Machine Learning & Deep Learning – Học Máy & Học Sâu

Nguyễn Quản Bá Hồng*

Ngày 21 tháng 2 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

1.1 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.

1.2 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

2.1 [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.

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 $A \in \mathbb{R}^{n \times n}$, $A \succcurlyeq 0$ means A is positive semidefinite (i.e., all of its eigenvalues are nonnegative), & for 2 symmetric matrices A, B, $A \succcurlyeq B$ means $A - B \succcurlyeq 0$. 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 $f''(\mathbf{x}) \in \mathbb{R}^{d \times d}$.

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

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:

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

      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.

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

      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

$$M^{-1} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{ad-bc}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}. \tag{3}$$

      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 $x_1, \dots, x_n$ 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 $x_1, \dots, x_n$, 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.** $\mathcal{Y} = \{1, \dots, k\}, l(y, z) = 1_{y \neq z}$ (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*

$$\mathcal{R}(f) = \mathbb{E}[l(y, f(x))] = \int_{\mathcal{X}\times\mathcal{Y}} l(y, f(x)) \, \mathrm{d}p(x, y). \tag{4}$$

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*

$$\widehat{\mathcal{R}}(f) = \frac{1}{n} \sum_{i=1}^{n} l(y_i, f(x_i)). \tag{5}$$

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

$$\mathcal{R}(f) = \mathbb{E}[l(y, f(x))] = \mathbb{E}[\mathbb{E}[l(y, f(x))|x]], \tag{6}$$

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

$$\mathcal{R}(f) = \mathbb{E}_{x' \sim p}[\mathbb{E}[l(y, f(x'))|x = x']] = \int_{\mathcal{X}} \mathbb{E}[l(y, f(x'))|x = x'] \, \mathrm{d}p(x'). \tag{7}$$

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'):

$$r(z|x') = \mathbb{E}[l(y, z)|x = x'], \tag{8}$$

which leads to

$$\mathcal{R}(f) = \int_{\mathcal{X}} r(f(x')|x') \, \mathrm{d}p(x'). \tag{9}$$

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)*

$$f_\star(x') \in \arg\min_{z \in \mathcal{Y}} \mathbb{E}[l(y, z)|x = x'] = \arg\min_{z \in \mathcal{Y}} r(z|x'). \tag{10}$$

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

$$\mathcal{R}^\star = \mathbb{E}_{x' \sim p}\left[\inf_{z \in \mathcal{Y}} \mathbb{E}[l(y, z)|x = x']\right]. \tag{11}$$

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:

1. Bayes predictor is not always unique, but that all lead to same Bayes risk (e.g., in binary classification when $\mathbb{P}(y = 1|x) = \frac{1}{2}$)
2. 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).*

$\Rightarrow$ 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.

$$f_*(x') \in \arg\min_{z \in \{\pm 1\}} \mathbb{P}(y \neq z | x = x') = \arg\min_{z \in \{\pm 1\}} 1 - \mathbb{P}(y = z | x = x') = \arg\min_{z \in \{\pm 1\}} \mathbb{P}(y = z | x = x'). \quad (12)$$

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]

$$f_*(x') \in \arg\min_{z \in \mathbb{R}} \mathbb{E}[(y - z)^2 | x = x'] = \arg\min_{z \in \mathbb{R}} \left\{ \mathbb{E}[(y - \mathbb{E}[y|x = x'])^2 | x = x'] + (z - \mathbb{E}[y|x = x'])^2 \right\}. \quad (13)$$

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 $\mathbb{E}[y|x]$.*

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).

[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.

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 $(x_1, y_1), \dots, (x_n, y_n)$ 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. $\theta \in \Theta$:

$$\widehat{\mathcal{R}}(f_\theta) = \frac{1}{n}\sum_{i=1}^{n} l(y_i, f_\theta(x_i)). \tag{14}$$

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

$$\mathcal{R}(f_{\hat{\theta}}) - \mathcal{R}^\star = \left\{\mathcal{R}(f_{\hat{\theta}}) - \inf_{\theta'\in\Theta} \mathcal{R}(f_{\theta'})\right\} + \left\{\inf_{\theta'\in\Theta} \mathcal{R}(f_{\theta'}) - \mathcal{R}^*\right\} = \text{estimation error} + \text{approximation error}. \tag{15}$$

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 $\mathcal{R}(f_{\theta'})$:

$$\mathcal{R}(f_{\hat{\theta}}) - \mathcal{R}(f_{\theta'}) = \{\mathcal{R}(f_{\hat{\theta}}) - \widehat{\mathcal{R}}(f_{\hat{\theta}})\} + \{\widehat{\mathcal{R}}(f_{\hat{\theta}}) - \widehat{\mathcal{R}}(f_{\theta'})\} + \{\widehat{\mathcal{R}}(f_{\theta'} - \mathcal{R}(f_{\theta'})\} \le 2\sup_{\theta\in\Theta} |\widehat{\mathcal{R}}(f_\theta) - \mathcal{R}(f_\theta)| + \text{empirical optimization error}, \tag{16}$$

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

$$\widehat{\mathcal{R}}(f_\theta) + \lambda\Omega(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(y_i, f_\theta(x_i)) + \lambda\Omega(\theta), \tag{17}$$

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

$$\min_{\theta\in\mathbb{R}^d} \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta^\top\varphi(x_i))^2 + \lambda\|\theta\|_2^2. \tag{18}$$

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 $\delta \in (0, 1), \varepsilon > 0$: $\mathbb{P}(\mathcal{P}_p(\mathcal{A}(\mathcal{D}_n(p))) - \mathcal{R}_p^* \le \varepsilon) \ge 1 - \delta$. 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:

$$\widehat{\mathcal{R}}(\theta) = \frac{1}{n}\sum_{i=1}^n (y_i - \varphi(x_i)^\top \theta)^2. \tag{19}$$

    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 (283) 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:

$$\widehat{\mathcal{R}}(\theta) = \frac{1}{n}\|y - \Phi\theta\|_2^2, \tag{20}$$

    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 $d \leq n$, 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 (20) 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).
  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.

[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 $\widehat{\Sigma} = \frac{1}{n}\sum_{i=1}^n \varphi(x_i)\varphi(x_i)^\top$.

- *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 $(x_i, y_i), i = 1, \dots, n$, 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:

$$\frac{1}{n} \sum_{i=1}^{n} l(y_i, f(x_i)) + \Omega(f), \tag{21}$$

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

$$f(\mathbf{x}) = \sum_{j=1}^{m} \eta_j \sigma(\mathbf{w}_j^\top \mathbf{x} + b_j), \tag{22}$$

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

$$-1_{y=1} \log g(x) - 1_{y=-1} \log(1-g(x)) = 1_{y=1} \log(1+e^{-f(x)}) + 1_{y=-1} \log(1+e^{f(x)}) \tag{23}$$

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:

1. optimization error (convergence properties of algorithms for minimizing risk),
2. estimation error (effect of having a finite amount of data on prediction performance),
3. 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)

$$\min_{\theta \in \mathbb{R}^{m(d+2)}} \frac{1}{n} \sum_{i=1}^{n} l\left(y_i, \sum_{j=1}^{m} \eta_j \sigma(\mathbf{w}_j^\top \mathbf{x}_i + b_j)\right), \tag{24}$$

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.

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

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 $\varepsilon_i \in \{\pm 1\}, i = 1,\dots,n$:

$$R_n(\mathcal{G}) = \mathbb{E}\left[\sup_{\theta\in\Theta} \frac{1}{n}\sum_{i=1}^n \varepsilon_i l(y_i, f_\theta(x_i))\right]. \tag{25}$$

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

$$\mathbb{E}\left[\mathcal{R}(\hat{f}) - \inf_{f\in\mathcal{F}} \mathcal{R}(f)\right] \leq 4R_n(\mathcal{G}). \tag{26}$$

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]`.

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 $\mathcal{F}$,*

$$\mathbb{E}\left[\mathcal{R}(\hat{f}) - \inf_{f\in\mathcal{F}} \mathcal{R}(f)\right] \leq \frac{16GDR}{\sqrt{n}}. \tag{27}$$

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 $a_1 < \cdots < a_{m-2}$ 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)

$$f(x) = \int_K (\mathbf{w}^\top \mathbf{x} + b)_+ \,\mathrm{d}\nu(\mathbf{w}, b), \tag{28}$$

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

$$\tilde{\gamma}_1(f) = \int_{\nu \in \widetilde{\mathcal{M}}(K)} \int_K |d\nu(\mathbf{w}, b)| \text{ s.t. } \forall x \in \mathcal{X},\ f(x) = \int_K (\mathbf{w}^\top \mathbf{x} + b)_+ \,\mathrm{d}\nu(\mathbf{w}, b), \tag{29}$$

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)

$$\gamma_1(f) = \inf_{\nu \in \mathcal{M}(K)} \int_K |d\nu(\mathbf{w}, b)| \text{ s.t. } \forall x \in \mathcal{X},\ f(x) = \int_K (\mathbf{w}^\top \mathbf{x} + b)_+ \,\mathrm{d}\nu(\mathbf{w}, b), \tag{30}$$

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 $\mathcal{F}_1$.*

[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.

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]`

$$\gamma_1(f) \leq \frac{1}{2}|\frac{1}{2R}[f(-R) + f(R)] + v_0| + \frac{1}{2}|\frac{1}{2R}[f(-R) + f(R)] - v_{m-2}| + \sum_{j=1}^{m-2} |v_j - v_{j-1}|. \tag{31}$$

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 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 $j \in \{0, \ldots, 2s\}$, & piecewise affine interpolation $\hat{f}$ from values $a_j, f(a_j)$ (& slopes v_j on $[a_j, a_{j+1}]$), with $j \in \{0, \ldots, 2s\}$, 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

$$\frac{1}{2}|\frac{1}{2R}[f(-R) + f(R)] + f'(-R)| + \frac{1}{2}|\frac{1}{2R}[f(-R) + f(R)] - f'(R)| + \frac{R}{s}\sum_{j=1}^{2s-1} |f''\left(-R + \frac{j}{s}R\right)|. \tag{32}$$

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)

$$\gamma_1(f) \leq \frac{1}{2}|\frac{1}{2R}[f(-R) + f(R)] + f'(-R)| + \frac{1}{2}|\frac{1}{2R}[f(-R) + f(R)] - f'(R)| + \int_{-R}^{R} |f''(x)|\,\mathrm{d}x. \tag{33}$$

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:

$$\gamma_1(f) \leq \frac{1}{2R}|f(-R) + f(R)| + \frac{1}{2}|f'(R)| + \frac{1}{2}|f'(-R)| + \int_{-R}^{R} |f''(x)|\,\mathrm{d}x. \tag{34}$$

Problem 11. *Assume $-R = x_1 < \cdots < x_n = R, y_1, \ldots, y_n \in \mathbb{R}$, 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)

$$\gamma_1(f) \leq \frac{1}{(2\pi)^d} \int_{\mathbb{R}^d} |\hat{f}(\boldsymbol{\omega})| \gamma_1(\mathbf{x} \mapsto e^{\mathrm{i}\boldsymbol{\omega}^\top \mathbf{x}}) \, \mathrm{d}\boldsymbol{\omega} \leq \frac{2}{(2\pi)^d R} \int_{\mathbb{R}^d} |\hat{f}(\boldsymbol{\omega})| (1 + 2R^2 \|\boldsymbol{\omega}\|_2^2) \, \mathrm{d}\boldsymbol{\omega}. \tag{35}$$

  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

$$\mathbf{y}^{(k)} = (W^{(k)})^\top \mathbf{x}^{(k-1)}, \tag{36}$$

$$\mathbf{x}^{(k)} = \sigma(\mathbf{y}^{(k)}), \tag{37}$$

  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:

$$\mathbf{y}^{(k)} = (W^{(k)})^\top \mathbf{x}^{(k-1)}, \tag{38}$$

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} + \sigma(\mathbf{y}^{(k)}). \tag{39}$$

  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):

$$\text{attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V. \tag{40}$$

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.

2.2 [BHH06]. Francis R. Bach, David Heckerman, Eric Horvitz. Considering Cost Asymmetry in Learning Classifiers

[188 citations]

- Abstract. Receiver Operating Characteristics (ROC) curves are a standard way to display performance of a set of binary classifiers ∀ feasible ratios of costs associated with false positives & false negatives. For linear classifiers, set of classifiers is typically obtained by training once, holding constant estimated slope & then varying intercept to obtain a parameterized set of classifiers whose performances can be plotted in ROC plane. Consider alternative of varying asymmetry of cost function used for training. Show: ROC curve obtained by varying both intercept & asymmetry, & hence slope, always outperforms ROC curve obtained by varying only intercept. In addition, present a path-following algorithm for support vector machine (SVM) that can compute efficiently entire ROC curve, & that has the same computational complexity as training a single classifier. Finally, provide a theoretical analysis of relationship between asymmetric cost model assumed when training a classifier & cost model assumed in applying classifier. In particular, show: mismatch between step function used for testing & its convex upper bounds, usually used for training, leads to a provable & quantifiable difference around extreme asymmetries.

  Keywords. SVM, receiver operating characteristic (ROC) analysis, linear classification

- 1. Introduction. Receiver Operating Characteristic (ROC) analysis has seen increasing attention in recent statistics & ML literature (Pepe, 2000; Provost and Fawcett, 2001; Flach, 2003). ROC is a representation of choice for displaying performance of a classifier when costs assigned by end users to false positives & false negatives are not known at time of training. E.g., when training a classifier for identifying cases of undesirable unsolicited email, end users may have different preferences about likelihood of a false negative & false positive. ROC curve for such a classifier reveals ratio of false negatives & positives at different probability thresholds for classifying an email message as unsolicited or normal email.

  In this paper, consider linear binary classification of points in an Euclidean space – noting: it can be extended in a straightforward manner to nonlinear classification problems by using Mercer kernels (Schölkopf & Smola, 2002). I.e., given data $\mathbf{x} \in \mathbb{R}^d$, $d \geq 1$, consider classifiers of form $f(\mathbf{x}) = \text{sign}(\mathbf{w}^\top \mathbf{x} + b)$, where $\mathbf{w} \in \mathbb{R}^d, b \in \mathbb{R}$ are referred to as *slope* & *intercept*. To date, ROC curves have been usually constructed by training once, holding constant estimated slope & varying intercept to obtain curve. In this paper, show: while that procedure appears to be most practical thing to do, it may lead to classifiers with poor performance in some parts of ROC curve.

Crux of our approach: allow asymmetry of cost function to vary, i.e., vary ratio of cost of a false positive & cost of a false negative. For each value of ratio, obtain a different slope & intercept, each optimized for this ratio. In a naive implementation, varying asymmetry would require a retraining of classifier for each point of ROC curve, which would be computational expensive. In Sect. 3.1, present an algorithm that can compute solution of a support vector machine (SVM) (see, e.g., Sch ölkopf & Smola, 2002; Shawe-Taylor & Cristianini, 2004) for all possible costs of false positives & false negatives, with same computational complexity as obtaining solution for only 1 cost function. Algorithm extends to asymmetric costs algorithm of Hastie et al. (2005) & is based on path-following techniques that take advantage of piecewise linearity of path of optimal solutions. In Sect. 3.2, show how path-following algorithm can be used to obtain ROC curves. In particular, by allowing both asymmetry & intercept to vary, can obtain better ROC curves than by methods that simply vary intercept.

In Sect. 4, provide a theoretical analysis of relationship between asymmetry of costs assumed in training a classifier & asymmetry desired in its application. In particular, show: even in population (i.e., infinite sample) case, use of a training loss function which is a convex upper bound on true or testing loss function (a step function) creates classifiers with sub-optimal accuracy. Quantify this problem around extreme asymmetries for several classical convex-upper-bound loss functions, including square loss & *erf loss*, an approximation of logistic loss based on normal cummulative distribution functions (also referred to as "error function", & usually abbreviated as erf). Analysis is carried through for Gaussian & mixture of Gaussian class-conditional distributions (see Sect. 4 for more details). Main result of this analysis: given an extreme user-defined testing asymmetry, training asymmetry should almost always be chosen to be less extreme.

Consequences of potential mismatch between cost functions assumed in testing vs. training underscore value of using algorithm that we introduce in Sect. 4.3. Even when costs are known (i.e., when only 1 point on ROC curve is needed), classifier resulting from our approach, which builds entire ROC curve, is never less accurate & can be more accurate than one trained with known costs using a convex-upper-bound loss function. Indeed, show in Sect. 4.3: computing entire ROC curve using our algorithm can lead to substantial gains over simply training once.

Paper is organized as follows:

- Sect. 2: give an introduction of linear classification problem & review ROC framework.
- Sect. 3: contains algorithm part of paper, while
- Sect. 4 provide a theoretical analysis of discrepancy between testing & training asymmetries, together with empirical results.

This paper is an extended version of previously published work (Bach et al., 2005a).

- 2. Problem Overview. Given data $\mathbf{x} \in \mathbb{R}^d$ & labels $y \in \{\pm 1\}$, consider linear classifiers of form $f(\mathbf{x}) = \text{sign}(\mathbf{w}^\top \mathbf{x} + b)$, where $\mathbf{w}$: *slope* of classifier & b: *intercept*. A classifier is determined by parameters $(\mathbf{w}, b) \in \mathbb{R}^{d+1}$. In Sect. 2.1, introduce notation & defs; in Sect. 2.2: lay out necessary concepts of ROC analysis, while in Sect. 2.3, describe how these classifiers & ROC curves are typically obtained from data.

  - 2.1. Asymmetric Cost & Loss Functions. Positive (resp. negative) examples are those for which $y = 1$ (resp. $y = -1$). 2 types of misclassification, false positives & false negatives, are assigned 2 different costs. Let C_+ denote cost of a false negative & C_- cost of a false positive. Total *expected* cost is equal to

$$R(C_+, C_-, \mathbf{w}, b) = C_+ P\{\mathbf{w}^\top \mathbf{x} + b < 0, y = 1\} + C_- P\{\mathbf{w}^\top \mathbf{x} + b \geq 0, y = -1\}.$$

    In context of large margin classifiers (see, e.g., Bartlett et al., 2004), expected cost is usually expressed in terms of *0–1 loss function*; indeed, if let $\phi_{0-1}(u) = 1_{u<0}$ be 0–1 loss, can write expected cost as

$$R(C_+, C_-, \mathbf{w}, b) = C_+ E\{1_{y=1}\phi_{0-1}(\mathbf{w}^\top \mathbf{x} + b)\} + C_- E\{1_{y=-1}\phi_{0-1}(-\mathbf{w}^\top \mathbf{x} - b)\},$$

    where E denotes expectation w.r.t. joint distribution of $(\mathbf{x}, y)$.

    Expected cost defined using 0–1 loss is cost that end users are usually intersected in during use of classifier, while other cost functions that define below are used solely for training purposes. Convexity of these cost functions makes learning algorithms convergent without local minima, & leads to attractive asymptotic properties (Bartlett et al., 2004).

    A traditional set-up for learning linear classifiers from labeled data: consider a convex upper bound ϕ on 0–1 loss ϕ_{0-1}, & to use expected ϕ-cost:

$$R_\phi(C_+, C_-, \mathbf{w}, b) = C_+ E\{1_{y=1}\phi(\mathbf{w}^\top \mathbf{x} + b)\} + C_- E\{1_{y=-1}\phi(-\mathbf{w}^\top \mathbf{x} - b)\}.$$

    Refer to ratio $\frac{C_+}{C_- + C_+}$ as *asymmetry*. Shall use *training asymmetry* to refer to asymmetry used for training a classifier using a ϕ-cost, & *testing asymmetry* to refer to asymmetric cost characterizing testing situation (reflecting end user preferences) with actual cost based on 0–1 loss. In Sect. 4, show: these may be different in general case.

    Shall consider several common loss functions. Some of loss functions (square loss, hinge loss) leads to attractive computational properties, while others (square loss, erf loss) are more amenable to theoretical manipulations (see Fig. 1: Loss functions. Left: 0–1 loss, hinge loss, erf loss, square loss. Right: 0–1 loss, probit loss, logistic loss. for plot of loss functions, as they are commonly used & defined below. Note: by rescaling, each of these loss functions can be made to be an upper bound on 0–1 loss which is tight at 0.):

    * **square loss**: $\phi_{\text{sq}}(u) = \frac{1}{2}(u-1)^2$; classifier is equivalent to linear regression on y,

∗ **hinge loss**: $\phi_{\text{hi}}(u) = \max\{1-u, 0\}$; classifier is support vector machine (Shawe-Taylor & Cristianini, 2004),

∗ **erf loss**: $\phi_{\text{erf}}(u) = [u\Psi(u) - u + \Psi'(u)]$, where Π: cumulative distribution of standard normal distribution, i.e.: $\Psi(v) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{v} e^{-\frac{t^2}{2}}\,\text{d}t$ & $\Psi'(v) = \frac{1}{\sqrt{2\pi}}e^{-\frac{v^2}{2}}$. Erf loss can be used to provide a good approximation of *logistic loss* $\phi_{\log}(u) = \log(1+e^{-u})$ as well as its derivative, & is amenable to closed-form computations for Gaussians & mixture of Gaussians (see Sect. 4 for more details). Note: erf loss is different from *probit loss* – $\log\Psi(u)$, which leads to probit regression (Hastie et al., 2001).

◦ 2.2. ROC Analysis. Aim of ROC analysis: display in a single graph performance of classifiers ∀ possible costs of misclassification. Consider sets of classifiers $f_\gamma(x)$, parameterized by a variable $\gamma \in \mathbb{R}$ (γ can either be intercept or training asymmetry). For a classifier $f(x)$, can define a point (u, v) in "ROC plane", where u: proportion of false positives $u = P(f(x) = 1|y = -1)$, & v: proportion of true positives $v = P(f(x) = 1|y = 1)$. When γ is varied, obtain a curve in ROC plane, ROC curve (see, e.g., Fig. 2: Left: ROC curve: regular ROC curve; convex envelope. Points a, c are ROC-consistent & point b is ROC-inconsistent. Right: ROC curve & dashed equi-cost lines: All lines have direction (p_+C_+, p_-C_-), plain line is optimal & point a is optimal classifier.). Whether γ is intercept or training asymmetry, ROC curve always passes through point $(0,0), (1,1)$, which corresponds to classifying all points as negative (resp. positive).

Upper convex envelope of curve is set of optimal ROC points that can be achieved by set of classifiers; indeed, if a point in envelope is not 1 of original points, it must lie in a segment between 2 points $(u(\gamma_0), v(\gamma_0)), (u(\gamma_1), v(\gamma_1))$, & all points in a segment between 2 classifiers can always be attained by choosing randomly between 2 classifiers. Note: this classifier itself is not a linear classifier; its performance, defined by given true positive & false positive rates, can only be achieved by a mixture of 2 linear classifiers. However, if user is only interested in testing cost, which is a weighted linear combination of true positive & false positive rates, lowest testing cost is always achieved by both of these 2 classifiers (Sect. 4.3).

Denoting $p_+ = P(y = 1), p_- = P(y = -1)$, expected (C_+, C_-)-cost for a classifier (u, v) in ROC space, is simply $p_+C_+(1-v)+p_-C_-u$, & thus optimal classifiers for (C_+, C_-)-cost can be found by looking at lines of slope that are normal to direction $(p_-C_-, -p_+C_+)$, which intersects ROC curve & are as close as point $(0, 1)$ as possible (see Fig. 2).

A point $(u(\gamma), v(\gamma))$ is said to be *ROC-consistent* if it lies on upper convex envelope; in this case, tangent direction $\left(\frac{du}{d\gamma}, \frac{dv}{d\gamma}\right)$ defines a cost $(C_+(\gamma), C_-(\gamma))$ for which classifier is optimal (for testing cost, which is defined using 0–1 loss). Condition introduced earlier, namely that $\left(\frac{du}{d\gamma}, \frac{dv}{d\gamma}\right)$ is normal to direction $(p_-C_-, -p_+C_+)$, leads to:

$$p_-C_-\frac{du}{d\gamma}(\gamma) - p_+C_+\frac{dv}{d\gamma}(\gamma) = 0.$$

Optimal testing asymmetry $\beta(\gamma)$ defined as ratio $\frac{C_+(\gamma)}{C_+(\gamma)+C_-(\gamma)}$, is thus =

$$\beta(\gamma) \coloneqq \frac{C_+(\gamma)}{C_+(\gamma)+C_-(\gamma)} = \frac{1}{1+\frac{p_+\frac{dv}{d\gamma}(\gamma)}{p_-\frac{du}{d\gamma}(\gamma)}}.$$

If a point $(u(\gamma), v(\gamma))$ is ROC-inconsistent, then quantity $\beta(\gamma)$ has no meaning, & such a classifier is generally useless, because, ∀ settings of misclassification cost, that classifier can be outperformed by other classifiers or a combination of classifiers. See Fig. 2 for examples ROC-consistent & ROC-inconsistent points.

In Sect. 4, relate optimal asymmetry of cost in testing or eventual use of a classifier in real world, to asymmetry of cost used to train that classifier; in particular, show: they differ & quantify this difference for extreme asymmetries (i.e., close to corner points $(0,0), (1,1)$). This analysis highlights value of generating entire ROC curve, even when only 1 point is needed (Sect. 4.3).

Handling ROC surfaces. Also consider varying both asymmetry of cost function & intercept, leading to a set of points in ROC plane parameterized by 2 real values. Although concept of ROC-consistency could easily be extended to ROC surfaces, for simplicity do not consider it here. In all experiments, those ROC surfaces are reduced to curves by computing their convex upper envelopes.

◦ 2.3. Learning From Data. Given n labeled data points $(x_i, y_i), i = 1, \ldots, n$, *empirical cost* =

$$\hat{R}(C_+, C_-, \mathbf{w}, b) = \frac{C_+}{n}\#\{y_i(\mathbf{w}^\top\mathbf{x}_i + b) < 0, y_i = 1\} + \frac{C_-}{n}\#\{y_i(\mathbf{w}^\top\mathbf{x}_i + b) < 0, y_i = -1\}.$$

Empirical ϕ-cost =

$$\hat{R}_\phi(C_+, C_-, \mathbf{w}, b) = \frac{C_+}{n}\sum_{i\in I_+}\phi(y_i(\mathbf{w}^\top\mathbf{x}_i + b)) + \frac{C_-}{n}\sum_{i\in I_-}\phi(y_i(\mathbf{w}^\top\mathbf{x}_i + b)),$$

where $I_+ = \{i, y_i = 1\}, I_- = \{i, y_i = -1\}$. When learning a classifier from data, a classical setup: minimize sum of *empirical ϕ-cost* & a regularization term $\frac{1}{2n}\|\mathbf{w}\|^2$, i.e., to minimize $\hat{J}_\phi(C_+, C_-, \mathbf{w}, b) = \hat{R}_\phi(C_+, C_-, \mathbf{w}, b) + \frac{1}{2n}\|\mathbf{w}\|^2$.

Note: objective function is no longer homogeneous in (C_+, C_-); sum $C_+ + C_-$ is referred to as *total amount of regularization*. Thus, 2 end-user-defined parameters are needed to train a linear classifier: *total amount of regularization* $C_+ + C_- \in \mathbb{R}^+$, & *asymmetry* $\frac{C_+}{C_+ + C_-} \in [0, 1]$. In Sect. 3.1, show how minimum of $\hat{J}_\phi(C_+, C_-, \mathbf{w}, b)$, w.r.t. $\mathbf{w}$ & b, can be computed efficiently

for hinge loss, for many values of (C_+, C_-), with a computational cost that is within a constant factor of computational cost of obtaining a solution for 1 value of (C_+, C_-).

Building an ROC curve from data. If a sufficiently large validation set is available, can train on training set & use empirical distribution of validation data to plot ROC curve. If sufficient validation data is not available, then can use several (typically 10 or 25) random splits of data & average scores over those splits to obtain ROC scores. Can also use this approach to obtain confidence intervals (Flach, 2003).

- 3. Building ROC Curves for SVM. Present an algorithm to compute ROC curves for SVM that explores 2D space of cost parameters (C_+, C_-) efficiently. 1st show how to obtain optimal solutions of SVM without solving optimization problems many times for each value of (C_+, C_-). This method generalizes results of Hastie et al. (2005) to case of asymmetric cost functions. Then describe how space (C_+, C_-) can be appropriately explored & how ROC curves can be constructed.
  - 3.1. Building Paths of Classifiers. Given n data points x_i, $i = 1, \ldots, n$ which belongs to $\mathbb{R}^d$, & n labels $y_i \in \{\pm 1\}$, minimizing regularized empirical hinge loss $\Leftrightarrow$ solving following convex optimization problem (Schölkopf & Smola, 2002):

$$\min_{\mathbf{w},b,\xi} \sum_i C_i \xi_i + \frac{1}{2}\|\mathbf{w}\|^2 \text{ s.t. } \forall i,\ \xi_i \geq 0,\ \xi_i \geq 1 - y_i(\mathbf{w}^\top \mathbf{x}_i + b),$$

where

$$C_i = \begin{cases} C_+ & \text{if } y_i = 1, \\ C_- & \text{if } y_i = -1. \end{cases}$$

Optimality conditions & dual problems. Know derive usual Karush–Kuhn–Tucker (KKT) optimality conditions (Boyd & Vandenberghe, 2003). Lagrangian of problem is (with dual variables $\alpha, \beta \in \mathbb{R}^n_+$):

$$L(\mathbf{w}, b, \xi, \alpha, \beta) = \sum_i C_i \xi_i + \frac{1}{2}\|\mathbf{w}\|^2 + \sum_i \alpha_i (1 - y_i(\mathbf{w}^\top \mathbf{x}_i + b)) - \xi_i - \sum_i \beta_i \xi_i.$$

Derivatives w.r.t. primal variables are

$$\frac{\partial L}{\partial w} = w - \sum_i \alpha_i y_i x_i,\ \frac{\partial L}{\partial b} = -\sum_i \alpha_i y_i,\ \frac{\partial L}{\partial \xi_i} = C_i - \alpha_i - \beta_i.$$

1st set of KKT conditions corresponds to nullity of Lagrangian derivatives w.r.t. primal variables, i.e.: (2)

$$w = \sum_i \alpha_i y_i x_i,\ \sum_i \alpha_i y_i = \alpha^\top \mathbf{y} = 0,\ \forall i,\ C_i = \alpha_i + \beta_i.$$

Slackness conditions (Điều kiện lỏng lẻo) are:

$$\forall i,\ \alpha_i (1 - \xi_i + y_i(\mathbf{w}^\top \mathbf{x}_i + b)) = 0,\ \beta_i \xi_i = 0.$$

Finally dual problem can be obtained by computing minimum of Lagrangian w.r.t. primal variables. If denote K $n \times n$ Gram matrix of inner products, i.e., defined by $K_{ij} = \mathbf{x}_i \top \mathbf{x}_j$ & $\widetilde{K} = \text{Diag}(\mathbf{y}) K \text{Diag}(\mathbf{y})$, dual problem is:

$$\max_{\alpha \in \mathbb{R}^n} -\frac{1}{2}\alpha^\top \widetilde{K} \alpha + 1^\top \alpha \text{ s.t. } \alpha^\top \mathbf{y} = 0,\ \forall i,\ 0 \leq \alpha_i \leq C_i.$$

In following, only consider primal variables $(\mathbf{w}, b, \xi)$ & dual variables (α, β) that verify 1st set of KKT condition (2), which implies: $\mathbf{w}, \beta$ are directly determined by α.

Piecewise affine solutions. Following Hastie et al. (2005), for an optimal set of primal-dual variables $(\mathbf{w}, b, \alpha)$, can separate data points in 3 disjoint sets, depending on values of $y_i(\mathbf{w}^\top \mathbf{x}_i + b) = (\widetilde{K}\alpha)_i + b y_i$.

  * Margin: $\mathcal{M} = \{i,\ y_i(\mathbf{w}^\top \mathbf{x}_i + b) = 1\}$,
  * Left of margin: $\mathcal{M} = \{i,\ y_i(\mathbf{w}^\top \mathbf{x}_i + b) < 1\}$,
  * Right of margin: $\mathcal{M} = \{i,\ y_i(\mathbf{w}^\top \mathbf{x}_i + b) > 1\}$.

  - 3.2. Constructing ROC Curve.

- 4. Training vs. Testing Asymmetry.
  - 4.1. Optimal Solutions for Extreme Cost Asymmetries.
  - 4.2. Expansion of Testing Asymmetries.
  - 4.3. Building Entire ROC Curve for a Single Point.

- 5. Conclusion. Have presented an efficient algorithm to build ROC curves by varying training cost asymmetries for SVMs. Algorithm is based on piecewise linearity of path of solutions when cost of false positives & false negatives vary. Have also provided a theoretical analysis of relationship between potentially different cost asymmetries assumed in training & testing classifiers, showing: they differ under certain circumstances. In particular, in case of extreme asymmetries, our theoretical analysis suggests: training asymmetries should be chosen less extreme than testing asymmetry.

  Have characterized key relationships, & have worked to highlight potential practical value of building entire ROC curve even when a single point may be needed. All learning algorithms considered in this paper involve using a convex surrogate (lồi thay thế) to correct non differentiable non convex loss function. Our theoretical analysis implies: because use a convex surrogate, using testing asymmetry for training leads to non-optimal classifiers. Thus propose to generate all possible classifiers corresponding to all training asymmetries, & select the one that optimizes a good approximation to true loss function on unseen data (i.e., using held out data or cross validation). As shown in Sect. 3, it turns out: this can be done efficiently for SVM. Such an approach can lead to a significant improvement of performance with little added computational cost.

  Note: although have focused in this paper on single kernel learning problem, our approach can be readily extended to multiple kernel learning setting (Bach et al., 2005b) with appropriate numerical path following techniques.

- Appendix A. Proof of Expansion of Optimal Solutions.

- Appendix B. Proof of Expansion of Testing Asymmetries.

2.3 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.

$$\theta(u) = [u, u^2, u_x, uu_x, u^2u_x, u_{xx}, uu_{xx}, u^2u_{xx}, u_{xxx}, uu_{xxx}, u^2u_{xxx}]. \tag{41}$$

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

$$u_t = \sum_{i=1}^{K} \xi_i f_i(u) = \theta(u)\boldsymbol{\xi}. \tag{42}$$

  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)

$$(\mathbf{U}_t)_c = (\boldsymbol{\Theta}(u))_c\boldsymbol{\xi}, \tag{43}$$

  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)

$$\boldsymbol{\xi} = \arg\min_{\boldsymbol{\xi}} \|(\mathbf{U}_t)_c - (\boldsymbol{\Theta}(u))_c\boldsymbol{\xi}\|_2^2 + R(\boldsymbol{\xi}). \tag{44}$$

  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

$$L(u_t) = L(\theta)\boldsymbol{\xi}, \tag{45}$$

    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

$$\widetilde{\mathbf{U}}_t = \boldsymbol{\Theta}_t^\top \mathbf{U}_t \boldsymbol{\Theta}_x, \ \widetilde{\mathbf{F}}_k = \boldsymbol{\Theta}_t^\top \mathbf{F}_k \boldsymbol{\Theta}_x, \tag{46}$$

    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

$$(\widetilde{\mathbf{U}}_t)_c = (\widetilde{\boldsymbol{\Theta}})_c\boldsymbol{\xi}, \tag{47}$$

    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 $\boldsymbol{\Theta}_t$ & $\boldsymbol{\Theta}_x$ are given by singular value decomposition of $\mathbf{U}$:

$$\mathbf{U} \cong \mathbf{W}_r\boldsymbol{\Sigma}_r\mathbf{V}_r^* = \boldsymbol{\Theta}_t\boldsymbol{\Sigma}_r\boldsymbol{\Theta}_x^*, \tag{48}$$

    where $\boldsymbol{\Theta}_x^*$: transpose of $\boldsymbol{\Theta}_x$, r: a user-specified hyperparameter, which indicates that only 1st r bases with larger energy are retained, & $\boldsymbol{\Sigma}_r = \mathrm{diag}\lambda_i$ 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]

$$u_n = u + \sigma \times \mathrm{std}(u) \times \mathrm{randn}, \tag{49}$$

  where u: numerical solution, σ: noise level, $\mathrm{std}(u)$: 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

$$\mathrm{error}_i = \left| \frac{\widehat{\xi}_i - \xi_i}{\xi_i} \right|, \tag{50}$$

  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 $u_t = \xi_1 u u_x + \xi_2 u_{xx} + \xi_3 u_{xxxx}$ 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

$$u_t = D\nabla^2 u + \lambda u - \omega v, \tag{51}$$

$$v_t = D\nabla^2 v + \omega u + \lambda u, \tag{52}$$

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:

$$\boldsymbol{\theta}_{u_t} = [\nabla^2 u, u, u^2, u^3, v, v^2, v^3, uv, u^2 v, uv^2],\ \boldsymbol{\theta}_{v_t} = [\nabla^2 v, u, u^2, u^3, v, v^2, v^3, uv, u^2 v, uv^2]. \tag{53}$$

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.

2.4 [Cho21]. Francois Chollet. Deep Learning with Python. 2021

[401 Amazon ratings]

Amazon review. Unlock groundbreaking advances of DL with this extensively revised new edition of bestselling original. Learn directly from creator of Keras & master practical Python DL techniques that are easy to apply in real world.

In *DL with Python 2e* will learn:

- DL from 1st principles
- Image classification & image segmentation
- Timeseries forecasting
- Text classification & machine translation
- Text generation, neural style transfer, & image generation

DL Learning with Python has taught thousands of readers how to put full capabilities of DL into action. This extensively revised full color 2nd edition introduces DL using Python & Keras, & is loaded with insights for both novice (người mới vào nghề) & experienced ML practitioners. Learn practical techniques that are easy to apply in real world, & important theory for perfecting neural networks.

About technology. Recent innovations in DL unlock exciting new software capabilities like automated language translation, image recognition, & more. DL is quickly becoming essential knowledge for every software developer, & modern tools like Keras & TensorFlow put it within your reach – even if have no background in mathematics or DS. This book shows you how to get started.

About book. *DL Learning with Python 2e* introduces field of DL using Python & powerful Keras library. In this revised & expanded new edition, Keras creator Francois Chollet offers insights for both novice & experienced ML practitioners. As move through this book, build understanding through intuitive explanations, crisp color illustrations, & clear examples. Quickly pick up skills need to start developing DL applications.

About reader. For readers with immediate Python skills. No previous experience with Keras, TensorFlow, or ML is required.

About Author. Francois Chollet is a software engineer at Goolge & creator of Keras DL library.

"Chollet explains complex concepts with minimal fuss. A joy to read." – Martin Görner, Google

- Preface. If picked up this book, probably aware of extraordinary process that DL has represented for field of AI in recent past. Went from near-unusable computer vision & natural language processing to highly performant systems deployed at scale in products you use every day. Consequences of this sudden progress extend to almost every industry. Already applying DL to an amazing range of important problems across domains as different as medical imaging, agriculture, autonomous driving, education, disaster prevention, & manufacturing.

  Believe DL is still in its early days. DL has only realized a small fraction of its potential so far. Over time, DL will make its way to every problem where it can help – a transformation that will take place over multiple decades.

  In order to begin deploying DL technology to every problem that it could solve, need to make it accessible to as many people as possible, including non-experts – people who aren't researchers or graduate students. For DL to reach its full potential, need to radically democratize (dân chủ hóa triệt để) it. & today, believe we are at cusp of a historical transition, where DL is moving out of academic labs & R&D departments of large tech companies to become a ubiquitous part of toolbox of every developer out there – not unlike trajectory of web development in late 1990s. Almost anyone can now build a website or web app for their business or community of a kind that would have required a small team of specialist engineers in 1998. In not-so-distant future, anyone with an idea & basic coding skills will be able to build smart applications that learn from data.

  When Chollet released 1st version of Keras DL framework in Mar 2015, democratization of AI wasn't what Chollet had in mind. Had been doing research in ML for several years & had built Keras to help with his own experiments. But since 2015, hundreds of thousands of newcomers have entered field of DL; many of them picked up Keras as their tool of choice. As watched scores of smart people use Keras in unexpected, powerful ways, came to care deeply about accessibility & democratization of AI. Realized: further spread these technologies, more useful & valuable they become. Acessibility quickly became an explicit goal in development of Keras, & over a few short years, Keras developer community has made fantastic achievements on this front. Put DL into hands of hundreds of thousands of people, who in turn are using it to solve problems that were until recently thought to be unsolvable.

  Book is another step on way to making DL available to as many people as possible. Keras had always needed a companion course to simultaneously cover fundamentals of DL, DL best practices, & Keras usage patterns. In 2016 & 2017, did best to produce such a course, which became 1e of this book, released in Dec 2017. Quickly became a ML best seller that solder > 50000 copies & was translated into 12 languages.

  However, field of DL advances fast. Since release of 1e, many important developments have taken place – release of TensorFlow 2, growing popularity of Transformer architecture, & more. & so, in late 2019, set out to update book. Originally thought, quite naively, that it would feature about 50% new content & would end up being roughly same length as 1e. In practice, after 2 years of work, it turned out to be over a 3rd longer, with about 75% novel content. More than a refresh, it is a whole new book.

  Wrote it with a focus on making concepts behind DL, & their implementation, as approachable as possible. Doing so didn't require to dumb down anything – strongly believe: there are no difficult ideas in DL. Hope find this book valuable & it will enable you to begin building intelligent applications & solve problems that matter to you.

  Over past 6 years, Keras has grown to have hundreds of open source contributors & > 1 million users. Fantastic to see Keras adopted as TensorFlow's high-level API. A smooth integration between Keras & TensorFlow greatly benefits both TensorFlow users & Keras users, & makes DL accessible to most.

- About this book. This book was written for anyone who wishes to explore DL from scratch or broaden their understanding of DL. Whether a practicing ML engineer, a software developer, or a college student, find value in these pages.

  Explore DL in an approachable way – starting simply, then working up to state-of-art techniques. Find: this book strikes a balance between intuition, theory, & hands-on practice. It avoids mathematical notation, preferring instead to explain code ideas of ML & DL via detailed code snippets & intuitive mental models. Learn from abundant code examples that include extensive commentary, practical recommendations, & simple high-level explanations of everything need to know to start using DL to solve concrete problems.

Code examples use Python DL framework Keras, with TensorFlow 2 as its numerical engine. They demonstrate modern Keras & TensorFlow 2 best practices as of 2021.

After reading this book, have a solid understand of what DL is, when it's applicable, & what its limitations are. Familiar with standard workflow for approaching & solving ML problems, & know how to address commonly encountered issues. Be able to use Keras to tackle real-world problems ranging from computer vision to natural language processing: image classification, image segmentation, timeseries forecasting, text classification, machine translation, text generation, & more.

- Who should read this book. This book is written for people with Python programming experience who want to get started with ML & DL. But this book can also be valuable to many different types of readers:
  - If a data scientist familiar with ML, this book will provide with a solid, practical introduction to DL, fastest-growing & most significant subfield of ML.
  - If a DL researcher or practitioner looking to get started with Keras framework, find this book to be ideal Keras crash course.
  - If a graduate student studying DL in a formal setting, find this book to be a practical complement to your education, helping build intuition around behavior of deep neural networks & familiarizing you with key best practices.

  Even technically minded people who don't code regularly will find this book useful as an introduction to both basic & advanced DL concepts.

  In order to understand code examples, need reasonable Python proficiency. Additionally, familiarity with NumPy library will be helpful, although it isn't required. Don't need previous experience with ML or DL: this book covers, from scratch, all necessary basics. Don't need an advanced mathematics background, either – high school-level mathematics should suffice in order to follow along.
- About code. This book contains many examples of source code both in numbered listings, & in line with normal text. In both cases, source code is formatted in a `fixed-width font` to separate it from ordinary text.

  In many cases, original source code has been reformatted, added line breaks & reworked indentation to accommodate available page space in book. Additionally, comments in source code have often been removed from listings when code is described in text. Code annotations accompany many of listings, highlighting important concepts.

  Jupyter notebooks on GitHub at https://github.com/fchollet/deep-learning-with-python-notebooks can be run directly in browser via Google Colaboratory, a hosted Jupyter notebook environment that you can use for free. An internet connection & a desktop web browser are all you need to get started with DL.
- About Author. Francois Chollet is creator of Keras, 1 of most widely used DL frameworks. Currently a software engineer at Google, where leads Keras team. In addition, he does research on abstraction, reasoning, & how to achieve greater generality in AI.

- 1. What is DL? Cover: High-level definitions of fundamental concepts. Timeline of development of ML. Key factors behind DL's rising popularity & future potential.

  In past few years, AI has been a subject of intense media hype. ML, DL, & AI come up in countless articles, often outside of technology-minded publications. Promised a future of intelligent chatbots, self-driving cars, & virtual assistants – a future sometimes painted in a grim light & other times as utopian, where human jobs will be scarce & most economic activity will be handled by robots or AI agents. For a future or current practitioner of ML, important to be able to recognize signal amid noise, so that you can tell world-changing developments from overhyped press releases. Our future is at stake, & it's a future in which you have an active role to play: after reading this book, will be 1 of those who develop those AI systems. So tackle these questions: What has DL achieved so far? How significant is it? Where are we headed next? Should you believe hype?

  This chap provides essential context around AI, ML, & DL.

  - 1.1. AI, ML, & DL. 1st, need to define clearly what talking about when mention AI. What are AI, ML, & DL. How do they relate to each other?
    - 1.1.1. AI. AI was born in 1950s, when a handful of pioneers from nascent field of CS started asking whether computers could be made to "think" – a question whose ramifications still exploring today.
      While many of underlying ideas had been brewing in years & even decades prior, "AI" finally crystallized as a field of research in 1956, when John McCarthy, then a young Assistant Professor of Mathematics at Dartmouth College, organized a summer workshop under the following proposal:

      > Study: proceed on basis of conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it. An attempt will be made to find how to make machines use language, form abstractions & concepts, solve kinds of problems now reserved for humans, & improve themselves. Think: a significant advance can be made in 1 or more of these problems if a carefully selected group of scientists work on it together for a summer.

      At end of summer, workshop concluded without having fully solved riddle it set out to investigate. Nevertheless, it was attended by many people who would move on to become pioneers in field, & it set in motion an intellectual revolution that is still ongoing to this day.
      Concisely, AI can be described as *effort to automate intellectual tasks normally performed by humans*. As such, AI is a general field that encompasses ML & DL, but that also includes many more approaches that may not involve any

learning. Consider that until 1980s, most AI textbooks didn't mention "learning" at all! Early chess programs, e.g., only involved hardcoded rules crafted by programmers, & didn't qualify as ML. In fact, for a fairly long time, most experts believed: human-level AI could be achieved by having programmers handcraft a sufficiently large set of explicit rules for manipulating knowledge stored in explicit databases. This approach is known as *symbolic AI*. It was dominant paradigm in AI from 1950s to late 1980s, & it reached its peak popularity during *expert systems* boom o 1980s.

Although symbolic AI proved suitable to solve well-defined, logical problems, e.g. playing chess, it turned out to be intractable to figure out explicit rules for solving more complex, fuzzy problems, e.g. image classification, speech recognition, or natural language translation. A new approach arose to take symbolic AI's place: *ML*.

∗ 1.1.2. ML. In Victorian England, Lady Ada Lovelace was a friend & collaborator of Charles Babbage, inventor of *Analytical Engine*: 1st-known general-purpose mechanical computer. Although visionary & far ahead of its time, Analytical Engine wasn't meant as a general-purpose computer when it was designed in 1830s & 1840s, because concept of general-purpose computation was yet to be invented. It was merely meant as a way to use mechanical operations to automate certain computations from field of mathematical analysis – hence name Analytical Engine. As such, it was intellectual descendant of earlier attempts at encoding mathematical operations in gear form, e.g. Pascaline, or Leibniz's step reckoner, a refined version of Pascaline. Designed by Blaise Pascal in 1642 (at age 19), Pascaline was world's 1st mechanical calculator – it could add, subtract, multiply, or even divide digits.

In 1843, Ada Lovelace remarked on invention of Analytical Engine,

> Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform … Its province is to assist us in making available what we've already acquainted with.

Even with 178 years of historical perspective, Lady Lovelace's observation remains arresting. Could a general-purpose computer "originate" anything, or would it always be bound to dully execute processes we humans fully understand? Could it ever be capable of any original thought? Could it learn from experience? Could it show creativity?

Her remark was later quoted by AI pioneer Alan Turing as "Lady Lovelace's objection" in his landmark 1950 paper "Computing Machinery & Intelligence," which introduced *Turing test* as well as key concepts that would come to shape AI.[6] Turing was of opinion – highly provocative at time – computers could in principle be made to emulate all aspects of human intelligence.

Usual way to make a computer do useful work: have a human programmer write down *rules* – a computer program – to be followed to turn input data into appropriate answers, just like Lady Lovelace writing down step-by-step instructions for Analytical Engine to perform. ML turns this around: machine looks at input data & corresponding answers, & figures out what rules should be (Fig. 1.2: ML: a new programming paradigm.). A ML system is *trained* rather than explicitly programmed. It's presented with many examples relevant to a task, & it finds statistical structure in these examples that eventually allows system to come up with rules for automating task. E.g., if wished to automate task of tagging vacation pictures, could present a ML system with many examples of pictures already tagged by humans, & system would learn statistical rules for associating specific pictures to specific tags.

Although ML only started to flourish in 1990s, it has quickly become most popular & most successful subfield of AI, a trend driven by availability of faster hardware & larger datasets. ML is related to mathematical statistics, but it differs from statistics in several important ways, in same sense: medicine is related to chemistry but cannot be reduced to chemistry, as medicine deals with its own distinct systems with their own distinct properties. Unlike statistics, ML tends to deal with large, complex datasets (e.g. a dataset of millions of images, each consisting of tens of thousands of pixels) for which classical statistical analysis e.g. Bayesian analysis would be impractical. As a result, ML, & especially DL, exhibits comparatively little mathematical theory – maybe too little – & is fundamentally an engineering discipline. Unlike theoretical physics or mathematics, ML is a very hands-on field driven by empirical findings & deeply reliant on advances in software & hardware.

∗ 1.1.3. Learning rules & representations from data. To define *DL* & understand difference between DL & other ML approaches, 1st need some idea of what ML algorithms do. Just stated: ML discovers rules for executing a data processing task, given examples of what's expected. So, to do ML, need 3 things:

- *Input data points* – e.g., if task is speech recognition, these data points could be sound files of people speaking. If task is image tagging, they could be pictures.
- *Examples of expected output* – In a speech-recognition task, these could be human-generated transcripts of sound files. In an image task, expected outputs could be tags e.g. "dog", "cat", & so on.
- *A way to measure whether algorithm is doing a good job* – This is necessary in order to determine distance between algorithm's current output & its expected output. Measurement is used as a feedback signal to adjust way algorithm works. This adjustment step is what we call *learning*.

A ML model transforms its input data into meaningful outputs, a process that is "learned" from exposure to known examples of inputs & outputs. Therefore, central problem in ML & DL: *meaningfully transform data*: i.e., to learn useful *representations* of input data at hand – representations that get us closer to expected output.

Before go any further: what's a representation? At its core, it's a different way to look at data – to represent or encode data. E.g., a color image can be encoded in RGB format (red-green-blue) or in HSV format (hue-saturation-value): these are 2 different representations of same data. Some tasks that may be difficult with 1 representation can become easy with another. E.g., task "select all red pixels in image" is simpler in RGB format, whereas "make image less saturated" is simpler

[6] Although Turing test has sometimes been interpreted as a literal test – a goal of field of AI should set out to reach – Turing merely meant it as a conceptual device in a philosophical discussion about nature of cognition.

in HSV format. ML models are all abound finding appropriate representations for their input data – transformations of data that make it more amenable to task at hand.

Make its concrete. Consider an x-axis, a y-axis, & some points represented by their coordinates in (x, y) system, as shown in Fig. 1.3: Some sample data.

As can see, have a few white points & a few black points. Say want to develop an algorithm that can take coordinates (x, y) of a point & output whether that point is likely to be black or to be white. In this case,

- Inputs are coordinates of our points.
- Expected outputs are colors of our points.
- A way to measure whether our algorithm is doing a good job could be, e.g., percentage of points that are being correctly classified.

What need here is a new representation of our data that cleanly separates white points from black points. 1 transformation could use, among many other possibilities, would be a coordinate change, illustrated in Fig. 1.4: Coordinate change.

In this new coordinate system, coordinates of our points can be said to be a new representation of our data. & it's a good one! With this representation, black/white classification problem can be expressed as a simple rule: "Black points are s.t. $x > 0$," or "White points are s.t. $x < 0$." This new representation, combined with this simple rule, neatly solves classification problem.

In this case defined coordinate change by hand: used our human intelligence to come up with our own appropriate representation of data. This is fine for such an extremely simple problem, but could do same if task were to classify images of handwritten digits? Could you write down explicit, computer-executable image transformations that would illuminate difference between a 6 & an 8, between a 1 & a 7, across all kinds of different handwriting?

This is possible to an extent. Rules based on representations of digits e.g. "number of closed loops" or vertical & horizontal pixel histograms can do a decent job of telling apart handwritten digits. But finding such useful representations by hand is hard work, &, as can imagine, resulting rule-based system is brittle (giòn, mỏng manh) – a nightmare to maintain. Every time you come across a new example of handwriting that breaks your carefully thought-out rules, you will have to add new data transformations & new rules, while taking into account their interaction with every previous rule.

Probably think: if this process is so painful, could automate it? What if tried systematically searching for different sets of automatically generated representations of data & rules based on them, identifying good ones by using a feedback percentage of digits being correctly classified in some development dataset? Would then be doing ML. *Learning*, in context of ML, describes an automatic search process for data transformations that produce useful representations of some data, guided by some feedback signal – representations that are amentable to simpler rules solving task at hand.

These transformations can be coordinate changes (like in 2D coordinates classification example), or taking a histogram of pixels & counting loops (like in digits classification example), but they could also be linear projections, translations, nonlinear operations (e.g. "select all points s.t. $x > 0$"), & so on. ML algorithms aren't usually creative in finding these transformations; they're merely searching through a predefined a set of operations, called a *hypothesis space*. E.g., space of all possible coordinate changes would be our hypothesis space in 2D coordinates classification example.

So that's what ML is, concisely: searching for useful representations & rules over some input data, within a predefined space of possibilities, using guidance from a feedback signal. This simple idea allows for solving a remarkably broad range of intellectual tasks, from speech recognition to autonomous driving.

Now understand what mean by *learning*, take a look at what makes *DL* special.

∗ 1.1.4. "Deep" in "DL". DL is a specific subfield of ML: a new take on learning representations from data that puts an emphasis on learning successive layers of increasingly meaningful representations. "Deep" in "DL" isn't a reference to any kind of deeper understanding achieved by approach; rather, it stands for this idea of successive layers of representations. How many layers contribute to a model of data is called *depth* of model. Other appropriate names for field could have been *layered representations learning* or *hierarchical representations learning*. Modern DL often involves 10s or even hundreds of successive layers of representations, & they're all learned automatically from exposure to training data. Meanwhile, other approaches to ML tend to focus on learning only 1 or 2 layers of representations of data (say, taking a pixel histogram & then applying a classification rule); hence, they're sometimes called *shallow learning*.

In DL, these layered representations are learned via models called *neural networks*, structured in literal layers stacked on top of each other. Term "neural network" refers to neurobiology, but although some of central concepts in DL were developed in part by drawing inspiration from our understanding of brain (in particular, visual cortex), DL models are not models of brain. There's no evidence that brain implements anything like learning mechanisms used in modern DL models. May come across pop-science articles proclaiming: DL works like brain or was modeled after brain, but that isn't case. It would be confusing & counterproductive for newcomers to field to think of DL as being in any way related to neurobiology; don't need that shroud of "just like our minds" mystique & mystery, & you may as well forget anything you may have read about hypothetical links between DL & biology. For our purposes, DL is a mathematical framework for learning representations from data.

What do representations learned by a DL algorithm look like? Examine how a network several layers deep (Fig. 1.5: A deep neural network for digit classification.) transforms an image of a digit in order to recognize what digit it is.

As can see in Fig. 1.6: Data representations learned by a digit-classification model., network transforms digit image into representations that are increasingly different from original image & increasingly informative about final result. Can think of a deep network as a multistage *information-distillation* process, where information goes through successive filters & comes out increasingly *purified* (i.e., useful with regard to some task).

So that's what DL is, technically: a multistage way to learn data representations. It's a simple idea – but, as it turns

out, very simple mechanisms, sufficiently scaled, can end up looking like magic.

∗ 1.1.5. Understanding how DL works, in 3 figures. At this point, know: ML is about mapping inputs (e.g. images) to targets (e.g. label "cat"), which is done by observing many examples of input & targets. Also know: deep neural networks do this input-to-target mapping via a deep sequence of simple data transformations (layers) & these data transformations are learned by exposure to examples. Look at how this learning happens, concretely.

Specification of what a layer does to its input data is stored in layer's *weights*, which in essence are a bunch of numbers. In technical terms, say: transformation implemented by a layer is *parameterized* by its weights (Fig. 1.7: A neural network is parameterized by its weights. Goal: find right values for these weights.). (Weights are also sometimes called *parameters* of a layer.) In this context, *learning* means finding a set of values for weights of all layers in a network, s.t. network will correctly map example inputs to their associated targets. But here's the thing: a deep neural network can contain 10s of millions of parameters. Finding correct values for all of them may seem like a daunting task, especially given that modifying value of 1 parameter will affect behavior of all the others!

To control sth, 1s need to be able to observe it. To control output of a neural network, need to be able to measure how far this output is from what you expected. This is job of *loss function* of network, also sometimes called *objective function* or *cost function*. Loss function takes predictions of network & true target (what you wanted network to output) & computes a distance score, capturing how well network has done on this specific example (Fig. 1.8: A loss function measures quality of network's output.).

Fundamental trick in DL: use this score as a feedback signal to adjust value of weights a little, in a direction that will lower loss score for current example (Fig. 1.9: Loss score to used as a feedback signal to adjust weights.). This adjustment is job of *optimizer*, which implements what's called *Backpropagation* algorithm: central algorithm in DL. Next chap explains in more detail how backpropagation works.

Initially, weights of network are assigned random values, so network merely implements a series of random transformations. Naturally, its output is far from what it should ideally be, & loss score is accordingly very high. But with every example network processes, weights are adjusted a little in correct direction, & loss score decreases. This is *training loop*, which, repeated a sufficient number of times (typically 10s of iterations over thousands of examples), yields weight values that minimize loss function. A network with a minimal loss is 1 for which outputs are as close as they can be to targets: a trained network. Once again, it's a simple mechanism that, once scaled, ends up looking like magic.

∗ 1.1.6. What DL has achieved so far. Although DL is a fairly old subfield of ML, it only rose to prominence in early 2010s. In few years since, it has achieved nothing short of a revolution in field, producing remarkable results on perceptual tasks & even natural language processing tasks – problems involving skills that seem natural & intuitive to humans but have long been elusive for machines.

In particular, DL has enabled following breakthroughs, all in historically difficult areas of ML:

· Near-human-level image classification
· Near-human-level speech transcription
· Near-human-level handwriting transcription
· Dramatically improved machine translation
· Dramatically improved text-to-speech conversion
· Digital assistants e.g. Google Assistant & Amazon Alexa
· Near-human-level autonomous driving
· Improved ad targeting, as used by Google, Baidu, or Bing
· Improved search results on web
· Ability to answer natural language questions
· Superhuman Go playing

Still exploring full extent of what DL can do. Have started applying it with great success to a wide variety of problems that were thought to be impossible to solve just a few years ago – automatically transcribing 10s of thousands of ancient manuscripts held in Vatican's Apostolic Archive, detecting & classifying plant diseases in fields using a simple smartphone, assisting oncologists or radiologists with interpreting medical imaging data, predicting natural disasters e.g. floods, hurricanes, or even earthquakes, & so on. With every milestone, getting closer to an age where DL assists us in every activity & every field of human endeavor – science, medicine, manufacturing, energy, transportation, software development, agriculture, & even artistic creation.

∗ 1.1.7. Don't believe short-term hype. Although DL has led to remarkable achievements in recent years, expectations for what field will be able to achieve in next decade tend to run much higher than what will likely be possible. Although some world-changing applications like autonomous cars are already within reach, many more are likely to remain elusive for a long time, e.g. believable dialogue systems, human-level machine translation across arbitrary languages, & human-level natural language understanding. In particular, talk of human-level general intelligence shouldn't be taken too seriously. Risk with high expectations for short term: as technology fails to deliver, research investment will dry up, slowing progress for a long time.

This has happened before. Twice in past, AI went through a cycle of intense optimism followed by disappointment & skepticism, with a dearth (sự khan hiếm) of funding as a result. It started with symbolic AI in 1960s. In those early days, projections about AI were flying high. 1 of best-known pioneers & proponents of symbolic AI approach was Marvin Minsky, who claimed in 1967, "Within a generation ... problem of creating 'artificial intelligence' will substantially be

solved." 3 years later, in 1970, he made a more precisely quantified prediction: "In from 3–8 years will have a machine with general intelligence of an average human being." In 2021 such an achievement will appears to be far in future – so far that have no way to predict how long it will take – but in 1960s & early 1970s, several experts believed it to be right around corner (as do many people today). A few years later, as these high expectations failed to materialize, researchers & government funds turned away from field, marking start of 1st *AI winter* (a reference to a nuclear winter, because this was shortly after height of Cold War).

It wouldn't be last one. In 1980s, a new take on symbolic AI, *expert systems*,, started gathering steam among large companies. A few initial success stories triggered a wave of investment, with corporations around world starting their own in-house AI departments to develop expert systems. Around 1985, companies were spending over $1 billion each year on technology; but by early 1990s, these systems had proven expensive to maintain, difficult to scale, & limited in scope, & interest died down. Thus began 2nd AI winter.

May be currently witnessing 3rd cycle of AI hype & disappointment, & still in phase of intense optimism. Best to moderate our expectations for short term & make sure people less familiar with technical side of field have a clear idea of what DL can & can't deliver.

* 1.1.8. Promise of AI. Although may have unrealistic short-term expectations for AI, long-term picture is looking bright. Only getting started in applying DL to many important problems for which it could prove transformative, from medical diagnoses to digital assistants. AI research has been moving forward amazingly quickly in past 10 years, in large part due to a level of funding never before seen in short history of AI, but so far relatively little of this progress has made its way into products & processes that form our world. Most of research findings of DL aren't yet applied, or at least are not applied to full range of problems they could solve across all industries. Your doctor doesn't yet use AI, & neither does your accountant. Probably don't use AI technologies very often in day-to-day life. Of course, can ask smartphone simple questions & get reasonable answers, can get fairly useful product recommendations on Amazon.com, & can search for "birthday" on Google Photos & instantly find those pictures of daughter's birthday party from last month. That's a far cry from where such technologies used to stand. But such tools are still only accessories to our daily lives. AI has yet to transition to being central to way we work, think, & live.

  Right now, it may seem hard to believe: AI could have a large impact on world, because it isn't yet widely deployed (triển khai rộng rãi) – much as, back in 1995, it would have been difficult to believe in future impact of internet. Back then, most people didn't see how internet was relevant to them & how it was going to change their lives. Same is true for DL & AI today. But make no mistake: AI is coming. In a not-so-distant future, AI will be your assistant, even your friend; it will answer questions, help educate your kids, & watch over your health. It will deliver your groceries to door & drive you from point A to point B. It will be your interface to an increasingly complex & information-intensive world. &, even more important, AI will help humanity as a whole move forward, by assisting human scientists in new breakthrough discoveries across all scientific fields, from genomics to mathematics.

  On way, may face a few setbacks & maybe even a new AI winter – in much same way internet industry was overhyped in 1998–99 & suffered from a crash that dried up investment throughout early 2000s. But get there eventually. AI will end up being applied to nearly every process that makes up society & daily lives, much like internet is today.

  Don't believe short-term hype, but do believe in long-term vision. It may take a while for AI to be deployed to its true potential – a potential full extent of which no one has yet dared to dream – but AI is coming, & it will transform world in a fantastic way.

◦ 1.2. Before DL: A brief history of ML. DL has reached a level of public attention & industry investment never before seen in history of AI, but it isn't 1st successful form of ML. Safe to say: most of ML algorithms used in industry today aren't DL algorithms. DL isn't always right tool for job – sometimes there isn't enough data for DL to be applicable, & sometimes problem is better solved by a different algorithm. if DL is your 1st contact with ML, may find yourself in a situation where all you have is DL hammer, & every ML problem starts to look like a nail. Only way not to fall into this trap: be familiar with other approaches & practice them when appropriate.

  A detailed discussion of classical ML approaches is outside of scope of this book, but briefly go over them & describe historical context in which they were developed. This will allow us to place DL in broader context of ML & better understand where DL comes from & why it matters.

  * 1.2.1. Probabilistic modeling. *Probabilistic modeling* is application of principles of statistics to data analysis. It is 1 of earliest forms of ML, & still widely used to this day. 1 of best-known algorithms in this category: Naive Bayes algorithm. Naive Bayes is a type of ML classifier based on applying Bayes' theorem while assuming: features in input data are all independent (a strong, or "naive" assumption, which is where name comes from). This form of data analysis predates computers & was applied by hand decades before its 1st computer implementation (most likely dating back to 1950s). Bayes' theorem & foundations of statistics date back to 18th century, & these are all you need to start using Naive Bayes classifiers.

    A closely related model is *logistic regression* (logreg for short), which is sometimes considered to be "Hello World" of modern ML. Don't be misled by its name – logreg is a classification algorithm rather than a regression algorithm. Much like Naive Bayes, logreg predates computing by a long time, yet it's still useful to this day, thanks to its simple & versatile nature (bản chất đa dạng). It's often 1st thing a data scientist will try on a dataset to get a feel for classification task at hand.

  * 1.2.2. Early neural networks. Early iterations of neural networks have been completely supplanted by modern variants covered in these pages, but it's helpful to be aware of how DL originated. Although core ideas of neural networks were investigated in toy forms as early as 1950s, approach took decades to get started. For a long time, missing piece was an

efficient way to train large neural networks. This changed in mid-1980s, when multiple people independently rediscovered Backpropagation algorithm – a way to train chains of parametric operations using gradient-descent optimization (precisely define these concepts later) – & started applying it to neural networks.

1st successful practical application of neural nets came in 1989 from Bell Labs, when Yann LeCun combined earlier ideas of convolutional neural networks & backpropagation, & applied them to problem of classifying handwritten digits. Resulting network, dubbed *LeNet*, was used by United States Postal Service in 1990s to automate reading of ZIP codes on mail envelopes.

* 1.2.3. Kernel methods. As neural networks started to gain some respect among researchers in 1990s, thanks to this 1st success, a new approach to ML rose to fame & quickly sent neural nets back to oblivion: kernel methods. *Kernel methods* are a group of classification algorithms, best known of which is *Support Vector Machine* (SVM). Modern formulation of an SVM was developed by Vladimir Vapnik & Corinna Cortes in early 1990s at Bell Labs & published in 1995 [Vladimir Vapnik & Corinna Cortes, "Support-Vector Networks," Machine Learning 20, no. 3 (1995): 273–297.] although an older linear formulation was published by Vapnik & Alexey Chervonenkis as early as 1963. [Vladimir Vapnik & Alexey Chervonenkis, "A Note on 1 Class of Perceptrons," Automation and Remote Control 25 (1964).]

SVMis a classification algorithm that works by finding "decision boundaries" separating 2 classes (Fig. 1.10: A decision boundary). SVMs proceed to find these boundaries in 2 steps:

1. Data is mapped to a new high-dimensional representation where decision boundary can be expressed as a hyperplane (if data was 2D, as in Fig. 1.10, a hyperplane would be a straight line).
2. A good decision boundary (a separation hyperplane) is computed by trying to maximize distance between hyperplane & closest data points from each class, a step called *maximizing margin*. This allows boundary to generalize well to new samples outside of training dataset.

Technique of mapping data to a high-dimensional representation where a classification problem becomes simpler may look good on paper, but in practice it's often computationally intractable. That's where *kernel trick* comes in (key idea that kernel methods are named after). Here's gist (ý chính) of it: to find good decision hyperplanes in new representation space, don't have to explicitly compute coordinates of your points in new space; just need to compute distance between pairs of points in that space, which can be done efficiently using a kernel function. A *kernel function* is a computationally tractable operation that maps any 2 points in your initial space to distance between these points in your target representation space, completely bypassing explicit computation of new representation. Kernel functions are typically crafted by hand rather than learned from data – in case of an SVM, only separation hyperplane is learned.

At time they were developed, SVMs exhibited state-of-art performance on simple classification problems & were 1 of few ML methods backed by extensive theory & amenable to serious mathematical analysis, making them well understood & easily interpretable. Because of these useful properties, SVMs became extremely popular in field for a long time.

– Vào thời điểm chúng được phát triển, SVM đã thể hiện hiệu suất tiên tiến trên các vấn đề phân loại đơn giản & là 1 trong số ít phương pháp ML được hỗ trợ bởi lý thuyết mở rộng & có thể thực hiện phân tích toán học nghiêm túc, khiến chúng được hiểu rõ & dễ diễn giải. Nhờ những đặc tính hữu ích này, SVM đã trở nên cực kỳ phổ biến trong lĩnh vực này trong một thời gian dài.

But SVMs proved hard to scale to large datasets & didn't provide good results for perceptual problems e.g. image classification. Because an SVM is a shallow method, applying an SVM to perceptual problems requires 1st extracting useful representations manually (a step called *feature engineering*), which is difficult & brittle. E.g., if want to use an SVM to classify handwritten digits, can't start from raw pixels; should 1st find by hand useful representations that make problem more tractable, like pixel histograms.

* 1.2.4. Decision trees, random forests, & gradient boosting machines. *Decision trees* are flowchart-like structures that let you classify input data points or predict output values given inputs (Fig. 1.11: A decision tree: parameters that are learned are questions about data. A question could be, e.g., "Is coefficient 2 in data > 3.5?). They are easy to visualize & interpret (hình dung & diễn giải). Decision trees learned from data began to receive significant research interest in 2000s, & by 2010 they were often preferred to kernel methods.

In particular, *Random Forest* algorithm introduced a robust, practical take on decision-tree learning that involves building a large number of specialized decision trees & then ensembling their outputs. Random forests are applicable to a wide range of problems – could say: they are almost always 2nd-best algorithm for any shallow ML task. When popular ML competition website Kaggle http://kaggle.com got started in 2010, random forests quickly became a favorite on platform – until 2014, when *gradient boosting machines* took over. A gradient boosting machine, much like a random forest, is a ML technique based on ensembling weak prediction models, generally decision trees. It uses *gradient boosting*, a way to improve any ML model by iteratively training new models that specialize in addressing weak points of previous models. Applied to decision trees, use of gradient boosting technique results in models that strictly outperform random forests most of time, while having similar properties. It may be 1 of best, if not *the* best, algorithm for dealing with nonperceptual data today. Alongside DL, it's 1 of most commonly used techniques in Kaggle competitions.

