} = \frac{N(z)}{D(z)}. \tag{219}$$

To guarantee stability, p roots of denominator polynomial of $H(z)$, i.e. values of z for which $D(z) = 0$, poles of transfer function, must lie within unit circle in z-plane, $|z| < 1$. In digital signal processing, cascade, lattice, parallel & wave filters have been proposed for realization of transfer function described by (3.13) (Oppenheim et al. 1999). For prediction applications, however, direct form, as in Fig. 3.4: Structure of an autoregressive moving average filter ARMA(p, q), & lattice structures are most commonly employed.

In signal modeling, rather than being deterministic, input $e(k)$ to filter in (3.10) is assumed to be an independent identically distributed (i.i.d.) discrete time random signal. This input is an integral part of a rational transfer function discrete time signal model. Filtering operations described by (3.10)–(3.12), together with such an i.i.d. input with prescribed finite variance σ_e^2, represent resp., ARMA(p, q), MA(q), AR(p) signal models. Autocorrelation function of input $e(k)$ is given by $\sigma_e^2 \delta(k) \Rightarrow$ its power spectral density (PSD) is $P_e(f) = \sigma_e^2$ for all f. PSD of an ARMA model is therefore:

$$P_y(f) = |H(f)|^2 P_e(f) = \sigma_e^2 |H(f)|^2, \ f \in \left(-\frac{1}{2}, \frac{1}{2}\right], \tag{220}$$

where f: normalized frequency. Quantity $|H(f)|^2$: magnitude squared frequency domain transfer function found from (3.13) by replacing $z = e^{j2\pi f}$. Role of filter is therefore to shape PSD of driving noise to match PSD of physical system. Such an ARMA model is well motivated by the Wold decomposition, which states: any stationary discrete time random signal can be split into sum of uncorrelated deterministic & random components. In fact, an ARMA(∞, ∞) model is sufficient to model any stationary discrete time random signal (Theiler et al. 1993).

[22] Notice $e(k)$ is used as filter input, rather than $x(k)$, for consistency with later sections on prediction error filtering.

[23] This type of stability is commonly denoted as BIBO stability in contrast to other types of stability, e.g. global asymptotic stability (GAS).

- Nonlinear Predictors. If a measurement is assumed to be generated by an ARMA(p,q) model, optimal conditional mean predictor of discrete time random signal $\{y(k)\}$

$$\hat{y}(k) = E[y(k)|y(k-1), y(k-2), \ldots, y(0)] \tag{221}$$

is given by

$$\hat{y}(k) = \sum_{i=1}^{p} a_i y(k-i) + \sum_{j=1}^{q} b_j \hat{e}(k-j), \tag{222}$$

where residuals $\hat{e}(k-j) = y(k-j) - \hat{y}(k-j)$, $j = 1, \ldots, q$. Notice predictor described by (3.16) utilizes past value of actual measurement, $y(k-i)$, $i = 1, \ldots, p$; whereas estimates of unobservable input signal $e(k-j)$, $j = 1, \ldots, q$, are formed as difference between actual measurements & past predictions. Feedback present within (3.16), which is due to residuals $\hat{e}(k-j)$, results from presence of MA(q) part of model for $y(k)$ in (3.10). No information is available about $e(k) \Rightarrow$ it cannot form part of prediction. On this basis, simplest form of nonlinear autoregressive moving average NARMA(p,q) model takes form,

$$y(k) = \Theta\left(\sum_{i=1}^{p} a_i y(k-i) + \sum_{j=1}^{q} b_j e(k-j)\right) + e(k), \tag{223}$$

where $\Theta(\cdot)$ is an unknown differentiable zero memory nonlinear function. Notice $e(k)$ is not included within $\Theta(\cdot)$ as it is unobservable. Term NARMA(p,q) is adopted to define (3.17), since save for $e(k)$, output of an ARMA(p,q) model is simply passed through zero-memory nonlinearity $\Theta(\cdot)$.
Corresponding NARMA(p,q) predictor is given by

$$\hat{y}(k) = \Theta\left(\sum_{i=1}^{p} a_i y(k-i) + \sum_{j=1}^{q} b_j \hat{e}(k-j)\right), \tag{224}$$

where residuals $\hat{e}(k-j) = y(k-j) - \hat{y}(k-j)$, $j = 1, \ldots, q$. Equivalently, simplest form of nonlinear autoregressive NAR(p) model is described by

$$y(k) = \Theta\left(\sum_{i=1}^{p} a_i y(k-i)\right) + e(k) \tag{225}$$

& its associated predictor is

$$\hat{y}(k) = \Theta\left(\sum_{i=1}^{p} a_i y(k-i)\right). \tag{226}$$

Associated structures for predictors described by (3.18) & (3.20) are shown in Fig. 3.5: Structure of NARMA(p,q) & NAR(p) predictors. Feedback is present within NARMA(p,q) predictor, whereas NAR(p) predictor is an entirely feedforward structure. Structures are simply those of linear filters described in Sect. 3.6 with incorporation of a zero-memory nonlinearity.
In control applications, most generally, NARMA(p,q) models also include so-called *exogeneous inputs* $u(k-s)$, $s = 1, \ldots, r$, & following approach of (3.17) & (3.19) simplest example takes form

$$y(k) = \Theta\left(\sum_{i=1}^{p} a_i y(k-i) + \sum_{j=1}^{q} b_j e(k-j) + \sum_{s=1}^{r} c_s u(k-s)\right) + e(k), \tag{227}$$

& is termed a nonlinear autoregressive moving average with exogeneous inputs model, NARMAX(p,q,r), with associated predictor

$$\hat{y}(k) = \Theta\left(\sum_{i=1}^{p} a_i y(k-i) + \sum_{j=1}^{q} b_j e(k-j) + \sum_{s=1}^{r} c_s u(k-s)\right), \tag{228}$$

which again exploits feedback (Chen & Billings 1989; Siegelmann et al. 1997). This is most straightforward form of nonlinear predictor structure derived from linear filters.

- Feedforward Neural Networks: Memory Aspects. See also [NP23]: *A rigorous framework for the mean field limit of multilayer neural networks*. Nonlinearity present in predictors described by (3.18), (3.20), & (3.22) only appears at overall output, in same manner as in simple neuron depicted in Fig. 3.3. These predictors could therefore be referred to as single neuron structures. More generally, however, in neural neuworks, nonlinearity is distributed through certain layers, or stages, of processing.
In Fig. 3.6: Multilayer feedforward neural network a multiplayer feedforward neural network is shown. Measurement samples appear at input layer, & output prediction is given from output layer. To be consistent with problem of prediction of a single discrete time random signal, only a single output is assumed. In between, there exist so-called *hidden layers*. Notice outputs of each layer are only connected to inputs of adjacent layer. Nonlinearity inherent in network is due to overall action of all activation functions of neurons within structure.

In problem of prediction, nature of inputs to multilayer feedforward neural network must capture something about time evolution of underlying discrete time random signal. Simplest situation is for inputs to be time-delayed versions of signal, i.e. $y(k-i), i = 1, \ldots, p$, & is commonly termed a tapped delay line or delay space embedding (Mozer 1993). Such a block of inputs provides network with a short-term memory of signal. At each time sample k, inputs of network only see effect of 1 sample of $y(k)$, & Mozer (1994) terms this a high-resolution memory. Overall predictor can then be represented as

$$\hat{y}(k) = \Phi(y(k-1), y(k-2), \ldots, y(k-p)), \tag{229}$$

where Φ represents nonlinear mapping of neural network.
Other forms of memory for network include: samples with nonuniform delays, i.e. $y(k-i), i = \tau_1, \tau_2, \ldots, \tau_p$; exponential, where each input to network, denoted $\tilde{y}_i(k), i = 1, \ldots, p$, is calculated recursively from $\tilde{y}_i(k) = \mu_i \tilde{y}_i(k-1) + (1-\mu_i) y_i(k)$, where $\mu_i \in [-1, 1]$: *exponential factor* which controls depth (Mozer 1993) or time spread of memory & $y_i(k) = y(k-i), i = 1, \ldots, p$. A delay line memory is therefore termed high-resolution low-depth, while an exponential memory is low-resolution but high-depth. In continuous time, Principe et al. (1993) proposed Gamma memory, which provided a method to trade resolution for depth. A discrete time version of this memory is described by

$$\tilde{y}_{\mu,j}(k) = \mu \tilde{y}_{\mu,j}(k-1) + (1-\mu)\tilde{y}_{\mu,j-1}(k-1), \tag{230}$$

where index j is included because necessary to evaluate (3.24) for $j = 0, 1, \ldots, i$, where i: delay of particular input to network & $\tilde{y}_{\mu,-1}(k) = y(k+1)$, $\forall k \geq 0$, & $\tilde{y}_{\mu,j}(0) = 0$, $\forall j \geq 0$. Form of equation is, moreover, a convex mixture. Choice of μ controls trade-off between depth & resolution; small μ provides low-depth & high-resolution memory, whereas high μ yields high-depth & low-resolution memory.
Restricting memory in a multilayer feedforward neural network to input layer may, however, lead to structures with an excessively large number of parameters. Wan (1993) therefore utilizes a time-delay network where memory is integrated within each layer of network. Fig. 3.7: Structure of neuron of a time delay neural network shows form of a neuron within a time-delay network, in which multipliers of basic neuron of Fig. 3.3 are replaced by FIR filters to capture dynamics of input signals. Networks formed from such neurons are functionally equivalent to networks with only memory at their input but generally have many fewer parameters, which is beneficial for learning algorithms.
Integration of memory into a multilayer feedforward network yields structure for nonlinear prediction $\Rightarrow$ Clear: such networks belong to class of nonlinear filters.

- Recurrent Neural Networks: Local & Global Feedback. In Fig. 3.6, inputs to network are drawn from discrete time signal $y(k)$. Conceptually, straightforward to consider connecting delayed versions of output $\hat{y}(k)$ of network to its input. Such connections, however, introduce feedback into network & therefore stability of such networks must be considered, this is a particular focus of later parts of this book. Provision of feedback, with delay, introduces memory to network & so is appropriate for prediction.
  Feedback within recurrent neural networks can be achieved in either a local or global manner. An example of a recurrent neural network is shown in Fig. 3.8: Structure of a recurrent neural network with local & global feedback with connections for both local & global feedback. Local feedback is achieved by introduction of feedback within hidden layer, whereas global feedback is produced by connection of network output to network input. Inter-neuron connections can also exist in hidden layer, but they are not shown in Fig. 3.8. Although explicit delays are not shown in feedback connections, they are assumed to be present within neurons in order that network is realizable. Operation of a recurrent network predictor that employs global feedback can now be represented by

$$\hat{y}(k) = \Phi(y(k-1), y(k-2), \ldots, y(k-p), \hat{e}(k-1), \ldots, \hat{e}(k-q)), \tag{231}$$

  where again $\Phi(\cdot)$ represents nonlinear mapping of neural network &

$$\hat{e}(k-j) = y(k-j) - \hat{y}(k-j), \ j = 1, \ldots, q. \tag{232}$$

  A taxonomy of recurrent neural networks architectures is presented by Tsoi & Back (1997). Choice of structure depends upon dynamics of signal, learning algorithm & ultimately prediction performance. There is, unfortunately, no hard & fast rule as to best structure to use for a particular problem (Personnaz & Dreyfus 1998).
- State-Space Representation & Canonical Form. Structures in this chapter have been developed on basis of difference equation representations. Simple nonlinear predictors can be formed by placing a zero-memory nonlinearity within output stage of a classical linear predictor. In this case, nonlinearity is restricted to output stage, as in a single layer neural network realization. On the other hand, if nonlinearity is distributed through many layers of weighted interconnections, concept of neural networks is fully exploited & more powerful nonlinear predictors may ensue. For purpose of prediction, memory stages may be introduced at input or within network. Most powerful approach is to introduce feedback & to unify feedback networks. Nerrand et al. (1994) proposed an insightful canonical state-space representation:

  > Any feedback network can be cast into a canonical form that consists of a feedforward (static) network:
  > whose outputs are the outputs of the neurons that have desired values, & the values of the state variables,
  > whose inputs are the inputs of the network & the values of the state variables, the latter being delayed by 1 time unit.

  Note: in prediction of a single discrete-time random signal, network will have only 1 output neuron with a predicted value. For a dynamical system, e.g. a recurrent neural network for prediction, state represents *a set of quantities that*

summarizes all information about past behavior of system that is needed to uniquely describe its future behavior, except for purely external effects arising from applied input (excitation) (Haykin 1999b).
Note: whereas always possible to rewrite a nonlinear input-output model in a state-space representation, an input-output model equivalent to a given state-space model might not exist &, if it does, it is surely of higher order. Under fairly general conditions of observability of a system, however, an equivalent input-output model does exist but it may be of high order. A state-space model is likely to have lower order & require a smaller number of past inputs &, hopefully, a smaller number of parameters. This has fundamental importance when only a limited number of data samples is available. Takens' theorem (Wan 1993) implies: for a wide class of deterministic systems, there exists a diffeomorphism (1-1 differential mapping) between a finite window of time series & underlying state of dynamic system which gives rise to time series. A neural network can therefore approximate this mapping to realize a predictor.
In Fig. 3.9: Canonical form of a recurrent neural network for prediction, general canonical form of a recurrent neural network is represented. If state is assumed to contain N variables, then a state vector is defined as $\mathbf{s}(k) = [s_1(k), \ldots, s_N(k)]^\top$, & a vector of p external inputs is given by $\mathbf{y}(k-1) = [y(k-1), y(k-2), \ldots, y(k-p)]^\top$. State evolution & output equations of recurrent network for prediction are given, resp., by

$$\mathbf{s}(k) = \varphi(\mathbf{s}(k-1), \mathbf{y}(k-1), \hat{y}(k-1)), \tag{233}$$
$$\hat{y}(k) = \psi(\mathbf{s}(k-1), \mathbf{y}(k-1), \hat{y}(k-1)), \tag{234}$$

where φ, Ψ represent general classes of nonlinearities. Particular choice of N minimal state variables is not unique, therefore several canonical forms[24] exist. A procedure for determination of N for an arbitrary recurrent neural network is described by Nerrand et al. (1994). NARMA & NAR predictors described by (3.18) & (3.20), however, follow naturally from canonical state-space representation because elements of state vector are calculated from inputs & outputs of network. Moreover, even if recurrent neural network contains local feedback & memory, still possible to convert network into above canonical form (Personnaz & Dreyfus 1998).

- Summary. Aim of this chapter: to show commonality between structures of nonlinear filters & neural networks. Basic building blocks for both structures have been shown to be adders, delayers, multipliers, & zero-memory nonlinearities, & manner in which these elements are interconnected defines particular structure. Theory of linear predictors, for stationary discrete time random signals, which are optimal in minimum mean square prediction error sense, has been shown to be well established. Structures of linear predictors have also been demonstrated to be established in signal processing & statistical modeling. Nonlinear predictors have then been developed on basis of defining dynamics of a discrete time random signal by a nonlinear model. In essence, in their simplest form these predictors have 2 stages: a weighted linear combination of iputs &/or past outputs, as for linear predictors, & a 2nd stage defined by a zero-memory nonlinearity.
Neuron, fundamental processing element in neural networks, has been introduced. Multilayer feedforward neural networks have been introduced in which nonlinearity is distributed throughout structure. To operate in a prediction mode, some local memory is required either at input or integral to network structure. Recurrent neural networks have then been formulated by connecting delayed versions of global output to input of a multilayer feedforward structure; or by introduction of local feedback within network. A canonical state-space form has been used to represent an arbitrary neural network.

- Activation Functions Used in Neural Networks.
  - Perspective. Choice of nonlinear activation function has a key influence on complexity & performance of artificial neural networks, note: term *neural network* will be used interchangeably with term *artificial neural network*. Brief introduction to activation functions given in Chap. 3 is therefore extended. Although sigmoidal nonlinear activation functions are most common choice, there is no strong a priori justification why models based on such functions should be preferred to others.
  ⇒ introduce neural networks as universal approximators of functions & trajectories, based upon Kolmogorov universal approximation theorem, which is valid for both feedforward & recurrent neural networks. From these universal approximation properties, then demonstrate need for a sigmoidal activation function within a neuron. To reduce computational complexity, approximations to sigmoid function are further discussed. Use of nonlinear activation functions suitable for hardware realization of neural networks is also considered.
  For rigor, extend analysis to complex activation functions & recognize that a suitable complex activation function is a Möbius transformation. In that context, a framework for rigorous analysis of some inherent properties of neural networks, e.g. fixed points, nesting & invertibility based upon theory of modular groups of Möbius transformations is provided.
  All relevant defs, thms, & other mathematical terms are given in Appendices B–C.
  - Introduction. A century ago, a set of 23 (originally) unsolved problems in mathematics was proposed by David Hilbert (Hilbert 1901–1902). In his lecture 'Mathematische Probleme' at 2nd International Congress of Mathematics held in Paris in 1900, he presented 10 of them. These problems were designed to serve as examples for kinds of problems whose solutions would lead to further development of disciplines in mathematics. His 13th problem concerned solutions of polynomial equations. Although his original formulation dealt with properties of solution of 7th degree algebraic equation,[25] this problem can be restated as: *Prove that there are continuous functions of n variables, not representable by a superposition of continuous functions of* $(n-1)$ *variables.* I.e., could a general algebraic equation of a high degree be expressed by sums

[24] These canonical forms stem from Jordan canonical forms of matrices & companion matrices. Notice: in fact $\hat{y}(k)$ is a state variable but shown separately to emphasize its role as predicted output.

[25] Hilbert conjectured: roots of equation $x^7 + ax^3 + bx^2 + cx + 1 = 0$ as functions of coefficients a, b, c are not representable by sums & superpositions of functions of 2 coefficients, or 'Show impossibility of solving general 7th degree equation by functions of 2 variables.'

& compositions of single-variable functions?[26] In 1957, KOLMOGOROV showed: conjecture of HILBERT was not correct (Kolmogorov 1957).

Kolmogorov's theorem is a general representation theorem stating: any real-valued continuous function f defined on an n-dimensional cube I^n, $n \geq 2$, can be represented as

$$f(x_1, \ldots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \psi_{pq}(x_p) \right), \tag{235}$$

where $\Phi_q(\cdot)$, $q = 1, \ldots, 2n+1$, & $\psi_{pq}(\cdot)$, $p = 1, \ldots, n, q = 1, \ldots, 2n+1$, are typically nonlinear continuous functions of 1 variable.

For a neural network representation, this means: an activation function of a neuron has to be nonlinear to form a universal approximator. This also means: every continuous function of many variables can be represented by a 4-layered neural network with 2 hidden layers & an input & output layer, whose hidden units represent mappings Φ, ψ. However, this does not mean: a network with 2 hidden layers necessarily provides an accurate representation of function f. In fact, functions ψ_{pq} of KOLMOGOROV's theorem are quite often highly nonsmooth, whereas for a neural network we want smooth nonlinear activation functions, as is required by gradient-descent learning algorithms (Poggio & Girosi 1990). Vitushkin (1954) showed: there are functions of > 1 variable which do not have a representation by superpositions of differentiable functions (Beiu 1998). Important questions about KOLMOGOROV's representation are therefore existence, constructive proofs & bounds on size of a network needed for approximation.

KOLMOGOROV's representation has been improved by several authors. Sprecher (1965) replaced functions ψ_{pq} in KOLMOGOROV REPRESENTATION by $\lambda^{pq}\psi_q$, where λ is a constant, & ψ_q are monotonic increasing functions which belong to class of Lipschitz functions. Lorentz (1976) showed: functions Φ_q can be replaced by only 1 function Φ. Hecht-Nielsen reformulated this result for MLPs so that they are able to approximate any function. In this case, functions ψ are nonlinear activation functions in hidden layers, whereas functions Φ are nonlinear activation functions in output layer. Functions Φ, ψ are found, however, to be generally highly nonsmooth. Further, in Katsuura & Sprecher (1994), function ψ is obtained through a graph that is limit point of an iterated composition of contraction mappings on their domain.

In applications of neural networks for universal approximation, existence proof for approximation by neural networks is provided by KOLMOGOROV's theorem, which is neural network community was 1st recognized by Hecht-Nielsen (1987) & Lippmann (1987). 1st constructive proof of neural networks as universal approximators was given by Cybenko (1989). Most of analyzes rest on denseness property of nonlinear functions that approximate desired function in space in which desired function is defined. In CYBENKO's results, e.g., if σ is a continuous discriminatory function,[27] then finite sums of form

$$g(\mathbf{x}) = \sum_{i=1}^{N} w_i \sigma(\mathbf{a}_i^\top \mathbf{x} + b_i), \tag{237}$$

where $w_i, b_i, i = 1, \ldots, N$, are coefficients, are dense in space of continuous functions defined on $[0,1]^n$. Following classical approach to approximation, this means: given any continuous function f defined on $[0,1]^N$ & any $\varepsilon > 0$, there is a $g(\mathbf{x})$ given by (4.2) for which $|g(\mathbf{x}) - f(\mathbf{x})| < \varepsilon$, $\forall \mathbf{x} \in [0,1]^N$. CYBENKO then concludes: any bounded & measurable sigmoidal function is discriminatory (Cybenko 1989), & a 3-layer neural network with a sufficient number of neurons in its hidden layer can represent an arbitrary function (Beiu 1998; Cybenko 1989).

Funahashi (1989) extended this to include sigmoidal functions so that any continuous function is approximately realizable by 3-layer networks with bounded & monotonically increasing activation functions within hidden units. Hornik et al. (1989) showed: output function does not have to be continuous, & they also proved: a neural network can approximate simultaneously both a function & its derivative (Hornik et al. 1990). Hornik (1990) further showed: activation function has to be bounded & nonconstant (but not necessarily continuous), Kurkova (1992) revealed existence of an approximate representation of functions by superposition of nonlinear functions within constraints of neural networks. Leshno et al. (1993) relaxed condition for activation function to be 'locally bounded piecewise continuous' (i.e., iff activation function is not a polynomial). This result encompasses most of activation functions commonly used.

Funahashi & Nakamura (1993), in their article 'Approximation of dynamical systems by continuous time recurrent neural networks', proved: universal approximation theorem also holds for trajectories & patterns & for recurrent neural networks. Li (1992) also showed: recurrent neural networks are universal approximators. Some recent results, moreover, suggest: 'smaller nets perform better' (Elsken 1999), which recommends recurrent neural networks, since a small-scale RNN has dynamics that can be achieved only by a large scale feedforward neural network. Sprecher (1993) considered problem of dimensionality of neural networks & demonstrated: number of hidden layers is independent of number of input variables N. Barron (1993) described spaces of functions that can be approximated by relaxed algorithm of Jones using functions computed by single-hidden-layer networks or perceptrons. Attali & Pages (1997) provided an approach based upon Taylor series expansion. Maiorov & Pinkus have given lower bounds for neural network based approximation (Maiorov & Pinkus 1999). Approximation ability of neural networks has also been rigorously studied in Williamson & Helmke (1995).

[26] E.g., function xy is a composition of functions $g(\cdot) = \exp(\cdot), h(\cdot) = \log \cdot$, therefore $xy = e^{\log x + \log y} = g(h(x) + h(y))$ (Gorban & Wunsch 1998).

[27] $\sigma(\cdot)$ is discriminatory if for a Borel measure μ on $[0,1]^N$, $\int_{[0,1]^N} \sigma(\mathbf{a}^\top \mathbf{x} + b)\, \mathrm{d}\mu(\mathbf{x}) = 0$, $\forall \mathbf{a} \in \mathbb{R}^N, \forall b \in \mathbb{R}$, implies $\mu = 0$. The sigmoids CYBENKO considered had limits

$$\sigma(t) = \begin{cases} 0 & t \to -\infty, \\ 1 & t \to +\infty. \end{cases} \tag{236}$$

This justifies use of logistic function $\sigma(x) = \frac{1}{(1+e^{-\beta x})}$ in neural network applications.

Sigmoid neural units usually use a 'bias' or 'threshold' term in computing activation potential (combination function, net input $\text{net}(k) = \mathbf{x}^\top(k)\mathbf{w}(k)$) of neural unit. Bias term is a connection weight from a unit with a constant value as shown in Fig. 3.3. Bias unit is connected to every neuron in a neural network, weight of which can be trained just like any other weight in a neural network.

From geometric point of view, for an MLP with N output units, operation of network can be seen as defining an N-dimensional hypersurface in space spanned by inputs to network. Weights define position of this surface. Without a bias term, all hypersurfaces would pass through origin (Mandic & Chambers 2000c), which in turn means: universal approximation property of neural networks would not hold if bias was omitted.

A result by Hornik (1993) shows: a sufficient condition for universal approximation property without biases is that no derivative of activation function vanishes at origin, which implies that a fixed nonzero bias can be used instead of a trainable bias.

Why use activation functions? To introduce nonlinearity into a neural network, employ nonlinear activation (output) functions. Without nonlinearity, since a composition of linear functions is again a linear function, an MLP would not be functionally different from a linear filter & would not be able to perform nonlinear separation & trajectory learning for nonlinear & nonstationary signals.

Due to Kolmogorov theorem, almost any nonlinear function is a suitable candidate for an activation function of a neuron. However, for gradient-descent learning algorithms, this function ought to be differentiable. It also helps if function is bounded.[28] For output neuron, one should either use an activation function suited to distribution of desired (target) values, or preprocess inputs to achieve this goal. If, e.g., desired values are positive but have no known upper bound, an exponential nonlinear activation function can be used.

Important to identify classes of functions & processes that can be approximated by artificial neural networks. Similar problems occur in nonlinear circuit theory, where analogue nonlinear devices are used to synthesis desired transfer functions (gyrators, impedance converters), & in digital signal processing where digital filters are designed to approximate arbitrarily well any transfer function. Fuzzy sets are also universal approximators of functions & their derivatives (Kreinovich et al. 2000; Mitaim & Kosko 1996, 1997).

- Overview. 1st explain requirements of an activation function mathematically. Introduce different types of nonlinear activation functions & discuss their properties & realizability. Finally, a complex form of activation functions within framework of Möbius transformations will be introduced.
- Neural Networks & Universal Approximation. Learning an input–output relationship from examples using a neural network can be considered as problem of approximating an unknown function $f(x)$ from a set of data points (Girosi & Poggio 1989a). This is why analysis of neural networks for approximation is important for neural networks for prediction, & also system identification & trajectory tracking. Property of uniform approximation is also found in algebraic & trigonometric polynomials, e.g. in case of Weierstrass & Fourier representation, resp.

  A neural activation function $\sigma(\cdot)$ is typically chosen to be a continuous & differentiable nonlinear function that belongs to class $S = \{\sigma_i | i = 1, \ldots, n\}$ of sigmoid[29] functions having following desirable properties[30]
  - $\sigma_i \in S$ for $i = 1, \ldots, n$
  - $\sigma_i(x_i)$ is a continuously differentiable function
  - $\sigma_i'(x_i) = \frac{d\sigma_i(x_i)}{dx_i} > 0$, $\forall x_i \in \mathbb{R}$
  - $\sigma_i(\mathbb{R} = (a_i, b_i)$, $a_i, b_i \in \mathbb{R}, a_i \neq b_i$
  - $\sigma_i'(x) \to 0$ as $x \to \pm\infty$
  - $\sigma_i'(x)$ takes a global maximal value $\max_{x\in\mathbb{R}} \sigma_i'(x)$ at a unique point $x = 0$
  - a sigmoidal function has only 1 inflection point, preferably at $x = 0$
  - from (iii), function σ_i is monotonically nondecreasing, i.e., if $x_1 < x_2$ for each $x_1, x_2 \in \mathbb{R} \Rightarrow \sigma_i(x_1) \leq \sigma_i(x_2)$
  - σ_i is uniformly Lipschitz, i.e. there exists a constant $L > 0$ s.t. $\|\sigma_i(x_1) - \sigma_i(x_2)\| \leq L\|x_1 - x_2\|$, $\forall x_1, x_2 \in \mathbb{R}$, i.e., $\frac{\sigma_i(x_1) - \sigma_i(x_2)}{x_1 - x_2} \leq L$, $\forall x_1, x_2 \in \mathbb{R}, x_1 \neq x_2$.

  Briefly discuss some of above requirements. Property (ii) represents continuous differentiability of a sigmoid function, which is important for higher order learning algorithms, which require not only existence of Jacobian matrix, but also existence of a Hessian & matrices containing higher-order derivatives. This is also necessary if behavior of a neural network is to be described via Taylor series expansion about current point in state space of network. Property (iii) states: a sigmoid should have a positive 1st derivative, which in turns means: a gradient descent algorithm which is employed for training of a neural network should have gradient vectors pointing towards bottom of bowl shaped error performance surface, which is global minimum of surface. Property (vi) means: point around which 1st derivative is centered is origin. This is connected with property (vii) which means that 2nd derivative of activation function should change its sign at origin. Going back to error performance surface, this means: irrespective of whether current prediction error is positive or negative, gradient vector of network at that point should point downwards. Monotonicity, required by (vii) is useful for uniform convergence of algorithms & in search for fixed points of neural networks. Finally, Lipschitz condition is connected with boundedness of an activation function & degenerates into requirements of uniform convergence given by contraction mapping theorem for $L < 1$.

[28] Function $f(x) = e^x$ is a suitable candidate for an activation function & is suitable for unbounded signals, is also continuously differentiable. However, to control dynamics, fixed points & invertibility of a neural network, desirable to have bounded, 'squashing' activation functions for neurons.

[29] Sigmoid means S-shaped.

[30] Constraints we impose on sigmoidal functions are stricter than ones commonly employed.

Surveys of neural transfer functions can be found in Duch & Jankowski (1999) & Cichocki & Unbehauen (1993). Examples of sigmoidal functions are

$$\sigma_1(x) = \frac{1}{1+e^{-\beta x}},\ \beta \in \mathbb{R}, \tag{238}$$

$$\sigma_2(x) = \tanh(\beta x) = \frac{e^{\beta x} - e^{-\beta x}}{e^{\beta x} + e^{-\beta x}},\ \beta \in \mathbb{R}, \tag{239}$$

$$\sigma_3(x) = \frac{2}{\pi}\arctan(\frac{\pi}{2}\beta x),\ \beta \in \mathbb{R}, \tag{240}$$

$$\sigma_4(x) = \frac{x^2}{1+x^2}\mathrm{sgn}x, \tag{241}$$

where $\sigma(x) = \Phi(x)$ as in Chap. 3. For $\beta = 1$, these functions & their derivatives are given in Fig. 4.1. Function σ_1, also known as *logistic function*,[31] is unipolar, whereas other 3 activation functions are bipolar. 2 frequently used sigmoid functions in neural networks are σ_1, σ_2. Their derivatives are also simple to calculate, & are

$$\sigma_1'(x) = \beta\sigma_1(x)(1-\sigma_1(x)), \tag{242}$$

$$\sigma_2'(x) = \beta\mathrm{sech}^2 x = \beta(1-\sigma_2^2(x)). \tag{243}$$

Can easily modify activation functions to have different saturation values. For logistic function $\sigma_1(x)$, whose saturation values are $(0,1)$, to obtain saturation values $(-1,1)$, perform $\sigma_s(x) = \frac{2}{1+e^{-\beta x}} - 1$. To modify input data to fall within range of an activation function, can normalize, standardize or rescale input data, using mean μ, standard deviation std & minimum & maximum range $R_{\min}, R_{\max}$.[32] Cybenko (1989) has shown: neural networks with a single hidden layer of neurons with sigmoidal functions are universal approximators & provided they have enough neurons, can approximate an arbitrary continuous function on a compact set with arbitrary precision. These results do not mean that sigmoidal functions always provide an optimal choice.[33] 2 functions determine way signals are processed by neurons.

* **Combination functions.** Each processing unit in a neural network performs some mathematical operation on values that are fed into it via synaptic connections (weights) from other units. Resulting value is called *activation potential* or 'net input'. Any combination function is a net: $\mathbb{R}^N \to \mathbb{R}$ function, & its output is a scalar. Most frequently used combination functions are inner product (linear) combination functions (as in MLPs & RNNs) & Euclidean or Mahalanobis distance combination functions (a sin RBF networks).
* **Activation functions.** Neural networks for nonlinear processing of signals map their net input provided by a combination function onto the output of a neuron using a scalar function called a 'nonlinear activation function', 'output function' or sometimes even 'activation function'. Entire functional mapping performed by a neuron (composition of a combination function & a nonlinear activation function) is sometimes called a 'transfer' function of a neuron $\sigma : \mathbb{R}^N \to \mathbb{R}$. Nonlinear activation functions with a bounded range are often called 'squashing' functions, e.g. commonly used tanh & logistic functions. If a unit does not transform its net input, it is said to have an 'identity' or 'linear' activation function.

Distance based combination functions (proximity functions) $D(\mathbf{x},\mathbf{t}) \propto \|\mathbf{x}-\mathbf{t}\|$, are used to calculate how close $\mathbf{x}$ is to a prototype vector $\mathbf{t}$. Also possible to use some combination of inner product & distance activation functions, e.g. in form $\alpha\mathbf{w}^\top\mathbf{x} + \beta\|\mathbf{x}-\mathbf{t}\|$ (Duch & Jankowski 1999). Many other functions can used to calculate net input, as e.g. (Ridella et al. 1997).

$$A(\mathbf{x},\mathbf{w}) = w_0 + \sum_{i=1}^N w_i x_i + w_{N+1}\sum_{i=1}^N x_i^2. \tag{245}$$

∘ Other Activation Functions. By universal approximation theorems, there are many choices of nonlinear activation function ⇒ describe some commonly used application-motivated activation functions of a neuron.
Hard-limiter Heaviside (step) function was frequently used in 1st implementations of neural networks, due to its simplicity, given by

$$H(x) = \begin{cases} 0 & x \le \theta, \\ 1 & x > \theta, \end{cases} \tag{246}$$

where θ is some threshold. A natural extension of step function is multistep function $H_{\mathrm{MS}}(x;\boldsymbol{\theta}) = y_i,\ \theta_i \le x \le \theta_{i+1}$. A variant of this function resembles a staircase $\theta_1 < \theta_2 < \cdots < \theta_N \Leftrightarrow y_1 < y_2 < \cdots < y_N$, & is often called *staircase*

[31] The logistic map $\dot{f} = rf\left(1-\frac{f}{K}\right)$ (Strogatz 1994) is used to describe population dynamics, where f: growth of a population of organisms, r: growth rate & K: carrying capacity (population cannot grow unbounded). Fixed points of this map in phase space are 0 & K, hence population always approaches carrying capacity. Under these conditions, graph of $f(t)$ belongs to class of sigmoid functions.

[32] To normalize input data to $\mu = 0, \mathrm{std} = 1$, calculate $\mu = \frac{\sum_{i=1}^N x_i}{N}, \mathrm{std} = \sqrt{\frac{\sum_{i=1}^N (x_i-\mu)^2}{N}}$, & perform standardization of input data as $\tilde{x}_i = \frac{x_i-\mu}{\mathrm{std}}$. To translate data to midrange 0 & standardize to range R, perform

$$Z = \frac{\max_i\{x_i\} + \min_i\{x_i\}}{R},\ S_x = \max_i\{x_i\} - \min_i\{x_i\},\ x_i^n = \frac{x_i - Z}{S_x/R}. \tag{244}$$

[33] Rational transfer functions (Leung & Haykin 1993) & Gaussian transfer functions also allow NNs to implement universal approximators.

function. Semilinear function is defined as

$$H_{\mathrm{SL}}(x) = \begin{cases} 0 & x \leq \theta_1, \\ \dfrac{x-\theta_1}{\theta_2-\theta_1} & \theta_1 < x \leq \theta_2, \\ 1 & x > \theta_2, \end{cases} \tag{247}$$

Functions (246) & (247) are depicted in Fig. 4.2: Step & semilinear activation function. Both mentioned functions have discontinuous derivatives, preventing use of gradient-based training procedures. Although they are, strictly speaking, *S*-shaped, do not use them for network networks for real-time processing, & this is why restricted ourselves to differentiable functions in our 9 requirements that a suitable activation function should satisfy. With development of neural network theory, these discontinuous functions were later generalized to logistic functions, leading to *graded response neurons*, which are suitable for gradient-based training. Indeed, logistic function

$$\sigma(x) = \frac{1}{1+e^{-\beta x}} \tag{248}$$

degenerates into step function (246) as $\beta \to \infty$.

Many other activation functions have been designed for special purposes. E.g., a modified activation function which enables single layer perceptrons to solve some linearly inseparable problems has been proposed in Zhang & Sarhadi (1993) & takes form (4.9)

$$f(x) = \frac{1}{1+e^{-(x^2+\mathrm{bias})}}. \tag{249}$$

Function (4.9) is differentiable $\Rightarrow$ a network based upon this function can be trained using gradient descent methods. Square operation in exponential term of function enables individual neurons to perform limited nonlinear classification. This activation function has been employed for image segmentation (Zhang & Sarhadi 1993). There have been efforts to combine 2 or more forms of commonly used functions to obtain an improved activation function. E.g., a function defined by

$$f(x) = \lambda\sigma(x) + (1-\lambda)H(x), \tag{250}$$

where $\sigma(x)$ is a sigmoid function, $H(x)$ is a hard-limiting function & $0 \leq \lambda \leq 1$, has been used in Jones (1990). Function (4.10) is a weighted sum of functions σ & H. Functions (4.9) & (4.10) are depicted in Fig. 4.3: Other activation functions.

Another possibility is to use a linear combination of sigmoid functions instead of a single sigmoid function as an activation function of a neuron. A sigmoid packet f is therefore defined as a linear combination of a set of sigmoid functions with different amplitudes h, slopes β, & biases b (Peng et al. 1998). This function is defined as

$$f(x) = \sum_{n=1}^{N} h_n \sigma_n = \sum_{n=1}^{N} \frac{h_n}{1+e^{-\beta_n x+b_n}}. \tag{251}$$

During learning phase, all parameters (h, β, b) can be adjusted for adaptive shape-refining. Intuitively, a Gaussian-shaped activation function can be, e.g., approximated by a difference of 2 sigmoids, as shown in Fig. 4.4. Other options include spline neural networks[34] (Guarnieri et al. 1999; Vecci et al. 1997) & wavelet based neural networks (Zhang et al. 1995), where structure of network is similar to RBF, except RBFs are replaced by orthonormal scaling functions that are not necessarily radial-symmetric.

For neural systems operating on chaotic input signals, most commonly used activation function is a sinusoidal function. Another activation function that is often used in order to detect chaos in input signal is so-called *saturated-modulus function* given by (Dogaru et al. 1996; Nakagawa 1996)

$$\varphi(x) = \begin{cases} |x| & |x| \leq 1, \\ 1 & |x| > 1. \end{cases} \tag{252}$$

This activation function ensures chaotic behavior even for a very small number of neurons within network. This function corresponds to rectifying operation used in electronic instrumentation & is therefore called a *saturated modulus* or *saturated rectifier function*.

- Implementation Driven Choice of Activation Functions. When neurons of a neural network are realized in hardware, due to limitation of processing power & available precision, activation functions can be significantly different from their ideal forms (Al-Ruwaihi 1997; Yang et al. 1998). Implementations of nonlinear activation functions of neurons proposed by various authors are based on a look-up table, McLaurin polynomial approximation, piecewise linear approximation or stepwise linear approximation (Basaglia et al. 1995; Murtagh & Tsoi 1992). These approximations require more iterations of learning algorithm to converge as compared with standard sigmoids.

  For neurons based upon look-up tables, samples of a chosen sigmoid are put into a ROM or RAM to store desired activation function. Alternatively, use simplified activation functions that approximate chosen activation function & are

[34] Splines are piecewise polynomials (often cubic) that are smooth & can retain 'squashing property'.

not demanding regarding processor time & memory. Thus, e.g., for logistic function, its derivative can be expressed as $\sigma'(x) = \sigma(x)(1 - \sigma(x))$, which is simple to calculate. Logistic function can be approximated using

$$f(x) = \begin{cases} 0 & x \leq -1, \\ 0.5 + x\left(1 - \frac{|x|}{2}\right) & -1 < x < 1, \\ 1 & x \geq 1. \end{cases} \tag{253}$$

Maximal absolute deviation between this function & logistic function is < 0.02. Function (4.13) is compared with logistic function: Fig. 4.5: Logistic function & its approximation. There are several other approximations. To save on computational complexity, can approximate sigmoid functions with a series of straight lines, i.e. by a piecewise-linear functions. Another sigmoid was proposed by David Elliott (Elliott 1993)

$$\sigma(x) = \frac{x}{1 + |x|}, \ \sigma'(x) = \frac{1}{(1 + |x|)^2} = (1 - |\sigma|)^2, \tag{254}$$

which is also easy to calculate. Function (4.14) & its derivative are shown in Fig. 4.6: Sigmoid function & its derivative.
In a digital VLSI implementation of an MLP, computation of activation function of each neuron is performed using a look-up table (LUT), i.e. a RAM or ROM memory which is addressed in some way (Piazza et al. 1993). An adaptive LUT based neuron is depicted in Fig. 4.7: LUT neuron.
Although sigmoidal functions are a typical choice for MLPs, several other functions have been considered. Recently, use of polynomial activation functions has been proposed (Chon & Cohen 1997; Piazza et al. 1992; Song & Manry 1993). Networks with polynomial neurons have been shown to be isomorphic to Volterra filters (Chon & Cohen 1997; Song & Manry 1993). However, calculating a polynomial activation $f(x) = a_0 + a_1(x) + \cdots + a_M x^M$ for every neuron & every time instant is extremely computationally demanding & is unlikely to be acceptable for real-time applications. Since their calculation is much slower than simple arithmetic operations, other sigmoidal functions might be useful for hardware implementations of neural networks for online applications. An overview of such functions is given in Duch & Jankowski (1999).

- MLP vs. RBF Networks. MLP- & RBF-based neural networks are the 2 most commonly used types of feedforward networks. A fundamental difference between the 2 is way in which hidden units combine values at their inputs. MLP networks use inner products, whereas RBFs use Euclidean or Mahalanobis distance as a combination function. An RBF is given by

$$f(\mathbf{x}) = \sum_{i=1}^{N} c_i G(\mathbf{x}; \mathbf{t}_i), \tag{255}$$

where $G(\cdot)$ is a basis function, $c_i, i = 1, \ldots, N$, are coefficients, $\mathbf{t}_i, i = 1, \ldots, N$, are centers of radial bases, & $\mathbf{x}$: input vector.
Both multilayer perceptrons & RBFs have good approximation properties & are related for normalized inputs. In fact, an MLP network can always simulate a Gaussian RBF network, whereas converse is true only for certain values of bias parameter (Poggio & Girosi 1990; Yee et al. 1999).

- Complex Activation Functions. Recent results suggest that despite existence of universal approximation property, approximation by real-valued neural networks might not be continuous (Kainen et al. 1999) for some standard types of neural networks, e.g. Heaviside perceptrons for Gaussian radial basis functions.[35] For many functions there is not a best approximation in $\mathbb{R}$. However, there is always a unique best approximation in $\mathbb{C}$.
Many apparently real-valued mathematical problems can be better understood if they are embedded in complex plane. Every variable is then represented by a complex number $z = x + \mathrm{i}y$, where $x, y \in \mathbb{R}, \mathrm{i} = \sqrt{-1}$. Example problems cast into complex plane include analysis of transfer functions & polynomial equations. This has motivated researchers to generalize neural networks to complex plane (Clarke 1990). Concerning hardware realization, complex weights of neural network represent impedance as opposed to resistance in real-valued networks.
If again consider approximation, (4.17)

$$f(x) = \sum_{i=1}^{N} c_i \sigma(x - a_i), \tag{256}$$

where σ: a sigmoid function, different choices of σ will give different realizations of f. An extensive analysis of this problem is given in Helmke & Williamson (1995) & Williamson & Helmke (1995). Going back to elementary function approximation, if $\sigma(x) = x^{-1}$, then (4.17) represents a partial fraction expansion of a rational function f. Coefficients a_i, c_i are, resp., poles & residuals of (4.17). Notice, however, both a_i, c_i are allowed to be complex.[36]

[35] Intuitively, since a measure of quality of an approximation is a distance function, e.g., $\mathcal{L}_2$ distance given by $\left(\int_a^b (f(x) - g(x))^2 \, \mathrm{d}x\right)^{\frac{1}{2}}$, there might occur a case where we have to calculate an integral which is not possible to be calculated within field $\mathbb{R}$, but which is easy to calculate in field $\mathbb{C}$ – recall function e^{-x^2}.

[36] Going back to transfer function approximation in signal processing, functions of type (4.17) are able to approximate a Butterworth function of any degree if (4.17) is allowed to have complex coefficients (e.g. in case of an RLC realization). On the other hand, functions with real coefficients (an RC network) cannot approximate Butterworth function whose order is ≥ 2.

A complex sigmoid is naturally obtained by analytic continuation of a real-valued sigmoid onto complex plane. In order to extend a gradient-based learning algorithm for complex signals, employed activation function should be analytic. Using analytic continuation to extend an activation function to complex plane, however, has a consequence: by Liouville Theorem C.14, only bounded differentiable functions defined on entire complex plane are constant functions. For commonly used activation functions, however, singularities occur in a limited set.[37] For logistic function $\sigma(z) = \frac{1}{1+e^{-z}} = u + \mathrm{i}v$, if z approaches any value in set $\{0 \pm \mathrm{i}(2n+1)\pi,\ n \in \mathbb{Z}\}$, then $|\sigma(z)| \to \infty$. Similar conditions for tanh are $\{0 \pm \mathrm{i}\frac{2n+1}{2}\pi,\ n \in \mathbb{Z}\}$, whereas for e^{-z^2}, have singularities for $z = 0 + \mathrm{i}y$ (Georgiou & Koutsougeras 1992).
Hence, a function obtained by an analytic continuation to complex plane is generally speaking not an appropriate activation function. Generalizing discussion for real activation functions, properties that a function $\sigma(z) = u(x, y) + \mathrm{i}v(x, y)$ should satisfy so that it represents a suitable activation function in complex plane are (Georgiou & Koutsougeras 1992)

* u, v are nonlinear in x, y
* $\sigma(z)$ is bounded $\Rightarrow u, v$ are bounded
* $\sigma(z)$ has bounded partial derivatives u_x, u_y, v_x, v_y, which satisfy Cauchy–Riemann conditions (Mathews & Howell 1997)
* $\sigma(z)$ is not entire (not a constant).

Regarding fixed point iteration & global asymptotic stability of neural networks, which will be discussed in more detail in Chap. 7, complex-valued neural networks can generate dynamics $z \leftarrow \Phi(z)$. Functions of form $cz(1 - z)$, e.g., give rise to Mandelbrot & Julia sets (Clarke 1990; Devaney 1999; Strogatz 1994). A single complex neuron with a feedback connection is thus capable of performing complicated discriminations & generation of nonlinear behavior.
To provide conditions on capability of complex neural networks to approximate nonlinear dynamics, a density theorem for complex MLPs with nonanalytic activation function & a hidden layer is proved in Arena et al. (1998b). The often cited denseness conditions are, as pointed out by Cotter (1990), special cases of Stone–Weierstrass theorem.
In context of learning algorithms, Leung & Haykin (1991) developed their complex backpropagation algorithm considering following activation function (4.19)

$$f(z) = \frac{1}{1 + e^{-z}} : \mathbb{C}^N \to \mathbb{C}, \tag{257}$$

whose magnitude is shown in Fig. 4.8: Complex extension of logistic function $\sigma(z) = \frac{1}{1+e^{-z}}$. This complex extension of logistic function has singularities due to complex exponential in denominator fo (4.19). Safe to use (4.19) if inputs are scaled to range of complex logistic function where it is analytic. In Benvenuto & Piazza (1992), the activation function is proposed: (4.20)

$$\sigma(z) = \sigma(x) + \mathrm{i}\sigma(y), \tag{258}$$

where $\sigma(z)$ is a 2D extension of a standard sigmoid. Magnitude of this function is shown in Fig. 4.9: Complex sigmoid $\sigma(z) = \sigma(z_r) + \mathrm{i}\sigma(z_i)$. Function (4.20) is not analytic & bounded on $\mathbb{C}$. It is, however, discriminatory, & linear combinations of functions of this type are dense in $\mathbb{C}$ (Arena et al. 1998a).
Another proposed complex sigmoid is (Benvenuto & Piazza 1992)

$$\sigma(z) = \frac{2c_1}{1 + e^{-2c_2 z}} - c_1, \tag{259}$$

where c_1, c_2 are suitable parameters. Derivative of this function is $\sigma'(z) = \frac{c_2}{2c_1}(c_1^2 - \sigma^2(z))$. Other work on complex backpropagation was proposed in Kim & Guest (1990).
A suitable complex activation function would have property: an excitation near 0 would remain close to 0, & large excitations would be mapped into bounded outputs. 1 such function is given by (Clarke 1990)

$$\sigma(z) = \frac{(\cos\theta + \mathrm{i}\sin\theta)(z - s)}{1 - \alpha^\star z}, \tag{260}$$

where θ: a rotation angle & α: a complex constant of magnitude < 1. Operator $(\cdot)^\star$: complex conjugation; sign of imaginary part of asterisked variable is changed. This function is a conformal mapping of unit disc in complex plane onto itself & is unique. Further, σ maps large complex numbers into $-\frac{1}{\alpha}$ & thus satisfies above criteria. 1 flaw in σ is a singularity at $z = \frac{1}{\alpha}$, but in view of Liouville's theorem this is unavoidable. Magnitude plot of this function is shown in Fig. 4.10: A complex activation function.
A simple function that satisfies all above properties is (Georgiou & Koutsougeras 1992)

$$f(z) = \frac{z}{c + \frac{|z|}{r}}, \tag{261}$$

where c, r: real positive constants. This function maps any point in complex plane onto open disc $\{z : |z| < r\}$ as shown in Fig. 4.11: Magnitude of function $f(z) = \frac{z}{c+\frac{|z|}{r}}$.