* 1.2.5. Back to neural networks. Around 2010, although neural networks were almost completely shunned by scientific community at large, a number of people still working on neural networks started to make important breakthroughs: group of Geoffrey Hinton at University of Toronto, Yoshua Bengio at the University of Montreal, Yann LeCun at New York University, & IDSIA in Switzerland.

In 2011, Dan Ciresan from IDSIA began to win academic image-classification competitions with GPU-trained deep neural networks – 1st practical success of modern DL. But watershed moment came in 2012, with entry of Hilton's group in yearly large-scale image-classification challenge ImageNet (ImageNet Large Scale Visual Recognition Challenge,

or ILSVRC for short). ImageNet challenge was notoriously difficult at time consisting of classifying high-resolution color images into 1000 different categories after training on 1.4 million images. In 2011, top-5 accuracy of winning model, based on classical approaches to computer vision, was only 74.3%.[7] Then, in 2012, a team led by Alex Krizhevsky & advised by Geoffrey Hinton was able to achieve a top-5 accuracy of 83.6% – a significant breakthrough. Competition has been dominated by deep convolutional neural networks every year since. By 2015, winner reached an accuracy of 96.4%, & classification task on ImageNet was considered to be a completely solved problem.

Since 2012, deep convolutional neural networks (*convnets*) have become go-to algorithm ∀ computer vision tasks; more generally, they work on all perceptual tasks. At any major computer vision conference after 2015, it was nearly impossible to find presentations that didn't involve convnets in some form. At same time, DL has also found applications in many other types of problems, e.g. natural language processing. It has completely replaced SVMs & decision trees in a wide range of applications. e.g., for several years, European Organization for Nuclear Research, CERN, used decision tree-based methods for analyzing particle data from ATLAS detector at Large Hadron Collider (LHC), but CERN eventually switched to Keras-based deep neural networks due to their highest performance & east of training on large datasets.

∗ 1.2.6. What makes DL different. Primary reason DL took off so quickly: it offered better performance for many problems. But that's not only reason. DL also makes problem-solving much easier, because it completely automates what used to be most crucial step in a ML workflow: feature engineering.

Previous ML techniques – shallow learning – only involved transforming input data into 1 or 2 successive representation spaces, usually via simple transformations e.g. high-dimensional nonlinear projections (SVMs) or decision trees. But refined representations required by complex problems generally can't be attained by such techniques. As such, humans had to go to great lengths to make initial input data more amenable to processing by these methods: they had to manually engineer good layers of representations for their data. This is called *feature engineering*. DL, on other hand, completely automates this step: with DL, learn all features in 1 pass rather than having to engineer them yourself. This has greatly simplified ML workflows, often replacing sophisticated multistage pipelines with a single, simple, end-to-end DL model.

May ask, if crux of issue is to have multiple successive layers of representations, could shallow methods be applied repeatedly to emulate effects of DL? In practice, successive applications of shallow-learning methods produce fast-diminishing returns, because optimal 1st representation layer in a 3-layer model isn't optimal 1st layer in a 1-layer or 2-layer model. What is transformative about DL: DL allows a model to learn all layers of representation *jointly*, at same time, rather than in succession (*greedily*, as it's called). With joint feature learning, whenever model adjusts 1 of its internal features, all other features that depend on it automatically adapt to change, without requiring human intervention. Everything is supervised by a single feedback signal: every change in model serves end goal. This is much more powerful than greedily stacking shallow models, because it allows for complex, abstract representations to be learned by breaking them down into long series of intermediate spaces (layers); each space is only a simple transformation away from previous one.

– Có thể hỏi, nếu cốt lõi của vấn đề là có nhiều lớp biểu diễn liên tiếp, thì các phương pháp học nông có thể được áp dụng nhiều lần để mô phỏng các hiệu ứng của DL không? Trong thực tế, các ứng dụng liên tiếp của các phương pháp học nông tạo ra lợi nhuận giảm dần nhanh chóng, vì lớp biểu diễn thứ nhất tối ưu trong mô hình 3 lớp không phải là lớp thứ nhất tối ưu trong mô hình 1 lớp hoặc 2 lớp. DL có đặc điểm gì là biến đổi: DL cho phép một mô hình học tất cả các lớp biểu diễn *kết hợp*, cùng một lúc, thay vì tuần tự (*tham lam*, như cách gọi của nó). Với việc học tính năng kết hợp, bất cứ khi nào mô hình điều chỉnh 1 trong các tính năng bên trong của nó, tất cả các tính năng khác phụ thuộc vào nó sẽ tự động thích ứng với sự thay đổi, mà không cần sự can thiệp của con người. Mọi thứ đều được giám sát bởi một tín hiệu phản hồi duy nhất: mọi thay đổi trong mô hình đều phục vụ cho mục tiêu cuối cùng. Điều này mạnh hơn nhiều so với việc xếp chồng các mô hình nông một cách tham lam, vì nó cho phép học các biểu diễn trừu tượng, phức tạp bằng cách chia chúng thành một chuỗi dài các không gian trung gian (lớp); mỗi không gian chỉ là một phép biến đổi đơn giản so với không gian trước đó.

These are 2 essential characteristics of how DL learns from data: *incremental, layer-by-layer way in which increasingly complex representations are developed*, & fact: *these intermediate incremental representations are learned jointly*, each layer being updated to follow both representational needs of layer above & needs of layer below. Together, these 2 properties have made DL vastly more successful than previous approaches to ML.

∗ 1.2.7. Modern ML landscape. A great way to get a sense of current landscape ML algorithms & tools: look at ML competitions on Kaggle. Due to its highly competitive environment (some contests have thousands of entrants & million-dollar prizes) & to wide variety of ML problems covered, Kaggle offers a realistic way to assess what works & what doesn't. So what kind of algorithm is reliably winning competitions? What tools do top entrants use?

In early 2019, Kaggle ran a survey asking teams: ended in top 5 of any competition since 2017 which primary software tool they had used in competition (Fig. 1.12: ML tools used by top teams on Kaggle.). Turn out: top teams tend to use either DL methods (most often via Keras library) or gradient boosted trees (most often via LightGBM or XGBoost libraries).

It's not just competition champions, either. Kaggle also runs a yearly survey among ML & DS professionals worldwide. With 10s of thousands of respondents, this survey is 1 of our most reliable sources about state of industry. Fig. 1.13: Tool usage across ML & DS industry www.kaggle.com/kaggle-survey-2020 shows percentage of usage of different ML software frameworks.

From 2016–2020, entire ML & DS industry has been dominated by these 2 approaches: DL & gradient boosted trees. Specifically, gradient boosted trees is used for problems where structured data is available, whereas DL is used for perceptual problems e.g. image classification.

Users of gradient boosted trees tend to use Scikit-learn, XGBoost, or LightGBM. Meanwhile, most practitioners of DL

[7] "Top-5 accuracy" measures how often model selects correct answer as part of its top 5 guesses (out of 1000 possible answers, in case of ImageNet).

use Keras, often in combination with its parent framework TensorFlow. Common point of these tools is they're all Python libraries: Python is by far most widely used language for ML & DS>

These are 2 techniques you should be most familiar with in order to be successful in applied ML today: gradient boosted trees, for shallow-learning problems; & DL, for perceptual problems. In technical terms, this mean: need to be familiar withScikit-learn, XGBoost, & Keras – 3 libraries that currently dominate Kaggle competitions. With this book in hand, already 1 big step closer.

- 1.3. Why DL? Why now? 2 key ideas of DL for computer vision – convolutional neural networks & backpropagation – were already well understood by 1990. Long Short-Term Memory (LSTM) algorithm, which is fundamental to DL for timeseries, was developed in 1997 & has barely changed since. So why did DL only take off after 2012? What changed in these 2 decades?

  In general, 3 technical forces are driving advances in ML:

  * Hardware
  * Datasets & benchmarks
  * Algorithmic advances

  Because field is guided by experimental findings rather than by theory, algorithmic advances only become possible when appropriate data & hardware are available to try new ideas (or to scale up old ideas, as is often case). ML isn't mathematics or physics, where major advances can be done with a pen & a piece of paper. It's an engineering science.

  Real bottlenecks throughout 1990s & 2000s were data & hardware. But here's what happened during that time: internet took off & high-performance graphics chips were developed for needs of gaming market.

  * 1.3.1. Hardware. Between 1990 & 2010, off-shelf CPUs became faster by a factor of approximately 5000. As a result, nowadays it's possible to run small DL models on laptop, whereas this would have been intractable 25 years ago.

    But typical DL models used in computer vision or speech recognition require orders of magnitude more computational power than your laptop can deliver. Throughout 2000s, companies like NVIDIA & AMD invested billions of dollars in developing fast, massively parallel chips (graphical processing units, or GPUs) to power graphics of increasingly photorealistic video games – cheap, single-purpose supercomputers designed to render complex 3D scenes on your screen in real time. This investment came to benefit scientific community when, in 2007, NVIDIA launched CUDA https://developer.nvidia.com/about-cuda, a programming interface for its line of GPUs. A small number of GPUs started replacing massive clusters of CPUs in various highly parallelizable applications, beginning with physics modeling. Deep neural networks, consisting mostly of many small matrix multiplications, are also highly parallelizable, & around 2011 some researchers began to write CUDA implementations of neural nets – Dan Ciresan & Alex Krizhevsky were among 1st: ["Flexible, High Performance Convolutional Neural Networks for Image Classification," Proceedings of the 22nd International Joint Conference on Artificial Intelligence (2011), www.ijcai.org/Proceedings/11/Papers/210.pdf.] ["ImageNet Classification with Deep Convolutional Neural Networks," Advances in Neural Information Processing Systems 25 (2012), http://mng.bz/2286.]

    What happened: gaming market subsidized (được trợ cấp) supercomputing for next generation of AI applications. Sometimes, big things begin as games. Today, NVIDIA Titan RTX, a GPU that cost $2,500 at end of 2019, can deliver a peak of 16 teraFLOPS in single precision (16 trillion `float32` operations per sec). That's about 500 times more computing power than world's fastest supercomputer from 1990s, Intel Touchstone Delta. On a Titan RTX, it takes only a few hours to train an ImageNet model of sort that would have won ILSVRC competition around 2012 or 2013. Meanwhile, large companies train DL models on clusters of hundreds of GPUs.

    What's more, DL industry has been moving beyond GPUs & is investing in increasingly specialized, efficient chips for DL. In 2016, at its annual I/O convention, Google revealed its Tensor Processing Unit (TPU) project: a new chip design developed from ground up to run deep neural networks significantly faster & far more energy efficient than top-of-line GPUs. Today, in 2020, 3rd iteration of TPU card represent 420 teraFLOPs of computing power. That's 10000 times more than Intel Touchstone Delta from 1990.

    These TPU cards are designed to be assembled into large-scale configurations, called "pods". 1 pod (1024 TPU cards) peaks at 100 petaFLOPs. For scale, that's about 10% of peak computing power of current largest supercomputer, IBM Summit at Oak Ridge National Lab, which consists of 27000 NVIDIA GPUs & peaks at around 1.1 exaFLOPS.

  * 1.3.2. Data. AI is sometimes heralded as new industrial revolution. If DL is steam engine of this revolution, then data is its coal: raw material that powers intelligent machines, without which nothing would be possible. When it comes to data, in addition to exponential progress in storage hardware over past 20 years (following Moore's law), game changer has been rise of internet, making it feasible to collect & distribute very large datasets for ML. Today, large companies work with image datasets, video datasets, & natural language datasets that couldn't have been collected without internet. User-generated image tags on Flickr, e.g., have been a treasure trove of data for computer vision. So are YouTube videos. & Wikipedia is a key dataset for natural language processing.

    If there's 1 dataset that has been a catalyst for rise of DL, it's ImageNet dataset, consisting of 1.4 million images that have been hand annotated with 1000 image categories (1 category per image). But what makes ImageNet special isn't just its large size, but also yearly competition associated with it. ImageNet Large Scale Visual Recognition Challenge (ILSVRC), www.image-net.org/challenges/LSVRC.

    As Kaggle has been demonstrating since 2010, public competitions are an excellent way to motivate researchers & engineers to push envelope. Having common benchmarks that researchers compete to beat has greatly helped rise of DL, by highlighting its success against classical ML approaches.

∗ 1.3.3. Algorithms. In addition to hardware & data, until late 2000s, were missing a reliable way to train very deep neural networks. As a result, neural networks were still fairly shallow, using only 1 or 2 layers of representations; thus, they weren't able to shine against more-refined shallow methods e.g. SVMs & random forests. Key issue was that of *gradient propagation* through deep stacks of layers. Feedback signal used to train neural networks would fade away as number of layers increased.

This changed around 2009–2010 with advent of several simple but important algorithmic improvements that allowed for better gradient propagation:

· Better *activation functions* for neural layers

· Better *weight-initialization schemes*, starting with layer-wise pretraining, which was then quickly abandoned

· Better *optimization schemes*, e.g. RMSProp & Adam

Only when these improvements began to allow for training models with 10 or more layers did DL start to shine.

Finally, in 2014–2016, even more advanced ways to improve gradient propagation were discovered, e.g. batch normalization, residual connections, & depthwise separable convolutions.

Today, can train models that are arbitrarily deep from scratch. This has unlocked use of extremely large models, which hold considerable representational power, i.e., which encode very rich hypothesis spaces. This extreme scalability is 1 of defining characteristics of modern DL. Large-scale model architectures, which feature 10s of layers & 10s of millions of parameters, have brought about critical advances both in computer vision (e.g., architectures e.g. ResNet, Inception, or Xception) & natural language processing (e.g., large Transformer-based architectures e.g. BERT, GPT-3, or XLNet).

∗ 1.3.4. A new wave of investment. As DL became new state of art for computer vision in 2012–2013, & eventually for all perceptual tasks, industry leaders took note. What followed was a gradual wave of industry investment far beyond anything previously seen in history of AI (Fig. 1.14: OECD estimate of total investments in AI startup.).

In 2011, right before DL took spotlight, total venture capital investment (tổng vốn đầu tư mạo hiểm) in AI worldwide was $<$ a billion dollars, which went almost entirely to practical applications of shallow ML approaches. In 2015, it had risen to over $5 billion, & in 2017, to a staggering $16 billion. Hundreds of startups launched in these few years, trying to capitalize on DL hype. Meanwhile, large tech companies e.g. Google, Amazon, & Microsoft have invested in internal research departs in amounts that would most likely dwarf flow of venture-capital money.

ML – in particular, DL – has become central to product strategy of these tech giants. In late 2015, Google CEO Sundar Pichai stated "ML is a core, transformative way by which we're rethinking how we're doing everything. Thoughtfully applying it across all products, be it search, ads, YouTube, or Play. & we're in early days, but you'll see us – in a systematic way – apply ML in all these areas." [Sundar Pichai, Alphabet earnings call, Oct. 22, 2015.]

As a result of this wave of investment, number of people working on DL went from a few hundred to 10s of thousands in $<$ 10 years, & research progress has reached a frenetic pace.

∗ 1.3.5. Democratization of DL. 1 of key factors driving this inflow of new facts in DL has been democratization of toolsets in field. In early days, doing DL required significant C++ & CUDA expertise, which few people possessed.

Nowadays, basic Python scripting skills suffice to do advanced DL research. This has been driven most notably by development of now-defunct Theano library, & then TensorFlow library – 2 symbolic tensor-manipulation frameworks for Python that support autodifferentiation, greatly simplifying implementation of new models – & by rise of user-friendly libraries e.g. Keras, which makes DL as easy as manipulating LEGO bricks. After its release in early 2015, Keras quickly became go-to DL solution for large numbers of new startups, graduate students, & researchers pivoting into field.

∗ 1.3.6. Will it last?. Is there anything special about deep neural networks that makes them "right" approach for companies to be investing in & for researchers to flock to? Or is DL just a fad that may not last? Will still be using deep neural networks in 20 years?

DL has several properties that justify its status as an AI revolution, & it's here to stay. May not be using neural networks 2 decades from now, but whatever we use will directly inherit from modern DL & its core concepts. These important properties can be broadly sorted into 3 categories:

· *Simplicity*: DL removes need for feature engineering, replacing complex, brittle, engineering-heavy pipelines with simple, end-to-end trainable models that are typically built using only 5 or 6 different tensor operations.

· *Scalability*: DL is highly amenable to parallelization (rất dễ dàng để song song hóa) on GPUs or TPUs, so it can take full advantage of Moore's law. In addition, DL models are trained by iterating over small batches of data, allowing them to be trained on datasets of arbitrary size. (Only bottleneck is amount of parallel computational power available, which, thanks to Moore's law, is a fast-moving barrier.)

· *Versatility & reusability*: Unlike many prior ML approaches, DL models can be trained on additional data without restarting from scratch, making them viable for continuous online learning – an important property for very large production models. Furthermore, trained DL models are repurposable & thus reusable: e.g., possible to take a DL model trained for image classification & drop it into a video-processing pipeline. This allows us to reinvest previous work into increasingly complex & powerful models. This also makes DL applicable to fairly small datasets.

DL has only been in spotlight for a few years, & may not yet have established full scope of what it can do. With every passing year, learn about new use cases & engineering improvements that lift previous limitations. Following a scientific revolution, progress generally follows a sigmoid curve: it starts with a period of fast progress, which gradually stabilizes a researchers hit hard limitations, & then further improvements become incremental.

When Chollet was writing of this book, in 2016, predicted: DL was still in 1st half of that sigmoid, with much more transformative progress to come in following few years. This has proven true in practice, as 2017 & 2018 have seen rise of

Transformer-based DL models for natural language processing, which have been a revolution in field, while DL also kept delivering steady progress in computer vision & speech recognition. Today, in 2021, DL seems to have entered 2nd half of that sigmoid. Should still expect significant progress in years to come, but probably out of initial phase of explosive progress.

Today, extremely excited about deployment of DL technology to every problem it can solve – list is endless. DL is still a revolution in making, & will take many years to realize its full potential.

- 2. Mathematical building blocks of neural networks. Cover: A 1st example of a neural network. Tensors & tensor operations. How neural networks learn via backpropagation & gradient descent.

  Understanding DL requires familiarity with many simple mathematical concepts: *tensors, tensor operations, differentiation, gradient descent*, & so on. Goal of this chap: build up your intuition about these notions without getting overly technical. In particular, steer away from mathematical notation, which can introduce unnecessary barriers for those without any mathematics background & isn't necessary to explain things well. Most precise, unambiguous description of a mathematical operation is its executable code.

  To provide sufficient context for introducing tensors & gradient descent, begin chap with a practical example of a neural network. Then go over every new concept that's been introduced, point by point. Keep in mind: these concepts will be essential for you to understand practical examples in following chaps.

  After reading this chap, have an intuitive understanding of mathematical theory behind DL, & ready to start diving into Keras & TensorFlow in Chap. 3.

  - 2.1. A 1st look at a neural network. Look at a concrete example of a neural network that uses Python library Keras to learn to classify handwritten digits. Unless already have experience with Keras or similar libraries, won't understand everything about this 1st example right away. That's fine. In next chap, review each element in example & explain them in detail. So don't worry if some steps seem arbitrary or look like magic to you! Got to start somewhere.

    Problem trying to solve: classify grayscale images of handwritten digits 28×28 pixels into their 10 categories (0 through 9). Use MNIST dataset, a classic in ML community, which has been around almost as long as field itself & has been intensively studied. It's a set of 60000 training images, plus 10000 test images, assembled by National Institute of Standards & Technology (NIST in MNIST) in 1980s. Can think of "solving" MNIST as "Hello World" of DL – it's what you do to verify that your algorithms are working as expected. As become a ML practitioner, see MNIST come up over & over again in scientific papers, blog posts, & so on. Can see some MNIST samples in Fig. 2.1: MNIST sample digits.

    Note 1. *In ML, a* category *in a classification problem is called a* class. *Data points are called* samples. *Class associated with a specific sample is called a* label.

    Don't need to try to reproduce this example on your machine just now. If wish to, 1st need to set up a DL workspace, covered in Chap. 3.

    MNIST dataset comes preloaded in Keras, in from of a set of 4 NumPy arrays. Listing 2.1: Loading MNIST dataset in Keras.

    ```
    from tensorflow.keras.datasets import mnist
    (train_images, train_labels), (test_images, test_labels) = mnist.load_data()
    ```

    `train_images, train_labels` form training set, data that model will learn from. Model will then be tested on test set, `test_images, test_labels`. Images are encoded as NumPy arrays, & labels are an array of digits, ranging from 0 to 9. Images & labels have a 1-1 correspondence. Look at training data:

    ```
    >>> train_images.shape
    (60000, 28, 28)
    >>> len(train_labels)
    60000
    >>> train_labels
    array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
    ```

    Test data:

    ```
    >>> test_images.shape
    (10000, 28, 28)
    >>> len(test_labels)
    10000
    >>> test_labels
    array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
    ```

    Workflow will be as follows: 1st, feed neural network training data, `train_images, train_labels`. Network will then learn to associate images & labels. Finally, ask network to produce predictions for `test_images`, & verify whether these predictions match labels from `test_labels`.

    Build network – again, remember: aren't expected to understand everything about this example yet. Listing 2.2: Network architecture:

```
from tensorflow import keras
from tensorflow.keras import layers
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

Core building block of neural networks is *layer*. Can think of a layer as a filter for data: some data goes in, & it comes out in a more useful form. Specifically, layers extract *representations* out of data fed into them – hopefully, representations that are more meaningful for problem at hand. Most of DL consists of chaining together simple layers that will implement a form of progressive *data distillation*. A DL model is like a sieve for data processing, made of a succession of increasingly refined data filters – layers.

Here, model consists of a sequence of 2 `Dense` layers, which are densely connected (also called *fully connected*) neural layers. 2nd (& last) layer is a 10-way *softmax classification* layer, i.e. will return an array of 10 probability scores (summing to 1). Each score will be probability: current digit image belongs to 1 of our 10 digit classes.

To make model ready for training, need to pick 3 more things as part of *compilation* step:

* *An optimizer*: Mechanism through which model will update itself based on training data it sees, so as to improve its performance.
* *A loss function*: How model will be able to measure its performance on training data, & thus how it will be able to steer itself in right direction.
* *Metrics to monitor during training & testing*: Here only care about accuracy (fraction of images that were correctly classified).

Exact purpose of loss function & optimizer will be made clear throughout next 2 chaps. Listing 2.3: Compilation step.

```
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

Before training, preprocess data by reshaping it into shape model expects & scaling it so that all values are in $[0, 1]$ interval. Previously, training images were stored in an array of shape $(60000, 28, 28)$ of type `uint8` with values in $[0, 255]$ interval. Transform it into a `float32` array of shape `(60000, 28*28)` with values between 0 & 1. Listing 2.4: Preparing image data.

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255
```

Now ready to train model, which in Keras is done via a call to model's `fit()` method – *fit* model to its training data. Listing 2.5: "Fitting" model.

```
>>> model.fit(train_images, train_labels, epochs=5, batch_size=128)
Epoch 1/5
60000/60000 [==============================] - 5s - loss: 0.2524 - acc: 0.9273
Epoch 2/5
51328/60000 [====================>.....] - ETA: 1s - loss: 0.1035 - acc: 0.9692
```

2 quantities are displayed during training: loss of model over training data, & accuracy of model over training data. Quickly reach an accuracy of $0.989 = 98.9\%$ on training data.

Now: have a trained model, can use it to predict class probabilities for *new* digits – images that weren't part of training data, like those from test set. Listing 2.6: Using model to make predictions.

```
>>> test_digits = test_images[0:10]
>>> predictions = model.predict(test_digits)
>>> predictions[0]
array([1.0726176e-10, 1.6918376e-10, 6.1314843e-08, 8.4106023e-06,
       2.9967067e-11, 3.0331331e-09, 8.3651971e-14, 9.9999106e-01,
       2.6657624e-08, 3.8127661e-07], dtype=float32)
```

Each number of index `i` in that array corresponds to probability that digit image `test_digits[0]` belongs to class `i`.

This 1st test digit has highest probability score (0.99999106, almost 1) at index 7, so according to our model, it must be a 7:

```
>>> predictions[0].argmax()
7
>>> predictions[0][7]
0.99999106
```

Can check: test label agrees:

```
>>> test_labels[0]
7
```

On average, how good is our model at classifying such never-before-seen digits? Check by computing average accuracy over entire test set. Listing 2.7: Evaluating model on new data.

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> print(f"test_acc: {test_acc}")
test_acc: 0.9785
```

Test-set accuracy turns out to be 97.8% – that's quite a bit lower than training set accuracy 98.9%. This gap between training accuracy & test accuracy is an example of *overfitting*: fact that ML models tend to perform worse on new data than on their training data. Overfitting is a central topic in Chap. 3.

This concludes 1st example – just saw how can build & train a neural network to classify handwritten digits in < 15 lines of Python code. In this chap & next, go into detail about every moving piece just previewed & clarify what's going on behind scenes. Learn about tensors, data-storing objects going into model; tensor operations, which layers are made of; & gradient descent, which allows your model to learn from its training examples.

- 2.2. Data representations for neural networks. In prev example, started from data stored in multidimensional NumPy arrays, also called *tensors*. In general, all current ML systems use tensors as their basic data structure. Tensors are fundamental to field – so fundamental that TensorFlow was named after them. So what's a tensor?

  At its score, a tensor is a container for data – usually numerical data. So, it's a container for numbers. May be already familiar with matrices, which are rank-2 tensors: tensors are a generalization of matrices to an arbitrary number of *dimensions* (note: in context of tensors, a dimension is often called an *axis*).

  - 2.2.1. Scalars (rank-0 tensors). A tensor that contains only 1 number is called a *scalar* (or scalar tensor, or rank-0 tensor, or 0D tensor). In NumPy, a `float32` or `float64` number is a scalar tensor (or scalar array). Can display number of axes of a NumPy tensor via `ndim` attribute; a scalar tensor has 0 axes `ndim == 0`. Number of axes of a tensor is also called its *rank*. A NumPy scalar:

    ```
    >>> import numpy as np
    >>> x = np.array(12)
    >>> x
    array(12)
    >>> x.ndim
    0
    ```

  - 2.2.2. Vectors (rank-1 tensors). An array of numbers is called a *vector*, or rank-1 tensor, or 1D tensor. A rank-1 tensor is said to have exactly 1 axis. A NumPy vector:

    ```
    >>> x = np.array([12, 3, 6, 14, 7])
    >>> x
    array([12, 3, 6, 14, 7])
    >>> x.ndim
    1
    ```

    This vector has 5 entries & so is called a *5-dimensional vector*. Don't confuse a 5D vector with a 5D tensor! A 5D vector has only 1 axis & has 5 dimensions along its axis, whereas a 5D tensor has 5 axes (& may have any number of dimensions along each axis). *Dimensionality* can denote either number of entries along a specific axis (as in case of our 5D vector) or number of axes in a tensor (e.g. a 5D tensor), which can be confusing at times. In latter case, technically more correct to talk about a *tensor of rank* 5 (rank of a tensor being number of axes), but ambiguous notation *5D tensor* in common regardless.

  - 2.2.3. Matrices (rank-2 tensors). An array of vectors is a *matrix*, or rank-2 tensor, or 2D tensor. A matrix has 2 axes (often referred to as *rows, columns*). Can visually interpret a matrix as a rectangular grid of numbers. A NumPy matrix:

    ```
    >>> x = np.array([[5, 78, 2, 34, 0], [6, 79, 3, 35, 1], [7, 80, 4, 36, 2]])
    >>> x.ndim
    2
    ```

Entries from 1st axis are called *rows*, & entries from 2nd axis are called *columns*. In prev example, `[5, 78, 2, 34, 0]` is 1st row of `x`, & `[5, 6, 7]` is 1st column.

∗ 2.2.4. Rank-3 & higher-rank tensors. If pack such matrices in a new array, obtain a rank-3 tensor (or 3D tensor), which can visually interpret as a cube of numbers. Following is a NumPy rank-3 tensor:

```
>>> x = np.array([[[5, 78, 2, 34, 0], [6, 79, 3, 35, 1], [7, 80, 4, 36, 2]], [[5, 78, 2, 34, 0], [6, 7
>>> x.ndim
3
```

By packing rank-3 tensors in an array, can create a rank-4 tensor, & so on. In DL, generally manipulate tensors with ranks 0–4, although may go up to 5 if process video data.

∗ 2.2.5. Key attributes. A tensor is defined by 3 key attributes:

· *Number of axes (rank).* E.g., a rank-3 tensor has 3 axes, & a matrix has 2 axes. This is also called tensor's `ndim` in Python libraries e.g. NumPy or TensorFlow.

· *Shape*: This is a tuple of integers that describes how many dimensions tensor has along each axis. E.g., prev matrix example has shape `(3, 5)`, & rank-3 tensor example has shape `(3, 3, 5)`. A vector has a shape with a single element, e.g. `(5,)`, whereas a scalar has an empty shape ().

· *Data type (usually called* `dtype` *in Python libraries).* This is type of data contained in tensor; e.g., a tensor's type could be `float16, float32, float64, uint8`, & so on. In TensorFlow, also likely to come across `string` tensors.

To make this more concrete, look back at data processed in MNIST example. 1st, load MNIST dataset:

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

Display number of axes of tensor `train_images`, `ndim` attribute:

```
>>> train_images.ndim
3
```

Its shape:

```
>>> train_images.shape
(60000, 28, 28)
```

& this is its data type, `dtype` attribute:

```
>>> train_images.dtype
uint8
```

So what have here is a rank-3 tensor of 8-bit integers. More precisely, it's an array of 60000 matrices of 28×28 integers. Each such matrix is a grayscale image, with coefficients between 0 & 255.
Display 4th digit in this rank-3 tensor, using Matplotlib library (a well-known Python data visualization library, which comes preinstalled in Colab); Fig. 2.2: 4th sample in our dataset. Listing 2.8: Displaying 4th digit.

```
import matplotlib.pyplot as plt
digit = train_images[4]
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

Naturally, corresponding label is integer 9:

```
>>> train_labels[4]
9
```

∗ 2.2.6. Manipulating tensors in NumPy. In prev example, selected a specific digit alongside 1st axis using syntax `train_images[i]`. Selecting specific elements in a tensor is called *tensor slicing*. Look at tensor-slicing operations can do on NumPy arrays. Following example selects digits #10 to #100 (#100 isn't included) & puts them in an array of shape `(90. 28, 28)`:

```
>>> my_slice = train_images[10:100]
>>> my_slice.shape
(90, 28, 28)
```

⇔ this more detailed notation, which specifies a start index & stop index for slice along each tensor axis. Note: `:` ⇔ selecting entire axis:

```
>>> my_slice = train_images[10:100, :, :]
>>> my_slice.shape
(90, 28, 28)
>>> my_slice = train_images[10:100, 0:28, 0:28]
```

```
>>> my_slice.shape
(90, 28, 28)
```

In general, may select slices between any 2 indices along each tensor axis. E.g., in order to select 14×14 pixels in bottom-right corner of all images, would do this:

```
my_slice = train_images[:, 14:, 14:]
```

Also possible to use negative indices. Much like negative indices in Python lists, they indicate a position relative to end of current axis. In order to crop images to patches of 14×14 pixels centered in middle, do this:

```
my_slice = train_images[:, 7:-7, 7:-7]
```

∗ 2.2.7. Notion of data batches. In general, 1st axis (axis 0, because indexing starts at 0) in all data tensors come across in DL will be *samples axis* (sometimes called *samples dimension*). In MNIST example, "samples" are images of digits.
In addition, DL models don't process an entire dataset at once; rather, they break data into small batches. Concretely, 1 batch of MNIST digits, with a batch size of 128:

```
batch = train_images[:128]
```

Next batch:

```
batch = train_images[128:256]
```

nth batch:

```
n = 3
batch = train_images[128 * n:128 * (n + 1)]
```

When considering such a batch tensor, 1st axis (axis 0) is called *batch axis* or *batch dimension*. This is a term frequently encounter when using Keras & other DL libraries.

∗ 2.2.8. Real-world examples of data tensors. Make data tensors more concrete with a few examples similar to what encounter later. Data you'll manipulate will almost always fall into 1 of following categories:

· *Vector data*: Rank-2 tensors of shape `(samples, features)`, where each sample is a vector of numerical attributes ("features")

· *Timeseries data or sequence data*: Rank-3 tensors of shape `(samples, timesteps, features)`, where each sample is a sequence (of length `timesteps`) of feature vectors

· *Images*: Rank-4 tensors of shape `(samples, height, width, channels)`, where each sample is a 2D grid of pixels, & each pixel is represented by a vector of values ("channels")

· *Video*: Rank-5 tensors of shape `(samples, frames, height, width, channels)`, where each sample is a sequence (of length `frames`) of images

∗ 2.2.9. Vector data. This is 1 of most common cases. In such a dataset, each single data point can be encoded as a vector, & thus a batch of data will be encoded as a rank-2 tensor (i.e., an array of vectors), where 1st axis is *samples axis* & 2nd axis is *features axis*.
2 examples:

· An actuarial dataset of people, where consider each person's age, gender, & income. Each person can be characterized as a vector of 3 values, & thus an entire dataset of 100000 people can be stored in a rank-2 tensor of shape `(100000, 3)`.

· A dataset of text documents, where represent each document by counts of how many times each word appears in it (out of a dictionary of 20000 common words). Each document can be encoded as a vector of 20000 values (1 count per word in dictionary), & thus an entire dataset of 500 documents can be stored in a tensor of shape `(500, 20000)`.

∗ 2.2.10. Timeseries data or sequence data. Whenever time matters in your data (or notion of sequence order), it makes sense to store it in a rank-3 tensor with an explicit time axis. Each sample can be encoded as a sequence of vectors (a rank-2 tensor), & thus a batch of data will be encoded as a rank-3 tensor (Fig. 2.3: A rank-3 timeseries data tensor.).
Time axis is always 2nd axis (axis of index 1) by convention. Examples:

· A dataset of stock prices. Every minute, store current price of stock, highest price in past minute, & lowest price in past minute. Thus, every minute is encoded as a 3D vector, an entire day of trading is encoded as a matrix of shape `(390, 3)` (there are 390 minutes in a trading day), & 250 days' worth of data can be stored in a rank-3 tensor of shape `(250, 390, 3)`. Here, each sample would be 1 day's worth of data.

· A dataset of tweets, where encode each tweet as a sequence of 280 characters out of an alphabet of 128 unique characters. In this setting, each character can be encoded as a binary vector of size 128 (an all-0s vector except for a 1 entry at index corresponding to character). Then each tweet can be encoded as a rank-2 tensor of shape `(280, 128)`, & a dataset of 1 million tweets can be stored in a tensor of shape `(1000000, 280, 128)`.

∗ 2.2.11. Image data. Images typically have 3 dimensions: height, width, & color depth. Although grayscale images (like MNIST digits) have only a single color channel & could thus be stored in rank-2 tensors, by convention image tensors are

always rank-3, with a 1D color channel for grayscale images. A batch of 128 grayscale images of size 256×256 could thus be stored in a tensor of shape `(128, 256, 256, 1)`, & a batch of 128 color images could be stored in a tensor of shape `(128, 256, 256, 3)` (Fig. 2.4: A rank-4 image data tensor.).
There are 2 conventions for shapes of image tensors: *channels-last* convention (which is standard in TensorFlow) & *channels-1st* convention (which is increasingly falling out of favor).
Channels-last convention places color-depth axis at end: `(samples, height, width, color_depth)`. Meanwhile, channels-1st convention places color depth axis right after batch axis: `(samples, color_depth, height, width)`. With channels-1st convention, prev examples would become `(128, 1, 256, 256)`, `(128, 3, 256, 256)`. Keras API provides support for both formats.

* 2.2.12. Video data. Video data is 1 of few types of real-world data for which need a rank-5 tensors. A video can be understood as a sequence of frames, each frame being a color image. Because each frame can be stored in a rank-3 tensor `(height, width, color_depth)`, a sequence of frames can be stored in a rank-4 tensor `(frames, height, width, color_depth)`, & thus a batch of different videos can be stored in a rank-5 tensor of shape `(samples, frames, height, width, color_depth)`. E.g., a 60-sec, 144×256 YouTube video clip sampled at 4 frames per sec would have 240 frames. A batch of 4 such video clips would be stored in a tensor of shape `(4, 240, 144, 256, 3)`: a total of 106,168,320 values! If `dtype` of tensor was `float32`, each value would be stored in 32 bits, so tensor would represent 405 MB. Heavy! Videos encounter in real life are much lighter, because they aren't stored in `float32`, & they're typically compressed by a large factor (e.g. in MPEG format).

- 2.3. Gears (Bánh răng) of neural networks: Tensor operations. Much as any computer program can be ultimately reduced to a small set of binary operations on binary inputs (AND, OR, NOR, & so on), all transformations learned by deep neural networks can be reduced to a handful of *tensor operations* (or *tensor functions*) applied to tensors of numeric data. E.g., possible to add tensors, multiply tensors, & so on.

  In initial example, build model by stacking `Dense` layers on top of each other. A Keras layer instance looks like this:

```
keras.layers.Dense(512, activation="relu")
```

  This layer can be interpreted as a function, which takes as input a matrix & returns another matrix – a new representation for input tensor. Specifically, function is as follows (where `W`: a matrix & `b`: a vector, both attributes of layer):

```
output = relu(dot(input, W) + b)
```

  Unpack this: Have 3 tensor operations here:

  * A dot product `dot` between input tensor & a tensor named `W`
  * An addition + between resulting matrix & a vector `b`
  * A `relu` operation: `relu(x)` is `max(x, 0)`; "relu" stands for "rectified linear unit"

  Note 2. *Although this sect deals entirely with linear algebra expressions, won't find any mathematical notation here. Found: mathematical concepts can be more readily mastered by programmers with no mathematical background if they're expressed as short Python snippets instead of mathematical equations. So use NumPy & TensorFlow code throughout.*

  * 2.3.1. Element-wise operations. `relu` operation & addition are element-wise operations: operations that are applied independently to each entry in tensors being considered. I.e. these operations are highly amenable to massively parallel implementations (*vectorized* implementations, a term that comes from *vector processor* supercomputer architecture from 1970–90 period). If want to write a native Python implementation of an element-wise operation, use a `for` loop, as in this naive implementation of an element-wise `relu` operation:

```
def naive_relu(x):
    assert len(x.shape) == 2
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```

  Could do same for addition:

```
def naive_add(x, y):
    assert len(x.shape) == 2
    assert x.shape == y.shape
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x
```

On same principle, can do element-wise multiplication, subtraction, & so on.
In practice, when dealing with NumPy arrays, these operations are available as well-optimized built-in NumPy functions, which themselves delegate (đại biểu) heavy lifting to a Basic Linear Algebra Subprograms (BLAS) implementation. BLAS are low-level, highly parallel, efficient tensor-manipulation routines that are typically implemented in Fortran or C.
So, in NumPy, can do following element-wise operation, & it will be blazing fast:

```
import numpy as np
z = x + y # element-wise addition
z = np.maximum(z, 0.) # element-wise relu
```

Actually time difference:

```
import time
x = np.random.random((20, 100))
y = np.random.random((20, 100))
t0 = time.time()
for _ in range(1000):
    z = x + y
    z = np.maximum(z, 0.)
print("Took: {0:.2f} s".format(time.time() - t0))
```

This takes 0.02s. Meanwhile, naive version takes a stunning 2.45 s:

```
t0 = time.time()
for _ in range(1000):
    z = naive_add(x, y)
    z = naive_relu(z)
print("Took: {0:.2f} s".format(time.time() - t0))
```

Likewise, when running TensorFlow code on a GPU, element-wise operations are executed via fully vectorized CUDA implementations that can best utilize highly parallel GPU chip architecture.

∗ 2.3.2. Broadcasting. Earlier naive implementation of `naive_add` only supports addition of rank-2 tensors with identical shapes. But in `Dense` layer introduced earlier, added a rank-2 tensor with a vector. What happens with addition when shapes of 2 tensors being added differ?
When possible, & if there's no ambiguity, smaller tensor will be *broadcast* to match shape of larger tensor. Broadccasting consists of 2 steps:

1. Axes (called *broadcast axes*) are added to smaller tensor to match `ndim` of larger tensor.
2. Smaller tensor is repeated alongside these new axes to match full shape of larger tensor.

A concrete example. Consider `X` with shape `(32, 10)` & `y` with shape `(10,)`:

```
import numpy as np
X = np.random.random((32, 10))
y = np.random.random((10,))
```

1st, add an empty 1st axis to `y`, whose shape becomes `(1, 10)`:

```
y = np.expand_dims(y, axis=0)
```

Then, repeat `y` 32 times along this new axis, so that end up with a tensor `Y` with shape `(32, 10)`, where `Y[i, :] == y` for `i` in `range(0, 32)`:

```
Y = np.concatenate([y] * 32, axis=0)
```

At this point, can proceed to add `X, Y`, because they have same shape.
In terms of implementation, no new rank-2 tensor is created, because that would be terribly inefficient. Repetition operation is entirely virtual: it happens at algorithmic level rather than at memory level. But thinking of vector being repeated 10 times alongside a new axis is a helpful mental model. Here's what a naive implementation would look like:

```
def naive_add_matrix_and_vector(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[j]
    return x
```

With broadcasting, can generally perform element-wise operations that take 2 inputs tensors if 1 tensor has shape `(a, b, ... n, n + 1, ... m)` & other has shape `(n, n + 1, ... m)`. Broadcasting will then automatically happen for axes `a` through `n - 1`.
Following example applies element-wise `maximum` operation to 2 tensors of different shapes via broadcasting:

```
import numpy as np
x = np.random.random((64, 3, 32, 10))
y = np.random.random((32, 10))
z = np.maximum(x, y)
```

∗ 2.3.3. Tensor product. *Tensor product*, or *dot product* (not to be confused with an element-wise product, * operator), is 1 of most common, most useful tensor operations.
In NumPy, a tensor product is done using `np.dot` function (because mathematical notation for tensor product is usually a dot):

```
x = np.random.random((32,))
y = np.random.random((32,))
z = np.dot(x, y)
```

In mathematical notation: $z = x \cdot y$. Mathematically, what does dot operation do? Start with dot product of 2 vectors `x, y`: computed as follows:

```
def naive_vector_dot(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]
    z = 0.
    for i in range(x.shape[0]):
        z += x[i] * y[i]
    return z
```

Notice: dot product between 2 vectors is a scalar & that only vectors with same number of elements are compatible for a dot product.
Can also take dot product between a matrix `x` & a vector `y`, which returns a vector where coefficients are dot products between `y` & rows of `x`. Implement it as follows:

```
def naive_matrix_vector_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            z[i] += x[i, j] * y[j]
    return z
```

Could also reuse code wrote previously, which highlights relationship between a matrix-vector product & a vector product:

```
def naive_matrix_vector_dot(x, y):
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        z[i] = naive_vector_dot(x[i, :], y)
    return z
```

Note: as soon as 1 of 2 tensors has an `ndim` > 1, `dot` is no longer *symmetric*, i.e. `dot(x, y)` isn't same as `dot(y, x)`.
Of course, a dot product generalizes to tensors with an arbitrary number of axes. Most common applications may be dot product between 2 matrices. Can take dot product of 2 matrices `x, y` (`dot(x, y)`) iff `x.shape[1] == y.shape[0]`. Result is a matrix with shape `(x.shape[0], y.shape[1])`, where coefficients are vector products between rows of `x` & columns of `y`. Naive implementation:

```
def naive_matrix_dot(x, y):
    assert len(x.shape) == 2 # x & y are NumPy matrices.
    assert len(y.shape) == 2
    assert x.shape[1] == y.shape[0]
    z = np.zeros((x.shape[0], y.shape[1]))
    for i in range(x.shape[0]):
        for j in range(y.shape[1]):
            row_x = x[i, :]
```

```
            column_y = y[:, j]
            z[i, j] = naive_vector_dot(row_x, column_y)
    return z
```

To understand dot-product shape compatibility, help to visualize input & output tensors by aligning them as shown in Fig. 2.5: Matrix dot-product box diagram.

In figure, `x, y, z` are pictured as rectangles (literal boxes of coefficients). Because rows of `x` & columns of `y` must have same size, it follows: width of `x` must match height of `y`. If go on to develop new ML algorithms, likely be drawing such diagrams often.

More generally, can take dot product between higher-dimensional tensors, following same rules for shape compatibility as outlined earlier for 2D case: $(a, b, c, d) \cdot (d,) \to (a, b, c)$, $(a, b, c, d) \cdot (d, e) \to (a, b, c, e)$, & so on.

∗ 2.3.4. Tensor reshaping. A 3rd type of tensor operation that's essential to understand is *tensor reshaping*. Although it wasn't used in `Dense` layers in 1st neural network example, used it when preprocessed digits data before feeding it into our model:

```
train_images = train_images.reshape((60000, 28 * 28))
```

Reshaping a tensor means rearranging its rows & columns to match a target shape. Naturally, reshaped tensor has same total number of coefficients as initial tensor. Reshaping is best understood via simple examples:

```
>>> x = np.array([[0., 1.], [2., 3.], [4., 5.]])
>>> x.shape
(3, 2)
>>> x = x.reshape((6, 1))
>>> x
array([[ 0.], [ 1.], [ 2.], [ 3.], [ 4.], [ 5.]])
>>> x = x.reshape((2, 3))
>>> x
array([[ 0., 1., 2.],
       [ 3., 4., 5.]])
```

A special case of reshaping that's commonly encountered is *transposition*. *Transposing* a matrix means exchanging its rows & its columns, so that `x[i, :]` becomes `x[:, i]`:

```
>>> x = np.zeros((300, 20))
>>> x = np.transpose(x)
>>> x.shape
(20, 300)
```

∗ 2.3.5. Geometric interpretation of tensor operations. Because contents of tensors manipulated by tensor operations can be interpreted as coordinates of points in some geometric space, all tensor operations have a geometric interpretation. E.g., consider addition. Start with vector: `A = [0.5, 1]` – Fig. 2.6: A point in 2D space. Common to picture a vector as an arrow linking origin to point Fig. 2.7: A point in a 2D space pictured as an arrow. Consider a new point `B = [1, 0.25]`, which add to prev one. This is done geometrically by changing together vector arrows, with resulting location being vector representing sum of prev 2 vectors (Fig. 2.8: Geometric interpretation of sum of 2 vectors). As can see, adding a vector B to a vector A represents action of copying point A in a new location, whose distance & direction from original point A is determined by vector B. If apply same vector addition to a group of points in plane (an "object"), would be creating a copy of entire object in a new location (Fig. 2.9: 2D translation as a vector addition). Tensor addition thus represents action of *translating an object* (moving object without distorting it) by a certain amount in a certain direction.

In general, elementary geometric operations e.g. translation, rotation, scaling, skewing, & so on can be expressed as tensor operations. A few examples:

- *Translation*: As just saw, adding a vector to a point will move point by a fixed amount in a fixed direction. Applied to a set of points (e.g. a 2D object), called a "translation".
- *Rotation*: A counterclockwise rotation of a 2D vector by an angle theta (Fig. 2.10: 2D rotation (counterclockwise) as a dot product) can be achieved via a dot product with a 2×2 matrix `R = [[cos(theta), -sin(theta)], [sin(theta), cos(theta)]]`.

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

- *Scaling*: A vertical & horizontal scaling of image (Fig. 2.11: 2D scaling as a dot product) can be achieved via a dot product with a 2×2 matrix `S = [[horizontal_factor, 0], [0, vertical_factor]]` (note: such a matrix is called a "diagonal matrix", because it only has nonzero coefficients in its "diagonal", going from top left to bottom right).
- *Linear transform*: A dot product with an arbitrary matrix implements a linear transform. Note: *scaling & rotation* are by definition linear transforms.
- *Affine transform*: An affine transform (Fig. 2.12: Affine transform in plane.) is combination of a linear transform (achieved via a dot product with some matrix) & a translation (achieved via a vector addition). As have probably recognized,

that's exactly `y = W • x + b` computation implemented by `Dense` layer! A `Dense` layer without an activation function is an affine layer.

- `Dense` *layer with* `relu` *activation*: An important observation about affine transforms: if apply many of them repeatedly, still end up with an affine transform (so could just have applied that 1 affine transform in 1st place). Try it with 2: `affine2(affine1(x)) = W2 • (W1 • x + b1) + b2 = (W2 • W1) • x + (W2 • b1 + b2)`. That's an affine transform where linear part is matrix `W2 • W1` & translation part is vector `W2 • b1 + b2`. As a consequence, a multilayer neural network made entirely of `Dense` layers without activations would be ⇔ a single `Dense` layer. This "deep" neural network would just be a linear model in disguise! This is why need activations, like `relu` (seen in action in Fig. 2.13: Affine transform followed by `relu` activation). Thanks to activation functions, a chain of `Dense` layers can be made to implement very complex, nonlinear geometric transformations, resulting in very rich hypothesis spaces for deep neural networks. Cover this idea in more detail in next chap.

* 2.3.6. A geometric interpretation of DL. Just learned: neural networks consist entirely of chains of tensor operations, & that these tensor operations are just simple geometric transformations of input data. It follows: you can interpret a neural network as a very complex geometric transformation in a high-dimensional space, implemented via a series of simple steps. In 3D, following mental image may prove useful. Imagine 2 sheets of colored paper: 1 red & 1 blue. Put 1 on top of the other. Now crumple (nhàu nát) them together into a small ball. That crumpled paper ball is input data, & each sheet of paper is a class of data in a classification problem. What a neural network is meant to do is figure out a transformation of paper ball that would uncrumple it, so as to make 2 classes cleanly separable again (Fig. 2.14: Uncrumbling a complicated manifold of data). With DL, this would be implemented as a series of simple transformations of 3D space, e.g. those you could apply on paper ball with fingers, 1 movement at a time.

  Uncrumpling paper balls is what ML is about: finding neat representations for complex, highly folded data *manifolds* in high-dimensional spaces (a manifold is a continuous surface, like crumpled sheet of paper). At this point, should have a pretty good intuition as to why DL excels as this: it takes approach of incrementally decomposing a complicated geometric transformation into a long chain of elementary ones, which is pretty much strategy a human would follow to uncrumple a paper ball. Each layer in a deep network applies a transformation that disentangles data a little, & a deep stack of layers makes tractable an extremely complicated disentanglement process.

  – Gỡ rối những quả bóng giấy là mục đích của ML: tìm ra các biểu diễn gọn gàng cho dữ liệu phức tạp, có nhiều nếp gấp *manifolds* trong các không gian có nhiều chiều (một đa tạp là một bề mặt liên tục, giống như tờ giấy nhàu nát). Tại thời điểm này, bạn nên có trực giác khá tốt về lý do tại sao DL lại vượt trội như thế này: nó sử dụng phương pháp phân tích gia tăng một phép biến đổi hình học phức tạp thành một chuỗi dài các phép biến đổi cơ bản, đây gần như là chiến lược mà con người sẽ áp dụng để gỡ rối một quả bóng giấy. Mỗi lớp trong một mạng sâu áp dụng một phép biến đổi giúp gỡ rối dữ liệu một chút, & một chồng lớp sâu giúp quá trình gỡ rối cực kỳ phức tạp trở nên dễ xử lý.

◦ 2.4. Engine of neural networks: Gradient-based optimization. Each neural layer from 1st model example transforms its input data as follows:

```
output = relu(dot(input, W) + b)
```

In this expression, `W, b`: tensors that are attributes of layer, called *weights* or *trainable parameters* of layer (*kernel, bias* attributes, resp.). These weights contain information learned by model from exposure to training data.

Initially, these weight matrices are filled with small random values (a step called *random initialization*). Of course, there's no reason to expect: `relu(dot(input, W) + b)`, when `W, b` are random, will yield any useful representations. Resulting representations are meaningless – but they're a starting point. What comes next is to gradually adjust these weights, based on a feedback signal. This gradual adjustment, also called *training*, is learning that ML is all about.

This happens within what's called a *training loop*, which works as follows. Repeat these steps in a loop, until loss seems sufficiently low:

1. Draw a batch of training samples `x` & corresponding targets `y_true`.
2. Run model on `x` (a step called *forward pass*) to obtain predictions `y_pred`.
3. Compute loss of model on batch, a measure of mismatch between `y_pred, y_true`.
4. Update all weights of model in a way that slightly reduces loss on this batch.

Eventually end up with a model that has a very low loss on its training data: a low mismatch between predictions `y_pred`, & expected targets `y_true`. Model has "learned" to map its inputs to correct targets. From afar, it may look like magic, but when reduce it to elementary steps, it turns out to be simple.

Step 1 sounds easy enough – just I/O code. Steps 2 & 3 are merely application of a handful of tensor operations, so could implement these steps purely from what learned in prev sect. Difficult part is step 4: updating, model's weights. Given an individual weight coefficient in model, how can compute whether coefficient should be increased or decreased, & by how much?

1 naive solution would be to freeze all weights in model except 1 scalar coefficient being considered, & try different values for this coefficient. Say initial value of coefficient is 0.3. After forward pass on a batch of data, loss of model on batch is 0.5. If change coefficient's value to 0.35 & return forward pass, loss increases to 0.6. But if lower coefficient to 0.25, loss falls to 0.4. In this case, it seems: updating coefficient by -0.05 would contribute to minimizing loss. This would have to be repeated for all coefficients in model.

But such an approach would be horribly inefficient, because need to compute 2 forward passes (which are expensive) for every individual coefficient (of which there are many, usually thousands & sometimes up to millions). Thankfully, there's a much better approach: *gradient descent*.

Gradient descent is optimization technique that powers modern neural networks. Here's gist ý chính) of it. All of functions used in our models (e.g. `dot` or +) transform their input in a smooth & continuous way: if look at `z = x + y`, e.g., a small change in `y` only results in a small change in `z`, & if you know direction of change in `y`, can infer direction of change in `z`. Mathematically, say these functions are *differentiable*. If chain together such functions, bigger function obtain is still differentiable. In particular, this applies to function that maps model's coefficients to loss of model on a batch of data: a small change in model's coefficients results in a small, predictable change in loss value. This enables to use a mathematical operator called *gradient* to describe how loss varies as you move model's coefficients in different directions. If compute this gradient, can use it to move coefficients (all at once in a single update, rather than one at a time) in a direction that decreases loss.

If already know what *differentiable* means & what a *gradient* is, can skip to Sect. 2.4.3. Otherwise, following 2 sects help understand concepts.

∗ 2.4.1. What's a derivative? Consider a continuous, smooth function $f(x) = y$, mapping a number x to a new number y. Can use function in Fig. 2.15: A continuous smooth function as an example.

Because function is *continuous*, a small change in x can only result in a small change in y – that's intuition behind *continuity*. Say: increase x by a small factor, `epsilon_x`: this results in a small `epsilon_y` changes to y, as shown in Fig. 2.16: With a continuous function, a small change in x results in a small change in y.

In addition, because function is *smooth* (its curve doesn't have any abrupt angles), when `epsilon_x` is small enough, around a certain point p, possible to approximate f as a linear function of slope a, so that `epsilon_y` becomes `a * epsilon_x`:

```
f(x + epsilon_x) = y + a * epsilon_x
```

Obviously, this linear approximation is valid only when x is close enough to p.

Slope a is called *derivative* of f in p. If a is negative, it means a small increase in x around p will result in a decrease of $f(x)$ (as shown in Fig. 2.17: Derivative of f in p), & if $a > 0$, a small increase in x will result in an increase of $f(x)$. Further, absolute value of a (*magnitude* of derivative) tells how quickly this increase or decrease will happen.

For every differentiable function $f(x)$ (*differentiable* means "can be derived": e.g., smooth, continuous functions can be derived), there exists a derivative function $f'(x)$, that maps values of x to slope of local linear approximation of f in those points. E.g., derivative of $\cos x$ is $-\sin x$, derivative of $f(x) = ax$ is $f'(x) = a$, ...

Being able to derive functions is a very powerful tool when it comes to *optimization*, task of finding values of x that minimize value of $f(x)$. If trying to update x by a factor `epsilon_x` in order to minimize $f(x)$, & know derivative of f, then your job is done: derivative completely describes how $f(x)$ evolves as you change x. If want to reduce value of $f(x)$, just need to move x a little in opposite direction from derivative.

∗ 2.4.2. Derivative of a tensor operation: gradient. Function we were just looking at turned a scalar value x into another scalar value y: could plot it as a curve in a 2D plane. Now imagine a function that turns a tuple of scalars (x, y) into a scalar value z: that would be a vector operation. Could plot it as a 2D *surface* in a 3D space (indexed by coordinates `x, y, z`). Likewise, can imagine functions that take matrices as inputs, functions that take rank-3 tensors as inputs, etc.

Concept of derivation can be applied to any such function, as long as surfaces they describe are continuous & smooth. Derivative of a tensor operation (or tensor function) is called a *gradient*. Gradients are just generalization of concept of derivatives to functions that take tensors as inputs. Remember how, for a scalar function, derivative represents *local slope* of curve of function? In same way, gradient of a tensor function represents *curvature* of multidimensional surface described by function. It characterizes how output of function varies when its input parameters vary.

An example grounded in ML. Consider:

· An input vector `x` (a sample in a dataset)

· A matrix `W` (weights of a model)

· A target `y_true` (what model should learn to associate to `x`)

· A loss function `loss` (meant to measure gap between model's current predictions & `y_true`)

Can use `W` to compute a target candidate `y_pred`, & then compute loss, or mismatch, between target candidate `y_pred` & target `y_true`:

```
y_pred = dot(W, x)
loss_value = loss(y_pred, y_true)
```

Like to use gradients to figure out how to update `W` so as to make `loss_value` smaller. How do we do that?

Given fixed inputs `x, y_true`, preceding operations can be interpreted as a function mapping values of `W` (model's weights) to loss values:

```
loss_value = f(W)
```

Say current value of `W` is `W0`. Then derivative of `f` at point `W0` is a tensor `grad(loss_value, W0)`, with same shape as `W`, where each coefficient `grad(loss_value, W0)[i, j]` indicates direction & magnitude of change in `loss_value` you observe when modifying `W0[i, j]`. That tensor `grad(loss_value, W0)` is gradient of function `f(W) = loss_value` in `W0`, also called "gradient of `loss_value` w.r.t. `W` around `W0`."

Remark 1 (Partial derivatives). *Tensor operation* `grad(f(W), W)` *(which takes as input a matrix* `W`*) can be expressed as a combination of scalar functions,* `grad_ij(f(W), w_ij)`*, each of which would return derivative of* `loss_value = f(W)` *w.r.t. coefficient* `W[i, j]` *of* `W`*, assuming all other coefficients are constant.* `grad_ij` *is called* partial derivative *of* `f` *w.r.t.* `W[i, j]`.

Concretely, what does `grad(loss_value, W0)` represent? Saw earlier: derivative of a function $f(x)$ of a single coefficient can be interpreted as slope of curve of f. Likewise, `grad(loss_value, W0)` can be interpreted as tensor describing *direction of steepest ascent* of `loss_value = f(W)` around `W0`, as well as slope of this ascent. Each partial derivative describes slope of f in a specific direction.

For this reason, in much same way: for a function $f(x)$, can reduce value of $f(x)$ by moving x a little in opposite direction from derivative, with a function `f(W)` of a tensor, can reduce `loss_value = f(W)` by moving `W` in opposite direction from gradient: e.g., `W1 = W0 - step * grad(f(W0), W0)` (where `step` is a small scaling factor). I.e., going against direction of steepest ascent of f, which intuitively should put you lower on curve. Note: scaling factor `step` is needed because `grad(loss_value, W0)` only approximates curvature when close to `W0`, so don't want to get too far from `W0`.

∗ 2.4.3. Stochastic gradient descent. Given a differentiable function, it's theoretically possible to find its minimum analytically: known: a function's minimum is a point where derivative is 0, so all you have to do is find all points where derivative goes to 0 & check for which of these points function has lowest value.

Applied to a neural network, i.e. finding analytically combination of weight values that yields smallest possible loss function. This can be done by solving equation `grad(f(W), W) = 0` for `W`. This is a polynomial equation of N variables, where N: number of coefficients in model. Although it would be possible to solve such an equation for $N = 2$ or $N = 3$, doing so is intractable for real neural networks, where number of parameters is never $<$ a few thousand & can often be several 10s of millions.

Instead, can use 4-step algorithm outlined at beginning of this sect: modify parameters little by little based on current loss value for a random batch of data. Because dealing with a differentiable function, can compute its gradient, which gives you an efficient way to implement Step 4. If update weights in opposite direction from gradient, loss will be a little less every time:

1. Draw a batch of training samples x & corresponding targets `y_true`.
2. Run model on x to obtain predictions `y_pred` (this is called *forwards pass*).
3. Compute loss of model on batch, a measure of mismatch between `y_pred, y_true`.
4. Compute gradient of loss w.r.t. model's parameters (called *backward pass*).
5. Move parameters a little in opposite direction from gradient – e.g., `W -= learning_rate * gradient` – thus reducing loss on batch a bit. *Learning rate* (`learning_rate` here) would be a scalar factor modulating "speed" of gradient descent process.

Easy enough! What just described is called *mini-batch stochastic gradient descent* (mini-batch SGD). Term *stochastic* refers to fact: each batch of data is drawn at random (*stochastic* is a scientific synonym of *random*). Fig. 2.18: SGD down a 1D loss curve (1 learnable parameter) illustrates what happens in 1D, when model has only 1 parameter & have only 1 training sample.

Intuitively important to pick a reasonable value for `learning_rate` factor. If it's too small, descent down curve will take many iterations, & it could get stuck in a local minimum. If `learning_rate` is too large, your updates may end up taking you to completely random locations on curve.

Note: a variant of mini-batch SGD algorithm would be to draw a single sample & target at each iteration, rather than drawing a batch of data. This would be *true*SGD (as opposed to *mini-batch*SGD). Alternatively, going to opposite extreme, could run every step on *all* data available, called *batch gradient descent*. Each update would then be more accurate, but far more expensive. Efficient compromise between these 2 extremes: us mini-batches of reasonable size.

Although Fig. 2.18 illustrates gradient descent in a 1D parameter space, in practice use gradient descent in highly dimensional spaces: every weight coefficient in a neural network is a free dimension in space, & there may be 10s of thousands or even millions of them. To help build intuition about loss surfaces, can also visualize gradient descent along a 2D loss surface, as shown in Fig. 2.19: Gradient descent down a 2D loss surface (2 learnable parameters).. But can't possibly visualize what actual process of training a neural network looks like – can't represent a 1000000-dimensional space in a way that makes sense to humans. As such, good to keep in mind: intuitions develop through these low-dimensional representations may not always be accurate in practice. This has historically been a source of issues in world of DL research.

Additionally, there exist multiple variants of SGD that differ by taking into account prev weight updates when computing next weight update, rather than just looking at current value of gradients. There is, e.g., SGD with momentum, as well as Adagrad, RMSprop, & several others. Such variants are known as *optimization methods* or *optimizers*. In particular, concept of *momentum*, which is used in many of these variants, deserves your attention. Momentum addresses 2 issues with SGD: convergence speedh & local minima. Configure Fig. 2.20: A local minimum & a global minimum, which shows curve of a loss as a function of a model parameter.

Around a certain parameter value, there is a *local minimum*: around that point, moving left would result in loss increasing, but so would moving right. If parameter under consideration were being optimized via SGD with a small learning rate, optimization process could get stuck at local minimum instead of making its way to global minimum.

Can avoid such issues by using momentum, which draws inspiration from physics. A useful mental image here: think of optimization process as a small ball rolling down loss curve. If it has enough momentum, ball won't get stuck in a ravine & will end up at global minimum. Momentum is implemented by moving ball at each step based not only on current

slope value (current acceleration) but also on current velocity (resulting from past acceleration). In practice, i.e., updating parameter `w` based not only on current gradient value but also prev parameter update, e.g. in this naive implementation:

```
past_velocity = 0.
momentum = 0.1
while loss > 0.01:
    w, loss, gradient = get_current_parameters()
    velocity = past_velocity * momentum - learning_rate * gradient
    w = w + momentum * velocity - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)
```

∗ 2.4.4. Chaining derivatives: Backpropagation algorithm. In preceding algorithm, casually assumed: because a function is differentiable, can easily compute its gradient. But is that true? How can compute gradient of complex expressions in practice? In 2-layer model started chap with, how can get gradient of loss w.r.t. weights? That's where *Backpropagation algorithm* comes in.

· Chain rule. Backpropagation is a way to use derivatives of simple operations (e.g. addition, relu, or tensor product) to easily compute gradient of arbitrarily complex combinations of these atomic operations. Crucially, a neural network consists of many tensor operations chained together, each of which has a simple, known derivative. E.g., model defined in listing 2.2 can be expressed as a function parameterized by variables `W1, b1, W2, b2` (belonging to 1st & 2nd `Dense` layers resp.), involving atomic operations `dot, relu, softmax, & +`, as well as our loss function `loss`, which are all easily differentiable:

```
loss_value = loss(y_true, softmax(dot(relu(dot(inputs, W1) + b1), W2) + b2))
```

Calculus tells: such a chain of functions can be derived using *chain rule*.
Consider 2 functions f, g, as well as composed function fg s.t. `fg(x) == f(g(x))`:

```
def fg(x):
    x1 = g(x)
    y = f(x1)
    return y
```

Then chain rule states: `grad(y, x) == grad(y, x1) * grad(x1, x)`. This enables to compute derivative of `fg` as long as know derivatives of f, g. Chain rule is named as it is because when add more intermediate functions, it starts looking like a chain:

```
def fghj(x):
    x1 = j(x)
    x2 = h(x1)
    x3 = g(x2)
    y = f(x3)
    return y

grad(y, x) == (grad(y, x3) * grad(x3, x2) * grad(x2, x1) * grad(x1, x))
```

Applying chain rule to computation of gradient values of a neural network gives rise to an algorithm called *backpropagation*. See how that works, concretely.

· Automatic differentiation with computation graphs. A useful way to think about backpropagation is in terms of *computation graphs*. A computation graph is data structure at heart of TensorFlow & DL revolution in general. It's a directed acyclic graph of operations – in our case, tensor operations. E.g., Fig. 2.21: Computation graph representation of 2-layer model shows graph representation of 1st model.
Computation graphs have been an extremely successful abstraction in CS because they enable us to *treat computation as data*: a computable expression is encoded as a machine-readable data structure that can be used as input or output of another program. E.g., could imagine a program that receives a computation graph & returns a new computation graph that implements a large-scale distributed version of same computation – i.e. could distribute any computation without having to write distribution logic yourself. Or imagine a program that receives a computation graph & can automatically generate derivative of expression it represents. Much easier to do these things if your computation is expressed as an explicit graph data structure rather than, say, lines of ASCII characters in a `.py` file.
To explain backpropagation clearly, look at a really basic example of a computation graph (Fig. 2.22: A basic example of a computation graph.). Consider a simplified version of Fig. 2.21, where only have 1 linear layer & where all variables are scalar. Take 2 scalar variables `w, b`, a scalar input `x`, & apply some operations to them to combine them into an output `y`. Finally, apply an absolute value error-loss function: `loss_val = abs(y_true - y)`. Since want to update `w, b` in a way that will minimize `loss_val`, interested in computing `grad(loss_val, b), grad(loss_val, w)`.
Set concrete values for "input nodes" in graph, i.e., input `x`, target `y_true, w, b`. Propagate these values to all nodes in graph, from top to bottom, until reach `loss_val`. This is *forward pass* (Fig. 2.23: Running a forward pass).

"Reverse" graph: for each edge in graph going from `A` to `B`, will create an opposite edge from `B` to `A`, & ask, how much does `B` vary when `A` varies? I.e., what is `grad(B, A)`? Annotate each inverted edge with this value. This backward graph represents *backward pass* (Fig. 2.24: Running a backward pass). Have:

1. `grad(loss_val, x2) = 1`, because as `x2` varies by an amount epsilon, `loss_val = abs(4 - x2)` varies by same amount.
2. `grad(x2, x1) = 1`, because as `x1` varies by an amount epsilon, `x2 = x1 + b = x1 + 1` varies by same amount.
3. `grad(x2, b) = 1`, because as `b` varies by an amount epsilon, `x2 = x1 + b = 6 + b` varies by same amount.
4. `grad(x1, w) = 2`, because as `w` varies by an amount epsilon `x1 = x * w = 2 * w` varies by `2 * epsilon`.

What chain rule says about this backward graph: can obtain derivative of a node w.r.t. another node by *multiplying derivatives for each edge along path linking 2 nodes.*
E.g., `grad(loss_val, w) = grad(loss_val, x2) * grad(x2, x1) * grad(x1, w)` (Fig. 2.25: Path from `loss_val` to `w` in backward graph). By applying chain rule to graph, obtain what were looking for:

```
grad(loss_val, w) = 1 * 1 * 2 = 2
grad(loss_val, b) = 1 * 1 = 1
```

Note 3. *If there are multiple paths linking 2 models of interest,* `a, b`*, in backward graph, would obtain* `graph(b, a)` *by summing contributions of all paths.*

& with that, just saw backpropagation in action! Backpropagation is simply application of chain rule to a computation graph. There's nothing more to it. Backpropagation starts with final loss value & works backward from top layers to bottom layers, computing contribution that each parameter had in loss value. That's where name "backpropagation" comes from: "back propagate" loss contributions of different nodes in a computation graph.
Nowadays people implement neural networks in modern frameworks that are capable to *automatic differentiation*, e.g. TensorFlow. Automatic differentiation is implemented with kind of computation graph just seen. Automatic differentiation makes it possible to retrieve gradients of arbitrary compositions of differentiable tensor operations without doing any extra work besides writing down forward pass. When Chollet wrote his 1st neural networks in C in 2000s, had to write gradients by hand. Now, thanks to modern automatic differentiation tools, never have to implement backpropagation yourself. Consider yourself lucky!