- Complex Valued Neural Networks as Modular Groups of Compositions of Möbius Transformations. Offer a different perspective upon some inherent properties of neural networks, e.g. fixed points, nesting & invertibility, by exposing representations of neural networks in framework of Möbius transformations. This framework includes consideration of complex weights & inputs to network together with complex sigmoidal activation functions.

[37] Exponential function $\exp : \mathbb{C} \to \mathbb{C}^\star$ maps set $\{z = a + (2k+1)\mathrm{i}\pi\}, a \in \mathbb{R}, k \in \mathbb{Z}$ onto negative real axis, which determine singularities of complex sigmoids.

    - Möbius Transformation.
    - Activation Functions & Möbius Transformations.
    - Existence & Uniqueness of Fixed Points in a Complex Neural Network via Theory of Modular Groups.
  - Summary. An overview of nonlinear activation functions used in neural networks has been provided. Started from problem of function approximation & trajectory learning & evaluated neural networks suitable for these problems. Properties of neural networks realized in hardware have also been addressed. For rigor, analysis has been extended to neural networks with complex activation functions, for which we have built a unified framework via modular groups of Möbius transformations.
    Existence & uniqueness conditions of fixed points & invertibility of such mappings have been derived. These results apply both for general input–output relationship in a neural network as well as for a single neuron. This analysis provides a strong mathematical background for further analysis of neural networks for adaptive filtering & prediction.
- Recurrent Neural Networks Architectures.
  - Perspective. Use of neural networks, in particular recurrent neural networks, in system identification, signal processing & forecasting is considered. Ability of neural networks to model nonlinear dynamical systems is demonstrated, & correspondence between neural networks & block-stochastic models is established. Provide further discussion of recurrent neural network architectures.
  - Introduction. There are numerous situations in which use of linear filters & models is limited. E.g., when trying to identify a saturation type nonlinearity, linear models will inevitably fail. This is also case when separating signals with overlapping spectral components.
    Most real-world signals are generated, to a certain extent, by a nonlinear mechanism & therefore in many applications choice of a nonlinear model may be necessary to achieve an acceptable performance from an adaptive predictor. Communications channels, e.g., often need nonlinear equalizers to achieve acceptable performance. Choice of model has crucial importance[38] & practical applications have shown: nonlinear models can offer a better prediction performance than their linear counterparts. They also reveal rich dynamical behavior, e.g. limit cycles, bifurcations & fixed points, that cannot be captured by linear models (Gershenfeld & Weigend 1993).
    By *system* we consider actual underlying physics[39] that generate data, whereas by *model* we consider a mathematical description of system. Many variations of mathematical models can be postulated on basis of datasets collected from observations of a system, & their suitability assessed by various performance metrics. Since not possible to characterize nonlinear system by their impulse response, one has to resort to less general models, e.g. homomorphic filters, morphological filters & polynomial filters. Some of most frequently used polynomial filters are based upon Volterra series (Mathews 1991), a nonlinear analogue of linear impulse response, threshold autoregressive models (TAR) (Priestley 1991) & Hammerstein & Wiener models. The latter 2 represent structures that consist of a linear dynamical model & a static zero-memory nonlinearity. An overview of these models can be found in Haber & Unbehauen (1990). Notice: for nonlinear systems, ordering of modules within a modular structure[40] plays an important role.
    To illustrate some important features associated with nonlinear neurons, consider a squashing nonlinear activation function of a neuron, shown in Fig. 5.1: Effects of $y = \tanh v$ nonlinearity in a neuron model upon 2 example inputs. For 2 identical mixed sinusoidal inputs with different offsets, passed through this nonlinearity, output behavior varies from amplifying & slightly distorting input signal to attenuating & considerably nonlinearly distorting input signal. From viewpoint of system theory, neural networks represent nonlinear maps, mapping 1 metric space to another.
    Nonlinear system modeling has traditionally focused on Volterra–Wiener analysis. These models are nonparametric & computationally extremely demanding. Volterra series expansion is given by

$$y(k) = h_0 + \sum_{i=0}^{N} h_1(i)x(k-i) + \sum_{i=0}^{N}\sum_{j=0}^{N} h_2(i,j)x(k-i)x(k-j) + \cdots \tag{262}$$

    for representation of a causal system. A nonlinear system represented by a Volterra series is completely characterized by its Volterra kernels $h_i, i = 0, 1, 2, \ldots$ The Volterra modeling of a nonlinear system requires a great deal of computation, & mostly 2nd- or 3rd-order Volterra systems are used in practice.
    Since Volterra series expansion is a Taylor series expansion with memory, they both fail when describing a system with discontinuities, e.g. $y(k) = A\text{sgn}(x(k))$, where $\text{sgn}(\cdot)$ is signum function.
    To overcome this difficulty, nonlinear parametric models of nonlinear systems, termed NARMAX, that are described by nonlinear difference equations, have been introduced (Billings 1980; Chon & Cohen 1997; Chon et al. 1999; Connor 1994). Unlike Volterra–Wiener representation, NARMAX representation of nonlinear systems offers compact representation.
    NARMAX model describes a system by using a nonlinear functional dependence between lagged inputs, outputs &/or prediction errors. A polynomial expansion of transfer function of a NARMAX neural network does not comprise of delayed versions of input & output of order higher than those presented to network. Therefore, input of an insufficient order will result in undermodeling, which complies with Takens' embedding theorem (Takens 1981).
    Applications of neural networks in forecasting, signal processing & control require treatment of dynamics associated with input signal. Feedforward networks for processing of dynamical systems tend to capture dynamics by including past

[38] System identification, e.g., consists of choice of model, model parameter estimation & model validation.
[39] Technically, notions of *system* & *process* are equivalent (Pearson 1995; Sjöberg et al. 1995).
[40] To depict this, for 2 modules performing nonlinear functions $H_1 = \sin x, H_2 = e^x$, have $H_1(H_2(x)) \neq H_2(H_1(x))$ since $\sin e^x \neq e^{\sin x}$. This is reason to use term *nesting* rather than cascading in modular neural networks.

inputs in input vector. However, for dynamical modeling of complex systems, there is a need to involve feedback, i.e., to use recurrent neural networks. There are various configurations of recurrent neural networks, which are used by Jordan (1986) for control of robots, by Elman (1990) for problems in linguistics & by Williams & Zipser (1989a) for nonlinear adaptive filtering & pattern recognition. In Jordan's network, past values of network outputs are fed back into hidden units, in Elman's network, past values of outputs of hidden units are fed back into themselves, whereas in Williams–Zipser architecture, network is fully connected, having 1 hidden layer.

There are numerous modular & hybrid architectures, combining linear adaptive filters & neural networks. These include pipelined recurrent neural network & networks combining recurrent networks & FIR adaptive filters. Main idea: linear filter captures linear 'portion' of input process, whereas a neural network captures nonlinear dynamics associated with process.

- Overview. Introduce basic modes of modeling, e.g. *parametric, nonparametric, white box, black box & grey box* modeling. Afterwards, dynamical richness of neural models is addressed & feedforward & recurrent modeling for noisy time series are compared. Block-stochastic models are introduced & neural networks are shown to be able to represent these models. Chapter concludes with an overview of recurrent neural network architectures & recurrent neural networks for NARMAX modeling.
- Basic Modes of Modeling. Explain notions of parametric, nonparametric, black box, grey box, & white box modeling. These can be used to categorize neural network algorithms, e.g. direct gradient computation, a posteriori & normalized algorithms. Basic idea behind these approaches to modeling is *not to estimate what is already known* $\Rightarrow$ One should utilize prior knowledge & knowledge about physics of system, when selecting neural network model prior to parameter estimation.
  - Parametric vs. Nonparametric Modeling. A review of nonlinear input–output modeling techniques is given in Pearson (1995). 3 classes of input–output models are *parametric, nonparametric, & semiparametric* models. Briefly address them:
    - *Parametric* modeling assumes a fixed structure for model. Model identification problem then simplifies to estimating a finite set of parameters of this fixed model. This estimation is based upon prediction of real input data, so as to best match input data dynamics. An example of this technique is broad class of ARIMA/NARMA models. For a given structure of model (e.g. NARMA) we recursively estimate parameters of chosen model.
    - *Nonparametric* modeling seeks a particular model structure from input data. Actual model is not known beforehand. An example taken from nonparametric regression is: look for a model in form of $y(k) = f(x(k))$ without knowing function $f(\cdot)$ (Pearson 1995).
    - *Semiparametric* modeling is combination of above. Part of model structure is completely specified & known beforehand, whereas other part of model is either not known or loosely specified.

    Neural networks, especially recurrent neural networks, can be employed within estimators of all of above classes of models. Closely related to above concepts are white, grey, & black box modeling techniques.
  - White, Grey, & Black Box Modeling. To understand & analyze real-world physical phenomena, various mathematical models have been developed. Depending on some a priori knowledge about process, data & model, differentiate between 3 fairly general modes of modeling. Idea: to distinguish between 3 levels of prior knowledge, which have been 'color-coded'. An overview of white, grey, & black box modeling techniques can be found in Aguirre (2000) & Sjöberg et al. (1995).

    Given data gathered from planet movements, then Kepler's gravitational laws might well provide initial framework in building a mathematical model of process. This mode of modeling is referred to as *white box* modeling (Aguirre 2000), underlying its fairly deterministic nature. Static data are used to calculate parameters, & to do that underlying physical process has to be understood $\Rightarrow$ possible to build a *white box* model entirely from physical insight & prior knowledge. However, underlying physics are generally not completely known, or are too complicated & often one has to resort to other types of modeling.

    Exact form of input–output relationship that describes a real-world system is most commonly unknown, & therefore modeling is based upon a chosen set of known functions. In addition, if model is to approximate system with an arbitrary accuracy, set of chosen nonlinear continuous functions must be dense: case with polynomials. In this light, neural networks can be viewed as another mode of functional representations. *Black box* modeling therefore assumes no previous knowledge about system that produces data. However, chosen network structure belongs to architectures that are known to be flexible & have performed satisfactorily on similar problems. Aim: to find a function F that approximates process y based on previous observations of process y_{PAST} & input u as $y = F(u_{\mathrm{PAST}}, u)$. This 'black box' establishes a functional dependence between input & output, which can be either linear or nonlinear. Downside: generally not possible to learn about true physical process that generates data, especially if a linear model is used. Once training process is complete, a neural network represents a *black box*, nonparametric process model. Knowledge about process is embedded in values of network parameters (i.e. synaptic weights).

    A natural compromise between 2 previous models is so-called *grey box* modeling, obtained from *black box* modeling if some information about system is known a priori. This can be a probability density function, general statistics of process data, impulse response or attractor geometry. In Sjöberg et al. (1995), 2 subclasses of *grey box* models are considered: *physical* modeling, where a model structure is built upon understanding of underlying physics, as e.g. state-space model structure; & *semiphysical* modeling, where, based upon physical insight, certain nonlinear combinations of data structures are suggested, & then estimated by *black box* methodology.

- NARMAX Models & Embedding Dimension. For neural networks, number of input nodes specifies dimension of network input. In practice, true state of system is not observable & mathematical model of system that generates dynamics is not known. Question arises: is sequence of measurements $\{y(k)\}$ sufficient to reconstruct nonlinear system dynamics? Under some regularity conditions, Takens' (1981) & Mane's (1981) embedding theorems establish this connection. To ensure that dynamics of a nonlinear process estimated by a neural network are fully recovered, convenient to use Takens' embedding theorem (Takens 1981), which states: to obtain a faithful reconstruction of system dynamics, *embedding dimension* d must satisfy $d \geq 2D+1$, where D: dimension of system attractor. Takens' embedding theorem (Takens 1981; Wan 1993) estalishes a diffeomorphism between a finite window of time series $[y(k-1), y(k-2), \ldots, y(k-N)]$ & underlying state of dynamic system which generates time series. This implies: a a nonlinear regression $y(k) = g[y(k-1), y(k-2), \ldots, y(k-N)]$ can model nonlinear time series. An important feature of delay-embedding theorem due to Takens (1981): it is physically implemented by delay lines.
  There is a deep connection between time-lagged vectors & underlying dynamics. Delay vectors are not just a representation of a state of system, their length is key to recovering full dynamical structure of a nonlinear system. A general starting point would be to use a network for which input vector comprises delayed inputs & outputs, as shown in Fig. 5.2: Nonlinear prediction configuration using a neural network model. For network in Fig. 5.2, both input & output are passed through delay lines, hence indicating NARMAX character of this network. The switch in this figure indicates 2 possible modes of learning explained in Chap. 6.
- How Dynamically Rich are Nonlinear Neural Models? To make an initial step toward comparing neural & other nonlinear models, perform a Taylor series expansion of sigmoidal nonlinear activation function of a single neuron model as (Billings et al. 1992) (5.7)

$$\Phi(v(k)) = \frac{1}{1+e^{-\beta v(k)}} = \frac{1}{2} + \frac{\beta}{4}v(k) - \frac{\beta^3}{48}v^3(k) + \frac{\beta^5}{480}v^5(k) - \frac{17\beta^7}{80640}v^7(k) + \cdots \tag{263}$$

  Depending on steepness β & activation potential $v(k)$, polynomial representation (5.7) of transfer function of a neuron exhibits a complex nonlinear behavior.
  Consider a NARMAX recurrent perceptron with $p = 1, q = 1$ as shown in Fig. 5.3, which is a simple example of recurrent neural networks. Its mathematical description is given by

$$y(k) = \Phi(w_1 x(k-1) + w_2 y(k-1) + w_0). \tag{264}$$

  Expanding this using (5.7):

$$y(k) = \frac{1}{2} + \frac{1}{4}[w_1 x(k-1) + w_2 y(k-1) + w_0] - \frac{1}{48}[w_1 x(k-1) + w_2 y(k-1) + w_0]^3 + \cdots, \tag{265}$$

  where $\beta = 1$. This expression illustrates dynamical richness of squashing activation functions. Associated dynamics, when represented in terms of polynomials are quite complex. Networks with more neurons & hidden layers will produce more complicated dynamics than those in (5.9). Following same approach, for a general recurrent neural network, obtain (Billings et al. 1992) (5.10)

$$y(k) = c_0 + c_1 x(k-1) + c_2 y(k-1) + c_3 x^2(k-1) + c_4 y^2(k-1) + c_5 x(k-1)y(k-1) + c_6 x^3(k-1) + c_7 y^3(k-1) + c_8 x^2(k-1)y(k-1) + \cdots. \tag{266}$$

  (5.10) does not comprise delayed versions of input & output samples of order higher than those presented to network. If input vector were of an insufficient order, undermodeling would result, which complies with Takens' embedding theorem $\Rightarrow$ when modeling an unknown dynamical system or tracking unknown dynamics, important to concentrate on embedding dimension of network. Representation (5.10) also models an offset (mean value) c_0 of input signal.
  - Feedforward vs. Recurrent Networks for Nonlinear Modeling. Choice of which neural network to employ to represent a nonlinear physical process depends on dynamics & complexity of network that is best for representing problem in hand. E.g., due to feedback, recurrent networks may suffer from instability & sensitivity to noise. Feedback networks, on the other hand, might not be powerful enough to capture dynamics of underlying nonlinear dynamical system. To illustrate this problem, resort to a simple IIR (ARMA) linear system described by following 1st-order difference equation (5.11)

$$z(k) = 0.5z(k-1) + 0.1x(k-1). \tag{267}$$

    System (5.11) is stable, since pole of its transfer function is at 0.5, i.e., within unit circle in z-plane. However, in a noisy environment, output $z(k)$ is corrupted by noise $e(k)$, so that noisy output $y(k)$ of system (5.11) becomes $y(k) = z(k) + e(k)$, which will affect quality of estimation based on this model. This happens because noise terms accumulate during recursions[41] (5.11) as

$$y(k) = 0.5y(k-1) + 0.1x(k-1) + e(k) - 0.5e(k-1). \tag{268}$$

    An equivalent FIR (MA) representation of same filter (5.11), using method of long division, gives

$$z(k) = 0.1x(k-1) + 0.05x(k-2) + 0.025x(k-3) + 0.0125x(k-4) + \cdots \tag{269}$$

[41] Notice: if noise $e(k)$ is zero mean & white it appears colored in (5.13), i.e., correlated with previous outputs, which leads to biased estimates.

& representation of a noisy system now becomes (5.15)

$$y(k) = 0.1x(k-1) + 0.05x(k-2) + 0.025x(k-3) + 0.0125x(k-4) + \cdots + e(k). \tag{270}$$

Clearly, noise in (5.15) is not correlated with previous outputs & estimates are unbiased.[42] Price to pay, however, is infinite length of exact representation of (5.11).
A similar principle applies to neural networks. In Chap. 6, address modes of learning in neural networks & discuss bias/variance dilemma for recurrent neural networks.

- Wiener & Hammerstein Models & Dynamical Neural Networks. Under relatively mild conditions,[43] output signal of a nonlinear model can be considered as a combination of outputs from some suitable submodels. Structure identification, model validation & parameter estimation based upon these submodels are more convenient than those of whole model. Block oriented stochastic models consist of static nonlinear & dynamical linear modules. Such models often occur in practice, examples of which are:
  - Hammerstein model, where a zero-memory nonlinearity is followed by a linear dynamical system characterized by its transfer function $H(z) = \frac{N(z)}{D(z)}$
  - Wiener model, where a linear dynamical system is followed by a zero-memory nonlinearity.
  - Overview of Block-Stochastic Models. Defs of certain stochastic models are given by
    (a) Wiener system

$$y(k) = g(H(z^{-1})u(k)), \tag{271}$$

where $u(k)$: input to system, $y(k)$: output, $H(z^{-1}) = \frac{C(z^{-1})}{D(z^{-1})}$: z-domain transfer function of linear component of system & $g(\cdot)$: a nonlinear function

    (b) Hammerstein system

$$y(k) = H(z^{-1})g(u(k)) \tag{272}$$

    (c) Uryson system, defined by

$$y(k) = \sum_{i=1}^{M} H_i(z^{-1})g_i(u(k)). \tag{273}$$

Theoretically, there are finite size neural systems with dynamic synapses which can represent all of above. Moreover, some modular neural architectures, e.g. PRNN (Haykin & Li 1995), are able to represent block-cascaded Wiener–Hammerstein systems described by (Mandic & Chambers 1999c)

$$y(k) = \Phi_N(H_N(z^{-1})\Phi_{N-1}(H_{N-1}(z^{-1}) \cdots \Phi_1(H_1(z^{-1})u(k)))), \tag{274}$$

$$y(k) = H_N(z^{-1})\Phi_N(H_{N-1}(z^{-1})\Phi_{N-1} \cdots \Phi_1(H_1(z^{-1}u(k)))) \tag{275}$$

under certain constraints relating size of networks & order of block-stochastic models. Due to its parallel nature, however, a general Uryson model is not guaranteed to be representable this way.

  - Connection Between Block-Stochastic Models & Neural Networks. Block diagrams of Wiener & Hammerstein systems are shown in Fig. 5.4: Nonlinear stochastic model used in control & signal processing. Nonlinear function from Fig. 5.4a can be generally assumed to be a polynomial[44] i.e., $v(k) = \sum_{i=0}^{M} \lambda_i u^i(k)$. Hammerstein model is a conventional parametric model, usually used to represent processes with nonlinearities involved with process inputs, as shown in Fig. 5.4a. Equation describing output of a SISO Hammerstein system corrupted with additive output noise $\eta(k)$ is

$$y(k) = \Phi[u(k-1)] + \sum_{i=2}^{\infty} h_i \Phi[u(k-i)] + \nu(k), \tag{276}$$

where Φ is a nonlinear function which is continuous. Other requirements: linear dynamical subsystem is stable. This network is shown in Fig. 5.5: Discrete-time SISO Hammerstein model with observation noise.
Neural networks with locally distributed dynamics (LDNN) can be considered as locally recurrent networks with global feedforward features. An example of these networks is the *dynamical multilayer perceptron* (DMLP) which consists of dynamical neurons & is shown in Fig. 5.6: Dynamic perceptron. Model of this dynamic perceptron is described by

$$y(k) = \Phi(v(k)), \tag{277}$$

$$v(k) = \sum_{i=0}^{\deg N(z)} n_i(k)x(k-i) + 1 + \sum_{j=1}^{\deg D(z)} d_j(k)v(k-j), \tag{278}$$

$$x(k) = \sum_{l=1}^{p} w_l(k)u_l(k), \tag{279}$$

[42] Under usual assumption that external additive noise $e(k)$ is not correlated with input signal $x(k)$.
[43] A finite degree polynomial steady-state characteristic.
[44] By Weierstrass theorem, polynomials can approximate arbitrarily well any nonlinear function, including sigmoid functions.

where n_i, d_i denote, resp., coefficients of polynomials in $N(z), D(z)$ & '1' is included for a possible bias input. From Fig. 5.6: transfer function between $y(k), x(k)$ represents a Wiener system. Hence, combinations of dynamical perceptrons (e.g. a recurrent neural network) are able to represent block-stochastic Wiener–Hammerstein models. Gradient-based learning rules can be developed for a recurrent neural network representing block-stochastic models. Both Wiener & Hammerstein models can exhibit a more general structure, as shown in Fig. 5.7: Generalized Hammerstein model, for Hammerstein model. Wiener & Hammerstein models can be combined to produce more complicated block-stochastic model. A representative of these models is Wiener–Hammerstein model, shown in Fig. 5.8: Wiener–Hammerstein model. This figure shows a Wiener stochastic model, followed by a linear dynamical system represented by its transfer function $H_2(z) = \frac{N_2(z)}{D_2(z)}$, hence building a Wiener–Hammerstein block-stochastic system. In practice, can build complicated block cascaded systems this way.

Wiener & Hammerstein systems are frequently used to compensate each other (Kang et al. 1998). This includes finding an inverse of 1st module in combination. If these models are represented by neural networks, Chap. 4 provides a general framework for uniqueness, existence, & convergence of inverse neural models. Following example from Billings & Voon (1986) shows: Wiener model can be represented by a NARMA model, which, in turn can be modeled by a recurrent neural network.

Example 13. *Wiener model*

$$w(k) = 0.8w(k-1) + 0.4u(k-1), \tag{280}$$

$$y(k) = w(k) + w^3(k) + e(k), \tag{281}$$

was identified as [complicated (5.25)] which is a NARMA model, & hence can be realized by a recurrent neural network.

- Recurrent Neural Network Architectures. 2 straightforward ways to include recurrent connections in neural networks are *activation feedback & output feedback*, as shown, resp., in Fig. 5.9a: Activation feedback scheme & Fig. 5.9b: Output feedback scheme. These schemes are closely related to state space representation of neural networks. A comprehensive & insightful account of canonical forms & state space representation of general neural networks is given in Nerrand et al. (1993) & Dreyfus & Idan (1998). In Fig. 5.9: Recurrent neural network architectures, blocks labeled 'linear dynamical systems' comprise of delays & multipliers, hence providing linear combination of their input signals. Output of a neuron shown in Fig. 5.9a can be expressed as

$$v(k) = \sum_{i=0}^{M} w_{u,i}(k)u(k-i) + \sum_{j=1}^{N} w_{v,j}(k)v(k-j), \tag{282}$$

$$y(k) = \Phi(v(k)), \tag{283}$$

where $w_{u,i}, w_{v,j}$ correspond to weights associated with u, v, resp.

Transfer function of a neuron shown in Fig. 5.9b can be expressed as

$$v(k) = \sum_{i=0}^{M} w_{u,i}(k)u(k-i) + \sum_{j=1}^{N} w_{y,j}(k)y(k-j), \tag{284}$$

$$y(k) = \Phi(v(k)), \tag{285}$$

where $w_{y,j}$ correspond to weights associated with delayed outputs. A comprehensive account of types of synapses & short-term memories in dynamical neural networks is provided by Mozer (1993).

Networks mentioned so far exhibit a locally recurrent architecture, but when connected into a larger network, they have a feedforward structure. Hence they are referred to as locally recurrent-globally feedforward (LRGF) architectures. A general LRGF architecture is shown in Fig. 5.10: General LRGF architecture. This architecture allows for dynamic synapses both within input (represented by $H_1, \ldots, H_M$) & output feedback (represented by H_{FB}), hence comprising some of aforementioned schemes.

Elman network is a recurrent network with a hidden layer, a simple example of which is shown in Fig. 5.11: An example of Elman recurrent neural network. This network consists of an MLP with an additional input which consists of delayed state space variables of network. Even though it contains feedback connections, it is treated as a kind of MLP. Network shown in Fig. 5.12: An example of Jordan recurrent neural network is an example of Jordan network. It consists of a multilayer perceptron with 1 hidden layer & a feedback loop from output layer to an additional input called *context layer*. In context layer, there are self-recurrent loops. Both Jordan & Elman networks are structurally locally recurrent globally feedforward (LRGF), & are rather limited in including past information.