· Gradient tape in TensorFlow. API through which you can leverage (đòn bẩy) TensorFlow's powerful automatic differentiation capabilities is `GradientTape`. It's a Python scope that will "record" tensor operations that run inside it, in form of a computation graph (sometimes called a "tape"). This graph can then be used to retrieve gradient of any output w.r.t. any variable or set of variables (instances of `tf.Variable` class). A `tf.Variable` is a specific kind of tensor meant to hold mutable state – e.g., weights of a neural network are always `tf.Variable` instances.

```
import tensorflow as tf
x = tf.Variable(0.)
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)
```

`GradientTape` works with tensor operations:

```
x = tf.Variable(tf.random.uniform((2, 2)))
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)
```

It works works with list of variables:

```
W = tf.Variable(tf.random.uniform((2, 2)))
b = tf.Variable(tf.zeros((2,)))
x = tf.random.uniform((2, 2))
with tf.GradientTape() as tape:
    y = tf.matmul(x, W) + b
grad_of_y_wrt_W_and_b = tape.gradient(y, [W, b])
```

◦ 2.5. Looking back at 1st example. Should now have a general understanding of what's going on behind scenes in a neural network. What was a magical black box at start of chap has turned into a clearer picture, as illustrated in Fig. 2.26: Relationship between network, layers, loss function, & optimizer.: model, composed of layers that are chained together, maps input data to predictions. Loss function then compares these predictions to targets, producing a loss value: a measure of how well model's predictions match what was expected. Optimizer uses this loss value to update model's weights.
Go back to 1st example in this chap & review each piece of it in light of what learned since. This was input data:

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
```

```
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255
```

Now understand: input images are stored in NumPy tensors, which are here formatted as `float32` tensors of shape `(60000, 784)` (training data) & `(10000, 784)` (test data) resp. This was our model:

```
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

Now understand: this model consists of a chain of 2 `Dense` layers, that each layer applies a few simple tensor operations to input data, & these operations involve weight tensors. Weight tensors, which are attributes of layers, are where *knowledge* of model persists. This was model-compilation step:

```
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

Now understand: `sparse_categorical_crossentropy` is loss function used as a feedback signal for learning weight tensors, & which training phase will attempt to minimize. Also know that this reduction of loss happens via mini-batch stochastic gradient descent. Exact rules governing a specific use of gradient descent are defined by `rmsprop` optimizer passed as 1st argument. This was training loop:

```
model.fit(train_images, train_labels, epochs=5, batch_size=128)
```

Now understand what happens when call `fit`: model will start to iterate on training data in mini-batches of 128 samples, 5 times over (each iteration over all training data is called an *epoch*). For each batch, model will compute gradient of loss w.r.t. weights (using Backpropagation algorithm, which derives from chain rule in calculus) & move weights in direction that will reduce value of loss for this batch.

After these 5 epochs, model will have performed 2345 gradient updates (469 per epoch), & loss of model will be sufficiently low: model will be capable of classifying handwritten digits with high accuracy.

At this point, already know most of what there is to know about neural networks. Prove it by reimplementing a simplified version of that 1st example "from scratch" in TensorFlow, step by step.

- Summary.
  - *Tensors* form foundation of modern ML systems. They come in various flavors of `dtype, rank, shape`.
  - You can manipulate numerical tensors via *tensor operations* (e.g. addition, tensor product, or element-wise multiplication), which can be interpreted as encoding geometric transformations. In general, everything in DL is amenable to a geometric interpretation.
  - DL models consist of chains of simple tensor operations, parametrized by *weights*, which are themselves tensor. Weights of a model are where its "knowledge" is stored.
  - *Learning* means finding a set of values for model's weights that minimizes a *loss function* for a given set of training data samples & their corresponding targets.
  - Learning happens by drawing random batches of data samples & their targets, & computing gradient of model parameters w.r.t. loss on batch. Model parameters are then moved a bit (magnitude of move is defined by learning rate) in opposite direction from gradient. This is called *mini-batch stochastic gradient descent*.
  - Entire learning process is made possible by fact: all tensor operations in neural networks are differentiable, & thus possible to apply chain rule of derivation to find gradient function mapping current parameters & current batch of data to a gradient value. This is called *backpropagation*.
  - 2 key concepts see frequently in future chaps are *loss* & *optimizers*. These are 2 things need to define before begin feeding data into a model.
    - *Loss* is quantity you'll attempt to minimize during training, so it should represent a measure of success for task you're trying to solve.
    - *Optimizer* specifies exact way in which gradient of loss will be used to update parameters: e.g., it could be RMSProp optimizer, SGD with momentum, & so on.

- 3. Introduction to Keras & TensorFlow. Cover: A closer look at TensorFlow, Keras, & their relationship. Setting up a DL workspace. An overview of how core DL concepts translate to Keras & TensorFlow.

  This chap is meant to give everything need to start doing DL in practice. Give a quick presentation of Keras https://keras.io & TensorFlow https://tensorflow.org, Python-based DL tools used throughout book. Find out how to set up a DL workspace, with TensorFlow, Keras, & GPU support. Finally, building on top of 1st contact had with Keras & TensorFlow in Chap. 2, review core components of neural networks & how they translate to Keras & TensorFlow APIs.

  By end of this chap, ready to move on to practical, real-world applications, which will start with Chap. 4.

- 3.1. What's TensorFlow? TensorFlow is a Python-based, free, open source ML platform, developed primarily by Google. Much like NumPy, primary purpose of TensorFlow: enable engineers & researchers to manipulate mathematical expressions over numerical tensors. But TensorFlow goes far beyond scope of NumPy in following ways:
  - It can automatically compute gradient of any differentiable expression, making it highly suitable for ML.
  - It can run not only on CPUs, but also on GPUs & TPUs, highly parallel hardware accelerators.
  - Computation defined in TensorFlow can be easily distributed across many machines.
  - TensorFlow programs can be exported to other runtimes, e.g. C++, JavaScript (for browser-based applications), or TensorFlow Lite (for applications running on mobile devices or embedded devices), etc. This makes TensorFlow applications easy to deploy in practical settings.

  Important to keep in mind: TensorFlow is much more than a single library. It's really a platform, home to a vast ecosystem of components, some developed by Google & some developed by 3rd parties. E.g., there's TF-Agents for reinforcement-learning research, TFX for industry-strength ML workflow management, TensorFlow Serving for production deployment, & there's TensorFlow Hub repository of pretrained models. Together, these components cover a very wide range of use cases, from cutting-edge research to large-scale production applications.

  TensorFlow scales fairly well: e.g., scientists from Oak Ridge National Lab have used it to train a 1.1. exaFLOPS extreme weather forecasting model on 27000 GPUs of IBM Summit supercomputer. Likewise, Google has used TensorFlow to develop very compute-intensive DL applications, e.g. chess-playing & Go-playing agent AlphaZero. For own models, if have budget, can realistically hope to scale to around 10 petaFLOPS on a small TPU pod or a large cluster of GPUs rented on Google Cloud or AWS. That would still be around 1% of peak compute power of top supercomputer in 2019!
- 3.2. What's Keras? Keras is a DL API for Python, built on top of TensorFlow, that provides a convenient way to define & train any kind of DL model. Keras was initially developed for research, with aim of enabling fast DL experimentation.

  Through TensorFlow, Keras can run on top of different types of hardware (Fig. 3.1: Keras & TensorFlow: TensorFlow is a low-level tensor computing platform, & Keras is a high-level DL API) – GPU, TPU, or plain CPU – & can be seamlessly scaled to thousands of machines.

  Keras is known for prioritizing developer experience. It's an API for human beings, not machines. It follows best practices for reducing cognitive load: it offers consistent & simple workflows, it minimizes number of actions required for common use cases, & it provides clear & actionable feedback upon user error. This makes Keras easy to learn for a beginner, & highly productive to use for an expert.

  Keras has well over a million users as of late 2021, ranging from academic researchers, engineers, & data scientists at both startups & large companies to graduate students & hobbyists. Keras is used at Google, Netflix, Uber, CERN, NASA, Yelp, Instacart, Square, & hundreds of startups working on a wide range of problems across every industry. YouTube recommendations originate from Keras models. Waymo self-driving cars are developed with Keras models. Keras is also a popular framework on Kaggle, ML competition website, where most DL competitions have been won using Keras.

  Because Keras has a large & diverse user base, it doesn't force you to follow a single "true" way of building & training models. Rather, it enables a wide range of different workflows, from very high level to very low level, corresponding to different user profiles. E.g., have an array to ways to build models & an array of ways to train them, each representing a certain trade-off between usability & flexibility. In Chap. 5, review in detail a good fraction of this spectrum of workflows. Could be using Keras like would use Scikit-learn – just calling `fit()` & letting framework do its thing – or could be using it like NumPy – taking full control of every little detail.

  I.e., everything learning now as getting started will still be relevant once become an expert. Can get started easily & then gradually dive into workflows where writing more & more logic from scratch. Won't have to switch to an entirely different framework as go from student to researcher, or from data scientist to DL engineer.

  This philosophy is not unlike that of Python itself! Some languages only offer 1 way to write programs – e.g., OOP or functional programming. Meanwhile, Python is a multiparadigm language: it offers an array of possible usage patterns that all work nicely together. This makes Python suitable to a wide range of very different use cases: system administration, DS, ML engineering, web development ... or just learning how to program. Likewise, can think of Keras as Python of DL: a user-friendly DL language that offers a variety of workflows to different user profiles.
- 3.3. Keras & TensorFlow: A brief history. Keras predates TensorFlow by 8 months. It was released in Mar 2015, & Tensorflow was released in Nov 2015. May ask if Keras is built on top of TensorFlow, how it could exist before TensorFlow was released? Keras was originally built on top of Theano, another tensor-manipulation library that provided automatic differentiation & GPU support – earliest of its kind. Theano, developed at Montréal Institute for Learning Algorithms (MILA) at the Université de Montréal, was in many ways a precursor (tiền thân) of TensorFlow. It pioneered idea of using static computation graphs for automatic differentiation & for compiling code to both CPU & GPU.

  In late 2015, after release of TensorFlow, Keras was refactored to a multibackend architecture: it became possible to use Keras with either Theano or TensorFlow, & switching between 2 was as easy as changing an environment variable. By Sep 2016, TensorFlow had reached a level of technical maturity where it became possible to make it default backend option for Keras. In 2017, 2 new additional backend options were added to Keras: CNTK (developed by Microsoft) & MXNet (developed by Amazon). Nowadays, both Theano & CNTK are out of development, & MXNet is not widely used outside of Amazon. Keras is back to being a single-backend API – on top of TensorFlow.

  Keras & TensorFlow have had a symbiotic relationship for many years. Throughout 2016 & 2017, Keras became well known as user-friendly way to develop TensorFlow applications, funneling new users into TensorFlow ecosystem. By late 2017, a

majority of TensorFlow users were using it through Keras or in combination with Keras. In 2018, TensorFlow leadership picked Keras & TensorFlow's official high-level API. As a result, Keras API is front & center in TensorFlow 2.0, released in Sep 2019 – an extensive redesign of TensorFlow & Keras that takes into account > 4 years of user feedback & technical progress.

By this point, must be eager to start running Keras & TensorFlow code in practice.

- 3.4. Setting up a DL workspace. Before can get started developing DL applications, need to set up your development environment. It's highly recommended, although not strictly necessary, that run DL code on a modern NVIDIA GPU rather than computer's CPU. Some applications – in particular, image processing with convolutional networks – will be excruciatingly slow (chậm kinh khủng) on CPU, even a fast multicore CPU. & even for applications that can realistically be run on CPU, generally see speed increase by a factor of 5 or 10 by using a recent GPU.

  To do DL on a GPU, have 3 options:

  * Buy & install a physical NVIDIA GPU on your workstation.
  * use GPU instances on Google Cloud or AWS EC2.
  * Use free GPU runtime on Colaboratory, a hosted notebook service offered by Google.

  Colaboratory is easiest way to get started, as it requires no hardware purchase & no software installation – just open a tab in browser & start coding. It's the opinion recommend for running code examples in this book. However, free version of Colaboratory is only suitable for small workloads. If want to scale up, have to use 1st or 2nd option.

  If don't already have a GPU that can use for DL (a recent, high-end NVIDIA GPU), then running DL experiments in cloud is a simple, low-cost way to move to larger workloads without having to buy any additional hardware. If develop using Jupyter notebooks, experience of running in cloud is no different from running locally.

  But if a heavy user of DL, this setup isn't sustainable in long term (bền vững trong dài hạn) – or even for more than a few months. Cloud instances aren't cheap: pay $2.48 per hour for a V100 GPU on Google Cloud in mid-2021. Meanwhile, a solid consumer-class GPU will cost you somewhere between $1500–$2500 – a price that has been fairly stable over time, even as specs of these GPUs keep improving. If a heavy user of DL, consider setting up a local workstation with 1 or more GPUs.

  Additionally, whether running locally or in cloud, better to be using a Unix workstation. Although it's technically possible to run Keras on Windows directly, don't recommend it. If a Windows user wants to do DL on own workstation, simplest solution to get everything running is to set up an Ubuntu dual boot on your machine, or to leverage Windows Subsystem for Linux (WSL), a compatibility layer that enables to run Linux applications from Windows. It may seem like hassle, but it will save you a lot of time & trouble in long run.

  * 3.4.1. Jupyter notebooks: preferred way to run DL experiments. Jupyter notebooks are a great way to run DL experiments – in particular, many code examples in this book. They're widely used in DS & ML communities. A *notebook* is a file generated by Jupyter Notebook app `https://jupyter.org` that can edit in browser. It mixes ability to execute Python code with rich text-editing capabilities for annotating what doing. A notebook also allows to break up long experiments into smaller pieces that can be executed independently, which makes development interactive & means don't have to rerun all of your prev code if sth goes wrong late in an experiment.
    Recommend using Jupyter notebooks to get started with Keras, although that isn't a requirement: can also run standalone Python scripts or run code from within an IDE e.g. PyCharm. All code examples in this book are available as open source notebooks; can download: `github.com/fchollet/deep-learning-with-python-notebooks`.
  * 3.4.2. Using Colaboratory. Colaboratory (or Colab for short) is a free Jupyter notebook service that requires no installation & runs entirely in cloud. Effectively, it's a web page that lets you write & execute Keras scripts right away. It gives access to a a free (but limited) GPU runtime & even a TPU runtime, so don't have to buy your own GPU. Colaboratory is what recommend for running code examples in this book.
    - 1st steps with Colaboratory. To get started with Colab, go to `https://colab.research.google.com` & click `New Notebook` button. See standard Notebook interface shown in Fig. 3.2: A Colab notebook. Notice 2 buttons in toolbar: `+ Code, + Text`. They're for creating executable Python code cells & annotation text cells, resp. After entering code in a code cell, Pressing Shift-Enter will execute it (Fig. 3.3: Creating a code cell)

- 3.5. 1st steps with TensorFlow.

- 4. Getting started with neural networks: Classification & regression.
- 5. Fundamentals of ML.
- 6. Universal workflow of ML.
- 7. Working with Keras: A deep dive.
- 8. Introduction to DL for computer vision.
- 9. Advanced DL for computer vision.
- 10. DL for timeseries.
- 11. DL for text.

- 12. Generative DL.
- 13. Best practices for real world.
- 14. Conclusions.

2.5 [DG24]. Shlomo Dubnov, Ross Greer. Deep & Shallow: Machine Learning in Music & Audio. 2023

- "*Deep & Shallow* by Shlomo Dubnov & Ross Greer is an exceptional journey into convergence of music, AI, & signal processing. Seamlessly weaving together intricate theories with practical programming activities, book guides readers, whether novices or experts, toward a profound understanding of how AI can reshape musical creativity. A true gem for both enthusiasts & professionals, this book eloquently bridges gap between foundational concepts of music information dynamics as an underlying basis for understanding music structure & listening experience, & cutting-edge applications, ushering us into future of music & AI with clarity & excitement." – Gil Weinberg, Professor & Founding Director, Georgia Tech Center for Music Technology
- "The authors make an enormous contribution, not only as a textbook, but as essential reading on music information dynamics, bridging multiple disciplines of music, information theory, & ML. Theory is illustrated & grounded in plenty of practical information & resources." – Roger B. Dannenberg, Emeritus Professor of Computer Science, Art & Music, Carnegie Mellon University
- "Deep & Shallow by Dubnov & Greer is ... a great introductory textbook on topic of ML for music & audio. Especially useful are Jupyter Notebooks, providing examples of everything covered in book. Advanced undergrads, graduate students, or working professionals with some experience in statistics & Python should find book perfect for learning basics, & recent/advanced topics e.g. DL." – Perry R. Cook, PhD, Professor (Emeritus), Princeton University Computer Science (also Music)

Deep & Shallow. Providing an essential & unique bridge between theories of signal processing, ML, & AI in music, this book provides a holistic overview (tổng quan toàn diện) of foundational ideas in music, from physical & mathematical properties of sound to symbolic representations. Combining signals & language models in 1 place, this book explores how sound may be represented & manipulated by computer systems, & how our devices may come to recognize particular sonic patterns as musically meaningful or creative through lens of information theory.

Introducing popular fundamental ideas in AI at a comfortable pace, more complex discussions around implementations & implications in musical creativity are gradually incorporated as book progresses. Each chap is accompanied by guided programming activities designed to familiarize readers with practical implications of discussed theory, without frustrations of free-form coding.

Surveying state-of-art methods in applications of deep neural networks to audio & sound computing, as well as offering a research perspective that suggests future challenges in music & AI research, this book appeals to both students of AI & music, as well as industry professionals in fields of ML, music, & AI.

Shlomo Dubnov is a Professor in Music Department & Affiliate Professor in Computer Science & Engineering at University of California, San Diego. He is best known for his research on poly-spectral analysis of musical timbre & inventing method of Music Information Dynamics with applications in Computer Audition & Machine improvisation. His previous books on *The Structure of Style: Algorithmic Approaches to Understanding Manner & Meaning* & *Cross-Cultural Multimedia Computing: Semantic & Aesthetic Modeling* were published by Springer.

Ross Greer is a PhD Candidate in Electrical & Computer Engineering at University of California, San Diego, where he conducts research at intersection of AI & human agent interaction. Beyond exploring technological approaches to musical expression, Ross creates music as a conductor & orchestrator for instrumental ensembles. Ross received his B.S. & B.A. degrees in EECS, Engineering Physics, & Music from UC Berkeley, & an M.S. in Electrical & Computer Engineering from US San Diego.

Preface. This book complements a largely missing textbook for a popular & growing in demand field of ML for Music & Audio. 1 of difficulties in covering this subject is interdisciplinary nature of domain knowledge that creates a barrier to entry to this field, which combines background in signal processing, symbolic sequence processing, statistical analysis, classical ML, & more recently quickly growing field of deep neural networks. As a result of breath of required background, students or practitioners who are interested in learning about field have to look for relevant information across multiple disciplines. Moreover, focus of book is on generative methods, which are usually considered as more advanced subjects in ML that are often not explained in introductory or basic-level courses on ML.

Book is intended for upper-division undergraduate- or graduate-level courses, with basic understanding of statistics & Python programming. Music concepts & basic concepts in analysis of audio signals will be covered in book, providing a 1-stop reference to questions of musical representation & feature extraction, without assuming extensive prior musical & acoustics knowledge.

Starting with representation aspects in audio & symbolic music in Chap. 1, 2nd chap goes immediately into stochastic aspects in both music & audio, addressing conceptual premise of thinking about relations between noise & structure. Idea of composition using random dice with lookup tables, or so-called "Mozart Dice Game", shows early on how concepts are related, also providing historical context & relation to later formal algorithmic methods.

Although symbolic music vs. audio signals are capturing sound data at different timescales & levels of detail, parallels between 2 domains are demonstrated throughout book by considering aspects of anticipation both in perceptual (sound vs. noise) & cognitive (events surprisal) realms. In Chap. 2, introduce a unifying analysis framework called "Music Information

Dynamics" that stems from conceptual formulation of music & audio modeling & understanding as a problem of information processing. Introducing Markov models as basic statistical models for text & music, proceed to consider information dynamics for case of Markov models, relating it to questions of what type of random processes generates music that are more or less interesting to a predictive listener.

Accordingly, concepts from information theory & Shannon's laws are covered in Chap. 3. Central idea of thinking about music as an information source forms basis to way this book approaches & represents musical ML problem. Using "off the shelf" data compression methods, one can build efficient representations of musical style. Demonstrate how style learning & continuation of musical phrases in a particular style can be accomplished by traversing a codebook that was derived from famous Lempel-Ziv compression method.

In Chap. 4, look deeper into properties of audio signals by introducing basic mathematical tools of digital signal processing. After briefly introducing concepts of sampling, talk about frequency analysis & conditions for short-time Fourier transform (STFT) prefect reconstruction. Such signal reconstruction is not possible when only magnitude of STFT is available, so cover Griffin-Lim phase reconstruction that is often used for generating audio (also called vocoding) from magnitude spectrograms. Conclude chap with discussion of source-filter models & their application to speech modeling. Finally, use concept of spectral analysis &, in particular, idea of spectral flatness to distinguish noise vs. structure in audio, relating it to questions of prediction & information dynamics.

More advance information theory concepts are developed in Chap. 5, as part of concatenative & recombinant audio generative methods. Using a symbolization process that renders audio into a sequence of symbolic representation, known as Audio Oracle, demonstrate use of Variable-Memory Markov models & string-matching compression algorithms for creating new sequences of audio features & audio descriptors. These new symbolic sequences are then rendered back into audio using concatenative method. Problem of finding repetitions in audio data & use of recopying for compression & prediction are also linked to questions of computational aesthetics, namely aesthetics measures of Birkhoff & Bense as ratios of order & complexity, as well as ethical aspects of reuse of existing music in training generative models.

Chap. 6 serves as a bridge between signal processing & statistical methods of earlier chaps & materials in later chaps that are dealing with deep neural networks for music & audio. After introducing basic building blocks of artificial neural networks, discuss aspects of representation learning by looking into similarities & differences between auto-encoder & a classical statistical data reduction & representation method known as *principal components analysis* (PCA). Specifically, use of a PCA-like method for finding an optimal transform of audio signals, known as Krahunen-Loeve transform, is demonstrated in context of noise removal.

Chaps. 7–10 cover deep neural network approaches to music, starting in Chap. 7 with question of unsupervised representation learning using sequence & image models that use Recurrent Neural Networks (RNN) & Convolutional Neural Networks (CNN). Idea of tokenizations is basic to ability of capturing complex sequential features in music into a neural recurrent representation. Relations across both time & pitch or frequency are captured using CNN approach.

In Chap. 8, return to fundamental concept of turning noise into structure through notion of neural networks that imagine. Concept of generative models is introduced by extending previously discussed autoencoder into a Variational Autoencoder (VAE) that allows probabilistic approximation & representation. In this view, random seed or input to generative networks results in generation of variable replicas or samples from same statistical model of music that was learned by neural network. This chap also covers Generative Adversarial Networks, typing representation learning of VAE to broader generative tasks, & also introducing conditional generation methods similar to style transfer in images. Such methods are important for specifying constraints on generative processes in order to produce music or audio that correspond to user specifications or other contextual input.

Chap. 9 covers recently extremely successful Transformers models, relating them to problem of finding repetition & memory structures that were encountered earlier in Variable Markov Oracle (VMO) of Chap. 5 & RNN in Chap. 7. Several state-of-art transformer models are surveyed here both in audio & symbolic representations.

Book concludes by reviewing & summarizing broad problem of viewing music as a communication process in Chap. 10. By relating phenomena of music understanding & creation to models of music information dynamics, able to discuss future of AI in music in terms of building new compositional tools that operate by learning musical structures from examples & generating novel music according to higher level meta-creative specifications, approaching novel levels of man-machine co-creative interaction.

Interactive code exercises are an important feature of this book. Instructor solutions are available by request through publisher's book website. Referenced code will will be available as Jupyter Notebooks at https://github.com/deep-and-shallow/notebooks.

Authors provide following possible suggestions for curriculum of an introductory & advanced course on ML for Music & Audio. Focus of 1st course is an introduction to basic concept of music & audio representation, modeling music as a probabilistic information source, music information dynamics, & an introduction to representation learning & sequence modeling using neural networks.

- Class 1: Representation of Sound & Music Data (MIDI, audio), Audio features, MFCC, Chroma
- Class 2: Aleatoric music, stochastic processes in music & sound (Mozart Dice Game), Noise & Electronic Music
- Class: Sampling, Spectral Analysis, Fourier transform, FFT, Spectral Flatness
- Class 4: Short-Time Fourier Analysis, Perfect Reconstruction (COLA), Griffith-Lim phase reconstruction
- Class 5: Information Theory & Music, String Matching, Universal Compression & Prediction
- Class 6: Markov Models for Text & Music, Lempel-Ziv Algorithm for Motif Continuation

- Class 7: Introduction to Neural Networks, Neural Network Models of Music
- Class 8: Autoencoder (AE) & PCA, Representation Learning
- Class 9: Neural Language Models, Recurrent Neural Network for Music
- Class 10: Introduction to Generative Models & next course overview

Advanced 2nd course dives deeper into signal processing of speech signals & vocoders, generative neural networks for audio & music with focus on variational methods, & return to information theory for deep musical information dynamics with possible implications toward understanding creativity in neural networks.

- Class 1: Review of Music & Audio Representation, Short-Time Fourier Analysis
- Class 2: Linear Filters & Convolution Theorem, Pole-Zero Plots
- Class 3: History of voder/vocoder, Source-Filter Models, & Linear Prediction
- Class 4: Concatenative Synthesis, Variable Markov Oracle (VMO), Symbolization & Information Dynamics
- Class 5: Review of Neural Networks, AutoEncoders & RNN
- Class 6: Convolutional NN for Music & Audio (U-Net) for Genre Classification & Source Separation
- Class 7: Variational AE, ELBO, Generating Music with VAE
- Class 8: Generative Adversarial Networks, W-GAN, MuseGAN, CycleGAN
- Class 9: Attention & Transformers, MuseNet, Audio Spectrogram Transformer
- Class 10: Information Theory Models of NN, Deep Music Information Dynamics

Including extra time for quizzes, projects, & exams, these course contents should suffice for a semester-long course each, or as 2 quarters of instruction followed by a quarter of practical projects. Overall, estimate: with additional practical & theoretical assignments, materials of book represent about 1 year of academic study that is needed for students with focus on AI & music or a concentration in ML with specialization in music & audio. Hope: current text would help establish this as a field of study.

- Chap. 1. Introduction to Sounds of Music. Music is a particular form of art that is particularly hard to grasp, define, or put in a mold. In many aspects, music deals with shaping air vibrations into meaningful sound perceptions, & each type of music, depending on culture, style, period, or location, has its own particular ways of expression that might be grounded in practices that are close to hearts of some & meaningless to others. In this book, trying to understand music from AI & ML perspectives. To make question about meaning of music even more acute, may ask ourselves – what does it take to make music meaningful to a machine? If desire to build artificial agents that can compose music or engage creatively in real time with musicians in a process of improvisation, hope: these interactions will convey to intelligent machine same significance we ascribe to music (gán cho âm nhạc).
  - 1.1. From Sound to Features. Broadly speaking, meaning of music is established by human musicians themselves who have creative intent of producing particular types of sound structures, & by their listeners who engage with these sounds to experience them in interesting & exciting ways. Music, in many ways, is most abstract art, devoid of a particular reference to outside world. It may excite us or calm us down, affect our mood or be simply regarded as an intricate fabric of sound structures, where pleasure can be found in act of listening itself. Accordingly, asking question of what establishes meaning in sounds is a formidable problem to which we unfortunately will not be able to give a definite answer.

    In journey from physical sounds to musical meaning, need to go through several intermediate stages. 1st steps is choosing palette of music sounds at our disposal, which can be later organized in time into a complete musical piece. Since 1 person's music may be another person's noise, challenge for machine is very much same as that of uninitiated listener: to 1st of all distinguish noise from music or find structure in sounds that makes it musical. In journey into creative musical machines, start with sound structures that are most common in music, namely notes, rhythms, & scales. Most elementary musical sounds, often called "tones" (tông màu) comprise duration (length), frequency (pitch), amplitude (loudness), & timbre (âm sắc) (sound quality). Tones are organized in patterns, where times & lengths of their appearances (onsets & durations) are organized into rhythm; selection of allowed frequencies is organized into scales & tonalities; multiple notes playing together create harmonies; & so on. Each of these aspects can be considered as a musical representation; it summarizes a physical phenomena – sound – into few essential properties that are considered musical. Since this is not a music theory book, will not go into elaborate theories of how music is represented or "rules" for musical composition that depend also on musical style & culture, but will rather stay as close as possible to examples in representation related to common Western music notation (Note: many of these techniques can be applied to analysis in other music notation schemes, given appropriate methods of tokenization & abstraction.) & representations that can be extracted from a sound recording before further processing to discover statistical relations by ML methods. Accordingly, may summarize basic sound properties or features as follows:

* **Pitch** is a perceptual phenomenon which encapsulates relationship between frequency generated by an instrument & frequency perceived by listener. E.g., when musicians in a modern orchestra tune to concert pitch, they typically listen to an oboe play pitch A4. This pitch is produced at a frequency of 440 Hertz (Hertz: unit used to measure frequency, referring to occurrences per sec of any periodic phenomena.). Slight changes in frequency have little effect on perceived pitch. Perception of pitch varies logarithmically with generated frequency; i.e., to hear same pitch at a higher octave, frequency of sound must be doubled.
  Tuning systems explore problem of mapping a discrete number of possible notes to continuous frequency space. E.g., a piano has a limited number of keys spanning an octave, but there are infinite number of frequencies which theoretically exist between any set of notes. Western music traditionally uses a 12-tone equal-tempered scale. In an *equal-tempered scale*, every pair of adjacent notes has an identical frequency ratio. Thus, each octave is divided into 12 logarithmically equal parts.

  Example 1 (Notes on a Keyboard)**.** *Musical Instrument Digital Interface, or MIDI, is a protocol for recording & playing music on computers. Lowest note on a standard acoustic piano, named A0, is pitch number 21 in MIDI convention. Highest note on a keyboard, named C8, is pitch number 108 in MIDI. By defining note named A4 (MIDI pitch 69) to be reference pitch at 440 Hz, can define following relationship between any MIDI pitch number p & its corresponding frequency:*

$$F_{\text{pitch}}(p) = 2^{\frac{p-69}{12}} \cdot 440.$$

  To interpret this equation, note: exponential term effectively counts how many steps of an 12-step-octave input note is from reference pitch, then uses this as (fractional) power due to logarithmic relationship between perceived notes & requisite frequency. This coefficient is then used as a multiple to reference pitch of 440 Hz. Can see: a pitch located 12 half step[8]
* **Timbre**, also referred to as tone quality or tone color, is quality of musical sound that distinguishes different types of sound production, e.g. different voices or instruments. When hear same note played at same volume & same duration 1st by 1 instrument & then by another, difference you hear is due to timbre. Physically, this property depends on energy distribution in harmonics & relationships between constituents sinusoids, phenomena introduced in Chap. 4.

As a prototype of sound, consider a sinusoidal wave (i.e., a wave with a fixed frequency, phase, & amplitude). Period is time it takes for wave cycle to repeat & can be measured using successive peaks in wave (unit: secs), & is inverse of frequency (unit: Hertz). Amplitude is 1-half pressure difference between high & low points (often measured in micro Pascals).

* 1.1.1. Loudness. Loudness is subjective perception of sound pressure, i.e., loudness is a combination of not only physical attributes of a sound but also physiological & psychological phenomena.
  Torben Poulsen [1] shows: physical aspect of loudness is related to sound pressure level, frequency content, & sound duration. *Sound pressure level* is used to measure effective pressure of a sound relative to a reference pressure value (often 20 μPa, a canonical threshold for human hearing). Sound pressure level is described by formula

$$20 \log_{10} \frac{p}{p_0},$$

  where p: measured sound pressure & p_0: reference pressure, giving a unit of decibels (dB).
  Steven's power law, formulated as $\Psi(I) = kI^{\alpha}$, connects sound pressure levels I to perceived loudness of a single tone; i.e., it defines relationship between a change in stimulus sound pressure level & corresponding increase in perceived sensation. Steven's power law typically uses an exponent α of 0.67, & this exponent & scalar constant k are empirically sourced approximations; accordingly, there exist other methods & models to relate sound pressure levels to perception. Another interesting perceptual phenomena is relationship between stimulus frequency & perceived loudness, portrayed by a so-called *equal-loudness graph* as shown in Fig. 1.1: Equal-loudness graph created by Suzuki & Takeshima [2] shows: different frequencies create same perceived loudness at different sound pressure levels. Following any 1 curve on graph provides frequency-SPL pairs which were found to create same perceived loudness.
* 1.1.2. MFCC. In field of timbral analysis, i.e., extracting sound properties that are unrelated to pitch, duration, or loudness, Mel-Frequency Cepstral Coefficients (MFCC) became "go-to" method for capturing overall shape of frequency distribution, also called *spectral envelope*. Spectral envelope may characterize color of sound that is independent of specific note an instrument plays, which is useful in distinguishing between different types of instruments. MFCCs have historically been dominant features in speech recognition since they are invariant to pitch or loudness of speech & rather capture broad shape of energy distribution of voice (that comprises so-called "speech formants": Voice as a combination of speech formants is introduced in Chap. 4.). Moreover, MFCC analysis has been applied to complex music e.g. polyphonic music (multiple voices or multiple instruments, e.g. orchestra) to characterize distribution of energies at different frequencies. E.g., they were used in music retrieval systems, music summarization, & musical audio analysis toolboxes (Musical audio analysis toolboxes e.g. Librosa are introduced in Appendix B.).
  A detailed description of MFCC requires technical terms that assume knowledge of spectral analysis, which is using Fourier transform to discover underlying frequency components in a sound signal. While Fourier analysis is introduced in next

[8] Here must acknowledge some possible confusion in terminology: though octave is divided into 12 portions, musicians refer to these portions as *half steps*. 2 half steps span a *whole step*, & sequential combinations of half & whole steps are used to define different *scales*. Musical term *flat* refers to a pitch 1 half step below reference, while term *sharp* refers to a pitch 1 half step above reference. This provides a variety of possible descriptors for same pitch, referred to as an *enharmonic* relationship; e.g., C♯ (C-sharp) & D♭ (D-flat) refer to same pitch. Nuances in use of these symbols can provide context to a performer about special tuning to play within a particular modes or harmony, but for sake of our examples, can consider these pitches equivalent & thus must be careful that our schemes for abstracting these pitches do not mistakenly contain 2 possible table entries for same pitch.

chap, what is important to note at this point: each step in process of MFCC computation is motivated by perceptual or computational considerations. Briefly, it 1st converts Fourier analysis, which is linear in frequency, to a Mel-scale that is perceptually scaled according to human judgments of pitches being equal distance from each other. I.e., when comparing distances between any 2 points on Mel scale, same distance should correspond to same perceived change in pitch. Within linear frequency scale, this relationship is not preserved since relationship of frequency to perceived pitch is logarithmic. A common formula for relating Mel (a unit whose name is derived from "melody" [3]) to frequency in Hertz (Hz) is

$$m = 2595 \log_{10} \left(1 + \frac{f}{700}\right).$$

After that, a technique called "cepstral analysis" is applied to scaled spectrum. Idea of cepstral analysis: separate short-time structure vs. longer period repetitions in sounds by assuming a "source-filter" model. By looking at spectral analysis of a sound, one notices: there are harmonics, or partials showing as individual peaks, & an overall "spectral envelope" that shapes spectrum in terms of broad areas of resonances. In source-filter view, spectral peaks of harmonics originate from repetitions of excitation source, e.g. flapping of vocal chords in speech or vibrations of reed or string in a musical instrument. Broad shape of spectrum is attributed to filter, representing resonance of body of instrument or speech formants in a vocal tract. Operation of a filter on another signal amounts to an operation of multiplication in frequency domain between filter's & signal's spectra, resp. In cepstral analysis, a log function is applied to log of amplitude-spectrum, & then a Fourier analysis is applied to split it into low-order & high-order components. Idea of cepstral analysis: log operation, at least approximately, separates aspects of source & filter into 2 additive components, one that contains harmonic without spectral envelope & the other that captures envelope. Then by applying Fourier transform result is rendered back into time-like units, called "Quefrency". By going back to a time-like domain, spectral envelope is translated into lower cepstral coefficients (low quefrency), while harmonic structure due to pitch shows up peaks at higher quenfrencies. Eventually, only lower cepstral coefficients (filter characteristics) are kept as features, as they capture broad shape of spectrum, removing spectral shape of harmonics due to pitch (source). Formally written, cepstral analysis is defined as follows: (1.3)

$$C(q) = \left|\mathcal{F}^{-1}\left\{\log\left(|\mathcal{F}\{x(t)\}|^2\right)\right\}\right|$$

with $C(q)$ being cepstrum & different quefrencies, $x(t)$: original time signal, $\mathcal{F}, \mathcal{F}^{-1}$ being Fourier & inverse Fourier transforms, resp. A diagram of MFCC analysis is shown in Fig. 1.2: Main steps in calculation of MFCC. To obtain a frequency scale that better corresponds to human hearing, a mel-scale is used instead of linear frequency spacing. Moreover, since sound properties change over time, MFCC analysis is applied to short instances of sound by applying a windowing function that extracts a portion of a signal in time. It should be noted: total energy or overall loudness of sound is captured by 1st cepstral coefficient & is often disregarded too[9]. An investigation suggests: MFCC are approximately optimal representations, at least for case of speech & song, s they approximate a statistically optimal basis[10], derived by Principal Component Analysis (PCA) applied to a Mel-Spectra correlation matrix. Technical method of PCA will be described in Chap. 6. To summarize, vector of MFCC is a timbral feature that captures overall spectral shape of a sound in its lower coefficients, independent of aspects of pitch or energy, in approximately optimal & independent manner.

∗ 1.1.3. Chroma. Chromatic features of sound (Đặc điểm sắc độ của âm thanh) refer to phenomenon that doubling or halving frequency of sound results in a similar perceived sound, just higher or lower. A musical example of this *octave* phenomenon is shown in Fig. 1.3: 3rd movement of Beethoven's 9th Symphony famously begins with octave jumps in strings. Each set of notes begins at a high pitch, then drops an octave. From a physical standpoint, frequency of string's vibration is halved. Listen to a recording of this scherzo to hear remarkable similarity between pitches spaced an octave apart; this similarity is described as *chroma* feature.

This chromatic feature is best visualized with *chromagram*, shown in Fig. 1.4: Helix pattern of chromagram. Projections onto circle at base provide chroma perceived, while height provides reference to general frequency level associated with pitch. There is a property of chromatic invariance between pitches separated by 1 or more octaves; as move up or down helix of chromagram, projection of position onto chromatic wheel provides chroma perceived, while height indicates how "high" or "low" sound is perceived. Distance between any point & point directly above is equal to 1 octave (i.e., next time same chroma will occur). Thus, every frequency can be constructed as a combination of height & chroma:

$$f = 2^{h+c},$$

where height $h \in \mathbb{N}$ & chroma $c \in [0, 1)$. Similarly, chroma itself can be thought of as "carry" after subtracting nearest height level from frequency:

$$c = \log 2f - \lfloor \log 2f \rfloor.$$

Having an understanding of these properties & features of sound will be a useful frame for analysis of music, especially in relation to musical events we perceive as consonant or dissonant, to which may ascribe musical significance we seek to represent & recreate in our models.

– Hiểu được các đặc tính & đặc điểm của âm thanh này sẽ là một khuôn khổ hữu ích để phân tích âm nhạc, đặc biệt là liên quan đến các sự kiện âm nhạc mà chúng ta coi là hòa hợp hay bất hòa, từ đó có thể gán ý nghĩa âm nhạc mà chúng ta muốn thể hiện & tái tạo trong các mô hình của mình.

[9]Since Fourier transform performs & integral operation over a signal multiplied by a phasor at each frequency, Fourier analysis at 0 frequency is simply an integral over whole signal, which in case of (1.3) is an integral over log of energy at all frequencies, thus giving total log-energy of signal.

[10]PCA finds basis vectors that optimally represent randomly distributed vectors, e.g. sound features, in a way that is both decorrelated & sorted according to amount of variance they capture in original signal. A basis is a set of vectors that can be combined in multiples to construct any other vector within a defined vector space.

- 1.2. Representation of Sound & Music Data.
  - 1.2.1. Symbolic Representations. Though may never answer classic question of "What *is* music?", here hope to make discussion of some ways may *represent* music.

    If music is composed of sounds, might consider Western music notation to be an example of *symbolic representation* of sounds. Note: while this system of symbols is capable of representing creative ideas, this music notation is incomplete in its ability to represent every possible sound & is of course subject to interpretation by its time & cultural context. This incompleteness is a property we will see across representations, even in our "best" audio encodings. Always limited by number of bits (A *bit* (binary digit) is smallest unit of information that can be represented by a computer; conceptually, unit is an entity that can take 1 of 2 states, which can be imagined as "on" or "off", 1 or 0, "True" or "False", etc.) available to represent sonic information, so a representation which allows for perfect reconstruction of physical sound is a near-impossible task. Of course, can come close, adopting representations which can encode information to resolution & rate of human hearing.

    Though a trivial statement, note: representation scheme offered by Western music notation is, in general, sufficient to represent Western music. So, define some of its common structures & features used to describe common musical expressions, as use these structures as building blocks for composing similar music.

    - A *note* describes a pitch & duration. A very brief introduction to Western music notation is provided in Fig. 1.5: This excerpt from Tchaikovsky's Overture to Romeo & Juliet illustrates characteristic symbols used to notate Western music. Each note has a **head** which sits on a position within **staff** (name given to each series of 5 horizontal lines). Placement of head on staff indicates pitch relative to some reference, provided by **clef** (symbol sitting in left-most position of staff). Flags, beams, & stems are used to indicate note durations. A brief catalog of some note & rest duration is shown in this excerpt, ranging from whole (entire measure of time) to 16th (1-16th of measure duration). Top staff contains whole notes, 2nd staff half notes, 3rd staff 16th notes, 4th staff 8th notes, & bottom staff quarter notes. Similarly, 3rd staff contains a half rest, 4th staff an 8th rest, & bottom staff a quarter rest. While this fractional notation of duration is in theory unbounded, beyond a certain limit (32nd notes or so), smaller fraction, more rare its occurrence in commonly performed music.
    - A *chord* is combination of 2 or more notes sounding simultaneously.
    - A *melody* is a sequence of notes.
    - *Rhythm* is timing pattern assigned to a set of notes.

    Symbolic representation relates abstract symbols to continuous-time sound waves (audio) perceive or imagine, when resp. listening or composing. However, it can be difficult to precisely derive these features from audio. Can explore associated challenges by trying provided *Introduction to Music Representation* programming exercises, where explore a means of converting a symbolic representation to a pseudo-physical representation (audio), then transcribing this generated audio back to a symbolic representation.

    Most musicians may be especially familiar with symbolic representation of music as a printed notation system. Consider 1st 5 bars of famous 5th Symphony of Beethoven, shown in Fig. 1.6: Score depicting well-known motif which opens Beethoven's 5th Symphony.[11] This is a graphical notation form which indicates which notes are to be played, & for what duration. This representation relies on an agreed upon convention of understanding, & can be considered a set of performance instructions. These instructions are translated by a musician & their instrument into physical vibrations of air, forming acoustic signal we hear. Performance instructions we are used to reading draw from a very limited set of possible sounds when consider noise that surrounds us every day (speaking voices, chirping birds, clanging machinery, etc.). In this sect, introduce some common symbolic notation systems adopted by digital devices.

    - 1.2.1.1. MIDI. How might a computer represent symbolic music? Considering 1st violin line of Fig. 1.6, information communicated to performer is sequences of pitches to be played (silence, then a series of 3 Gs, then an E-flat), & duration for these pitches or silence (8th notes, at a tempo of Allegro con brio, a rate of 108 beats per minute; in case of this piece, each beat comprises a measure of 4 8th notes).

      There are some challenges to translate this to a computer instruction (Exercise 2). 1st, though 3 notes are given in indirect sequence, there must be some brief (& unmeasured) separation between them, which is conventionally understood by performer; without this, would hear sequential notes performed as a continuous tone. I.e., sound must not be sustained between 3 notes since separation defines note length. MIDI [Musical Instrument Digital Interface] standard requires definition of PPQ (pulses per quarter note) & tempo (measured as microseconds per quarter note). Using these rates, can set durations (measured in ticks) for each note, with appropriate pauses in between, resulting in memory table illustrated in Fig. 1.7: A table representing MIDI instructions for 1st violin of Beethoven 5. In addition to duration of each note, must also communicate which pitch is to be played (in this case, a G4), which map to MIDI index standard (reference pitch C4 = Note Number 60) to determine G4 = Note Number 67. Having covered pitch & duration, remaining information available in a score is relative volume at which notes are played, often reflected in parameter of Velocity in MIDI, which can indicate power at which a note is struck or released [4]. MIDI table is therefore a sequence of these parameters, given as instructions to turn given note ON or OFF for given duration at given velocity.
    - 1.2.1.2. Piano Roll. Similar to Western music notation, *piano roll* notation system uses a horizontal time axis & a vertical pitch axis. Notes are represented by activation of a point on piano roll, indicating pitch should be sounded at that time. Original piano rolls were perforated paper (giấy đục lỗ), with perforations read be a player piano to activate a particular keypress. May also be familiar with piano roll as a popular visualization of keyboard music on YouTube

[11] Technically, missing clarinets (kèn clarinet) from this figure (& many instruments which are resting), but motif is clear & should spark intended musical memory.

(Check out a great variety of piano roll notation examples, 1st example being a true piano roll: https://www.youtube.com/watch?v=TgrqjkWqhE8, https://www.youtube.com/watch?v=IAOKJ14Zaoo, https://www.youtube.com/watch?v=8QorZ9fcg3o.) (though note: sometimes time & pitch axes are interchanged). Fig. 1.8: Notice similarities between notated score, MIDI representation, & now piano roll representation of 1st bars of Beethoven's 5th Symphony. Quantized cells are highlighted to indicate activation; as reader scrolls through piano roll, these pitches will be sounded as cursor (whether a reader's eyes, a computer scan, or a turning mechanism) passes over respective column. shows continued example of opening of Beethoven 5, this time in piano roll notation.

· 1.2.1.3. Note-Tuple Representation. MusPy package reads music in note-tuple representation, where music is framed as a sequence of (pitch, time, duration, velocity) tuples. If want to encode beginning of Beethoven 5 for a violin part, would could use chart in Fig. 1.7: $(67, 0, 55, 100), (67, 60, 55, 100), (67, 120, 55, 100), (63, 180, 240, 100)$. Note: events are represented with an initial onset time & duration, rather than a set of note-on & note-off instructions.

· 1.2.1.4. ABC Notation. ABC notation https://abcnotation.com/ (stylized "abc") was designed to notate folk tunes (ký hiệu giai điệu dân gian) (i.e., single melodies of 1 stave – giai điệu đơn của 1 khuông nhạc) in plain text format. Since its inception by Chris Walshaw in 1993 & subsequent popularization, features have expanded to allow for multiple voices, articulation, dynamics, & more. 8th note is given unit length, so Beethoven 5 could be written as [|zGGG|_E4|]; check out coding scheme included on ABC notation reference site to understand how different notes, rests, & durations are encoded.

· 1.2.1.5. Music Macro Language. Music Macro Language (MML) is another text-based language, similar to abc notation. Many variations have developed with different use cases; 1 distinguishing feature: use of operators that change parser state while parsing notes (e.g. an "octave up" or "octave down" symbol which reads continually in a new octave until reset). Usual Beethoven example might be notated as "pg8g8g8e-2" (but note: this would be preceded in an MML file by some global parameters e.g. default note length, tempo, etc.). A description of modern MML can be found in Byrd et al.'s reference handbook [5].

∗ 1.2.2. Audio Representations. Unlike symbolic representations of music, *audio* does not explicitly specify musical structures & events e.g. note onsets, instead modeling physical sound itself. When considering audio representation, referring to digital representation of physical sonic information, as opposed to coding schemes & data-structures used to format such information into "files". I.e., when speak of audio, speaking of vector of values that correspond to pressure amplitudes perceived by microphone, & not to specific encoding formats (e.g. "WAV", "MP3", etc.) used to represent this information. When record an orchestra playing score from Fig. 1.6, microphones record air pressure patterns in form of digital audio, depicted in Fig. 1.9: Visualization of a recording of opening of Beethoven 5. Horizontal axis is time (secs), & vertical axis is amplitude. Digital audio, at its finest level, represents sound encoded as a series of binary numbers. Wave-like nature of sound is not apparent due to physically compressed viewing window, so a zoomed-in view is offered in Fig. 1.10: Zoomed-in visualization of a recording of 1 of 1st 3 notes of Beethoven 5 (orchestra playing G2, G3, & G4 simultaneously). Horizontal axis is time (secs), & vertical axis is amplitude. As zoom in on audio signal, wave-like nature may be more clear. These visualizations are generated by free audio software Audacity, introduced in Appendix C.

Basic process of capturing an audio file is recording acoustic vibration using some sort of a sound-capturing mechanism, normally a mechanically driven microphone that translates air pressure changes to electrical signals. Will not discuss microphone & recording techniques in this book. Since dealing with computational methods, also skip analog ways of recording & storing sounds, e.g. magnetic tapes or vinyl records (băng từ hoặc đĩa than). In next chap, discuss digitizing sounds, also called analog-to-digital conversion (ADC) which is a process of sampling a continuous sound at fixed, albeit very short or fast-paced intervals. Understanding theory of sampling & reconstruction of analog signals from their digital version, called digital-to-analog conversion (DAC), is important for assuring: quality of digital sound will be equal to its analog version, i.e., to assure perfect reconstruction of acoustic waveform to prevent distortion due to sampling process.

Considering representation of audio from content perspective, there are also multiple levels of analysis that mus be addressed. Immediate level is conversion of audio signal information initially represented as a sequence of numbers in time, to a collection of frequencies that change over time. This type of representation change, or transformation of audio data from 1 format to another is purely technical or mathematical & is actually a common technique in audio processing which can be quite revealing. Note: also call a digital sound a waveform, even though it is in fact a time series or an array of numbers.

Sound can be framed as superposition of sinusoids (called *Fourier analysis*). Notes produced by a musical instrument are a complex superposition of pure tones, as well as other noise-like components. *Harmonics* are integer-multiples of fundamental frequency which produces sound.

Example 2 (Harmonics & Overtones)**.** *If begin with a fundamental frequency of 440 Hz, 1st integer multiple of frequency* ($2 \times 440 = 880$ *Hz) is referred to as 1st harmonic. 2nd integer multiple of frequency* ($3 \times 440 = 1320$ *Hz) is referred to as 2nd harmonic.*

How are these harmonics generated? When energize a physical oscillator (pluck a string, strike a drum – gảy 1 dây đàn, đánh 1 cái trống), medium has several modes of vibration possible, & usually case: multiple modes will vibrate simultaneously. These additional modes beyond fundamental generate overtones*, additional pitches present (usually with lower energy than the fundamental) when an instrument produces a sound. Because these overtones often fall very close to harmonic series, they are often referred to as* harmonics*.*

At next abstracted level of audio representation, more & more complex features can be extracted. 1 immediate caveat of moving up in hierarchy of features: in order to get deeper insight into sound, also need to discard some information. E.g., might want to consider sinusoidal amplitude or energies at different frequency bands, rather than preserve exact

detail about timing of sinusoids, thus discarding their phase information. This immediately causes representation to be non-invertible, i.e., original waveform cannot be directly reconstructed from features. Pitch, loudness, MFCC, & Chroma features that were discussed earlier in this chap are considered to be higher level representation of audio. General term applied to such representation is "audio descriptors", which can vary in granularity (độ chi tiết) in time or extent of sound events that it represents. General study of such features belongs to a relatively young field of *Music Information Retrieval* (MIR) since 1 of immediate uses of such features is classification or recognition of music that can be used for various retrieval applications.

One might consider *transcription*, process of converting music from an audio representation to a symbolic notation, as ultimate representation task. Some musicians transcribe by ear to handwritten notation; likewise, researcher explore methods of computer transcription. Many commercial software applications offer their solutions toward automatic transcription as a tool for composers & performers, but there is still much to be improved in these systems, much like an auto-captioning system may make mistakes while transcribing spoken language to text. To understand complexity of transcription task, in our computer exercises provide an example of matching MIDI piano roll representation to an acoustic representation called Constant-Q Transform (CQT). Difficulty in finding such match depends on complexity of sounds performing MIDI score, i.e., timbre of instrument performing music complicates frequency contents of signal. Today, state of art transcription methods e.g. MT3 system [6] use neural networks especially trained to recognize different musical instruments, significantly outperforming traditional signal-processing-based methods.

- 1.3. Acoustics & Basics of Notes. 1 of basic tenets (nguyên lý) of applying ML to music: acoustic signals that we consider musical have a certain common structure & organization that can be captured by a machine, if it is exposed to a rich variety of examples. There is an inherent trade-off or tension between randomness & rules in music. Expect music to be exciting, changing, & dynamic, but at same time predictable enough to make sense, be recognizable, & provide engagement. On most rudimentary level, might consider musical acoustics from perspective of periodicity vs. noise. After all, all musical cultures have selected from infinite possibilities of sounds 2 large categories – musical tones, which are sound phenomena that have a relatively clear perception of pitch or musical height (i.e., notes), & percussive events (sự kiện gõ) that have sharp transitory sounds, where main structuring element & characteristic would be organization into rhythms. Other types of sounds, e.g. many natural sounds & sound effects (think about sounds of water in a river sound or splashing waves, animal calls in a jungle, or industrial noises) have came into use only recently, with invention of recording technology & electronic sound synthesis. Such sounds are much more difficulty to categorize, organize, & compose. Fascination with such sounds was a driving force behind many modern music experimental practices (a term used to denote music innovations of 20th century), as well as instrumental music innovations till this day.

  Keeping in mind broad distinction between structure & randomness (Terence Tao's book with same title), deal 1st with question of periodicity vs. noise on short term acoustic level. Broadly speaking, distinguish between:

  * periodic or almost periodic signals
  * noise or a-periodic signals

  Framework we will use to consider such signals is so-called "random processes", which considers random signals as signals whose instantaneous values are largely unknown, but can still be predicted with a certain level of probability. Moreover, largely divide random signals into stationary & non-stationary, where stationary signals retain their statistical characteristics in long run, which can be either done by long enough observation of signal statistics that do not change, or by repeated independent observations of signal on different occasions. Strictly speaking, this equivalence of long observation or many short independent observations is called "ergodic" property, which is a common assumption in stochastic analysis. E.g., every time pluck a string on a guitar or blow air into a flute, exact vibrations of air waves will be different, but if repeat it many times, will be able to observe (or hear) all such variations. In practice, most audio signals, & formidably music, is non-stationary. Play different notes, & express different sounds when we speak (vowels, consonants, change in intonation – nguyên âm, phụ âm, thay đổi ngữ điệu). Even machinery that keeps running in same manner (think about a car engine), will have different sounds depending on changes in way it works (press gas pedal). Separately deal with non-stationary sounds when try to segment audio recording into frames of short-time sound events, or treat notes as an abstraction of a short-term fixed event, which in practice will be synthesized or rendered into sound slightly differently every time. Each note will be characterized by some typical periodic property, e.g. name or MIDI number of note being played. These tones are periodic, but need to define periodicity 1st.

  Periodic signals are considered random signals having some repeated property or phenomena that is same after a particular time. Periodic phenomena do not need to be exact repetitions. Periodic signals return approximately to their previous values after a specified time interval, an effect that can be statistically captured by computing *correlation* or other statistical similarities.

  Example 3 (Correlation)**.** *In interest of creating ML models which generate audio that is similar to existing audio, important: have a means of measuring similarity of 2 audio signals.*

  Consider a sine wave. Sine wave is offset from cosine wave, unless given a delay of half its period, at which point 2 signals perfectly align. Because of this, difficult to produce exactly 1 number that captures strength of similarity between 2 signals – there is a natural dependence on amount of delay.

  Correlation function accounts for this, providing a measure of similarity (or coherence) as a function of signals' relative delay n:

$$(f \star g)[n] = \sum_{m=-\infty}^{\infty} \overline{f[m]} g[m+n],$$

where f, g: 2 signals being compared. Here, bar over $f[m]$ refers to complex conjugate; multiplying a conjugate term with a non-conjugated term gives a value which increases when 2 complex numbers are in near phase angles.

Readers interested in understanding mathematics behind cross-correlation & similar topics of autocorrelation & convolution are encouraged to visit resources in signal processing provided in Chap. 4. Recommend 2 primary takeaways:

- *Similarity between signals is an important property that we seek to quantify as a tool for training ML models.*
- *A product between 2 signal samples, then summed over all possible steps through signal, will be a recurring theme in both signal processing & neural network architectures, giving networks their ability to extract features from signal they act upon.*

Use Fourier Analysis to compare signal to idea repeating signals which are sinusoidal wave-forms (dạng sóng hình sin). When multiple sinusoidal wave-forms can be found in a signal, call these constituents "partials" or "harmonics". For a signal to be periodic, its partials have to be related by integer ratios, which is a reason for calling them "harmonics". One can show this applying convolution theorem states: convolution (or filtering operation) in time is multiplication in frequency & vice versa. For a time-domain signal to be periodic, in need to repeat same block of signal every period. Such periodic signal can be expressed as a short basic waveform comprising of a single period waveshape convolved with a Dirac comb. Thus, its Fourier transform will be Fourier transform of basic waveform multiplied by Fourier transform of a Dirac comb, which turn to be another Dirac comb, thus resulting in a signal that is nonzero at discrete frequencies. Another way to prove why periodic signals have discrete frequencies: think about physics of standing waves. Since boundary conditions of a vibrating string or pressure levels for a sound wave propagating along a pipe are fixed, only waves that can exist are standing waves that respect these boundary conditions. From geometric considerations (or by looking at solutions of wave equation) one can see: basic frequency is lowest period that satisfies boundary condition, but then other waves are possible too, but they always will need to be multiples of basic frequency to satisfy boundary conditions.

Sounds can be in-harmonic (e.g. bell sounds) that have multiple periods that sound more like musical chords then a single note with well defined pitch. Period of note is often called "fundamental frequency;; & is closely related to a perception of pitch. To be precise, pitch is a perception of note height, while its fundamental frequency is mathematical property of lowest partial in a harmonic signal (& sometimes perceived pitch is higher than fundamental frequency).

Analysis of sound into its sinusoidal constituents is also known as spectral analysis since individual partials are considered as equivalent of a specific color or spectral line in visual domain. Fig. 1.11: Prism splits lights according to colors *schematically demonstrates this effect fo splitting white light into its constituent colors by a prism (*Hình 1.11: Lăng kính phân tách ánh sáng theo màu sắc *minh họa sơ đồ hiệu ứng này để phân tách ánh sáng trắng thành các màu thành phần của nó bằng lăng kính). Light analogy should be carried with some caution since underlying prism phenomena occurs due to physical effect called diffraction (sự nhiễu xạ) that causes bending of light when it passes from 1 material to another. When diffraction depends on frequency, this causes different colors to land at different angles. A much more interesting physical decomposition of sound into constituent frequencies happens in ear, when due to specific geometry of cohlea (ốc tai) standing waves occur at different positions on basilar membrane (màng đáy), which are then captured by hair cells & transmitted to brain according to position & time of firing that contains information about magnitude & relative phase of each frequency. Details of auditory processing (xử lý thính giác) happening in cohlea are beyond scope of this book, & interested readers are referred to a beautiful animation by Howard Huges Medical Institute https://www.youtube.com/watch?v=dyenMluFaUw .*

So same way spectral analysis of visual signals or even electromagnetic radiation tries to separate signal into individual spectral lines, Fourier analysis does that to audio signals. On opposite side of periodic signal stands noise. Mathematical speaking, noise does not have spectral lines. Its spectral analysis comprises broad spectral bands, or continuous spectrum.

An interesting relation exists between periodicity & information. 1 common property of signals, musical or not (think about radio waves), is its ability to carry information. Encoding information into a physical signal & later extracting it from signal (a process known as decoding) requires extracting or recognizing salient properties (tính chất nổi bật) of signal. E.g., when fluctuation in air pressure from a sound reaches eardrum, this signal is converted into electrical impulses that are sent to brain, which is later interpreted as something "meaningful" or sth carrying or containing information. Later define concept of information as a relation between incoming signal & expectations or predictions of decoder. This subtle, & somewhat confusing term, is very important. Basically say: information is carried between 2 random phenomena, & ability to decode or extract information is related to reduction in uncertainty about incoming signal done in our brain, or by a model that a computer has learned. A signal's decoding (as well as perception, from an information-theoretic view) allows information aspects of signal to be removed from any specific physical nature of that signal, e.g., current, voltage, or amplitude of an acoustic wave. In this view, noise is unpredictable or disturbing element that prevents or makes it more difficult to extract useful information from a signal. Mathematical model of signal is some function that often has ability to capture essential structure, e.g. periodicity of signal, & distinguish it from noise. Focus will be not on physical models, but rather predictive or information-theoretical models.

- 1.4. Perception & Cognition: Anticipation & Principles of Music Information Dynamics. (Nhận thức & Nhận thức: Dự đoán & Nguyên tắc của động lực thông tin âm nhạc) Same as in case of representation, question of music perception & cognition exists on various levels: it deals with physical transduction of acoustic phenomena to nerve impulses, translation of these impulses to perception of sound in auditory cortex, segmentation & segregation of audio flow into sound events & their recognition, all way to cognitive models dealing with expectation, emotions, & feelings of surprise vs. familiarity of incoming sound stream – including enculturation & entrainment of specific musical styles & learning of normative sound expressions. Note: dealing with music understanding will often require to consider questions of perception & cognition in rather abstract terms. Unlike speech or natural or man-made sound events, e.g. understanding spoken word, recognizing a bird chirp or

detecting a submarine from a clutter of sonar blips, music as an art form often does not have a concrete or specific denotation with a universally agreed meaning. Moreover, different musical cultures use different methods for structuring sounds into music – some may put more emphasis on structuring notes in harmonies, tonalities, & long-term structure, while others may exhibit more sensibilities to subtle variations in sound or emotional inflections of sound production itself, while allowing more freedom or deviations in terms of tuning, or molding voice relations into strict harmonic relations.

– Tương tự như trong trường hợp biểu diễn, câu hỏi về nhận thức âm nhạc & nhận thức tồn tại ở nhiều cấp độ khác nhau: nó liên quan đến sự chuyển đổi vật lý của hiện tượng âm thanh thành xung thần kinh, dịch các xung này thành nhận thức âm thanh trong vỏ não thính giác, phân đoạn & phân tách luồng âm thanh thành các sự kiện âm thanh & nhận dạng chúng, tất cả đều hướng đến các mô hình nhận thức liên quan đến kỳ vọng, cảm xúc, & cảm giác ngạc nhiên so với sự quen thuộc của luồng âm thanh đến – bao gồm cả sự đồng hóa & sự đồng bộ của các phong cách âm nhạc cụ thể & học các biểu đạt âm thanh chuẩn mực. Lưu ý: khi giải quyết vấn đề hiểu âm nhạc thường đòi hỏi phải xem xét các câu hỏi về nhận thức & nhận thức theo những thuật ngữ trừu tượng hơn. Không giống như lời nói hoặc các sự kiện âm thanh tự nhiên hoặc do con người tạo ra, ví dụ như hiểu từ được nói, nhận ra tiếng chim hót hoặc phát hiện ra tàu ngầm từ một mớ tín hiệu sonar, âm nhạc như một hình thức nghệ thuật thường không có một biểu thị cụ thể hoặc cụ thể với ý nghĩa được thống nhất chung. Hơn nữa, các nền văn hóa âm nhạc khác nhau sử dụng các phương pháp khác nhau để cấu trúc âm thanh thành âm nhạc – một số có thể nhấn mạnh hơn vào cấu trúc nốt nhạc trong sự hòa âm, âm điệu, & cấu trúc dài hạn, trong khi những nền văn hóa khác có thể thể hiện nhiều nhạy cảm hơn với những thay đổi tinh tế trong âm thanh hoặc sự biến đổi cảm xúc của chính quá trình tạo ra âm thanh, đồng thời cho phép nhiều sự tự do hoặc độ lệch hơn về mặt điều chỉnh hoặc định hình mối quan hệ giọng nói thành mối quan hệ hòa âm chặt chẽ.

Relating computer modeling & ML methods to human perception & cognition is a formidable task that is beyond scope of this book. Way we may relate statistical & ML models to aspects of human perception is through identifying & naming different types of musical data on continuum of acoustics-perception-cognition. As such, make a distinction between musical timbre, or quality of sound itself, musical texture, which is a relatively short-term aggregation of musical elements governed by common statistics, & musical structure that deals with organization of shorted musical elements into longer repetitions, e.g. chorus-verse or other formal arrangements of musical repetitions with variations, comprising so-called musical form.

In our approach, expectancies will be playing a central role in shaping experience of musical structure. An idea put forward many years ago by music theorists e.g. Meyer & Narmour states: listening to music consists of forming expectations & continual fulfillment or denial thereof [7,8]. Information-theoretic measures of audio structure have been proposed in attempt to characterize musical contents according to its predictive structure [9–12]. These measures consider statistical relations between past, present & future in signals, e.g. predictive information that measures mutual information between limited past & complete future, information rate that measures information between unlimited past & immediate present, & predictive information rate that tries to combine past & future in view of a known present. Additional models of musical expectancy structure build upon short-long-term memory neural networks introduced in Chap. 7 & predictive properties that are local to a single piece vs. broader knowledge collected through corpus-based analysis. Later models also include use of more elaborate & sophisticated neural models, e.g. WaveNet, SampleRNN, Transformers & more, covered in Chaps. 7 & 9. 1 common property of such models: they assume: some hidden or *latent* (ẩn hoặc *tiềm ẩn*) representation exists that is capable of capturing a great variety of music in 1 model. Such mapping is often termed encoder-decoder to emphasize information communicated by system, where a mapping of musical notes or acoustic data (sometimes referred to as musical surface or foreground) into its latent representation (also called background or reduction) is done in terms of probabilistic model. When time aspects are taken into account, such musical processes are broadly analyzed in terms of changes in information called "Music Information Dynamics". Underlying assumption in investigation of Musical Information Dynamics (MID): changes in information content, which could be measured in terms of statistical properties e.g. entropy & mutual information, correlate with musically significant events, which in parallel could be captured by cognitive processes related to music perception & acquired through exposure & learning of regularities present in a corpus of music in a certain style. These models may provide an explanation for inverted-U relationship proposed by Daniel Berlyne that is often found between simple measures of complexity & judgments of aesthetic value [13]. When applied to music & its predictive properties, see: very simple music, though predictable, has less MID effect since relative reduction in complexity through prediction is not significant. Very complex music that is hardly predictable has also small MID since complexity before & after prediction remain high. "Sweet-spot", according to MID theory is when music appears complex initially, but by careful listening & prediction, complexity is drastically reduced. In other ways, ability of listener ot "explain out" a lot of music complexity by active listening has an important effect on their enjoyment of music, which is captured by MID in terms of relative reduction of complexity through prediction.

- Chap. 2. Noise: Hidden Dynamics of Music. Noise has always played a central role in music. As sound production, noise often serves as initial source of energy in musical instruments that "carve out" notes from a pluck of a string or from a turbulent air stream blown into a pipe. Have seen in prev chap how an unvoiced hiss or pulses of a glottal buzz are shaped into speech sound via a source-filter voice model (cách một tiếng rít vô thanh hoặc các xung của tiếng vo ve thanh quản được định hình thành âm thanh lời nói thông qua mô hình giọng nói nguồn-bộ lọc). Filters & resonances help turn flat frequency response of an impulses or of a continuous noise into a periodic, more predictable sound structure. As an idea generator, noise is driving musical choices during composition process. As much as music theory tells us how choices of notes need to be done according to certain rules, role of noise is in creating uncertainties & ambiguities that are an essential component in music listening experience. Random methods, often called "aleatoric" music (nhạc ngẫu nhiên) in contemporary practice, have been used historically to trigger & drive process of music creation by generating initial musical choices, which were further constrained & shaped during later stages of composition. Moreover, on a broad cultural & historical perspective, Attali's book *Noise: The Political Economy*

of Music examines relations of music to political-economic structure of societies through noise. Although such social-science & economic investigations of music are outside of scope of this book, a consideration of what Attali understands by term "Music' might be appropriate for discussion. For Attali music is organization of noise, & entirely of evolution of Western knowledge through development of scientific, technical, & market tools is viewed through lens of shaping & controlling noise:

> "World is not for beholding, it is for hearing. Our science has desired to monitor, measure, abstract, & castrate meaning, forgetting that life is full of noise & that death alone is silent. Nothing essential happens in absence of noise."

- 2.1. Noise, Aleatoric Music, & Generative Processes in Music. Consider generative processes that turn randomness into structure. In prev chaps considered 2 main data formats for music: one is as a sound recording captured into an audio file, the other is a sequence of performance actions captures as notes in a symbolic MIDI or music sheet format. To understand better structures present in these 2 data types, introduced various features that allow more meaningful representations by extracting salient aspects of sound, going back & forth between signals & symbols. Music is a unique human endeavor (nỗ lực độc đáo của con người): it comprises deliberate & intentional selection of sounds from infinite possibilities of air vibrations, narrowed into a much smaller palette of musical instruments producing notes from periodic vibrations. These notes are further organized into scales, tonalities, melodies, & chords, building progressions & repetition of themes arranged into verse, chorus, & other musical forms. Fact: all of these constructions are deliberate human choices is often taken for granted. Different cultures follow different musical rules, & when musical styles evolve & instruments change, type of music expression may drastically change. It takes a while to get used to & find beauty in unfamiliar music, or come up with new ways of sonic expression or break existing musical rules to create artistic novelty & impact.

  As listeners, or even as composers, often do not think about such formal aspects of musical evolution or change, but when challenge of finding or generating meaning in musical data is given to machine, need to revisit basic premises (tiền đề cơ bản) of what makes musical sounds meaningful in order to be able to program musical algorithms or build musical learning machines. Music can be defined as *Organization of Sounds to Create an Experience*. In this most abstract & general way of defining music, capture some of fundamental principles for a formal study of music. 1st, a selection of sounds is made, representing them as latent variables or features extracted from sound data. Their organization assumes certain temporal & sequential relations that could be further established by learning dynamics of these latent variables. & finally, creating an experience requires including listener as part of model.

  - 2.1.1. Music as Communication. Understanding music requires both a sender & a receiver, or encoder & a decoder that communicate information. But unlike in vision, speech, or computer communications, where a typical inference task is 1 of recognizing a particular shape, transcribing speech, or executing a command, effect of musical information is 1 of eliciting an emotional or aesthetic response. Such response assumes an active listening framework that triggers a wide variety of reactions, some of which might be innate or pre-wired, & others learned from exposure to existing musical cultures. Can such structures be learned by a machine in a so-called end-to-end fashion by feeding it with music data paired with a desired target output? Can novel applications & unheard musical style be created by going outside of statistical models of existing styles or emerge through machine interaction with a human?

    In journey of ML in music, find concept of noise particularly useful. On signal level, spectral analysis allows distinguishing acoustic noise from periodic sounds. Noise, in its "purest" physical or engineering sense, is a signal characterized by a white spectrum (i.e., composed of equal or almost-equal energies in all frequencies). Mathematically, to qualify as a random or stochastic process, probability density of signal frequency components must have a continuous spectrum, whereas periodic components that are more predictable present themselves in frequency domain as spectral lines, i.e. sharp & narrow spectral peaks. Moreover, in terms of its musical use, noise does not allow much structural manipulation or organization. In contrast to noise, notes are commonly basic building blocks of larger sound phenomena that we call music, & as such could be considered as elements of sonic "structure". On symbolic level, noise is more closely related to predictability of a sequence of elements, often encoded into tokens. When applying language models to music, noise refers to uncertainty or variability of next token as possible continuations to past sequence of tokens. Although seemingly unrelated, these 2 aspects of noise have common mathematical foundations that are related by signal information contents measured in terms of residual uncertainty passed in a signal over time. Better our prediction about future becomes, smaller prediction error will be, & overall uncertainty will be reduced. In this chap, take a new look at relations between randomness & structure in both symbolic & audio signals. In order to do so, consider meaning of randomness as a phenomena that has an aspect of uncertainty & surprise to it. This is contrasted with *structure*, an aspect of data that is more predictable, rule-based, or even deterministic.

    From listener perspective, acoustic noise belongs to perceptual domain, while perception of musical organization belongs to cognitive domain. 2 domains are bridged by Prediction Coding framework in cognitive science that is centered around idea: organisms represent world by constantly predicting their own internal states. Comparisons between predictions & bottom-up signals occur in a hierarchical manner, where "error signal" corresponds to unpredicted aspect of input that passes across various stages of representation & processing. Core framework posited by such theories is 1 of deep generative models that can be understood both in terms of Bayesian theories of cognition that consider perception as a hypothesis-testing mechanism, & as a simulation model that considers signal representation in terms of statistical models that incorporate signal uncertainty as part of their generative structure. Such fundamental questions about mental representations are outside scope of this book, but similar concerns emerge in design of musical applications where practical issues of structural representation are often given by more historically mainstream accounts of music theories, & later extended through statistical-probabilistic structures that are designed to bear musically relevant similarity to some musical

goal or idea. Such similarity can vary from stylistic imitation, sometimes called "deep fake" models of existing music, to experimental practices that explore novel musical ideas & possibilities for creative interaction with machines.

Throughout book, often refer to a particular approach to musical modeling that views anticipation as key to understanding music, planning its structure, & a possible way of understanding effects that music has on listener's mind. Already mentioned anticipation in prev chap where introduced concept of Musical Information Dynamics as a generic framework for modeling processes in terms of changes in its information contents. But before exploring further statistical tools that are needed in order to extract & quantify information in music & sound, it helps to see anticipation in context of historical developments that led to contemporary ways of music modeling & formalization.

- 2.2. History of Mathematical Theory of Music & Compositional Algorithms. Continuing discussion of noise from historical perspective, important to note: creation of music has been always linked to certain technical tools, namely musical instruments, that enabled creation of sounds & determined their structuring principles. As such, randomness & uncertainty in sound production has always been linked to its production & composition methods.

  Ancient Greek philosopher Pythagoras (570–495 B.C.) is generally credited with having discovered mathematical relations of musical intervals which result in a sensation of consonance. Such relations are deeply rooted in acoustics of string & wind instruments, & human voice, which are highly harmonic. In Pythagorean tuning, frequency ratios of notes are all derived from number ratio 3:2, which is called an interval of a *5th* in music theory.

  Remark 2. *In musical terms, an* interval *is difference in pitch between 2 notes. In a physical sense, interval is a ratio of frequencies between 2 notes. Different tuning systems may assign different frequency ratios to intervals of same name. In Western music, intervals are usually described by a number (unison, 2nd, 3rd, 4th, 5th, etc.) & a quality (perfect, major, minor, augmented, or diminished). In general, number indicates spacing, while quality indicates some further tuning on a smaller scale than represented by number.*

  When 2 notes are played in a 5th, this music interval produces a combined sound where most of partials overlap, thus creating a seamless blending & a highly coherent precept. Using 5th & movign along circle of 5ths (Fig. 2.1: Circle of 5ths is a helpful construct for musicians to relate patterns in key signatures, harmonic progressions, & other uses of chromatic scale. Each step clockwise around circle represents a note an interval of a perfect 5th above the previous.) results in pentatonic & diatonic scales (taking 5 & 7 steps, accordingly) that are common in Western music. Although this tuning was adjusted later on in history to accommodate for more complex chromatic musical styles, idea: music sensations can be related to mathematical proportions & rules had been rooted deeply in music theory. Moreover, cultures that use different musical instruments, e.g. metallophones (metalic percussion instruments) dominant in Gamelan music, result in different tuning system, e.g. Pelog & Slendro, where preferred intervals are s.t. different tones optimally blend by maximizing overlap between partials of their respective timbres.

  In terms of composition practices, probably best known early example of using formal rules for writing music can be attributed to Guido d'Arezzo who, around 1026, set text to music by assigning pitches to different vowels. According to this method a melody was created according to vowels present in text. In 14th & 15th centuries isorhythmic techniques were used to create repeated rhythmic cycles (talea), which were often juxtaposed with melodic cycles of different length to create long varying musical structures. Dufay's motet Nuper Rosarum Flores (1436) used repetition of talea patterns at different speeds in 4 secs according to length ratios 6:4:2:3, which some claim were taken from proportions of nave, crossing, apse, & height of arch of Florence Cathedral or from dimensions of Temple of Solomon given in Kings 6:120. Formal manipulations e.g. retrograde (backward motion) or inversion (inverting direction of intervals in a melody) are found in music of J.S. Bach & became basis for 20th century 12-tone (dodecaphonic) serial techniques of Arnold Schoenberg (1874–1951). Exploiting recombinant structure of harmonic progression (Khai thác cấu trúc tái tổ hợp của tiến trình hài hòa), Mozart devised his Musikalisches Wurfelspiel ("Musical Dice Game") (1767: Though this game is attributed to Mozart, earlier examples have been discovered, including Johann Kirnberger's 1757 *Der allezeit fertige Menuetten- und Polonaisencomponist* – Nhà soạn nhạc minuette và polonaise luôn sẵn sàng) that uses outcomes of a dice throw & a table of pre-composed musical fragments to create a Minuet. There are 176 possible Minuet measures & 96 possible Trio measures to choose from. 2 6-sided dice are used to determine each of 16 Minuet measures & 1 6-sided die is used to determine each of 16 Trio measures. This provides an early example of a stochastic process driving musical output, a theme in Variational Autoencoder & other examples. In this case, compositional blocks are defined in a discrete codebook, but in later cases, find these constituent pieces to be learned from bodies of musical data.

  Search for formal composition techniques took on a life of its own in post-WWII academic music. In a trend which is sometimes attributed to desire to break away from stylistic confines & associations of late Romanticism, mathematical rules e.g. serialism were sought in order to construct musical materials that sounded both new & alien to traditional tonal & rhythmical musical languages. 1 such feeling was expressed by Czech novelist Milan Kundera in his writing about music of Greek composer & microsound theorist Iannis Xenakis 1922–2001) – "Music has played an integral & decisive part in ongoing process of sentimentalization (Âm nhạc đã đóng một vai trò không thể thiếu & quyết định trong quá trình tình cảm hóa đang diễn ra). But it can happen at a certain moment (in life of a person or of a civilization that sentimentality) (previously considered as a humanizing force, softening coldness of reason) becomes unmasked as 'supra-structure of a brutality'. ... Xenakis opposes whole of European history of music. His point of departure is elsewhere; not in an artificial sound isolated from nature in order to express a subjectivity, but in an earthly 'objective' sound, in a mass of sound which does not rise from human heart, but which approaches us from outside, like raindrops or voice of wind;; [4].

  Use of automation or process is common thread in above examples. Composers have increasingly allowed portions of their musical decision-making to be controlled according to processes & algorithms. In age of computer technology, this trend

has grown exponentially. Before going into more technical aspects of musical modeling with computers, helpful to compare these experimental academic music approaches to those in non-academic musical world. To gain this perspective requires briefly surveying different stylistic approaches for using computers in popular music making.

∗ 2.2.1. Generative Music in Popular Music Practices. 1 particular genre of generative music that has become popular among general listening public is *ambient music*. Governed by recombinant processes of chords & melodies, ambient music emphasizes mood & texture over directionality & contrast, blending with environment in which it is heard. This music often lacks dramatic or sentimental design of classical & popular music. Unlike generative methods in academic experimental genres, e.g. Xenakis's exploration of masses of sounds & processes in opposition to Western music tradition, ambient genre centers around familiar & often pleasant sounds drawn either from tonal harmonic language or from recordings of environmental sounds that blend into each other with no particular larger scale form or intention.

Perhaps 1st explicit notion of ambient music comes from Erik Satie's concept of *musique d'amebulement*, translated as "furniture music". Satie's music is often built from large blocks of static harmony, & creates a sense of timelessness & undirected sound, but this was music which was meant not to be listened to, only to be heard as background to some other event [15]. There is arguably no composer more associated with ambient & environmental sound than John Cage, 20th-century experimentalist who redefined materials & procedures of musical composition. Throughout his career, Cage explored outer limits of sonic experience, but his most notorious work is famous "silent piece", 4'33", composed in 1952. Instead of defining musical materials, Cage defined a musical structure, & allowed ambient sounds of concert environment to fill structure [16]. His other groundbreaking approaches to music included reliance on chance operations, or so-called indetedeminate or "aleatoric techniques" for its random selection of notes. In his *Music of Changes* Chinese I-Ching book technique was used to select note pitches & durations. I-Ching is also used to determine how many layers should there be in a given phrase to create different densities of musical voices. These chance operations were done independently of previous outcome. When selection of next notes are done in ways that are dependent on already generated events, this induces so-called *Markov property* to chance operations. Lejaren Hiller & Leonard Isaacson's 1957 Illiac Suite used Markov chains to selected consecutive intervals for each instrument according to greatest weight to consonant vs. smallest intervals. In 1959, Iannis Xenakis wrote several works in which he used many simultaneous Markov chains, which called "screens", to control successions of large-scale events in which some number of elementary "grains" of sound may occur.

Late 20th century saw an explosion of musical styles which challenged traditional notions of musical experience & experimented with ways in which sound can communicate space, place, & location. "Soundscape" composition & use of field recordings help to create a sense of a real or imagined place, while genres of electronic dance music like chill-out are often associated with specific club environments. Some artists have also bridged gap between contemporary visual & installation art & music, & "sound art" is now commonplace. These artists often deal with psychoacoustic phenomena & relationships between visual & auditory media, often work with multiple forms of media, & take influence from minimalism, electronic music, conceptual art, & other trends in 20th-century art.

Most importantly for our discussion: way in which composers of ambient & related genres often delegate (ủy nhiệm, giao quyền) high-level aspects of their work to algorithmic processes or generative procedures – or in case of field-recording work, to features of sound materials themselves. Techniques which originated in most avant-garde (tiên phong) & experimental traditions, e.g. looping, automated mixing, & complex signal processing, are increasingly used in popular music & are now commonly found in commercial, off-shelf music production software.

Brian Eno's Ambient 1: Music for Airports (1978) provides template for many later works: repetitive, with an absence of abrasive or abrupt attacks, & using long decays, harmony & melody seem to continue indefinitely. During compositional process, Eno constructed tape loops of differing lengths out of various instrumental fragments [17]. As these loops played, sounds & melodies moved in & out of phase, producing shifting textures & patterns. Studio configuration itself became a generative instrument. In 1996 Brian Eno released title "Generative Music 1" with SSEYO Koan Software. The software, published in 1994, won 2001 Iterative Entertainment BAFTA (British Academy of Film & Television Arts) Award for Technical Innovation. Koan was discontinued in 2007, & following year a new program was released. Noatikl "Inmo" – or "in the moment" ("inmo") – Generative Music Lab was released by ntermorphic ltd. as a spiritual successor to Koan. Software had new generative music & lyric engines, & creators of software claimed:

> "... learning to sit-back & delegate responsibility for small details to a generative engine is extremely liberating. ... Yes, can focus on details in Noatikl, but can also take more of a gardener's view to creating music: casting musical idea seeds on ground, & selecting those ideas which blossom & discarding those that don't appeal."

Electronic dance music producers, like English duo Autechre, often embrace most avant-garde trends & techniques, while still producing music which remains accessible & maintains a level of popular success. Autechre have often employed generative procedures in their music-making, using self-programmed software & complex hardware routings to produce endless variations on & transformations between thematic materials. In a recent interview, Rob Brown spoke of attempting to algorithmically model their own style:

> "Algorithms are a great way of compressing your style. ... if you can't go back to that spark or moment where you're created sth new & reverse-engineering it, it can be lost to that moment ... It has always been important to us to be able to reduce sth that happened manually into sth that is contained in an algorithm." [18]

Commercial digital audio workstation (DAWs) – computer software for recording, mixing, & editing music & sound – often support various types of generative procedures. E.g., "Follow" function in Ableton Live, a popular DAW, allows composers to create Markov chain-like transitions between clips. 1 of typical uses of this: add naturalness to a clip. Instead of having a single, fixed loop, one records multiple variations or adds effects, e.g. cuts, amplitude envelopes or pitch changes, to a single loop & uses "Follow" to sequence clips randomly. This allows variations both in terms of musical materials contents &

their expressive inflections that breaks away from mechanical regularity & synthetic feel often associated with loop-based musical production.

In field of computer & video games, composers & sound designers, with help of custom software (often called *music middle-ware*), frequently make use of generative & procedural music & audio in their work. Advantages of procedural audio include interactivity, variety, & adaptation [19]. Karen Collins provides a good overview of procedural music techniques [20]. Collins describes challenges face by game composers, who must create music which both supports & reacts to an unpredictable series of events, often within constraints imposed by video game hardware &/or storage. In fact, Collins points out: a contributing factor to use of generative strategies in video games was need to fit a large mount of music into a small amount of storage. Generative techniques allow for creation of large amounts of music with smaller amounts of data. Some strategies for this kind of composition can include:

- Simple transformations: changes in tempo, restructuring of largely pre-composed materials, conditional repeats of material based on in-game activities
- Probabilistic models for organizing large numbers of small musical fragments
- Changes in instrumentation of music based on presence of specific in-game characters
- Generation of new materials within melodic/rhythmic constraints
- Re-sequencing, also known as horizontal remixing, allowing nonlinear playback of pre-recorded music
- Remixing, or vertical remixing, which operates by adding or muting parallel simultaneous musical tracks.

- 2.3. Computer Modeling of Musical Style & Machine Improvisation. Aug 09, 1956 "Iilliac Suite" by Lejaren Hiller & Leonard Isaacson, named after computer used to program composition, saw its world premiere at University of Illinois [21]. Work, though performed by a traditional string quartet, was composed using a computer program, & employed a handful of algorithmic composition techniques. This work, together with work of Xenakis described in *Formalized Music*, marks beginning of modern mathematical & computational approaches to representing, modeling, & generating music. Among most prominent algorithmic composition techniques are stochastic or random modeling approaches, formal grammars, & dynamical systems that consider fractals & chaos theory, as well as genetic algorithms & cellular automata for generation & evolution of musical materials. More recently, advanced ML techniques e.g. neural networks, hidden Markov models, dynamic texture models, & DL are being actively researched for music retrieval, & to a somewhat lesser extent, also for creative or generative applications. With advent of ML methods, models could be trained to capture statistics of existing music, imitate musical styles, or produce novel operations in computer-aided composition systems. With these developments, goals & motivation behind using random operations for composing has changed from thinking about mass events with densities & texture, to capturing fine details of musical style that were not previously amenable to probabilistic modeling.

  Markov models are probably 1 of best-established paradigm of algorithmic composition, both in experimental & style learning domains. In this approach, a piece of music is represented by a sequence of events or states, which correspond to symbolic music elements e.g. notes or note combinations, or sequences of signal features extracted from a recording. Markov sources are random processes that produce new symbols depending on a certain number of past symbols. I.e., whole process of music generation from a Markov model has sopme kind of "memory" whose effect is that new symbols are produced in a manner dependent on its past. I.e., old symbols influence new symbols by determining probability of generating certain continuations. Markov chains are statistical models of random sequences that assume: probability for generating next symbol depends only on a limited past. A Markov model establishes a sequence of states & transition probabilities, which are extracted by counting statistics of events & their continuations from a musical corpus. Length of context (number of prev elements in musical sequence used for calculation of next continuation) is called *order* of Markov model. Markov sources can have different orders: simple independent random sequences are considered as Markov of order 0, then processes of a memory of a single symbol are Markov of order 1, of a memory of length 2 are Markov of order 2, & so on. Since higher order models produce sequences that are more similar to corpus, but are harder to estimate & may also have disadvantageous effects in terms of model's ability to deal with variations, several researchers have turned their attention to predictive models that use variable memory length models, dictionary models, & style-specific lexicons for music recognition & generation. See [22] for a survey of music generation from statistical models & [23] for comparison of several predictive models for learning musical style & stylistic imitation.

  Generative grammars are another well-established & powerful formalism for generation of musical structure. Generative grammars function through specification of rewriting rules of different expressiveness. Study of grammars in music in many ways paralleled that of natural language. Chomsky erroneously believed: grammatical sentence production cannot be achieved by finite state methods, because they cannot capture dependencies between non-adjacent words in sentences or model recursive embedding of phrase structure found in natural language [24]. Since such complex grammars could not be easily "learned" by a machine, this belief limited types of music handled by grammar models to those that could be coded by experts in terms of rules & logic operations. This approach is sometimes called an *expert system* or *knowledge engineering*. Although these methods achieve impressive results, they are labor intensive as they require explicit formulation of musical knowledge in terms that are often less than intuitive.

  Some more specific models, e.g. Transition Networks [25] & Petri Nets [26,27] have been suggested for modeling musical structure. These are usually used to address higher levels of description e.g. repetition of musical objects, causality relations among them, concurrency among voices, parts, sections, & so on. David Cope's work describes an interesting compromise between formal grammar & pattern-based approaches. Cope uses a grammatical-generation system combined with what he calls "signatures": melodic micro-gestures typical of individual composers [28]. By identifying & reusing such signatures, Cope reproduced style of past composers & preserved a sense of naturalness in computer-generated music.

In following, describe some of research on learning musical structure that begins by attempting to build a model by discovering phrases or patterns which are idiomatic to a certain style or performer. This is done automatically, & in an *unsupervised* manner. This model is then assigned stochastic generation rules for creating new materials that have similar stylistic characteristics to learning example. 1 of main challenges in this approach is formulating rules for generation of new materials that would obey aesthetic rules or take into account perceptual & cognitive constraints on music making & listening that would render pleasing music materials.

Remark 3. *A learning process is* unsupervised *if process learns patterns in data without being provided additional expected output. By contrast,* supervised *methods rely on an expected, defined output (commonly referred to as* ground truth*) to drive learning. Supervised & unsupervised methods are applied to different types of problems based on what can be learned in absence (or presence) of ground truth (i.e., known labels or answers to question at hand). E.g., problem of* classification *(predicting class associated with a data instance) can be addressed with supervised learning, given a collection of datum-class pairs. On other hand, without knowing explicit class membership for each data instance, process becomes unsupervised – but can still learn associations between like-data-points, a problem known as* clustering.

Another interesting field of research is machine improvisation, wherein a computer program is designed to function as an improvisational partner to an instrumentalist. Software receives musical input from instrumentalist, in form of MIDI or audio, & responds appropriately according to some stylistic model or other algorithm. An important early work in this style is George Lewis's *Voyager* (1988), a "virtual improvising orchestra". *Voyager* software receives MIDI input derived from an instrumental audio signal, analyzes it, & produces output with 1 of a set of many timbres [29]. Input data is analyzed in terms of average pitch, velocity, probability of note activity, & spacing between notes. In response, *Voyager* is capable of both producing variations on input material & generating completely new material according to musical parameters e.g. tempo (speed), probability of playing a note, spacing between notes, melodic interval width, choice of pitch material based on last several notes received, octave range, microtonal transposition, & volume. Due to coarse statistical nature of both analysis & generative mechanisms, type of music produced by system is limited to abstract free improvisation style.

Another family of "virtual improvisor" software will be introduced in later chap, including PyOracle & VMO https://github.com/wangsix/vmo, & musical improvisation systems OMax & SOMax [12]. These programs use sequence matching based on an algorithm called Factor Oracle, to create a model of repetition structure specific to incoming music material both for audio or MIDI input. By recombining carefully selected segments of original recording according to this repetition structure, programs can produce new variations on input, while maintaining close resemblance to original musical style. Although essentially a procedural remixing method, once determination of next musical step is performed by chance operations that are governed by probability of continuations of motifs of variable length as found in a training corpus, this improvisation process can be considered as an extension of Markov approach to variable memory length models. In next sect introduce mathematical foundation for Markov models that are tool of trade for modeling sequences of data, notably in domain of text & natural language modeling. This will allow a more profound look into concepts of entropy, uncertainty & information, paving road to understanding musical creation & listening as a communication process.

- 2.4. Markov Models & Language Models for Music. Assume: have a sequence of notes (a melody). If notes are generated in an independent manner (e.g. in John Cage's "Music of Changes" mentioned earlier), best description of that music could be in terms of frequencies of appearance of different notes. Claude Shannon, considered to be father of information theory, suggested: if look at large blocks of symbols (long sequences), this information around frequency of appearance "reveals itself" without any need to calculate probabilities. This property (called *asymptotic equipartition property* or AEP, discussed in next chap) basically says: if look at a long enough sequence, see only "typical" messages. Entropy of source defines amount of typical messages. That is why a completely random (aleatoric) sequence, where all notes appear with equal probability, has maximal entropy:number of typical sequences is equal to all possible sequences. On other hand, more structure a sequence has, less will be number of typical sequences. Exponential rate of growth of number of typical sequences, relative to number of all possible sequences of same length, is *entropy*, introduced in next sect. Thus, looking at longer blocks of symbols approach "true" statistics of source.
  - 2.4.1. Entropy, Uncertainty, & Information. Dealing with noise as a driving force of music, learn: not all noise is created equal. In process of throwing dice or passing an acoustic noise through a filter, shaping it into sounds & music structures by making choices of note sequences, some being more probable than others, distinguishing between more probable & more surprising events. Can we measure such surprises? Formally, *entropy* is defined as a measure of uncertainty of a random variable x when it is drawn from a distribution $x \sim P(x)$:

$$H(x) = -\sum_x P(x) \log_2 P(x),$$

    where sum is over all possible outcomes of x. It can be shown: $H(x)$ is nonnegative, with entropy being equal to 0 when x is deterministic, i..e., when only 1 of outcomes x is possible, say x^*, so that $P(x = x^*) = 1, P(x \neq x^*) = 0$ otherwise. Maximal entropy is achieved when all outcomes are equally probable. One can easily show: for x taking values over a set of size N (also understood as alphabet of size N or cardinality of x), then $H(X) = \log_2 N$, which is, up to integer constraints, number of bits needed to represent symbols in alphabet of that size.

    Example 4 (Entropy of a Binary Variable)**.** *For a binary variable, with $x \in \{0, 1\}$, with probability $p, 1-p$ resp., entropy is*

$$H(X) = -p \log_2 p - (1-p) \log_2 (1-p).$$

[12] https://forum.ircam.fr/collections/detail/improvisation-et-generativite/

This entropy is shown graphically in Fig. 2.2: Binary entropy as a function of probability p.

Entropy can be considered as a functional, which is a function that accepts another function & returns a number. In our case, function that entropy expression operates on is probability distribution, & it summarizes whole probability distribution into a single number that gives a sense of overall uncertainty of that distribution.

Example 5 (Chromatic Entropy)**.** *As an example of musical application, entropy can be considered as a measure of uncertainty in note distributions. Given 12 notes of a musical chromatic scale, possible to measure uncertainty of notes when assuming: each note is drawn independently from a scale. Tonal profiles were found by musicologists (Krumhansl & Schmuckler) [30] to determine a scale in terms of probability of notes, with most probable notes being 5th (dominant) & 1st (tonic) note of a scale. Of course, basic units of music do not have to be individual notes. These could be pairs of notes, or longer sequences, & growing length of melodies leads to increasingly more complex distributions. A Markov model is 1 such modeling approach that tries to account for such complexity.*

Another important notion that is built "on top" of entropy is *mutual information*. Mutual information measures relative reduction in uncertainty between 2 random variables X, Y, when outcome of 1 of them is know. To be able to define mutual information when need 1st to define *conditional entropy*, which is

$$H(X|Y) = -\sum_{x,y} P(x,y) \log_2 P(x|y) = -\sum_{y} P(y) \sum_{x} P(x|y) \log_2 P(x|y).$$

Note: probabilities appearing in 2 components in conditional entropy expression are not same. Logarithm contains conditional probability only, that is averaged both over x, y. I.e., conditional entropy estimates entropy of x for each possible conditioning value of y, averaged over all possible ys.

Thus, mutual information, difference in uncertainty about 1 variable when the other is known, can be written as:

$$I(X,Y) = H(X) - H(X|Y) = H(Y) - H(Y|X) = H(X) + H(Y) - H(X,Y).$$

Mutual information can be visualized as intersection or overlap between entropies of variables X, Y, as represented in Venn diagram in Fig. 2.3: Mutual information as a cross-section between entropies of 2 variables.

In following apply notion of mutual information to times series, as a way to quantify amount of information passing between past & present in a musical sequence. Thus, instead of considering long sequences, will focus on entropy of a single symbol in a sequence given its past. This entropy is in fact representing uncertainly of continuation. By exploiting an implicit knowledge of how to complete unfinished phrases, SHANNON played his famous game of having human subjects guess successive characters in a string of text selected at random from various sources [31]. SHANNON recorded probability of taking r guesses until correct letter is guessed & showed: these statistics can be effectively used as a bound on entropy of English. 1 of main problems with application of Markov theory "as is": true order of process is unknown. Accordingly, when N-gram models are used to capture dependence of next symbol on past N symbols, N needs to being fixed as a model parameter of choice. Main difficulty in Markov modeling is handling large orders, or even more interestingly, dealing with sources of variable-length memory. In music, reasonable to assume: memory length is variable since musical structures sometimes use short melodic figuration, sometimes longer motives, & sometimes rather long phrases can repeat, often with some variations. SHANNON's experiment establishes several interesting relations between language complexity & its continuation statistics. Since only certain pairs, triplets & higher N-grams tend to appear together, amount of guesses needed to predict next letter is drastically reduced for longer patterns. Moreover, show: one can obtain an efficient representation of language by encoding only continuation choices. This representation, SHANNON called a *reduced text*, can be used to recover original text, but requiring less storage. Later in book, use text compression schemes that encode letter continuations for larger & larger blocks of symbols as an effective way to capture variable memory Markov models in music & use then for generating novel musical sequences.

∗ 2.4.2. Natural Language Models. Natural Languages are often characterized by long structured relations between their words, arranged into sequences of letters or words arranged into dictionary entries. In following chaps, consider statistical modeling of sequences that is based on information theory & deep neural network learning. In both cases, goal of modeling: create a probability representation of long sequences of symbols or audio units that can be used for inference (finding probability of a sequences for classification purpose) or generation (producing more sequences of same kind). In this way, modeling music becomes a similar task to modeling natural language.

As a motivation & perspective about Markov & future models, discuss briefly general problem of modeling long temporal sequences, known as Neural Language models.

In general, a language model is a probability distribution over sequences of tokens. Traditionally, languages were represented using formal grammars, or learning equivalent automata that can generate specific strings that obey language rules. In modern approaches, problem in language modeling became 1 of learning a language model from examples, e.g. a model of English sentences from a large corpus of sentences.

Mathematically speaking, let $P(w_1, \ldots, w_n)$ denote probability of a sequence $(w_1, \ldots, w_n)$ of tokens. Aim of learning: learn such probabilities from a training set of sequences.

Consider related problem of predicting next token of a sequence. Model this a $P(w_n|w_1, \ldots, w_{n-1})$. Note: if can accurately predict probabilities $P(w_n|w_1, \ldots, w_{n-1})$, can chain them together using rules of conditional probability to get

$$P(w_1, \ldots, w_n) = P(w_1)P(w_2|w_1)P(w_3|w_1, w_2)P(w_n|w_1, \ldots, w_{n-1}).$$

Most learning approaches target language modeling as "next token" prediction problem, where learning system learns a mapping of $(w_1, \ldots, w_{n-1})$ input to w_n output. Such learning can be considered supervised learning, though there is no

need to meticulously label datasets (dán nhãn tập dữ liệu một cách tỉ mỉ) since future provides target "label". When done statistically, output of such system is a probability distribution over various values that w_n could take. Such output could be a high-dimensional probability vector, with hundreds of thousands to millions of options for a language model on English sentences.

Moreover, in case of language, input is not a fixed-length vector. Next token is often determined or largely depends on a long phrase, or in case of music, a motif that comprises a sequence of length n that varies. This phenomena is sometimes called "variable memory length" & is 1 of primary factors that make language, & music modeling, difficult.

Statistical approaches to estimating $P(w_n|w_1, \ldots, w_{n-1})$ when n is large must make some assumptions to make problem tractable. Even with an alphabet of just 2 symbols, 0 & 1, minimal training set size to reliably estimate $P(w_n|w_1, \ldots, w_{n-1})$ is of order 2^n. Such a big dataset is required in order to reliably visit or sample all possible input combinations in order to be able to predict next token reasonably in each case.

Most widely made assumption in statistical approaches in Markovian. Specifically,

$$P(w_n|w_1, \ldots, w_{n-1})$$

is approximated by a limited k-tuple,

$$P(w_n|w_{n-k}, \ldots, w_{n-1}).$$

This assumes: next token probabilities depend only on last k tokens. Most often k is a small number of order 1 or 2. This vastly simplifies modeling & estimation problem at cost of not being able to model long-range influences.

Early neural network approach to learning temporal predictions is so-called Recurrent Neural Network that will be covered in a later chap. Such models combine 2 important aspects that are made possible through end-to-end optimization approaches that are hallmark of DL – they learn distributed representations for input tokens, & at same time they learns dependencies or relations between these representation in time. Distributed representations, more commonly called word *embeddings*, map input onto a d-dimensional numeric space. 1 of motivations of such embeddings: semantically related tokens (words in language, piano-roll slices, or signal vectors), will be near each other in this space. Implicitly, this maps probabilities or likelihoods of appearance to distances in some, often non-euclidean geometry, but will not go into fascinating field of information geometry that actually tries to formalize this connection in rigorous mathematical ways.

In this chap consider a simple Markov transition probability model that begins with a set of discrete events (or states), into which all observations (e.g., notes o sound units or features) can be aggregated. Suppose there are R discrete categories into which all observations can be ordered. Can define a transition matrix $P = [p_{ij}]$ as a matrix of probabilities showing likelihood of a sound token staying unchanged or moving to any of other $R-1$ categories over a given time horizon. Each element of matrix p_{ij} shows: probability of even i in period $t-1$ moving or transitioning to event j in period t. Impose a simple Markov structure on transition probabilities, & restrict our attention to 1st-order stationary Markov processes, for simplicity.

Estimating a transition matrix is a relatively straightforward process, if can observe sequence of states for each individual unit of observation (i.e., if individual transitions are observed). E.g., if observe a sequence of musical chords in a corpus of music, then can estimate probability of moving from 1 chord to another. Probability of a chord change is given by simple ratio of number of times a chord appeared in different songs with same chord label & transitioned to another (or same) chord label. In such case, identify chord labels with states, but more sophisticated state-token mappings could be defined. More generally, can let n_{ij} denote number of chord appearances who were in state i in period $t-1$ & are in state j in period t. Can estimate probability using following formula:

$$P_{ij} = \frac{n_{ij}}{\sum_j n_{ij}}.$$

Thus probability of transition from any given state i = proportion of chord changes that started in state i & ended in state j as a proportion of all chords that started in state i. In exercises, prove: estimator given in above equation is a maximum-likelihood estimator that is consistent but biased, with bias tending toward 0 as sample size increases. Thus, possible to estimate a consistent transition matrix with a large enough sample. A *consistent* estimator converges in probability to true parameter value, & may be *biased* for small sample size, i.e., estimator's expected value differs from true parameter value.

As a demonstration of concept of Markov chains applied to music, interesting to note: a representation of chord progressions as a state machine is often used for teaching music composition. 1 such example is provided in Fig. 2.4: This diagram illustrates harmonic progression patterns common to Western music, where each Roman numeral indicates a triad built from scale degree corresponding to numeral. A base chord (I or i for major & minor, resp.) can transition to any chord. Subsequent chords tend to follow a pattern toward what are known as "pre-dominant" & "dominant" functions, leading back to base chord (or "tonic"). From a mathematical perspective, such models can be formally considered Finite State Automata, with transition probabilities learned from data. From this, it becomes possible to teach machines to compose chord progressions by creating & using such a transition map. One can think of Markov models as a probabilistic extension of Finite State Automata (FSA) that can be learned from data [32] instead of being manually designed from music theory rules.

- 2.5. Music Information Dynamics. In prev discussion of music perception & cognition in Chap. 1, introduced concept of Music Information Dynamics as a generic framework for analysis of predictive structure in music, relating it to a sense of anticipation by listener who actively performs predictions during audition process. Underlying assumption in MID framework: musical signals, from audio level to higher level symbolic structures, carry changing amounts of information that can be measured

using appropriate information theoretical tools. In terms of cognitive or computational processes, information is measured as a relative reduction in uncertainty about music events that is allowed through process of anticipation. As have seen in prev sect, def of mutual information basically says: information is carried between 2 random phenomena, & ability to decode or extract information is related to reduction in uncertainty about 1 signal from another. In MID approach consider musical past as 1 of random signals, & present instance of sound or music sect as 2nd random variable. Information is measured as a difference between entropy of 2nd variable & its conditional entropy give past. Moreover, important to note: conditional information of a random variable given its past $\Leftrightarrow$ *entropy rate* of that variable. Accordingly, information dynamics is a measure of information passing from past to present measured by difference in signal entropy before & after prediction, or equivalently difference between signal entropy & its entropy rate. In next 2 sects show now how MIR can be measured for symbolic sequences modeled by a Markov process, & for an acoustic signal whose structure & predictive properties are captured by its spectrum.

* 2.5.1. Information Rate in Musical Markov Processes. Equipped with basic knowledge of predictive modeling, interesting to consider case of Markov processes in music in terms of its anticipation structure. More formally, consider notion of information passing from past to present called Information Rate (IR) & characterize reduction in uncertainty when a Markov predictor is applied to reduce uncertainty in next instance of symbolic sequence. Moreover, same principle of information rate can be used for characterizing amount of structure vs. noise in acoustic signals. What differs between IR algorithms for 2 domains are their statistical models which in turn are used to estimate uncertainty, or entropy of signal based on its past.
  Assume a Markov chain with a finite state space $S = s_1, \dots, s_N$ & transition matrix $P(s_{t+1} = i | s_t = j) = a_{ij}$, which gives probability of moving from a state i at some point in time t to another state j at next step in time $t+1$. When a Markov process is started from a random initial state, after many iterations a stationary distribution emerges. Denoting stationary distribution as π, meaning of such stationary state: probability of visiting any state remains unchanged for additional transition steps, which is mathematically expressed as $\pi_{t+1} = A\pi_t$, where A: transitions matrix $A = [a_{ij}]$. Calculating entropy of stationary distribution & entropy rate is given following expressions

$$H(S) = H(\pi) = -\sum_{i=1}^{N} \pi_i \log_2 \pi_i,$$
$$H_r(S) = H_r(A) = -\sum_{i=1}^{N} \pi_i \sum_{i=j}^{N} a_{ij} \log_2 a_{ij}.$$

  Using these expressions, IR can be simply obtained as difference between entropy of stationary distribution & entropy rate.

$$\mathrm{IR}(S) = I(S_{t+1}; S_t) = H(\pi) - H_r(A).$$

  E.g., consider a Markov chain that moves repeatedly from state 1 to N, with negligible probability for jumping between non-adjacent states. Such a situation can be described by matrix A that is nearly diagonal (non-diagonal elements will be very small). One can verify: for such a matrix, $H_r(A) \approx 0$ & IR will be close to entropy of stationary state $\mathrm{IR} \approx \log_2 N$. This is maximal information rate for such process, showing: such a very predictable process has high amount of dependency on its past. If on contrary matrix A is fully mixing, with $a_{ij} \approx \frac{1}{N}$, then stationary distribution will be $\pi \approx \frac{1}{N}$ & entropy & entropy rate will be same $H(S) = H_r(A) = \log_2 N$. This results in IR close to 0, i.e., knowledge of prev step tell close to nothing about next step. Such process is very unpredictable. Applying notion of IR to Markov processes can guide choice of Markov processes in terms of anticipation or average suprisal that such a process creates.
  A more sophisticated measure was proposed by [12] that takes a separate consideration of past, present, & next future step. Termed *Predictive Information Rate* (PIR), measure is given as

$$\mathrm{PIR} \coloneqq I(S_{t+1}; S_t | S_{t-1}) = I(S_{t+1}; (S_t, S_{t-1})) - I(S_{t+1}; S_{t-1}) = H_r(A^2) - H_r(A).$$

  1st line is def of PIR. 2nd line is derived by using def of mutual information with a conditional variable & adding a subtracting $H(S_{t+1})$

$$I(S_{t+1}; S_t | S_{t-1}) = \cdots = -I(S_{t+1}; S_{t-1}) + I(S_{t+1}; (S_{t-1}, S_{t+1})).$$

  Last line is derived from entropy rate expressions of Markov process, where conditional entropy is used 1 time for a single step between S_{t+1} & S_t without S_{t-1} due to Markov property, & 2nd time for 2 steps between S_{t+1}, S_{t-1}. Given a transition matrix $A = [a_{ij}]$

$$H(S_t) = H(\pi) = -\Sigma_i \pi_i \log \pi_i,$$
$$H(S_{t+1} | S_t, S_{t-1}) = H(S_{t+1} | S_t) = H_r(A) = -\Sigma_i \pi_i \Sigma_j a_{ij} \log a_{ij},$$
$$H(S_{t+1} | S_{t-1}) = H_r(A^2) = -\Sigma_i \pi_i \Sigma_j a_{ij}^2 \log a_{ij}^2.$$

  Considering prev example of a fully sequential repetitive process with $A \approx I$ diagonal matrix, one finds: since $H_r(A) \approx H_r(A^2)$, resulting PIR is approximately 0. So in terms of considering mutual information for 1 step prediction given past since process is nearly deterministic, knowing past will determine both next & following steps, so there is little information passing from present to future when past is known.

A simple example demonstrating principal difference between IR & PIR is shown in Fig. 2.5: A simple example of a nearly repetitive sequence of 5 states represented as a Markov model of a probabilistic finite state machine. Nearly deterministic transitions are marked by continuous lines & nearly zero probability transitions are marked by dashed line. In this example a 5 state probabilistic finite state machine is constructed with nearly deterministic (probability near 1) for transitions each state i to state $i+1$, cyclically repeating from state 4 back to state 0, with negligible transition probabilities outside of sequence, i.e., negligible probabilities of staying in a state or jumping to a non-consequential states. This generative model can be analyzed as Markov model of 1st order, with transition matrix $A = [a_{ij}]$ & stationary distribution π $A_{4\times4} =$ `[an` ε `matrix]` & stationary distribution $\pi = [0.25, 0.25, 0.25, 0.25]$. For such border case have $H(\pi) \approx \log_2 5, H_r(A) \approx 0, H_r(A^2) \approx 0$. Accordingly have maximal IR $\approx \log_2 5$ & minimal PIR ≈ 0. This raises question which measure best captures our interest in sequence. According to IR, learning that sequence is repetitive & using this knowledge to reduce our uncertainty about future is maximally beneficial to listener, while according to PIR, this sequence is not "interesting" since once past outcome is known, very little information is passed between next present & next sample.

Not clear which measure represents better human sensation of surprise, as it seems: both make sense in different situations. If consider very predictable sequence as informative, then IR captures: but one might as well say: such sequence is dull in terms of its music effect, & listener will favor music that carries a lot of local or immediate information, but without past making it too redundant. In following chaps consider yet another way of computing anticipation (tính toán dự đoán) or average surprisal in case of variable memory processes. In Markov case, all information about future was contained in prev state. Since music builds long motifs that continue, diverge, or recombine in unexpected ways, being able to understand such processes requires models that capture longer term relations in non-Markov ways. Extending Markov models to longer memories is computationally hard since using tuples or N-tuples grows state space exponentially, & resulting models & their estimated transition matrices are difficult to compute or estimate. In next chap describe string matching models & use of compression methods to find motifs & calculate continuations of phrases at different memory lengths.

- Chap. 3. Communicating Musical Information. "Music is greatest communication in world. Even if people don't understand language that you're singing in, they still know good music when they hear it", said great American record producer, singer, & composer Lou Rawls. What is missing from his quote is def of what "knowing good music" means, & why this knowledge amounts to communication. Evidently music is a signal that passes from musician to their audience, so there is a sender-receiver structure to way music operates. & there is an aspect of "knowing" inherent in music listening process, which makes both encoder/musician & decoder/listener share some common understanding. In this chap explore basic premise of this book – ML of music, production, & computer audition are all parts of a communication process.

  - 3.1. Music as Information Source. basic 1-directional model of communication assumes: process of musical creation & listening comprises a series of activities that transmit information from a source/musician to a receiver/listener. Research on information theory of music is based on Shannon and Weavers (1949) communication model [33]. This basic communication model comprises an information source that is encoded into messages that are transmitted over a noisy channel, & a decoder that recovers input data from corrupted &/or compressed representation that was received at output of channel. 1st step in this information-theoretical approach to modeling of music & musical style is accepting idea: music can be treated as an information source. Accordingly, process of music generation is based on notion of information itself that essentially treats any specific message, & in our case any particular musical composition, as an instance of a larger set of possible messages that bare common statistical properties. A Claude Shannon, father of information theory, expressed in his work *The Mathematical Theory of Communication* [33], concept of information concerns possibilities of a message rather than its content: "That is, information is a measure of ones freedom of choice when one selects a message". Measure of amount of information contained in a source is done by means of entropy, sometimes also called "uncertainty", a concept covered in prev chap.

    Initial interest in information came from communications: how a message can be transmitted over a communication channel in best possible way. Relevant questions in that setting concerned compression & errors that happen during encoding or transmission process. Model of information source is generative in sense that it assumes an existence of a probability distribution from which messages are randomly drawn. Theory of information, as introduced by Shannon, encompassed 3 different domains of communication:

    1. Lossless compression: any information source can be compressed to its entropy.
    2. Error correction: all errors in communication can be corrected if transmission is done at a rate that is $\leq$ channel capacity. Concept of channel capacity is beyond scope of this book, as will focus on Thms 1 & 3 & their relation to music generation.
    3. Lossy compression: for any given level of reconstruction error or distortion, a minimal rate exists that will not exceed that error, & vice versa, for a given rate of transmission a coding scheme can be found that minimizes distortion.

    An example of a communications channel is shown in Fig. 3.1: Basic model of a communication channel that includes encoder, decoder, & a noisy channel that establishes uncertain relations between source messages & what appears in output at receiver's end. Since both source & channel errors are described probabilistically, information theory studies terms under which communication can be made more efficient. This incldues methods for encoding or compression of source in a lossless or lossy manner, & methods for coding that provide correction or more robustness to channel errors. In our work, mostly focus on compression aspects as a way for finding efficient representation of source data, & investigating tradeoff between compression & reconstruction error in case of reduced representation modeled as lossy encoding.

    Not all of Shannon concepts are directly applicable to music generation, so will limit ourselves to Thms. 1 & 3. Main concept borrow from Thm. 1: compression can be used as an efficient way to represent structure of information source.

Music generation using compression methods uses concepts from Thm. 1 in a reverse manner to original Shannon .design – instead of encoding a known information source, having an efficient encoding can be used for generating novel instances of same information source, & in our case composing novel music in same style. In original information-theoretic framework goal of encoding: find an efficient representation to a known probability distribution of source messages. Use compression as a way to learn distribution, i.e., given an efficient compression scheme can use it to produce more similar instances of same information source, which is our case is generating novel music by sampling from a its compressed representation. A good compression scheme is one which has learned statistical regularities over a set of examples, & can exploit these regularities in novel & creative ways. Moreover, Thm. 3 allows considering reduced musical representation through lossy compression that further reduces amount of information contained in a musical message in order to focus on more robust & essential aspects of musical structure. As such, statistical models derived from applying compression schemes to musical corpora in a particular style became also useful tool for *Stylometrics*, which is study of structures & variations that are characteristic of musical materials that may be broadly defined as "style".

Initial work on using information theory for music started in late 90's [34]. Today, problem of generative modeling in music has become 1 of formidable challenges (những thách thức to lớn) facing ML community & DL research in particular. Music is a complex, multidimensional temporal data (dữ liệu thời gian đa chiều) that has long-term & short-term structures, & capturing such structure requires developing sophisticated statistical models. For historical as well as conceptual reasons, thinking about music in terms of a complex information source provides powerful insights to building novel ML methods. Using communication theory in music research not only puts music into a transmitter-receiver communication setting that takes into account both composer & listener, but it also captures dynamics of music entertainment & creativity since co-evolution of musical knowledge that happens over time between musician & listener can be captured by methods of information theory, opening novel possibilities for man-machine co-creative interactions.

∗ 3.1.1. Style Modeling. Style Modeling using information theory started with application of universal sequence models, specifically well-known Lempel-Ziv (LZ) universal compression technique [35] to MIDI files [34]. Universality of LZ compression can be understood in terms of it not assuming any a-priori knowledge of statistics of information source, but rather having its asymptotic compression performance as good as any statistical model, e.g. Markov or finite state model. This is an empirical learning approach where a statistical model is induced through application of compression algorithms to existing compositions. Moreover, it captures statistics of musical sequences of different lengths, a situation that is typical to music improvisation (ngẫu hứng âm nhạc) that creates new musical materials at different time-levels, from short motifs or even instantaneous sonority (âm thanh tức thời), to melodies & complete thematic phrases. Specifically, universal modeling allows application of notion of entropy to compare sequences in terms of similarity between their statistical sources, as well as generation of novel musical sequences without explicit knowledge of their statistical model. One should note: "universality" in universal modeling approach still operates under assumption: musical sequences are realizations of some high-order Markovian sources. Information theoretical modeling method allows following musical operations to be applied to musical data:

  · stochastic generation of new sequences that have similar phrase structure as training sequence, creating musical results in between improvisation & original on motivic or melodic level, &

  · stochastic morphing or interpolation between musical styles, where generation of new sequences is done in a manner where new statistics are obtained by a mixing procedure that creates a "mutual source" between 2 or more training styles. Extent to which new improvisation is close to 1 of original sources can be controlled by mixture parameters, providing a gradual transition between 2 styles which is correct in statistical sense.

  Early experiments with style morphing [36] used a joint source algorithm [37], later using more efficient dictionary & string matching methods [38]. In addition to generative applications, universal models are used for music information retrieval, e.g. performing hierarchical classification by repeatedly agglomerating closest sequences (thực hiện phân loại theo thứ bậc bằng cách kết tụ nhiều lần các chuỗi gần nhất), or selecting most significant phrases from dictionary of parsed phrases in a MIDI sequence, selected according to probability of their appearance.

∗ 3.1.2. Stochastic Modeling, Prediction, Compression, & Entropy. Underlying assumption in information-theoretical approach to machine improvisation: a given musical piece can be produced by an unknown stochastic source, & all musical pieces in that style are generated by same stochastic sources. Finite Markov-order assumptions do not allow capturing arbitrarily complex music structures, e.g. very long structure dependency due to musical form; nevertheless, by allowing for sufficiently long training sequences that capture dependence on past, universal model can capture much of melodic structure of variable length in a musical piece.

  This connection between compression & prediction is a consequence of asymptotic equipartition property (AEP) [39], which is information-theoretic analog to law of large numbers in probability theory. AEP tells us: if $x_1, x_2, \ldots$ are independent & identically distributed random variables distributed with probability $P(x)$, then

$$-\frac{1}{n}\log_2 P(x_1, x_2, \ldots) \to H(P)$$

  where $H(x)$: Shannon Entropy of $x \sim P(x)$ (i.e., $H(x) = -\sum_x P(x)\log_2 P(x)$, where averaging is over all possible occurrences of sequences x).

  AEP property is graphically represented in Fig. 3.2: Asymptotic Equipartition Property of long sequences. Outer circle represents all possible sequences of length n, which are combinatorial possibilities depending on size of alphabets. Shannon's theory proves: in view of different probabilities of occurrence of each symbol (in simplest case, these are unbalanced heads or tails, known as Bernoulli distribution, with binary choice which takes value 1 with probability p & value 0 with

probability $q = 1 - p$), entropy of probability can be used to define an effectively much smaller set of outcomes whose probability will tend to be 1.

Remark 4 (Bernoulli Distribution)**.** *Bernoulli distribution is a discrete probability distribution that represents outcome of a single trial with 2 possible outcomes. Distribution is named after Swiss mathematician* Jacob Bernoulli*, who introduced it in late 1600s. Probability distribution function of Bernoulli distribution is* $P(X = k) = p^k(1-p)^{1-k}$*, where* $P(X = k)$*: probability of getting outcome* k *(either 0 or 1) &* p*: probability of outcome 1 on a single trial, with* $p \in [0, 1]$*. Mean (expected value) & variance of Bernoulli distribution are* $E[X] = p, \mathrm{Var}[X] = p(1-p)$*.*
Bernoulli distribution is often used as a building block for more complex distributions, e.g. binomial distribution, which models number of successes in a fixed number of independent Bernoulli trials.

This set of outcomes is called "Typical Set" & is denoted here as A_n, where n: number of elements in a sequence that needs to be sufficiently large. Moreover, all sequences of typical set are equiprobable, i.e., one can index these events in a way that assigns equal length codes to these events, & zero codes to any remaining events of non typical set. I.e., for long enough observations from a probability source, possible outcomes split into 2 – sequences that deviate from probability statistics, e.g. sequences that do not achieve expected mean number of heads or tails in a Bernoulli probability, will never occur, while those that do occur, happen with equal probability. No further structure exists in typical set, or otherwise additional compression could have been applied to exploit any remaining imbalance. In our case, for generative purposes, this means: one can generate new instances of a data source by accessing strings from its typical set using a uniform random number generator. Conceptually i.e.: sampling from a typical set turns white noise into outcomes that have desired statistical structure of that information source. For stationary ergodic processes, & in particular, finite order Markov processes, generalization of AEP is called Shannon-McMillan-Breiman theorem [39]. Connection with compression: for long x lower limit on compressibility is $H(x)$ bits per symbol. Thus, if can find a good algorithm that reaches entropy, then dictionary of phrases it creates can be used for generating new instances from that source. When algorithm has compressed phrase dictionary to entropy, any excess compressible structures have been eliminated. I.e.: dictionary is very efficient, so can sample from source by random selection from that dictionary.

- ∗ 3.1.3. Generative Procedure. Dictionary-based predictions methods sample data so that a few selected phrases represent most of information. Below described 2 dictionary based generative approaches: *incremental parsing* (IP) (phân tích gia tăng) & *prediction suffix trees* (PST). IP is based on universal prediction, a method derived from information theory. PST is a learning technique that initially was developed to statistically model complex sequences, & has found applications also in linguistics & computational biology. Both IP & PST methods belong to general class of dictionary-based prediction methods. These methods operate by parsing an existing musical text into a lexicon of phrases or patterns, called *motifs*, & provide an inference rule for choosing next musical object that best follows a current past context. Parsing scheme must satisfy 2 conflicting constraints: on 1 hand, maximally increasingly dictionary helps to achieve a better prediction, but on other, enough evidence must be gathered before introducing a new phrase to allow obtaining a reliable estimate of conditional probability for generation of next symbol. "Trick" of dictionary-based prediction (& compression) methods: they cleverly sample data so that only a few selected phrases reliably represent most of information. In contrast to dictionary-based methods, fixed-order Markov models build potentially large probability tables for appearance of a next symbol at every possible context entry. To avoid this pitfall, more advanced "selective prediction" methods build more complex variable memory Markov models. Although it may seem: Markov & dictionary-based methods operate in a different manner, they both stem from similar statistical insights.
  Use dictionary-based methods to model musical (information) source in terms of a lexicon of motifs & their associated prediction probabilities. To generate new instances (messages), these models "stochastically browse" prediction tree in following manner:
  - · Given a current context, check if it appears as a motif in tree. If found, choose next symbol according to prediction probabilities.
  - · If context is not found, shorten it by removing oldest (leftmost) symbol & go back to prev step.

  These steps iterate indefinitely, producing a sequence of symbols that presumably corresponds to a new message originating from same source. In some cases, this procedure might fail to find an appropriate continuation & end up with an empty context, or it might tend to repeat same sequence over & over again in an infinite loop. Specific aspects of how dictionary is created & how continuity is being maintained between random draws will be discussed next.

- ◦ 3.2. Lempel-Ziv Algorithm & Musical Style. In *A universal algorithm for sequential data compression* [40], Lempel & Ziv take a series of symbols from a finite alphabet as input, & build a tree of observed continuations of combinations of input symbols. This tree grows dynamically as input is parsed using what is called *incremental parsing* (IP) method. If input is a sample of a stationary stochastic process, LZ asymptotically achieves an optimal description of input in sense: resulting tree can be used to encode input at lowest possible bit rate (entropy of process). This implies: coding tree somehow encodes law of process; e.g., if one uses same tree (as it stands after a sufficiently large input) to encode another string, obeying a different law, resulting bit rate is higher than entropy.
  - ∗ 3.2.1. Incremental Parsing.
  - ∗ 3.2.2. Generative Model Based on LZ.
- ◦ 3.3. Lossy Prediction Using Probabilistic Suffix Tree.
- ◦ 3.4. Improved Suffix Search Using Factor Oracle Algorithm.

- Chap. 4. Understanding & (Re)Creating Sound.

- Chap. 5. Generating & Listening to Audio Information.
- Chap. 6. Artificial Musical Brains.
  - 6.1. Neural Network Models of Music. In this chap, describe musical applications of ML with neural networks, with emphasis on representation learning or feature learning (methods by which a system automatically discovers representations needed for feature detection or classification from original data).

    Musical neural networks are not a new research direction; a 1991 book by Peter M. Todd & Gareth Loy, Music & Connectionism, explores the topic. At time, *connectionism* referred to movement in cognitive science that explained intellectual abilities using artificial neural networks ANNs. Now, see neural networks, especially DL techniques, applied widespread among many domains of interest. Naturally, this has renewed interested in musical applications, with research in other domains spilling over to music as well, providing interesting & important results. With technical advances of algorithm optimization, computing power, & widespread data, possible for many to enjoy creating & experimenting with their own musical neural networks.

    Before visiting these applications, begin with a preliminary tour of vocabulary, algorithms, & structures associated with neural networks.

    * 6.1.1. Neural Networks. Consider a simplified model of brain as a network of interconnected neurons. Structure of linked neurons allows information (in form of electrical & chemical impulses) to propagate through brain. Organizing a series of mathematical operations & instructions as a *neural network* allows computer to function analogously to a brain in certain tasks.

      In a standard circuit, may expect an electronic signal to be passed forward, either as-is or in an amplified or reduced form. In a neuron, this message passes through an additional barrier; if its excitement passes a particular threshold, it will emit an action potential, or "fire", & otherwise, signal will not move forward. In this simplified model, can imagine a collection of these neurons to act like a collection of interconnected light switches; if right sequence are left "on", signals may pass through in a combination s.t. a correct destination "bulb" may light.

    * 6.1.2. Training Neural Networks.

      Definition 5. *A* neural network *is composed of a collection of scalar values which are used in mathematical operations to achieve a desired output. These scalar values are referred to as network* weights *or* parameters*. Can measure network's performance on a task using a function which returns a value called* loss*.*

      Goal of training a neural network: modify network weights s.t. forward passes of data through network lead to minimal loss. Achieve this through combining processes known as backpropagation [82] (used to calculate gradients of weights in neural network) & optimization (methods by which weights are modified to reduce loss). Some popular optimization methods include gradient descent (& its stochastic & mini-batch variants) & Adaptive Moment Estimation (Adam) [83].

      · 6.1.2.1. Loss Functions: Quantifying Mistakes. Imagine multiple pedagogical environments in which a person might learn to play piano. In 1 classroom, this person may have an excellent teacher who explains a technique in such great, concrete, retainable detail that the person is immediately able to execute performance technique after lesson. In another classroom, this teacher may briefly instruct student on intended technique, then ask student to perform. Student gives a suboptimal performance, & teacher points out student's mistakes. From these corrections (repeated as often as necessary), student then (eventually) learns proper performance technique.

        When it comes to neural networks, learning process is much like that of our 2nd piano student. Teacher (in this case, expected labels or *ground truth* associated with our data) must be cleverly turned into a quantifiable metric which can be used by learning algorithm to adjust network parameters. Call such a metric a *loss function*.

        Utilized loss functions will vary from problem to problem. 2 notably recurring loss functions are Mean Squared Error (MSE) & Categorical Cross-Entropy.

        1. Mean Squared Error is typically used when performing regression tasks (i.e., possible output exists on a continuum, like real numbers). For a given vector of length N, MSE generally follows formula

        $$\text{MSE} = \frac{1}{N}\sum_{n=1}^{N}(Y_{\text{output}_n} - Y_{\text{truth}_n})^2. \tag{54}$$

        2. Categorical Cross-Entropy is typically used when performing classification tasks (i.e.g, assigning of input to 1 of C possible classes). Cross-Entropy generally follows formula

        $$\text{CCE} = -\sum_{c=1}^{C}\mathbf{1}(c = y_{\text{truth}})\log P(c), \tag{55}$$

        where $\mathbf{1}(c = y_{\text{truth}})$ is an indicator function with value 1 when c is equal to true class, & 0 otherwise. Because classification problems expect as output a vector of class probabilities, this function essentially leverages properties of logarithm function to create strong values when model fails to predict known class (i.e., a log value approaching negative infinity, inverted by negative sign in front of equation) & weak, near-0 values when model correctly (or near correctly) predicts known class since $\log 1 = 0$.

· 6.1.2.2. Gradient Descent. After quantifying value of loss in form of a loss function, next desire to modify weights of our neural network s.t. loss is reduced. A common method to achieve this is gradient descent:

$$W_{\text{new}} = W - \alpha \nabla L(W). \tag{56}$$

For a fixed input x, value of loss function varies only w.r.t. weights W. Can compute a gradient by considering derivative of loss function w.r.t. each weight w_i in W. This gradient describes direction & magnitude of steepest ascent toward maximum value of loss; therefore, negative of gradient points in steepest direction toward minimum, with magnitude proportional to rate of change of loss in associated direction. Therefore, gradient descent equation adjusts weights in a direction toward minimal loss for input, scaled by hyperparameter α, also referred to as learning rate.

6.1.2.2.1. Chain Rule for MLP. In a feedforward neural network, input is passed through an initial layer where it interacts with network weights, & in case of a multi-layer network, these resulting values (often referred to as features) are passed to a next layer, & so on. Mathematically, a *multilayer perceptron* (our classic feedforward network) is essentially a composite of functions. To compute gradient, make use of chain rule. Consider basic example network shown in Fig. 6.1: An example multi-layer perceptron, consisting of input, network parameters (weights, bias), intermediate notes, & output., consisting of a single perceptron unit.

For some loss function L, desire to compute components of gradient $\frac{\partial L}{\partial w_i}$ for each weight w_i. However, in a multi-layer perceptron, each weight is effectively buried under a composition of functions which eventually lead to network output (& loss). Chain rule allows us to compute desired gradient: $\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial w_i}$, where y_j is an output in a preceding layer directly used in computing loss. This rule can be iterated over $\frac{\partial y_j}{\partial w_i}$ using preceding nodes, until input layer of network is reached.

As an example, consider network shown in Fig. 6.1. Provide to this network some training sample $\vec{x} = \{x_1 = 1, x_2 = 2\}$, associated with expected training output $z_{\text{truth}} = 5$. Assume: network parameters have been initialized s.t. all $w_i = 1, b_i = 1$. When network weights are applied to input, reach $z = w_5 y_1 + w_6 y_2 + b_3 = \cdots = 9$.

Using squared-error as a measure of deviation from expected value, there is a loss (our term for this deviation) of 16 associated with this sample. Now, would like to adjust weights to improve network performance. Rewrite loss expression as a function of weights.

$$L = (z - z_{\text{truth}})^2 = (w_5 y_1 + w_6 y_2 + b_3 - z_{\text{truth}})^2. \tag{57}$$

Considering influence of just 1 of our parameters, w_1, on network performance, chain rules states

$$\frac{\partial L}{\partial w_1} = \frac{\partial (z - z_{\text{truth}})^2}{\partial w_1} = \frac{\partial (z - z_{\text{truth}})^2}{\partial z}\frac{\partial z}{\partial w_1} = \frac{\partial (z - z_{\text{truth}})^2}{\partial z}\frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial w_1}. \tag{58}$$

Backpropagation is recursive algorithm by which compute derivative of final function value (in our case, L) w.r.t. an intermediate value in computation (e.g. any w_i) using recursive calls to compute values $\frac{\partial L}{\partial w_j}\ \forall w_j$ computed directly from w.

Now with possibility to compute a complete gradient over loss for each network parameter w, b, how do we use training data to improve network performance?

Consider 1st training sample. Loss value associated with this sample is $L = (9-5)^2 = 16$. Compute an exact value for our gradient term $\frac{\partial L}{\partial w_1}$:

$$\frac{\partial L}{\partial w_1} = \frac{\partial (z - z_{\text{truth}})^2}{\partial z}\frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial w_1} = 2(z - z_{\text{truth}}) w_5 x_1 = \cdots = 8. \tag{59}$$

Compute $\nabla L = \left[\frac{\partial L}{\partial w_1}, \ldots, \frac{\partial L}{\partial w_6}, \frac{\partial L}{\partial b_1}, \ldots, \frac{\partial L}{\partial b_3}\right]$: this is where gradient descent comes into play. Adjust weights according to formula given in (6.3), employing a learning rate of 0.1 for our example. $W_{\text{new}} = \cdots$. Our loss drops from 16 to $L = (3.72-5)^2 = 1.6384$ exactly like we wanted! Could continue our descent until approximately reach a local minimum (or reach suitable performance for particular problem).

While this example illustrates a method for calculating useful gradient, note: these gradients are computed in efficient, vectorized fashion in DL software.

· 6.1.2.3. A Note on Initialization. Before we can begin training, network weights must be initialized to some value so that a gradient can eventually be computed. Saw: our "set all weights to 1" approach in previous example may not have been too effective, especially since both halves of network effectively learned symmetrically, making them redundant. A better option: initialize weights randomly. This is a very important step, because if network weights are too small, corresponding nodes will never be activated by data, & weights may be too small for loss gradient to enact meaningful changes to weights. Similarly, if network weights begin too large, values may "explode", creating too strong of a response from corresponding nodes, & consume all of gradient descent effects to prevent learning on other network weights (especially when values exceed limits of your programming language's numeric representations & become NaN or infinite).

Xavier Glorot & Yoshua Bengio show: using an initialization randomly sampled from a uniform distribution bounded by $\pm\frac{\sqrt{6}}{\sqrt{n_\in + n_{\text{out}}}}$, now popularly termed "Xavier initialization", maintains near identical variances of weight gradients across network layers during backpropagation [84]. In this equation, n_{in}: number of edges into layer, & n_{out}: number of edges out of layer.

· 6.1.2.4. Modeling Nonlinearity. Our simplified network example is missing 1 major component which mimics "firing" behavior we seek from our neurons. This ability to propagate a signal when a certain threshold is passed is a *nonlinear*

behavior, as opposed to structured in our simplified example, which can effectively be reduced to a series of linear equations (matrix multiplications). *Activation functions* are applied after such standard multiplication layers to introduce thresholding. Importantly, activation functions are (mostly[13]) differentiable, which allows us to continue to use backpropagation to learn network parameters. This is exactly why binary activation function shown in Fig. 6.2: Binary activation function cannot serve as our activation function; given input 0, then unclear whether weights should face no change from associated derivative, or *infinite* change.

However, there exist plenty of other functions which share an ability to operate as a thresholding function while still maintaining near differentiability. Some common activation functions include:

1. Sigmoid (or Logistic) Function:
$$s(x) = \frac{1}{1 + e^{-x}} \tag{60}$$
this function tends to 0 on left & 1 on right, just like binary step. Critically, it is smooth & differentiable between these limits Fig. 6.3: Sigmoid activation function.
2. Hyperbolic Tangent: function $\tanh x$ is very similar to signmoid function in shape, but its larger range allows for a wider range of output values to propagate forward in network Fig. 6.4: Hyperbolic tangent activation function.
3. Rectified Linear Unit (ReLU):
$$\text{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0, \\ x & \text{if } x \geq 0. \end{cases} \tag{61}$$

With these tools in place, now possible to construct neural network architectures which model real-world phenomenon (i.e., patterns which are not necessarily linear) through connections of matrix-multiplication-like layers (often fully-connected tapestries of nodes) followed by activation functions, with successive layers repeated, modified, & tailored to content of task at hand.

◦ 6.2. Viewing Neural Networks in a Musical Frame. 1 application of neural networks for music: field of music information retrieval (tìm kiếm thông tin âm nhạc). Information retrieval deals with ability to search & find content by some query. A few examples of problems in music information retrieval include

1. Music Classification: sorting music based on properties e.g. genre, style, mood, etc.,
2. Music Recommendation: determining musical similarity between samples & corresponding suggestion of music based on user history,
3. Music Source Separation & Instrument Identification: separating a mixed audio signal to its original signals, as well as recognizing & extracting a particular instrument within a mixed signal,
4. Music Transcription: converting an audio recording to a symbolic notation, &
5. Music Generation: automatic creation of musical compositions or audio signals.

Many of above problems require an output used for making a prediction over multiple predefined possibilities. E.g., in classifying music, may be predicting 1 genre out of n possibilities. In generation, may be predicting next note, given n possible notes in instrument's range. Typically represent such outputs using a vector of size n, where each entry of vector represents confidence of model in selecting particular element. Often beneficial to treat these values way we treat probabilities, so that their numeric values can be used to generate output that matches some learned distribution. However, this requires values in vector to be between 0 & 1 (inclusive), with sum of all entries = 1. May wonder, why not just select element of greatest value, without normalizing vector? Such an *argmax* function is non-differentiable, so network will lose ability to learn from this structure. To overcome this, *softmax* function is used, notable for its normalization property, its ability to preserve numeric ranking among entries, & its differentiability.

Example 6 (Softmax Learning)**.** *Softmax equation is*
$$\text{softmax}(v)_i = \frac{\exp v_i}{\sum_{j=0}^{n-1} \exp v_j}. \tag{62}$$

Note: output of softmax is another vector of length n. Recall: vector output by softmax can be imagined as a probability distribution over classes.

Sometimes, want distribution to be very sharp, with nearly all probability associated with 1 particular value, indicating a strong confidence for a particular class. Other times, particularly with generating so-called creative output, may want a softer distribution, where values still preserve order but may introduce some ambiguity & lack of confidence (quantified by magnitude of probability). Put differently: there is a tradeoff between distribution flatness & novelty of repeated sampling. If repeatedly sample a uniform distribution to predict next note of a sequence, will have a very random (aleotoric – ngẫu nhiên) sequence. By contrast, if repeatedly sample a distribution with probability 1 for note G & 0 for all other notes, our piece will certainly not contain variance we come to expect in music.

Mathematical mechanism we can use to influence these distributions is referred as as temperature. Temperature is reflected in coefficient T, using which we rewrite softmax with temperature:
$$\text{softmax}_T(v)_i = \frac{\exp \frac{v_i}{T}}{\sum_{j=0}^{n-1} \exp \frac{v_i j}{T}}. \tag{63}$$

[13]Some activation functions have points of non-differentiability; in practice, these points are assigned derivative associated with nearest point. E.g., ReLU (rectified linear unit) is not differentiable at 0, so a value of 0 (left derivative) or 1 (right derivative) may be assigned at this point.

Can think of T as a scaling factor which increases (or decreases) distance between points of v on number line; concavity of exponential function then amplifies these differences, meaning: points pushed further apart on number line have even greater relative values to their neighbors, & points pushed inward have less strength. Accordingly, $0 < T < 1$ will create a sharper distribution, while $1 < T < \inf$ creates a softer distribution. In exercises at end of chap, show: distribution approaches uniformly when T approaches infinity.

Consider again our equation for categorical cross-entropy for classification:

$$\text{CCE} = -\sum_{c=1}^{C} \mathbf{1}(c = y_{\text{truth}}) \log P(c). \tag{64}$$

Using properties of logarithms, can reformulate this as:

$$\text{CCE} = -\log \prod_{c=1}^{C} P(c)^{(c=y_{\text{truth}})}. \tag{65}$$

In spirit of temperature, can reformulate this expression for cases when goal is not to learn a single-probability result, but rather a distribution over outcomes. Replace indicator function $\mathbf{1}(c = y_{\text{truth}})$ with target probability for class c, t_c:

$$\text{CCE} = -\sum_{c=1}^{C} t_c \log P(c). \tag{66}$$

Using properties of logarithms, can reformulate this as:

$$\text{CCE} = -\log \prod_{c=1}^{C} P(c)^{t_c}. \tag{67}$$

Term inside logarithm is in fact likelihood function of network parameters θ given observation $X = (x, y_{\text{truth}})$. This is stylized as $\mathcal{L}(\boldsymbol{\theta}|X)$ & it $= P(X|\boldsymbol{\theta})$.

Our measure of error depends on these output probabilities $P(c)$, which will refer to as output vector y. Also clear: these probabilities depend on input provided to softmax layer (values that become "scaled" into probabilities). Refer to these inputs as vector z. A natural question arises in order to backpropagate: how does error change w.r.t. z? I.e., what is gradient $\frac{\partial L}{\partial z}$? For any component,

$$\frac{\partial L}{\partial z_i} = -\sum_{j=1}^{c} \frac{\partial t_c \log y_i}{\partial z_j} = -\sum_{j=1}^{C} t_c \frac{\partial \log y_i}{\partial z_j} = -\sum_{j=1}^{C} t_c \frac{1}{y_j} \frac{\partial y_i}{\partial z_j}. \tag{68}$$

Show: $\frac{\partial y_i}{\partial z_j} = y_i(1 - y_i)$ when $i = j$, & $\frac{\partial y_i}{\partial z_j} = -y_i y_j$ when $i \neq j$. Use this result in gradient computation:

$$\frac{\partial L}{\partial z_i} = \cdots = -t_i + y_i \sum_{j=1}^{C} t_j = y_i - t_i. \tag{69}$$

With this gradient available, our network can learn a distribution associated with a sample using softmax function.

Now: have shown a way to learn by tuning weights, think through some immediate architectural ideas & consequences. Begin by considering 2D of contraction or expansion of our network. Can grow vertically (i.e., adding or subtracting neurons to a particular layer) or horizontally (adding additional layers to network).

* What happens when we shrink vertically? In this case, may *underparameterize* our model; i.e., trying to represent a complex problem or pattern with too few variables. When we do so, model may succeed in generalizing & perform well in most typical cases, but may fail for nuanced, less frequent cases.
* What happens when we grow vertically? In this case, may *overparameterize* our model; have created so many variables that each sample in our data may be fit to its own subset of variables. Model may fail to generalize to underlying pattern in data, so while it may perform perfectly in training, it will be unlikely to perform on unseen data.
* What happens when we shrink horizontally (i.e., make network more "shallow")? In this case, model will lose its ability to combine preceding pieces of information, a process refer to as generating higher-level features. In this type of abstract reasoning, may begin to observe a numerical piece & reason about its individual notes at early layers (e.g., hearing an F#, then another F#, then a G ...). In a middle layer, might combine some of these notes together to form an idea of a small motif (like a scale ascending a minor 3rd). At even deeper layers, motifs may be combined to form a phrase (a scale ascending a minor 3rd, then descending a perfect 5th ... ah, perhaps this is Ode to Joy!). Our model's ability to combine features & learn on these combinations is limited by depth of network.
* Might think: a solution to this shallow learning challenge: make network even deeper. But, what happens when a network is made too deep? This leads to what is commonly referred to as *vanishing gradient problem.*

◦ 6.3. Learning Audio Representations.

- 6.4. Audio-Basis Using PCA & Matrix Factorization.
- 6.5. Representation Learning with Auto-Encoder.
- 6.6. Exercises.

- Chap. 7. Representing Voices in Pitch & Time. Music & Audio AI is an interdisciplinary field between computer science & music. In comparison to more established AI fields e.g. natural language processing (NLP) & computer vision (CV), it is still in its infancy & contains a greater number of unexplored topics. As a result of this, DL models from natural language processing & computer vision are frequently applied to music & audio processing tasks. Take neural network architectures as an example. Convolutional neural networks (CNNs) have been extensively used in audio classification, separation, & melody extraction tasks, whereas recurrent neural networks (RNNs) & transformer models are widely used in music generation & recommendation tasks. Have seen various models of RNN, CNN, & transformer gradually outperform traditional methods to achieve new state-of-arts in recent years.

  In this chap, introduce 2 basic neural network architectures that are used to learn musical structures in time & pitch or time & frequency, namely Recurrent Neural Network (RNN) & Convolutional Neural Network (CNN). However, extent to which these architectures are capable of handling music & audio tasks remains debatable. In this chap, illustrate similarities of music & audio problems with NLP & CV tasks by comparing music generation with text generation, & music melody extraction with image semantic segmentation. This provides an overview of how RNN, CNN, & transformer architectures can be applied to these problems. Then, examine uniqueness of music & audio problems in comparison to tasks in other fields. Previous research has demonstrated that certain designs for these architectures, both in terms of representation & multitaskstructure, can result in improved performance in music & audio tasks.

  RNN caught attention of music research as early as 1990s, after opening up field of neural networks to modeling language & sequences. In [91], Peter Todd describes a connectionist approach to algorithmic composition based on Elman & Jordan networks.[14] Early approaches were mostly monophonic (đơn âm) & were difficulty to train. With advent of long short-term memory networks (LSTMs) & success of modern RNNs in Natural Lagrange Processing, these tools became 1st go-to neural architecture for modeling sequential structure in music. Inspired by success of CNNs in computer vision, & thinking about music as a time-frequency 2D structure, CNNs were also applied to music. Big difference between 2 architectures is in their treatment of relations between frequencies & times. In RNN model is purely sequential; all temporal relations are summarized into tokens that are used to represent either single or multiple voices. Thus, tokenization will be our 1st topic of study. In CNN, convolutional filters are applied both in time & across notes (in MIDI pianoroll representation) or frequencies (in audio spectral representation). Main difficulty in using CNNs for music: limited span or "receptive field" of convolutional kernel (filter). While RNN has a decaying memory that is arbitrarily long, CNNs field of view is limited to size of filter. Using dilated (giãn nở) CNNs (kernels that skip samples) & hierarchical structures, e.g. Wavenet, became possible ways to overcome limitation of short-time scope of filter response. Some hybrid variants were proposed as well, including biaxial-RNNs that apply RNNs both to time & frequency/notes axis, or combinations of CNN-RNN that try to combine local feature extraction aspect of CNNs with long term memory aspect of RNNs.

  - 7.1. Tokenization. How do we pass musical input to a neural network? This is question addressed by tokenization (phân loại), process of converting musical information into smaller units called *tokens*.

    Tokenization is used widely in Natural Language Processing. To illustrate variety of tokenization approaches, consider 2 possible tokenizations of Einstein's sentence, "If I were not a physicist, I would probably be a musician." If 1 such letter-based tokenization, create a representation for each token {I,f,w,e,r,n,o,t,a,p,h,y,s,c,u,l,d,b,m}. In a different word-based tokenization, create a representation for each token {If, I, were, not, a, physicist, would, probably, be, musician}. What are possible benefits of different tokenization methods? If have a model with which we would like to learn how to spell or construct words, letter-based tokenization will be much more useful. On other hand, if would like to learn relationship between words in a sentence for proper grammatical & semantic usage, word-based tokenization is more appropriate. Similarly, way we tokenize musical data has a direct impact on what our models can efficiently learn from data, & types of output they can produce.

    Now thinking in terms of input & output, if want to learn end-to-end a complete MIDI file, why is MIDI not best representation? Network would need to learn formatting specifications of MIDI encoding, which are not related to musical information itself. So, can tokenize musical information itself to help our network learn musical patterns – in essence, extract & translate musical building blocks from MIDI data.

    MidiTok [92] is a Python package which implements multiple popular MIDI tokenization schemes:

    1. MIDI-Like [93]: This scheme contains 413 tokens, including 128 note-on & note-off events (1 per MIDI pitch), 125 time-shift events (increments of 8ms to 1 sec), & 32 velocity events. MidiTok added additional quantization parameters in their implementation to allow for different numbers of tokens to accommodate different memory constraints or precision goals.
    2. Revamped (Đã được cải tiến) MIDI-Derived Events (REMI) [94]: REMI maintains note-on & note velocity events from MIDI-like. Note-off events are instead replaced with Note Duration events; a single note is therefore represented by 3 consecutive tokens (note-on, note velocity, & note duration). Authors propose this to be advantageous in modeling rhythm than placing a series of time-shift events between a note-on & note-off in MIDI-like. Additionally, problem of a "dangling" note-on event is resolved, should model fail to learn that note-on & note-off events must (naturally) appear in pairs.

[14] Elman network, which became modern RNN, uses feedback connections from hidden units, rather than from output in Jordan design.

3. Structured [95]: Unlike MIDI-Like, with Structured tokenization, tokens are provided in standardized order of pitch, velocity, duration, time shift. This encoding cannot accommodate rests or chords, so it is only suitable for modeling particular musical examples.
4. Compound Word [96]: At this point, may have considered an odd artifact of MIDI encoding. Oftentimes, there are events which are intended to be simultaneous, but due to 1-instruction-at-a-time nature of MIDI, they are provided as a sequence. E.g., a combination of note-on instruction & its associated note velocity would be provided as a sequence of 2 sequential instructions in MIDI, but these instructions are intended to describe same event. This idiosyncrasy (tính cách lập dị) is corrected in Compound Word. These compound words are supertokens – i.e., combinations of tokens for pitch (vocabulary size 86), duration (17), velocity (24), chord (133), position (17), tempo (58), track (2), & family (4). Including "ignore" tokens for each category, there are a total of 338 vocabulary items to form a Compound Word supertoken. "Family" token allows for distinction between track events, note events, metric events, & end-of-sequence events. Similar encodings include Octuple [97] & MuMidi [98].

MidiTok uses 4 parameters to tokenize for each of implemented schemes:

1. Pitch Range: minimum & maximum pitch values (integers in MIDI).
2. Number of Velocities: how many MIDI velocity levels (range 0 to 127) to quantize to.
3. Beat Resolution: sample rate per beat, & beat ranges for which sample rate should be applied. Specifying a high resolution for notes with a short number of beats can improve precision of time shifts & note durations.
4. Additional Tokens: opportunity to specify additional tokens to be included in addition to standard set. This includes:
   (a) Chord: a token which indicates a chord is being played at particular timestep.
   (b) Rest: explicit definition of a time-shift event. Helpful to insert a tokenized rest for understanding input/outputk, as opposed to simply placing next note at a later timestep since rests are an important, notated feature of most music.
   (c) Tempo: a token to represent current tempo; a range of possible tempos & number of tempos to quantize within this range should be provided to MidiTok.

∗ 7.1.1. Music Generation vs. Text Generation. What is relation between Music Generation vs. Text? Music generation's objective: create musical content from input data or machine representations. Music content can be in form of symbolic music notes or audio clips. Typically, input data serves as generative condition, e.g. previous musical context, motivation, or other musical information (e.g., chord progression, structure indication, etc.). Text generation's objective: generate natural language in response to a condition. This condition could be a couple of initial words, conclusion of a sequence, or change in mode (e.g., professional, neural, creative, etc.). By & large, music & text generation have a similar conditional probability of decomposing formats:

$$\mathbf{y} = y(y_1, y_2, \ldots, y_T),\ p(\mathbf{y}|\mathbf{c}) = \prod_t p(y_t|\mathbf{y}_{1:t-1}, \mathbf{c}), \tag{70}$$

where y denotes token in different sequential tasks, & c denotes different conditions as discussed above. In symbolic music generation, this token can be a single musical note or a latent embedding of a measure/phrase of music. In text generation, this token can be a single character, a word embedding, or even a sentence embedding with prompt learning.
Music & text generation both follow a similar probability model. As a result, successful RNN & transformer models for text generation can be easily transferred to music generation tasks. Earlier models of music generation, e.g. MidiNet [99] & Performance-RNN, directly applied RNN-family architectures (LSTM, GRU) to melody generation task.

∗ 7.1.2. Representation Design of Music Data. While text & music generation share a time sequential model, music generation is distinguished by 2 characteristics: (1) spatiality of musical content; & (2) hierarchy of musical conditions.
Musical note is fundamental element of music. As illustrated in Fig. 7.1: Music score of Mozart Sonata K331, 1st variation (top), & text data from internet (bottom). Black part is given context & red one is text generation., music sequences contain information beyond temporal dimension – vertical or spatial relationship of notes. Each time step in a text sequence contains a single word or letter; in a music sequence, each time step contain multiple notes. Whereas piano piece depicted in figure is a relatively straightforward form of music composition, multi-track compositions (e.g., symphony, concerto, polyphony) contain more vertical relationships between notes & even different instruments. This data violates 1 of fundamental assumptions of RNN & transformer architectures (based on positional encoding) – sequence naturally contains sequential temporal information for each step of input. Simply encoding music data results in many steps of input containing same temporal information. As a result, architecture introduces erroneous conditional dependencies on various music notes mathematically.
2nd, as illustrated in Fig. 7.2: Music structure analysis of a jazz composition – Autumn Leaves, by Bill Evans., music has a more complex structure than text. A piano piece is composed of paragraphs, phrases, sub-phrases, harmonic functions, melodies, & textures. It becomes even more complex when expressiveness structure is added (volume changes, rhythmic changes). While such a structure may exist in a typical novel's text generation (prologue, scenes, characters, paragraphs, & content), frequently absent from a large number of text generation samples. More importantly, music's structure is dynamic as well as hierarchical in nature. As illustrated in Fig. 7.2, melody & texture change with each beat; harmonic functions change with certain intervals; phrases & sub-phrases undergo even greater changes. A significant flaw in current music generation model is absence of long-term structure in generated samples (e.g., repetition & variation of musical motives). In following secs, discuss some of designs of music data representations & discuss how they affect performance of music generation models.

- 7.1.2.1. 3-state Melody Tokens. Basic representation of music data is shown in Fig. 7.3: 3-state melody tokens of music data. It encodes music into 3 types of tokens:
  1. Pitch, musical pitch of note, range from [A0, C8].
  2. Hold, duration of note, to maintain last pitch tone.
  3. Rest, rest note.

  MidiNet [100], Performance-RNN [101], & Folk-RNN generated music directly from tokenized representation. Consequently, MusicVAE [102], Music InpaintNet [103], EC2-VAE [104], & Music SketchNet [105] combined it with variational autoencoder (VAE) to generate latent variables to handle a piece of music as basic generation unit. This is an intuitive representation of music that retains essential musical information. Model's inputs & outputs are directly representation itself, eliminating need for additional structures to decode. Disadvantage of this representation is also self-evident: it cannot describe polyphonic music (i.e., multiple notes played simultaneously), but only monophonic melodies, because its temporal information has already been bound to sequence's position, as is case with text generation.
- 7.1.2.2. Simultaneous Note Group. To encode polyphonic music as input, must consider overlapping notes structure. To address this issue, Pianotree-VAE [106] proposes a vertical representation – simultaneous note group. As illustrated in Fig. 7.4: Simultaneous note group encoding of music data, by Pianotree-VAE [106], it groups simultaneous notes by grouping multiple note events that share same onset, where each note has several attributes e.g. pitch & duration. Model's input is further encoded using RNN with simultaneous group as fundamental unit, which is further transformed into hidden states by GRU in Pitch-Duration Encoder; in decoder, model is expanded with another GRU layer for predicting pitches of each group & an final fully connected layer for predicting duration of each note. This results in final latent z vector representation, that captures structure of music for every 2 bars without overlap. Paper compares midi-event tokens, simultaneous note structure & latent z for a downstream music generative applications, showing: simultaneous notes & latent vector representation are superior to midi-event encoding.
- 7.1.2.3. MIDI Message. Although above 3 representations have their own ways to process music data, they miss a crucial part of encoding other music information besides notes. MIDI message encoding, from Music Transformer [107], breaks this limitation by introducing more music content information to music generation task. As shown in Fig. 7.5: MIDI message encoding of music data, MIDI message is a 4-state token:
  1. NOTE-ON (128 pitch classes): starting signal of 1 note.
  2. NOTE-OFF (128 pitch classes): ending signal of 1 note.
  3. TIME-SHIFT (32 interval classes): duration mark to move timeline.
  4. VEL (128 volume classes): velocity changes.

  In REMI [108], MIDI message is expanded into 6 states, including note onset, tempo, bar line, phrase, time, & volume. With this representation, model is able to acquire additional musical data & discover new patterns, resulting better well-structure & varied musical composition.
- 7.1.2.4. Music Compound Word. Music Compound Word [109] proposes a new structural design for MIDI messages, complete with an improved input & output network module. As illustrated in Fig. 7.6: Compound word encoding of music data, cf. MIDI-message encoding., state of music input in a compound word transformer is comparable to that of a MIDI message, but input & output mechanisms are entirely different.

  Music compound word can be adapted into different scenarios of music generation by changing types of states along with decoding layers. Fig. 7.7: Compound word encoding of 3 music generation scenarios. shows 3 types of music compound words for
  1. music generation with performance labels
  2. multi-track music generations with multiple instruments
  3. music generation with structural labels

  Besides that, because music compound word contains position state (e.g., beat, bar, etc.), transformer architecture can be used without any positional encoding. This begins to differentiate music generation task from text generation task when same architecture is used.
- 7.1.2.5. Discussion: Signal Embedding vs. Symbolic Embedding. In previous chap, saw an embedding of an audio signal that can be used to obtain salient audio components, e.g. audio basis, compression, or noise removal. Used analogy between PCA & linear AE to get an intuition that embedding is a subspace where meaningful part of signal resides. In particular, showed for sinusoid+noise example: this embedding is very similar to signal transformation using Fourier methods. Projection to a lower dimensional space of sinusoids not only allows discarding noise, but can also be used as a representation of "meaningful" part of signal. Representation is latent space found by AE & representation found by Fourier transform share similar basic vectors.

  But can such embedding also be applied to MIDI? Autoencoding of categorical variables is a poorly defined problem – after all, neural network embeddings assume: similar concepts are placed nearby in embedding space. These promixities can be used to understand music concepts based on cluster categories. Then, they can also serve as an input to a ML model for a supervised task & for visualization of concepts & relations between categories.

  Natural to represent categorical variables as 1-hot vectors. Operation of 1-hot encoding categorical variables is actually a simple embedding where each category is mapped to a different vector. This process takes discrete entities & maps each observation to a vectors of 0s & a single 1 signaling specific category. 1-hot encoding technique has 2 main drawbacks:
  1. For high-cardinality variables – those with many unique categories – dimensionality of transformed vector becomes unmanageable.

2. Mapping is completely uninformed: "similar" categories are not placed closer to each other in embedding space.

So embeddings could be obtained using either a manually designed representation, or could try to obtain it from data. This type of contextual embedding is what commonly refer to as "representation learning". In case of temporal sequences, might want to obtain a representation of a complete sequence. In many musical applications, RNN is used to pre-process sequences before representation learning happens.

- 7.2. Recurrent Neural Network for Music.
  - 7.2.1. Sequence Modeling with RNNs. RNNs are a way to model time sequences, an early neural network application. They are used to represent a function where output depends on input as well as previous output. I.e., model must have memory for what has happened before, maintaining some representation of interval state.

    Can imagine RNN to be learning probability of a sequence of tokens, $P(w_1, \ldots, w_n)$, from a collection of example sequences used in training.

    Once this language model is learned, can solve problem of predicting next token of a sequence (i.e., computing $P(w_n|w_1, \ldots, w_n)$). This can be framed as a supervised learning problem, in which sequence $(w_1, \ldots, w_{n-1})$ acts as input which corresponds to output w_n.

    Essentially, an RNN is 2 neural networks: one which acts on input, & one which acts on hidden state.

$$h_t = f(h_{t-1}, x_t) \text{ (general form)}, \tag{71}$$
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}X_t) \text{ (NN with activation)}, \tag{72}$$
$$o_t = W_{hy}h_t \text{ (output)} \tag{73}$$

    In these equations, h represents hidden state & x represents input.

    Example 7 (Hello World from an RNN)**.** *To introduce RNN architecture, build an example which learns to say "Hello World". More accurately, this model will learn to output a sequence of letters that match sequences it has been trained on (in this case, "Hello World"). Given a starting letter (or set of letters), model generates next character of output. Continue building this example through sec as cover foundations of RNNs.*

    1st, how shall we represent input to this network? Because mathematical operations in neural network must operate on numbers, must covert our letter data ("h", "e", "l", "l", "o") into numerical data.

    For simplicity, work in a hypothetical world with a reduced alphabet: E-D-H-L-O-R-W.

    1 scheme for encoding letter "h" would be to replace letter with its positional index in alphabet (2, in case of our reduced alphabet [& using 0-indexing]). However, this introduces an issue as value of different numbers will carry different apparent weights; mathematically, recognize $5 > 2$*, but would not say: letter R has a different implicit significance than letter H.*

    Solution for this is to use previously introduced 1-hot encoding. In this schema, letters are treated as categorical (rather than quantitative) variables, so any 1 letter is replaced by a 7-vector of 0s – with exception of index associated with letter we represent, where we find a one. In this way, every letter has an equal magnitude in representation, while information distinguishing between letters is preserved.

    Important: this method fails for large (or expanding) vocabulary sizes. To overcome these particular challenges (or simply to reduce input dimensionality), technique of embedding can be used, a concept recurring through our discussion in prior & following chaps.

    But even though we've learned to encode single letters to vectors, input is still not ready for RNN. RNNs are meant to operate on sequences *rather than singular input. So, input* length *may actually vary depending on use cases. In Hello World example, use an input length of 3 (so a* 3×7 *matrix when considering encoding for each sequence element), but in other problems this length can grow to 1000s of values. Explore challenges associated with large input sequence lengths in this chap.*

    What about output? Like other classification problems, our network will output a fixed-length probability vector, representing probability that next token is a particular word from a predefined vocabulary set. Think about how dimensionality of this vector may change between domains; for forming English words, have 26 letters, but for forming English language, there are > 171000 *words in dictionary alone (excluding all sorts of proper nouns). For Western music, have 12 chroma, but for a single instrument, vocabulary may span entire playable range, & for some instruments (e.g. piano), our vocabulary grows exponentially with number of possible simultaneous notes possible in a chord.*

    Start RNN example by preparing 1-hot encoding scheme. 1st, import necessary packages:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import tensorflow.keras.backend as K
```