A network with a rich representation of past outputs, which will be extensively considered in this book, is a fully connected recurrent neural network, known as Williams–Zipser network (Williams & Zipser 1989a), shown in Fig. 5.13: A fully connected recurrent neural network. Give a detailed introduction to this architecture. This network consists of 3 layers: input layer, processing layer, & output layer. For each neuron $i = 1, \ldots, N$, elements $u_j, j = 1, 2, \ldots, p+N+1$, of input vector to a neuron $\mathbf{u}$ (5.31), are weighted, then summed to produce an internal activation function of a neuron v (5.30), which is finally fed through a nonlinear activation function Φ (5.28), to form output of ith neuron y_i (5.29). Function Φ is a monotonically increasing sigmoid function with slope β, as e.g. logistic function, $\Phi(v) = \frac{1}{1+e^{-\beta v}}$. At time instant k, for ith neuron, its weights form a $(p+N+1)\times 1$ dimensional weight vector $\mathbf{w}_i^\top(k) = [w_{i,1}(k), \ldots, w_{i,p+N+1}(k)]$,

where p: number of external inputs, N: number of feedback connections & $(\cdot)^\top$ denotes vector transpose operation. 1 additional element of weight vector $\mathbf{w}$: bias input weight. Feedback consists of delayed output signals of RNN. Following equations fully describe RNN from Fig. 5.13: A fully connected recurrent neural network,

$$y_i(k) = \Phi(v_i(k)),\ i = 1, \ldots, N, \tag{286}$$

$$v_i(k) = \sum_{l=1}^{p+N+1} w_{i,l}(k) u_l(k), \tag{287}$$

$$\mathbf{u}_i^\top = [s(k-1), \ldots, s(k-p), 1, y_1(k-1), y_2(k-1), \ldots, y_N(k-1)], \tag{288}$$

where $(p+N+1) \times 1$ dimensional vector $\mathbf{u}$ comprises both external & feedback inputs to a neuron, as well as unity valued constant bias input.

- Hybrid Neural Network Architectures. These networks consist of a cascade of a neural network & a linear adaptive filter. If a neural network is considered as a complex adaptable nonlinearity, then hybrid neural networks resemble Wiener & Hammerstein stochastic models. An example of these networks is given in Khalaf & Nakayama (1999), for prediction of noisy time series. A neural subpredictor is cascaded with a linear FIR predictor, hence making a hybrid predictor. Block diagram of this type of neural network architecture is given in Fig. 5.14: A hybrid neural predictor. Neural network from Fig. 5.14 can be either a feedforward neural network or a recurrent neural network.
  Another example of hybrid structures is so-called *pipelined recurrent neural network* (PRNN), introduced by Haykin & Li (1995) & shown in Fig. 5.15. It consists of a modular nested structure of small-scale fully connected recurrent neural networks & a cascaded FIR adaptive filter. In PRNN configuration, M modules, which are FCRNNs, are connected as shown in Fig. 5.15: Pipelined recurrent neural network. Cascaded linear filter is omitted. Description of this network follows approach from Mandic et al. (1998) & Baltersee & Chambers (1998). Uppermost module of PRNN, denoted by M, is simply an FCRNN, whereas in modules $(M-1, \ldots, 1)$, only difference: feedback signal of output neuron within module m, denoted by $y_{m,1}, m = 1, \ldots, M-1$, is replaced with appropriate output signal $y_{m+1,1}, m = 1, \ldots, M-1$, from its left neighbor module $m+1$. $(p \times 1)$-dimensional external signal vector $\mathbf{s}^\top(k) = [s(k), \ldots, s(k-p+1)]$ is delayed by m time steps ($z^{-m}\mathbf{I}$) before feeding module m, where $z^{-m}, m = 1, \ldots, M$, denotes m-step time delay operator & $\mathbf{I}$: $(p \times p)$-dimensional identity matrix. Weight vectors $\mathbf{w}_n$ of each neuron n, are embodied in an $(p+N+1) \times N$ dimensional weight matrix $\mathbf{W}(k) = [\mathbf{w}_1(k), \ldots, \mathbf{w}_N(k)]$, with N: number of neurons in each module. All modules operate using same weight matrix $\mathbf{W}$. Overall output signal of PRNN is $y_{\text{out}}(k) = y_{1,1}(k)$, i.e., output of 1st neuron of 1st module. A full mathematical description of PRNN is given in equations:

$$y_{i,n}(k) = \Phi(v_{i,n}(k)), \tag{289}$$

$$v_{i,n}(k) = \sum_{l=1}^{p+N+1} w_{n,l}(k) u_{i,l}(k), \tag{290}$$

$$\mathbf{u}_i^\top(k) = [s(k-i), \ldots, s(k-i-p+1), 1, y_{i+1,1}(k), y_{i,2}(k-1), \ldots, y_{i,N}(k-1)] \text{ for } 1 \le i \le M-1, \tag{291}$$

$$\mathbf{u}_M^\top(k) = [s(k-M), \ldots, s(k-M-p+1), 1, y_{M,1}(k-1), y_{M,2}(k-1), \ldots, y_{M,N}(k-1)] \text{ for } i = M. \tag{292}$$

At time step k for each module $i = 1, \ldots, M$, 1-step forward prediction error $e_i(k)$ associated with a module is then defined as a difference between desired response of that module $s(k-i+1)$, which is actually next incoming sample of external input signal, & actual output of ith module $y_{i,1}(k)$ of PRNN, i.e., (5.36)

$$e_i(k) = s(k-i+1) - y_{i,1}(k),\ i = 1, \ldots, M. \tag{293}$$

Thus, overall cost function of PRNN becomes a weighted sum of all squared error signals $E(k) = \sum_{i=1}^{M} \lambda^{i-1} e_i^2(k)$, where $e_i(k)$ is defined in (5.36) & $\lambda \in (0, 1]$, is a forgetting factor.
Other architectures combining linear & nonlinear blocks include so-called 'sandwich' structure which was used for estimation of Hammerstein systems Ibnkahla et al. 1998). Architecture used was a linear-nonlinear-linear combination.

- Nonlinear ARMA Models & Recurrent Networks. A general NARMA(p, q) recurrent network model can be expressesd as (Chang and Hu 1997)

$$\hat{x}(k) = \Phi\left(\sum_{i=1}^{p} w_{1,i}(k) x(k-i) + w_{1,p+1}(k) + \sum_{j=p+2}^{p+q+1} w_{1,j}(k) \hat{e}(k+j-2-p-q) + \sum_{l=p+q+2}^{p+q+N} w_{1,l}(k) y_{l-p-q}(k-1) \right). \tag{294}$$

A realization of this model is shown in Fig. 5.16: Alternative recurrent NARMA(p, q) network. NARMA(p, q) scheme shown in Fig. 5.16 is a common Williams–Zipser type recurrent neural network, which consists of only 2 layers, output layer of output & hidden neurons $y_1, \ldots, y_N$, & input layer of feedforward & feedback signals

$$x(k-1), \ldots, x(k-p), +1, \hat{e}(k-1), \ldots, \hat{e}(k-q), y_2(k-1), \ldots, y_N(k-1). \tag{295}$$

Nonlinearity in this case is determined by both nonlinearity associated with output neuron of recurrent neural network & nonlinearities in hidden neurons.

Inputs to this network, given in (5.38), however, comprise prediction error terms (residuals) $\hat{e}(k-1),\ldots,\hat{e}(k-q)$, which make learning in such networks difficult. Namely, well-known real-time recurrent learning (RTRL) algorithm (Haykin 1994; Williams & Zipser 1989a) was derived to minimize the instantaneous squared prediction error $\hat{e}(k)$, & hence cannot be applied directly to RNN realizations of NARMA(p,q) network since inputs to network comprise delayed prediction error terms $\{\hat{e}\} \Rightarrow$ desirable to find another equivalent representation of NARMA(p,q) network, which would be more suited for RTRL-based learning.

If, for sake of clarity, denote predicted values $\hat{x}$ by y, i.e., to match notation common in RNNs with NARMA(p,q) theory, & have $y_1(k)=\hat{x}(k)$, & keep symbol x for exact values of input signal being predicted, NARMA network from (5.38), can be approximated further as (Connor 1994) [complicated (5.39)]. In that case, scheme shown in Fig. 5.16 should be redrawn, remaining topologically same, with y_1 replacing corresponding $\hat{e}$ terms among inputs to network.

On the other hand, alternative expression for conditional mean predictor, depicted in Fig. 5.16 can be written as

$$\hat{x}(k)=\Phi\left(\sum_{i=1}^{p} w_{1,i}(k)x(k-i)+w_{1,p+1}(k)+\sum_{j=p+2}^{p+q+1} w_{1,j}(k)\hat{x}(k+j-2-p-q)+\sum_{l=p+q+2}^{p+q+N} w_{1,l}(k)y_{l-p-q}(k-1)\right) \quad (296)$$

or, bearing in mind (5.39), notation used earlier (Haykin & Li 1995; Mandic et al. 1998) for examples on prediction of speech, i.e., $x(k)=s(k)$, & $y_1(k)=\hat{s}(k)$,

$$\hat{s}(k)=\Phi\left(\sum_{i=1}^{p} w_{1,i}(k)s(k-i)+w_{1,p+1}(k)+\sum_{j=p+2}^{p+q+1} w_{1,j}(k)y_1(k+j-2-p-q)+\sum_{l=p+q+2}^{p+q+N} w_{1,l}(k)y_{l-p-q}(k-1)\right), \quad (297)$$

which is common RNN lookalike notation. This scheme offers a simpler solution to NARMA(p,q) problem, as compared to previous one, since only nonlinear function used is activation function of a neuron Φ, while set of signals being processed is same as in previous scheme. Furthermore, scheme given in (5.41) & depicted in Fig. 5.17: Recurrent NARMA(p,q) implementation of prediction model resembles basic ARMA structure.

Li (1992) has shown: recurrent network of (5.41) with a sufficiently large number of neurons & appropriate weights can be found by performing RTRL algorithm s.t. sum of squared prediction errors $E<\delta$ for an arbitrary $\delta>0$. I.e., $\|\mathbf{s}-\hat{\mathbf{s}}\|_D<\delta$, where $\|\cdot\|_D$ denotes $\mathcal{L}_2$ norm w.r.t. training set D. Moreover, this scheme, shown also in Fig. 5.17, fits into well-known learning strategies, e.g. RTRL algorithm, which recommends this scheme for NARMA/NARMAX nonlinear prediction applications (Baldi & Atiya 1994; Draye et al. 1996; Kosmatopoulos et al. 1995; McDonnell & Waagen 1994; Nerrand et al. 1994; Wu & Niranjan 1994).

- Summary. A review of recurrent neural network architectures in fields of nonlinear dynamical modeling, system identification, control, signal processing, & forecasting has been provided. A relationship between neural network models & NARMA/NARMAX models, as well as Wiener & Hammerstein structures has been established. Particular attention has been devoted to fully connected recurrent neural network & its use in NARMA/NARMAX modeling has been highlighted.

- Neural Networks as Nonlinear Adaptive Filters.
  - Perspective. Neural networks, in particular recurrent neural networks, are cast into framework of nonlinear adaptive filters. In this context, relation between recurrent neural networks & polynomial filters is 1st established. Learning strategies & algorithms are then developed for neural adaptive system identifiers & predictors. Finally, discuss issues concerning choice of a neural architecture w.r.t. bias & variance of prediction performance.
  - Introduction. Representation of nonlinear systems in terms of NARMA/NARMAX models has been discussed at length in work of Billings & others (Billings 1980; Chen & Billings 1989; Connor 1994; Nerrand et al. 1994). Some cognitive aspects of neural nonlinear filters are provided in Maass & Sontag (2000). Pearson (1995), in his article on nonlinear input–output modeling, shows: block oriented nonlinear models are a subset of class of Volterra models. So, e.g., Hammerstein model, which consists of a static nonlinearity $f(\cdot)$ applied at output of a linear dynamical system described by its z-domain transfer function $H(z)$, can be represented[45] by Volterra series.

    In previous chapter, shown: neural networks, be they feedforward or recurrent, cannot generate time delays of an order higher than dimension of input to network. Another important feature: capability to generate subharmonics in spectrum of output of a nonlinear neural filter (Pearson 1995). Key property for generating subharmonics in nonlinear systems is recursion, hence, recurrent neural networks are necessary for their generation. Notice: as pointed out in Pearson (1995), block-stochastic models are, generally speaking, not suitable for this application.

    In Hakim et al. (1991), by using Weierstrass polynomial expansion theorem, relation between neural networks & Volterra series is established, which is then extended to a more general case & to continuous functions that cannot be expanded via a Taylor series expansion.[46] Both feedforward & recurrent networks are characterized by means of a Volterra series & vice versa.

    Neural networks are often referred to as 'adaptive neural networks'. Adaptive filters & neural networks are formally equivalent, & neural networks, employed as nonlinear adaptive filers, are generalizations of linear adaptive filters. However, in neural network applications, they have been used mostly in such a way that network is 1st trained on a particular

[45] Under condition: function f is analytic & Volterra series can be thought of as a generalized Taylor series expansion, then coefficients of model (6.2) that do not vanish are $h_{i,j,\ldots,z}\neq 0 \Leftrightarrow i=j=\cdots=z$.

[46] E.g. nonsmooth functions e.g. $|x|$.

training set & subsequently used. This approach is not an online adaptive approach, which is in contrast with linear adaptive filters, which undergo continual adaptation.
2 groups of learning techniques are used for training recurrent neural networks: a direct gradient computation technique (used in nonlinear adaptive filtering) & a recurrent backpropagation technique (commonly used in neural networks for offline applications). Real-time recurrent learning (RTRL) algorithm (Williams & Zipser 1989a) is a technique which uses direct gradient computation, & is used if the network coefficients change slowly with time. This technique is essentially an LMS learning algorithm for a nonlinear IIR filter. Notice: with same computation time, might be possible to unfold recurrent neural network into corresponding feedforward counterparts & hence to train it by backpropagation. Backpropagation through time (BPTT) algorithm is such a technique (Werbos 1990).
Some of benefits involved with neural networks as nonlinear adaptive filters are that no assumptions concerning Markov property, Gaussian distribution or additive measurement noise are necessary (Lo 1994). A neural filter would be a suitable choice even if mathematical models of input process & measurement noise are not known (black box modeling).

- Overview. Start with relationship between Volterra & bilinear filters & neural networks. Recurrent neural networks are then considered as nonlinear adaptive filters & neural architectures for this case are analyzed. Learning algorithms for online training of recurrent neural networks are developed inductively, starting from corresponding algorithms for linear adaptive IIR filters. Some issues concerning problem of vanishing gradient & bias/variance dilemma are finally addressed.
- Neural Networks & Polynomial Filters. Shown in Chap. 5: a small-scale neural network can represent high-order nonlinear systems, whereas a large number of terms are required for an equivalent Volterra series representation. E.g., as already shown, after performing a Taylor series expansion for output of a neural network depicted in Fig. 5.3, with input signals $u(k-1), u(k-2)$, obtain

$$y(k) = c_0 + c_1 u(k-1) + c_2 u(k-2) + c_3 u^2(k-1) + c_4 u^2(k-2) + c_5 u(k-1)u(k-2) + c_6 u^3(k-1) + c_7 u^3(k-2) + \cdots , \quad (298)$$

which has form of a general Volterra series, given by (6.2)

$$y(k) = h_0 + \sum_{i=0}^{N} h_1(i)x(k-i) + \sum_{i=0}^{N}\sum_{j=0}^{N} h_2(i,j)x(k-i)x(k-j) + \cdots . \quad (299)$$

Representation by a neural network is therefore more compact. As pointed out in Schetzen (1981), Volterra series are not suitable for modeling saturation type nonlinear functions & systems with nonlinearities of a high order, since they require a very large number of terms for an acceptable representation. Order of Volterra series & complexity of kernels $h(\cdot)$ increase exponentially with order of delay in system (6.2). This problem restricts practical applications of Volterra series to small-scale systems.
Nonlinear system identification, on the other hand, has been traditionally based upon Kolmogorov approximation theorem (neural network existence theorem), which states: a neural network with a hidden layer can approximate an arbitrary nonlinear system. Kolmogorov's theorem, however, is not that relevant in context of networks for learning (Girosi & Poggio 1989b). Problem: inner functions in Kolmogorov's formula (4.1), although continuous, have to be highly nonsmooth. Following analysis from Chap. 5, straightforward: multilayered & recurrent neural networks have ability to approximate an arbitrary nonlinear system, whereas Volterra series fail even for simple saturation elements.
Another convenient form of nonlinear system: bilinear (truncated Volterra) system described by

$$\boxed{y(k) = \sum_{j=1}^{N-1} c_j y(k-j) + \sum_{i=0}^{N-1}\sum_{j=1}^{N-1} b_{i,j} y(k-j)x(k-i) + \sum_{i=0}^{N-1} a_i x(k-i).} \quad (300)$$

Despite its simplicity, this is a powerful nonlinear model & a large class of nonlinear systems (including Volterra systems) can be approximated arbitrarily well using this model. Its functional dependence (6.3) shows: it belongs to a class of general recursive nonlinear models. A recurrent neural network that realizes a simple bilinear model is depicted in Fig. 6.1: Recurrent neural network representation of bilinear model. Multiplicative input nodes, denoted by $\times$, have to be introduced to represent bilinear model. Bias terms are omitted & chosen neuron is linear.

Problem 24. *Show: recurrent network shown in Fig. 6.1 realizes a bilinear model. Also show: this network can be described in terms of NARMAX models.*

Solution. Functional description of recurrent network depicted in Fig. 6.1 is given by

$$y(k) = c_1 y(k-1) + b_{0,1}x(k)y(k-1) + b_{1,1}x(k-1)y(k-1) + a_0 x(k) + a_1 x(k-1), \quad (301)$$

which belongs to class of bilinear models (6.3). Functional description of network from Fig. 6.1 can also be expressed as $y(k) = F(y(k-1), x(k), x(k-1))$, which is a NARMA representation of model (6.4). □

This example confirms duality between Volterra, bilinear, NARMA/NARMAX & recurrent neural models. To further establish connection between Volterra series & a neural network, express activation potential of nodes of network as

$$\mathrm{net}_i(k) = \sum_{j=0}^{M} w_{i,j} x(k-j), \quad (302)$$

where $\text{net}_i(k)$: activation potential of ith hidden neuron, $w_{i,j}$: weights, $x(k-j)$: inputs to network. If nonlinear activation functions of neurons are expressed via an Lth-order polynomial expansion[47] as

$$\Phi(\text{net}_i(k)) = \sum_{l=0}^{L} \xi_{il} \text{net}_i^l(k), \tag{303}$$

then neural model described in (6.6) & (6.7) can be related to Volterra model (6.2). Actual relationship is rather complicated, & Volterra kernels are expressed as sums of products of weights from input to hidden units, weights associated with output neuron, & coefficients ξ_{il} from (6.7). Chon et al. (1998) have used this kind of relationship to compare Volterra & neural approach when applied to processing of biomedical signals.
Hence, to avoid difficulty of excessive computation associated with Volterra series, an input–output relationship of a nonlinear predictor that computes output in terms of past inputs & outputs may be introduced as[48] (6.8)

$$\hat{y} = F(y(k-1), \ldots, y(k-N), u(k-1), \ldots, u(k-M)), \tag{304}$$

where $F(\cdot)$ is some nonlinear function. Function F may change for different input variables or for different regions of interest. A NARMAX model may therefore be a correct representation only in a region around some operating point. Leontaritis & Billings (1985) rigorously proved: a discrete time nonlinear time invariant system can always be represented by model (6.8) in vicinity of an equilibrium point provided that

∗ response function of system is finite realizable, &

∗ possible to linearize system around chosen equilibrium point.

Some of other frequently used models, e.g. bilinear polynomial filter, given by (6.3), are obviously cases of a simple NARMAX model.

- 6.5. Neural Networks & Nonlinear Adaptive Filters.

- 7. Stability Issues in RNN Architectures.
- 8. Data-Reusing Adaptive Learning Algorithms.
- 9. A Class of Normalized Algorithms for Online Training of Recurrent Neural Networks.
- 10. Convergence of Online Learning Algorithms in Neural Networks.
- 11. Some Practical Considerations of Predictability & Learning Algorithms for Various Signals.
- 12. Exploiting Inherent Relationships Between Parameters in Recurrent Neural Networks.
- Appendix A: The $\mathcal{O}$ Notation & Vector & Matrix Differentiation.
- Appendix B: Concepts from the Approximation Theory.
- Appendix C: Complex Sigmoid Activation Functions, Holomorphic Mappings & Modular Groups.
- Appendix D: Learning Algorithms for RNNs.
- Appendix E: Terminology Used in the Field of Neural Networks.
- Appendix F: On the *A Posteriori* Approach in Science & Engineering.
- Appendix G: Contraction Mapping Theorems.
- Appendix H: Linear GAS Relaxation.
- Appendix I: The Main Notions in Stability Theory.
- Appendix J: Deseasonalizing Time Series.

3. Robin M. Schmidt. *Recurrent Neural Networks (RNNs): A gentle Introduction & Overview.*

Abstract. State-of-the-art solutions in areas of "Language Modeling & Generating Text", "Speech Recognition", "Generating Image Descriptions" or "Video Tagging" have been using Recurrent Neural Networks as foundation for their approaches. Understanding underlying concepts is therefore of tremendous importance if want to keep up with recent or upcoming publications in those areas. In this work, give a short overview over some of most important concepts in realm of Recurrent Neural Networks which enables readers to easily understand fundamentals e.g. but not limited to "Backpropagation through Time" or "Long Short-Term Memory Units" as well as some of more recent advances like "Attention Mechanism" or "Pointer Networks". Also give recommendations for further reading regarding more complex topics where necessary.

- Introduction & Notation. Recurrent Neural Networks (RNNs) are a type of neural network architecture which is mainly used to detect patterns in a sequence of data. Such data can be handwriting, genomes, text or numerical time series which are often produced in industry settings (e.g. stock markets or sensors) [7, 12]. However, they are also applicable to images if these get respectively decomposed into a series of patches & treated as a sequence [12]. On a higher level, RNNs find applications in *Lagrange Modeling & Generating Text, Speech Recognition, Generating Image Descriptions* or *Video Tagging*. What differentiates Recurrent Neural Networks from Feeforward Neural Networks also known as Multi-Layer Perceptrons

[47] Using Weierstrass theorem, this expansion can be arbitrarily accurate. However, in practice resort to a moderate order of this polynomial expansion.
[48] This model is referred to as NARMAX model (nonlinear ARMAX), since it resembles linear model $\hat{y}(k) = a_0 + \sum_{j=1}^{N} a_j y(k-j) + \sum_{i=1}^{M} b_i u(k-i)$.

(MLPs) is how information gets passed through network. While Feedforward Networks pass information through network without cycles, RNN has cycles & transmits information back into itself. This enables them to extend functionality of Feedforward Networks to also take into account previous inputs $\mathbf{X}_{0:t-1}$ & not only current input $\mathbf{X}_t$. This difference is visualized on a high level in Fig. 1: Visualization of differences between Feedforward NNs & Recurrent NNs. Note: option of having multiple hidden layers is aggregated to 1 Hidden Layer block $\mathbf{H}$. This block can obviously be extended to multiple hidden layers.

Can describe this process of passing information from previous iteration to hidden layer with mathematical notation proposed in [24]. For that, denote hidden state & input at time step t resp. as $\mathbf{H}_t \in \mathbb{R}^{n \times h}, \mathbf{X}_t \in \mathbb{R}^{n \times d}$ where n: number of samples, d: number of inputs of each sample, & h: number of hidden units. Further, use a weight matrix $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, hidden-state-to-hidden-state matrix $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ & a bias parameter $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$. Lastly, all these informations get passed to a activation function ϕ which is usually a logistic sigmoid or tanh function to prepair gradients for usage in backpropagation. Putting all these notations together yields (1) as hidden variable & (2) as output variable.

$$\mathbf{H}_t = \phi_h(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h), \tag{305}$$

$$\mathbf{O}_t = \phi_o(\mathbf{H}_t \mathbf{W}_{ho} + \mathbf{b}_o). \tag{306}$$

Since $\mathbf{H}_t$ recursively includes $\mathbf{H}_{t-1}$ & this process occurs for every time step RNN includes traces of all hidden states that preceded $\mathbf{H}_{t-1}$ as well as $\mathbf{H}_{t-1}$ itself.