    Define our letter encoding:

```
def letter_to_encoding(letter):
    letters = ['e','d','h','l','o','r','w']
    vec = np.zeros((7),dtype="float32")
    vec[letters.index(letter)] = 1
    return vec
```

Can see effects of this encoding with this sample:

```
for letter in "hello":
    print(letter_to_encoding(letter))
```

which gives output

```
[0. 0. 1. 0. 0. 0. 0.]
[1. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 1. 0. 0. 0.]
[0. 0. 0. 1. 0. 0. 0.]
[0. 0. 0. 0. 1. 0. 0.]
```

Remark 5 (NQBH). *If use* `import keras.backend as K` *then get error*

```
Traceback (most recent call last):
  File "/home/nqbh/advanced_STEM_beyond/machine_learning/Python/hello_world.py", line 53, in <module
    print_output = K.eval(out)
                   ^^^^^^
AttributeError: module 'keras.backend' has no attribute 'eval'
```

then change it to `import tensorflow.keras.backend as K`. *Link:* https://stackoverflow.com/questions/58047454/how-to-fix-module-keras-backend-tensorflow-backend-has-no-attribute-is-tf .

For our example, use a single-layer RNN. Network will take an input with shape $3 \times 7 \times N$, *where 3 represents input sequence length, 7 represents size of data vector associated with each input, & N: number of samples in a training batch. RNN layer is a Keras SimpleRNN; each of 3 nodes of this RNN will create a 7-dimensional output (again used to represent likelihood associated with a single character in our alphabet). Though each length-3 input will map to a length-3 output, take only 3rd (final) output to represent predicted "next character" of our phrase. Had we left off* `return_sequences` *parameter, network would output only this final character (but for sake of example, it will be interesting to see entire predicted sequence). To normalize raw output into a probability-like representation, apply a softmax activation function prior to output.*

Model definition:

```
model = keras.Sequential()
model.add(keras.Input(shape=(3,7,)))
model.add(layers.SimpleRNN(7, activation="softmax", return_sequences=True))
model.summary()
```

Exercises at end of chap asks you to think about why this network has 105 trainable parameters. As a hint, recall: input to RNN node is made up of input vector, plus hidden state, plus a bias term, & these inputs are fully connected to generated output.

In below code, create our training data. In our hypothetical world, only phrase that exists is "helloworld", so train network on fragments of this phrase. For those following along by running code, can see in below example: for each 3-letter portion, output matches next predicted letter following each letter of input.

```
train_text = "helloworld"*30
def generate_train_set(train_text, as_words=False):
    x_train = []
    y_train = []
    for i in range(len(train_text) - 4):
        if as_words:
            x_train += [[train_text[i:i+3]]]
            y_train += [[train_text[i+1:i+4]]]
        else:
            x_train += [[letter_to_encoding(letter) for letter in train_text[i:i+3]]]
            y_train += [[letter_to_encoding(letter) for letter in train_text[i+1:i+4]]]
    if as_words:
        print(x_train[0][:5])
        print(y_train[0][:5])
    else:
        print(np.array(x_train)[0,:5])
        print(np.array(x_train).shape)
    return np.array(x_train), np.array(y_train)

generate_train_set(train_text, True)
x_train, y_train = generate_train_set(train_text)
```

So what happens when we generate an output phrase using untrained network?

```
# What happens when generate an output phrase using untrained network?
letters = ['e','d','h','l','o','r','w']

seed = "hel"
result = "hel"
input_data = np.array([[letter_to_encoding(letter) for letter in seed]])
model.get_weights()

for i in range(7):
    out = model(input_data)
    print_output = K.eval(out)
    for row in print_output[0]:
        next_letter = letters[np.argmax(row)]
    result += next_letter
    print(result)
    input_data = np.array([[letter_to_encoding(letter) for letter in result[-3:]]])
```

After running above code, reach sth like this:

```
hele
helee
heleee
heleeee
heleeeee
heleeeeee
heleeeeeee
```

Model weights are not yet optimized; train with following lines:

```
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.01), loss=tf.keras.losses.CategoricalCr
model.fit(x_train, y_train, batch_size=24, epochs=300, verbose=0)
```

After training, try again to produce output:

```
input_data = np.array([[letter_to_encoding(letter) for letter in seed]])
model.get_weights()
result = "hel"
for i in range(7):
    out = model(input_data)
    print_output = K.eval(out)
    for row in print_output[0]:
        next_letter = letters[np.argmax(row)]
    result += next_letter
    print(result)
    input_data = np.array([[letter_to_encoding(letter) for letter in result[-3:]]])
```

Give following output from our simple RNN:

```
hell
hello
hellow
hellowo
hellowor
helloworl
helloworld
```

Is a recurrent neural model of music appropriate for generating music that "sounds like" other music by a particular composer or within a genre? An ideal statistical model would learn from all possible sequence inputs to create an expected musical output, but size of possible inputs is v^n, where v: vocabulary size & n: input sequence length. Even considering just 88 piano keys & 10 notes in a sequence creates 2.785^{19} combinations!

To overcome this, think back to Markov assumption introduced in Chap. 2. If approximate: a prediction depends only on a small number of preceding tokens, problem becomes much more tractable to model. But, in doing so, lose ability to model long-range effects or earlier tokens.

So, how do neural models maintain long-range effects?

A 1st attempt to solve this problem could consider combining sequence of input tokens into 1 mega-token, for which a corresponding prediction can be made. A neural network can be applied over this input matrix, with dimensions of

vocabulary size or embedding vector length by number of elements in input sequence. Note: have a built-in limitation with such a model: tokens further away than number of elements in input sequence cannot influence model output.

Pros of RNNs. Recurrent neural networks overcome this constraint; i.e., recurrent neural networks can learn extensive long-range influences despite limited-length input sequences. It does so by encoding sequential history into a hidden state, which will refer to at time t by h_t in this discussion (can imagine "h" to stand for "history"). This state h_t is updated with input x_t, then used in predicting output y_t. Encasing this history state in a black box, RNN appears to receive a sequence of vectors $x_0, x_1, \ldots, x_T$ as input, & produces a sequence of vectors $o_0, o_1, \ldots, o_T$ as output. Internally, can imagine process to begin with computation of history state:

$$h_t = F(h_{t-1}, x_t), \tag{74}$$

then prediction of output using updated history:

$$o_t = G(h_t). \tag{75}$$

Relationships between inputs, multi-dimensional hidden history state, & outputs can be learned with neural network training procedures, as studied in previous chaps. These relationships are represented by functions F, G, which may be composed as:

$$F(h, x) = A_1(W_{hh} * h + W_{xh} * x), \tag{76}$$

$$G(h) = A_2(W_{ho} * h). \tag{77}$$

In this notation, A_i stands for an **a**ctivation function, & W_{IO} stands for a matrix of learnable **w**eights which map from dimensionality of matrix's **i**nput vector class to matrix's **o**utput vector class. W_{xh} describes influence of input on next state, W_{hh} describes influence of current state on next state, & W_{ho} describes influence of next state on next output. An illustration of relationship between input, hidden state, & output are shown in Fig. 7.8: In its compact form (left), & RNN consists of a learned hidden state h, which is driven by both current input & past values, & itself drives output o, as described in RNN equations. To make explicit development of state h over time & its relationship to input & output, can imagine RNN in its "unfolded" form (right).

If ignore intermediate outputs $o_1, \ldots, o_{N-1}$, this is really just a feedforward network, with initial inputs h_0 & x_1, & with additional inputs concatenated to outputs of particular intermediate layers. However, in a traditional feedforward network, weights between layers are uniquely learned. In case of an RNN, these weights are *shared*, i.e., same weights will act on input x_t regardless of time, & same weights will act on 1 state h_t to create next state (or to create output) regardless of time.

How do we learn in this shared-weight scenario? In previous backpropagation example, derivative of loss w.r.t. a particular weight could be calculated, & weights adjusted accordingly. Now, for same weight, have multiple possible derivatives since there are multiple inputs influenced by that parameter!

Without diving too deep into theory, consider an approach by which we take average of these gradient contributions. I.e., for any of N instances of w_i in network, which can call w_{i_n}, can compute $\frac{\partial L}{\partial w_{i_n}}$. Then, when deciding how to modify shared parameter w_i, use

$$\frac{\partial L}{\partial w_i} = \frac{1}{N} \sum_{n=1}^{N} \frac{\partial L}{\partial w_{i_n}}. \tag{78}$$

In principle, average should move us closer toward an optimal solution since largest amount of "mistake" explained by weight applied at a particular layer will have largest amount of influence on this averaged gradient component.

– Về nguyên tắc, giá trị trung bình sẽ đưa chúng ta đến gần hơn với giải pháp tối ưu vì lượng "sai sót" lớn nhất được giải thích bằng trọng lượng áp dụng cho một lớp cụ thể sẽ có ảnh hưởng lớn nhất đến thành phần gradient trung bình này.

There is 1 critical issue with this approach (& in fact, with any networks sufficiently deep), referred to as *vanishing gradient problem*, as discussed in Chap. 6. This problem is particularly notable for modeling sequential data since it effectively limits how far back in time a model can learn to "reach" to influence its output; while RNN is capable of representing long sequences of data, it loses ability to learn at length.

– Có 1 vấn đề quan trọng với cách tiếp cận này (thực tế là với bất kỳ mạng nào đủ sâu), được gọi là *sự cố gradient biến mất*, như đã thảo luận trong Chương 6. Sự cố này đặc biệt đáng chú ý đối với việc mô hình hóa dữ liệu tuần tự vì nó hạn chế hiệu quả khoảng thời gian mà một mô hình có thể học để "tiếp cận" nhằm tác động đến đầu ra của nó; trong khi RNN có khả năng biểu diễn các chuỗi dữ liệu dài, thì nó lại mất khả năng học theo chiều dài.

Methods to mitigate problem of long-range influence have been developed, including Gated Recurrent Unit (GRU) & Long Short-Term Memory (LSTM) network.

Example 8 (Gated Architectures: Long Short-Term Memory Network & Gated Recurrent Unit)**.** *A Long Short-Term Memory network, or LSTM, is an RNN architecture designed to deal with vanishing gradient problem that plagues traditional RNNs, causing difficulty in learning long-term dependencies commonly found in music & audio.*

LSTM architecture introduces a new type of unit called memory cell *that can store information for long periods of time. This represents "long" component of LSTM. Memory cell is controlled by 3 gates:*

1. *input gate,*
2. *forget gate,*
3. *output gate.*

Each gate is typically activated using a sigmoid function which determines how much of input should be let through to memory cell, how much of memory cell's contents should be forgotten, & how much of memory cell's contents should be output.

"Short-term" component of LSTM is accounted for by addition of a hidden state at each time t, similar to an RNN. Hidden state is updated based on current input, previous hidden state, & (a key distinction) memory cell output. This state can help to carry time forward from each step of input sequence as well as call upon long-term information.

All 3 gates of memory cell receive current input & previous hidden state to inform their evaluation. By using memory cells & gates to control flow of information, LSTMs are better able to learn long-term dependencies in input sequence.

Like classic RNN & LSTM, a Gated Recurrent Unit architecture, or GRU, processes input 1 sequence element at a time & maintains a hidden state updated by a combination of both current input & previous hidden state. What differentiates GRUs is addition of new types of gating mechanisms to control flow of information through network. GRUs use 2 types of gates:

1. *an update gate, to decide how much information from previous state should be retained & incorporated into new hidden state, &*
2. *a reset gate, to decide how much of previous hidden state should be discarded.*

Benefit of GRU is its simpler architecture, with fewer parameters than a comparable LSTM. This helps to avoid overfitting on small datasets, a common problem to overparameterized models. Both LSTMs & GRUs offer a gated mechanism for modeling long-term dependencies, & choice between architectures ultimately falls to considerations related to task at hand, e.g. dataset size, expected time dependencies, & computational constraints.

∗ 7.2.2. Sequence-to-Sequence Modeling with RNNs. Sometimes seek not to learn only an estimated distribution for a sequential pattern, but a mapping from 1 sequence to another. E.g., pretend you are a jazz saxophonist, & you attend a friend's classical piano recital (buổi độc tấu piano cổ điển). A striking melody from 1 of their pieces catches your attention, & you'd like to improvise on it when you practice later that day. In this case, you are taking an input sequence of melody in a classical context, & translating it to a jazz melody (which may have different styles of ornamentation). Such a problem is solved using sequence-to-sequence (often abbreviated seq2seq) modeling.

Formally, can describe such problems this way: Given vocabularies V_z, V_x & training data pairs (z_n, x_n) independently & identically distributed from some distribution p, can we estimate distribution $p(x|z)$?

Framing above problem in this notation, V_z are possible notes from piano, & V_x are possible notes from saxophone. Can collect samples of same melody played once by classical pianist z_n & again by jazz saxophonist x_n, & use these to learn a distribution of what we expect jazz saxophone to sound like x for a given piano melody z. There are numerous other problems we can frame as `seq2seq`, including problem of transcription (given an audio sequence, returning same represented music in symbolic notation).

∗ 7.2.3. Performance RNN. Now, extend our RNN exploration to a musical example. Performance RNN [101] is a network which generates performance-quality MIDI piano output[15] In fact, Performance RNN was project which introduced MIDI-Like tokenization scheme introduced in previous sect. To understand Performance RNN, examine its components:

· Data: Performance RNN is trained on Yamaha e-Piano Competition dataset (introduced in Appendix D). This is a collection of ≈ 1400 MIDI files, generated by recording keypresses of highly skilled classical pianists. Accordingly, music is expressive in both timing (rubato) & velocity (dynamics).

· Tokenization: MIDI-Like tokenization is employed. At each step, input to RNN is a single 1-hot 413-dimensional vector representing 413 possible note-on, note-off, velocity, & time-shift events.

· Model Architecture: 3 hidden layers of LSTMs with 512 cells.

· Training & Loss: Model is trained using RNN technique of *teacher forcing*.

Example 9. Teacher forcing *helps models reach stable convergence during training by providing correct (rather than predicted) output as input to each sequential step. E.g., imagine a model meant to learn a C-major scale (C-D-E-F-G-A-B-C). In a non-forced training, if model is given input C to 1st cell, & incorrectly predicts an output of E, E would propagate as input to 2nd sequential is forced to D to match ground truth. While this is beneficial to efficiently learn given pattern, it should be noted: this reduces model's ability to learn from variations on pattern which may not be represented in training data (but may exist otherwise). For generative models which seek to perform & innovate (rather than replicate), this could be detrimental in some cases.*

Log loss (i.e. categorical cross-entropy) is loss function used to drive training backpropagation.

· Output Generation: beam search.

Example 10 (Beam Search)**.** *Beam search is a heuristic search algorithm used to find most likely sequence of output given an input sequence. It is a variant of* breadth-1st search *algorithm (thuật toán tìm kiếm theo chiều rộng), which attempts to* balance exploration & exploitation*, a problem common to search algorithms as well as AI subfield of reinforcement learning.*[16]

Basic idea behind beam search: generate multiple possible outputs at each step, but keep only top-K most probable candidates. This value K is referred to as beam width*. Algorithm then proceeds to next step, generating a new set of candidate outputs for each of previous K candidates, continuing until a stopping criterion is met.*

[15] Performance RNN is a product of Google Magenta, introduced in Appendix A. Can learn more about Performance RNN on authors' blog https://magenta.tensorflow.org/performance-rnn, & run code yourself with instructions available in their GitHub Repository https://github.com/magenta/magenta/tree/main/magenta/models/performance_rnn.

[16] While reinforcement learning is outside scope of this book, readers are encouraged to consider: generation in a creative or imaginative sense requires this same consideration of exploring new ideas vs. utilizing known patterns.

In theory, beam search algorithm eliminates low-probability candidates early on to speed up & improve quality of output sequence search. However, beam search can also suffer from problem of getting stuck in local optima*, where most likely candidates at each step do not necessarily lead to overall most likely output sequence or reflect desired long-term behavior. E.g., consider a diatonic melody (giai điệu diatonic) which starts on C, & will choose next note from C major scale. Predictive model may output probabilities for each of these notes:* $(C, 0.1), (D, 0.1), (E, 0.1), (F, 0.2), (G, 0.3), (A, 0.15), (B, 0.05)$. *If use a beam width of 3, select top 3 most likely possible sequences, CF, CG, CA, & associate these with respective probabilities* $0.2, 0.3, 0.15$.

Now, next 3 sets of next sequence options are generated, conditioned on each *of these selections; this may be sth like*

1. *CF:* $(C, 0.05), (D, 0.025), (E, 0.25), (F, 0.3), (G, 0.3), (A, 0.05), (B, 0.025)$
2. *CG:* $(C, 0.3), (D, 0.025), (E, 0.25), (F, 0.05), (G, 0.3), (A, 0.05), (B, 0.025)$
3. *CA:* $(C, 0.2), (D, 0.025), (E, 0.2), (F, 0.3), (G, 0.2), (A, 0.05), (B, 0.025)$

Score for each of resulting possibilities is probability of prior sequence times probability of next token. E.g., considering 1st row above, would have as possibilities $(CFC, 0.01), (CFD, 0.005), (CFE, 0.05), (CFF, 0.06), (CFG, 0.06), (CFA, 0.01), (CFB, 0$
Repeat this $\forall 3$ *beams, & from resulting scores, select again top 3 candidate sequences – these are only sequences which will continue propagating forward. Once desired sequence length is reached, beam search is concluded & available sequence assigned most probability can be selected.*

* 7.2.4. SampleRNN. In introduction of *SampleRNN: An Unconditioned End-to-End Neural Audio Generation Model* [64], Mehri et al. highlight primary challenges associated with audio generation:
  1. A large discrepancy (sự khác biệt, sự tương phản) between dimensionality of audio signal & semantically meaningful unit (e.g., 1 word is, on average, around 6000 samples at a 16 kHz sample rate).
  2. Structure occurs at various scales (correlations between neighboring samples as well as samples thousands of units apart).
  3. Compression of raw audio to meaningful spectral (or other handcrafted) features.
  4. Necessary correction of audio after decompression (e.g., consider complexity of Griffin-Lim for reconstructing phase).

  SampleRNN seeks to reduce this complicated, engineering-heavy pipeline by allowing a neural network to learn these compression & reconstruction processes as well as relationships between many samples comprising meaningful audio signals.

  SampleRNN approaches this task using a hierarchical framework, where lower-level modules operate on individual samples, but higher-level modules operate at a larger timescale & lower temporal resolution. This is 1 example approach toward learning audio (or musical) trends which operate at different scales to overcome limitations fo a rudimentary sample-by-sample RNN framework.

  How are model outputs of SampleRNN evaluated (&, in general, how can we evaluate performance of generative audio models)? In this case, **AB Testing** with human rates is employed. Pairwise combinations of SampleRNN with other comparative models (including WaveNet) are presented to human raters, who then express preference for "A" over "B" (or vice versa). Then, one can compute a number of statistics, most common being percentage of times that samples from model in question were preferred over alternative. Subjectivity of evaluating artistic or preference-based generative models creates difficulties in presenting conclusive metrics, but AB preference testing is 1 such approach that is generally accepted & useful.

◦ 7.3. Convolutional Neural Networks for Music & Audio. In this sec, introduce Convolutional Neural Networks (CNN) for 2D representation of music & audio. CNNs are popular for image processing & use an implementation of a local filtering approach over neighboring pixels. Idea of CNN, in a way similar to adaptive filtering approaches encountered in DSP part of book: data is used to estimate or train filter parameters to perform some desired task. Thinking about CNN as a filter, must note:

* CNNs implement multiple local filters,
* They are often trained for feature extraction & inference, i.e. some classification tasks & not as an audio filter,
* Local properties (finite impulse response) of CNNs are often implemented in 2D, which in our case will span both time & frequency, or time & voices in spectral or piano-roll representations, resp.

One must note: 1 of big advantages of CNNs in visual domain: many of features in images are local & translation invariant. I.e., want to detect lines, corners, or part of faces & so on, regardless of exact position where these visual elements appear in picture. In music, local relations might exist between notes or frequency elements in time, & also across frequencies or notes, e.g. in case of harmonic or polyphonic (multi-voice) relations. Big difficulty: such relations are usually much longer or distant in time then what a normal visual patch would be. Talk about capturing long term relation when discuss attention in RNN & transformers in a later chap. Regardless of limitations, CNNs have been found to be very useful tools for analysis of time-frequency structures, i.e., representing voices in time.

* 7.3.1. What is a CNN? CNNs are defined by presence of *convolutional* layers. Many signal processing textbooks explain mathematical details of convolution, & here present material in a conceptual manner. Convolution is result of iterating through & operating upon portions of a signal using another signal (referred to as a convolutional filter or kernel); nature of this interaction is typically a sum of elementwise multiplications. An example is shown in Fig. 7.9: 2D convolution operation. A kernel (middle) is applied to input image (left) in a dot-product operation where each grid cell of kernel is multiplied by its corresponding cell in input image, then added to a running sum for a given kernel position. This sum becomes output value in feature map (right). Kernel then slides across image to next position, until every position of image has been

covered. During convolution, filter is applied to every sub-region of signal that it passes over, generating a scalar value for each position it assumes. Pseudo-image created from mosaic of these scalar values is called a *feature map*. When defining convolution operation, possible to vary spatial pacing of filter as it passes through image (referred to as *stride*) as well as extent to which filter is used at edges of signal (commonly addressed via *padding*). Stride, filter size, & number of filters are sufficient to define a convolutional layer within a CNN architecture.

These filters become a powerful tool for extracting meaningful information from a signal. Readers who have studied signal processing are likely familiar with a basic ensemble of filters (low-pass, high-pass, bandpass, etc.) which allow certain frequency components to be extracted from a composite signal. When it comes to 2D convolution, filters can be used for many informational & transformational purposes; as 2 examples of many possibilities, some filters can extract meaningful qualities e.g. edges (location in image with intense transitions from dark to light or vice versa), & others can transform an image by taking local averages (referred to as image smoothing or blurring).

In above cases, image filter is strictly defined, then applied to image. E.g., a Laplacian operator, which acts as a great detector of strong intensity changes by approximating a local 2nd derivative, always features a strong, positive value at central location, surrounded by a ring of weaker, negative values. Other times, it may be useful to use filters which adapt to problem you intend to solve. Such is case with object template matching, where a filter is designed by selecting values that exhibit a strong value when convolved with template object,, & weak values elsewhere. Will reserve details about implementation & theory of such filters for a digital image processing textbook, but driving principle should be remembered: filters can be tuned to highlight particular types of information from an image. Selecting or handcrafting these filters often leads to 2 situations:

- Selected theoretical filter is so general that it can only find particular low-level features (e.g., local intensity changes in a particular direction).
- Handcrafted filter is so specific that it can only find exact matches to particular high-level entities (e.g., a very specific pattern in a spectrogram from a violinist playing a phrase, which would not have same strength of response if a different violinist played same phrase).

This is where *convolutional neural networks* come into play. Instead of selecting or crafting convolutional filters, each filter's parameters are *learned* during training. Each convolutional layer consists of a bank of n filters, defined by their height & width. These filters are eventually optimized for task at hand through usual methods of backpropagation & gradient descent. Convolutional layers are stacked (creating *deep* neural networks), & through this stacking mechanism, features are extracted & combined at low, mid, & high levels of abstraction (e.g., edges combine to become geometric shapes, which combine to become recognizable object patterns).

- 7.3.1.1. Pooling. Consider effect of a bank of 10 3×3 filters acting on a 20×20 pixel single-channel image. This layer would require 90 parameters. After this convolutional layer (& assuming appropriate padding), output feature map would maintain a spatial dimension of 20×20, but with a depth of 10 features. If this convolutional layer is comprised of an additional 10 filters, each filter would now need $3 \times 3 \times 10$ parameters, requiring 900 parameters for layer.

  Following this trend, can see: requisite parameters quickly explodes deeper data propagates through network. As a means to reduce this issue, can use technique of *pooling*. Pooling provides a means of reducing spatial dimensionality of a feature map, operating on assumption: reduced feature map will still exhibit useful, informative patterns available at full scale.

  In a pooling scheme, a square region of an image is replaced by a single value intended to represent region. Example selected values may be region's maximum (*maxpooling*) or average (*averagepooling*).

  As convolutional & pooling layers progress (with padding reduced), eventually reach a spatial dimension of 1 with a deep set of features; from this point, network is typically concluded with a series of fully connected layers.

* 7.3.2. CNN Audio Example: WavenNet. Slightly contrary to above discussion, begin examples of CNNs with WaveNet, a model which actually does not act on image-like 2D representations (piano roll, spectrogram, etc.); WaveNet is a deep generative model used to process raw audio waveforms.

  WaveNet has been shown to generate speech mimicking human voice[17] & synthesize musical audio signals. WaveNet directly models raw audio waveform sample-by-sample, in direct contrast to historical methods of speech synthesis where phonemes (âm vị) from a bank can be concatenated to create words, or vocoders can produce synthetic sound fragments. This allows for improved realism & flexibiity in model, but why has this approach been a long-standing challenge?

  Large issue at play: rate of prediction necessary to support sample-level modeling; while phonemes may change on scale of milliseconds, audio files change value at a rate of 16000 samples (or more!) per sec. To make matters worse, have seen from our earlier discussion on deep neural networks that long-term patterns become increasingly hard to model further back they exist in time (relative to current predictive position); when we are moving at 16000 samples per second, there is a huge gap between our current position & information we may need to effectively model sound!

  Researchers at Google DeepMind had a solid basis for taking leap of faith to model massive sequential lengths; previously, research term developed PixelCNN & PixelRNN. These neural networks acted on images, but there are themes drawn from this work that translate to audio domain:

  - Recurrent Layers, as studied in previous sect, parameterize series of conditional distributions.
  - Residual Connections are used to share information between different network depths.
  - Generated samples are crisp, varied, & globally coherent, 3 properties which are of equal concern when generating audio.

[17] Examples are available from DeepMind at https://www.deepmind.com/blog/wavenet-a-generative-model-for-raw-audio.

After seeing promising results that an image could be generated pixel-by-pixel (i.e., row-wise sequentially, as opposed to a more expensive unsampling method), similar techniques were applied to audio data, which is naturally sequential.
Dilation (Sự giãn nở) of WaveNet architecture varies at each layer; this allows WaveNet to maintain an enormous *receptive field*, referring to range of time from original signal subtended by network to generate its next output. This WaveNet architecture has become a standard tool to other works in audio processing. Considering original goals of authors, it proved effective in modeling realistic voices for text-to-speech translation, & even showed promise in learning patterns across different speaking voices. Authors additionally trained WaveNet on classical piano recordings, & results are unique piano recordings which sound realistic, with minute amounts of noise seemingly an artifact of a phantom recording process rather than spurious musical decisions.
– Sự giãn nở của kiến trúc WaveNet thay đổi ở mỗi lớp; điều này cho phép WaveNet duy trì một *trường tiếp nhận* khổng lồ, đề cập đến phạm vi thời gian từ tín hiệu gốc được mạng lưới bao bọc để tạo ra đầu ra tiếp theo của nó. Kiến trúc WaveNet này đã trở thành một công cụ tiêu chuẩn cho các tác phẩm khác trong xử lý âm thanh. Xem xét các mục tiêu ban đầu của các tác giả, nó đã chứng minh được hiệu quả trong việc mô hình hóa giọng nói thực tế để dịch văn bản thành giọng nói, & thậm chí còn cho thấy triển vọng trong các mẫu học tập trên các giọng nói khác nhau. Các tác giả cũng đã đào tạo WaveNet về các bản ghi âm piano cổ điển, & kết quả là các bản ghi âm piano độc đáo nghe có vẻ thực tế, với lượng tiếng ồn nhỏ dường như là sản phẩm của quá trình ghi âm ma thay vì các quyết định âm nhạc giả tạo.

* 7.3.3 CNN Audio Example: UNet for Source Separation. Earlier introduced problem of *source separation*, isolation of individual sounds (or sources) in an audio mixture[18] Techniques e.g. active source estimation can help to solve for mixture of source energies between spectral frames of audio signal, & can even be extended with information from an audio-aligned written score for enhanced estimation [111]. CNNs have also become a popular approach toward this task, as their convolution & transpose (reverse) convolution operations can be used to generate masks to filter spectral representations of audio into individual sources.
Such a *mask* layer can be overlaid on original spectrogram (or mathematically speaking, element-wise multiplied) to allow only information associated with a particular source to pass through. From this masked spectrogram, sound of a single source can be approximately reconstructed. To do this, size of output of CNN must match size of input (i.e., mask must be same size as spectrogram so that it perfectly covers). This naturally requires an encoder-decoder structure to network, so that learned parameters can be structured in a way that rebuilds downsampled input to its original size. UNet architecture [112] is a popular architecture for such problems due to its symmetric U-shape, use of residual "skip connections" to pass information between spatial resolutions, & inclusion of layers of parameters at decoding side to assist in reconstruction.
2 examples (of many) which use this technique can be found in works by Geng et al. [113] & Kong et al. [114]. Geng et al. use a UNet variant called "gated nested" UNet (GNUnet), where there are nested series of layers "filling in" center of U. Original U "backbone" has gating units applied, a mechanism originally proposed for RNNs which control information flow throughout network to allow for modeling more sophisticated interactions. Outputs of GNUNet are used to create a time-frequency spectral mask to generate 2 masks: 1 for singing, 1 for accompaniment – sự đệm đàn. 2 masks can be multiplied by magnitude & phase spectra of mixture, then transformed back into time-domain signals simultaneously, as opposed to networks which isolate & extract only 1 source. Also using a UNet architecture, Kong et al. additionally estimate complex ideal ratio masks (i.e., decoupling masks into a mask for magnitude & a mask for phase) to decrease reconstruction error. Result is a system effective at separating vocal tracks, bass, drums, & more.

  - 7.4. Pretrained Audio Neural Networks.
  - 7.5. Exercises.

- Chap. 8. Noise Revisited: Brains that Imagine.
  - 8.1. Why study generative modeling?
  - 8.2. Mathematical Definitions.
  - 8.3. Variational Methods: Autoencoder, Evidence Lower Bound.
  - 8.4. Generating Music & Sound with Variational Autoencoder.
  - 8.5. Discrete Neural Representation with Vector-Quantized VAE.
  - 8.6. Generative Adversarial Networks.
  - 8.7. Exercises.

- Chap. 9. Paying (Musical) Attention.
  - 9.1. Introduction.
  - 9.2. Transformers & Memory Models in RNN.
  - 9.3. MIDI Transformers.
  - 9.4. Spectral Transformers.
  - 9.5. Attention, Memory, & Information Dynamics.
  - 9.6. Exercises.

[18] A fantastic resource to learn about this problem & its applications comes from a tutorial by Ethan Manilow, Prem Seetharaman, & Justin Salamon at 2020 International Society for Music Information Retrieval conference [110].

- Chap. 10. Last Noisy Thoughts, Summary, & Conclusion.
  - 10.1. Music Communication Revisited: Information Theory of VAE.
  - 10.2. Big Picture: Deep Music Information Dynamics.
  - 10.3. Future of AI in Music & Man-Machine Creative Interaction.
- Appendix A. Introduction to Neural Network Frameworks: Keras, Tensorflow, Pytorch. Throughout this book & accompanying exercises, have shared examples of code drawing from a variety of neural network frameworks. In this Appendix, briefly introduce these programming tools, as well as some common pitfalls & recommended practices.

  3 libraries imported in this book are Keras, TensorFlow, & PyTorch.

  - Keras is an open-source neural network library written in Python. Designed to be user-friendly, flexible, & modular, which makes it easy for users to build & experiment with different types of neural networks. Keras has a high-level API that allows users to quickly build & train neural networks using pre-built layers & models. This makes it a popular choice for beginners who are just getting started with neural networks.
  - TensorFlow is another open-source ML library that was developed by Google Brain. Also written in Python & is designed for large-scale ML projects. In fact, Keras was built on top of a TensorFlow backend, allowing for abstraction of certain implementation details while still maintaining access to powerful features of TensorFlow. TensorFlow has a low-level API that allows users to build custom models & operations, as well as a high-level API that is similar to Keras. TensorFlow has become 1 of most popular DL libraries because of its scalability, flexibility, & robustness.
  - PyTorch is another popular DL library that was developed by Facebook (now Meta). Also written in Python & is known for its dynamic computation graph, which allows users to build & modify neural networks on fly. PyTorch has a user-friendly API that is similar to Keras, which makes it easy for beginners to get started. PyTorch is also known for its ease of debugging, which makes it a popular choice for researchers who are developing new DL models.

  A note on libraries: As you may have seen from programming examples & notebooks, Python libraries are changing all time, & many scripts we write draw on multiple such libraries. Sometimes a change made to 1 version of a library affects others which depend on it. For this reason, important for users to be aware of which versions of a library they should install to run a program, & try to provide this information when possible.

  A frequent problem: have to run many programs with conflicting dependencies; while 1 program might want to use TensorFlow version 1, another may need TensorFlow version 2. As a possible tool, virtual environments provide a way for us to manage which libraries our python interpreters "sees" when we execute program. These can be created & managed through `virtualenv` package, or alternatively through Anaconda environment manager. There are plenty of other virtual environment tools that exist, & most serve same purpose. In these setups, have our default (base) environment, & then can additionally create virtual environments. These virtual environments are blank slates until one installs desired libraries within environment. Recommended practice to make a virtual environment to run your code, then install your associated libraries to this virtual environment before running program. This will help you to avoid package conflicts between programming examples, problems, & projects.

  In general, recommend readers consult documentation of any library used in case of errors introduced during implementation.
- Appendix B. Summary of Programming Examples & Exercises. Accompanying this book are a set of programming examples & exercises, available at https://github.com/deep-and-shallow/notebooks & formatted as Jupyter notebooks. Here, provide a brief description of each notebook & a recommended chap of content for which notebook could be paired.
  - Chap. 1: **Introduction to Music Representation** lets readers convert same music through a variety of representations & visualizations (MIDI, Piano Roll, Audio) & observe strengths, weaknesses, & challenges associated with such representations & conversions.
  - Chap. 2: **Probability** invites readers to implement Mozart's Dice Game & experiment with random number generation to create music from noise.
  - Chap. 3: **Markov Chain & LZify** guides readers in exploring Markov Models to generate pop music chord progressions, then provides an exercise in implementing & using Lempel-Ziv algorithm to parse, analyze, & generate a musical melody.
  - Chap.4: **Discrete Fourier Transform** lets readers construct & use DFT matrix to quickly compute Fourier transform of audio signals.

    Spectrograms STFT & Griffin Lim will introduce readers to generation of Short-Time Fourier Transform plots, & readers can implement Griffin Lim algorithm to reconstruct phase when discarded from frequency representations of signals. Readers are invited to record their own short audio samples for these exercises.

    Speech Formants & LPC lets readers play with source-filter model of human voice, investigating combinations of frequencies to create vowel sounds.
  - Chap. 5: **VMO Audio Oracle** uses VMO model to construct a recombinant model from a given saxophone target recording. Then, readers will use VMO query function to "drive" this model using another sound file. Query sound file will contain an accompaniment part of a song, so result will be a new saxophone improvisation over that song.
  - Chap. 6: **PCA with Linear Autoencoder** walks readers through training an autoencoder for purpose of comparing its output to that of PCA, using basic sinusoidal audio input as training data.

- Chap. 7: **Generating Music with RNN** helps readers to practice with Keras to create a generative model based on music created by stochastic Mozart Dice Game.

  Parallel CNN–RNN explores use of CNN-RNN for genre classification, comparing its performance to similar CNN-only for RNN-only models.

- Chap. 8: **GAN, chroma (MIDI) & pix2pix** explores a style transfer application that tries to change musical texture of a piece while maintaining harmonic structure. In order to do so, readers train a pix2pix type of model that learns relations between chroma & musical texture (distribution of note). For this purpose readers extract chroma from MIDI data & learn a generator that complements notes from a given texture.

- Chap. 9: **Attention & Transformers** introduces readers to effects of attention in Transformer model, visualizing effects of models with & without attention on weight applied to various aspects of input, drawing on examples from both NLP & symbolic music.

• Appendix C. Software Packages for Music & Audio Representation & Analysis. In this Appendix, briefly introduce some of programming frameworks that are used for loading, processing & analysis of audio & symbolic music data, often as a 1st step before proceeding with statistical modeling using deep or shallow methods. Some of these packages are used in programming exercises accompanying this book as a 1st step in preparing data for fitting into AI models.

- Librosa. Librosa is a Python package for audio analysis, designed mostly for music information retrieval. It includes a range of commonly used functions, broadly falling into 4 categories: audio & time-series operations, spectrogram calculation, time & frequency conversion, & pitch operations. Audio & time-series operations include functions for reading audio from disk `load`, resampling a signal at a desired rate `resample`, stereo to mono conversion `to_mono`, time-domain bounded auto-correlation `autocorrelate`, zero-crossing detection `zero_crossings`, & estimate of dominant frequency of STFT bins `piptrack` & pitch tracking `yin`. Spectrogram operations include short-time Fourier transform `stft`, inverse STFT `istft`, & instantaneous frequency spectrogram `ifgram`, & constant-Q transform `cqt`. It provides also mapping between different time representations: seconds, frames, or samples; & frequency representations: hertz, constantQ basis index, Fourier basis index, Mel basis index, MIDI note number, or note in scientific pitch notation. Since many of spectral manipulation operations are performed on STFT magnitude, it offers phase recovery from magnitude spectrum using Griffin-Lim algorithm `griffinlim`. Beat module provides functions to estimate global tempo & positions of beat events. Higher level structural analyses are provided using recurrence or self-similarity plots `segment.recurrence_matrix`. Decompose module factors spectrograms, or general feature arrays, into components & activations. By default, this is done with nonnegative matrix factorization NMF, but any sklearn.decomposition-type object will work. Additionally, it provides several signal transformation & signal generation functions.

- Pretty Midi provides functions for representing & handling MIDI data. At top is PrettyMIDI class, which contains global information e.g. tempo changes & MIDI resolution. It also contains a list of Instrument class instances, with each Instrument specified by a MIDI program number that correspond to names of General MIDI instruments convention. Instrument class instances contain 3 lists for Note, Pitchbend, & ControlChange. Note class is a container for MIDI notes, with velocity, pitch, & start & end time attributes. PitchBend & ControlChange classes contain attributes for bend or control change's time & value.

  Functions for performing analysis are defined in PrettyMIDI class & their a corresponding Instrument class. Some of implemented functions include extracting MIDI tempo change event `get_tempo_changes`, tempo estimate according to inner-onset intervals `estimate_tempo`, beats for every quarter note for 3/4 or 4/4 time signature & every 3rd denominator note for 6/8 or 6/16 time `get_beats`, lust of MIDI note onsets `start_times` & downbeats `get_downbeats`. A particularly useful & intuitive representation is a piano roll `get_piano_roll` that returns a matrix representation of MIDI notes, with each column spaced apart by fixed amount of seconds as provided by user. Called "sampling frequency" this should not be confused with audio sampling, but it is rather a conversion of MIDI events from tempo dependent "ticks relative to previous event, to absolute time in seconds. Another useful aggregation is in terms of pitch classes independent of octave, also known as MIDI chroma `get_chroma`. Manipulating & synthesizing MIDI file from a piano roll matrix is not provided in original package, as this requires some fine tuning of user to determine best sampling rate & other properties that are missing in order to invert matrix representation to MIDI. These inversion functions `examples/reverse_pianoroll` are provided as examples on a private github by Chris Raffael (1 of Pretty Midi authors) https://github.com/craffel/pretty-midi. See also https://github.com/craffel/pretty-midi/blob/main/Tutorial.ipynb for basic functions summarized in a Jupyter notebook.

- Music21 (21 in the title comes from designation for MIT's classes in Music, Course 21M) is a Python-based toolkit for computer-aided musicology. Core music21 object is Stream object with its subclasses (Score, Part, Measure) designed as containers for `music21` objects e.g. `Note, Chord, Clef, & TimeSignature` objects. Streams can store other Streams, permitting a wide variety of nested, ordered, & time structures. Elements within Streams are accessed with methods e.g. `getElementById()`, similar to Document Object Model (DOM) for retrieving elements in XML & HTML documents. 1 of most important `music21` classes is a Note class that has both a `.pitch` attribute, which contains a Pitch object, & a `.duration` attribute, which contains a Duration object. 1 of convenient utilities of Music21: Streams can be easily visualized in Lilypond music notation software or with other programs that support MusicXML (such Musescore, Finale or Sibelius). Other important objects are Chords that combine multiple Pitch objects on a single stem & Interval Object that calculates distance in semitones between 2 notes, with or without taking octave into account. Another important object is Durations object that represents a span of musical time for Note & all Music21Objects, e.g. TimeSignature objects.

Containers e.g. Stream & Score also have durations which are equal to position of ending of last object in Stream. Other notable tools in music21 are Chordify which is a process of making chords out of non-chords. Chordify is used for reducing a complex score with multiple parts to a succession of chords, which can be further queried for various chord properties, e.g. checking for presence of a specific chord in a polyphonic score. TinyNotation is a lightweight notation syntax, Calling a `converter.parse` function on a simple text string preceded by a "tinyNotation:" tag results in a Stream. Part subclass that is meant for designating music that is considered a single part.

- `jchord` is a Python package for working with chord progressions. It has object representations for notes, chords, & progressions designed for Western 12-tone system. 1 of notable utilities of package is its ability to convert between different naming conventions for chords, & can convert back & forth between objects & names. Also it provides simple command line tools for converting between progression represented as a string in `.txt, .xlsx` or `.midi` format to an output file in strings `.txt, .xlsx, .midi` or `.pdf` formats allows a quick & easy viewing or sonification of strings representing sequences of chords. Another useful functionality: `jchord` can parse MIDI files to sequence of block chord by approximate grouping of notes into chords allowing some imperfections.
- Mingus is a music theory & notation package for Python with MIDI file & playback support. It can also be used to create sheet music with LilyPond. Musical object include notes, intervals, chords, scales, keys & meters. Chords include natural diatonic triads, 7th chords, augmented chords, suspended chords, 6ths, 9ths, 11ths, 13ths, altered chords, & special: "5", "NC", "hendrix" & absolute chords represented from shorthand (min7, m/M7, etc.). `mingus` also supports inversions, slashed chords & polychords, & referoing to chords by their diatonic function (tonic, subtonic, etc. or I, ii, iii, IV, etc.). Substitution algorithms, diatonic scales & their modes (ionian, mixolydian, etc.), minor (natural, harmonic, & melodic) & chromatic or whole note scales are represented. Library can recognize intervals, scales, & hundreds of chords from lists of notes & their harmonic functions. To compose music with Mingus a Composition class is used to organize Tracks, to which notes are added.
- MusPy is an open source Python library for management of symbolic music datasets for DL research by providing interfaces between music data & PyTorch & TensorFlow. It supports working with common symbolic music formats e.g. MIDI, MusicXML, & ABC, & interfaces to other symbolic music libraries e.g. music21, mido, `pretty_midi`, & Pypianoroll. It also provides evaluation tools for music generation systems, including audio rendering, score, & piano-roll visualizations & objective metrics.

- Appendix D. Free Music & Audio-Editing Software.
  - Audacity is a free & open-source digital audio editor. Users can import or record audio in Audacity. Download Audacity & learn more at https://www.audacityteam.org/.
  - `LMMS` (originally named Linux MultiMedia Studio, but now available on multiple platforms) is a free, open-source digital audio workstation with an array of features to synthesize & mix sounds. Download LMMS & learn more at lmmms.io.
  - MuseScore is a free music notation software suite & growing online community for writing & sharing music. MuseScore can be used to notate & export scores as PDF or MIDI.
  - Magenta https://magenta.tensorflow.org/ is an open-source Google project built in TensorFlow to explore role of ML as a tool in creative process.

    Readers are encouraged to visit their demo page https://magenta.tensorflow.org/demos to play with existing tools & find inspiration for further ML projects. For those interested in technical details, Magenta blogposts are an excellent starting point with references to & explanations of associated research works.
- Appendix E. Datasets.
- Appendix F. Figure Attributions.

2.6 [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
    $$f(x) = \sqrt{x^2 + e^{x^2}} + \cos(x^2 + e^{x^2}). \tag{79}$$
    By application of chain rule, & noting that differentiation is linear, compute gradient:
    $$\frac{df}{dx} = 2x \left(\frac{1}{2\sqrt{x^2 + e^{x^2}}} - \sin(x^2 + e^{x^2}) \right) (1 + e^{x^2}). \tag{80}$$
    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
    $$\mathbf{y} = (f_K \circ f_{K-1} \circ \cdots \circ f_1)(\mathbf{x}) = f_K(f_{K-1}(\cdots(f_1(\mathbf{x}))\cdots)), \tag{81}$$
    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
    $$\mathbf{f}_0 = \mathbf{x}, \tag{82}$$
    $$\mathbf{f}_i = \sigma_i(\mathbf{A}_{i-1}\mathbf{f}_{i-1} + \mathbf{b}_{i-1}), \; i = 1, \ldots, K, \tag{83}$$
    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
    $$L(\boldsymbol{\theta}) = \|\mathbf{y} - \mathbf{f}_K(\boldsymbol{\theta}, \mathbf{x})\|^2 \tag{84}$$

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

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{K-1}} = \frac{\partial L}{\partial \mathbf{f}_K} \frac{\partial \mathbf{f}_K}{\partial \boldsymbol{\theta}_{K-1}}, \tag{85}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{K-2}} = \frac{\partial L}{\partial \mathbf{f}_K} \frac{\partial \mathbf{f}_K}{\partial \mathbf{f}_{K-1}} \frac{\partial \mathbf{f}_{K-1}}{\partial \boldsymbol{\theta}_{K-2}}, \tag{86}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_i} = \frac{\partial L}{\partial \mathbf{f}_K} \frac{\partial \mathbf{f}_K}{\partial \mathbf{f}_{K-1}} \cdots \frac{\partial \mathbf{f}_{i+2}}{\partial \mathbf{f}_{i+1}} \frac{\partial \mathbf{f}_{i+1}}{\partial \boldsymbol{\theta}_i}, \tag{87}$$

$$\tag{88}$$

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

$$\frac{dy}{dx} = \frac{dy}{db}\frac{db}{da}\frac{da}{dx}. \tag{89}$$

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

$$\frac{dy}{dx} = \left(\frac{dy}{db}\frac{db}{da}\right)\frac{da}{dx} = \frac{dy}{db}\left(\frac{db}{da}\frac{da}{dx}\right). \tag{90}$$

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 11. *Consider function*

$$f(x) = \sqrt{x^2 + e^{x^2}} + \cos\left(x^2 + e^{x^2}\right). \tag{91}$$

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

$$a = x^2, b = \exp a, c = a + b, d = \sqrt{c}, e = \cos c, f = d + e. \tag{92}$$

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:

$$\frac{\partial a}{\partial x} = 2x, \frac{\partial b}{\partial a} = e^a, \frac{\partial c}{\partial a} = 1 = \frac{\partial c}{\partial b}, \frac{\partial d}{\partial c} = \frac{1}{2\sqrt{c}}, \frac{\partial e}{\partial c} = -\sin c, \frac{\partial f}{\partial d} = 1 = \frac{\partial f}{\partial e}. \tag{93}$$

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

$$\frac{\partial f}{\partial c} = \frac{\partial f}{\partial d}\frac{\partial d}{\partial c} + \frac{\partial f}{\partial e}\frac{\partial e}{\partial c}, \frac{\partial f}{\partial b} = \frac{\partial f}{\partial c}\frac{\partial c}{\partial b}, \frac{\partial f}{\partial a} = \frac{\partial f}{\partial b}\frac{\partial b}{\partial a} + \frac{\partial f}{\partial c}\frac{\partial c}{\partial a}, \frac{\partial f}{\partial x} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial x}. \tag{94}$$

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

$$\frac{\partial f}{\partial c} = 1 \cdot \frac{1}{2\sqrt{c}} + 1 \cdot (-\sin c), \frac{\partial f}{\partial b} = \frac{\partial f}{\partial c} \cdot 1, \frac{\partial f}{\partial a} = \frac{\partial f}{\partial b} e^a + \frac{\partial f}{\partial c} \cdot 1, \frac{\partial f}{\partial x} = \frac{\partial f}{\partial a} \cdot 2x. \tag{95}$$

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 $x_1, \dots, x_d$: input variables to function $x_{d+1}, \dots, x_{D-1}$ be intermediate variables, & x_D: output variable. Then computation graph can be expressed as follows: (5.143)

$$\text{For } i = d+1, \dots, D: \; x_i = g_i(x_{\text{Pa}(x_i)}), \tag{96}$$

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

$$\frac{\partial f}{\partial x_D} = 1. \tag{97}$$

For other variables x_i, apply chain rule (5.145)

$$\frac{\partial f}{\partial x_i} = \sum_{x_j : x_i \in \text{Pa}(x_j)} \frac{\partial f}{\partial x_j} \frac{\partial x_j}{\partial x_i} = \sum_{x_j : x_i \in \text{Pa}(x_i)} \frac{\partial f}{\partial x_j} \frac{\partial g_j}{\partial x_i}, \tag{98}$$

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)

$$\mathbb{E}_{\mathbf{x}}[f(\mathbf{x})] = \int f(\mathbf{x}) p(\mathbf{x}) \, \mathrm{d}\mathbf{x}. \tag{99}$$

  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 $n = 1, \dots, N$. 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 $d = 1, \dots, D$. 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 $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n), \dots, (\mathbf{x}_N, y_N)\}$.