If compare that notation for RNNs with similar notation for Feedforward Neural Networks, can clearly see difference described earlier. In (3) can see computation for hidden variable while (4) shows output variable.

$$\mathbf{H} = \phi_h(\mathbf{X} \mathbf{W}_{xh} + \mathbf{b}_h), \tag{307}$$

$$\mathbf{O} = \phi_o(\mathbf{H} \mathbf{W}_{ho} + \mathbf{b}_o). \tag{308}$$

If you are familiar with training techniques for Feedforward Neural Networks e.g. backpropagation, 1 question: how to properly backpropagate error through a RNN. Here, a technique called Backpropagation Through Time (BPTT) is used which gets described in detail in next sect.

- 2. Backpropagation Through Time (BPTT) & Truncated BPTT. Backpropagation Through Time (BPTT) is adaptation of backpropagation algorithm for RNNs [24]. In theory, this unfolds RNN to construct a traditional Feedforward Neural Network where can apply backpropagation. For that, use same notations for RNN as proposed before.

  When forward pass input $\mathbf{X}_t$ through network compute hidden state $\mathbf{H}_t$ & output state $\mathbf{O}_t$ 1 step at a time. Can then define a loss function $\mathcal{L}(\mathbf{O}, \mathbf{Y})$ to describe difference between all outputs $\mathbf{O}_t$ & target values $\mathbf{Y}_t$ as shown in (5). This basically sums up every loss term l_t of each update step so far. This loss term l_t can have different defs based on specific problem (e.g. Mean Squared Error, Hinge Loss, Cross Entropy Loss, etc.). (5)

$$\mathcal{L}(\mathbf{O}, \mathbf{Y}) = \sum_{t=1}^{T} l_t(\mathbf{O}_t, \mathbf{Y}_t). \tag{309}$$

  Since have 3 weight matrices $\mathbf{W}_{xh}, \mathbf{W}_{hh}, \mathbf{W}_{ho}$ need to compute partial derivative w.r.t. each of these weight matrices. With chain rule which is also used in normal backpropagation get to result for $\mathbf{W}_{ho}$ shown in (6)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ho}} = \sum_{t=1}^{T} \frac{\partial l_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial \mathbf{W}_{ho}} = \sum_{t=1}^{T} \frac{\partial l_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \mathbf{H}_t. \tag{310}$$

  For partial derivative w.r.t. $\mathbf{W}_{hh}$ get

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hh}} = \cdots. \tag{311}$$

  For partial derivative w.r.t. $\mathbf{W}_{xh}$ get:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xh}} = \cdots. \tag{312}$$

  Since each $\mathbf{H}_t$ depends on previous time step, can substitute last part from above equations to get [(9)–(12)].

  From here, can see: need to store powers of $\mathbf{W}_{hh}^k$ as proceed through each loss term l_t of overall loss function $\mathcal{L}$ which can become very large. For these large values this method becomes numerically unstable since eigenvalues < 1 & eigenvalues > 1 diverge [5]. 1 method of solving this problem is truncate sum at a computationally convenient size [24]. When you do this, you're using Truncated BPTT [22]. This basically establishes an upper bound for number of time steps gradient can flow back to [15]. One can think of this upper bound as a moving window of past time steps which RNN considers. Anything before cut-off time step doesn't get taken into account. Since BPTT basically unfolds RNN to create a new layer for each time step, can also think of this procedure as limiting number of hidden layers.

- 3. Problems of RNNs: Vanishing & Exploring Gradients. As in most neural networks, vanishing or exploding gradients is a key problem of RNNs [12]. In (9)–(10) can see $\frac{\partial \mathbf{H}_t}{\partial \mathbf{H}_k}$ which basically introduces matrix multiplication over (potentially very long) sequence, if there are small values (< 1) in matrix multiplication this causes gradient to decrease with each layer (or time step) & finally vanish [6]. This basically stops contribution of states that happened far earlier than current time

step towards current time step [6]. Similarly, this can happen in opposite direction if have large values (> 1) during matrix multiplication causing an exploding gradient which is result values each weight too much & changes it heavily [6].

This problem motivated introduction of long short term memory units (LSTMs) to particularly handle vanishing gradient problem. This approach was able to outperform traditional RNNs on a variety of tasks [6]. In next sect, want to go deeper on proposed structure of LSTMs.

- 4. Long Short-Term Memory Units (LSTMs). Long Short-Term Memory Units (LSTMs) [9] were designed to properly handle vanishing gradient problem. Since they use a more constant error, they allow RNNs to learn over a lot more time steps (way over 1000) [12]. To achieve that, LSTMs store more information outside of traditional neural network flow in structures called *gated cells* [6, 12]. To make things work in an LSTM use an output gate $\mathbf{O}_t$ to read entries of cell, an input gate $\mathbf{I}_t$ to read data into cell & a forget gate $\mathbf{F}_t$ to reset content of cell. Computations for these gates are shown in (13)–(15):

$$\mathbf{O}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o), \tag{313}$$

$$\mathbf{I}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i), \tag{314}$$

$$\mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f). \tag{315}$$

Shown equations use $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d\times h}$ & $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h\times h}$ as weight matrices while $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1\times h}$ are their respective biases. Further, they use sigmoid activation function σ to transform output $\in (0, 1)$ which each results in a vector with entries $\in (0, 1)$.

Next, need a candidate memory cell $\widetilde{\mathbf{C}}_t \in \mathbb{R}^{n\times h}$ which has a similar computation as previously mentioned gates but instead uses a tanh activation function to have an output $\in (-1, 1)$. Further, it again has its own weights $\mathbf{W}_{xc} \in \mathbb{R}^{d\times h}$, $\mathbf{W}_{hc} \in \mathbb{R}^{h\times h}$ & biases $\mathbf{b}_c \in \mathbb{R}^{1\times h}$. Respective computation is shown in (16):

$$\widetilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c). \tag{316}$$

To plug some things together, introduce old memory content $\mathbf{C}_{t-1} \in \mathbb{R}^{n\times h}$ which together with introduced gates controls how much of old memory content we want to preserve to get to new memory content $\mathbf{C}_t$. This is shown in (17)

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \widetilde{\mathbf{C}}_t, \tag{317}$$

where $\odot$ denotes element-wise multiplication. Structure so far can be seen in Fig. 10 in Appendix A.

Last step: to introduce computation for hidden states $\mathbf{H}_t \in \mathbb{R}^{n\times h}$ into framework. This can be seen in (18):

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh \mathbf{C}_t. \tag{318}$$

With tanh function, ensure: each element of $\mathbf{H}_t \in (-1, 1)$. Full LSTM framework can be seen in Fig. 11: Computation of hidden state in an LSTM [24].

- 5. Deep Recurrent Neural Networks (DRNNs). Deep Recurrent Neural Networks (DRNNs) are in theory a really easy concept. To construct a deep RNN with L hidden layers, simply stack ordinary RNNs of any type on top of each other. Each hidden state $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n\times h}$ is passed to next time step of current layer $\mathbf{H}_{t+1}^{(l)}$ as well as current time step of next layer $\mathbf{H}_t^{(l+1)}$. For 1st layer, compute hidden state as proposed in previous models shown in (19) while for subsequent layer use (20) where hidden state from previous layer is treated as input.

$$\mathbf{H}_t^{(1)} = \phi_1(\mathbf{X}_t, \mathbf{H}_{t-1}^{(1)}), \tag{319}$$

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)}, \mathbf{H}_{t-1}^{(l)}). \tag{320}$$

Output $\mathbf{O}_t \in \mathbb{R}^{n\times o}$ where o: number of outputs is then computed as shown in (21)

$$\mathbf{O}_t = \phi_o(\mathbf{H}_t^{(L)}\mathbf{W}_{ho} + \mathbf{b}_o) \tag{321}$$

where only use hidden state of layer L.

- 6. Bidirectional Recurrent Neural Networks (BRNNs). Take an example of language modeling. Based on our current models, able to reliably predict next sequence element (i.e. next word) based on what we have seen so far. However, there scenarios where might want to fill in a gap in a sentence & part of sentence after gap conveys significant information. This information is necessary to take into account to perform well on this kind of task [24]. On a more generalized level, want to incorporate a look-ahead property for sequences.

  To achieve this look-ahead property Bidirectional Recurrent Neural Networks (BRNNs) [14] got introduced which basically add another hidden layer which run sequence backwards starting from last element [24]. An architectural overview is visualized in Fig. 2: Architecture of a bidirectional recurrent neural network. Introduce a forward hidden state $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n\times h}$ & a backward hidden state $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n\times h}$. Their respective calculations are shown in (22)–(23): [technical details]

  Keep in mind: 2 directions can have different number of hidden units.

- 7. Encoder-Decoder Architecture & Sequence to Sequence (`seq2seq`). Encoder-Decoder architecture is a type of neural network architecture where network is 2fold. It consists of encoder network & a decoder network whose respective roles are to *encode* input into a state & *decode* state to an output. This state usually has shape of a vector or a tensor [24]. A visualization of this structure in shown in Fig. 3: Encoder-Decoder Architecture Overview alternated from [24].

  Based on this Encoder-Decoder architecture a model called Sequence to Sequence (`seq2seq`) [16] got proposed for generating a sequence output based on a sequence input. This model uses RNNs for encoder as well as decoder where hidden state of encoder gets passed to hidden state of decoder. Common applications of model are Google Translate [16, 23], voice-enabled devices [13] or labeling video data [18]. It mainly focuses on mapping a fixed length input sequence of size n to an fixed length output sequence of size m where $n \neq m$ can be true but isn't a necessity.

  A de-rellod visualization of proposed architecture is shown in Fig. 4: Visualization of Sequence to Sequence (seq2seq) Model. Here, have a encoder which consists of a RNN accepting a single element of sequence $\mathbf{X}_t$ where t is order of sequence element. These RNNs can be LSTMs or Gated Recurrent Units (GRUs) to further improve performance [16]. Further, hidden states $\mathbf{H}_t$ are computed according to definition of hidden states in used RNN type (e.g. LSTM or GRU). Encoder Vector (context) is a representation of last hidden state of encoder network which aims to aggregate all information from all previous input elements. This functions as initial hidden state of decoder network of mode & enables decoder to make accurate predictions. Decoder network again is built of a RNN which predicts an output $\mathbf{Y}_t$ at a time step t. Produced output is again a sequence where each $\mathbf{Y}_t$ is a sequence element with order t. At each time step RNN accepts a hidden state from previous unit & itself produces an output as well as a new hidden state.

  Encoder Vector (context) was shown to be a bottleneck for these type of models since it needed to contain all necessary information of a source sentence in a fixed-length vector which was particularly problematic for long sequences. There have been approaches to solve this problem by introducing Attention in e.g. [4] or [10]. In Sect. 8, take a closer look at proposed solutions.

- 8. Attention Mechanism & Transformer. Attention Mechanism for RNNs is partly motivated by human visual focus & peripheral perception [21]. It allows humans to focus on a certain region to achieve high resolution while adjacent objects are perceived with a rather low resolution. Based on these focus points & adjacent perception, can make inference about what we expect to perceive when shifting our focus point. Similarly, can transfer this method on our sequence of words where we are able to perform inference based on observed words. E.g., if perceive the word *eating* in sequence "She is eating a green apple" we assume to observe a food object in the near future [21].

  Generally, Attention takes 2 sentences & transforms them into a matrix where each sequence element (i.e. a word) corresponds to a row or column. Based on this matrix layout, can fill in entries to identify relevant context or correlations between them. An example of this process can be seen in Fig. 5: Example of an Alignment matrix of "L'accord sur la zone économique européen a été signé en août 1992" (French) & its English translation "The agreement on the European Economic Area was signed in August 1992": [4].

  - 8.1. Def.
  - 8.2. Different types of score functions.
  - 8.3. Transformer.

- 9. Pointer Networks (Ptr-Nets). Pointer Networks (Ptr-Nets) [19] adapts `seq2seq` model with attention to improve it by not fixing discrete categories (i.e. elements) of output dictionary *a priori*. Instead of yielding an output sequence generated from an input sequence, a pointer networks creates a succession of pointers to elements of input series [25]. In [19] show: using Pointer Networks they can solve combinatorial optimization problems e.g. computing planar convex hulls, Delaunay triangulations & symmetric planar Traveling Salesman Problem (TSP).

  Generally, apply additive attention (from Table 1: Different score functions with their respective equations & usage alternated from [21]) between states & then normalize it by applying softmax function to model output conditional probability as seen in (29):

$$\mathbf{Y}_t = \text{softmax}(\text{score}(\mathbf{S}_t, \mathbf{H}_{t'})) = \text{softmax}(\mathbf{v}_a^\top \tanh \mathbf{W}_a[\mathbf{S}_t; \mathbf{H}_{t'}]). \tag{322}$$

  Attention mechanism is simplified, as Ptr-Net does not blend encoder states into output with attention weights. In this way, output only responds to positions but not input content [21].

- 10. Conclusion & Outlook. In this work, gave an introduction into fundamentals for Recurrent Neural Networks (RNNs). This includes general framework for RNNs, Backpropagation through time, problems of traditional RNNs, LSTMs, Deep & Bidirectional RNNs as well as more recent advances e.g. Encoder-Decoder Architecture, seq2seq model, Attention, Transformer & Pointer Networks. Most topics are only covered conceptionally & don't go too deep into implementation specifications. To get a broader understanding of covered topics, recommend looking into some of cited original papers. Additionally, most recent publications use some of presented concepts so recommend taking a look at such papers.

  1 recent publication which uses many of presented concepts is "Grandmaster level in StarCraft II using multi-agent reinforcement learning" by Vinyals et al. [20]. Here, they present their approach to train agents to play real-time strategy game Starcraft II with great success. If presented concepts were a little too theoretical for you, recommend reading that paper to see LSTMs, Transformer or Pointer Networks in a setting which can be deployed in a more practical environment.

7 Miscellaneous

7.1 Scholarpedia/recurrent neural networks

"A *recurrent neural network* (RNN) is any network whose neurons send feedback signals to each other. This concept includes a huge number of possibilities. A number of reviews already exist of some types of RNNs. These include:

- Wikipedia/recurrent neural network
- Recurrent Neural Networks for Temporal Data Processing editored by Hubert Cardot. The RNNs (Recurrent Neural Networks) are a general case of artificial neural networks where the connections are not feed-forward ones only. In RNNs, connections between units form directed cycles, providing an implicit internal memory. Those RNNs are adapted to problems dealing with signals evolving through time. Their internal memory gives them the ability to naturally take time into account. Valuable approximation results have been obtained for dynamical systems.
- Jürgen Schmidhuber/Recurrent Neural Networks.

Typically, these reviews consider RNNs that are artificial neural networks (aRNN) useful in technological applications. To complement these contributions, the present summary focuses on biological recurrent neural networks (bRNN) that are found in the brain. Since feedback is ubiquitous in the brain, this task, in full generality, could include most of the brain's dynamics. The current review divides bRNNS into those in which feedback signals occur in neurons within a single processing layer, which occurs in networks for such diverse functional roles as storing spatial patterns in short-term memory, winner-take-all decision making, contrast enhancement & normalization, hill climbing, oscillations of multiple types (synchronous, traveling waves, chaotic), storing temporal sequences of events in working memory, & serial learning of lists; & those in which feedback signals occur between multiple processing layers, e.g. occurs when bottom-up adaptive filters activate learned recognition categories & top-down learned expectations focus attention on expected patterns of critical features & thereby modulate both types of learning.

7.1.1 Types of Recurrent Neural Networks

There are at least 3 streams of bRNN research: binary, linear, & continuous-nonlinear (Grossberg, 1988):

1. Binary. Binary systems were inspired in part by neurophysiological observations showing that signals between many neurons are carried by all-or-none spikes. The binary stream was initiated by the classical McCulloch & Pitts (1943) model of threshold logic system that describes how the activities, or short-term memory (STM) traces, x_i of the ith node in a network interact in discrete time according to the equation:

$$x_i(t+1) = \text{sgn}\left(\sum_j A_{ij}x_j(t) - B_j\right).$$

The McCulloch-Pitts model had an influence far beyond the field of neural networks through its influence on von Neumann's development of the digital computer.

Caianiello (1961) used a binary STM equation that is influenced by activities at multiple times in the past:

$$x_i(T+\tau) = 1\left[\sum_{j=1}^{n}\sum_{k=0}^{l(m)} A_{ij}^{(k)}x_j(t-k\tau) - B_i\right],$$

where $l(w) = 1$ if $w \geq 0$ & 0 if $w < 0$.

Rosenblatt (1962) used an STM equation that evolves in continuous time, whose activities can spontaneously decay, & which can generate binary signals above a nonzero threshold:

$$\frac{d}{dt}x_i = -Ax_i + \sum_{j=1}^{n}\phi(B_j + x_j)C_{ij},$$

where $\phi(w)$ if $w \geq \theta$ & 0 if $w < \theta$. This equation was used in the classical Perceptron model.

Both Caianiello (1961) & Rosenblatt (1962) introduced equations to change the weights $A_{ij}^{(k)}$ in (2) & C_{ij} in (3) through learning. Such adaptive weights are often called *long-term memory* (LTM) traces. In both these models, interactions between STM & LTM were uncoupled in order to simplify the analysis. These LTM equations also had a digital aspect. the Caianiello (1961) LTM equations increased or decreased at constant rates until they hit finite upper or lower bounds. The Rosenblatt (1962) LTM equations were used to classify patterns into 2 distinct classes, as in the Perception Learning Theorem.

2. Linear. Widrow (1962) drew inspiration from the brain to introduce the gradient Adeline adaptive pattern recognition machine. Anderson (1968) initially described his intuitions about neural pattern recognition using a spatial cross-correlation function. Concepts from linear system theory were adapted to represent some aspects of neural dynamics, including solutions of simultaneous linear equations $Y = AX$ using matrix theory, & concepts about cross-correlation. Kohonen (1971) made a transition from linear algebra concepts e.g. the Moore-Penrose pseudoinverse to more biologically motivated studies that are summarized in his books (Kohonen, 1977, 1984). These ideas began with a mathematically familiar engineering framework before moving towards more biologically motivated nonlinear interactions.

3. Continuous-Nonlinear. Continuous-nonlinear network laws typically arose from an analysis of behavioral or neural data. Neurophysiological experiments on the lateral eye of the Limulus, or horseshoe crab, led to the award of a Nobel prize to H. K. Hartline. These data inspired the steady state Ratliff model (Hartline & Ratliff, 1957):

$$r_i = e_i - \sum_{j=1}^{n} k_{ij}[r_j - r_{ij}]^+, \tag{323}$$

where $[w]^+ \coloneqq \max\{w, 0\}$. Equation (4) describes how cell activations e_i are transformed into smaller net responses r_i due to recurrent inhibitory threshold-linear signals $-k_{ij}[r_i - r_{ij}]^+$. The Hartline-Ratliff model is thus a kind of continuous threshold-logic system. Ratliff et al. (1963) extended this steady-state model to a dynamical model:

$$r_i(t) = e_i(t) - \sum_{j=1}^{n} k_{ij} \left[\frac{1}{\tau} \int_0^t e^{-\frac{t-s}{\tau}} r_j(s)\, \mathrm{d}s - r_{ij} \right]^+, \tag{324}$$

which also behaves linearly above threshold. This model is a precursor of the Additive Model that is described below.

Another classical tradition arose from the analysis of how the excitable membrane of a single neuron can generate electrical spikes capable of rapidly & non-decrementally traversing the axon, or pathway, from 1 neuron's cell body to a neuron to which it is sending signals. This experimental & modeling work on the squid giant axon by Hodgkin & Huxley (1952) also led to the award of a Nobel prize. Since this work focused on individual neurons rather than neural networks, it will not be further discussed herein except to note that it provides a foundation for the Shunting Model described below.

Another source of continuous-nonlinear RNNs arose through a study of adaptive behavior in real time, which led to the derivation of neural networks that form the foundation of most current biological neural network research (Grossberg, 1967, 1968b, 1968c). These laws were discovered in 1957–58 when Grossberg, then a college Freshman, introduced the paradigm of using nonlinear systems of differential equations to model how brain mechanisms can control behavioral functions. The laws were derived from an analysis of how psychological data about human & animal learning can arise in an individual learner adapting autonomously in real time. Apart from the Rockerfeller Institute student monograph Grossberg (1964), it took a decade to get them published.

4. Additive STM equation. The following equation is called the Additive Model because it adds the terms, possibly nonlinear, that determine the rate of change of neuronal activities, or potentials, x_i:

$$\frac{d}{dt}x_i = -A_i x_i + \sum_{j=1}^{n} f_j(x_j) B_{ji} z_{ji}^{(+)} - \sum_{j=1}^{n} g_j(x_j) C_{ji} z_{ji}^{(-)} + I_i. \tag{325}$$

Equation (6) includes a term for passive decay $-A_i x_i$, positive feedback $\sum_{j=1}^{n} f_j(x_j) B_{ji} z_{ji}^{(+)}$, negative feedback $-\sum_{j=1}^{n} g_j(x_j) C_{ji} z_{ji}^{(-)}$, & input I_i. Each feedback term includes an activity-dependent (possibly) nonlinear signal $(f_j(x_j), g_j(x_j))$; a connection, or path, strength (B_{ji}, C_{ji}), & an adaptive weight, or LTM trace $(z_{ij}^{(+)}, z_{ij}^{(-)})$. If the positive & negative feedback terms are lumped together & the connection strengths are lumped with the LTM traces, then the Additive Model may be written in the simpler form:

$$\frac{d}{dt}x_i = -A_i x_i + \sum_{j=1}^{n} f_j(x_j) z_{ji} + I_i. \tag{326}$$

Early applications of the Additive Model included computational analyses of vision, learning, recognition, reinforcement learning, & learning of temporal order in speech, language, & sensory-motor control (Grossberg, 1969b, 1969c, 1969d, 1970a, 1970b, 1971a, 1971b, 1972a, 1972b, 1974, 1975; Grossberg & Pepe, 1970, 1971). The Additive Model has continued to be a cornerstone of neural network research to the present time; e.g., in decision-making (Usher & McClelland, 2001). Physicists & engineers unfamiliar with the classical status of the Additive Model in neural networks called it the Hopfield model after the 1st application of this equation in Hopfield (1984). Grossberg (1988) summarizes historical factors that contributed to their unfamiliarity with the neural network literature. The Additive Model in (7) may be generalized in many ways, including the effects of delays & other factors. In the limit of infinitely many cells, an abstraction which does not exist in the brain, the discrete sum in (7) may be replaced by an integral (see Neural fields).

5. Shunting STM equation. Grossberg (1964, 1968b, 1969b) also derived an STM equation for neural networks that more closely model the shunting dynamics of individual neurons (Hodgkin, 1964). In such a shunting equation, each STM trace is bounded within an interval $[-D, B]$. Automatic gain control, instantiated by multiplicative shunting, or mass action, terms, interacts with balanced positive & negative signals & inputs to maintain the sensitivity of each STM trace within its interval (see The Noise-Saturation Dilemma):

$$\frac{d}{dt}x_i = -A_i x_i + (B - x_i)\left[\sum_{j=1}^{n} f_j(x_j) C_{ji} z_{ji}^{(+)} + I_i\right] - (D + x_i)\left[\sum_{j=1}^{n} g_j(x_j) E_{ji} z_{ji}^{(-)} + J_i\right]. \tag{327}$$

The Shunting Model is approximated by the Additive Model in cases where the inputs are sufficiently small that the resulting activities x_i do not come close to their saturation values – D, B.

The Wilson-Cowan model (Wilson & Cowan, 1972) also uses a combination of shunting & additive terms, as in (8). However, instead of using sums of sigmoid signals that are multiplied by shunting terms, as in RHS of (8), the Wilson-Cowan model uses a sigmoid of sums that is multiplied by a shunting term, as in the expression $(B - x_i)f_j(\sum_j C_{ji}x_j z_{ji}^{(+)} - x_j E_{ji} z_{ji}^{(-)} + I_i)$. This form can saturate activities when inputs or recurrent signals get large, unlike (8), as noted in Grossberg (1973).