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

$$\boxed{f(\mathbf{x}) = \boldsymbol{\theta}^\top \mathbf{x} + \theta_0} \tag{100}$$

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 6. *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 7. *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 $(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)$. 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.,

$$f(\mathbf{x}_n, \boldsymbol{\theta}^\star) \approx y_n, \ \forall n = 1, \dots, N. \tag{101}$$

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

Remark 8. *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 12. *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., $\mathbf{x}_n = [1, x_n^{(1)}, x_n^{(2)}, \dots, x_n^{(d)}]^\top$. Parameter vector is correspondingly $\boldsymbol{\theta} = [\theta_0, \theta_1, \dots, \theta_D]^\top$, allowing us to write predictor as a linear function*

$$f(\mathbf{x}_n, \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \mathbf{x}_n. \tag{102}$$

This linear predictor is equivalent to affine model

$$f(\mathbf{x}_n, \boldsymbol{\theta}) = \theta_0 + \sum_{d=1}^{D} \theta_d x_n^{(d)}. \tag{103}$$

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)

$$\mathbf{R}_{\text{emp}}(f, \mathbf{X}, \mathbf{y}) = \frac{1}{N} \sum_{n=1}^{N} l(y_n, \hat{y}_n), \tag{104}$$

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 13 (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*

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^D} \frac{1}{N} \sum_{n=1}^{N} (y_n - f(\mathbf{x}_n, \boldsymbol{\theta}))^2, \tag{105}$$

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

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^D} \frac{1}{N} \sum_{n=1}^{N} (y_n - \boldsymbol{\theta}^\top \mathbf{x}_n)^2. \tag{106}$$

This equation can be equivalently expressed in matrix form

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^D} \frac{1}{N} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2. \tag{107}$$

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*

$$\mathbf{R}_{\text{true}}(f) = \mathbb{E}_{\mathbf{x},y}[l(y, f(\mathbf{x}))], \tag{108}$$

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 9. *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 14 (Regularized Least Squares)**.** *Regularization is an approach that discourages complex or extreme solutions to an optimization problem. Simplest regularization: replace least-squares problem*

$$\min_{\boldsymbol{\theta}} \frac{1}{N} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2, \tag{109}$$

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

$$\min_{\boldsymbol{\theta}} \frac{1}{N} \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2 + \lambda \|\boldsymbol{\theta}\|^2. \tag{110}$$

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

$$\mathbb{E}_{\mathcal{V}}[R(f, \mathcal{V})] \approx \frac{1}{K} \sum_{k=1}^{K} R(f^{(k)}, \mathcal{V}^{(k)}), \tag{111}$$

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

$$\mathcal{L}_{\mathbf{x}}(\boldsymbol{\theta}) = -\log p(\mathbf{x}|\boldsymbol{\theta}). \tag{112}$$

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 15. *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 $\mathbf{x}_n, y_n$,*

$$p(y_n|\mathbf{x}_n, \boldsymbol{\theta}) = \mathcal{N}(y_n|\mathbf{x}_n^\top \boldsymbol{\theta}, \sigma^2). \tag{113}$$

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)

$$p(\mathcal{Y}|\mathcal{X}, \boldsymbol{\theta}) = \prod_{n=1}^{N} p(y_n|\mathbf{x}_n, \boldsymbol{\theta}), \tag{114}$$

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)

$$\mathcal{L}(\boldsymbol{\theta}) = -\log p(\mathcal{Y}|\mathcal{X}, \boldsymbol{\theta}) = -\sum_{n=1}^{N} \log p(y_n|\mathbf{x}_n, \boldsymbol{\theta}). \tag{115}$$

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

$$\mathcal{L}(\boldsymbol{\theta}) = \cdots = \frac{1}{2\sigma^2}\sum_{n=1}^{N}(y_n - \mathbf{x}_n^\top\boldsymbol{\theta})^2 - \sum_{n=1}^{N}\log\frac{1}{\sqrt{2\pi\sigma^2}}. \tag{116}$$

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*):

$$p(\boldsymbol{\theta}|\mathbf{x}) = \frac{p(\mathbf{x}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathbf{x})}. \tag{117}$$

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

$$p(\boldsymbol{\theta}|\mathbf{x}) \asymp p(\mathbf{x}|\boldsymbol{\theta})p(\boldsymbol{\theta}). \tag{118}$$

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 17. *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 11. *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.,

$$p(y_n|\mathbf{x}_n, \boldsymbol{\theta}) = \varphi(\boldsymbol{\theta}^\top \mathbf{x}_n), \tag{119}$$

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 & 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

$$p(\boldsymbol{\theta}|\mathcal{X}) = \frac{p(\mathcal{X}|\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\mathcal{X})},\ p(\mathcal{X}) = \int p(\mathcal{X}|\boldsymbol{\theta})p(\boldsymbol{\theta})\,\mathrm{d}\boldsymbol{\theta}, \tag{120}$$

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)

$$p(\mathbf{x}) = \int p(\mathbf{x}|\boldsymbol{\theta})p(\boldsymbol{\theta})\,\mathrm{d}\boldsymbol{\theta} = \mathbb{E}_{\boldsymbol{\theta}}[p(\mathbf{x}|\boldsymbol{\theta})], \tag{121}$$

& 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 & 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 12. *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 & 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)

$$p(\mathbf{x}|\boldsymbol{\theta}) = \int p(\mathbf{x}|\mathbf{z}, \boldsymbol{\theta}) p(\mathbf{z}) \, \mathrm{d}\mathbf{z}, \tag{122}$$

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)

$$p(\boldsymbol{\theta}|\mathcal{X}) = \frac{p(\mathcal{X}|\boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(\mathcal{X})} \tag{123}$$

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

$$p(\mathbf{z}|\mathcal{X}) = \frac{p(\mathcal{X}|\mathbf{z}) p(\mathbf{z})}{p(\mathcal{X})}, \; p(\mathcal{X}|\mathbf{z}) = \int p(\mathcal{X}|\mathbf{z}, \boldsymbol{\theta}) p(\boldsymbol{\theta}) \, \mathrm{d}\boldsymbol{\theta}, \tag{124}$$

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.,

$$p(\mathbf{z}|\mathcal{X}, \boldsymbol{\theta}) = \frac{p(\mathcal{X}|\mathbf{z}, \boldsymbol{\theta}) p(\mathbf{z})}{p(\mathcal{X}|\boldsymbol{\theta})}, \tag{125}$$

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 13. *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) & 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 14. *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 18. *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 19. *Looking at graphical model in* Fig. 8.9(b)*, exploit 2 properties: (i) Joint distribution $p(x_1, \dots, x_5)$ 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 $p(\mathbf{x}) = p(x_1, \dots, x_K)$ is given as

$$p(\mathbf{x}) = \prod_{k=1}^{K} p(x_k|\mathsf{Pa}_k), \tag{126}$$

    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

$$p(x|\mu) = \mathrm{Ber}(\mu). \tag{127}$$

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

$$p(x_1, \ldots, x_N|\mu) = \prod_{n=1}^{N} p(x_n|\mu). \tag{128}$$

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 20 (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 & 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 & Föglein, 1984; Sucar & 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 & 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 & Rubin, 2015; Peters et al., 2017; Rosenbaum, 2017).

○ 8.6. Model Selection. In ML, often need to make high-level modeling decisions that critically influence performance of model. Choices we make (e.g., functional form of likelihood) influence number & type of free parameters in model & thereby also flexibility & expressivity of model. More complex models are more flexible in sense that they can be used to describe more datasets. E.g., a polynomial of degree 1 (a line $y = a_0 + a_1 x$) can only be used to describe linear relations between inputs x & observations y. A polynomial of degree 2 can additional describe quadratic relationships between inputs x & observations y. A polynomial of degree 2 can additionally describe quadratic relationships between inputs & observations.

One would now think: very flexible models are generally preferable to simple models because they are more expressive. A general problem: at training time can only use training set to evaluate performance of model & learn its parameters. However, performance on training set is not really what we are interested in. In Sect. 8.3, have seen: maximum likelihood estimation can lead to overfitting, especially when training dataset is small. Ideally, our model (also) works well on test set (which is not available at training time). Therefore, need some mechanisms for assessing how a model *generalizes* to unseen test data. *Model selection* is concerned with exactly this problem.

A polynomial $y = a_0 + a_1 x + a_2 x^2$ can also describe linear functions by setting $a_2 = 0$, i.e., strictly more expressive than a 1st-order polynomial.

- 8.6.1. Nested Cross-Validation. Have already seen an approach (cross-validation in Sect. 8.2.4) that can be used for model selection. Recall: cross-validation provides an estimate of generalization error by repeatedly splitting dataset into training & validation sets. Can apply this idea 1 more time, i.e., for each split, can perform another round of cross-validation. This is sometimes referred to as *nested cross-validation*; see Fig. 8.13: Nested cross-validation. Perform 2 levels of K-fold cross-validation. Inner level is used to estimate performance of a particular choice of model or hyperparameter on a internal validation set. Outer level is used to estimate generalization performance for best choice of model chosen by inner loop. Can test different model & hyperparameter choices in inner loop. To distinguish 2 levels, set used to estimate generalization performance is often called *test case* & set used for choosing best model is called *validation set*. Inner loop estimates expected value of generalization error for a given model (8.39), by approximating it using empirical error on validation set, i.e.,

$$\mathbb{E}_{\mathcal{V}}[\mathbf{R}(\mathbf{V}|M)] \approx \frac{1}{K}\sum_{k=1}^{K}\mathbf{R}(\mathcal{V}^{(k)}|M), \tag{129}$$

where $\mathbf{R}(\mathcal{V}|M)$ is empirical risk (e.g., root mean square error) on validation set $\mathcal{V}$ for model M. Repeat this procedure $\forall$ models & choose model that performs best. Note: cross-validation not only gives us expected generalization error, but can also obtain high-order statistics, e.g., standard error, an estimate of how uncertain mean estimate is. Once model is chosen, can evaluate final performance on test set.

- 8.6.2. Bayesian Model Selection.
- 8.6.3. Bayes Factors for Model Comparison.
- 8.6.4. Further Reading. Mentioned at start of sect: there are high-level modeling choices that influence performance of model. Examples include:
  - Degree of a polynomial in a regression setting
  - Number of components in a mixture model
  - Network architecture of a (deep) neural network
  - Type of kernel in a support vector machine
  - Dimensionality of latent space in PCA
  - Learning rate (schedule) in an optimization algorithm

  In parametric models, number of parameters is often related to complexity of model class.

  Rasmussen & Ghahramani (2001) showed: automatic Occam's razor does not necessarily penalize number of parameters in a model, but it is active in terms of complexity of functions. Also showed: automatic Occam's razor also holds for Bayesian nonparametric models with many parameters, e.g., Gaussian processes.

  If focus on maximum likelihood estimate, there exist a number of heuristics for model selection that discourage overfitting. They are called *information criteria*, & choose model with largest value. *Akaike information criterion* (ALC) (Akaike, 1974) $\log p(\mathbf{x}|\boldsymbol{\theta}) - M$ corrects for bias of maximum likelihood estimator by addition of a penalty term to compensate for overfitting of more complex models with lots of parameters. Here, M: number of model parameters. AIC estimates relative information lost by a given model.

  Bayesian information criterion (BIC) (Schwarz, 1978)

$$\log p(\mathbf{x}) = \log \int p(\mathbf{x}|\boldsymbol{\theta})p(\boldsymbol{\theta})\,\mathrm{d}\boldsymbol{\theta} \approx \log p(\mathbf{x}|\boldsymbol{\theta}) - \frac{1}{2}M\log N \tag{130}$$

  can be used for exponential family distributions. Here, N: number of data points & M: number of parameters. BIC penalizes model complexity more heavily than AIC.

- 9. Linear Regression. In following, will apply mathematical concepts from Chaps. 2, 5, 6, & 7 to solve linear regression (curve fitting) problems. In *regression*, aim: find a function f that maps input $\mathbf{x} \in \mathbb{R}^D$ to corresponding function values $f(\mathbf{x}) \in \mathbb{R}$. Assume: given a set of training inputs $\mathbf{x}_n$ & corresponding noisy observations $y_n = f(\mathbf{x}_n) + \varepsilon$, where ε is an i.i.d. random variable that describes measurement/observation noise & potentially unmodeled processes (which will not consider further in this chap). Throughout this chap, assume zero-mean Gaussian noise. Our task: find a function that not only models training data, but generalizes well to predicting function values at input locations that are not part of training data (see Chap. 8). An illustration of such a regression problem is given in Fig. 9.1: (a) Dataset: Regression problem: observed noisy function values from which we wish to infer underlying function that generated data. (b) Possible solution to regression problem: Regression solution: possible function that could have generated data (blue) with indication of measurement noise of function value at corresponding

inputs (orange distributions). A typical regression setting is given in Fig. 9.1(a): For some input values x_n, observe (noisy) function values $y_n = f(x_n) + \epsilon$. Task: infer function f that generated data & generalizes well to function values at new input locations. A possible solution is given in Fig. 9.1(b), where we also show 3 distributions centered at function values $f(x)$ that represent noise in data.

Regression is a fundamental problem in ML, & regression problems appear in a diverse range of research areas & applications, including time-series analysis (e.g., system identification), control & robotics (e.g, reinforcement learning, forward/inverse model learning), optimization (e.g., line searches, global optimization), & DL applications (e.g., computer games, speech-to-text translation, image recognition, automatic video annotation). Regression is also a key ingredient of classification algorithms. Finding a regression function requires solving a variety of problems, including following:

- **Choice of model (type) & parametrization** of regression function. Given a dataset, what function classes (e.g., polynomials) are good candidates for modeling data, & what particular parametrization (e.g., degree of polynomial) should we choose? Model selection, as discussed in Sect. 8.6, allows us to compare various models to find simplest model that explains training data reasonably well.
- **Finding good parameters.** Having chosen a model of regression function, how to we find good model parameters? Here, need to look at different loss/objective functions (they determine what a "good" fit is) & optimization algorithms that allow us to minimize this loss.
- **Overfitting & model selection.** Overfitting is a problem when regression function fits training data "too well" but does not generalize to unseen test data. Overfitting typically occurs if underlying model (or its parameterization) is overly flexible & expressive; see Sect. 8.6. Look at underlying reasons & discuss ways to mitigate effect of overfitting in context of linear regression.
- **Relationship between loss functions & parameter priors.** Loss functions (optimization objectives) are often motivated & included by probabilistic models. Look at connection between loss functions & underlying prior assumptions that induce these losses.
- **Uncertainty modeling.** In any practical setting, have access to only a finite, potentially large, amount of (training) data for selecting model class & corresponding parameters. Given that this finite amount of training data does not cover all possible scenarios, may want to describe remaining parameter uncertainty to obtain a measure of confidence of model's prediction at test time; smaller training set, more important uncertainty modeling. Consistent modeling of uncertainty equips model predictions with confidence bounds.

In following, will be using mathematical tools from Chaps. 3, 5, 6, & to solve linear regression problems. Discuss maximum likelihood & maximum a posteriori (MAP) estimation to find optimal model parameters. Using these parameter estimates, have a brief look at generalization errors & overfitting. Toward end of this chap, will discuss Bayesian linear regression, which allows us to reason about model parameters at a high level, thereby removing some of problems encountered in maximum likelihood & MAP estimation.

- 9.1. Problem Formulation. Because of presence of observation noise, adopt a probabilistic approach & explicitly model noise using a likelihood function. More specifically, throughout this chap, consider a regression problem with likelihood function (9.1)

$$p(y|\mathbf{x}) = \mathcal{N}(y|f(\mathbf{x}), \sigma^2). \tag{131}$$

Here $\mathbf{x} \in \mathbb{R}^D$: inputs & $y \in \mathbb{R}$: noisy function values (targets). With (9.1), functional relationship between $\mathbf{x}$ & y is given as

$$y = f(\mathbf{x}) + \epsilon, \tag{132}$$

where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ is independent, identically distributed (i.i.d.) Gaussian measurement noise with mean 0 & variance σ^2. Objective: find a function that is close (similar) to unknown function f that generated data & that generalizes well.

In this chap, focus on parametric models, i.e., choose a parametrized function & find parameters $\boldsymbol{\theta}$ that "work well" for modeling data. For time being, assume: noise variance σ^2 is known & focus on learning model parameters $\boldsymbol{\theta}$. In linear regression, consider special case: parameters $\boldsymbol{\theta}$ appear linearly in our model. An example of linear regression is given by (9.3)–(9.4)

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|\mathbf{x}^\top \boldsymbol{\theta}, \sigma^2) \Leftrightarrow y = \mathbf{x}^\top \boldsymbol{\theta} + \epsilon,\ \epsilon \sim \mathcal{N}(0, \sigma^2), \tag{133}$$

where $\boldsymbol{\theta} \in \mathbb{R}^D$: parameters we seek. Class of functions described by (9.4) are straight lines that pass through origin. In (9.4), chose a parametrization $f(\mathbf{x}) = \mathbf{x}^\top \boldsymbol{\theta}$.

Likelihood in (9.3) is probability density function of y evaluated at $\mathbf{x}^\top \boldsymbol{\theta}$. Note: only source of uncertainty originates from observation noise (as $\mathbf{x}$ & $\boldsymbol{\theta}$ are assumed known in (9.3)). Without observation noise, relationship between $\mathbf{x}$ & y would be deterministic & (9.3) would be a Dirac delta.

> A Dirac delta (delta function) is zero everywhere except at a single point, & its integral is 1. It can be considered a Gaussian in limit of $\sigma^2 \to 0$.

Example 21. *For $x, \theta \in \mathbb{R}$ linear regression model in (9.4) describes straight lines (linear functions), & parameter θ: slope of line.* Fig. 9.2(a) Linear regression example. (a) Example functions that fall into this category: Example functions (straight lines) that can be described using linear model in (9.4). (b) training set. (c) maximum likelihood estimate. *shows some example functions for different values of θ.*

Linear regression refers to models that are linear in parameters.

Linear regression model in (9.3)–(9.4) is not only linear in parameters, but also linear in inputs x. Fig. 9.2(a) shows examples of such functions. $y = \boldsymbol{\phi}^\top(\mathbf{x})\boldsymbol{\theta}$ for nonlinear transformations ϕ is also a linear regression model because "linear regression" refers to models that are "linear in parameters", i.e., models that describe a function by a linear combination of input features. Here, a "feature" is a representation $\boldsymbol{\phi}(\mathbf{x})$ of inputs $\mathbf{x}$.

In following, discuss in more detail how to find good parameters $\boldsymbol{\theta}$ & how to evaluate whether a parameter set "works well". For time being, assume: noise variation σ^2 is known.

◦ 9.2. Parameter Estimation. Consider linear regression setting (9.4) & assume given a *training set* $\mathcal{D} \coloneqq \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N\}$ consisting of N inputs $\mathbf{x}_n \in \mathbb{R}^D$ & corresponding observations/targets $y_n \in \mathbb{R}$, $n = 1, \ldots, N$. Corresponding graphical model is given in Fig. 9.3: Probabilistic graphical model for linear regression. Observed random variables are shaded, deterministic/known values are without circles.. Note: y_i & y_j are conditionally independent given their respective inputs $\mathbf{x}_i, \mathbf{x}_j$ so that likelihood factorizes according to

$$p(\mathcal{Y}|\mathcal{X}, \boldsymbol{\theta}) = p(y_1, \ldots, y_N|\mathbf{x}_1, \ldots, \mathbf{x}_N, \boldsymbol{\theta}) = \prod_{i=1}^{N} p(y_n|\mathbf{x}_n, \boldsymbol{\theta}) = \prod_{n=1}^{N} \mathcal{N}(y_n|\mathbf{x}_n^\top\boldsymbol{\theta}, \sigma^2), \tag{134}$$

where defined $\mathcal{X} \coloneqq \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ & $\mathcal{Y} \coloneqq \{y_1, \ldots, y_N\}$ as sets of training inputs & corresponding targets, resp. Likelihood & factors $p(y_n|\mathbf{x}_n, \boldsymbol{\theta})$ are Gaussian due to noise distribution; see (9.3).

In following, discuss how to find optimal parameters $\boldsymbol{\theta}^* \in \mathbb{R}^D$ for linear regression model (9.4). Once parameters $\boldsymbol{\theta}^*$ are found, can predict function values by using this parameter estimate in (9.4) so that at an arbitrary test input $\mathbf{x}_*$ distribution of corresponding target y_* is

$$p(y_*, \mathbf{x}_*, \boldsymbol{\theta}^*) = \mathcal{N}(y_*|\mathbf{x}_*^\top\boldsymbol{\theta}^*, \sigma^2). \tag{135}$$

In following, have a look at parameter estimation by maximizing likelihood, a topic that already covered to some degree in Sect. 8.3.

∗ 9.2.1. Maximum Likelihood Estimation. A widely used approach to finding desired parameters $\boldsymbol{\theta}_{\mathrm{ML}}$ is *maximum likelihood estimation*, where find parameters $\boldsymbol{\theta}_{\mathrm{ML}}$ that maximize likelihood (9.5b). Intuitively, maximizing likelihood means maximizing predictive distribution of training data given model parameters. Obtain maximum likelihood parameters as

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} p(\mathcal{Y}|\mathcal{X}, \boldsymbol{\theta}). \tag{136}$$

Remark 15. *Likelihood $p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta})$ is not a probability distribution in $\boldsymbol{\theta}$: It is simply a function of parameters $\boldsymbol{\theta}$ but does not integrate to 1 (i.e., it is unnormalized), & may not even be integrable w.r.t. $\boldsymbol{\theta}$. However, likelihood in (9.7) is a normalized probability distribution in $\mathbf{y}$.*

Maximizing likelihood means maximizing predictive distribution of (training) data given parameters.

Likelihood is not a probability distribution in parameters.

To find desired parameters $\boldsymbol{\theta}_{\mathrm{ML}}$ that maximize likelihood, typically perform gradient ascent (or gradient descent on negative likelihood). In case of linear regression considered here, however, a closed-form solution exists, which makes iterative gradient descent unnecessary. In practice, instead of maximizing likelihood directly, apply log-transformation to likelihood function & minimize negative log-likelihood.

Since logarithm is a (strictly) monotonically increasing function, optimum of a function f is identical to optimum of $\log f$.

Remark 16 (Log-Transformation)**.** *Since likelihood (9.5b) is a product of N Gaussian distributions, log-transformation is useful since (a) it does not suffer from numerical underflow, & (b) differentiation rules will turn out simpler. More specifically, numerical underflow will be a problem when multiply N probabilities, where N: number of data points, since cannot represent very small numbers, e.g. 10^{-256}. Furthermore, log-transform will turn product into a sum of log-probabilities s.t. corresponding gradient is a sum of individual gradients, instead of a repeated application of product rule to compute gradient of a product of N terms.*

To find optimal parameters $\boldsymbol{\theta}_{\mathrm{ML}}$ of our linear regression problem, minimize negative log-likelihood (9.8)

$$-\log p(\mathcal{Y}|\mathcal{X}, \boldsymbol{\theta}) = -\log \prod_{n=1}^{N} p(y_n|\mathbf{x}_n, \boldsymbol{\theta}) = -\sum_{n=1}^{N} \log p(y_n|\mathbf{x}_n, \boldsymbol{\theta}), \tag{137}$$

where exploited: likelihood (9.5b) factorizes over number of data points due to our independence assumption on training set.

In linear regression model (9.4), likelihood is Gaussian (due to Gaussian additive noise term), s.t. arrive at

$$\log p(y_n|\mathbf{x}_n, \boldsymbol{\theta}) = -\frac{1}{2\sigma^2}(y_n - \mathbf{x}_n^\top\boldsymbol{\theta})^2 + \mathrm{const}, \tag{138}$$

where constant includes all terms independent of $\boldsymbol{\theta}$. Using (9.9) in negative log-likelihood (9.8), obtain (ignoring constant terms)

$$\mathcal{L}(\boldsymbol{\theta}) \coloneqq \frac{1}{2\sigma^2}\sum_{n=1}^{N}(y_n - \mathbf{x}_n^\top\boldsymbol{\theta})^2 = \frac{1}{2\sigma^2}(\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^\top(\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) = \frac{1}{2\sigma^2}\|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|^2, \tag{139}$$

where define *design matrix* $\mathbf{X} \coloneqq [\mathbf{x}_1, \dots, \mathbf{x}_N]^\top \in \mathbb{R}^{N \times D}$ as collection of training inputs & $\mathbf{y} \coloneqq [y_1, \dots, y_N]^\top \in \mathbb{R}^N$ as a vector that collects all training targets. Note: nth row in design matrix $\mathbf{X}$ corresponds to training input $\mathbf{x}_n$. In (9.10b), used fact: sum of squared errors between observations y_n & corresponding model prediction $\mathbf{x}_n^\top \boldsymbol{\theta}$ equals squared distance between $\mathbf{y}$ & $\mathbf{X}\boldsymbol{\theta}$.

Negative log-likelihood function is also called *error function*.

Squared error is often used as a measure of distance. Recall from Sect. 3.1: $\|\mathbf{x}\|^2 = \mathbf{x}^\top \mathbf{x}$ if choose dot product as inner product.

With (9.10b), have now a concrete form of negative log-likelihood function we need to optimize. Immediately see: (9.10b) is quadratic in $\boldsymbol{\theta}$. I.e., can find a unique global solution $\boldsymbol{\theta}_{\mathrm{ML}}$ for minimizing negative log-likelihood $\mathcal{L}$. Can find global optimum by computing gradient of $\mathcal{L}$, setting it to $\mathbf{0}$ & solving for $\boldsymbol{\theta}$.
Using results from Chap. 5, compute gradient of $\mathcal{L}$ w.r.t. parameters as

$$\frac{d\mathcal{L}}{d\boldsymbol{\theta}} = \dots = \frac{1}{\sigma^2}(-\mathbf{y}^\top \mathbf{X} + \boldsymbol{\theta}^\top \mathbf{X}^\top \mathbf{X}) \in \mathbb{R}^{1 \times D}. \tag{140}$$

Maximum likelihood estimator $\boldsymbol{\theta}_{\mathrm{ML}}$ solves $\frac{d\mathcal{L}}{d\boldsymbol{\theta}} = \mathbf{0}^\top$ (necessary optimality condition) & obtain

$$\frac{d\mathcal{L}}{d\boldsymbol{\theta}} = \mathbf{0}^\top \Leftrightarrow \dots \Leftrightarrow \boldsymbol{\theta}_{\mathrm{ML}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}. \tag{141}$$

Could right-multiply 1st equation by $(\mathbf{X}^\top \mathbf{X})^{-1}$ because $\mathbf{X}^\top \mathbf{X}$ is positive definite if $\mathrm{rk}(\mathbf{X}) = D$, where $\mathrm{rk}(\mathbf{X})$ denotes rank of $\mathbf{X}$.

Ignoring possibility of duplicate data points, $\mathrm{rk}(\mathbf{X}) = D$ if $N \geq D$, i.e., do not have more parameters than data points.

Remark 17. *Setting gradient to $\mathbf{0}^\top$ is a necessary & sufficient condition, & obtain a global minimum since Hessian $\nabla^2_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) = \mathbf{X}^\top \mathbf{X} \in \mathbb{R}^{D \times D}$ is positive definite.*

Remark 18. *Maximum likelihood solution in (9.12c) requires us to solve a system of linear equations of form $\mathbf{A}\boldsymbol{\theta} = \mathbf{b}$ with $\mathbf{A} = (\mathbf{X}^\top \mathbf{X})$ & $\mathbf{b} = \mathbf{X}^\top \mathbf{y}$.*

Example 22 (Fitting Lines)**.** *Look at Fig. 9.2, where we aim to fit a straight line $f(x) = \theta x$, where θ is an unknown slope, to a dataset using maximum likelihood estimation. Examples of functions in this model class (straight lines) are shown in* Fig. 9.2(a)*. For dataset shown in* Fig. 9.2(b)*, find maximum likelihood estimate of slope parameter θ using (9.12c) & obtain maximum likelihood linear function in* Fig. 9.2(c)*.*

· Maximum Likelihood Estimation with Features.

Linear regression refers to "linear-in-the-parameters" regression models, but inputs can undergo any nonlinear transformation.

So far, considered linear regression setting described in (9.4), which allowed us to fit straight lines to data using maximum likelihood estimation. However, straight lines are not sufficiently expressive when it comes to fitting more interesting data. Fortunately, linear regression offers us a way to fit nonlinear functions within linear regression framework: Since "linear regression" only refers to "linear in parameters", can perform an arbitrary nonlinear transformation $\boldsymbol{\phi}(\mathbf{x})$ of inputs $\mathbf{x}$ & then linearly combine components of this transformation. Corresponding linear regression model is

$$p(y|\mathbf{x}, \boldsymbol{\theta}) = \mathcal{N}(y|\boldsymbol{\phi}^\top(\mathbf{x})\boldsymbol{\theta}, \sigma^2) \Leftrightarrow y = \boldsymbol{\phi}^\top(\mathbf{x})\boldsymbol{\theta} + \epsilon = \sum_{k=0}^{K-1} \theta_k \phi_k(\mathbf{x}) + \epsilon, \tag{142}$$

where $\phi : \mathbb{R}^D \to \mathbb{R}^K$: a (nonlinear) transformation of inputs $\mathbf{x}$ & $\phi_k : \mathbb{R}^D \to \mathbb{R}$: kth component of *feature vector* $\boldsymbol{\phi}$. Note: model parameters $\boldsymbol{\theta}$ still appear only linearly.

Example 23 (Polynomial Regression)**.** *Concerned with a regression problem $y = \boldsymbol{\theta}^\top x\boldsymbol{\theta} + \epsilon$, where $x \in \mathbb{R}, \boldsymbol{\theta} \in \mathbb{R}^K$. A transformation often used in this context is $\boldsymbol{\phi}(x) = [\phi_0(x), \phi_1(x), \dots, \phi_{K-1}(x)]^\top = [1, x, x^2, \dots, x^{K-1}] \in \mathbb{R}^K$ (9.14). I.e., "lift" original 1D input space into a K-dimensional feature space consisting of all monomials x^k for $k = 0, \dots, K-1$. With these features, can model polynomials of degree $\leq K-1$ within framework of linear regression: A polynomial of degree $K-1$ is*

$$f(x) = \sum_{k=0}^{K-1} \theta_k x^k = \boldsymbol{\theta}^\top(x)\boldsymbol{\theta}, \tag{143}$$

where $\boldsymbol{\theta}$ is defined in (9.14) & $\boldsymbol{\theta} = [\theta_0, \dots, \theta_{K-1}]^\top \in \mathbb{R}^K$ contains (linear) parameters θ_k.

Now have a look at maximum likelihood estimation of parameters $\boldsymbol{\theta}$ in linear regression model (9.13). Consider training inputs $\mathbf{x}_n \in \mathbb{R}^D$ & targets $y_n \in \mathbb{R}, n = 1, \dots, N$, & define *feature matrix (design matrix)* is $\boldsymbol{\Phi} \coloneqq [\boldsymbol{\phi}^\top(\mathbf{x}_1), \dots, \boldsymbol{\phi}^\top(\mathbf{x}_N)]^\top = \dots \in \mathbb{R}^{N \times K}$, where $\Phi_{ij} = \phi_j(\mathbf{x}_i)$ & $\phi_j : \mathbb{R}^D \to \mathbb{R}$.

Example 24 (Feature Matrix for 2nd-order Polynomials)**.** *For a 2nd-order polynomial & N training points $x_n \in \mathbb{R}$, $n = 1, \dots, N$, feature matrix is*

$$\boldsymbol{\Phi} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 \end{bmatrix}. \tag{144}$$

With feature matrix $\mathbf{\Phi}$ defined in (9.16), negative log-likelihood for linear regression model (9.13) can be written as (9.18)

$$-\log p(\mathcal{Y}|\mathcal{X},\boldsymbol{\theta}) = \frac{1}{2\sigma^2}(\mathbf{y}-\mathbf{\Phi}\boldsymbol{\theta})^\top(\mathbf{y}-\mathbf{\Phi}\boldsymbol{\theta}) + \text{const.} \tag{145}$$

Comparing (9.18) ***

- 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.

2.7 Fatemeh Jamshidi, Gary Pike, Amit Das, Richard Chapman. Machine Learning Techniques in Automatic Music Transcription: A Systematic Survey. 2024

[2 citations]

- Abstract. In domain of Music Information Retrieval (MIR), Automatic Music Transcription (AMT) emerges as a central challenge, aiming to convert audio signals into symbolic notations like musical notes or sheet music. This systematic review accentuates (nhấn mạnh) pivotal role (vai trò then chốt) of AMT in music signal analysis, emphasizing its importance due to intricate & overlapping spectral structure of musical harmonies. Through a thorough examination of existing ML techniques utilized in AMT, explore progress & constraints of current models & methodologies. Despite notable advancements, AMT systems have yet to match accuracy of human experts, largely due to complexities of musical harmonies & need for nuanced interpretation (diễn giải sắc thái). This review critically evaluates both fully automatic & semi-automatic AMT systems, emphasizing importance of minimal user intervention & examining various methodologies proposed to date. By addressing limitations of prior techniques & suggesting avenues for improvement, objective: steer (lái) future research towards fully automated AMT systems capable of accurately & efficiently translating intricate audio signals into precise symbolic representations. This study not only synthesizes latest advancements but also lays out a road-map for overcoming existing challenges in AMT, providing valuable insights for researchers aiming to narrow gap between current systems & human-level transcription accuracy.

- 1. Introduction. Automatic Music Transcription (AMT) is process of converting an acoustic signal (tín hiệu âm thanh) into its equivalent notation, pitch, duration, onset & offset time, musical score or sheet, or any other musical representation [1–5]. Applications of AMT include music education (e.g., through systems for automatic instrument tutoring), music creation (e.g., dictating improvised musical ideas & automatic music accompaniment), music production (e.g., music content visualization & intelligent content-based editing), music search (e.g., indexing & recommendation of music by melody, bass, rhythm, or chord progression), & musicology (e.g., analyzing jazz improvisations & other annotated music) [4].

  AMT problem can be divided into several sub-tasks [2, 6]: Multi-pitch detection, Note onset/offset detection, Loudness estimation & quantization (lượng tử hóa), Instrument recognition, Extraction of rhythmic information, Time quantization, Extraction of velocity & dynamic.

  Fig. 1: Automatic music transcription system [7] illustrates data representations in an AMT system. AMT system takes an audio waveform as input, computes a time-frequency representation of audio, outputs a representation of pitches over time in a spectrogram, & generates a typeset music score [3]. Previous studies have tackled Automatic Music Transcription (AMT) using 2 main approaches: Nonnegative Matrix Factorization (NMF) [8], & Neural Networks [2, 9]. NN techniques typically involve processing spectrograms with various neural network architectures, e.g. long short-term memory layers or Convolutional Neural Networks (CNNs). Many AMT studie rely on NNs, particularly in context of polyphonic piano transcription. 1 notable model is Google Magenta Onsets & Frames (OaF) [10], which comprises 2 components: onsets head & frames head. Recent advancements introduce alternative methods aimed at enhancing transcription accuracy by reconstructing input spectrogram [11].

  Alternative approaches to Automatic Music Transcription (AMT) involve processing mixed signals using multitask deep learning techniques. This entails separating signal sources before conducting AMT, which necessitates incorporating a source separation component into network architecture. E.g., Cerberus [12] employs 3 components – source separation, deep clustering, & transcription heads – within its multitask DL model for AMT. Furthermore, researchers are exploring application of AMT in multi-instrument music transcript, wherein instrument identification is performed as a sub-task using self-attention mechanisms [13].

- 2. Frame-Level Transcription. Frame-level transcription, also referred to as Multi-Pitch Estimation (MPE), estimates number & pitch of notes concurrently present in each time frame, typically with a latency of approximately 10 milliseconds [4]. Each frame is usually processed independently, although contextual information may be considered through filtering frame-level pitch estimates in a post-processing stage. However, this method does not explicitly model musical notes or encompass high-level musical structures. Numerous existing Automatic Music Transcription (AMT) techniques operate at this level, including traditional signal processing methods [14, 15], probabilistic modeling [16], Bayesian approaches [17], Nonnegative Matrix Factorization NMF [18–21], & neural networks [22, 23]. Each of these methods presents distinct advantages & limitations, & research has not yet converged on a singular approach. E.g., traditional signal processing methods are characterized by their simplicity & speed, & they demonstrate good generalization across different instruments. Conversely, deep neural network methods generally achieve higher accuracy on specific instruments [9].

  In single-instrument AMT, process commences (bắt đầu) with frequency estimation, encompassing 2 subcategories: Fundamental Frequency Estimation (f0 estimations) & Multi-f0 estimation. Fundamental frequency estimation involves identifying fundamental frequency of notes in each time frame. Various methodologies have been proposed, including template matching, probabilistic algorithms, & salience function techniques (kỹ thuật hàm nổi bật). Some approaches are available for estimating f0 in monophonic signals, including SWIPE [24], which matches spectrum of a waveform with a template, & a probabilistic variant of YIN [25] that decodes a pitch value sequence by using a Hidden Markov Model (HMM). Latest & best-performing methods, e.g. CREPE [26], employ DL techniques, converting input signals into spectrograms & processing them through CNNs.

  On other hand, Multi-f0 estimation, tackles challenge of discerning multiple fundamental frequencies present in a polyphonic signal, where several notes coexist within each time frame. Earlier studies in multi-f0 estimation either model spectral peaks or utilize CNNs with constant Q-Transform as inputs to learn salience representations for estimating fundamental frequencies [27].

- 3. Note-Level Transcription in Polyphonic Music. Piano stands out as most thoroughly examined instrument for polyphonic music transcription. This prominence owes much to accessibility of comprehensive datasets & its percussive onset characteristics (đặc điểm khởi đầu của tiếng gõ), akin to other percussion instruments. Consequently, primary focus of study for nearly all contemporary [28] end-to-end models, leveraging Convolutional Neural Networks (CNNs) & Long Short-Term Memory networks (LSTMs), lies in multi-f0 estimation for piano transcription.

  End-to-end models, employing deep neural networks, are pivotal in facilitating Automatic Music Transcription (AMT). Nonetheless, certain models necessitate discrete sub-tasks, like utilizing waveform domain signals as input for deep neural networks or mandating a pre-processing phase to convert wave-forms into time-frequency representations.

  1 noteworthy model developed to tackle polyphonic piano & drums transcription is Onset & Frames (OaF) [10]. This model excels in detecting note onsets & predicting pitches, adaptable for transcribing piano pieces using MAESTRO dataset [29] or drums employing Expanded MIDI Groove (E-GMD) [30] dataset, contingent upon dataset used for training.

  Spectrograms emerge as preferred input for these models, showcasing remarkable AMT outcomes for piano datasets. This preference is underscored by architecture's ability to enhance transcription quality through accurate onset predictions tailored

for piano compositions. However, performance of these models remains untested with instruments characterized by disparate timbres & onset envelopes.

- 4. Stream-Level Transcription. Multipitch streaming (MPS), alternatively referred to as stream-level transcription, is a technique that groups estimated pitches or notes into streams, where each stream typically representing an individual instrument or musical voice. This technique is closely related to instrument source separation. Distinguished from note-level transcription, MPS entails a higher degree of complexity due to elongated pitch contours of each stream, encompassing multiple discontinuities arising from silent intervals, non-pitched sounds, & abrupt frequency shifts.

  Timbre (Âm sắc) emerges as a crucial factor in MPS that is not explored in MPE & note tracking. This is because notes within same stream tend to share similar timbral characteristics, distinguishing them from those in separate streams. Despite significance of this approach, existing literature remains somewhat sparse, with only a handful of examples e.g. [31,20,32].

  As transcription task becomes more complex from frame level to note level to stream level, it necessitates incorporation of additional musical structures & cues. Despite increasing complexity, transcription outputs at these 3 levels are all parametric transcriptions, which provide a parametric description of audio content. A prime instance of such transcription is MIDI piano roll, serving as an abstraction of musical audio that has not yet attained level of detail found in traditional music notation. Although it provides pitch & timing information, it still lacks fundamental concepts e.g. beats, bars, meter, key, & stream delineation (phân định dòng suối).

- 5. Notation-Level Transcription. Notation-level transcription aims to convert audio file into a human-readable musical score, necessitating a comprehensive grasp of musical structures e.g. harmonic, rhythmic, & stream structures. Prior research in this domain has predominantly focused on timing quantization & employed DL techniques. Some models have endeavored to address this challenge through end-to-end models that take either a signal or time-frequency representation as input & output score of musical piece. Conversely, other methodologies concentrate on transcribing from MIDI files to musical scores.

  2 primary methodologies exist for notation-level transcription: 1st perform f0 detection, while 2nd encompasses multi-f0 detection coupled with note-tracking algorithms. CREPE [26] pre-trained weights are used for f0 estimation, & model output is then used for note tracking. Proposed methods are tested on multiple musical instruments with different timbres. Results are compared with state-of-art Onset & Frames (OaF) [10] model, designed for polyphonic piano transcription, to evaluate how timbre affects pitch estimation & note tracking. Following meticulous evaluation, a multi-f0 model is developed to transcribe polyphonic signals emanating from single instruments across a diverse range of timbral variations. – Sau quá trình đánh giá tỉ mỉ, một mô hình đa f0 được phát triển để phiên âm các tín hiệu đa âm phát ra từ các nhạc cụ đơn lẻ trên nhiều biến thể âm sắc khác nhau.

- 6. Music Transcription Dataset for Training ML Models. MAESTRO ("MIDI & Audio Edited for Synchronous Tracks & Organization") dataset [29], contains over a week of paired audio & MIDI recordings from 9 years of International Piano-e-Competition events. Dataset includes annotation of isolated notes, duration of notes, chords, & complete piano pieces. MIDI data includes key strikes, velocities & sustain pedal positions. Achieving an alignment accuracy of approximately 3 ms accuracy, audio & MIDI files are segmented into individual musical pieces, each with composer, title, & year of performance. Use set of synthesized pieces of a single train/validation/test split designed to satisfy following criteria:

  - No composition should appear in > 1 split.
  - Train/validation/test should make up roughly 80/10/10% of dataset (in time), resp. These proportions should be true globally & also within each composer. Maintaining these proportions is not always possible because some composers have few compositions in dataset.
  - Validation & test splits should contain various compositions. More popular compositions performed by many performers should be placed in training split.

  Table 1: Statistics of MAESTRO dataset replicated [29] contains aggregate statistics (thống kê tổng hợp) of MAESTRO dataset.

  1st translate "sustain pedal" controls into longer note duration to process MAESTRO MIDI files for training & evaluation. Sustain will extend a note until either sustain is turned off or same note is played again if that note is active. As a result of this process, note durations are same as those included in dataset's text files.

  Several datasets containing both piano audio & MIDI have been published previously, enabling significant progress in automatic piano transcription. Table 2: Piano MIDI datasets represents existing piano MIDI datasets.

  MAESTRO differs from existing datasets in several properties that affect model training:

  MusicNet (Thickstun et al., 2017) contains recordings of human performances but separately-sourced scores. As Hawthorne et al. (2018) discussed, alignment between audio & score is not entirely accurate. MusicNet offers a greater variety of recording environments & instruments besides piano (not included in Table 2).

  MAPS (Emiya et al., 2010) includes synthesized audio created from MIDI files entered via sequencer & Disklavier recordings. As such, "performances" are not as natural as MAESTRO performances captured from live performances. Moreover, synthesized audio accounts for a significant portion of MAPS dataset. Aside from individual notes & chords, MAPS also contains syntheses & recordings.

  Saarland Music Data (SMD) (Muller et al., 2011) contains recordings of human performances on Disklaviers, but it is 30 times smaller than MAESTRO.

GiantMIDI-Piano is largest dataset, including 38,700,838 transcribed notes & 10,855 unique solo piano works composed by 2786 composers. Table 3: Piano transcription evaluation on GiantMIDI-Piano dataset based on [33] represents alignment performance. Median alignment S_M, D_M, I_M, ER_M on MAESTRO dataset are 0.009, 0.024, 0.021, & 0.061, resp. Median alignment S_G, D_G, I_G, ER_G on GiantMIDI-Piano dataset are 0.015, 0.051, 0.069, & 0.154, resp. [33].

Proposed research endeavors to offer a novel augmentation to dataset by through integration of MAESTRO dataset & GIANT-MIDI piano dataset, standardizing annotation, & applying augmentation. MAESTRO dataset, which contains > 172 hours of virtuosic piano performances captured with fine alignment between note labels & audio waveforms, is combined with GIANT-MIDI piano dataset. Annotation of combined dataset is standardized to produce pairs of audio & MIDI files time-aligned to represent same musical events.

- 6.1. Augmentation. (Tăng cường) Augmentation serves as a pivotal stage in enhancing dataset diversity & refining ML model generalization. 1 common technique is time stretching, altering audio signal duration while preserving pitch. This manipulation can be executed using formula

$$y(t) = x\left(\frac{t}{\alpha}\right), \tag{146}$$

where $x(t)$: original audio signal, $y(t)$: time-stretched signal, α: time-stretching factor. Another significant method is pitch shifting (thay đổi cao độ), which alters pitch while maintaining original duration. This is achieved through:

$$y(t) = x\left(\frac{t}{\beta}\right), \tag{147}$$

where $x(t)$: original audio, $y(t)$: pitch-shifted signal, & β: pitch-shifting factor. Applying these techniques to dataset increases its diversity & robustness, improving model performance. Our research aims to enhance automatic piano transcription with an expanded dataset, standardized annotation, & augmented data, facilitating more precise ML models. Aim: generate piano transcriptions containing perceptually relevant performance information without considering recording environment. To achieve this, need a numerical measure. Poor-quality transcriptions can still score high due to short, spurious, & repeated notes. High note-with-offset scores capture perceptual information from onsets & durations, ensuring dynamics capture. This study improves note-with-offset scores, achieves state-of-art results for frame & note scores, & extends model to transcribe velocity scores (ghi chép điểm tốc độ).

- 7. Feature Representations Computational Models.
- 8. Training.
- 9. Evaluation of Automatic Piano Transcription.

2.8 [Kut23]. Gitta Kutyniok. Mathematics of Artificial Intelligence

[35 citations]

- Abstract. Currently witness spectacular success of AI in both science & public life. However, development of a rigorous mathematical foundation is still at an early stage. In this survey article, which is based on an invited lecture at ICM2022, in particular focus on current "workhorse" of AI, namely deep neural networks. Present main theoretical directions along with several exemplary results & discuss key open problems.
- Keywords. applied harmonic analysis, approximation theory, DL, inverse problems, PDEs
- 1. Introduction. AI is currently leading to 1 breakthrough after another, both in public with, e.g., autonomous driving & speech recognition, & in sciences in areas e.g. medical diagnostics or molecular dynamics. In addition, research on AI &, in particular, on its theoretical foundations is progressing at an unprecedented rate. One can envision: corresponding methodologies will in future drastically change way we live in numerous respects.
  - 1.1. Rise of AI. AI is, however, not a new phenomenon. In fact, already in 1943, McCulloch & Pitts started to develop algorithmic approaches to learning by mimicking functionality of human brain, through artificial neurons which are connected & arranged in several layers to form artificial neural networks. Already at that time, they had a vision for implementation of AI. However, community did not fully recognize potential of neural networks. Therefore, this 1st wave of AI was not successful & vanished. Around 1980, ML became popular again, & several highlights can be reported from that period.

    Real breakthrough & with it a new wave of AI came around 2010 with extensive application of *deep* neural networks. Today, this model might be considered "workhorse" of AI, & in this article, focus predominantly on this approach. Structure of deep neural networks is precisely structure McCulloch & Pitts introduced, namely numerous consecutive layers of artificial neurons. Today 2 main obstacles from previous years have also been eliminated; due to drastic improvement of computing power, training of neural networks with hundreds of layers in sense of *deep* neural networks is feasible, & we are living in age of data, hence vast amounts of training data are easily available.

- 1.2. Impact on mathematics. Rise of AI also had a significant impact on various fields of mathematics. Maybe 1st area which embraced these novel methods was area of inverse problems, in particular, imaging science, where such approaches have been used to solve highly ill-posed problems e.g. denoising, inpainting, superresolution, or (limited-angle) computed tomography, to name a few. One might note: due to lack of a precise mathematical model of what an image is, this area is particularly suitable for learning methods. Thus, after a few years, a change of paradigm could be observed, & novel solvers are typically at least to some extent based on methods from AI.

  Area of PDEs was much slower to embrace these new techniques, reason being: it was not per se evident what advantage of methods from AI for this field would be. Indeed, there seems to be no need to utilize learning-type methods, since a PDE is a rigorous mathematical model. But, lately, observation: deep neural networks are able to beat curse of dimensionality in high-dimensional settings led to a change of paradigm in this area as well. Research at intersection of numerical analysis of PDEs & AI therefore accelerated since abut 2017. Will delve further into this topic in Sect. 4.2.
- 1.3. Problems of AI. However, as promising as all these developments seem to be, a word of caution is required. besides fact: practical limitations of methods e.g. deep neural networks have not been explored at all & at present neural networks are still considered a "jack-of-all-trades," even more worrisome that a comprehensive theoretical foundation is completely lacking. This was very prominently stated during major conference on AI & ML, which is NIPS (today called NeurIPS) in 2017, when Ali Rahimi from Google received Test of Time Award & during his plenary talk stated: "ML has become a form of alchemy." (Học máy đã trở thành một hình thức thuật giả kim). This raised a heated discussion to which extent a theoretical foundation does exist & is necessary at all. From a mathematical viewpoint, crystal clear: a fundamental mathematical understanding of AI is inevitably necessary, & one has to admit: its development is currently in a preliminary state at best.

  This lack of mathematical foundations, e.g., in case of deep neural networks, results in a time-consuming search for a suitable network architecture, a highly delicate trial-&-error-based (training) process, & missing error bounds for performance of trained neural network. One needs to stress: in addition, such approaches also sometimes unexpectedly fail dramatically when a small perturbation of input data causes a drastic change of output leading to radically different – & often wrong – decisions. Such adversarial examples are a well-known problem, which becomes severe in sensitive applications e.g. when minor alterations of traffic signs, e.g., placement of stickers, cause autonomous vehicles to suddenly reach an entirely wrong decision. Evident: such robustness problems can only be tackled by a profound mathematical approach.
- 1.4. A need for mathematics. These considerations show: there is a tremendous need for mathematics in area of AI. One can currently witness: numerous mathematicians move to this field, bringing in their own expertise. Indeed, discussed in Sect. 2.4, basically all areas of mathematics are required to tackle various difficult, but exciting challenges in area of AI.

  One can identify 2 different research directions at intersection of mathematics & AI:

  * *Mathematical Foundations for AI.* This direction aims for deriving a deep mathematical understanding. Based on this, it strives to overcome current obstacles e.g. lack of robustness or places entire training process on a solid theoretical foundation.
  * *AI for Mathematical Problems.* This direction focuses on mathematical problem settings e.g. inverse problems & PDEs with goal of employing methodologies from AI to develop superior solvers.
- 1.5. Outline. Both research directions will be discussed in this survey paper, showcasing some novel results & pointing out key future challenges for mathematics. Start with an introduction into mathematical setting, stating main defs & notations (Sect. 2). Next, in Sect. 3, delve into 1st main direction, namely mathematical foundations for AI, & discuss research threads of expressivity, optimization, generalization, & explainability. Sect. 4 is then devoted to 2nd main direction, which is AI for mathematical problems, & highlight some exemplary results. Finally, Sect. 5 states 7 main mathematical problems & concludes.

- 2. Mathematical Setting of AI. Get into more details on precise def of a deep neural network, which is after all a purely mathematical object. Also touch upon typical application setting & training process, as well as current key mathematical directions.

  - 2.1. Def of deep neural networks. Core building blocks are artificial neurons. For their def, recall structure & functionality of a neuron in human brain. Basic elements of such a neuron are dendrites, through which signals are transmitted to its soma while being scaled/amplified due to structural properties of respective dendrites. In soma of neuron, those incoming signals are accumulated, & a decision is reached whether to fire to other neurons or not, & also with which strength.

    – Các khối xây dựng cốt lõi là các tế bào thần kinh nhân tạo. Đối với định nghĩa của chúng, hãy nhớ lại cấu trúc & chức năng của một tế bào thần kinh trong não người. Các thành phần cơ bản của một tế bào thần kinh như vậy là các nhánh cây, qua đó các tín hiệu được truyền đến thân tế bào của nó trong khi được mở rộng/khuếch đại do các đặc tính cấu trúc của các nhánh cây tương ứng. Trong thân tế bào thần kinh, các tín hiệu đến đó được tích lũy, & một quyết định được đưa ra là có nên kích hoạt đến các tế bào thần kinh khác hay không, & cũng như với cường độ nào.

    This forms basis for a mathematical def of an artificial neuron.

    Definition 6. *An* artificial neuron *with* weights $w_1, \dots, w_n \in \mathbb{R}$*, bias* $b \in \mathbb{R}$*, &* activation function $\rho : \mathbb{R} \to \mathbb{R}$ *is defined as function* $f : \mathbb{R}^n \to \mathbb{R}$ *given by*

    $$f(x_1, \dots, x_n) = \rho\left(\sum_{i=1}^{n} x_i w_i - b\right) = \rho\left(\langle \mathbf{x}, \mathbf{w} \rangle - b\right), \tag{148}$$

    where $\mathbf{w} = (w_1, \dots, w_n), \mathbf{x} = (x_1, \dots, x_n)$.

By now, there exists a zoo of activation functions with most well-known ones being as follows:

1. Heaviside function
$$\rho(x) = \begin{cases} 1 & x > 0, \\ 0 & x \leq 0. \end{cases} \tag{149}$$
2. Sigmoid function $\rho(x) = \frac{1}{1+e^{-x}}$.
3. Rectifiable Linear Unit (ReLU) $\rho(x) = \max\{0, x\}$.

Remark that of these examples, by far most extensively used activation function is ReLU due to its simple piecewise-linear structure, which is advantageous in training process & still allows suprior performance.

Similar to structure of a human brain, these artificial neurons are now being concatenated & arranged in layers, leading to an (artificial feed-forward) neural network. Due to particular structure of artificial neurons, such a neural network consists of compositions of affine linear maps & activation functions. Traditionally, a deep neural network is then defined as resulting function. From a mathematical standpoint, this bears difficulty that different arrangements lead to same function. Therefore, sometimes a distinction is made between architecture of a neural network & corresponding realization function (see, e.g., [6]). For this article, will, however, avoid such technical delicacies & present most standard def.

Definition 7. *Let $d \in \mathbb{N}$ be dimension of input layer, L number of layers, $N_0 \coloneqq d, N_l, l = 1, \dots, L$, dimensions of hidden & last layer, $\rho : \mathbb{R} \to \mathbb{R}$ a (nonlinear) activation function, &, for $l = 1, \dots, L$, let T_l be affine functions*
$$T_l : \mathbb{R}^{N_{l-1}} \to \mathbb{R}^{N_l},\ T_l\mathbf{x} = W^{(l)}\mathbf{x} + b^{(l)}, \tag{150}$$
with $W^{(l)} \in \mathbb{R}^{N_l \times N_{l-1}}$ being weight matrices & $b^{(l)} \in \mathbb{R}^{N_l}$ bias vectors of lth layer. Then $\Phi : \mathbb{R}^d \to \mathbb{R}^{N_L}$, given by $\Phi(x) = T_L\rho(T_{L-1}\rho(\cdots\rho(T_1(x))))$, $x \in \mathbb{R}^d$, is called a (deep) neural network *of* depth L.

Weights & biases are free parameters which will be learned during training process. An illustration of multilayered structure of a deep neural network can be found in Fig. 1: Deep neural network $\Phi : \mathbb{R}^4 \to \mathbb{R}$ with depth 5.

◦ 2.2. Application of a deep neural network. Aiming to identify main mathematical research threads, 1st have to understand how a deep neural network is used for a given application setting.

∗ *Step 1 (Train-test split of dataset).* Assume given samples $(\mathbf{x}^{(i)}, y^{(i)})_{i=1}^{\tilde{m}}$ of inputs & outputs. Task of deep neural network is then to identify relation between those. E.g., in a classification problem, each output $y^{(i)}$ is considered to be label of respective class to which input $\mathbf{x}^{(i)}$ belongs. One can also take viewpoint: $(\mathbf{x}^{(i)}, y^{(i)})_{i=1}^{\tilde{m}}$ arise as samples from a function e.g. $g : \mathcal{M} \to \{1, 2, \dots, K\}$, where $\mathcal{M}$ might be a lower-dimensional manifold of $\mathbb{R}^d$, in sense of $y^{(i)} = g(\mathbf{x}^{(i)})\ \forall i = 1, \dots, \tilde{m}$. Set $(\mathbf{x}^{(i)}, y^{(i)})_{i=1}^{\tilde{m}}$ is then split into a training data set $(\mathbf{x}^{(i)}, y^{(i)})_{i=1}^{m}$ & a test data set $(\mathbf{x}^{(i)}, y^{(i)})_{i=m+1}^{\tilde{m}}$. Training data set is – as name indicates – used for training, whereas test data set will later on be solely exploited for testing performance of trained network. Emphasize: neural network is not exposed to test data set during entire training process.

∗ *Step 2 (Choice of architecture).* For preparation of learning algorithm, architecture of neural network needs to be decided upon, i.e., number of layers L, number of neurons in each layer $(N_l)_{l=1}^{L}$, & activation function ρ have to be selected. Known: a fully connected neural network is often difficult to train, hence, in addition, one typically preselects certain entries of weight matrices $(W^{(l)})_{l=1}^{L}$ to already be set to 0 at this point.
For later purposes, define selected class of deep neural networks by $\mathcal{NN}_\theta$ with θ encoding this chosen architecture.

∗ *Step 3 (Training).* Next step is actual training process, which consists of learning affine functions $(T_l)_{l=1}^{L} = (W^{(l)} \cdot + b^{(l)})_{l=1}^{L}$. This is accomplished by minimizing *empirical risk* (2.1)
$$\widehat{\mathcal{R}}(\Phi_{(W^{(l)}, b^{(l)})_l}) \coloneqq \frac{1}{m}\sum_{i=1}^{m}(\Phi_{(W^{(l)}, b^{(l)})_l}(\mathbf{x}^{(i)}) - y^{(i)})^2. \tag{151}$$
A more general form of optimization problem: (2.2)
$$\min_{(W^{(l)}, b^{(l)})_l} \sum_{i=1}^{m} \mathcal{L}(\Phi_{(W^{(l)}, b^{(l)})_l}(\mathbf{x}^{(i)}), y^{(i)}) + \lambda\mathcal{P}((W^{(l)}, b^{(l)})_l), \tag{152}$$
where $\mathcal{L}$: a loss function to determine a measure of closeness between network evaluated in training samples & (known) values $y^{(i)}$, with $\mathcal{P}$ being a penalty/regularization term to impose additional constraints on weight matrices & bias vectors. 1 common algorithmic approach is gradient descent. Since, however, m is typically very large, this is computationally not feasible. This problem is circumvented by randomly selecting only a few gradients in each iteration, assuming: they constitute a reasonable average, which is coined *stochastic gradient descent*.
Solving optimization problem then yields a network $\Phi_{(W^{(l)}, b^{(l)})_l} : \mathbb{R}^d \to \mathbb{R}^{N_L}$, where
$$\Phi_{(W^{(l)}, b^{(l)})_l}(\mathbf{x}) = T_L\rho(T_{L-1}\rho(\cdots\rho(T_1(\mathbf{x})))). \tag{153}$$

∗ *Step 4 (Testing).* Finally, performance (often also called *generalization ability*) of trained neural network is tested using test data set $(\mathbf{x}^{(i)}, y^{(i)})_{i=m+1}^{\tilde{m}}$ by analyzing whether
$$\Phi_{(W^{(l)}, b^{(l)})_l}(\mathbf{x}^{(i)}) \approx y^{(i)},\ \forall i = m+1, \dots, \tilde{m}. \tag{154}$$

◦ 2.3. Relation to a statistical learning problem. From procedure above, can already identify selection of architecture, optimization problem, & generalization ability as key research directions for mathematical foundations of deep neural network. Considering entire learning process of a deep neural network as a statistical learning problem reveals those 3 research directions as indeed natural ones for analyzing overall error.

For this, assume: there exists a function $g : \mathbb{R}^d \to \mathbb{R}$ s.t. training data $(\mathbf{x}^{(i)}, y^{(i)})_{i=1}^m$ is of form $(\mathbf{x}^{(i)}, g(\mathbf{x}^{(i)}))_{i=1}^m$ & $\mathbf{x}^{(i)} \in [0,1]^d$ $\forall i = 1, \dots, m$. A typical continuum viewpoint to measure success of training: consider *risk* of a function $f : \mathbb{R}^d \to \mathbb{R}$ given by

$$\mathcal{R}(f) \coloneqq \int_{[0,1]^d} (f(\mathbf{x}) - g(\mathbf{x}))^2 \, \mathrm{d}\mathbf{x}, \tag{155}$$

where used L^2-norm to measure distance between f, g. Error between trained deep neural network $\Phi^0 (\coloneqq \Phi_{(W^{(l)}, b^{(l)})_l}) \in \mathcal{NN}_\theta$ & optimal function g can then be estimated by ($\leq$ Optimization error + Generalization error + Approximation error)

$$\mathcal{R}(\Phi^0) \leq [\widehat{\mathcal{R}}(\Phi^0) - \inf_{\Phi \in \mathcal{NN}_\theta} \widehat{\mathcal{R}}(\Phi)] + 2 \sup_{\Phi \in \mathcal{NN}_\theta} |\mathcal{R}(\Phi) - \widehat{\mathcal{R}}(\Phi)| + \inf_{\Phi \in \mathcal{NN}_\theta} \mathcal{R}(\Phi). \tag{156}$$

These considerations lead to main research threads described in:

◦ 2.4. Main research threads. Can identify 2 conceptually different research threads, 1st being focused on developing mathematical foundations of AI & 2nd aiming to use methodologies from AI to solve mathematical problems. Intriguing to see how both have already led to some extent to a paradigm shift in some mathematical research areas, most prominently area of numerical analysis.

∗ 2.4.1. Mathematical foundations for AI. Following up on discussion in Sect. 2.3, can identify 3 research directions which are related to 3 types of errors which one needs to control in order to estimate overall error of entire training process:

· *Expressivity.* This direction aims to derive a general understanding whether & to which extent aspects of a neural network architecture affect best case performance of deep neural networks. More precisely, goal: analyze approximation error $\inf_{\Phi \in \mathcal{NN}_\theta} \mathcal{R}(\Phi)$ from (2.4), which estimates approximation accuracy when approximating g by hypothesis class $\mathcal{NN}_\theta$ of deep neural networks of a particular architecture. Typical methods for approaching this problem are from applied harmonic analysis & approximation theory.

· *Learning/ Optimization.* Main goal of this direction is analysis of training algorithm e.g. stochastic gradient descent, in particular, asking why it usually converges to suitable local minima even though problem itself is highly nonconvex. This requires analysis of optimization error, which is $\widehat{\mathcal{R}}(\Phi^0) - \inf_{\Phi \in \mathcal{NN}_\theta} \widehat{\mathcal{R}}(\Phi)$ (cf. (2.4)) & which measures accuracy with which learnt neural network Φ^0 minimizes empirical risk (2.1), (2.2). Key methodologies for attacking such problems come from areas of algebraic/differential geometry, optimal control, & optimization.

· *Generalization.* This direction aims to derive an understanding of out-of-sample error, namely $\sup_{\Phi \in \mathcal{NN}_\theta} |\mathcal{R}(\Phi) - \widehat{\mathcal{R}}(\Phi)|$ from (2.4), which measures distance of empirical risk (2.1), (2.2), & actual risk (2.3). Predominantly, learning theory, probability theory, & statistics provide required methods for this research thread.

A very exciting & highly relevant new research direction has recently emerged, coined explainability. At present, it is from standpoint of mathematical foundations still a wide open field.

· *Explainability.* This direction considers deep neural networks, which are already trained, but no knowledge about training is available; a situation one encounters numerous times in practice. Goal: derive a deep understanding of how a given trained deep neural network reaches decisions in sense of which features of input data are crucial for a decision. Range of required approaches is quite broad, including areas e.g. information theory or uncertainty quantification.

∗ 2.4.2. AI for mathematical problems. Methods of AI have also turned out to be extremely effective for mathematical problem settings. In fact, area of inverse problems, in particular, in imaging sciences, has already undergone a profound paradigm shift. & area of numerical analysis of PDEs seems to soon follow same path, at least in very high dimensional regime. Briefly characterize those 2 research threads similar to previous subsect on mathematical foundations of AI.

· *Inverse Problems.* Research in this direction aims to improve classical model-based approaches to solve inverse problems by exploiting methods of AI. In order to not neglect domain knowledge e.g. physics of problem, current approaches aim to take best out of both worlds in sense of optimally combining model- & data-driven approaches. This research direction requires a variety of techniques, foremost from areas e.g. imaging science, inverse problems, & microlocal analysis, to name a few.

· *PDEs.* Similar to area of inverse problems, goal: improve classical solvers of PDEs by using ideas from AI. A particular focus is on high-dimensional problems in sense of aiming to beat curse of dimensionality. This direction obviously requires methods from areas e.g. numerical mathematics & PDEs.

• 3. Mathematical Foundations for AI. This sect shall serve as an introduction into main research threads aiming to develop a mathematical foundation for AI. Introduce problem settings, showcase some exemplary results, & discuss open problems.

◦ 3.1. Expressivity. Expressivity is maybe richest area at present in terms of mathematical results. General question can be phrased as follows: Given a function class/space $\mathcal{C}$ & a class of deep neural networks $\mathcal{NN}_\theta$, how does approximation accuracy when approximating elements of $\mathcal{C}$ by networks $\Phi \in \mathcal{NN}_\theta$ relate to complexity of such Φ? Making this precise thus requires introduction of a complexity measure for deep neural networks. In sequel, choose canonical one, which is complexity in terms of memory requirements. Notice: certainly various other complexity measures exist. Further, recall: $\| \cdot \|_0$-"norm" counts number of nonzero components.

Definition 8. *Retaining same notation for deep neural networks as in Def. 2.2,* complexity $C(\Phi)$ *of a deep neural network* Φ *is defined by*

$$C(\Phi) \coloneqq \sum_{l=1}^{L} (\|W^{(l)}\|_0 + \|b^{(l)}\|_0). \tag{157}$$

Most well-known – & maybe even 1st – result on expressivity is universal approximation theorem [8,13]. State: each continuous function on a compact domain can be approximated up to an arbitrary accuracy by a shallow neural network.

- 3.2. Optimization.
- 3.3. Generalization.
- 3.4. Explainability.

- 4. AI for Mathematical Problems. Turn to research direction of AI for mathematical problems, with 2 most prominent problems being inverse problems & PDEs. As before, introduce problem settings, showcase some exemplary results, & also discuss open problems.

  - 4.1. Inverse problems. Methods of AI, in particular, deep neural networks, have a tremendous impact on area of inverse problems. 1 current major trend: optimally combine classical solvers with DL in sense of taking best out of model- & data-world.

    To introduce such results, start by recalling some basics about solvers of inverse problems. For this, assume: given an (ill-posed) inverse problem

    $$Kf = g, \tag{158}$$

    where $K : X \to Y$: an operator & X, Y: e.g., Hilbert spaces. Drawing from area of imaging science, examples include denoising, deblurring, or inpainting (recovery of missing parts of an image). Most classical solvers are of form (which includes Tikhonov regularization)

    $$f^\alpha \coloneqq \arg\min_f (\|Kf - g\|^2 + \alpha \mathcal{P}(f)), \tag{159}$$

    where $\|Kf - g\|^2$: data fidelity term, $\mathcal{P}(f)$: penalty/regularization term, $\mathcal{P} : X \to \mathbb{R}$, $f^\alpha \in X, \alpha > 0$: an approximate solution of inverse problem (4.1). 1 very popular & widely applicable special case is *sparse regularization*, where $\mathcal{P}$ is chosen by $\mathcal{P}(f) \coloneqq \|(\langle f, \varphi_i \rangle)_{i \in I}\|_1$ & $(\varphi_i)_{i \in I}$: a suitably selected orthonormal basis or a frame for X.

    Turn to DL approaches to solve inverse problems, which might be categorized into 3 classes:

    * *Supervised approaches.* An ad hoc approach in this regime is given in [14], which 1st applies a classical solver followed by a neural network to remove reconstruction artifacts. More sophisticated approaches typically replace parts of classical solver by a custom-built neural network [26] or a network specifically trained for this task [1].
    * *Semisupervised approaches.* These approaches encode regularization as a neural network with an example being adversarial regularizers [20].
    * *Unsupervised approaches.* A representative of this type of approaches: technique of deep image prior [29]. This method interestingly shows: structure of a generator network is sufficient to capture necessary statistics of data prior to any type of learning.

    Aiming to illustrate superiority of approaches from AI for inverse problems, now focus on inverse problem of computed tomography (CT) from medical imaging. Forward operator K in this setting is *Radon transform*, defined by

    $$\mathcal{R}f(s, \vartheta) = \int_{\mathbb{R}} f(s\omega(\vartheta) + t\omega(\varepsilon)^\perp)\, \mathrm{d}t, \ (s, \vartheta) \in \mathbb{R} \times (0, \pi). \tag{160}$$

    Here $\omega(\vartheta) \coloneqq (\cos\vartheta, \sin\vartheta)$: unitary vector with orientation described by angle ϑ w.r.t. x_1-axis & $\omega(\vartheta)^\perp \coloneqq (-\sin\vartheta, \cos\vartheta)$. Often, only parts of so-called sinogram $\mathcal{R}f$ can be acquired due to physical constraints as in, e.g., electron tomography. Resulting, more difficult problem is termed *limited-angle CT*. Notice: this problem is even harder than problem of low-dose CT, where not an entire block of measurements is missing, but angular component is "only" undersampled.

    Most prominent features in images f are edge structures. This is also due to fact: human visual system reacts most strongly to those. These structures in turn can be accurately modeled by microlocal analysis, in particular, by notion of wavefront sets $WF(f) \subseteq \mathbb{R}^2 \times \mathbb{S}$, which – coarsely speaking – consist of singularities together with their direction. Basing in this sense application of a deep neural network on microlocal considerations, in particular, also using a DL-based wavefront set detector [2] in regularization term, reconstruction performance significantly outperforms classical solvers e.g. sparse regularization with shearlets (see Fig. 6: CT reconstruction from Radon measurements with a missing angle of 40°., also refer to [3] for details). Notice: this approach is of a hybrid type & takes best out of both worlds in sense of combining model- & AI-based approaches.

    Finally, DL-based wavefront set extraction itself is yet another evidence of improvements on state-of-art now possible by AI. Fig. 7: Wavefront set detection by a model-based & a hybrid approach. shows a classical result from [23], whereas [2] uses shearlet transform as a coarse edge detector, which is subsequently combined with a deep neural network.

  - 4.2. PDE.s 2nd main range of mathematical problem settings, where methods from AI are very successfully applied to, are PDEs. Although benefit of such approaches was not initially clear, both theoretical & numerical results show their superiority in high-dimensional regimes.