6. Generalized STM equation. Equations (6) & (8) are special cases of an STM equation, introduced in Grossberg (1968c), which includes LTM & medium-term memory (MTM) terms that changes at a rate intermediate between the faster STM & the slower LTM. The laws for STM, MTM, & LTM are specialized to deal with different evolutionary pressures in neural models of different brain systems, including additional factors e.g. transmitter mobilization (Grossberg, 1969c, 1969b). This generalized STM equation is:

$$\frac{dx_i}{dt} = -Ax_i + (B - Cx_i)\left[\sum_{k=1}^{n} f_k(x_k)D_{ki}y_{ki}z_{ki} + I_i\right] - (E + Fx_i)\left[\sum_{k=1}^{n} g_k(x_k)G_{ki}Y_{ki}Z_{ki} + J_i\right]. \tag{328}$$

In the shunting model, the parameters $C \neq 0, F \neq 0$. The parameter $E = 0$ when there is "silent" shunting inhibition, whereas $E \neq 0$ describes th case of hyperpolarizing shunting inhibition. In the Additive Model, parameters $C = F = 0$. The excitatory interaction term $[\sum_{k=1}^{n} f_k(x_k)D_{ki}y_{ki}z_{ki} + I_i]$ describes an external input I_i plus the total excitatory feedback signal $[\sum_{k=1}^{n} f_k(x_k)D_{ki}y_{ki}z_{ki}]$ that is a sum of signals from other populations via their output signals $f_k(x_k)$. The term D_{ki} is a constant connection strength between cell populations v_k, v_i, whereas terms y_{ki} & z_{ki} describe MTM & LTM variables, resp. The inhibitory interaction term $[\sum_{k=1}^{n} g_k(x_k)G_{ki}Y_{ki}Z_{ki} + J_i]$ has a similar interpretation. Equation (9) assumes "fast inhibition", i.e., inhibitory

7. MTM: Habituative Transmitter Gates & Depressing Synapses.

8. LTM: Gated steepest descent learning: Not Hebbian learning.

7.1.2 Processing & STM of Spatial Patterns

1. Transformation & short-term storage of distributed input patterns by neural networks.

2. The Noise-Saturation Dilemma.

3. A though experiment to solve the noise-saturation dilemma.

4. Automatic gain control by the off surround prevents saturation.

5. Contrast normalization & pattern processing by real-time probabilities.

6. Weber Law & shift property.

7. Physiological interpretation of shunting dynamics: The membrane equation of neurophysiology.

8. Recurrent competitive fields.

9. Winner-take-all, contrast enhancement, normalization, & quenching threshold.

10. Shunting dynamics in cortical models.

11. Decision-making in Competitive Systems: Liapunov methods.

12. Competition, decision, & consensus.

13. Adaptation level systems: Globally-consistent decision-making.

14. Cohen-Grossberg model, Liapunov function, & theorem.

15. Symmetry does not imply convergence: Synchronized oscillations.

16. Unifying horizontal, bottom-up, & top-down STM & LTM interactions.

7.1.3 Interactions of STM & LTM during Neuronal Learning

1. Unbiased spatial pattern learning by Generalized Additive RNNs.

2. Outstar learning theorem.

3. Sparse stable category learning theorem.

4. Adaptive bidirectional associative memory.

5. Adaptive resonance theory.

7.1.4 Working memory: processing & STM of temporal sequences

1. Relative activity codes temporal order in working memory.
2. Working memory design enables stable learning of list chunks.
3. LTM Invariance & Normalization rule are realized by specialized RCFs.
4. Primacy, recency, & bowed activation gradients.
5. Experimental support.
6. Stable chunk learning implies the Magical Numbers 4 & 7.
7. Equations for some Item-&-Order RNNs.

7.1.5 Serial Learning: From Command Cells to values, Decisions, & Plans

1. Avalanches.
2. Command cells & nonspecific arousal.
3. Self-organizing avalanches: Instar-outstar maps & serial learning of temporal order.
4. Context-Sensitive Self-Organizing Avalanches: What categories control temporal order?
5. Serial learning.

" – Scholarpedia/recurrent neural networks

8 Wikipedia's

8.1 Wikipedia/large language model

"A *large language modle (LLM)* is a type of ML model designed for natural language processing tasks e.g. language generation. LLMs are language models with many parameters, & are trained with self-supervised learning on a vast amount of text.

The largest & most capable LLMs are generative pretrained transformers (GPTs). Modern models can be fine-tuned for specific tasks or guided by prompt engineering. These models acquire predictive power regarding syntax, semantics, & ontologies inherent in human language corpora, but they also inherit inaccuracies & biases present in data they are trained in.

8.1.1 History

8.1.2 Dataset preprocessing

8.1.3 Training & architecture

8.1.4 Training cost

Qualifier "large" in "large language model" is inherently vague, as there is no definitive threshold for number of parameters required to qualify as "large". As time goes on, what was previously considered "large" may evolve. GPT-1 of 2018 is usually considered 1st LLM, even though it has only 0.117 billion parameters. Tendency towards larger models is visible in list of LLMs.

Advances in software & hardware have reduced cost substantially since 2020, s.t. in 2023 training of a 12-billion-parameter LLM computational cost is 72300 A100-GPU-hours, while in 2020 cost of training a 1.5-billion-parameter LLM (which was 2 orders of magnitude smaller than state of art in 2020) was between $80,000 & $1,600,000. Since 2020, large sums were invested in increasingly large models. E.g., training of GPT-2 (i.e., a 1.5-billion-parameters model) in 2019 cost $50,000, while training of PaLM (i.e. a 540-billion-parameters model) in 2022 cost $8 million, & Megatron-Turing NLG 530B (in 2021) cost around $11 million.

For Transformer-based LLM, training cost is much higher than inference cost. It costs 6 FLOPs per parameter to train on 1 token, whereas it costs 1–2 FLOPs per parameter to infer on 1 token.

8.1.5 Tool use

There are certain tasks that, in principle, cannot be solved by any LLM, at least not without use of external tools or additional software. An example of such a task is responding to user's input `354 * 139 =`, provided that LLM has not already encountered a continuation of this calculation in its training corpus. In such cases, LLM needs to resort to running program code that calculates result, which can then be included in its response. Another example is "What is time now? It is", where a separate program interpreter would need to execute a code to get system time on computer, so that LLM can include it in its rely. This basic strategy can be sophisticated with multiple attempts of generated programs, & other sampling strategies.

Generally, in order to get an LLM to use tools, one must fine-tune it for tool-use. If number of tools is finite, then fine-tuning may be done just once. If number of tools can grow arbitrarily, as with online API services, then LLM can be fine-tuned to be able to read API documentation & call API correctly.

A simple form of tool use is retrieval-augmented generation: augmentation of an LLM with document retrieval. Given a query, a document retriever is called to retrieve most relevant documents. This is usually done by encoding query & documents into vectors, then finding documents with vectors (usually stored in a vector database) most similar to vector of query. LLM then generates an output based on both query & context included from retrieved documents.

8.1.6 Agency

8.1.7 Compression

8.1.8 Multimodality

8.1.9 Properties

8.1.10 Interpretation

8.1.11 Evaluation

8.1.12 Wider impact

" – Wikipedia/large language model

8.2 Wikipedia/recurrent neural network

"*Recurrent neural networks (RNNs)* are a class of artificial neural network commonly used for sequential data processing. Unlike feedforward neural networks, which process data in a single pass, RNNs process data across multiple time steps, making them well-adapted for modeling & processing text, speech, & time series.

– *Mạng nơ-ron hồi quy (RNN)* là 1 lớp mạng nơ-ron nhân tạo thường được sử dụng để xử lý dữ liệu tuần tự. Không giống như mạng nơ-ron truyền thẳng, xử lý dữ liệu trong 1 lần chạy, RNN xử lý dữ liệu qua nhiều bước thời gian, khiến chúng thích ứng tốt với việc mô hình hóa & xử lý văn bản, giọng nói, & chuỗi thời gian.

The building block of RNNs is the *recurrent unit*. This unit maintains a hidden state, essentially a form of memory, which is updated at each time step based on the current input & the previous hidden state. This feedback loop allows the network to learn from past inputs, & incorporate that knowledge into its current processing.

– Khối xây dựng của RNN là *đơn vị tái diễn*. Đơn vị này duy trì trạng thái ẩn, về cơ bản là 1 dạng bộ nhớ, được cập nhật tại mỗi bước thời gian dựa trên đầu vào hiện tại & trạng thái ẩn trước đó. Vòng phản hồi này cho phép mạng học hỏi từ các đầu vào trước đó, & kết hợp kiến thức đó vào quá trình xử lý hiện tại của nó.

Early RNNs suffered from the vanishing gradient problem, limiting their ability to learn long-range dependencies. This was solved by the long short-term memory (LSTM) variant in 1997, thus making it the standard architecture for RNN.

– Các RNN ban đầu gặp phải vấn đề độ dốc biến mất, hạn chế khả năng học các phụ thuộc tầm xa. Vấn đề này đã được giải quyết bằng biến thể bộ nhớ dài hạn ngắn hạn (LSTM) vào năm 1997, do đó trở thành kiến trúc tiêu chuẩn cho RNN.

RNNs have been applied to tasks e.g. unsegmented, connected handwriting recognition, speech recognition, natural language processing, & neural machine translation.

– RNN đã được áp dụng cho các nhiệm vụ như nhận dạng chữ viết tay không phân đoạn, có kết nối, nhận dạng giọng nói, xử lý ngôn ngữ tự nhiên và dịch máy thần kinh.

8.2.1 History

- Before modern. 1 origin of RNN was neuroscience. The word "recurrent" is used to describe loop-like structures in anatomy. In 1901, Cajal observed "recurrent semicircles" in the cerebellar cortex formed by parallel fiber, Purkinje cells, & granule cells. In 1933, Lorente de Nó discovered "recurrent, reciprocal connections" by Golgi's method, & proposed that excitatory loops explain certain aspects of the vestibulo-ocular reflex. During 1940s, multiple people proposed the existence of feedback in the brain, which was a contrast to the previous understanding of the neural system as a purely feedforward structure. Hebb considered "reverberating circuit" as an explanation for short-term memory. The McCulloch & Pitts paper (1943), which proposed the McCulloch-Pitts neuron model, considered networks that contains cycles. The current activity of such networks can be affected by activity indefinitely far in the past. They were both interested in closed loops as possible explanations for e.g. epilepsy & causalgia. Recurrent inhibition was proposed in 1946 as a negative feedback mechanism in motor control. Neural feedback loops were a common topic of discussion at the Macy conferences. Grossberg, Stephen (2013-02-22). "Recurrent Neural Networks". Scholarpedia: An extensive review of recurrent neural network models in neuroscience.

  A close-loop cross-coupled perceptron network. Frank Rosenblatt in 1960 published "close-loop cross-coupled perceptrons", which are 3-layered perceptron networks whose middle layer contains recurrent connections that change by a Hebbian learning rule. Later, in *Principles of Neurodynamics* (1961), he described "closed-loop cross-coupled" & "back-coupled" perceptron networks, & made theoretical & experimental studies for Hebbian learning in these networks, & noted that a fully cross-coupled perceptron network is equivalent to an infinitely deep feedforward network.

  Similar networks were published by Kaoru Nakano in 1971, Shun'ichi Amari in 1972, & William A. Little in 1974, who was acknowledged by Hopfield in his 1982 paper.

Another origin of RNN was statistical mechanics. The Ising model was developed by Wilhelm Lenz & Ernest Ising in the 1920s as a simple statistical mechanical model of magnets at equilibrium. Glauber in 1963 studied by Ising model evolving in time, as a process towards equilibrium (Glauber dynamics), adding in the component of time.

The Sherrington-Kirkpatrick model of spin glass, published in 1975, is the Hopfield network with random initialization. Sherrington & Kirkpatrick found that it is highly likely for the energy function of the SK model to have many local minima. In the 1982 paper, Hopfield applied this recently developed theory to study the Hopfield network with binary activation functions. In a 1984 paper he extended this to continuous activation functions. It became a standard model for the study of neural networks through statistical mechanics.

- Modern. Modern RNN networks are mainly based on 2 architectures: LSTM & BRNN.

  At the resurgence of neural networks in the 1980s, recurrent networks were studied again. They were sometimes called "iterated nets". 2 early influential works were the Jordan network (1986) & the Elman network (1990), which applied RNN to study cognitive psychology. In 1993, a neural history compressor system solved a "Very Deep Learning" task that required > 1000 subsequent layers in an RNN unfolded in time.

  Long short-term memory (LSTM) networks were invented by Hochreiter & Schmidhuber in 1995 & set accuracy records in multiple applications domains. It became the default choice for RNN architecture.

  Bidirectional recurrent neural networks (BRNN) uses 2 RNN that processes the same input in opposite directions. These 2 are often combined, giving the bidirectional LSTM architecture.

  Around 2006, bidirectional LSTM started to revolutionize speech recognition, outperforming traditional models in certain speech applications. They also improved large-vocabulary speech recognition & text-to-speech synthesis & was used in Google voice search, & dictation on Android devices. They broke records for improved machine translation, language modeling, & Multilingual Language Processing. Also, LSTM combined with convolutional neural networks (CNNs) improved automatic image captioning.

  The idea of encoder-decoder sequence transduction had been developed in the early 2010s. The papers most commonly cited as the originators that produced `seq2seq` are 2 papers from 2014. A seq2seq architecture employs 2 RNN, typically LSTM, an "encoder" & a "decoder", for sequence transduction, e.g. machine translation. They became state of the art in machine translation, & was instrumental in the development of attention mechanism & Transformer.

8.2.2 Configurations

Main article: Layer (deep learning). An RNN-based model can be factored into 2 parts: configuration & architecture. Multiple RNN can be combined in data flow, & the data flow itself is the configuration. Each RNN itself may have any architecture, including LSTM, GRU, etc.

– Một mô hình dựa trên RNN có thể được chia thành hai phần: cấu hình & kiến trúc. Nhiều RNN có thể được kết hợp trong 1 luồng dữ liệu & luồng dữ liệu đó chính là cấu hình. Mỗi RNN có thể có bất kỳ kiến trúc nào, bao gồm LSTM, GRU, v.v.

- Standard. Compressed (left) & unfolded (right) basic recurrent neural network. RNNs come in many variants. Abstractly speaking, an RNN is a function f_θ of type $(x_t, h_t) \mapsto (y_t, h_{t+1})$, where x_t: input vector, h_t: hidden vector, y_t: output vector, θ: neural network parameters. In words, it is a neural network that maps an input x_t into an output y_t, with the hidden vector h_t playing the role of "memory", a partial record of all previous input-output pairs. At each step, it transforms input to an output, & modifies its "memory" to help it to better perform future processing.

  The illustration to the right may be misleading to many because practical neural network topologies are frequently organized in "layers" & the drawing gives that appearance. However, what appears to be layers are, in fact, different steps in time, "unfolded" to produce the appearance of layers.

- Stacked RNN. A *stacked RNN*, or *deep RNN*, is composed of multiple RNNs stacked one above the other. Abstractly, it is structured as follows

  1. Layer 1 has hidden vector $h_{1,t}$, parameters θ_1 & maps $f_{\theta_1} : (x_{0,t}, h_{1,t}) \mapsto (x_{1,t}, h_{1,t+1})$.
  2. Layer 2 has hidden vector $h_{2,t}$, parameters θ_2, & maps $f_{\theta_2} : (x_{1,t}, h_{2,t}) \mapsto (x_{2,t}, h_{2,t+1})$.
  3. ...
  4. Layer n has hidden vector $h_{n,t}$, parameters θ_n, & maps $f_{\theta_n} : (x_{n-1,t}, h_{n,t}) \mapsto (x_{n,t}, h_{n,t+1})$.

  Each layer operates as a stand-alone RNN, & each layer's output sequence is used as the input sequence to the layer above. There is no conceptual limit to the depth of stacked RNN.

- Bidirectional. Main article: Wikipedia/bidirectional recurrent neural networks. A *bidirectional RNN* (biRNN) is composed of 2 RNNs, one processing the input sequence in 1 direction, & another in the opposite direction. Abstractly, it is structured as follows:

  - The forward RNN processes in 1 direction: $f_\theta(x_0, h_0) = (y_0, h_1), f_\theta(x_1, h_1) = (y_1, h_2), \dots$
  - The backward RNN processes in the opposite direction: $f'_{\theta'}(x_N, h'_N) = (y'_N, h'_{N-1}), f'_{\theta'}(x_{N-1}, h'_{N-1}) = (y'_{N-1}, h'_{N-2}), \dots$

The 2 output sequences are then concatenated to give the total output: $((y_0, y'_0), (y_1, y'_1), \dots, (y_N, y'_N))$.

Bidirectional RNN allows the model to process a token both in the context of what came before it & what came after it. By stacking multiple bidirectional RNNs together, the model can process a token increasingly contextually. The ELMo model (2018) is a stacked bidirectional LSTM which takes character-level as inputs & produces word-level embeddings.

- Encoder-decoder. Main article: seq2seq. A decoder without an encoder. 2 RNNs can be run front-to-back in an *encoder-decoder* configuration. The encoder RNN processes an input sequence into a sequence of hidden vectors, & the decoder RNN processes the sequence of hidden vectors to an output sequence, with an optional attention mechanism. This was used to construct state of the art neural machine translators during the 2014–2017 period. This was an instrumental step towards the development of Transformers. Encoder-decoder RNN without attention mechanism. Encoder-decoder RNN with attention mechanism.

- PixelRNN. An RNN may process data with more than 1D. PixelRNN processes 2D data, with many possible directions. E.g., the row-by-row direction processes $n \times n$ grid fo vectors $x_{i,j}$ in the following order: $x_{1,1}, x_{1,2}, \dots, x_{1,n}, x_{2,1}, x_{2,2}, \dots, x_{2,n}, \dots, x_{n,n}$. The *diagonal BiLSTM* uses 2 LSTMs to process the same grid. One processes it from the top-left corner to the bottom-right, s.t. it processes $x_{i,j}$ depending on its hidden state & cell state on the top & the left side: $h_{i-1,j}, c_{i-1,j}$ & $h_{i,j-1}, c_{i,j-1}$. The other processes it from the top-right corner to the bottom-left.

8.2.3 Architectures

- Fully recurrent. A fully connected RNN with 4 neurons. *Fully recurrent neural networks* (FRNN) connect the outputs of all neurons to the inputs of all neurons. I.e., it is a fully connected network. This is the most general neural network topology, because all other topologies can be represented by setting some network topology, because all other topologies can be represented by setting some connection weights to 0 to simulate the lack of connections between those neurons.

- Hopfield. Main article: Wikipedia/Hopfield network. The Hopfield network is an RNN in which all connections across layers are equally sized. It requires stationary inputs & is thus not a general RNN, as it does not process sequences of patterns. However, it guarantees that it will converge. If the connections are trained using Hebbian learning, then the Hopfield network can perform as robust content-addressable memory, resistant to connection alteration.

- Elman networks & Jordan networks. A simple Elman network where $\sigma_h = \tanh, \sigma_y =$ Identity. An Elman network is a 3-layer network (arranged horizontally as x, y, z in the illustration) with the addition of a set of context units (u in the illustration). The middle (hidden) layer is connected to these context units fixed with a weight of 1. At each time step, the input is fed forward & a learning rule is applied. The fixed back-connections save a copy of the previous values of the hidden units in the context units (since they propagate over the connections before the learning rule is applied). Thus the network can maintain a sort of state, allowing it to perform tasks e.g. sequence-prediction that are beyond the power of a standard multilayer perceptron.

  Jordan networks are similar to Elman networks. The contexts units are fed from the output layer instead of the hidden layer. The context units in a Jordan network are also called the *state layer*. They have a recurrent connection to themselves.

  Elman & Jordan networks are also known as "Simple recurrent networks" (SRN).

  Elman network:

$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h), \ y_t = \sigma_y(W_y h_t + b_y). \tag{Elman nw}$$

  Jordan network:

$$h_t = \sigma_h(W_h x_t + U_h s_t + b_h), \ y_t = \sigma_y(W_y h_t + b_y), \ s_t = \sigma_s(W_{s,s} s_{t-1} + S_{s,y} y_{t-1} + b_s). \tag{Jordan nw}$$

  Variables & functions: x_t: input vector, h_t: hidden layer vector, s_t: "state" vector, y_t: output vector, W, U, b: parameter matrices & vector, σ: activation functions.

- Long short-term memory. Long short-term memory unit. Main article: Wikpedia/long short-term memory. *Long short-term memory* (LSTM) is the most widely used RNN architecture. It was designed to solve the vanishing gradient problem. LSTM is normally augmented by recurrent gates called "forget gates". LSTM prevents backpropagated errors from vanishing or exploding. Instead, errors can flow backward through unlimited numbers of virtual layers unfolded in space. I.e., LSTM can learn tasks that require memories of events that happened thousands or even millions of discrete time steps earlier. Problem-specific LSTM-like topologies can be evolved. LSTM works even given long delays between significant events & can handle signals that mix low & high-frequency components.

  Many applications use stacks of LSTMs, for which it is called "deep LSTM". LSTM can learn to recognize context-sensitive languages unlike previous models based on hidden Markov models (HMM) & similar concepts.

- Gated recurrent unit. Gated recurrent unit. Main article: Wikipedia/gated recurrent unit. *Gated recurrent unit* (GRU), introduced in 2014, was designed as a simplification of LSTM. They are used in the full form & several further simplified variants. They have fewer parameters than LSTM, as they lack an output gate.

  Their performance on polyphonic music modeling & speech signal modeling was found to be similar to that of long short-term memory. There does not appear to be particular performance difference between LSTM & GRU.

- Bidirectional associative memory. Main article: Wikipedia/bidirectional associative memory. Introduced by Bart Kosko, a bidirectional associative memory (BAM) network is a variant of a Hopfield network that stores associative data as a vector. The bidirectionality comes from passing information through a matrix & its transpose. Typically, bipolar encoding is preferred to binary encoding of the associative pairs. Recently, stochastic BAM models using Markov stepping were optimized for increased network stability & relevance to real-world applications.

  A BAM network has 2 layers, either of which can be driven as an input to recall an association & produce an output on the other layer.

- Echo state. Main article: Wikipedia/echo state network. Echo state network (ESN) have a sparsely connected random hidden layer. The weights of output neurons are the only part of the network that can change (be trained). ESNs are good at reproducing certain time series. A variant for spiking neurons is known as a liquid state machine.

- Recursive. Main article: Wikipedia/recursive neural network. A recursive neural network is created by applying the same set of weights recursively over a differentiable graph-like structure by traversing the structure in topological order. Such networks are typically also trained by the reverse mode of automatic differentiation. They can process distributed representations of structure, e.g. logical terms. A special case of recursive neural networks is the RNN whose structure corresponds to a linear chain. Recursive neural networks have been applied to natural language processing. The Recursive Neural Tensor Network uses a tensor-based composition function for all nodes in the tree.

- Neural Turing machines. Main articles: Wikipedia/neural Turing machine & Wikipedia/differentiable neural computer. *Neural Turing machines* (NTMs) are a method of extending recurrent neural networks by coupling them to external memory resources with which they interact. The combined system is analogous to a Turing machine or Von Neumann architecture but is differentiable end-to-end, allowing it to be efficiently trained with gradient descent.

  Differentiable neural computers (DNCs) are an extension of Neural Turing machines, allowing for the usage of fuzzy amounts of each memory address & a record of chronology.

  Neural network pushdown automata (NNPDA) are similar to NTMs, but tapes are replaced by analog stacks that are differentiable & trained. In this way, they are similar in complexity to recognizers of context free grammars (CFGs).

  Recurrent neural networks are Turing complete & can run arbitrary programs to process arbitrary sequences of inputs.

8.2.4 Training

- Teacher forcing. Encoder-decoder RNN without attention mechanism. Teacher forcing is shown in red. An RNN can be trained into a conditionally generative model of sequences, aka *autoregression*.

  Concretely, let us consider the problem of machine translation, i.e., given a sequence $(x_1, x_2, \ldots, x_n)$ of English words, the model is to produce a sequence $(y_1, \ldots, y_m)$ of French words. It is to be solved by a seq2seq model.

  Now, during training, the encoder half of the model would 1st ingest $(x_1, x_2, \ldots, x_n)$, then the decoder half would start generating a sequence $(\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_l)$. The problem is that if the model makes a mistake early on, say at $\hat{y}_2$, then subsequent tokens are likely to also be mistakes. This makes it inefficient for the model to obtain a learning signal, since the model would mostly learn to shift $\hat{y}_2$ towards y_2, but not the others.

  Teacher forcing makes it so that the decoder uses the correct output sequence for generating the next entry in the sequence. So e.g., it would see $(y_1, \ldots, y_k)$ in order to generate $\hat{y}_{k+1}$.

- Gradient descent. Main articles: Wikipedia/gradient descent & Wikipedia/vanishing gradient problem. Gradient descent is a 1st-order iterative optimization algorithm for finding the minimum of a function. In neural networks, it can be used to minimize the error term by changing each weight in proportion to the derivative of the error w.r.t. that weight, provided the nonlinear activation functions are differentiable.

  The standard method for training RNN by gradient descent is the "backpropagation through time" (BPTT) algorithm, which is a special case of the general algorithm of backpropagation. A more computationally expensive online variant is called "Real-Time Recurrent Learning" or RTRL, which is an instance of automatic differentiation in the forward accumulation mode with stacked tangent vectors. Unlike BPTT, this algorithm is local in time but not local in space.

  In this context, local in space means that a unit's weight vector can be updated using only information stored in the connected units & the unit itself s.t. update complexity of a single unit is linear in the dimensionality of the weight vector. Local in time means that the updates take place continually (on-line) & depend only on the most recent time step rather than on multiple time steps within a given time horizon as in BPTT. Biological neural networks appear to be local w.r.t. both time & space.