Most common approach aims to approximate solution of a PDE by a deep neural network, which is trained according to this task by incorporating PDE into loss function. More precisely, given a PDE $\mathcal{L}(u) = f$, train a neural network Φ s.t. $\mathcal{L}(\Phi \approx f$. Since 2017, research in this general direction has significantly accelerated. Some of highlights are Deep Ritz method [10] & Physics Informed Neural Networks [22], or a very general approach for high-dimensional parabolic PDEs [12].

Note: most theoretical results in this regime are of an expressivity type & also study phenomenon whether & to which extent deep neural networks are able to beat curse of dimensionality. In sequel, briefly discuss 1 such result as an example. In addition, notice: there already exist contributions – though very few – which analyze learning & generalization aspects.

Let $\mathcal{L}(u_y, y) = f_y$ denote a parametric PDE with y being a parameter from a high-dimensional parameter space $\mathcal{Y} \subset \mathbb{R}^p$ & u_y associated solution in a Hilbert space $\mathcal{H}$. After a high-fidelity discretization, let $b_y(u_y^h, v) = f_y(v)$ be associated variational form with u_y^h now belonging to associated high-dimensional space U^h, where set $D \coloneqq \dim U^h$. Denote coefficient vector of u_y^h w.r.t. a suitable basis of U^h by $\mathbf{u}_y^h$. Of key importance in this area: *parametric map* given by

$$\mathbb{R}^p \supseteq \mathcal{Y} \ni y \mapsto \mathbf{u}_y^h \in \mathbb{R}^D \text{ s.t. } b_y(u_y^h, v) = f_y(v),\ \forall v, \tag{161}$$

which in multiquery situations e.g. complex design problems needs to be solved several times. If p is very large, curse of dimensionality could lead to an exponential computational cost.

Aim to analyze whether parametric map can be solved by a deep neural network, which would provide a very efficient & flexible method, hopefully also circumventing curse of dimensionality in an automatic manner. From an expressivity viewpoint, one might ask whether, for each $\varepsilon > 0$, there exists a neural network Φ s.t. (4.2)

$$\|\Phi(y) - \mathbf{u}_y^h\| \leq \varepsilon\ \forall y \in \mathcal{Y}. \tag{162}$$

Ability of this approach to tackle curse of dimensionality can then be studied by analyzing how complexity of Φ depends on p, D. A result of this type was proven in [18].

Theorem 1. *There exists a neural network Φ which approximates parametric map, i.e., which satisfies (4.2), & dependence of $C(\Phi)$ on p, D can be (polynomially) controlled.*

Analyzing learning procedure & generalization ability of neural network in this setting is currently out of reach. Aiming to still determine whether a trained neural networks does not suffer from curse of dimensionality as well in [11] extensive numerical experiments were performed, which indicate: indeed, curse of dimensionality is also beaten in practice.

- 5. Conclusion: 7 Mathematical Key Problems. Conclude with 7 mathematical key problems of AI as they were stated in [6]. Those constitute main obstacles in *Mathematical Foundations for AI* with its subfields being expressivity, optimization, generalization, & explainability, as well as in *AI for Mathematical Problems*, which focus on application to inverse problems & PDEs.

  1. What is role of depth?
  2. Which aspects of a neural network architecture affect performance of DL?
  3. Why does stochastic gradient descent converge to good local minima despite nonconvexity of problem?
  4. Why do large neural networks not overfit?
  5. Why do neural networks perform well in very high-dimensional environments?
  6. Which features of data are learned by deep architectures?
  7. Are neural networks capable of replacing highly specialized numerical algorithms in natural sciences?

2.9 Gabriel Peyré. The Mathematics of Artificial Intelligence. 2025

- Abstract. This overview article highlights critical role of mathematics in AI, emphasizing: mathematics provides tools to better understand & enhance AI systems. Conversely, AI raises new problems & drives development of new mathematics at intersection of various fields. This article focuses on application of analytical & probabilistic tools to model neural network architectures & better understand their optimization. Statistical questions (particularly generalization capacity of these networks) are intentionally set aside, though they are of crucial importance. Also shed light on evolution of ideas that have enabled significant advances in AI through architectures tailored to specific tasks, each echoing distinct mathematical techniques. Goal: encourage more mathematicians to take an interest in & contribute to this exciting field.

- 1. Supervised Learning. Recent advancements in AI have mainly stemmed from development of neural networks, particularly deep networks. 1st significant successes, after 2010, came from supervised learning, where training pairs $(x^i, y^i)_i$ are provided, with x^i representing data (e.g., images or text) & y^i corresponding labels (typically, classes describing content of data). More recently, spectacular progress has been achieved in unsupervised learning, where labels y_i are unavailable, thanks to techniques known as generative AI. These methods will be discussed in Sect. 4.

  Empirical Risk Minimization. In supervised learning, objective: construct a function $f_\theta(x)$, dependent on parameters θ, s.t. it approximates data well: $y^i \approx f_\theta(x^i)$. Dominant paradigm involves finding parameters θ by minimizing an empirical risk function, defined as

$$\min_\theta E(\theta) \coloneqq \sum_i l(f_\theta(x^i), y^i), \tag{163}$$

where l: a loss function, typically $l(y, y') = \|y - y'\|^2$ for vector-valued $(y^i)_i$. Optimization of $E(\theta)$ is performed using gradient descent:

$$\theta_{t+1} = \theta_t - \tau \nabla E(\theta_t), \tag{164}$$

where τ: step size. In practice, a variant known as *stochastic gradient descent* [23] is used to handle large datasets by randomly sampling a subset at each step t. A comprehensive theory for convergence of this method exists in cases where f_θ depends linearly on θ, as $E(\theta)$ is convex. However, in case of deep neural networks, $E(\theta)$ is non-convex, making its theoretical analysis challenging & still largely unresolved. In some instances, partial mathematical analyses provide insights into observed practical successes & guide necessary modifications to improve convergence. For a detailed overview of existing results, refer to [Bac24].

Automatic Differentiation. Computing gradient $\nabla E(\theta)$ is fundamental. FDMs or traditional differentiation of composite functions would be too costly, with complexity on order of $O(DT)$, where D: dimensionality of θ, T: computation time of f_θ. Advances in deep network optimization rely on use of automatic differentiation in reverse mode [13], often referred to as "backpropagation of gradient", with complexity on order of $O(T)$.

- 2. 2-Layer Neural Networks.

Multi-Layer Perceptrons. A multi-layer network, or multi-layer perceptron (MLP) [25], computes function $f_\theta(x) = x_L$ in L steps (or layers) recursively starting from $x_0 = x$:

$$x_{l+1} = \sigma(W_l x_l + b_l), \tag{165}$$

where W_l: a weight matrix, b_l: a bias vector, & σ: a nonlinear activation function, e.g. sigmoid $\sigma(s) = \frac{e^s}{1+e^s}$, which is bounded, or ReLU (Rectified Linear Unit), $\sigma(s) = \max(s, 0)$, which is unbounded. Parameters to optimize are weights & biases $\theta = (W_l, b_l)_l$. If σ were linear, then f_θ would remain linear regardless of number of layers. Nonlinearity of σ enriches set of functions representable by f_θ, by increasing both width (dimension of intermediate vectors x_k) & depth (number L of layers).

2-Layer Perceptrons & Universality. Mathematically best-understood case is that of $L = 2$ layers. Denoting $W_1 = (v_k)_{k=1}^n$ & $W_2^\top = (u_k)_{k=1}^n$ as n rows & columns of 2 matrices (where n: network width), can write f_θ as a sum of contributions from its n neurons: (4)

$$f_\theta(x) = \frac{1}{n} \sum_{k=1}^{n} u_k \sigma(\langle v_k, x \rangle + b_k). \tag{166}$$

Parameters are $\theta = (u_k, v_k, b_k)_{k=1}^n$. Here, added a normalization factor $\frac{1}{n}$ (to later study case $n \to \infty$) & ignored nonlinearity of 2nd layer. A classical result by Cybenko [8] shows: these functions f_θ can uniformly approximate any continuous function on a compact domain. This result is similar to Weierstrass theorem for polynomial approximation, except (4) defines, for a fixed n, a nonlinear function space. Elegant proof by Hornik [15] relies on Stone–Weierstrass theorem, which implies result for $\sigma = \cos$ (since [4]) defines an algebra of functions when n is arbitrary). Proof is then completed by uniformly approximating a cosine in 1D using a sum of sigmoids or piecewise affine functions (e.g., for $\sigma = \text{ReLU}$).

Mean-Field Representation. This result is, however, disappointing, as it does not specify how many neurons are needed to achieve a given approximation error. Impossible without adding assumptions about regularity of function to approximate. Barron's fundamental result [4] introduces a Banach space of regular functions, defined by semi-norm $\|f\|_B \coloneqq \int |\hat{f}(\omega)||\omega| \, \mathrm{d}\omega$, where $\hat{f}$ denotes Fourier transform of f. Barron shows: in L^2 norm, for any n, there exists a network f_θ with n neurons s.t. approximation error on a compact Ω of radius R is of order $\|f - f_\theta\|_{L^2(\Omega)} = O\left(\frac{R\|f\|_B}{\sqrt{n}}\right)$. This result is remarkable because it avoids "curse of dimensionality": unlike polynomial approximation, error does not grow exponentially with dimension d (although dimension affects constant $\|f\|_B$).

Barron's proof relies on a "mean-field" generalization of [4], where a distribution (a probability measure) ρ is considered over parameter space (u, v, b), & network is expressed as: (5)

$$F_\rho(x) \coloneqq \int u\sigma(\langle v, x \rangle + b) \, \mathrm{d}\rho(u, v, b). \tag{167}$$

A finite-size network (4), $f_\theta = F_{\hat{\rho}}$, is obtained with a discrete measure $\hat{\rho} = \frac{1}{n} \sum_{k=1}^n \delta_{(u_k, v_k, b_k)}$. An advantage of representation (5): F_ρ depends linearly on ρ, which is crucial for Barron's proof. This proof uses a probabilistic Monte Carlo-like method (where error decreases as $\frac{1}{\sqrt{n}}$ as desired): it involves constructing a distribution ρ from f, then sampling a discrete measure $\hat{\rho}$ whose parameters $(u_k, v_k, b_k)_k$ are distributed according to ρ.

Wasserstein Gradient Flow. In general, analyzing convergence of optimization (2) $\theta_{t+1} = \theta_t - \tau \nabla E(\theta_t)$ is challenging because function E is non-convex. Recent analyses have shown: when number of neurons n is large, dynamics are not trapped in a local minimum. Fundamental result by Chizat & Bach [7] is based on fact: distribution ρ_t defined by gradient descent (2), as $\tau_t \to 0$, follows a gradient flow in space of probability distributions equipped with optimal transport distance. These Wasserstein gradient flows, introduced by [16] & studied extensively in book [1], satisfy a McKean-Vlasov-type PDE:

$$\partial_t \rho_t + \nabla \cdot (\rho_t \mathcal{V}(\rho)) = 0, \tag{168}$$

where $x \to \mathcal{V}(\rho)(x)$ is a vector field depending on ρ, which can be computed explicitly from data $(x^i, y^i)_i$. Result by Chizat & Bach can be viewed both as a PDE result (convergence to an equilibrium of a class of PDEs with a specific vector field $\mathcal{V}(\rho)$) & as a ML result (successful training of a 2-layer network via gradient descent when number of neurons is sufficiently large).

- 3. Very Deep Networks. Unprecedented recent success of neural networks began with work of [17], which demonstrated: deep networks, when trained on large datasets, achieve unmatched performance. 1st key point: to achieve these performances, necessary to use weight matrices W_k that exploit structure of data. For images, this means using convolutions [18]. However, this approach is not sufficient for extremely deep networks, with number of layers L reaching into hundreds.

  Residual Networks. Major breakthrough that empirically demonstrated that network performance increases with L was introduction of residual connections, known as ResNet [14]. Main idea: ensure: for most layers x_l, dimensions of x_l & x_{l+1} are identical, & to replace (3) with L steps: (7)

$$x_{l+1} = x_l + \frac{1}{L} U_l^\top \sigma(V_l x_l + b_l), \tag{169}$$

  where $U_l, V_l \in \mathbb{R}^{n\times d}$: weight matrices, with n: number of neurons per layer (which, as in (4), can be increased to enlarge function space). Intuition for success of (7): this formula allows, unlike (3), for steps that are small deformations near identity mappings. This makes network $f_\theta(x_0) = x_L$ obtained after L steps wellposed even when L is large, & it can be rigorously proven [20]: this well-posedness is preserved during optimization via gradient descent (2).

  Neural Differential Equation. As L approaches inf, (7) can be interpreted as a discretization of an ODE: (8)

$$\frac{dx_s}{ds} = U_s^\top \sigma(V_s x_s + b_s), \tag{170}$$

  where $s \in [0,1]$ indexes network depth. Network $f_\theta(x_0) \coloneqq x_1$ maps initialization $x_{s=0}$ to solution $x_{s=1}$ of (8) at time $s = 1$. Parameters are $\theta = (U_s, V_s, b_s)_{s\in[0,1]}$. This formalization, referred to as a *neural ODE*, was initially introduced in [6] to leverage tools from adjoint equation theory to reduce memory cost of computing gradient $\nabla E(\theta)$ during backpropapagation. It also establishes a connection to control theory, as training via gradient descent (2) computes an *optimal control* θ that interpolates between data x^i & labels y^i. However, specificity of learning theory compared to control theory lies in goal of computing such control using gradient descent (2). To date, no detailed results exist on this. Nonetheless, in his thesis, Raphael Barboni [3] demonstrated: if network is initialized near an interpolating network, gradient descent converges to it.

- 4. Generative AI for Vector Data.

  Self-Supervised Pre-Training. Remarkable success of large generative AI models for vector data, e.g. images (often referred to as "diffusion models"), & for text (large language models for LLMs), relies on use of new large-scale network architectures & creation of new training tasks known as "self-supervised". Manually defining labels y^i through human intervention is too costly, so these are calculated automatically by solving simple tasks. For images, these involve denoising tasks, while for text, they involve predicting next word. A key advantage of these simple tasks (known as pretraining tasks): it becomes possible to use a pre-trained network f_θ generatively. Starting from an image composed of pure noise & iterating on network, one can randomly generate a realistic image [27]. Similarly, for text, starting from a prompt & sequentially predicting next word, possible to generate text, e.g., to answer questions [5]. 1st describe case of vector data generation, e.g. images, & address LLMs for text in following sect.

  Sampling as an Optimization Problem. Generative AI models aim to generate (or "sample") random vectors x according to a distribution β, which is learned from a large training dataset $(x^i)_i$. This distribution is obtained by "pushing forward" a reference distribution α (most commonly an isotropic Gaussian distribution $\alpha = \mathcal{N}(0, \mathrm{Id})$) through a neural network f_θ. Specifically, if $X \sim \alpha$ is distributed according to α, then $f_\theta(X) \sim \beta$ follows law β, which is denoted as $(f_\theta)_\sharp \alpha = \beta$, where $\sharp$ represents pushforward operator (also known as image measure in probability theory).

  Early approaches to generative AI, e.g. "Generative Adversarial Networks" (GANs) [12] attempted to directly optimize a distance D between probability distributions (e.g., an optimal transport distance [22]): (9)

$$\min_\theta D((f_\theta)_\sharp \alpha, \beta). \tag{171}$$

  This problem is challenging to optimize because computing D is expensive, & β must be approximated from data $(x^i)_i$.

  Flow-Based Generation. Recent successes, particularly in generation of vector data, involve neural networks f_θ that are computed iteratively by integrating a flow [21], similar to a neural differential equation (8): (10)

$$f_0(\mathbf{x}_0) \coloneqq \mathbf{x}_1 \text{ where } \frac{d\mathbf{x}_s}{ds} = g_\theta(\mathbf{x}_s, s), \tag{172}$$

  where $g_\theta : (\mathbf{x}, s) \in \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$. Input space includes an additional temporal dimension s, compared to f_θ. Most effective neural networks for images are notably U-Nets (UNet) [24].

  Central mathematical question is how to replace (9) with a simpler optimization problem when f_θ is computed by integrating a flow (10). An extremely elegant solution was 1st proposed in context of diffusion models [27] (corresponding to specific case $\alpha = \mathcal{N}(0, \mathrm{Id})$) & later generalized under name "flow matching" [19].

  Main idea: if $x_0 \sim \alpha$ is initialized randomly, then x_s, solution at time s of (10), follows a distribution α_s that interpolates between $\alpha_0 = \alpha$ & α_1, which is desired to match β. This distribution satisfies a conservation equation: (11)

$$\partial_s \alpha_s + \nabla \cdot (\alpha_s \mathbf{v}_s) = 0, \tag{173}$$

where vector field is defined as $\mathbf{v}_s(\mathbf{x}) \coloneqq g_\theta(\mathbf{x}, s)$. This equation is similar to evolution of neuron distributions during optimization (6), but it is simpler because $\mathbf{v}_s$ does not depend on distribution α_s, making equation linear in α_s.

Denoising Pre-Training. Question: how to find a vector field $\mathbf{v}_s(\mathbf{x}) = g_\theta(\mathbf{x}, s)$ s.t. $\alpha_1 = \beta$, i.e., final distribution matches desired one. If $\mathbf{v}_s$ is known, distribution α_s is uniquely determined. Key idea: reason in reverse: starting from an interpolation α_s satisfying $\alpha_0 = \alpha, \alpha_1 = \beta$, how can we compute a vector field $\mathbf{v}_s$ s.t. (11) holds? There are infinitely many possible solutions (since $\mathbf{v}_s$ can be modified by a conservative field), but for certain specific interpolations α_s, remarkably simple expressions can be derived.

1 example: interpolation obtained via barycentric averaging: take a pair $x_0 \sim \alpha, x_1 \sim \beta$, & define α_s as distribution of $(1-s)x_0 + sx_1$. It can be shown [19]: an admissible vector field is given by a simple conditional expectation: (12)

$$v_s(\mathbf{x}) = \mathbb{E}_{x_0\sim\alpha, x_1\sim\beta}(x_1 - x_0 | (1-s)x_0 + sx_1 = x). \tag{174}$$

Key advantage of this formula: conditional mean corresponds to a linear regression, which can be approximated using a neural network $v_s \approx g_\theta(\cdot, s)$. Expectation over $x_1 \sim \beta$ can be replaced by a sum over training data $(x^i)_i$, leading to following optimization problem:

$$\min_\theta E(\theta) \coloneqq \int_0^1 \mathbb{E}_{x_0\sim\alpha} \sum_i \|x^i - x_0 - g_\theta((1-s)x_0 + sx^i, s)\|^2 \,\mathrm{d}s. \tag{175}$$

This function $E(\theta)$ is an empirical risk function (1), & similar optimization techniques are used to find an optimal θ. In particular, stochastic gradient descent efficiently handles both integral $\int_0^1$ & expectation $\mathbb{E}_{x_0\sim\alpha}$.

Problem (12) corresponds to an unsupervised pre-training task: there are no labels y^i, but an artificial supervision task is created by adding random noise x_0 to data x^i. This task is called *denoising*. For textual data, a different pre-training task is used:

- 5. Generative AI for Text.

Tokenization & Next-Word Prediction. Generative AI methods for text differ from those used for vector data generation. Neural network architectures are different (they involve transformers, as will describe), & pre-training method is based not on denoising but on next-word prediction [28]. Worth noting: these transformer neural networks are now also used for image generation [9], but specific aspects related to causal nature of text remain crucial. 1st preliminary step, called "tokenization", consists of transforming input text into a sequence of vectors $X = (x[1], \ldots, x[P])$, where number P is variable (& may increase, e.g., when generating text from a prompt). Each token $x[p]$ is a vector that encodes a group of letters, generally at level of a syllable. Neural network $x = f_\theta(X)$ is applied to all tokens & predicts a new token x. During training, a large corpus of text $(X^i)_i$ is available. If denote by $\tilde{X}^i$ text X^i with last token x^i removed, minimize an empirical risk for prediction:

$$\min_\theta E(\theta) \coloneqq \sum_i l(f_\theta(\tilde{X}^i), x^i), \tag{176}$$

which is exactly similar to (1). When using a pre-trained network f_θ for text generation, one starts with a prompt X & iteratively adds a new token in an auto-regressive manner: $X \leftarrow [X, f_\theta(X)]$.

Transformers & Attention. Large networks used for text generation tasks are Transformer networks [10]. Unlike ResNet (7), these networks f_θ no longer operate on a single vector $\mathbf{x}$, but on a set of vectors $X = (x[1], \ldots, x[P])$ of size P. In Transformers, ResNet layer (7), operating on a single vector, is replaced by an *attention* layer, where all tokens interact through a barycentric combination of vectors $(Vx[q])_q$ where V is a matrix: (13)

$$\boxed{A_\omega(X)_p \coloneqq \sum_q M_{p,q}(Vx[q]) \text{ where } M_{p,q} \coloneqq \frac{e^{\langle Qx[p], Kx[q]\rangle}}{\sum_l e^{\langle Qx[p], Kx[l]\rangle}} \text{ with } \omega \coloneqq (Q, K, V).} \tag{177}$$

Coefficients $M_{p,q}$ are computed by normalized correlation between $x[p], x[q]$, depending on 2 parameter matrices Q, K. A Transformer $f_\theta(X)$ is then defined similarly to ResNet (7), iterating L attention layers with residual connections: (14)

$$X_{l+1} = X_l + \frac{1}{L} A_{\omega_l}(X_l). \tag{178}$$

Parameters θ of Transformer $f_\theta(X_0) = X_L$ with L layers are $\theta = (\omega_l = (Q_l, K_l, V_l))_0^{L-1}$.

This description is simplified: in practice, a Transformer network also integrates normalization layers & MLPs operating independently on each token. For text applications, attention must be causal, imposing $M_{i,j} = 0$ for $j > i$. This constraint is essential for recursively generating text & ensuring next-word prediction task is meaningful.

Mean-Field Representation of Attention. Attention mechanism (13) can be viewed as a system of interacting particles, & (14) describes evolution of tokens across depth. As with ResNet (8), in limit $L \to +\infty$, one can consider a system of coupled ODEs: (15)

$$\frac{dX_s}{ds} = A_{\omega_s}(X_s). \tag{179}$$

A crucial point: for non-causal Transformers, system $X_s = (x_s[p])_p$ is invariant under permutations of indices. Thus, this system can be represented as a probability distribution $\mu_s \coloneqq \frac{1}{P}\sum_p \delta_{x_s[p]}$ over token space. This perspective was adopted in Michael Sander's thesis [26], which rewrites attention as:

$$\mathcal{A}_\omega(x) \coloneqq \frac{\int e^{\langle Qx, Kx'\rangle} Vx' \,\mathrm{d}\mu(x')}{\int e^{\langle Qx, Kx'\rangle} \,\mathrm{d}\mu(x')} \text{ where } \omega \coloneqq (Q, K, V). \tag{180}$$

Particle system (15) then becomes a conservation equation for advection by vector field $\mathcal{A}_{\omega_s}(\mu)$: (16)

$$\partial_s \mu_s + \nabla \cdot (\mu_s \mathcal{A}_{\omega_s}(\mu_s)) = 0. \tag{181}$$

Surprisingly, this yields a McKean-Vlasov-type equation, similar to the one describing training of 2-layer MLPs (6), but with velocity field $\mathcal{A}_\omega(\mu)(x)$ replacing $\mathcal{V}(\rho)(u, v, b)$. However, here evolution occurs in token space x rather than in neuron space (u, v, b), & evolution variable corresponds to depth s, not optimization time t.

Unlike (6), evolution (16) is not a gradient flow in Wasserstein metric [26]. Nonetheless, in certain cases, this evolution can be analyzed, & it can be shown: measure μ_s converges to a single Dirac mass [11] as $s \to \infty$: tokens tend to cluster. Better understanding these evolutions, as well as optimization of parameters $\theta = (Q_s, K_s, V_s)_{s\in[0,1]}$ via gradient descent (2), remains an open problem. This problem can be viewed as a control problem for PDE (16).

- Conclusion. Mathematics plays a critical role in understanding & improving performance of deep architectures while presenting new theoretical challenges. Emergence of Transformers & generative AI raises immense mathematical problems, particularly for better understanding training of these networks & exploring structure of "optimal" networks. 1 essential question remains: whether LLM merely interpolates training data or is capable of genuine reasoning. Moreover, issues of resource efficiency & privacy in AI system development demand significant theoretical advancements, where mathematics will play a pivotal role (đóng vai trò then chốt). Whether designing resource-efficient models, ensuring compliance with ethical standards, or exploring fundamental limits of these systems, mathematics is poised to be an indispensable tool for future of AI.

2.10 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:

$$\frac{dS(t)}{dt} = -\frac{\beta}{N} S(t) I(t), \tag{182}$$

$$\frac{dI(t)}{dt} = \frac{\beta}{N} S(t) I(t) - \gamma I(t), \tag{183}$$

$$\frac{dR(t)}{dt} = \gamma I(t), \tag{184}$$

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:

$$f^* = \min_{f \in \mathcal{H}} \mathcal{L}(f; \mathcal{D}) + \lambda \Omega(f), \tag{185}$$

  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:

$$\Omega(f) = \frac{1}{L} \sum_{i=1}^{L} (\partial f_{\text{NN}}(x_i) - \partial f_{\text{PDE}}(x_i))^2, \tag{186}$$

  where $\partial f_{\text{NN}}(x_i)$ denotes partial derivative of neural network evaluated at x_i & $\partial f_{\text{PDE}}(x_i)$ 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 $\boxed{\text{physics prior}}$, 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.

2.11 [RLM22]. Sebastian Raschka, Yuxi (Hayden) Liu, Vahid Mirjalili. Machine Learning with PyTorch & Scikit-Learn: Develop ML & DL Models with Python. 2023

PyTorch book of bestselling & widely acclaimed *Python ML* series. Foreword by Dmytro Dzhulgakov – PyTorch Core Maintainer

- Foreword. "Over recent years, ML methods, with their ability to make sense of vast amounts of data & automate decisions, have found widespread applications in healthcare, robotics, biology, physics, consumer products, internet services, & various other industries.

  Giant leaps in science usually come from a combination of powerful ideas & great tools. ML is no exception. Success of data-driven learning methods is based on ingenious ideas of thousands of talented researchers over field's 60-year history. But their recent popularity is also fueled by evolution of hardware & software solutions that make them scalable & accessible. Ecosystem of excellent libraries for numeric computing, data analysis, & ML built around Python like NumPy & scikit-learn gained wide adoption in research & industry. This has greatly helped propel Python to be most popular programming language.

  Massive improvements in computer vision, text, speech, & other tasks brought by recent advent of DL techniques exemplify this theme. Approaches draw on neural network theory of last 4 decades that started working remarkably well in combination with GPUs & highly optimized compute routines.

  Goal with building PyTorch over past 5 years has been to give researchers most flexible tool for expressing DL algorithms while taking care of underlying engineering complexities. benefited from excellent Python ecosystem. In turn, have been fortunate to see community of very talented people build advanced DL models across various domains on top of PyTorch. Authors of this book were among them.

  Dmytro Dzhulgakov has known Sebastian within this tight-knit community for a few years now. He has unmatched talent in easily explaining information & making complex accessible. Sebastian contributed to many widely used ML software packages & authored dozens of excellent tutorials on DL & data visualization.

  Mastery of both ideas & tools is also required to apply ML in practice. Getting started might feel intimidating, from making sense of theoretical concepts to figuring out which software packages to install.

  Luckily, this book does a beautiful job of combining ML concepts & practical engineering steps to guide you in this journey. You're in for a delightful ride from basics of data-driven techniques to most novel DL architectures. Within every chap, find concrete code examples applying introduced methods to a practical task.

  When 1e came out in 2015, it set a very high bar for *ML & Python* book category. But excellence didn't stop there. With every edition, Sebastian & team kept upgrading & refining material as DL revolution unfolded in new domains. In this new PyTorch edition, find new chaps on transformer architectures & graph neural networks. These approaches are on cutting edg of DL & have taken fields of text understanding & molecular structure by storm in last 2 years. You will get to practice them using new yet widely popular software packages in ecosystem like Hugging Face, PyTorch Lightning, & PyTorch Geometric.

  Excellent balance of theory & practice this book strikes is no surprise given authors' combination of advanced research expertise & experience in solving problems hands-on. Sebastian Raschka & Vahid Mirjalili draw from their background in DL research for computer vision & computational biology. Hayden Liu brings experience of applying ML methods to event prediction, recommendation systems, & other tasks in industry. All of authors share a deep passion for education, & it reflects in approachable way book goes from simple to advanced.

  Confident: find this book invaluable both as a broad overview of exciting field of ML & as a treasure of practical insights. Hope it inspires you to apply ML for greater good in your problem area, whatever it might be." Dmytro Dzhulgakov, PyTorch Core Maintainer

- Preface. Through exposure to news & social media, probably are familiar with fact: ML has become 1 of most exciting technologies of our time & age. Large companies, e.g. Microsoft, Google, Meta, Apple, Amazon, IBM, & many more, heavily invest in ML research & applications for good reasons. While it may seem: ML has become buzzword (từ thông dụng) of our time & age, certainly not hype (sự cường điệu). This exciting field opens way to new possibilities & has become indispensable to our daily lives. Talking to voice assistant on smartphones, recommending right product for our customers, preventing credit card fraud, filtering out spam from e-mail inboxes, detecting & diagnosing medical diseases, list goes on & on.

  If want to become a ML practitioner, a better problem solver, or even consider a career in ML research, then this book is for you! However, for a novice (cho người mới bắt đầu), theoretical concepts behind ML can be quite overwhelming. Yet, many practical books that have been published in recent years will help you get started in ML by implementing powerful learning algorithms.

  Getting exposed to practical code examples & working through example applications of ML is a great way to dive into this field. Concrete examples help to illustrate broader concepts by putting learned material directly into action. However, remember: with great power comes great responsibility! In addition to offering hands-on experience with ML using Python & Python-based ML libraries, this book also introduces mathematical concepts behind ML algorithms, which is essential for using ML successfully. Thus, this book is different from a purely practical book; it is a book that discuses necessary details regarding ML concepts, offers intuitive yet informative explanations on how ML algorithms work, how to use them, & most importantly, how to avoid most common pitfalls (cạm bẫy).

In this book, embark (tham gia) on exciting journey that covers all essential topics & concepts to give you a head start in this field. If find that your thirst for knowledge is not satisfied, this book references many useful resources that you can use to follow up on essential breakthroughs in this field.

- Who this book is for. This book is ideal companion for learning how to apply ML & DL to a wide range of tasks & datasets. If you are a programmer who wants to keep up with recent trends in technology, this book is definitely for you. Also, if you are a student or considering a career transition, this book will be both your introduction & a comprehensive guide to world of ML.
- What this book covers.
  - *Chap. 1: Giving Computers Ability to Learn from Data*, introduces to main subareas of ML to tackle various problem tasks. In addition, it discusses essential steps for creating a typical ML model building pipeline that will guide us through following chaps.
  - *Chap. 2: Training Simple ML Algorithms for Classification*, goes back to origins of ML & introduces binary perceptron classifiers & adaptive linear neurons. This chap is a gentle introduction to fundamentals of pattern classification & focuses on interplay of optimization algorithms & ML.
  - *Chap. 3: A Tour of ML Classifiers Using Scikit-Learn*, describes essential ML algorithms for classification & provides practical examples using 1 of most popular & comprehensive open-source ML libraries, `scikit-learn`.
  - *Chap. 4: Building Good Training Datasets – Data Preprocessing*, discusses how to deal with most common problems in unprocessed datasets, e.g. missing data. Also discuss several approaches to identify most informative features in datasets & teach how to prepare variables of different types as proper inputs for ML algorithms.
  - *Chap. 5: Compressing Data in Dimensionality Reduction*, describes essential techniques to reduce number of features in a dataset to smaller sets while retaining most of their useful & discriminatory information. Discuss standard approach to dimensionality reduction via principal component analysis & compare it to supervised & nonlinear transformation techniques.
  - *Chap. 6: Learning Best Practices for Model Evaluation & Hyperparameter Tuning*, discusses the do's & don'ts for estimating performances of predictive models. Moreover, discuss different metrics for measuring performance of our models & techniques to fine-tune ML algorithms.
  - *Chap. 7: Combining Different Models for Ensemble Learning*, introduces to different concepts of combining multiple learning algorithms effectively. Teach how to build ensembles of experts (làm thế nào để xây dựng các nhóm chuyên gia) to overcome weaknesses of individual learners, resulting in more accurate & reliable predictions.
  - *Chap. 8: Applying ML to Sentiment Analysis*, discusses essential steps to transform textual data into meaningful representations for ML algorithms to predict opinions of people based on their writing.
  - *Chap. 9: Predicting Continuous Target Variables with Regression Analysis*, discusses essential techniques for modeling linear relationships between target & response variables to make predictions on a continuous scale. After introducing different linear models, also talks about polynomial regression & tree-based approaches.
  - *Chap. 10: Working with Unlabeled Data – Clustering Analysis*, shifts focus to a different subarea of ML, unsupervised learning. Apply algorithms from 3 fundamental families of clustering algorithms to find groups of objects that share a certain degree of similarity.
  - *Chap. 11: Implementing a Multilayer Artificial Neural Network from Scratch*, extends concept of gradient-based optimization, which 1st introduced in Chap. 2, to build powerful, multilayer neural networks based on popular backpropagation algorithm in Python.
  - *Chap. 12: Parallelizing Neural Network Training with PyTorch*, builds upon knowledge from previous chap to provide you with a practical guide for training neural networks more efficiently. Focus of this chap is on PyTorch, an open-source Python library that allows us to utilize multiple cores of modern GPUs & construct deep neural networks from common building blocks via a user-friendly & flexible API.
  - *Chap. 13: Going Deeper – Mechanics of PyTorch*, picks up where previous chap left off & introduces more advanced concepts & functionality of PyTorch. PyTorch is an extraordinary vast & sophisticated library, & this chap walks you through concepts e.g. dynamic computation graphs & automatic differentiation. Also learn how to use PyTorch's object-oriented API to implement complex neural networks & how PyTorch Lightning helps with best practices & minimizing boilerplate code (giảm thiểu mã mẫu).
  - *Chap. 14: Classifying Images with Deep Convolutional Neural Networks*, introduces *convolutional neural networks (CNNs)*. A CNN represents a particular type of deep neural network architecture that is particularly well-suited for working with image datasets. Due to their superior performance compared to traditional approaches, CNNs are now widely used in computer vision to achieve state-of-art results for various image recognition tasks. Throughout this chap, learn how convolutional layers can be used as powerful feature extractors for image classification.
  - *Chap. 15: Modeling Sequential Data Using Recurrent Neural Networks*, introduces another popular neural network architectures for DL that is especially well suited for working with text & other types of sequential data & time series data. As a warm-up exercises, this chap introduces recurrent neural networks for predicting sentiment of movie reviews. Then, teach recurrent networks to digest information from books in order to generate entirely new text.
  - *Chap. 16: Transformers – Improving Natural Language Processing with Attention Mechanisms*, focuses on latest trends in natural language processing & explains how attention mechanisms help with modeling complex relationships in long

sequences. In particular, this chap describes influential transformer architecture & state-of-art transformer models e.g. BERT & GPT.

  - *Chap. 17: Generative Adversarial Networks for Synthesizing New Data*, introduces a popular adversarial training regime for neural networks that can be used to generate new, realistic-looking images. Chap starts with a brief introduction to autoencoders, which is a particular type of neural network architecture that can be used for data compression. Chap then shows how to combine decoder part of an autoencoder with a 2nd neural network that can distinguish between real & synthesized images. By letting 2 neural networks compete with each other in an adversarial training approach, will implement a generative adversarial network that generates new handwritten digits.
  - *Chap. 18: Graph Neural Networks for Capturing Dependencies in Graph Structured Data*, goes beyond working with tabular datasets, images, & text. Introduces graph neural networks that operate on graph-structured data, e.g. social media networks & molecules. After explaining fundamentals of graph convolutions, this chap includes a tutorial showing how to implement predictive models for molecular data.
  - *Chap. 19: Reinforcement Learning for Decision Making in Complex Environments*, covers a subcategory of ML that is commonly used for training robots & other autonomous systems. Chap starts by introducing basics of *reinforcement learning (RL)* to become familiar with agent/environment interactions, reward process of RL systems, & concept of learning from experience. After learning about main categories of RL, will implement & train an agent that can navigate in a grid world environment using Q-learning algorithm. Finally, this chap introduces deep Q-learning algorithm – a variant of Q-learning that uses deep neural networks.

- To get most out of book. Ideally, already comfortable with programming in Python to follow along with code examples provided to both illustrate & apply various algorithms & models. To get most out of this book, a firm grasp of mathematical notation will be helpful as well.

  A common laptop or desktop computer should be sufficient for running most of code in this book, & provide instructions for your Python environment in 1st chap. Later chaps will introduce additional libraries & installation recommendations when need arises.

  A recent *graphical processing unit (GPU)* can accelerate code runtimes in later DL chaps. However, a GPU is not required, & also provide instructions for using free cloud resources.

- Download example code files. All code examples are available for download through https://github.com/rasbt/machine-learning Also have other code bundles from rich catalog of books & videos available at https://github.com/PacktPublishing/.

  While recommend using Jupyter Notebook for executing code interactively, all code examples are available in both a Python script (e.g., `ch02/ch02.py`) & a Jupyter Notebook format (e.g., `ch02/ch02.ipynb`). Furthermore, recommend viewing `README.md` file that accompanies each individual chap for additional information & updates.

- Chap. 1: Giving Computers Ability to Learn from Data. ML, application & science of algorithms that make sense of data, is most exciting field of all computer sciences! Living in an age where data comes in abundance; using self-learning algorithms from field of ML, can turn this data into knowledge. Thanks to many powerful open-source libraries that have been developed in recent years, there has probably never been a better time to break into ML field & learn how to utilize powerful algorithms to spot patterns in data & make predictions about future events.

  In this chap, learn about main concepts & different types of ML. Together with a basic introduction to relevant terminology, lay groundwork for successfully using ML technique for practical problem solving.

  In this chap, cover topics:

  - General concepts of ML
  - 3 types of learning & basic terminology
  - Building blocks for successfully designing ML systems
  - Installing & setting up Python for data analysis & ML

  - Building intelligence machines to transform data into knowledge. In this age of modern technology, there is 1 resource that we have in abundance: a large amount of structured & unstructured data. In 2nd half of 20th century, ML evolved as a subfield of AI involving self-learning algorithms that derive knowledge from data to make predictions.

    Instead of requiring humans to manually derive rules & build models from analyzing large amounts of data, ML offers a more efficient alternative for capturing knowledge in data to gradually improve performance of predictive models & make data-driven decisions.

    Not only is ML becoming increasingly important in CS research, but it is also playing an ever-greater role in our everyday lives. Thanks to ML, enjoy robust email spam filters, convenient text & voice recognition software, reliable web search engines, recommendations on entertaining movies to watch, mobile check deposits, estimated meal delivery times, & much more. Hopefully, soon, will add safe & efficient self-driving cars to this list. Notable progress has been made in medical applications; e.g., researchers demonstrated: DL models can detect skin cancer with near-human accuracy https://www.nature.com/articles/nature21056. Another milestone was recently achieved by researchers at DeepMind, who used DL to predict 3D protein structures, outperforming physics-based approaches by substantial margin https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology. While accurate 3D protein structure prediction plays an essential role in biological & pharmaceutical research, there have been many

other important applications of ML in healthcare recently. E.g., researchers designed systems for predicting oxygen needs of COVID-19 patients up to 4 days in advance to help hospitals allocate resources for those in need https://ai.facebook.com/blog/new-ai-research-to-help-predict-covid-19-resource-needs-from-a-series-of-x-rays/. Another important topic of our day & age is climate change, which presents 1 of biggest & most critical challenges. Today, many efforts are being directed toward developing intelligent systems to combat it https://www.forbes.com/sites/robtoews/2021/06/20/these-are-the-startups-applying-ai-to-tackle-climate-change. 1 of many approaches to tackling climate change is emergent field of precision agriculture. Researchers aim to design computer vision-based ML systems to optimize resource deployment to minimize use & waste of fertilizers.

- 3 different types of ML. Take a look at 3 types of ML: *supervised learning, unsupervised learning, & reinforcement learning*. Learn about fundamental differences between 3 different learning types &, using conceptual examples, develop an understanding of practical problem domains where they can be applied: Fig. 1.1: 3 different types of ML.

  1. Supervised Learning: labeled data, direct feedback, predict outcome/future
  2. Unsupervised learning: no labels/targets, no feedback, find hidden structure in data
  3. Reinforcement learning: decision process, reward system, learn series of actions

  * Making predictions about future with supervised learning. Main goal in supervised learning: learn a model from labeled training data that allows us to make predictions about unseen or future data. Term "supervised" refers to a set of training examples (data inputs) where desired output signals (labels) are already known. Supervised learning is then process of modeling relationship between data inputs & labels. Can also think of supervised learning as "label learning".
    Fig. 1.2: Supervised learning process summarizes a typical supervised learning workflow, where labeled training data is passed to a ML algorithm for fitting a predictive model that can make predictions on new, unlabeled data inputs.
    Considering example of email spam filtering, can train a model using a supervised ML algorithm on a corpus (ngữ liệu) of labeled emails, which are correctly marked as spam or non-spam, to predict whether a new email belongs to either of 2 categories. A supervised learning task with discrete class labels, e.g. in previous email spam filtering example, is also called a *classification task*. Another subcategory of supervised learning is *regression*, where outcome signal is a continuous value.

    · Classification for predicting class labels. Classification is a subcategory of supervised learning where goal: predict categorical class labels of new instances or data points based on past observations. Those class labels are discrete, unordered values that can be understood as group memberships of data points. Previously mentioned example of email spam detection represents a typical example of a binary classification task, where ML algorithm learns a set of rules to distinguish between 2 possible classes: spam & non-spam emails.
      Fig. 1.3: Classifying a new data point illustrates concept of a binary classification task given 30 training examples; 15 training examples are labeled as class A & 15 training examples are labeled as class B. In this scenario, our dataset is 2D, i.e., each example has 2 values associated with it: x_1, x_2. Can use a supervised ML algorithm to learn a rule – decision boundary represented as a dashed line – that can separate those 2 classes & classify new data into each those 2 categories given its x_1, x_2 values.
      However, set of class labels does not have to be of a binary nature. Predictive model learned by a supervised learning algorithm can assign any class label that was presented in training dataset to a new, unlabeled data point or instance. A typical example of a *multiclass classification* task is handwritten character recognition. Can collect a training dataset that consists of multiple handwritten examples of each letter in alphabet. Letters ("A", "B", "C", ...) will represent different unordered categories or class labels that we want to predict. Now, if a user provides a new handwritten character via an input device, our predictive model will be able to predict correct letter in alphabet with certain accuracy. However, our ML system will be unable to correctly recognize any of digits between 0 & 9, e.g., if they were not part of training dataset.

    · Regression for predicting continuous outcomes. Learned in prev sec: task of classification: to assign categorical, unordered labels to instances. A 2nd type of supervised learning: prediction of continuous outcomes, also called *regression analysis*. In regression analysis, given a number of predictor (*explanatory*) variables & a continuous response variable (*outcome*), & try to find a relationship between those variables that allows us to predict an outcome.
      Note: in field of ML, predictor variables are commonly called "features", & response variables are usually referred to as "target variables". Adopt these conventions throughout this book.
      E.g., assume: interested in predicting math SAT scores of students. (SAT is a standardized test frequently used for college admissions in US.) If there is a relationship between time spent studying for test & final scores, could use it as training data to learn a model that uses study time to predict test scores of future students who are planning to take this test.

      Remark 19 (Regression toward mean)**.** *Term "regression" was devised by* Francis Galton *in his article* Regression towards Mediocrity in Hereditary Stature *in 1886.* Galton *described biological phenomenon that variance of height in a population does not increase over time.*
      Observed: height of parents is not passed on to their children, but instead, their children's height regresses toward population mean.

      Fig. 1.4: A linear regression example illustrates concept of linear regression. Given a feature variable x & a target variable y, fit a straight line to this data that minimizes distance – most commonly averaged squared distance – between data points & fitted line. Can now use intercept & slope learned from this data to predict target variable of new data. – Bây giờ có thể sử dụng giá trị chặn & độ dốc học được từ dữ liệu này để dự đoán biến mục tiêu của dữ liệu mới.

* Solving interactive problems with reinforcement learning. Another type of ML is *reinforcement learning*. In reinforcement learning, goal: develop a system (*agent*) that improves its performance based on interactions with environment. Since information about current state of environment typically also includes a so-called *reward signal*, can think of reinforcement learning as a field related to supervised learning. However, in reinforcement learning, this feedback is not correct ground truth label or value, but a measure of how well action was measured by a reward function. Through its interaction with environment, an agent can then use reinforcement learning to learn a series of actions that maximizes this reward via an exploratory trial-&-error approach or deliberative planning.
  A popular example of reinforcement learning is a chess program. Here, agent decides upon a series of moves depending on state of board (environment), & reward can be defined as *win* or *lose* at end of game: Fig. 1.5: Reinforcement learning process.
  There are many different subtypes of reinforcement learning. However, a general scheme: agent in reinforcement learning tries to maximize reward through a series of interactions with environment. Each state can be associated with a positive or negative reward, & a reward can be defined as accomplishing an overall goal, e.g. winning or losing a game of chess. E.g., in chess, outcome of each move can be thought of as a different state of environment.
  To explore chess example further, think of visiting certain configuration on chessboard as being associated with states that will more likely lead to winning – e.g., removing an opponent's chess piece from board or threatening queen. Other positions, however, are associated with states that will more likely result in losing game, e.g. losing a chess piece to opponent in following turn. Now, in game of chess, reward (either positive for winning or negative for losing game) will not be given until end of game. In addition, final reward will also depend on how opponent plays. E.g., opponent may sacrifice queen but eventually win game.
  In sum, reinforcement learning is concerned with learning to choose a series of actions that maximizes total reward, which could be earned either immediately after taking an action or via *delayed* feedback.
* Discovering hidden structures with unsupervised learning. In supervised learning, know right answer (label or target variable) beforehand when train a model, & in reinforcement learning, define a measure of reward for particular actions carried out by agent. In unsupervised learning, however, dealing with unlabeled data or data of an unknown structure. Using unsupervised learning techniques, able to explore structure of our data to extract meaningful information without guidance of a known outcome variable or reward function.
  - Finding subgroups with clustering. *Clustering* is an exploratory data analysis or pattern discovery technique that allows us to organize a pile of information into meaningful subgroups (*clusters*) without having any prior knowledge of their memberships. Each cluster that arises during analysis defines a group of objects that share a certain degree of similarity but are more dissimilar to objects in other clusters, which is why clustering is also sometimes called *unsupervised classification*. Clustering is a great technique for structuring information & deriving meaningful relationships from data. E.g., it allows marketers to discover customer groups based on their interests, in order to develop distinct marketing programs.
    Fig. 1.6: How clustering works illustrates how clustering can be applied to organizing unlabeled data into 3 distinct groups or clusters (A, B, & C, in arbitrary order) based on similarity of their features, x_1, x_2.
  - Dimensionality reduction for data compression. Another subfield of unsupervised learning is *dimensionality reduction*. Often, working with data of high dimensionality – each observation comes with a high number of measurements – that can present a challenge for limited storage space & computational performance of ML algorithms. Unsupervised dimensionality reduction is a commonly used approach in feature preprocessing to remove noise from data, which can degrade predictive performance of certain algorithms. Dimensionality reduction compresses data onto a smaller dimensional subspace while retaining most of relevant information.
    Sometimes, dimensionality reduction can also be useful for visualizing data; e.g., a high-dimensional feature set can be projected onto 1D, 2D, or 3D feature spaces to visualize it via 2D or 3D scatterplots or histograms. Fig. 1.7: An example of dimensionality reduction from 3D to 2D. shows an example where nonlinear dimensionality reduction was applied to compress a 3D Swiss roll onto a new 2D feature subspace.

o Introduction to basic terminology & notations. Discussed 3 broad categories of ML – supervised, unsupervised, & reinforcement learning – have a look at basic terminology that we will be using throughout this book. Following subsect covers common terms we will be using when referring to different aspects of a dataset, as well as mathematical notation to communicate more precisely & efficiently.
  As ML is a vast field & very interdisciplinary, guaranteed to encounter many different terms that refer to same concepts sooner rather than later. 2nd subsect collects many of most commonly used terms that are found in ML literature, which may be useful as a ref sect when reading ML publications.

* Notation & conventions used in this book. Fig. 1.8: Iris dataset depicts an excerpt of Iris dataset, which is a classic example in field of ML https://archive.ics.uci.edu/ml/datasets/iris. Iris dataset contains measurements of 150 Iris flowers from 3 different species – Setosa, Versicolor, & Virginica.
  Here, each flower example represents 1 row in our dataset, & flower measurements in centimeters are stored as columns, which also called *features* of dataset.
  To keep notation & implementation simple yet efficient, make use of some of basics of linear algebra. In following chaps, use a matrix notation to refer to our data. Follow common convention to represent each example as a separate row in a feature matrix, X, where each feature is stored as a separate column.
  Iris dataset, consisting of 150 examples & 4 features, can then be written as a 150×4 matrix, formally denoted as $\mathbf{X} \in \mathbb{R}^{150 \times 4}$.

Remark 20 (Notational conventions)**.** *For most parts of this book, unless noted otherwise, use superscript i to refer to ith training example, & subscript j to refer to jth dimension of training dataset.*
Use lowercase, bold-face letters to refer to vectors $\mathbf{x} \in \mathbb{R}^{n\times 1}$ & suppercase, bold-face letters to refer to matrices $\mathbf{X} \in \mathbb{R}^{n\times m}$. To refer to single elements in a vector or matrix, write letters in italics $x^{(n)}$ or $x_m^{(n)}$, resp.
E.g., $x_1^{(150)}$ refers to 1st dimension of flow example 150, sepal length (chiều dài lá đài). Each row in matrix $\mathbf{X}$ represents 1 flower instance & can be written as a 4-dimensional row vector $\mathbf{x}^{(i)} \in \mathbb{R}^{1\times 4}$: $\mathbf{X}^{(i)} = [x_1^{(i)}\ x_2^{(i)}\ x_3^{(i)}\ x_4^{(i)}]$. & each feature dimension is a 150-dimensional column vector $\mathbf{X}^{(i)} \in \mathbb{R}^{150\times 1}$, e.g.:

$$\mathbf{x}_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(150)} \end{bmatrix} \tag{187}$$

Similarly, can represent target variables (here, class labels) as a 150-dimensional column vector:

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(150)} \end{bmatrix} \quad \textit{where } y^{(i)} \in \{\textit{Setosa, Versicolor, Virginica}\} \tag{188}$$

- ML terminology. ML is a vast field & also very interdisciplinary as it brings together many scientists from other areas of research. As it happens, many terms & concepts have been rediscovered or redefined & may already be familiar to you but appear under different names. For your convenience, in following list, can find a selection of commonly used terms & their synonyms that you may find useful when reading this book & ML literature in general:
  - *Training example*: A row in a table representing dataset & synonymous with an observation, record, instance, or sample (in most contexts, sample refers to a collection of training examples).
  - *Training*: Model fitting, for parametric models similar to parameter estimation.
  - *Feature, abbr. x*: A column in a data table or data (design) matrix. Synonymous with predictor, variable, input, attribute, or covariate.
  - *Target, abbr. y*: Synonymous with outcome, output, response variable, dependent variable, (class) label, & ground truth.
  - *Loss function*: Often used synonymously with a *cost* function. Sometimes loss function is also called an *error* function. In some literature, term "loss" refers to loss measured for a single data point, & cost is a measurement that computes loss (averaged or summed) over entire dataset.

- A roadmap for building ML systems. In prev sects, discussed basic concepts of ML & 3 different types of learning. In this sect, discuss other important parts of a ML system accompanying learning algorithm.
  Fig. 1.9: Predictive modeling workflow shows a typical workflow for using ML in predictive modeling, which will discuss in following subsects.

  - Preprocessing – getting data into shape. Begin by discussing roadmap for building ML systems. Raw data rarely comes in form & shape that is necessary for optimal performance of a learning algorithm. Thus, preprocessing of data is 1 of most crucial steps in any ML application.
    If take Iris flower dataset from previous sect as an example, can think of raw data as a series of flower images from which want to extract meaningful features. Useful features could be centered around color of flowers or height, length, & width of flowers.
    Many ML algorithms also require: selected features are on same scale for optimal performance, which is often achieved by transforming features in range $[0, 1]$ or a standard normal distribution with zero mean & unit variance.
    Some of selected features may be highly correlated & therefore redundant to a certain degree. In those cases, dimensionality reduction techniques are useful for compressing features onto a lower-dimensional subspace. Reducing dimensionality of our feature space has advantage that less storage space is required, & learning algorithm can run much faster. In certain cases, dimensionality reduction can also improve predictive performance of a model if dataset contains a large number of irrelevant features (or noise); i.e., if dataset has a low signal-to-noise ratio.
    To determine whether our ML algorithm not only performs well on training dataset but also generalizes well to new data, also want to randomly divide dataset into separate training & test datasets. Use training dataset to train & optimize our ML model, while keep test dataset until very end to evaluate final model.
  - Training & selecting a predictive model. Many different ML algorithms have been developed to solve different problem tasks. An important point that can be summarized from David Wolpert's famous *No free lunch theorems*: we can't get learning "for free" (*Lack of A Priori Distinctions Between Learning Algorithms*, D.H. Wolpert, 1996; *No free lunch theorems for optimization*, D.H. Wolpert & W.G. Macready, 1997). Can relate this concept to popular saying, *I suppose it is tempting, if only tool you have is a hammer, to treat everything as if it were a nail* (Tôi cho rằng thật hấp dẫn nếu công cụ duy nhất bạn có là một cái búa, để coi mọi thứ như thể chúng là một cái đinh.) (Abraham Maslow, 1966). E.g., each classification algorithm has its inherent biases, & no single classification model enjoys superiority if we don't make any assumptions about task. In practice, therefore essential to compare at least a handful of different learning algorithms in order to train & select best performing model. But before can compare different models, 1st have to decide upon a metric

to measure performance. 1 commonly used metric is classification accuracy, which is defined as proportion of correctly classified instances.

1 legitimate question to ask is this: how do we know which model performs well on final test dataset & real-world data if we don't use this test dataset for model selection, but keep it for final model evaluation? To address issue embedded in this question, different techniques summarized as "cross-validation" can be used. In cross-validation, further divide a dataset into training & validation subsets in order to estimate generalization performance of model.

Finally, also cannot expect: default parameters of different learning algorithms provided by software libraries are optimal for our specific problem task. Therefore, will make frequent use of hyperparameter optimization techniques that help us to fine-tune performance of our model in later chaps.

Can think of those hyperparameters as parameters that are not learned from data but represent knobs (núm vặn) of a model that we can turn to improve its performance. This will become much clearer in later chaps when see actual examples.

* Evaluating models & predicting unseen data instances. After have selected a model that has been fitted on training dataset, can use test dataset to estimate how well it performs on this unseen data to estimate so-called *generalization error*. If satisfied with its performance, can now use this model to predict new, future data. Important to note: parameters for previously mentioned procedures, e.g. feature scaling & dimensionality reduction, are solely obtained from training dataset, & same parameters are later reapplied to transform test dataset, as well as any new data instances – performance measured on test data may be overly optimistic otherwise.

- Using Python for ML. Python is 1 of most popular programming languages for DS, & thanks to its very active developer & open-source community, a large number of useful libraries for scientific computing & ML have been developed.

  Although performance of interpreted languages, e.g. Python, for computation-intensive tasks is inferior to lower-level programming languages, extension libraries e.g. NumPy & SciPy have been developed that build upon lower-layer Fortran & C implementations for fast vectorized operations on multidimensional arrays.

  For ML programming tasks, mostly refer to `scikit-learn` library, which is currently 1 of most popular & accessible open-source ML libraries. In later chaps, when focus on a subfield of ML called *DL*, use latest version of PyTorch library, which specializes in training so-called *deep neural network* models very efficiently by utilizing graphics cards.

  * Installing Python & package from Python Package Index. Python is available for all 3 major OS – Microsoft Windows, macOS, & Linux – & installer, as well as documentation, can be downloaded from official Python website: https://www.python.org.

    Code examples provided in this book have been written for & tested in Python 3.9, generally recommend: use most recent version of Python 3 that is available. Some of code may also be compatible with Python .7, but as official support for Python 2.7 ended in 2019, & majority of open-source libraries have already stopped supporting Python 2.7 https://python3statement.org, strongly advise use Python 3.9 or newer. Can check Python version by executing `python -version` or `python3 -version` in terminal (or PowerShell if using Windows).

    Additional packages that we will be using throughout this book can be installed via `pip` installer program, which has been part of Python Standard Library since Python 3.3. More information about `pip`: https://docs.python.org/3/installing/index.html.

    After have successfully installed Python, can execute `pip` from terminal to install additional Python packages: `pip install SomePackage`. Already installed packages can be updated via `-upgrade` flag:

    ```
    pip install SomePackage --upgrade
    ```

  * Using Anaconda Python distribution & package manager. A highly recommended open-source package management system for installing Python for scientific computing contexts is `conda` by Continuum Analytics. Conda is free & licensed under a permissive open-source license. Its goal: help with installation & version management of Python packages for DS, math, & engineering across different OSs. If want to use conda, it comes in different flavors, namely Anaconda, Miniconda, & Miniforge:

    · Anaconda comes with many scientific computing packages pre-installed. Anaconda installer can be downloaded at https://docs.anaconda.com/anaconda/install/, & an Anaconda quick start guide is available at https://docs.anaconda.com/anaconda/user-guide/getting-started/.

    · Miniconda is a leaner alternative (thay thế gầy hơn) to Anaconda https://docs.conda.io/en/latest/miniconda.html. Essentially, similar to Anaconda but without any packages pre-installed, which many people (including authors) prefer.

    · Miniforge is similar to Miniconda but community-maintained & uses a different package repository `conda-forge` from Miniconda & Anaconda. Found: Miniforge is a great alternative to Miniconda. Download & installation instructions can be found in GitHub repository at https://github.com/conda-forge/miniforge.

    After successfully installing `conda` through either Anaconda, Miniconda, or Miniforge, can install new Python packages using command:

    ```
    conda install SomePackage
    ```

    Existing packages can be updated using command:

    ```
    conda update SomePackage
    ```

Packages that are not available through official conda channel might be available via community-supported `conda-forge` project https://conda-forge.org, which can be specified via `-channel conda-forge` flag. E.g.:

```
conda install SomePackage --channel conda-forge
```

Packages that are not available through default conda channel or conda-force can be installed via `pip`. E.g.:

```
pip install SomePackage
```

* Packages for scientific computing, DS, & ML. Throughout 1st half of this book, mainly use NumPy's multidimensional arrays to store & manipulate data. Occasionally, make use of `pandas`, which is a library built on top of NumPy that provides additional higher-level data manipulation tools that make working with tabular data even more convenient. To augment your learning experience & visualize quantitative data, which is often extremely useful to make sense of it, will use very customizable Matplotlib library.
Main ML library used in this book is `scikit-learn` (Chaps. 3–11). *Chap. 12: Parallelizing Neural Network Training with PyTorch*, will then introduce PyTorch library for DL.
Version numbers of major Python packages that were used to write this book are mentioned in following list. Make sure: version numbers of your installed packages are, ideally, equal to these version numbers to ensure code examples run correctly:
  · NumPy 1.21.2
  · SciPy 1.7.0
  · Scikit-learn 1.0
  · Matplotlib 3.4.3
  · pandas 1.3.2

  After installing these packages, can double-check installed version by importing package in Python & accessing its `__version__` attribute, e.g.:

```
>>> import numpy
>>> numpy.__version__
'1.26.4'
```

  For your convenience, included a `python-environment-check.py` script in this book's complimentary code repository at https://github.com/rasbt/machine-learning-book so that you can check both Python version & package versions by executing this script.

```
(base) nqbh@nqbh-dell:~/advanced_STEM_beyond/machine_learning/Python$ python python_environment_check.
/home/nqbh/advanced_STEM_beyond/machine_learning/Python/python_environment_check.py:4: DeprecationWarn
if LooseVersion(sys.version) < LooseVersion('3.8'):
[OK] Your Python version is 3.12.7 | packaged by Anaconda, Inc. | (main, Oct  4 2024, 13:27:36) [GCC 1
/home/nqbh/advanced_STEM_beyond/machine_learning/Python/python_environment_check.py:39: DeprecationWar
actual_ver, suggested_ver = LooseVersion(actual_ver), LooseVersion(suggested_ver)
[OK] numpy 1.26.4
[OK] scipy 1.13.1
/home/nqbh/advanced_STEM_beyond/machine_learning/Python/python_environment_check.py:40: DeprecationWar
if pkg_name == "matplotlib" \& actual_ver == LooseVersion("3.8"):
[OK] matplotlib 3.9.2
[OK] sklearn 1.5.1
[OK] pandas 2.2.2
```

  Certain chaps will require additional packages & will provide information about installations. E.g., do not worry about installing PyTorch at this point. Chap. 12 will provide tips & instructions when need them.
  If encounter errors even though your code matches code in chap exactly, recommend 1st check version numbers of underlying packages before spending more time on debugging or reaching out to publisher or authors. Sometimes, newer versions of libraries introduce backward-incompatible changes that could explain these errors.
  if do not want to change package version in your main Python installation, recommend using a virtual environment for installing packages used in this book. If use Python without `conda` manager, can use `venv` library to create a new virtual environment. E.g., can create & activate virtual environment via 2 commands:

```
python3 -m venv /Users/sebastian/Desktop/pyml-book
source /Users/sebastian/Desktop/pyml-book/bin/activate
```

  Note: need to activate virtual environment every time you open a new terminal or PowerShell. More information about `venv` at https://docs.python.org/3/library/venv.html.
  If using Anaconda with `conda` package manager, can create & activate a virtual environment as follows:

```
conda create -n pyml python=3.9
conda activate pyml
```

- Summary. In this chap, explored ML at a very high level & familiarized with big picture & major concepts that going to explore in following chaps in more detail. Learned: supervised learning is composed of 2 important subfields: classification & regression. While classification models allow to categorize objects into known classes, can use regression analysis to predict continuous outcomes of target variables. Unsupervised learning not only offers useful techniques for discovering structures in unlabeled data, but it can also be useful for data compression in feature preprocessing steps.

  Briefly went over typical roadmap for applying ML to problem tasks, which will use as a foundation for deeper discussions & hands-on examples in following chaps. Finally, set up our Python environment & installed & updated required packages to get ready to see ML in action.

  Later in this book, in addition to ML itself, introduce different techniques to preprocess a dataset, which will help you to get best performance out of different ML algorithms. While cover classification algorithms quite extensively throughout book, also explore different techniques for regression analysis & clustering.

  Have an exciting journey ahead, covering many powerful techniques in vast field of ML. However, approach ML 1 step at a time, building upon our knowledge gradually throughout chaps of this book. In following chap, start this journey by implementing 1 of earliest ML algorithms for classification, which will prepare for *Chap. 3: A Tour of ML Classifiers Using Scikit-Learn*, where cover more advanced ML algorithms using `scikit-learn` open-source ML library.

- Chap. 2: Training Simple ML Algorithms for Classification. In this chap, make use of 2 of 1st algorithmically described ML algorithms for classification: perceptron & adaptive linear neurons. Start by implementing a perceptron step by step in Python & training it to classify different flower species in Iris dataset. This will help us to understand concept of ML algorithms for classification & how they can be efficiently implemented in Python.

  Discussing basics of optimization using adaptive linear neurons will then lay groundwork for using more sophisticated classifiers via scikit-learn ML library in *Chap. 3: A Tour of ML Classifiers Using Scikit-Learn*.

  Topics covered in this chap:

  - Building an understanding of ML algorithms
  - Using pandas, NumPy, & Matplotlib to read in, process, & visualize data
  - Implementing linear classifiers for 2-class problems in Python

  - Artificial neurons – a brief glimpse into early history of ML. Before discuss perceptron & related algorithms in more detail, take a brief tour of beginnings of ML. Trying to understand how biological brain works in order to design an AI, Warren McCulloch & Walter Pitts published 1st concept of a simplified brain cell, so-called *McCullock-Pitts (MCP)* neuron, in 1943 (*A Logical Calculus of Ideas Immanent in Nervous Activity* by W.S. McCulloch & W. Piits, Bulletin of Mathematical Biophysics, 5(4): 115-133, 1943).

    Biological neurons are interconnected nerve cells in brain that are involved in processing & transmitting of chemical & electrical signals, illustrated in Fig. 2.1: A neuron processing chemical & electrical signals.

    McCulloch & Pitts described such a nerve cell as a simple logic gate with binary outputs; multiple signal arrive at dendrites (các nhánh cây, a short branch at the end of a nerve cell that receives signals from other cells – 1 nhánh ngắn ở cuối tế bào thần kinh nhận tín hiệu từ các tế bào khác), they are then integrated into cell body, &, if accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by axon (sợi trục).

    Only a few years later, Frank Rosenblatt published 1st concept of perceptron learning rule based on MCP neuron model (*The Perceptron: A Perceiving & Recognizing Automaton* by F. Rosenblatt, Cornell Aeronautical Laboratory, 1957). With his perceptron rule, Rosenblatt proposed an algorithm that would automatically learn optimal weight coefficients that would then be multiplied with input features in order to make decision of whether a neuron fires (transmits a signal) or not. In context of supervised learning & classification, e.g. algorithm could then be used to predict whether a new data point belongs to 1 class or the other.

    * Formal def of an artificial neuron. More formally, can put idea behind *artificial neurons* into context of a binary classification task with 2 classes: 0 & 1. Can then define a decision function $\sigma(z)$ that takes a linear combination of certain input values x & a corresponding weight vector w where z: so-called *net input* $z = \mathbf{w} \cdot \mathbf{x} = w_1 x_1 + \cdots + w_m x_m$. Now, if net input of a particular example $x^{(i)}$ is greater than a defined threshold θ, predict class 1, & class 0 otherwise. In perceptron algorithm, decision function $\sigma(\cdot)$ is a variant of a *unit step function*:

      $$\sigma(z) = \begin{cases} 1 & \text{if } z \geq \theta, \\ 0 & \text{otherwise.} \end{cases} \tag{189}$$

      To simplify code implementation later, can modify this setup via a couple of steps. 1st, move threshold θ to left side of equation: $z \geq \theta, z - \theta \geq 0$. 2nd, define a so-called *bias unit* as $b = -\theta$ & make it part of net input: $z = w_1 x_1 + \cdots + w_m x_m + b = \mathbf{w}^\top \mathbf{x} + b$. 3rd, given introduction of bias unit & redefinition of net input z above, redefine decision function as follows:

      $$\sigma(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{190}$$

      Fig. 2.2: A threshold function producing a linear decision boundary for a binary classification problem illustrates how net input $z = \mathbf{w}^\top \mathbf{x} + b$ is squashed into a binary output (0 or 1) by decision function of perceptron & how it can be used to discriminate between 2 classes separable by a linear decision boundary.

* Perceptron learning rule. Whole idea behind MCP neuron & Rosenblatt's *thresholded* perceptron model: use a reductionist approach to mimic how a single neuron in brain works: it either *fires* or it doesn't. Thus, Rosenblatt's classic perceptron rule is fairly simple, & perceptron algorithm can be summarized by following steps:
  1. Initialize weights & bias unit to 0 or small random numbers
  2. For each training example, $x^{(i)}$:
     (a) Compute output value $\hat{y}^{(i)}$
     (b) Update weights & bias unit

  Here, output value is class label predicted by unit step function defined earlier, & simultaneous update of bias unit & each weight w_j in weight vector $\mathbf{w}$ can be more formally written as

$$w_i = w_i + \Delta w_i, \tag{191}$$
$$b = b + \Delta b. \tag{192}$$

  Update values ("deltas") are computed as follows:

$$\Delta w_j = \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}, \tag{193}$$
$$\Delta b = \eta(y^{(i)} - \hat{y}^{(i)}). \tag{194}$$

  Note: unlike bias unit, each weight w_j corresponds to a feature x_j in dataset, which is involved in determining update value Δw_j defined above. Furthermore, η is *learning rate* (typically a constant between 0.0 & 1.0), $y^{(i)}$: *true class label* of ith training example, & $\hat{y}^{(i)}$: *predicted class label*. Important to note: bias unit & all weights in weight vector are being updated simultaneously, i.e., don't recompute predicted labels $\hat{y}^{(i)}$ before bias unit & all of weights are updated via respective update values Δw_j & Δb. Concretely, for a 2D dataset, write update as:

$$\Delta w_1 = \eta(y^{(i)} - \text{output}^{(i)})x_1^{(i)}, \tag{195}$$
$$\Delta w_2 = \eta(y^{(i)} - \text{output}^{(i)})x_2^{(i)}, \tag{196}$$
$$\Delta b = \eta(y^{(i)} - \text{output}^{(i)}). \tag{197}$$

  Before implement perceptron rule in Python, go through a simple thought experiment to illustrate how beautifully simple this learning rule really is. In 2 scenarios where perceptron predicts class label correctly, bias unit & weights remain unchanged, since update values are 0:

$$\cdots y^{(i)} = \hat{y}^{(i)} \Leftrightarrow \cdots \Leftrightarrow \Delta w_j = \Delta b = 0. \tag{198}$$

  However, in case of a wrong prediction, weights are being pushed toward direction of positive or negative target class $\Delta b = \pm\eta$.

  To get a better understanding of feature value as a multiplicative factor $x_j^{(i)}$, go through another simple example where $y^{(i)} = 1, \hat{y}^{(i)} = 0, \eta = 1$. Assume $x_j^{(i)} = 1.5$ misclassified as class 0: would increase corresponding weight by 2.5 in total so that net input $z = x_j^{(i)} w_j + b$ would be more positive next time we encountered this example, & thus be more likely to be above threshold of unit step function to classify example as class 1.

  Weight update Δw_j is proportional to value of $x_j^{(i)}$. E.g., $x_j^{(i)} = 2$ incorrectly classified as class 0, will push decision boundary by an even larger extent to classify this example correctly next time.

  Important to note: convergence of perceptron is only guaranteed if 2 classes are linearly separable, i.e., 2 classes can be perfectly separated by a linear decision boundary. Convergence proof in his lecture notes: https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L03_perceptron_slides.pdf. Fig. 2.3: Examples of linearly & nonlinearly separable classes. (a) Linearly separable: A linear decision boundary that separates 2 classes exist. (b) Not linearly separable: No linear decision boundary that separates 2 classes perfectly exists. shows visual examples of linearly separable & linear inseparable scenarios.