  For recursively computing the partial derivatives, RTRL has a time-complexity of O(number of hidden · number of weights) per time step for computing the Jacobian matrices, while BPTT only takes O(number of weights) per time step, at the cost of storing all forward activations within the given time horizon. An online hybrid between BPTT & RTRL with intermediate complexity exists, along with variants for continuous time.

  A major problem with gradient descent for standard RNN architectures is that error gradients vanish exponentially quickly with the size of the time lag between important events. LSTM combined with a BPTT/RTRL hybrid learning method attempts to overcome these problems. This problem is also solved in the independently recurrent neural network (IndRNN) by reducing

the context of a neuron to its own past state & the cross-neuron information can then be explored in the following layers. Memories of different ranges including long-term memory can be learned without the gradient vanishing & exploding problem.

The on-line algorithm called causal recursive backpropagation (CRBP), implements & combines BPTT & RTRL paradigms for locally recurrent networks. It works with the most general locally recurrent networks. The CRBP algorithm can minimize the global error term. This fact improves the stability of the algorithm, providing a unifying view of gradient calculation techniques for recurrent networks with local feedback.

1 approach to gradient information computation in RNNs with arbitrary architectures is based on signal-flow graphs diagrammatic derivation. It uses the BPTT batch algorithm, based on Lee's theorem for network sensitivity calculations. It was proposed by Wan & Beaufays, while its fast online version was proposed by Campolucci, Uncini, & Piazza.

- Connectionist temporal classification. The connectionist temporal classification (CTC) is a specialized loss function for training RNNs for sequence modeling problems where the timing is variable.

- Global optimization methods. Training the weights in a neural network can be modeled as a nonlinear global optimization problem. A target function can be formed to evaluate the fitness or error of a particular weight vector as follows: 1st, the weights in the network are set according to the weight vector. Next, the network is evaluated against the training sequence. Typically, the sum-squared difference between the predictions & the target values specified in the training sequence is used to represent the error of the current weight vector. Arbitrary global optimization techniques may then be used to minimize this target function.

  The most common global optimization method for training RNNs is genetic algorithms, especially in unstructured networks.

  Initially, the genetic algorithm is encoded with the neural network weights in a predefined manner where 1 gene in the chromosome represents 1 weight link. The whole network is represented as a single chromosome. The fitness function is evaluated as follows:

  - Each weight encoded in the chromosome is assigned to the respective weight link of the network.
  - The training set is presented to the network which propagates the input signals forward.
  - The mean-squared error is returned to the fitness function.
  - This function drives the genetic selection process.

  Many chromosomes make up the population; therefore, many different neural networks are evolved until a stopping criterion is satisfied. A common stopping scheme is:

  - When the neural network has learned a certain percentage of the training data or
  - When the minimum value of the mean-squared-error is satisfied or
  - When the maximum number of training generations has been reached.

  The fitness function evaluates the stopping criterion as it receives the mean-squared error reciprocal from each network during training. Therefore, the goal of the genetic algorithm is to maximize the fitness function, reducing the mean-squared error.

  Other global (&/or evolutionary) optimization techniques may be used to seek a good set of weights, e.g. simulated annealing or particle swarm optimization.

8.2.5 Other architectures

- Independently RNN (IndRNN). The independently recurrent neural network (IndRNN) addresses the gradient vanishing & exploding problems in the traditional fully connected RNN. Each neuron in 1 layer only receives its own past state as context information (instead of full connectivity to all other neurons in this layer) & thus neurons are independent of each other's history. The gradient backpropagation can be regulated to avoid gradient vanishing & exploding in order to keep long or short-term memory. The cross-neuron information is explored in the next layers. IndRNN can be robustly trained with non-saturated nonlinear functions e.g. ReLU. Deep networks can be trained using skip connections.

- Neural history compressor. The neural history compressor is an unsupervised stack of RNNs. At the input level, it learns to predict its next input from the previous inputs. Only unpredictable inputs of some RNN in the hierarchy become inputs to the next higher level RNN, which therefore recomputes its internal state only rarely. Each higher level RNN thus studies a compressed representation of the information in the RNN below. This is done s.t. the input sequence can be precisely reconstructed from the representation at the highest level.

  The system effectively minimizes the description length or the negative logarithm of the probability of the data. Given a lot of learnable predictability in the incoming data sequence, the highest level RNN can use supervised learning to easily classify even deep sequences with long intervals between important events.

  It is possible to distill the RNN hierarchy into 2 RNNs: the "conscious" chunker (higher level) & the "subconscious" automatizer (lower level). Once the chunker has learned to predict & compress inputs that are unpredictable by the automatizer, then the automatizer can be forced in the next learning phase to predict or imitate through additional unit the hidden units of the more slowly changing chunker. This makes it easy for the automatizer to learn appropriate, rarely changing memories across

long intervals. In turn, this helps the automatizer to make many of its once unpredictable inputs predictable, s.t. the chunker can focus on the remaining unpredictable events.

A generative model partially overcame the vanishing gradient problem of automatic differentiation or backpropagation in neural networks in 1992. In 1993, such a system solved a "Very Deep Learning" task that required > 1000 subsequent layers in an RNN unfolded in time.

- 2nd order RNNs. 2nd-order RNNs use higher order weights w_{ijk} instead of the standard w_{ij} weights, & states can be a product. This allows a direct mapping to a finite-state machine both in training, stability, & representation. Long short-term memory is an example of this but has no such formal mappings or proof of stability.

- Hierarchical recurrent neural network. Hierarchical recurrent neural networks (HRNN) connect their neurons in various ways to decompose hierarchical behavior into useful subprograms. Such hierarchical structures of cognition are present in theories of memory presented by philosopher Henri Bergson, whose philosophical views have inspired hierarchical models.

  Hierarchical recurrent neural networks are useful in forecasting, helping to predict disaggregated inflation components of the consumer price index (CPI). The HRNN model leverages information from higher levels in the CPI hierarchy to enhance lower-level predictions. Evaluation of a substantial dataset from the US CPI-U index demonstrates the superior performance of the HRNN model compared to various established inflation prediction methods.

- Recurrent multiplayer perceptron network. Generally, a recurrent multiplayer perceptron network (RMLP network) consists of cascaded subnetworks, each containing multiple layers of nodes. Each subnetwork is feed-forward except for the last layer, which can have feedback connections. Each of these subnets is connected only by feed-forward connections.

- Multiple timescales model. A multiple timescales recurrent neural network (MTRNN) is a neural-based computational model that can simulate the functional hierarchy of the brain through self-organization depending on the spatial connection between neurons & on distinct types of neuron activities, each with distinct time properties. With such varied neuronal activities, continuous sequences of any set of behaviors are segmented into reusable primitives, which in turn are flexibly integrated into diverse sequential behaviors. The biological approval of such a type of hierarchy was discussed in the memory-prediction theory of brain function by Hawkins in his book *On Intelligence*. Such a hierarchy also agrees with theories of memory posited by philosopher Henri Bergson, which have been incorporated into an MTRNN model.

- Memristive networks. Greg Snider of HP Labs describes a system of cortical computing with memristive nanodevices. The memristors (memory resistors) are implemented by thin film materials in which the resistance is electrically tuned via the transport of ions or oxygen vacancies within the film. DARPA's SyNAPSE project has funded IBM Research & HP Labs, in collaboration with the Boston University Department of Cognitive & Neural Systems (CNS), to develop neuromorphic architectures that may be based on memristive systems. Memristive networks are a particular type of physical neural network that have very similar properties to (Little-)Hopefield networks, as they have continuous dynamics, a limited memory capacity & natural relaxation via the minimization of a function which is asymptotic to the Ising model. In this sense, the dynamics of a memristive circuit have the advantage compared to a Resistor-Capacitor network to have a more interesting nonlinear behavior. From this point of view, engineering analog memristive networks account for a peculiar type of neuromorphic engineering in which the device behavior depends on the circuit wiring or topology. The evolution of these networks can be studied analytically using variations of the Caravelli-Traversa-Di Ventra equations.

- Continuous-time. A continuous-time recurrent neural network (CTRNN) uses a system of ODEs to model the effects on a neuron of the incoming inputs. They are typically analyzed by dynamical systems theory. Many RNN models in neuroscience are continuous-time.

  For a neuron i in the network with activation y_i, the rate of change of activation is given by $\tau_i \dot{y}_i = -y_i + \sum_{j=1}^{n} w_{ji}\sigma(y_j - \Theta_j) + I_i(t)$ where τ_i: time constant of postsynaptic node, y_i: activation of postsynaptic node, $\dot{y}_i$: rate of change of activation of postsynaptic node, w_{ji}: weight of connection from pre to postsynaptic node, $\sigma(x)$: sigmoid of x e.g. $\sigma(x) = \frac{1}{1+e^{-x}}$, y_j: activation of presynaptic node, Φ_j: bias of presynaptic node, $I_i(t)$: input (if any) to node. CTRNNs have been applied to evolutionary robotics where they have been used to address vision, co-operation, & minimal cognitive behavior.

  Note that, by the Shannon sampling theorem, discrete-time recurrent neural networks can be viewed as continuous-time recurrent neural networks where the differential equations have transformed into equivalent difference equations. This transformation can be thought of as occurring after the post-synaptic node activation functions $y_i(t)$ have been low-pass filtered but prior to sampling.

  They are in fact recursive neural networks with a particular structure: that of a linear chain. Whereas recursive neural networks operate on any hierarchical structure, combining child representations into parent representations, recurrent neural networks operate on the linear progression of time, combining the previous time step & a hidden representation into the representation for the current time step.

  From a time-series perspective, RNNs can appear as nonlinear versions of finite impulse respons & infinite impulse response filters & also as a nonlinear autoregressive exogenous model (NARX). RNN has infinite impulse response whereas convolutional neural networks have finite impulse response. Both classes of networks exhibit temporal dynamic behavior. A finite impulse recurrent network is a directed acyclic graph that can be unrolled & replaced with a strictly feedforward neural network, while an infinite impulse recurrent network is a directed cyclic graph that cannot be unrolled.

The effect of memory-based learning for the recognition of sequences can also be implemented by a more biological-based model which uses the silencing mechanism exhibited in neurons with a relatively high frequency spiking activity.

Additional stored states & the storage under direct control by the network can be added to both infinite-impulse & finite-impulse networks. Another network or graph can also replace the storage if that incorporates time delays or has feedback loops. Such controlled states are referred to as gated states or gated memory & are part of long short-term memory networks (LSTMs) & gated recurrent units. This is also called Feedback Neural Network (FNN).

8.2.6 Libraries

Modern libraries provide runtime-optimized implementations of the above functionality or allow to speed up the slow loop by just-in-time compilation.

- Apache Singa
- Caffe: Created by the Berkeley Vision & Learning Center (BVLC). It supports both CPU & GPU. Developed in C++, & has Python & MATLAB wrappers.
- Chainer: Fully in Python, production support fopr CPU, GPU, distributed training.
- Deeplearning4j: Deep learning in Java & Scala on multi-GPU-enabled Spark.
- Flux: includes interfaces for RNNs, including GRUs & LSTMs, written in Julia.
- Keras: High-level API, providing a wrapper to many other deep learning libraries.
- Microsoft Cognitive Toolkit
- MXNet: an open-source deep learning framework used to train & deploy deep neural networks.
- PyTorch: Tensors & Dynamic neural networks in Python with GPT acceleration.
- TensorFlow: Apache 2.0-licensed Theano-like library with support for CPU, GPU & Google's properietary TPU, mobile
- Theano: A deep-learning library for Python with an API largely compatible with the NumPy library.
- Torch: A scientific computing framework with support for machine learning algorithms, written in C & Lua.

8.2.7 Applications

Applications of recurrent neural networks include:

- Machine translation
- Robot control
- Time series prediction
- Speech recognition
- Speech synthesis
- Brain-computer interfaces
- Time series anomaly detection
- Text-to-Video model
- Rhythm learning
- Music composition
- Grammar learning
- Handwriting recognition
- Human action recognition
- Protein homology detection
- Predicting subcellular localization of proteins
- Several prediction tasks in the area of business process management
- Prediction in medical care pathways
- Predictions of fusion plasma disruptions in reactors (Fusion Recurrent Neural Network (FRNN) code)– Wikipedia/recurrent neural network

8.3 Wikipedia/torch (ML)

"`Torch` is an open-source ML library, a scientific computing framework, & a scripting language based on Lua. It provides LuaJIT interfaces to DL algorithms implemented in C. It was created by Idiap Research Institute at EPFL. Torch development moved in 2017 to PyTorch, a port of library to Python.

8.3.1 torch

Core package of Torch is `torch`. It provides a flexible N-dimensional array or Tensor, which supports basic routines for indexing, slicing, transposing, type-casting, resizing, sharing storage & cloning. This object is used by most other packages & thus forms core object of library. Tensor also supports mathematical operations like `max, min, sum`, statistical distributions like uniform, normal, & multinomial, & BLAS (Basic Linear Algebra Subprograms) operations like dot product, matrix-vector multiplication, matrix-matrix multiplication & matrix product.

Following exemplifies using torch via its REPL interpreter:

```
> a = torch.randn(3, 4)

> =a
-0.2381 -0.3401 -1.7844 -0.2615
0.1411  1.6249  0.1708  0.8299
-1.0434  2.2291  1.0525  0.8465
[torch.DoubleTensor of dimension 3x4]

> a[1][2]
-0.34010116549482

> a:narrow(1,1,2)
-0.2381 -0.3401 -1.7844 -0.2615
0.1411  1.6249  0.1708  0.8299
[torch.DoubleTensor of dimension 2x4]

> a:index(1, torch.LongTensor{1,2})
-0.2381 -0.3401 -1.7844 -0.2615
0.1411  1.6249  0.1708  0.8299
[torch.DoubleTensor of dimension 2x4]

> a:min()
-1.7844365427828
```

`torch` package also simplifies object-oriented programming & serialization by providing various convenience functions which are used throughout its packages. `torch.class(classname, parentclass)` function can be used to create object factories (classes). When constructor is called, `torch` initializes & sets a Lua table with user-defined metatable, which makes table an object.

Objects created with torch factory can also be serialized, as long as they do not contain references to objects that cannot be serialized, e.g. Lua coroutines, & Lua *userdata*. However, *userdata* can be serialized if it is wrapped by a table (or metatable) that provides `read(), write()` methods.

8.3.2 nn

`nn` package is used for building neural networks. It is divided into modular objects that share a common `Module` interface. Modules have a `forward(), backward()` method that allow them to feedforward & backpropagation, resp. Modules can be joined using module composites, like `Sequential, Parallel, Concat` to create complex task-tailored graphs. Simpler modules like `Linear, Tanh, Max` make up basic component modules. This modular interface provides 1st-order automatic gradient differentiation. What follows is an example use-case for building a multilayer perceptron using Modules:

```
> mlp = nn.Sequential()
> mlp:add(nn.Linear(10, 25)) -- 10 input, 25 hidden units
> mlp:add(nn.Tanh()) -- some hyperbolic tangent transfer function
> mlp:add(nn.Linear(25, 1)) -- 1 output
> =mlp:forward(torch.randn(10))
-0.1815
[torch.Tensor of dimension 1]
```

Loss functions are implemented as sub-classes of `Criterion`, which has a similar interface to `Module`. It also has `forward(), backward()` methods for computing loss & backpropagating gradients, resp. Criteria are helpful to train neural network on classical tasks. Common criteria are mean squared error criterion implemented in *MSECriterion* & cross-entropy criterion implemented in `ClassNNLCriterion`. What follows is an example of a Lua function that can be iteratively called to train an `mlp` Module on input Tensor `x`, target Tensor `y` with a scalar `learningRate`:

```
function gradUpdate(mlp, x, y, learningRate)
  local criterion = nn.ClassNLLCriterion()
  local pred = mlp:forward(x)
  local err = criterion:forward(pred, y);
  mlp:zeroGradParameters();
  local t = criterion:backward(pred, y);
  mlp:backward(x, t);
  mlp:updateParameters(learningRate);
end
```

It also has `StochasticGradient` class for training a neural network using stochastic gradient descent, although `optim` package provides much more options in this respect, like momentum & weight decay regularization.

8.3.3 Other packages

Many packages other than above official packages are used with Torch. These are listed in torch cheatsheet. These extra packages provide a wide range of utilities e.g. parallelism, asynchronous input/output, image processing, & so on. They can be installed with LuaRocks, Lua package manager which is also included with Torch distribution.

8.3.4 Applications

Torch is used by Facebook AI Research Group, IBM, Yandex, & Idiap Research Institute. Torch has been extended for use on Android & iOS. It has been used to build hardware implementations for data flows like those found in neural networks.

Facebook has released a set of extension modules as open source software." – Wikipedia/torch (ML)

8.4 Wikipedia/types of artificial neural networks

There are many *types of artificial neural networks (ANN)*.

Artificial neural networks are computational models inspired by biological neural networks, & are used to approximate functions that are generally unknown. Particularly, they are inspired by behavior of neurons & electrical signals they convey between input (e.g. from eyes or nerve endings in hand), processing, & output from brain (e.g. reacting to light, touch, or heat). The way neurons semantically communicate is an area of ongoing research. Most artificial neural networks bear only some resemblance to their more complex biological counterparts, but are very effective at their intended tasks (e.g. classification or segmentation).

Some artificial neural networks are adaptive systems & are used e.g. to model population & environments, which constantly change.

Neural networks can be hardware- (neurons are represented by physical components) or software-based (computer models), & can use a variety of topologies & learning algorithms.

8.4.1 Feedforward

Main article: Wikipedia/feedforward neural network. Feedforward neural network: 1st & simplest type. In this network information moves only from input layer directly through any hidden layers to output layer without cycles/loops. Feedforward networks can be constructed with various types of units, e.g. binary McCulloch–Pitts neurons, simplest of which is perceptron. Continuous neurons, frequently with sigmoidal activation, are used in context of backpropagation.

- Group method of data handling. Main article: Wikipedia/group method of data handling. Group Method of Data Handling (GMDH) features fully automatic structural & parametric model optimization. Node activation functions are Kolmogorov–Gabor polynomials that permit additions & multiplications. It uses a deep multilayer perceptron with 8 layers. It is a supervised learning network that grows layer by layer, where each layer is trained by regression analysis. Useless items are detected using a validation set, & pruned through regularization. Size & depth of resulting network depends on task.

- Autoencoder. Main article: Wikipedia/autoencoder. An autoencoder, autoassociator or Diabolo network: is similar to multilayer perceptron (MLP) – with an input layer, an output layer & 1 or more hidden layers connecting them. However, output layer has same number of units as input layer. Its purpose is to reconstruct its own inputs (instead of emitting a target value. Therefore, autoencoders are unsupervised learning models. An autoencoder is used for unsupervised learning of efficient coding, typically for purpose of dimensionality reduction & for learning generative models of data.

- Probabilistic. Main article: Wikipedia/probabilistic neural network. A probabilistic neural network (PNN) is a 4-layer feedforward neural network. Layers are Input, hidden pattern/summation, & output. In PNN algorithm, parent probability distribution function (PDF) of each class is approximated by a Parzen window & a non-parametric function. Then, using PDF of each class, class probability of a new input is estimated & Bayes' rule is employed to allocate it to class with highest posterior probability. It was derived from Bayesian network & a statistical algorithm called Kernel Fisher discriminant analysis. It is used for classification & pattern recognition.

- Time delay. Main article: Wikipedia/time delay neural network. A time delay neural network (TDNN) is a feedforward architecture for sequential data that recognizes features independent of sequence position. In order to achieve time-shift invariance, delays are added to input so that multiple data points (points in time) are analyzed together.

  It usually forms part of a larger pattern recognition system. It has been implemented using a perceptron network whose connection weights were trained with back propagation (supervised learning).

- Convolutional. Main article: Wikipedia/convolutional neural network. A convolutional neural network (CNN, or ConvNet or shift invariant or space invariant) is a class of deep network, composed of 1 or more convolutional layers with fully connected layers (matching those in typical ANNs) on top. It uses tied weights & pooling layers. In particular, max-pooling. It is often structured via Fukushima's convolutional architecture. They are variations of multilayer perceptrons that use minimal preprocessing. This architecture allows CNNs to take advantage of 2d structure of input data.

  Its unit connectivity pattern is inspired by organization of visual cortex. Units respond to stimuli in a restricted region of space known as receptive field. Receptive fields partially overlap, overcovering entire visual field. Unit response can be approximated mathematically by a convolution operation.

  CNNs are suitable for processing visual & other 2D data. They have shown superior results in both image & speech applications. They can be trained with standard backpropagation. CNNs are easier to train than other regular, deep, feed-forward neural networks & have many fewer parameters to estimate.

  Capsule Neural Networks (CapsNet) add structures called *capsules* to a CNN & reuse output from several capsules to form more stable (w.r.t. various perturbations) representations.

  Examples of applications in computer vision include DeepDream & robot navigation. They have wide applications in image & video recognition, recommender systems, & natural language processing.

- Deep stacking network. A deep stacking network (DSN) (deep convex network) is based on a hierarchy of blocks of simplified neural network modules. It was introduced in 2011 by Deng & Yu. It formulates learning as a convex optimization problem with a closed-form solution, emphasizing mechanism's similarity to stacked generalization. Each DSN block is a simple module that is easy to train by itself in a supervised fashion without backpropagation for entire blocks.

  Each block consists of a simplified multi-layer perceptron (MLP) with a single hidden layer. Hidden layer $\mathbf{h}$ has logistic sigmoidal units, & output layer has linear units. Connections between these layers are represented by weight matrix U; input-to-hidden-layer connections have weight matrix W. Target vectors $\mathbf{t}$ form columns of matrix T, & input data vectors $\mathbf{x}$ form columns of matrix X. Matrix of hidden units is $H = \sigma(W^\top X)$. Modules are trained in order, so lower-layer weights W are known at each stage. Function performs element-wise logistic sigmoid operation. Each block estimates same final label `class` `y`, & its estimate is concatenated with original input X to form expanded input for next block. Thus, input to 1st block contains original data only, while downstream blocks' input adds output of preceding blocks. Then learning upper-layer weight matrix U given other weights in network can be formulated as a convex optimization problem $\min_{U^\top} f = \|U^\top H - T\|_{\mathrm{F}}^2$, which has a closed-form solution.

  Unlike other deep architectures, e.g. DBNs, goal: not to discover transformed feature representation. Structure of hierarchy of this kind of architecture makes parallel learning straightforward, as a batch-mode optimization problem. In purely discriminative tasks, DSNs outperform conventional DBNs.

  - Tensor deep stacking networks. This architecture is a DSN extension. It offers 2 important improvements: it uses higher-order information from covariance statistics, & it transforms non-convex problem of a lower-layer to a convex sub-problem of an upper-layer. TDSNs use covariance statistics in a bilinear mapping from each of 2 distinct sets of hidden units in same layer to prediction, via a 3rd-order tensor.

    While parallelization & scalability are not considered seriously in conventional DNNs, all learning for DSNs & TDSNs is done in batch mode, to allow parallelization. Parallelization allows scaling design to larger (deeper) architectures & data sets.

    Basic architecture is suitable for diverse tasks e.g. classification & regression.

8.4.2 Regulatory feedback

Regulatory feedback networks started as a model to explain brain phenomena found during recognition including network-wide bursting & difficulty with similarity found universally in sensory recognition. A mechanism to perform optimization during recognition is created using inhibitory feedback connections back to same inputs that activate them. This reduces requirements during learning & allows learning & updating to be easier while still being able to perform complex recognition.

A regulatory feedback network makes inferences using negative feedback. Feedback is used to find optimal activation of units. It is most similar to a non-parametric method but is different from K-nearest neighbor in that it mathematically emulates feedforward networks.

8.4.3 Radial basis function

Main article: Wikipedia/radial basis function network. Radial basis functions are functions that have a distance criterion w.r.t. a center. Radial basis functions have been applied as a replacement for sigmoidal hidden layer transfer characteristic in multi-layer perceptrons. RBF networks have 2 layers: In the 1st, input is mapped onto each RBF in 'hidden' layer. RBF chosen is usually a

Gaussian. In regression problems output layer is a linear combination of hidden layer values representing mean predicted output. This interpretation of this output layer value is the same as a regression model in statistics. In classification problems output layer is typically a sigmoid function of a linear combination of hidden layer values, representing a posterior probability. Performance in both cases is often improved by shrinkage techniques, known as ridge regression in classical statistics. This corresponds to a prior belief in small parameter values (& therefore smooth output functions) in a Bayesian framework.

RBF networks have advantage of avoiding local minima in same way as multi-layer perceptrons. This is because only parameters that are adjusted in learning process are linear mapping from hidden layer to output layer. Linearity ensures: error surface is quadratic & therefore has a single easily found minimum. In regression problems this can be found in 1 matrix operation. In classification problems fixed nonlinearity introduced by sigmoid output function is most efficiently dealt with using iteratively re-weighted least squares.

RBF networks have disadvantage of requiring good coverage of input space by radial basis functions. RBF centers are determined with reference to distribution of input data, but without reference to prediction task. As a result, representational resources may be wasted on areas of input space that are irrelevant to task. A common solution is to associative each data point with its own center, although this can expand linear system to be solved in final layer & requires shrinkage techniques to avoid overfitting.

Associating each input datum with an RBF leads naturally to kernel methods e.g. support vector machines (SVM) & Gaussian processes (RBF is kernel function). All 3 approaches use a nonlinear kernel function to project input data into a space where learning problem can be solved using a linear model. Like Gaussian processes, & unlike SVMs, RBF networks are typically trained in a maximum likelihood framework by maximizing probability (minimizing error). SVMs avoid overfitting by maximizing instead a margin. SVMs outperform RBF networks in most classification applications. In regression applications they can be competitive when dimensionality of input space is relatively small.

- How RBF networks work. RBF neural networks are conceptually similar to K-nearest neighbor (k-NN) models. Basic idea: similar inputs produce similar outputs.

  Assume: each case in a training set has 2 predictor variables x, y, & target variable has 2 categories, positive & negative. Given a new case with predictor values $x = 6, y = 5.1$, how is target variable computed?

  Nearest neighbor classification performed for this example depends on how many neighboring points are considered. If 1-NN is used & closest point is negative, then new point should be classified as negative. Alternatively, if 9-NN classification is used & closest 9 points are considered, then effect of surrounding 8 positive points may outweigh closest 9th (negative) point.

  An RBF network positions neurons in space described by predictor variables (x, y in this example). This space has as many dimensions as predictor variables. Euclidean distance is computed from new point to center of each neuron, & a radial basis function (RBF, also called a *kernel function*) is applied to distance to compute weight (influence) for each neuron. Radial basis function is so named because radius distance is argument to function. Weight = RBF(distance).

  - Radial basis function. Value for new point is found by summing output values of RBF functions multiplied by weights computed for each neuron.

    Radial basis function for a neuron has a center & a radius (also called a *spread*). Radius may be different for each neuron, &, in RBF networks generated by DTREG, radius may be different in each dimension.

    With larger spread, neurons at a distance from a point have a greater influence.

  - Architecture. RBF networks have 3 layers:

    * **Input layer.** 1 neuron appears in input layer for each predictor variable. In case of categorical variables, $N - 1$ neurons are used where N: number of categories. Input neurons standardizes value ranges by subtracting median & dividing by interquartile range. Input neurons then feed values to each of neurons in hidden layer.
    * **Hidden layer.** This layer has a variable number of neurons (determined by training process). Each neuron consists of a radial basis function centered on a point with as many dimensions as predictor variables. Spread (radius) of RBF function may be different for each dimension. Centers & spreads are determined by training. When presented with x vector of input values from input layer, a hidden neuron computes Euclidean distance of test case from neuron's center point & then applies RBF kernel function to this distance using spread values. Resulting value is passed to summation layer.
    * **Summation layer.** Value coming out of a neuron in hidden layer is multiplied by a weight associated with neuron & adds to weighted values of other neurons. This sum becomes output. For classification problems, 1 output is produced (with a separate set of weights & summation unit) for each target category. Value output for a category is probability that the case being evaluated has that category.

  - Training. Following parameters are determined by training process:

    * Number of neurons in hidden layer
    * Coordinates of center of each hidden-layer RBF function
    * Radius (spread) of each RBF function in each dimension
    * Weights applied to RBF function outputs as they pass to summation layer

    Various methods have been used to train RBF networks. 1 approach 1st uses K-means clustering to find cluster centers which are then used as centers for RBF functions. However, K-means clustering is computationally intensive & it often does not generate optimal number of centers. Another approach is to use a random subset of training points as centers.

DTREG uses a training algorithm that uses an evolutionary approach to determine optimal center points & spreads for each neuron. It determines when to stop adding neurons to network by monitoring estimated leave-1-out (LOO) error & terminating when LOO error begins to increase because of overfitting.

Computation of optimal weights between neurons in hidden layer & summation layer is done using ridge regression. An iterative procedure computes optimal regularization Lambda parameter that minimizes generalized cross-validation (GCV) error.

- General regression neural network. Main article: Wikipedia/General regression neural network. A GRNN is an associative memory neural network that is similar to similar to probabilistic neural network but it is used for regression & approximation rather than classification.

8.4.4 Deep belief network

Main article: Wikipedia/deep belief network. A restricted Boltzmann machine (RBM) with fully connected visible & hidden units. Note there are no hidden-hidden or visible-visible connections. A deep belief network (DBN) is a probabilistic, generative model made up of multiple hidden layers. It can be considered a composition of simple learning modules.

A DBN can be used to generatively pre-train a deep neural network (DNN) by using learned DBN weights as initial DNN weights. Various discriminative algorithms can then tune these weights. This is particularly helpful when training data are limited, because poorly initialized weights can significantly hinder learning. These pre-trained weights end up in a region of weight space that is closer to optimal weights than random choices. This allows for both improved modeling & faster ultimate convergence.

8.4.5 Recurrent neural network

Main article: Wikipedia/recurrent neural network. Recurrent neural networks (RNN) propagate data forward, but also backwards, from later processing stages to earlier stages. RNN can be used as general sequence processors.

- Fully recurrent. This architecture was developed in 1980s. Its network creates a directed connection between every pair of units. Each has a time-varying, real-valued (more than just 0 or 1) activation (output). Each connection has a modifiable real-valued weight. Some of nodes are called *labeled nodes*, some output nodes, the rest hidden nodes.

  For supervised learning in discrete time settings, training sequences of real-valued input vectors become sequences of activations input nodes, 1 input vector at a time. At each time step, each non-input unit computes its current activation as a nonlinear function of weighted sum of activations of all units from which it receives connections. System can explicitly activate (independent of incoming signals) some output units at certain time steps. E.g., if input sequence is a speech signal corresponding to a spoken digit, final target output at end of sequence may be a label classifying digit. For each sequence, its error is sum of deviations of all activations computed by network from corresponding target signals. For a training set of numerous sequences, total error is sum of errors of all individual sequences.

  To minimize total error, gradient descent can be used to change each weight in proportion to its derivative w.r.t. error, provided nonlinear activation functions are differentiable. Standard method is called "backpropagation through time" or BPTT, a generalization of backpropagation for feedforward networks. A more computationally expensive online variant is called "Real-Time Recurrent Learning" or RTRL. Unlike BPTT this algorithm is *local in time but not local in space*. An online hybrid between BPTT & RTRL with intermediate complexity exists, with variants for continuous time. A major problem with gradient descent for standard RNN architectures: error gradients vanish exponentially quickly with size of time lag between important events. Long short-term memory architecture overcomes these problems.

  In reinforcement learning settings, no teacher provides target signals. Instead a fitness function or reward function or utility function is occasionally used to evaluate performance, which influences its input stream through output units connected to actuators that affect environment. Variants of evolutionary computation are often used to optimize weight matrix.

  - Hopfield. Hopfield network (like similar attractor-based networks) is of historic interest although it is not a general RNN, as it is not designed to process sequences of patterns. Instead it requires stationary inputs. It is an RNN in which all connections are symmetric. It guarantees that it will converge. If connections are trained using Hebbian learning Hopfield network can perform as robust content-addressable memory, resistant to connection alteration.
  - Boltzmann machine. Boltzmann machine can be thought of as a noisy Hopfield network. It is 1 of 1st neural networks to demonstrate learning of latent variables (hidden units). Boltzmann ML was at 1st slow to simulate, but contrastive divergence algorithm speeds up training for Boltzmann machines & Products of Experts.
  - Self-organizing map. Self-organizing map (SOM) uses unsupervised learning. A set of neurons learn to map points in an input space to coordinates in an output space. Input space can have different dimensions & topology from output space, & SOM attempts to preserve these.
  - Learning vector quantization. Learning vector quantization (LVQ) can be interpreted as a neural network architecture. Prototypical representatives of classes parametrize, together with an appropriate distance measure, in a distance-based classification scheme.

- Simple recurrent. Simple recurrent networks have 3 layers, with addition of a set of "context units" in input layer. These units connect from hidden layer or output layer with a fixed weight of 1. At each time step, input is propagated in a standard feedforward fashion, & then a backpropagation-like learning rule is applied (not performing gradient descent). Fixed back connections leave a copy of previous values of hidden units in context units (since they propagate over connections before learning rule is applied).
- Reservoir computing. Reservoir computing is a computation framework that may be viewed as an extension of neural networks. Typically an input signal is fed into a fixed (random) dynamical system called a *reservoir* whose dynamics map input to a higher dimension. A *readout* mechanism is trained to map reservoir to desired output. Training is performed only at readout stage. Liquid-state machine are a type of reservoir computing.
  - Echo state. Echo state network (ESN) employs a sparsely connected random hidden layer. Weights of output neurons are only part of network that are trained. ESN are good at reproducing certain time series.
- Long short-term memory. Long short-term memory (LSTM) avoids vanishing gradient problem. It works even when with long delays between inputs & can handle signals that mix low & high frequency components. LSTM RNN outperformed other RNN & other sequence learning methods e.g. HMM in applications e.g. language learning & connected handwriting recognition.
- Bi-directional. Main article: Wikipedia/bidirectional recurrent neural network. Bi-directional RNN, or BRNN, use a finite sequence to predict or label each element of a sequence based on both past & future context of element. This is done by adding outputs of 2 RNNs: one processing sequence from left to right, the other one from right to left. Combined outputs are predictions of teacher-given target signals. This technique proved to be especially useful when combined with LSTM.
- Hierarchical. Hierarchical RNN connects elements in various ways to decompose hierarchical behavior into useful subprograms.
- Stochastic. A district from conventional neural networks, stochastic artificial neural network used as an approximation to random functions.
- Genetic scale. A RNN (often a LSTM) where a series is decomposed into a number of scales where every scale informs primary length between 2 consecutive points. A 1st order scale consists of a normal RNN, a 2nd order consists of all points separated by 2 indices & so on. The Nth order RNN connects 1st & last node. Outputs from all various scales are treated as a Committee of Machines & associated scores are used genetically for next iteration.

8.4.6 Modular

- Committee of machines.
- Associative.

8.4.7 Physical

A physical neural network includes electrically adjustable resistance material to simulate artificial synapses. Examples include ADALINE memristor-based neural network. An optical neural network is a physical implementation of an artificial neural network with optical components.

8.4.8 Dynamic

Unlike static neural networks, dynamic neural networks adapt their structure &/or parameters to input during inference showing time-dependent behavior, e.g. transient phenomena & delay effects. Dynamic neural networks in which parameters may change over time are related to fast weights architecture (1987), where 1 neural network outputs weights of another neural network.

- Cascading. Cascade correlation is an architecture & supervised learning algorithm. Instead of just adjusting weights in a network of fixed topology, Cascade-Correlation begins with a minimal network, then automatically trains & adds new hidden units 1 by 1, creating a multilayer structure. Once a new hidden unit has been added to network, its input-side weights are frozen. This unit then becomes a permanent feature-detector in network, available for producing outputs or for creating other, more complex feature detectors. Cascade-Correlation architecture has several advantages: It learns quickly, determines its own size & topology, retains structures it has built even if training set changes & requires no backpropagation.
- Neuro-fuzzy. A neuro-fuzzy network is a fuzzy inference system in body of an artificial neural network. Depending on FIS type, several layers simulate processes involved in a fuzzy inference-like fuzzification, inference, aggregation, & defuzzification. Embedding an FIS in a general structure of an ANN has benefit of using available ANN training methods to find parameters of a fuzzy system.
- Compositional pattern-producing. Compositional pattern-producing networks (CPPNs) are a variation of artificial neural networks which differ in their set of activation functions & how they are applied. While typical artificial neural networks often contain only sigmoid functions (& sometimes Gaussian functions), CPPNs can include both types of functions & many others. Furthermore, unlike typical artificial neural networks, CPPNs are applied across entire space of possible inputs so that they can represent a complete image. Since they are compositions of functions, CPPNs in effect encode images at infinite resolution & can be sampled for a particular display at whatever resolution is optimal.

8.4.9 Memory networks

Memory networks incorporate long-term memory. Long-term memory can be read & written to, with goal of using it for prediction. These models have been applied in context of question answering (QA) where long-term memory effectively acts as a (dynamic) knowledge base & output is a textual response.

In sparse distributed memory or hierarchical temporal memory, patterns encoded by neural networks are used as addresses for content-addressable memory, with "neurons" essentially serving as address encoders & decoders. However, early controllers of such memories were not differentiable.

- 1-shot associative memory. This type of network can add new patterns without re-training. It is done by creating a specific memory structure, which assigns each new pattern to an orthogonal plane using adjacently connected hierarchical arrays. Network offers real-time pattern recognition & high scalability; this requires parallel processing & is thus best suited for platforms e.g. wireless sensor networks, grid computing, & GPGPUs.

- Hierarchical temporal memory. Hierarchical temporal memory (HTM) models some of structural & algorithmic properties of neocortex. HTM is a biomimetic model based on memory-prediction theory. HTM is a method for discovering & inferring high-level causes of observed input patterns & sequences, thus building an increasingly complex model of world.

  HTM combines existing ideas to mimic neocortex with a simple design that provides many capabilities. HTM combines & extends approaches used in Bayesian networks, spatial & temporal clustering algorithms, while using a tree-shaped hierarchy of nodes that is common in neural networks.

- Holographic associative memory. Holographic Associative Memory (HAM) is an analog, correlation-based, associative, stimulus-response system. Information is mapped onto phase orientation of complex numbers. Memory is effective for associative memory tasks, generalization & pattern recognition with changeable attention. Dynamic search localization is central to biological memory. In visual perception, humans focus on specific objects in a pattern. Humans can change focus from object to object without learning. HAM can mimic this ability by creating explicit representations for focus. It uses bi-modal representation of pattern & a hologram-like complex spherical weight state-space. HAMs are useful for optical realization because underlying hyper-spherical computations can be implemented with optical computation.

- LSTM-related differentiable memory structures. Apart from long short-term memory (LSTM), other approaches also added differentiable memory to recurrent functions. E.g.:

  - Differentiable push & pop actions for alternative memory networks called *neural stack machines*
  - Memory networks where control network's external differentiable storage is in fast weights of another network
  - LSTM forget gates
  - Self-referential RNNs with special output units for addressing & rapidly manipulating RNN's own weights in differentiable fashion (internal storage)
  - Learning to transduce with unbounded memory

- Neural Turing machines. Neural Turing machines (NTM) couple LSTM networks to external memory resources, with which they can interact by attentional processes. Combined system is analogous to a Turing machine but is differentiable end-to-end, allowing it to be efficiently trained by gradient descent. Preliminary results demonstrate: neural Turing machines can infer simple algorithms e.g. copying, sorting & associative recall from input & output examples.

  Differential neural computers (DNC) are an NTM extension. They out-performed Neural turing machines, long short-term memory systems & memory networks on sequence-processing tasks.

- Semantic hashing. Approaches that represent previous experiences directly & use a similar experience to form a local model are often called nearest neighbor or k-nearest neigbors methods. Deep learning is useful in semantic hashing where a deep graphical model word-count vectors obtained from a large set of documents. Documents are mapped to memory addresses in such a way that semantically similar documents are located at nearby addresses. Documents similar to a query document can then be found by accessing all addresses that differ by only a few bits from address of query document. Unlike sparse distributed memory that operates on 1000-bit addresses, semantic hashing works on 32 or 64-bit addresses found in a conventional computer architecture.

- Pointer networks. Deep neural networks can be potentially improved by deepening & parameter reduction, while maintaining trainability. While training extremely deep (e.g., 1 million layers) neural networks might not be practical, CPU-like architectures e.g. pointer networks & neural random-access machines overcome this limitation by using external random-access memory & other components that typically belong to a computer architecture e.g. registers, ALU, & pointers. Such systems operate on probability distribution vectors stored in memory cells & registers. Thus, model is fully differentiable & trains end-to-end. Key characteristic of these models: their depth, size of their short-term memory, & number of parameters can be altered independently.

8.4.10 Hybrids

- Encoder-decoder networks. Encoder-decoder frameworks are based on neural networks that map highly structured input to highly structured output. Approach arose in context of machine translation, where input & output are written sentences in 2 natural languages. In that work, an LSTM RNN or CNN was used as an encoder to summarize a source sentence, & summary was decoded using a conditional RNN language model to produce translation. These systems share building blocks: gated RNNs & CNNs & trained attention mechanisms.

8.4.11 Other types

- Instantaneously trained. Instataneously trained neural networks (ITNN) were inspired by phenomenon of short-term learning that seems to occur instantaneously. In these networks weights of hidden & output layers are mapped directly from training vector data. Ordinarily, they work on binary data, but versions for continuous data that require small additional processing exist.

- Spiking. Spiking neural networks (SNN) explicitly consider timing of inputs. Network input & output are usually represented as a series of spikes (delta function or more complex shapes). SNN can process information in time domain (signals that vary over time). They are often implemented as recurrent networks. SNN are also a form of pulse computer.

  Spiking neural networks with axonal conduction delays exhibit polychronization, & hence could have a very large memory capacity.

  SNN & temporal correlations of neural assemblies in such networks – have been used to model figure/ground separation & region linking in visual system.

- Spatial. Spatial neural networks (SNNs) constitute a supercategory of tailored neural networks (NNs) for representing & predicting geographic phenomena. They generally improve both statistical accuracy & reliability of a-spatial/classic NNs whenever they handle geo-spatial datasets, & also of the other spatial (statistical) models (e.g. spatial regression models) whenever geo-spatial Datasets; variables depict nonlinear relations. Examples of SNNs are OSFA spatial neural networks, SVANNs & GWNNs.

- Neocognitron. Neocognitron is a hierarchical, multilayered network that was modeled after visual cortex. It uses multiple types of units, (originally 2, called simple & complex cells), as a cascading model for use in pattern recognition tasks. Local features are extracted by S-cells whose deformation is tolerated by C-cells. Local features in input are integrated gradually & classified at higher layers. Among various kinds of neocognitron are systems that can detect multiple patterns in same input by using back propagation to achieve selective attention. It has been used for pattern recognition tasks & inspired convolutional neural networks.

- Compound hierarchical-deep models. Compound hierarchical-deep models compose deep networks with non-parametric Bayesian models. Features can be learned using deep architectures e.g. DBNs, deep Boltzmann machines (DBM), deep auto encoders, convolutional variants, ssRBMs, deep coding networks, DBNs with sparse feature learning, RNNs, conditional DBNs, de-noising autoencoders. This provides a better representation, allowing faster learning & more accurate classification with high-dimensional data. However, these architectures are poor at learning novel classes with few examples, because all network units are involved in representing input (a *distributed representation*) & must be adjusted together (high degree of freedom). Limiting degree of freedom reduces number of parameters to learn, facilitating learning of new classes from few examples. Hierarchical Bayesian (HB) models allow learning from few examples, e.g. for computer vision, statistics, & cognitive science.

  Compound HD architectures aim to integrate characteristics of both HB & deep networks. Compound HDP-DBM architecture is a *hierarchical Dirichlet process* (HDP) as a hierarchical model, incorporating DBM architecture. It is a full generative model, generalized from abstract concepts flowing through model layers, which is able to synthesize new examples in novel classes that look "reasonably" natural. All levels are learned jointly by maximizing a joint log-probability score.

  In a DBM with 3 hidden layers, probability of a visible input v is:

$$p(v,\psi) = \frac{1}{Z}\sum_h \exp\left(\sum_{ij} W_{ij}^{(1)} v_i h_j^1 + \sum_{jl} W_{jl}^{(2)} h_j^1 h_l^2 + \sum_{lm} W_{lm}^{(3)} h_l^2 h_m^3\right), \tag{329}$$

  where $h = \{h^{(1)}, h^{(2)}, h^{(3)}\}$: set of hidden units, $\psi = \{W^{(1)}, W^{(2)}, W^{(3)}\}$: model parameters, representing visible-hidden & hidden-hidden symmetric interaction terms.

  A learned DBM model is an undirected model that defines joint distribution $P(v, h^1, h^2, h^3)$. 1 way to express what was been learned is conditional model $P(v, h^1, h^2|h^3)$ & a prior term $P(h^3)$.

  Here $P(v, h^1, h^2|h^3)$ represents a conditional DBM model, which can be viewed as a 2-layer DBM but with bias terms given by states of h^3:

$$p(v,h^1,h^2|h^3) = \frac{1}{Z(\psi,h^3)}\sum_h \exp\left(\sum_{ij} W_{ij}^{(1)} v_i h_j^1 + \sum_{jl} W_{jl}^{(2)} h_j^1 h_l^2 + \sum_{lm} W_{lm}^{(3)} h_l^2 h_m^3\right). \tag{330}$$

- Deep predictive coding networks. A deep predictive coding network (DPCN) is a predictive coding scheme that uses top-down information to empirically adjust priors needed for a bottom-up inference procedure by means of a deep, locally connected, generative model. This works by extracting sparse features from time-varying observations using a linear dynamical model. Then, a pooling strategy is used to learn invariant feature representations. These units compose to form a deep architecture & are trained by greedy layer-wise unsupervised learning. Layers constitute a kind of Markov chain s.t. states at any layer depend only on preceding & succeeding layers.

  DPCNs predict representation of layer, by using a top-down approach using information in upper layer & temporal dependencies from previous states.

  DPCNs can be extended to form a convolutional network.

- Multilayer kernel machine. Multilayer kernel machines (MKM) are a way of learning highly nonlinear functions by iterative application of weakly nonlinear kernels. They use kernel principal component analysis (KPCA), as a method for unsupervised greedy layer-wise pre-training step of deep learning.

  Layer $l+1$ learns representation of previous layer l, extracting n_l principal component (PC) of projection layer l output in feature domain induced by kernel. To reduce dimensionality of updated representation in each layer, a supervised strategy selects best informative features among features extracted by KPCA. Process is:

  - rank n_l features according to their mutual information with class labels;
  - for different values of K & $m_l \in \{1, \dots, n_l\}$, compute classification error rate of a K-nearest neighbor (K-NN) classifier using only m_l most informative features on a validation set
  - value of m_l with which classifier has reached lowest error rate determines number of features to retain.

  Some drawbacks accompany KPCA method for MKMs.

  A more straightforward way to use kernel machines for deep learning was developed for spoken language understanding. Main idea: to use a kernel machine to approximate a shallow neural net with an infinite number of hidden units, then use a deep stacking network to splice output of kernel machine & raw input in building the next, higher level of kernel machine. Number of levels in deep convex network is a hyper-parameter of overall system, to be determined by cross validation." – Wikipedia/types of artificial neural networks

Tài liệu

[ABT18] Richard C. Aster, Brian Borchers, and Clifford H. Thurber. *Parameter estimation and inverse problems*. Third. Elsevier/Academic Press, Amsterdam, 2018, pp. xi+392. ISBN: 978-0-12-804651-7. DOI: 10.1016/C2015-0-02458-3. URL: https://doi.org/10.1016/C2015-0-02458-3.

[Bac24] Francis Bach. "Learning Theory from First Principles". In: Adaptive Computation and Machine Learning series (2024), p. 496.

[BB24] Christopher M. Bishop and Hugh Bishop. *Deep Learning: Foundations & Concepts*. 2024 edition. Springer, 2024, p. 669.

[DFO23] Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. 1st edition. Cambridge University Press, 2023, p. 398.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521 (2015), pp. 436–444. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[MC01] Danilo P. Mandic and Jonathon A. Chambers. "Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures and Stability". In: Wiley Series in Adaptive and Learning Systems for Signal Processing, Communications, and Control (2001), p. 304.

[NP23] Phan-Minh Nguyen and Huy Tuan Pham. "A rigorous framework for the mean field limit of multilayer neural networks". In: *Math. Stat. Learn.* 6.3-4 (2023), pp. 201–357. ISSN: 2520-2316. DOI: 10.4171/msl/42. URL: https://doi.org/10.4171/msl/42.

[RHP21] Rishikesh Ranade, Chris Hill, and Jay Pathak. "DiscretizationNet: a machine-learning based solver for Navier-Stokes equations using finite volume discretization". In: *Comput. Methods Appl. Mech. Engrg.* 378 (2021), Paper No. 113722, 20. ISSN: 0045-7825. DOI: 10.1016/j.cma.2021.113722. URL: https://doi.org/10.1016/j.cma.2021.113722.

[RPK19] M. Raissi, P. Perdikaris, and G. E. Karniadakis. "Physics-informed neural networks: a deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *J. Comput. Phys.* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2018.10.045. URL: https://doi.org/10.1016/j.jcp.2018.10.045.

[Rud87] Walter Rudin. *Real and complex analysis*. Third. McGraw-Hill Book Co., New York, 1987, pp. xiv+416. ISBN: 0-07-054234-1.

[Rud91] Walter Rudin. *Functional analysis*. Second. International Series in Pure and Applied Mathematics. McGraw-Hill, Inc., New York, 1991, pp. xviii+424. ISBN: 0-07-054236-8.

[Zha+23] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. 2023, p. 1111.