  If 2 classes can't be separated by a linear decision boundary, can set a maximum number of passes over training dataset (**epochs** – thời đại) &/or a threshold for number of tolerate misclassifications – perceptron would never stop updating weights otherwise. Later in this chap, cover Adaline algorithm that produces linear decision boundaries & converges even if classes are not perfectly linearly separable. In Chap. 3, learn about algorithms that can produce nonlinear decision boundaries.

  Remark 21 (Download example code)**.** *Can download all code examples & datasets directly from https://github.com/rasbt/machine-learning-book.*

  Before jump into implementation in next sect, what just learned can be summarized in a simple diagram that illustrates general concept of perceptron: Fig. 2.4: Weights & bias of model are updated based on error function. Preceding diagram illustrates how perceptron receives inputs of an example x & combines them with bias unit b & weights $\mathbf{w}$ to compute net input. Net input is then passed on to threshold function, which generates a binary output of 0 or 1 – predicted class label of example. During learning phase, this output is used to calculate error of prediction & update weights & bias unit.

◦ Implementing a perceptron learning algorithm in Python. In prev sec, learned how Rosenblatt's perceptron rule works, implement it in Python & apply it to Iris dataset.

* An object-oriented perceptron API. Take an object-oriented approach to defining perceptron interface as a Python class, which will allow us to initialize new `Perceptron` objects that can learn from data via a `fit` method & make predictions via a separate `predict` method. As a convention, append an underscore `_` to attributes that are not created upon initialization of object, but do this by calling object's other methods, e.g., `self.w_`.

  Remark 22 (Additional resources for Python's scientific computing stack)**.** *If not yet familiar with Python's scientific libraries or need a refresher, see resources:*

  · *NumPy:* https://sebastianraschka.com/blog/2020/numpy-intro.html

  · *pandas:* https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html

  · *Matplotlib:* https://matplotlib.org/stable/tutorials/introductory/usage.html

  Implementation of a perceptron in Python: see `perceptron.py`.
  Using this perceptron implementation, can now initialize new `Perceptron` objects with a given learning rate `eta` η & number of epochs `n_iter` (passes over training dataset).
  Via `fit` method, initialize bias `self.b_` to an initial value 0 & weights in `self.w_` to a vector $\mathbb{R}^m$ where m: number of dimensions (features) in dataset.
  Notice: initial weight vector contains small random numbers drawn from a normal distribution with a standard deviation of 0.01 via `rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])` where `rgen` is a NumPy random number generator that we seeded with a user-specified random seed so that we can reproduce previous results if desired.
  Technically, could initialize weights to 0 (in fact, this is done in original perceptron algorithm). However, if did that, then learning rate η `eta` would have no effect on decision boundary. If all weights are initialized to 0, learning rate parameter `eta` affects only scale of weight vector, not direction. If familiar with trigonometry, consider a vector $\mathbf{v}_1 = [1, 2, 3]$, where angle between $\mathbf{v}_1$ & a vector $\mathbf{v}_2 = 0.5\mathbf{v}_1$, would be exactly 0, as demonstrated by code snippet:

```
>>> v1 = np.array([1, 2, 3])
>>> v2 = 0.5 * v1
>>> np.arccos(v1.dot(v2) / (np.linalg.norm(v1) *
...           np.linalg.norm(v2)))
0.0
```

  i.e., $(\mathbf{v}_1, \mathbf{v}_2) = \arccos \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{\|\mathbf{v}_1\| \|\mathbf{v}_2\|}$. Here, `np.arccos`: trigonometric inverse cosin, `np.linalg.norm` is a function that computes length of a vector. (Our decision to draw random numbers from a random normal distribution – e.g., instead of from a uniform distribution – & to use a standard deviation of `0.01` was arbitrary: remember, just interested in small random values to avoid properties of all-zero vectors.)
  As an optional exercise after reading this chap, can change `self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])` to `self.w_ = np.zeros(X.shape[1])` & run perceptron training code presented in next sect with different values for `eta`. Observe: decision boundary does not change.

  Remark 23 (NumPy array indexing)**.** *NumPy indexing for 1D arrays works similarly to Python lists using square-bracket* `[]` *notation. For 2D arrays, 1st indexer refers to row number & 2nd indexer to column number. E.g., would see* `X[2, 3]` *to select 3rd row & 4th column of a 2D array* `X`*.*

  After weights have been initialized, `fit` method loops over all individual examples in training dataset & updates weights according to perceptron learning rule discussed in prev sect.
  Class labels are predicted by `predict` method, called in `fit` method during training to get class label for weight update; but `predict` can also be used to predict class labels of new data after we have fitted our model. Furthermore, also collect number of misclassifications during each epoch in `self.errors_` list so that can later analyze how well our perceptron performed during training. `np.dot` function that is used in `net_input` method simply calculates vector dot product $\mathbf{w}^\top \mathbf{x} + b$.

  Remark 24 (Vectorization: Replacing for loops with vectorized code)**.** *Instead of using NumPy to calculate vector dot product between 2 arrays* `a, b`*, via* `a.dot(b)` *or* `np.dot(a, b)`*, could also perform calculation in pure Python via* `sum([i * j for i, j in zip(a, b)])`*. However, advantage of using NumPy over classic Python* `for` *loop structures: its arithmetic operations are vectorized. Vectorizations means: an elemental arithmetic operation is automatically applied to all elements in an array. By formulating our arithmetic operations as a sequence of instructions on an array, rather than performing a set of operations for each element at a time, can make better use of modern* central processing unit (CPU) *architectures with* single instruction, multiple data (SIMD) *support. Furthermore, NumPy uses highly optimized linear algebra libraries, e.g.* Basic Linear Algebra Subprograms (BLAS) *&* Linear Algebra Package (LAPACK)*, that have been written in C or Fortran. Lastly, NumPy also allows us to write our code in a more compact & intuitive way using basics of linear algebra, e.g. vector & matrix dot products.*

* Training a perceptron model on Iris dataset. To test our perceptron implementation, restrict following analyses & examples in remainder of this chap to 2 feature variables (dimensions). Although perceptron rule is not restricted to 2D, considering only 2 features, sepal length & petal length, will allow to visualize decision regions of trained model in scatterplot for learning purposes.
  Note: also only consider 2 flower classes, setora & versicolor, from Iris dataset for practical reasons – remember, perceptron is a binary classifier. However, perceptron algorithm can be extended to multi-class classification – e.g., *1-vs-all (OvA)* technique.

Remark 25 (OvA method for multi-class classification). *OvA, sometimes also called* 1-vs-rest *(OvR), is a technique that allows us to extend & binary classifier to multi-class problems. Using OvA, can train 1 classifier per class, where particular class is treated as positive class & examples from all other classes are considered negative classes. If were to classify a new, unlabeled data instance, would use our n classifiers, where n: number of class labels, & assign class label with highest confidence to particular instance we want to classify. In case of perceptron, would use OvA to choose class label that is associated with largest absolute net input value.*

1st, use `pandas` library to load Iris dataset directly from `UCI ML Repository` into a `DataFrame` object & print last 5 lines via `tail` method to check: data was loaded correctly:

```
import os
import pandas as pd

try:
    s = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'
    print('From URL:', s)
    df = pd.read_csv(s, header=None, encoding='utf-8')
except HTTPError:
    s = 'iris.data'
    print('From local Iris path:', s)
    df = pd.read_csv(s, header=None, encoding='utf-8')

df.tail()
```

After executing prev code, should se following output, which shows last 5 lines of Iris dataset:

```
(base) nqbh@nqbh-dell:~/advanced_STEM_beyond/machine_learning/Python$ python perceptron_Iris_dataset.p
From URL: https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data
     0    1    2    3               4
145  6.7  3.0  5.2  2.3  Iris-virginica
146  6.3  2.5  5.0  1.9  Iris-virginica
147  6.5  3.0  5.2  2.0  Iris-virginica
148  6.2  3.4  5.4  2.3  Iris-virginica
149  5.9  3.0  5.1  1.8  Iris-virginica
```

Remark 26 (Loading Iris dataset). *Can find a copy of Iris dataset (& all other dataset used in this book) in code bundle of this book, which can use if working offline or if UCI server at https://archive.ics.uci.edu/ml/machine-learning-databa iris/iris.data is temporarily unavailable. E.g., to load Iris dataset from a local directory, can replace this line,*

```
df = pd.read_csv(
  'https://archive.ics.uci.edu/ml/'
  'machine-learning-databases/iris/iris.data',
header=None, encoding='utf-8')
```

with

```
df = pd.read_csv(
  'your/local/path/to/iris.data',
header=None, encoding='utf-8')
```

Next, extract 1st 100 class labels that correspond to 50 Iris-setosa & 50 Iris-versicolor flowers & convert class labels into 2 integer class labels, `1` (versicolor) & `0 (setosa)`, that we assign to a vector `y` where `values` method of a pandas `DataFrame` yields corresponding NumPy representation.
Similarly, extract 1st feature column (sepal length) & 3rd feature column (petal length) of those 100 training examples & assign them to a feature matrix `X` which we can visualize via a 2D scatterplot:

```
import numpy as np
import matplotlib.pyplot as plt
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', 0, 1)

# extract sepal length \& petal length
X = df.iloc[0:100, [0, 2]].values

# plot data
plt.scatter(X[:50, 0], X[:50, 1],
color='red', marker='o', label='Setosa')
plt.scatter(X[50:100, 0], X[50:100, 1],
```

```
color='blue', marker='s', label='Versicolor')

plt.xlabel('Sepal length [cm]')
plt.ylabel('Petal length [cm]')
plt.legend(loc='upper left')

# plt.savefig('images/02_06.png', dpi=300)
plt.show()
```

After executing preceding code example, should see following scatterplot Fig. 2.6: Scatterplot of setosa & versicolor flowers by sepal & petal length: show distribution of flower examples in Iris dataset along 2 feature axes: petal length & sepal length (measured in centimeters). In this 2D feature subspace, can see: a linear decision boundary should be sufficient to separate setosa from versicolor flowers. Thus, a linear classifier e.g. perceptron should be able to classify flowers in this dataset perfectly.
Time to train our perception algorithm on Iris data subset just extracted. Plot misclassification error for each epoch to check whether algorithm converged & found a decision boundary that separates 2 Iris flower classes:

```
ppn = Perceptron(eta=0.1, n_iter=10)

ppn.fit(X, y)

plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of updates')

# plt.savefig('images/02_07.png', dpi=300)
plt.show()
```

Note: number of misclassification errors & number of updates is the same, since perceptron weights & bias are updated each time it misclassifies an example. An executing preceding code, should see plot of misclassification errors vs. number of epochs, as shown in Fig. 2.7: A plot of misclassification errors against number of epochs. Our perceptron converged after 6th epoch & should now be able to classify training examples perfectly. Implement a small convenience function to visualize decision boundaries for 2 datasets. [code]
1st, define a number of `colors, makers` & create a colormap from list of colors via `ListedColormap`. Then, determine minimum & maximum values for 2 features & use those feature vectors to create a pair of grid arrays `xx1, xx2` via NumPy `meshgrid` function. Since trained our perceptron classifier on 2 feature dimensions, need to flatten grid arrays & create a matrix that has same number of columns as Iris training subset so that can use `predict` method to predict class labels `lab` of corresponding grid points.
After reshaping predicted class labels `lab` into a grid with same dimensions as `xx1, xx2`, can now draw a contour plot via Matplotlib's `contourf` function, which maps different decision regions to different colors for each predicted class in grid array:

```
>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('Sepal length [cm]')
>>> plt.ylabel('Petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing preceding code example, should now see a plot of decision regions, as shown in Fig. 2.8: A plot of perceptron's decision regions. As can see in plot, perceptron learned a decision boundary that can classify all flower examples in Iris training subset perfectly.

Remark 27 (Perceptron convergence)**.** *Although perceptron classified 2 Iris flower classes perfectly, convergence is 1 of biggest problems of perceptron.* Rosenblatt *proved mathematically: perceptron learning rule converges if 2 classes can be separated by a linear hyperplane. However, if classes cannot be separated perfectly by such a linear decision boundary, weights will never stop updating unless we set a maximum number of epochs. Interested readers can find a summary of proof in lecture notes at https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L03_perceptron_slides.pdf.*

- Adaptive linear neurons & convergence of learning. In this sect, take a look at another type of single-layer *neural network (NN) : ADAptive LInear NEuron (Adaline).* Adaline was published by Bernard Widrow & his doctoral student Tedd hoff only a few years after Rosenblatt's perceptron algorithm, & it can be considered an improvement on latter: *An Adaptive "Adaline" Neuron Using Chemical "Memistors"*, Technical Report Number 1553-2 by B. Widrow & colleagues, Stanford Electron Labs, Stanford, CA, October 1960).
  Adaline algorithm is particularly interesting because it illustrates key concepts of defining & minimizing continuous loss functions. This lays groundwork for understanding ML algorithms for classification, e.g. logistic regression, support vector machines, & multilayer neural networks, as well as linear regression models.

Key difference between Adaline rule (also known as *Widrow-Hoff rule*) & Rosenblatt's perceptron: weights are updated based on a linear activation function rather than a unit step function like in perception. In Adaline, this linear activation function $\sigma(z)$ is simply identity function of net input, so that $\sigma(z) = z$.

While linear activation function is used for learning weights, still us a threshold function to make final prediction, which is similar to unit step function covered earlier.

Main differences between perceptron & Adaline algorithm are highlighted in Fig. 2.9: A comparison between a perceptron & Adaline algorithm., indicates: Adaline algorithm compares true class labels with linear activation function's continuous valued output to compute model error & update weights. In contrast, perceptron compares true class labels to predicted class labels.

* Minimizing loss functions with gradient descent. 1 of key ingredients of supervised ML algorithms is a defined *objective function* that is to be optimized during learning process. This objective function is often a loss or cost function that we want to minimize. In case of Adaline, can define loss function L to learn model parameters as *mean squared error (MSE)* between calculated outcome & true class label:

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \sigma(z^{(i)}))^2. \tag{199}$$

Main advantage of this continuous linear activation function, in contrast to unit step function: loss function becomes differentiable. Another nice property of this loss function: it is convex; thus, can use a very simple yet powerful optimization algorithm called *gradient descent* to find weights that minimize our loss function to classify examples in Iris dataset.

As illustrated in Fig. 2.10: How gradient descent works, can describe main idea behind gradient descent as *climbing down a hill* until a local or global loss minimum is reached. In each iteration, take a step in opposite direction of gradient, where step size is determined by value of learning rate, as well as slope of gradient (for simplicity, following figure visualizes this only for a single weight w).

Using gradient descent, can now update model parameters by taking a step in opposite direction of gradient $\nabla L(\mathbf{w}, b)$ of our loss function $L(\mathbf{w}, b)$:

$$\mathbf{w} \coloneqq \mathbf{w} + \Delta\mathbf{w},\ b \coloneqq b + \Delta b. \tag{200}$$

Parameter changes, $\Delta\mathbf{w}, \Delta b$, are defined as negative gradient multiplied by learning rate η:

$$\Delta\mathbf{w} = -\eta\nabla_{\mathbf{w}}L(\mathbf{w}, b),\ \Delta b = -\eta\nabla_b L(\mathbf{w}, b). \tag{201}$$

To compute gradient of loss function, need to compute partial derivative of loss function w.r.t. each weight w_j:

$$\frac{\partial L}{\partial w_j} = -\frac{2}{n}\sum_i (y^{(i)} - \sigma(z^{(i)}))x_j^{(i)}. \tag{202}$$

Similarly, compute partial derivative of loss w.r.t. bias as:

$$\frac{\partial L}{\partial b} = -\frac{2}{n}\sum_i (y^{(i)} - \sigma(z^{(i)})). \tag{203}$$

Note: 2 in numerator is merely a constant scaling factor, & could omit it without affecting algorithm. Removing scaling factor has same effect as changing learning rate by a factor of 2. Following information box explains where this scaling factor originates.

So can write weight update as:

$$\Delta w_j = -\eta\frac{\partial L}{\partial w_j},\ \Delta b = -\eta\frac{\partial L}{\partial b}. \tag{204}$$

Since update all parameters simultaneously, Adaline learning rule becomes:

$$\mathbf{w} \coloneqq \mathbf{w} + \Delta\mathbf{w},\ b \coloneqq b + \Delta b. \tag{205}$$

Remark 28 (Mean squared error derivative)**.** *If familiar with calculus, partial derivative of MSE loss function w.r.t. jth weight can be obtained as follows: [...] Same approach can be used to find partial derivative* $\frac{\partial L}{\partial b}$ *except that* $\frac{\partial}{\partial b}(y^{(i)} - \sum_i (w_j^{(i)}x_j^{(i)} + b)) = -1$ *& thus last step simplifies to* $-\frac{2}{n}\sum_i (y^{(i)} - \sigma(z^{(i)}))$.

Although Adaline learning rule looks identical to perceptron rule, note: $\sigma(z^{(i)})$ with $z^{(i)} = \mathbf{w}^\top\mathbf{x}^{(i)} + b \in \mathbb{R}$ & not an integer class label. Furthermore, weight update is calculated based on all examples in training dataset (instead of updating parameters incrementally after each training example), which is why this approach is also referred to as *batch gradient descent* (giảm dần độ dốc hàng loạt). To be more explicit & avoid confusion when talking about related concepts later in this chap & this book, refer to this process as *full batch gradient descent* (giảm dần độ dốc hàng loạt đầy đủ).

* Implementing Adaline in Python. Since perceptron rule & Adaline are very similar, take perceptron implementation defined earlier & change `fit` method so that weight & bias parameters are now updated by minimizing loss function via gradient descent: `[code]`.

Instead of updating weights after evaluating each individual training example, as in perceptron, calculate gradient based on whole training dataset. For bias unit, this is done via `self.eta * 2.0 * errors.mean()`, where `errors` is an array containing partial derivative values ∂_b. Similarly, update weights. However note: weight updates via partial derivatives $\frac{\partial L}{\partial w_j}$ involve feature values x_j, which can compute by multiplying `errors` with each feature value for each weight:

```
for w_j in range(self.w_.shape[0]):
    self.w_[w_j] += self.eta * (2.0 * (X[:, w_j]*errors)).mean()
```

To implement weight update more efficiently without using a `for` loop, can use a matrix-vector multiplication between our feature matrix & error vector instead:

```
self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
```

Note: `activation` method has no effect on code since it is simply an identity function. Here, added activation function (computed via `activation` method) to illustrate general concept with regard to how information flows through a single-layer NN: features from input data, net input, activation, & output.
In next chap, learn about a logistic regression classifier that uses a non-identity, nonlinear activation function. See: a logistic regression model is closely related to Adaline, with only difference being its activation & loss function.
now, similar to previous perceptron implementation, collect loss values in a `self.losses_` list to check whether algorithm converged after training.

Remark 29 (Matrix multiplication)**.** *Performing a matrix multiplication is similar to calculating a vector dot-product where each row in matrix is treated as a single row vector. This vectorized approach represents a more compact notation & results in a more efficient computation using NumPy.*

In practice, often require some experimentation to find a good learning rate η for optimal convergence. So choose 2 different learning rates $\eta = 0.1, \eta = 0.0001$ to start with & plot loss functions vs. number of epochs to see how well Adaline implementation learns from training data.

Remark 30 (Hyperparameters)**.** *Learning rate η* `eta` *as well as number of epochs* `n_iter`*, are so-called* hyperparameters *(or* tuning parameters*) of perceptron & Adaline learning algorithms. In Chap. 6, take a look at different techniques to automatically find values of different hyperparameters that yield optimal performance of classification model.*

Plot loss against number of epochs for 2 different learning rates:

```
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
>>> ada1 = AdalineGD(n_iter=15, eta=0.1).fit(X, y)
>>> ax[0].plot(range(1, len(ada1.losses_) + 1),
...            np.log10(ada1.losses_), marker='o')
>>> ax[0].set_xlabel('Epochs')
>>> ax[0].set_ylabel('log(Mean squared error)')
>>> ax[0].set_title('Adaline - Learning rate 0.1')
>>> ada2 = AdalineGD(n_iter=15, eta=0.0001).fit(X, y)
>>> ax[1].plot(range(1, len(ada2.losses_) + 1),
...           ada2.losses_, marker='o')
>>> ax[1].set_xlabel('Epochs')
>>> ax[1].set_ylabel('Mean squared error')
>>> ax[1].set_title('Adaline - Learning rate 0.0001')
>>> plt.show()
```

As can see in resulting loss function plots, encountered 2 different types of problems. Left chart shows what could happen if choose a learning rate that is too large. Instead of minimizing loss function, MSE becomes larger in every epoch, because *overshoot* global minimum. On other hand, can see: loss decreases on right plot, but chosen learning rate $\eta = 0.0001$ is so small: algorithm would require a very large number of epochs to converge to global loss minimum: Fig. 2.11: Error plots for suboptimal learning rates.
Fig. 2.12: A comparison of a well-chosen learning rate & a learning rate that is too large illustrates what might happen if change value of a particular weight parameter to minimize loss function L. Left subfigure illustrates case of a well-chosen learning rate, where loss decreases gradually, moving in direction of global minimum.
Subfigure on right, however, illustrates what happens if choose a learning rate that is too large – overshoot global minimum.

∗ Improving gradient descent through feature scaling. Many ML algorithms that will encounter throughout this book require some sort of feature scaling for optimal performance, discussed in Chaps. 3–4.
Gradient descent is 1 of many algorithms that benefit from feature scaling. In this sect, use a feature scaling method called *standardization*. This normalization produce helps gradient descent learning to converge more quickly; however, it does not make original dataset normally distributed. Standardization shifts mean of each feature so that it is centered at 0 & each feature has a standard deviation of 1 (unit variance). E.g., to standardize jth feature, can simply subtract sample mean μ_j from every training example & divide it by its standard deviation σ_j:

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}. \tag{206}$$

Here x_j: a vector consisting of jth feature values of all training examples n, & this standardization technique is applied to each feature j in our dataset.
1 of reasons why standardization helps with gradient descent learning: easier to find a learning rate that works well for all weights (& bias). If features are on vastly different scales, a learning rate that works well for updating 1 weight might be

too large or too small to update other weight equally well. Overall, using standardized features can stabilize training s.t. optimizer has to go through fewer steps to find a good or optimal solution (global loss minimum). Fig. 2.13: A comparison of unscaled & standardized features on gradient updates illustrates possible gradient updates with unscaled features (left) & standardized features (right), where concentric circles represent loss surface as a function of 2 model weights in a 2D classification problem.

Standardization can easily be achieved by using built-in NumPy methods `mean, std`:

```
>>> X_std = np.copy(X)
>>> X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
>>> X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

After standardization, train Adaline again & see: it now converges after a small number of epochs using a learning rate of $\eta = 0.5$:

```
>>> ada_gd = AdalineGD(n_iter=20, eta=0.5)
>>> ada_gd.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada_gd)
>>> plt.title('Adaline - Gradient descent')
>>> plt.xlabel('Sepal length [standardized]')
>>> plt.ylabel('Petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
>>> plt.plot(range(1, len(ada_gd.losses_) + 1),
...          ada_gd.losses_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Mean squared error')
>>> plt.tight_layout()
>>> plt.show()
```

After executing this code, should see a figure of decision regions, as well as a plot of declining loss, as shown in Fig. 2.14: Plots of Adaline's decision regions & MSE by number of epochs.

As can see in plots, Adaline has now converged after training on standardized features. However, note: MSE remains non-zero even though all flower examples were classified correctly.

∗ Large-scale ML & stochastic gradient descent. In prev sect, learned how to minimize a loss function by taking a step in opposite direction of loss gradient that is calculated from whole traiing dataset; this is why this approach is sometimes also referred to as full batch gradient descent. Now imagine: have a very large dataset with millions of data points, which is not uncommon in many ML applications. Running full batch gradient descent can be computationally quite costly in such scenarios, since need to reevalate whole training dataset each time tak 1 step toward global minimum.

A popular alternative to batch gradient descent algorithm is *stochastic gradient descent (SGD)*, which is sometimes also called *iterative-* or *online gradient descent*. Instead of updating weights based on sum of accumulated errors over all training examples $x^{(i)}$:

$$\Delta w_j = \frac{2\eta}{n} \sum_i (y^{(i)} - \sigma(z^{(i)}))x_j^{(i)} \tag{207}$$

update parameters incrementally for each training example, e.g.:

$$\Delta w_j = \eta(y^{(i)} - \sigma(z^{(i)}))x_j^{(i)},\ \Delta b = \eta(y^{(i)} - \sigma(z^{(i)})). \tag{208}$$

Although SGd can be considered as an approximation of gradient descent, it typically reaches convergence much faster because of more frequent weight updates. Since each gradient is calculated based on a single training example, error surface is noisier than in gradient descent, which can also have advantage: SGD can escape shallow local minima more readily if working with nonlinear loss functions, as see in Chap. 11. To obtain satisfying results via SGD, important to present training data in a random order; also, want to shuffle training dataset for every epoch to prevent cycles.

Remark 31 (Adjusting learning rate during training)**.** *In SGD implementations, fixed learning rate η is often replaced by an adaptive learning rate that decreases over time, e.g.: $\frac{c_1}{[number\ of\ iterations]+c_2}$ where c_1, c_2: constants. Note: SGD does not reach global loss minimum but an area very close to it. & using an adaptive learning rate, can achieve further annealing to loss minimum.*

– Lưu ý: SGD không đạt đến mức tổn thất tối thiểu toàn cục mà là một khu vực rất gần với mức đó. & sử dụng tốc độ học thích ứng, có thể đạt được quá trình tôi luyện tiếp theo đến mức tổn thất tối thiểu.

Another advantage of SGD: can use it for *online learning*. In online learning, our model is trained on fly as new training data arrives. This is especially useful if accumulating large amounts of data, e.g., customer data in web applications. Using online learning, system can immediately adapt to changes, & training data can be discarded after updating model if storage space is an issue.

Remark 32 (Mini-batch gradient descent)**.** *A compromise between full batch gradient descent & SGD is so-called* mini-batch gradient descent*. Mini-batch gradient descent can be understood as applying full batch gradient descent to smaller*

subsets of training data, e.g., 32 training examples at a time. Advantage over full batch gradient descent: convergence is reached faster via mini-batches because of more frequent weight updates. Furthermore, mini-batch learning allows us to replace `for` *loop over training examples in SGD with vectorized operations leveraging concepts from linear algebra (e.g., implementing a weighted sum via dot product), which can further improve computational efficiency of our learning algorithm.*

Since already implemented Adaline learning rule using gradient descent, only need to make a few adjustments to modify learning algorithm to update weights via SGD. Inside `fit` method, now update weights after each training example. Furthermore, implement an additional `partial_fit` method, which does not reinitialize weights, for online learning. In order to check whether our algorithm converged after training, calculate loss as average loss of training examples in each epoch. Furthermore, add an option to shuffle training data before each epoch to avoid repetitive cycles when optimizing loss function; via `random_state` parameter, allow specification of a random seed for reproducibility `[code]`.
`_shuffle` method that now using in `AdalineSGD` classifier works as follows: via `permutation` function in `np.random`, generate a random sequence of unique numbers in range 0 to 100. Those numbers can then be used as indices to shuffle our feature matrix & class label vector.
Can then use `fit` method to train `AdalineSGD` classifier & use `plot_decision_regions` to plot our training results:

```
>>> ada_sgd = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada_sgd.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada_sgd)
>>> plt.title('Adaline - Stochastic gradient descent')
>>> plt.xlabel('Sepal length [standardized]')
>>> plt.ylabel('Petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
>>> plt.plot(range(1, len(ada_sgd.losses_) + 1), ada_sgd.losses_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Average loss')
>>> plt.tight_layout()
>>> plt.show()
```

2 plots obtained from executing preceding code example: Fig. 2.15: Decision regions & average loss plots after training an Adaline model using SGD.
Average loss goes down pretty quickly, & final decision boundary after 15 epochs looks similar to batch gradient descent Adaline. If want to update model, e.g., in an online learning scenario with streaming data, could simply call `partial_fit` method on individual training examples – e.g., `ada_sgd.partial_fit(X_std[0, :], y[0])`.

- Summary. In this chap, gained a good understanding of basic concepts of linear classifiers for supervised learning. After implemented a perceptron, saw how can train adaptive linear neurons effectively via a vectorized implementation of gradient descent & online learning via SGD.

  Have seen how to implement simple classifiers in Python, ready to move on to next chap, where use Python `scikit-learn` ML library to get access to more advanced & powerful ML classifiers, commonly used in academia as well as in industry.

  Object-oriented approach used to implement perceptron & Adaline algorithms will help with understanding scikit-learn API, which is implemented based on same core concepts used in this chap: `fit, predict` methods. Based on these core concepts, learn about logistic regression for modeling class probabilities & support vector machines for working with nonlinear decision boundaries. In addition, introduce a different class of supervised learning algorithms, tree-based algorithms, commonly combined into robust ensemble classifiers.

- Chap. 3: A Tour of ML Classifiers Using Scikit-Learn. In this chap, take a tour of a selection of popular & powerful ML algorithms commonly used in academia as well as in industry. While learning about differences between several supervised learning algorithms for classification, also develop an appreciation of their individual strengths & weaknesses. Take 1st steps with scikit-learn library, which offers a user-friendly & consistent interface for using those algorithms efficiently & productively.

  Topics covered:

  - An introduction to robust & popular algorithms for classification, e.g. logistic regression, support vector machines, decision trees, & k-nearest neighbors
  - Examples & explanations using scikit-learn ML library, which provides a wide variety of ML algorithms via a user-friendly Python API
  - Discussions about strengths & weaknesses of classifiers with linear & nonlinear decision boundaries

  - Choosing a classification algorithm. Choosing an appropriate classification algorithm for a particular problem task requires practice & experience; each algorithm has its own quirks & is based on certain assumptions. To paraphrase *no free lunch theorem* by David H. Wolpert, no single classifier works best across all possible scenarios *The Lack of A Priori Distinctions Between Learning Algorithms*, Wolpert, David H, Neural Computation 8.7 (1996): 1341-1390. In practice, always

recommended: compare performance of at least a handful of different learning algorithms to select best model for particular problem; these may differ in number of features or examples, amount of noise in a dataset, & whether classes are linearly separable.

Eventually, performance of a classifier – computational performance as well as predictive power – depends heavily on underlying data that is available for learning. 5 main steps involved in training a supervised ML algorithm can be summarized as follows:

1. Selecting features & collecting labeled training examples
2. Choosing a performance metric
3. Choosing a learning algorithm & training a model
4. Evaluating performance of model
5. Changing settings of algorithm & tuning model.

Since approach of this book is to build ML knowledge step by step, mainly focus on main concepts of different algorithms in this chap & revisit topics e.g. feature selection & preprocessing, performance metrics, & hyperparameter tuning for mode detailed discussions later in book.

- 1st steps with scikit-learn – training a perceptron. In Chap. 2, learned about 2 related learning algorithms for classification, *perceptron* rule & *Adaline*, which implemented in Python & NumPy by ourselves. Now take a look at scikit-learn API, which combines a user-friendly & consistent interface with a highly optimized implementation of several classification algorithms. Scikit-learn library offers not only a large variety of learning algorithms, but also many convenient functions to preprocess data & to fine-tune & evaluate our models. Discuss this in more detail, together with underlying concepts, in Chaps. 4–5.

  To get started with scikit-learn library, train a perceptron model similar to one implemented in Chap. 2. For simplicity, use already familiar *Iris dataset* throughout following sects. Conveniently, Iris dataset is already available via scikit-learn, since it is a simple yet popular dataset that is frequently used for testing & experimenting with algorithms. Similar to prev chap, only use 2 features from Iris dataset for visualization purposes.

  Assign petal length & petal width of 150 flower examples to feature matrix `X` & corresponding class labels of flower species to vector array `y`:

```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
>>> print('Class labels:', np.unique(y))
Class labels: [0 1 2]
```

  `np.unique(y)` function returned 3 unique class labels stored in `iris.target`, & Iris flower class names `Iris-setosa, Iris-versicolor, Iris-virginica`, are already stored as integers (here: `0, 1, 2`). Although many scikit-learn functions & class methods also work with class labels in string format, using integer labels is a recommended approach to avoid technical glitches (tránh trục trặc kỹ thuật) & improve computational performance due to a smaller memory footprint; furthermore, encoding class labels as integers is a common convention among most ML libraries.

  To evaluate how well a trained model performs on unseen data, further split dataset into separate training & test datasets. In Chap. 6, discuss best practices around model evaluation in more detail. Using `train_test_split` function from scikit-learn's `model_selection` module, randomly split `X, y` arrays into 30% test data (45 examples) & 70% training data (105 examples):

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=1, stratify=y
... )
```

  Note: `train_test_split` function already shuffles training datasets internally before splitting; otherwise, all examples from class 0 & class 1 would have ended up in training datasets, & test dataset would consist of 45 examples from class 2. Via `random_state` parameter, provided a fixed random seed `random_state=1` for internal pseudo-random number generator that is used for shuffling datasets prior to splitting. Using such a fixed `random_state` ensures: our results are reproducible.

  Lastly, took advantage of built-in support for stratification (sự phân tầng) via `stratify=y`. In this context, stratification means: `train_test_split` method returns training & test subsets that have same proportions of class labels as input dataset. Can use NumPy's `bincount` function, which counts number of occurrences of each value in an array, to verify that this is indeed the case:

```
>>> print('Labels counts in y:', np.bincount(y))
Labels counts in y: [50 50 50]
>>> print('Labels counts in y_train:', np.bincount(y_train))
Labels counts in y_train: [35 35 35]
```

```
>>> print('Labels counts in y_test:', np.bincount(y_test))
Labels counts in y_test: [15 15 15]
```

Many ML & optimization algorithms also require feature scaling for optimal performance, as saw in *gradient descent* example in Chap. 2. Here, standardize features using `StandardScaler` class from scikit-learn's `preprocessing` module:

```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> sc.fit(X_train)
>>> X_train_std = sc.transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Using preceding code, loaded `StandardScaler` class from `preprocessing` module & initialized a new `StandardScaler` object assigned to `sc` variable. Using `fit` method, `StandardScaler` estimated parameters, μ (sample mean) & σ (standard deviation), for each feature dimension from training data. By calling `transform` method, then standardized training data using those estimated parameters μ, σ. Note: used same scaling parameters to standardize test dataset so that both values in training & test dataset are compatible with 1 another.

Having standardized training data, can now train a perceptron model. Most algorithms in scikit-learn already support multiclass classification by default via *1-vs-rest (OvR)* method, which allows to feed 3 flower classes to perceptron all at once. Code:

```
>>> from sklearn.linear_model import Perceptron
>>> ppn = Perceptron(eta0=0.1, random_state=1)
>>> ppn.fit(X_train_std, y_train)
```

Scikit-learn interface will remind of our perceptron implementation in Chap. 2. After loading `Perceptron` class from `linear_model` module, initialized a new `Perceptron` object & trained model via `fit` method. Here, model parameter `eta0` is equivalent to learning rate `eta` that we used in our own perceptron implementation.

As in Chap. 2, finding an appropriate learning rate requires some experimentation. If learning rate is too large, algorithm will overshoot global loss minimum. If learning rate is too small, algorithm will require more epochs until convergence, which can make learning slow – especially for large datasets. Also, used `random_state` parameter to ensure reproducibility of initial shuffling of training dataset after each epoch.

Having trained a model in scikit-learn, can make predictions via `predict` method, just like in our own perceptron implementation in Chap. 2. Code:

```
>>> y_pred = ppn.predict(X_test_std)
>>> print('Misclassified examples: %d' % (y_test != y_pred).sum())
Misclassified examples: 1
```

Executing code, can see: perceptron misclassifies 1 out of 45 flower examples. Thus, misclassification error on test dataset is $\frac{1}{45} \approx 0.022 = 2.2\%$.

Remark 33 (Classification error vs. accuracy)**.** *Instead of misclassification error, many ML practitioners report classification accuracy of a model, which is simply calculated as follows:* $1 - \text{error} = 0.978 = 97.8\%$. *Whether use classification error or accuracy is merely a matter of preference.*

Note: scikit-learn also implements a large variety of different performance metrics that are available via `metrics` module. E.g., can calculate classification accuracy of perceptron on test dataset as follows:

```
>>> from sklearn.metrics import accuracy_score
>>> print('Accuracy: %.3f' % accuracy_score(y_test, y_pred))
Accuracy: 0.978
```

Here `y_test`: true labels, `y_pred`: class labels predicted previously. Alternatively, each classifier in scikit-learn has a `score` method, which computes a classifier's prediction accuracy by combining `predict` call with `accuracy_score`, as shown here:

```
>>> print('Accuracy: %.3f' % ppn.score(X_test_std, y_test))
Accuracy: 0.978
```

Remark 34 (Overfitting)**.** *Note: will evaluate performance of our models based on test dataset in this chap. In Chap. 6, learn about useful techniques, including graphical analysis, e.g. learning curves, to detect & prevent overfitting. Overfitting means: model captures patterns in training data well but fails to generalize well to unseen data.*

Finally, can use our `plot_decision_regions` function from Chap. 2 to plot *decision regions* of our newly trained perceptron model & visualize how well it separates different flower examples. However, add a small modifications to highlight data instances from test dataset via small circles: `[code]`.

With slight modification that we made to `plot_decision_regions` function, can now specify indices of examples that we want to mark on resulting plots. Code:

```
>>> X_combined_std = np.vstack((X_train_std, X_test_std))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X=X_combined_std, y=y_combined, classifier=ppn, test_idx=range(105, 150))
>>> plt.xlabel('Petal length [standardized]')
>>> plt.ylabel('Petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

In resulting plot, 3 flower classes can't be perfectly separated by a linear decision boundary: Fig. 3.1: Decision boundaries of a multi-class perceptron model fitted to Iris dataset. However, remember from discussion in Chap. 2: perceptron algorithm never converges on datasets that aren't perfectly linearly separable, which is why use of perceptron algorithm is typically not recommended in practice. In following sects, look at more powerful linear classifiers that converge to a loss minimum even if classes are not perfectly linearly separable.

Remark 35 (Additional perceptron settings)**.** `Perceptron`*, as well as other scikit-learn functions & classes, often has additional parameters that we omit for clarity. Can read more about those parameters using* `help` *function in Python (e.g.,* `help(Perceptron)`*) or by going through excellent scikit-learn online documentation at* *http://scikit-learn.org/stable/*.

- Modeling class probabilities via logistic regression. Although perceptron rule offers a nice & easy-going introduction to ML algorithms for classification, its biggest disadvantage: it never converges if classes are not perfectly linearly separable. Classification task in prev sect would be an example of such a scenario. Reason for this: weights are continuously being updated since there is always at least 1 misclassified training example present in each epoch. Of course, can change learning rate & increase number of epochs, but be warned: perceptron will never converge on this dataset.

  To make better use of our time, take a look at another simple, yet more powerful, algorithm for linear & binary classification problems: *logistic regression*. Note: despite its name, logistic regression is a model for classification, not regression.

  - Logistic regression & conditional probabilities. Logistic regression is a classification model that is very easy to implement & performs very well on linearly separable classes. It is 1 of most widely used algorithms for classification in industry. Similar to perceptron & Adaline, logistic regression model in this chap is also a linear model for binary classification.

    Remark 36 (Logistic regression for multiple classes)**.** *Note: logistic regression can be readily generalized to multi-class settings, which is known as* multinomial logistic regression*, or* softmax regression*. More detailed coverage of multinomial logistic regression is outside scope of this book, but interested reader can find more information in lecture note at* *https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L08_logistic__slides.pdf* *or* *https://www.youtube.com/watch?v=L0FU8NFpx4E*.
    Another way to use logistic regression in multiclass settings is via OvR technique, discussed previously.

    To explain main mechanics behind logistic regression as a probabilistic model for binary classification, 1st introduce the *odds*: odds in favor of a particular event. The odds can be written as $\frac{p}{1-p}$, where p stands for probability of positive event. Term "positive event" does not necessarily mean "good", but refers to event that we want to predict, e.g., probability that a patient has a certain disease given certain symptoms; can think of positive event as class label $y = 1$ & symptoms as features x. Hence, for brevity, can define probability p as $p \coloneqq p(y = 1|x)$, conditional probability that a particular example belongs to a certain class 1 given its features x.
    Can then further define *logit* function, which is simply logarithm of odds (log-odds):

$$\text{logit}(p) = \log \frac{p}{1-p}. \tag{209}$$

    Note: log refers to natural logarithm, as it is common convention in CS. Logit function takes input values in range 0 to 1 & transforms them into values over entire real-number range.
    Under logistic model, assume: there is a linear relationship between weighted inputs (referred to as net inputs in Chap. 2) & log-odds:

$$\text{logit}(p) = w_1 x_1 + \cdots + w_m x_m + b = \sum_{i=1}^{m} w_i x_i + b = \mathbf{w}^\top \mathbf{x} + b. \tag{210}$$

    While preceding describes an assumption we make about linear relationship between log-odds & net inputs, what actually interested in is probability p, class-membership probability of an example given its features. While logit function maps probability to a real-number range, can consider inverse of this function to map real-number range back to a $[0, 1]$ range for probability p.

This inverse of logit function is typically called *logistic sigmoid function*, sometimes simply abbreviated to *sigmoid function* due to its characteristic S-shape:

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{211}$$

Here z: net input, linear combination of weights, & inputs (i.e., features associated with training examples): $z = \mathbf{w}^\top \mathbf{x} + b$. Simply plot sigmoid function for some values in range $[-7, 7]$ to see how it works:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def sigmoid(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> z = np.arange(-7, 7, 0.1)
>>> sigma_z = sigmoid(z)
>>> plt.plot(z, sigma_z)
>>> plt.axvline(0.0, color='k')
>>> plt.ylim(-0.1, 1.1)
>>> plt.xlabel('z')
>>> plt.ylabel('$\sigma (z)$')
>>> # y axis ticks \& gridline
>>> plt.yticks([0.0, 0.5, 1.0])
>>> ax = plt.gca()
>>> ax.yaxis.grid(True)
>>> plt.tight_layout()
>>> plt.show()
```

As a result of executing previous code example, should now see S-shaped (sigmoidal) curve: Fig. 3.2: A plot of logistic sigmoid function. Can see: $\sigma(z)$ approaches 1 if $z \to \infty$ since e^{-z} becomes very small for large values of z. Similarly, $\sigma(z) \to 0$ for $z \to -\infty$ as a result of an increasingly large denominator. Thus, can conclude: this sigmoid function takes real-number values as input & transforms them into values in range $[0, 1]$ with an intercept at $\sigma(0) = 0.5$.
To build some understanding of logistic regression model, can relate it to Chap. 2. In Adaline, used identity function $\sigma(z) = z$ as activation function. In logistic regression, this activation function simply becomes sigmoid function defined earlier.
Difference between Adaline & logistic regression is illustrated in Fig. 3.3: Logistic regression compared to Adaline, where only difference is activation function.
Output of sigmoid function is then interpreted as probability of a particular example belonging to class 1, $\sigma(z) = p(y = 1|\mathbf{x}; \mathbf{w}, b)$, given its features $\mathbf{x}$ & parametrized by weights $\mathbf{w}$ & bias b. E.g., if compute $\sigma(z) = 0.8$ for a particular flower example, it means: chance this this example is an `Iris-versicolor` flower is 80%. Therefore, probability that this flower is an `Iris-setosa` flower can be calculated as $p(y = 0|\mathbf{x}; \mathbf{w}, b) = 1 - p(y = 1|\mathbf{x}; \mathbf{w}, b) = 0.2 = 20\%$.
Predicted probability can then simply be converted into a binary outcome via a threshold function:

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(z) \geq 0.5, \\ 0 & \text{otherwise.} \end{cases} \tag{212}$$

If look at preceding plot of sigmoid function, this $\Leftrightarrow$

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(z) \geq 0.0, \\ 0 & \text{otherwise.} \end{cases} \tag{213}$$

In fact, there are many applications where not only interested in predicted class labels, but where estimation of class-membership probability is particularly useful (output of sigmoid function prior to applying threshold function). Logistic regression is used in weather forecasting, e.g., not only to predict whether it will rain on a particular day, but also to report change of rain. Similarly, logistic regression can be used to predict chance that a patient has a particular disease given certain symptoms, which is why logistic regression enjoys great popularity in field of medicine.

∗ Learning model weights via logistic loss function. Have learned how we can use logistic regression model to predict probabilities & class labels; now, briefly talk about how we fit parameters of model, e.g., weights & bias unit $\mathbf{w}, b$. In prev chap, defined mean squared error loss functions as follows:

$$L(\mathbf{w}, b|\mathbf{x}) = \sum_i \frac{1}{2}(\sigma(z^{(i)}) - y^{(i)})^2. \tag{214}$$

Minimized this function in order to learn parameters for our Adaline classification model. To explain how we can derive loss function for logistic regression, 1st define likelihood $\mathcal{L}$ that we want to maximize when we build a logistic regression model, assuming: individual examples in our dataset are independent of 1 another. Formula is as follows:

$$\mathcal{L}(\mathbf{w}, b|\mathbf{x}) = p(y|\mathbf{x}; \mathbf{w}, b) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)}; \mathbf{w}, b) = \prod_{i=1}^{n} (\sigma(z^{(i)}))^{y(i)} (1 - \sigma(z^{(i)}))^{1-y^{(i)}}. \tag{215}$$

In practice, easier to maximize (natural) log of this equation, called *log-likelihood* function:

$$l(\mathbf{w}, b|\mathbf{x}) = \log \mathcal{L}(\mathbf{w}, b|\mathbf{x}) = \sum_{i=1}^{n} \left[y^{(i)} \log(\sigma(z^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(z^{(i)})) \right]. \tag{216}$$

1stly, applying log function reduces potential for numerical underflow, which can occur if likelihoods are very small. 2ndly, can convert product of factors into a summation of factors, which makes it easier to obtain derivative of this function via addition trick, as may remember from calculus.

Remark 37 (Deriving likelihood function)**.** *Can obtain expression for likelihood of model given data, $\mathcal{L}(\mathbf{w}, b|\mathbf{x})$ as follows. Given: have a binary classification problem with class labels 0 & 1, can think of label 1 as a Bernoulli variable – it can take on 2 values, 0 & 1, with probability p of being 1: $Y \sim \text{Bern}(p)$. For a single data point, can write this probability as $P(Y = 1|X = x^{(i)}) = \sigma(z^{(i)})$ & $P(Y = 0|X = x^{(i)}) = 1 - \sigma(z^{(i)})$.*
Putting these 2 expressions together, & using shorthand $P(Y = y^{(i)}|X = x^{(i)}) = p(y^{(i)}|x^{(i)})$, get probability mass function of Bernoulli variable:

$$p(y^{(i)}|x^{(i)}) = (\sigma(z^{(i)}))^{y^{(i)}} (1 - \sigma(z^{(i)}))^{1-y^{(i)}}. \tag{217}$$

Can write likelihood of training labels given assumption: all training examples are independent, using multiplication rule to compute probability that all events occur, as follows:

$$\mathcal{L}(\mathbf{w}, b|\mathbf{x}) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)}; \mathbf{w}, b). \tag{218}$$

Now, substituting probability mass function of Bernoulli variable, arrive at expression of likelihood, which we attempt to maximize by changing model parameters:

$$\mathcal{L}(\mathbf{w}, b|\mathbf{x}) = \prod_{i=1}^{n} (\sigma(z^{(i)}))^{y^{(i)}} (1 - \sigma(z^{(i)}))^{1-y^{(i)}}. \tag{219}$$

Could use an optimization algorithm e.g. gradient ascent to maximize this log-likelihood function. (Gradient ascent works exactly same way as gradient descent explained in Chap. 2, except: gradient ascent maximizes a function instead of minimizing it.) Alternatively, rewrite log-likelihood as a loss function L that can be minimized using gradient descent as in Chap. 2:

$$L(\mathbf{w}, b) = \sum_{i=1}^{n} \left[-y^{(i)} \log(\sigma(z^{(i)})) - (1 - y^{(i)}) \log(1 - \sigma(z^{(i)})) \right]. \tag{220}$$

To get a better grasp of this loss function, take a look at loss that we calculate for 1 single training example:

$$L(\sigma(z), y; \mathbf{w}, b) = -y \log(\sigma(z)) - (1 - y) \log(1 - \sigma(z)). \tag{221}$$

1st term becomes 0 if $y = 0$, & 2nd term becomes 0 if $y = 1$:

$$L(\sigma(z), y; \mathbf{w}, b) = \begin{cases} -\log(\sigma(z)) & \text{if } y = 1, \\ -\log(1 - \sigma(z)) & \text{if } y = 0. \end{cases} \tag{222}$$

Write a short code snippet to create a plot that illustrates loss of classifying a single training example for different values of $\sigma(z)$:

```
>>> def loss_1(z):
...     return - np.log(sigmoid(z))
>>> def loss_0(z):
...     return - np.log(1 - sigmoid(z))
>>> z = np.arange(-10, 10, 0.1)
>>> sigma_z = sigmoid(z)
>>> c1 = [loss_1(x) for x in z]
>>> plt.plot(sigma_z, c1, label='L(w, b) if y=1')
>>> c0 = [loss_0(x) for x in z]
>>> plt.plot(sigma_z, c0, linestyle='--', label='L(w, b) if y=0')
>>> plt.ylim(0.0, 5.1)
>>> plt.xlim([0, 1])
>>> plt.xlabel('$\sigma(z)$')
>>> plt.ylabel('L(w, b)')
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

Resulting plot shows sigmoid activation on x axis in range 0 to 1 (inputs to sigmoid function were $z \in [-10, 10]$) & associated logistic loss on y axis: Fig. 3.4: A plot of loss function used in logistic regression. Can see: loss approaches 0 (continuous line) if correctly predict that an example belongs to class 1. Similarly, can see on y axis: loss also approaches 0 if correctly predict $y = 0$ (dashed line). However, if prediction is wrong, loss $\rightarrow \infty$. Main point: penalize wrong predictions with an increasingly larger loss.

* Converting an Adaline implementation into an algorithm for logistic regression. If were to implement logistic regression ourselves, could simply substitute loss function L in our Adaline implementation from Chap. 2 with new loss function:

$$L(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} \left[-y^{(i)} \log(\sigma(z^{(i)})) - (1 - y^{(i)}) \log(1 - \sigma(z^{(i)})) \right]. \tag{223}$$

Use this to compute loss of classifying all training examples per epoch. Also, need to swap linear activation function with sigmoid. If make those changes to Adaline code, will end up with a working logistic regression implementation. Following is an implementation for full-batch gradient descent (but note: same changes could be made to stochastic gradient descent version as well): `[code]`.

When fit a logistic regression model, have to keep in mind: it only works for binary classification tasks.

So, consider only setosa & versicolor flowers (classes 0 & 1) & check: our implementation of logistic regression works:

```
>>> X_train_01_subset = X_train_std[(y_train == 0) | (y_train == 1)]
>>> y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]
>>> lrgd = LogisticRegressionGD(eta=0.3, n_iter=1000, random_state=1)
>>> lrgd.fit(X_train_01_subset, y_train_01_subset)
>>> plot_decision_regions(X=X_train_01_subset, y=y_train_01_subset, classifier=lrgd)
>>> plt.xlabel('Petal length [standardized]')
>>> plt.ylabel('Petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

Resulting decision region plot looks as Fig. 3.5: Decision region plot for logistic regression model.

Remark 38 (Gradient descent learning algorithm for logistic regression)**.** *If compared* `LogisticRegressionGD` *in prev code with* `AdalineGD` *code from Chap. 2, may have noticed: weight & bias update rules remained unchanged (except for scaling factor* 2*). Using calculus, can show: parameter updates via gradient descent are indeed similar for logistic regression & Adaline. However, note: following derivation of gradient descent learning rule is intended for readers who are interested in mathematical concepts behind gradient descent learning rule for logistic regression. Not essential for following rest of this chap.*

Fig. 3.6: Calculating partial derivative of log-likelihood function *summarizes how can calculate partial derivative of log-likelihood function w.r.t. jth weight: $\frac{\partial L}{\partial w_j} = \frac{\partial L}{\partial a} \frac{da}{dz} \frac{\partial z}{\partial w_j}$ where $a = \sigma(z) = \frac{1}{1+e^{-z}}$ .... Note: omitted averaging over training examples for brevity.*

Remember from Chap. 2: take steps in opposite direction of gradient. Hence, flip $\frac{\partial L}{\partial w_j} = -(y - a)x_j$ & update jth weight as follows, including learning rate η: $w_j \coloneqq w_j + \eta(y - a)x_j$. While partial derivative of loss function w.r.t. bias unit is not shown, bias derivation follows same overall concept using chain rule, resulting in following update rule: $b \coloneqq b + \eta(y - a)$. Both weight & bias unit updates are equal to the ones for Adaline in Chap. 2.

* Training a logistic regression model with scikit-learn. Just went through useful coding & math exercises in prev subsct, which helped to illustrate conceptual differences between Adaline & logistic regression. Learn how to use scikit-learn's more optimized implementation of logistic regression, which also supports multiclass settings off shelf. Note: in recent versions of scikit-learn, technique used for multiclass classification, multinomial, or OvR, is chosen automatically. In following code example, use `sklearn.linear_model.LogisticRegression` class as well as familiar `fit` method to train model on all 3 classes in standardized flower training dataset. Also, set `multi_class='ovr'` for illustration purposes. May want to compare results with `multi_class='multinomial'`. Note: `multinomial` setting is now default choice in scikit-learn's `LogisticRegression` class & recommended in practice for mutually exclusive classes, e.g. those found in Iris dataset. Here, "mutually exclusive" means: each training example can only belong to a single class (in contrast to multilabel classification, where a training example can be a member of multiple classes).

Code example:

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=100.0, solver='lbfgs', multi_class='ovr')
>>> lr.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined, classifier=lr, test_idx=range(105, 150))
>>> plt.xlabel('Petal length [standardized]')
>>> plt.ylabel('Petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

After fitting model on training data, plotted decision regions, training examples, & test examples, as shown in Fig. 3.7: Decision regions for scikit-learn's multiclass logistic regression model.

Remark 39 (Algorithms for convex optimization)**.** *Note: there exist many different algorithms for solving optimization problems. For minimizing convex loss functions, e.g. logistic regression loss, recommended to use more advanced approaches than regular* stochastic gradient descent (SGD)*. In fact, scikit-learn implements a whole range of such optimization algorithms, which can be specified via* `solver` *parameter, namely,* `'newton-cg'`, `'lbfgs'`, `'liblinear'`, `'sag'`, `'saga'`*. While logistic regression loss is convex, most optimization algorithms should converge to global loss minimum with ease. However, there are certain advantages of using 1 algorithm over other. E.g., in prev versions (e.g., v 0.21), scikit-learn used* `'liblinear'` *as a default, which cannot handle multinomial loss & is limited to OvR scheme for multiclass classification. However, in scikit-learn v 0.22, default solver was changed to* `'lbfgs'`*, which stands for limited-memory* Broyden–Fletcher–Goldfarb–Shanno (BFGS) *algorithm https://en.wikipedia.org/wiki/Limited-memory_BFGS & is more flexible in this regard.*

Looking at preceding code used to train `LogisticRegression` model, might now be wondering, "What is this mysterious parameter C?" Will discuss this parameter in next subsect, where introduce concepts of overfitting & regularization. However, before move on to those topics, finish discussion of class membership probabilities.

Probability that training examples belong to a certain class can be computed using `predict_proba` method. E.g., can predict probabilities of 1st 3 examples in test dataset as follows:

```
>>> lr.predict_proba(X_test_std[:3, :])
```

This code snippet returns following array:

```
array([[3.81527885e-09, 1.44792866e-01, 8.55207131e-01],
       [8.34020679e-01, 1.65979321e-01, 3.25737138e-13],
       [8.48831425e-01, 1.51168575e-01, 2.62277619e-14]])
```

1st row corresponds to class membership probabilities of 1st flower, 2nd row corresponds to class membership probabilities of 2nd flower, & so forth. Notice: column-wise sum in each row is 1, as expected. (Can confirm this by executing `lr.predict_proba(X_test_std[:3, :]).sum(axis=1)`.)

Highest value in 1st row is ≈ 0.85, i.e., 1st example belongs to class 3 `Iris-virginica` with a predicted probability of 85%. So, can get predicted class labels by identifying largest column in each row, e.g., using NumPy's `argmax` function:

```
>>> lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

Returned class indices are shown here (they correspond to `Iris-virginica, Iris-setosa, Iris-setosa`) (?):

```
array([2, 0, 0])
```

In preceding code example, computed conditional probabilities & converted these into class labels manually by using NumPy's `argmax` function. In practice, more convenient way of obtaining class labels when using scikit-learn: call `predict` method directly:

```
>>> lr.predict(X_test_std[:3, :])
array([2, 0, 0])
```

Lastly, a word of caution if want to predict class label of a single flower example: scikit-learn expects a 2D array as data input; thus, have to convert a single row slice into such a format 1st. 1 way to convert a single row entry into a 2D data array: use NumPy's `reshape` method to add a new dimension:

```
>>> lr.predict(X_test_std[0, :].reshape(1, -1))
array([2])
```

∗ Tackling overfitting via regularization. Overfitting is a common problem in ML, where a model performs well on training data but does not generalize well to unseen data (test data). If a model suffers from overfitting, also say: model has a high variance, which can be caused by having too many parameters, leading to a model that is too complex given underlying data. Similarly, our model can also suffer from *underfitting* (high bias), i.e., our model is not complex enough to capture pattern in training data well & therefore also suffers from low performance on unseen data.

Although have only encountered linear models for classification so far, problems of overfitting & underfitting can be best illustrated by comparing a linear decision boundary to more complex, nonlinear decision boundaries, as shown in Fig. 3.8: Examples of underfitted, well-fitted, & overfitted models.

Remark 40 (Bias-variance tradeoff)**.** *Often, researchers use terms "bias" & "variance" or "bias-variance tradeoff" to describe performance of a model – i.e., may stumble upon (vấp phải) talks, books, or articles where people say: a model has a "high variance" or "high bias." So, what does that mean? In general, might say: "high variance" is proportional to overfitting & "high bias" is proportional to underfitting.*

In context of ML models, variance measures consistency (or variability) of model prediction for classifying a particular example if train model multiple times, e.g., on different subsets of training dataset. Can say: model is sensitive to randomness in training data. In contrast, bias measures how far off predictions are from correct values in general if rebuild model multiple times on different training datasets; bias is measure of systematic error that is not due to randomness.

If interested in technical specification & derivation of "bias" & "variance" terms, see lecture notes https://sebastianraschka.com/pdf/lecture-notes/stat451fs20/08-model-eval-1-intro_notes.pdf .

1 way of finding a good bias-variance tradeoff: tune complexity of model via regularization. Regularization is a very useful method for handling collinearity (high correlation among features), filtering out noise from data, & eventually preventing overfitting.

Concept behind regularization: introduce additional information to penalize extreme parameter (weight) values. Most common form of regulariozation is so-called *L2 regularization* (sometimes also called L2 shrinkage or weight decay), which can be written as follows:

$$\frac{\lambda}{2n}\|\mathbf{w}\|^2 = \frac{\lambda}{2n}\sum_{i=1}^{m} w_i^2. \tag{224}$$

Here λ: so-called *regularization parameter*. Note: 2 in denominator is merely a scaling factor, s.t. it cancels when computing loss gradient. Sample size n is added to scale regularization term similar to loss.

Remark 41 (Regularization & feature normalization)**.** *Regularization is another reason why feature scaling e.g. standardization is important. For regularization to work properly, need to ensure: all our features are on comparable scales.*

Loss function for logistic regression can be regularized by adding a simple regularization term, which will shrink weights during model training:

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left[-y^{(i)}\log(\sigma(z^{(i)})) - (1-y^{(i)})\log(1-\sigma(z^{(i)}))\right] + \frac{\lambda}{2n}\|\mathbf{w}\|^2. \tag{225}$$

Partial derivative of unregularized loss is defined as:

$$\frac{\partial L(\mathbf{w}, b)}{\partial w_j} = \frac{1}{n}\sum_{i=1}^{n}\left(\sigma(\mathbf{w}^\top\mathbf{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}. \tag{226}$$

Adding regularization term to loss changes partial derivative to following form:

$$\frac{\partial L(\mathbf{w}, b)}{\partial w_j} = \left(\frac{1}{n}\sum_{i=1}^{n}\left(\sigma(\mathbf{w}^\top\mathbf{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}\right) + \frac{\lambda}{n}w_j. \tag{227}$$

Via regularization parameter λ, can then control how closely we fit training data, while keeping weights small. By increasing value of λ, increase regularization strength. Note: bias unit, which is essentially an intercept term or negative threshold, as learned in Chap. 2, is usually not regularized.

Parameter `C`, implemented for `LogisticRegression` class in scikit-learn comes from a convention in support vector machines. Term `C` is inversely proportional to regularization parameter λ. Consequently, decreasing value of inverse regularization parameter `C` means: increasing regularization strength, which can visualize by plotting L2 regularization path for 2 weight coefficients:

```
>>> weights, params = [], []
>>> for c in np.arange(-5, 5):
...     lr = LogisticRegression(C=10.**c,
multi_class='ovr')
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10.**c)
>>> weights = np.array(weights)
>>> plt.plot(params, weights[:, 0], label='Petal length')
>>> plt.plot(params, weights[:, 1], linestyle='--', label='Petal width')
>>> plt.ylabel('Weight coefficient')
>>> plt.xlabel('C')
>>> plt.legend(loc='upper left')
>>> plt.xscale('log')
>>> plt.show()
```

By executing preceding code, fitted 10 logistic regression models with different values for inverse-regularization parameter `C`. For illustration purposes, only collected weight coefficients of class 1 (here, 2nd class in datast: `Iris-versicolor`) vs. all classifiers – remember: we are using OvR technique for multiclass classification.

As can see in resulting plot, weight coefficients shrink if decrease parameter `C`, i.e., if increase regularization strength: Fig. 3.9: Impact of inverse regularization strength parameter C on L2 regularized model results.

Increasing regularization strength can reduce overfitting, so might ask why don't strongly regularize all models by default. Reason: have to be careful when adjusting regularization strength. E.g., if regularization strength is too high & weights coefficients approach 0, model can perform very poorly due to underfitting, as illustrated in Fig. 3.8.

Remark 42 (An additional resource on logistic regression)**.** *Since in-depth coverage of individual classification algorithms exceeds scope of this book, Logistic Regression: From Introductory to Advanced Concepts & Applications, Dr.* Scott Menard*, Sage Publications, 2009, is recommended to readers who want to learn more about logistic regression.*

- Maximum margin classification with support vector machines. Another powerful & widely used learning algorithm is *support vector machine (SVM)*, which can be considered an extension of perceptron. Using perceptron algorithm, minimized misclassification errors. However, in SVMs, our optimization objective: maximize margin. Margin is defined as distance between separating hyperplane (decision boundary) & training examples that are closest to this hyperplane, which are so-called *support vectors*. Illustrated in Fig. 3.10: SVM maximizes margin between decision boundary & training data points.
  - Maximum margin intuition. Rationale behind having decision boundaries with large margins: they tend to have a lower generalization error, whereas models with small margins are more prone to overfitting.
    Unfortunately, while main intuition behind SVMs is relatively simple, mathematics behind them is quite advanced & would require sound knowledge of constrained optimization.
    Hence, details behind maximum margin optimization SVMs are beyond scope of this book. However, recommend following resources if interested in learning more:
    - Chris J.C. Burges's excellent explanation in *A Tutorial on Support Vector Machines for Pattern Recognition* (Data Mining & Knowledge Discovery, 2(2): 121-167, 1998)
    - Vladimir Vapnik's book *The Nature of Statistical Learning Theory*,Springer Science+Business Media, Vladimir Vapnik, 2000
    - Andrew Ng's very detailed lecture notes at https://see.stanford.edu/materials/aimlcs229/cs229-notes3.pdf.
  - Dealing with a nonlinearly separable case using slack variables. (biến lỏng lẻo) Although don't want to dive much deeper into more involved mathematical concepts behind maximum-margin classification, briefly mention so-called *slack variable*, which was introduced by Vladimir Vapnik in 1995 & led to so-called *soft-margin classification*. Motivation for introducing slack variable was: linear constraints in SVM optimization objective need to be relaxed for nonlinearly separable data to allow convergence of optimization in presence of misclassifications, under appropriate loss penalization.
    Use of slack variable, in turn, introduces variable, which is commonly referred to as C in SVM contexts. Can consider C as a hyperparameter for controlling penalty for misclassification. Large values of C corresponds to large error penalties, whereas less strict about misclassification errors if choose smaller values for C. Can then use C parameter to control width of margin & therefore tune bias-variance tradeoff, as illustrated in Fig. 3.11: Impact of large & small values of inverse regularization strength C on classification.
    This concept is related to regularization, discussed in prev sect in context of regularized regression, where decreasing value of C increases bias (underfitting) & lowers variance (overfitting) of model.
    Have learned basic concepts behind a linear SVM, train an SVM mode to classify different flowers in Iris dataset:

```
>>> from sklearn.svm import SVC
    >>> svm = SVC(kernel='linear', C=1.0, random_state=1)
    >>> svm.fit(X_train_std, y_train)
    >>> plot_decision_regions(X_combined_std, y_combined, classifier=svm, test_idx=range(105, 150))
    >>> plt.xlabel('Petal length [standardized]')
    >>> plt.ylabel('Petal width [standardized]')
    >>> plt.legend(loc='upper left')
    >>> plt.tight_layout()
    >>> plt.show()
```

    3 decision regions of SVM, visualized after training classifier on Iris dataset by executing preceding code example, are shown in Fig. 3.12: SVM's decision regions.

    Remark 43 (Logistic regression vs. SVMs)**.** *In practical classification tasks, linear logistic regression & linear SVMs often yield very similar results. Logistic regression tries to maximize conditional likelihoods of training data, which makes it more prone to outliers than SVMs, which mostly care about points that are closest to decision boundary (support vectors). On other hand, logistic regression has advantage of being a simpler model & can be implemented more easily, & is mathematically easier to explain. Furthermore, logistic regression models can be easily updated, which is attractive when working with streaming data.*
  - Alternative implementations in scikit-learn. scikit-learn library's `LogisticRegression` class can make use of LIBLINEAR library by setting `solver='liblinear'`. LIBLINEAR is a highly optimized C/C++ library developed at National Taiwan University http://www.csie.ntu.edu.tw/~cjlin/liblinear/.
    Similar, SVC class used to train an SVM makes use of LIBSVM, which is an equivalent C/C++ library specialized for SVMs http://www.csie.ntu.edu.tw/~cjlin/libsvm/.
    Advantage of using LIBLINEAR & LIBSVM over, e.g., native Python implementations: they allow extremely quick training of large amounts of linear classifiers. However, sometimes our datasets are too large to fit into computer memory. Thus, scikit-learn also offers alternative implementations via `SGDClassifier` class, which also supports online learning via `partial_fit` method. Concept behind `SGDClassifier` class is similar to stochastic gradient algorithm implemented in Chap. 2 for Adaline.
    Could initialize SGD version of perceptron (`loss='perceptron'`), logistic regression (`loss='log'`), & an SVM with default parameters (`loss='hinge'`) (bản lề), as follows:

```
>>> from sklearn.linear_model import SGDClassifier
>>> ppn = SGDClassifier(loss='perceptron')
>>> lr = SGDClassifier(loss='log')
```

```
>>> svm = SGDClassifier(loss='hinge')
```

- Solving nonlinear problems using a kernel SVM. Another reason why SVMs enjoy high popularity among ML practitioners: they can be easily *kernelized* to solve nonlinear classification problems. Before discuss main concept behind so-called *kernel SVM*, most common variant of SVMs, 1st create a synthetic dataset to see what such a nonlinear classification problem may look like.
  - Kernel methods for linearly inseparable data.
  - Using kernel trick to find separating hyperplanes in a high-dimensional space.
- Decision tree learning.
- K-nearest neighbors – a lazy learning algorithm.
- Summary.

- Chap. 4: Building Good Training Datasets – Data Preprocessing.
  - Dealing with missing data.
  - Handling categorical data.
  - Partitioning a dataset into separate training & test datasets.
  - Bringing features onto same scale.
  - Selecting meaningful features.
  - Assessing feature importance with random forests.
  - Summary.
- Chap. 5: Compressing Data via Dimensionality Reduction.
  - Unsupervised dimensionality reduction via principal component analysis.
  - Supervised data compression via linear discriminant analysis.
  - Nonlinear dimensionality reduction & visualization.
  - Summary.
- Chap. 6. Learning Best Practices for Model Evaluation & Hyperparameter Tuning.
  - Streamlining workflows with pipelines.
  - Using k-fold cross-validation to assess model performance.
  - Debugging algorithms with learning & validation curves.
  - Fine-tuning ML models via grid search.
  - Looking at different performance evaluation metrics.
  - Summary.
- Chap. 7: Combining Different Models for Ensemble Learning.
  - Learning with ensembles.
  - Combining classifiers via majority vote.
  - Bagging – building an ensemble of classifiers from bootstrap samples.
  - Leveraging weak learners via adaptive boosting.
  - Gradient boosting – training an ensemble based on loss gradients.
  - Summary.
- Chap. 8: Applying ML to Sentiment Analysis.
  - Preparing IMDb movie review data for text processing.
  - Introducing bag-of-words model.
  - Training a logistic regression model for document classification.
  - Working with bigger data – online algorithms & out-of-core learning.
  - Topic modeling with latent Dirichlet allocation.
  - Summary.
- Chap. 9: Predicting Continuous Target Variables with Regression Analysis.
  - Introducing linear regression.

- Exploring Ames Housing dataset.
- Implementing an ordinary least squares linear regression model.
- Fitting a robust regression model using RANSAC.
- Evaluating performance of linear regression models.
- Using regularized methods for regression.
- Turning a linear regression model into a curve – polynomial regression.
- Dealing with nonlinear relationships using random forests.
- Summary.

- Chap. 10: Working with Unlabeled Data – Clustering Analysis.
  - Grouping objects by similarity using k-means.
  - Organizing clusters as a hierarchical tree.
  - Locating regions of high density via DBSCAN.
  - Summary.

- Chap. 11: Implementing a Multilayer Artificial Network Network from Scratch. DL is getting a lot of attention from press & is, without doubt, hottest topic in ML field. DL can be understood as a subfield of ML that is concerned with training artificial neural networks (NNs) with many layers efficiently. In this chap, learn basic concepts of artificial NNs so that you are well equipped for following chaps, which will introduce advanced Python-based DL libraries & *deep neural network* (DNN) architectures that are particularly well suited for image & text analyses. Topics covered:
  - Gaining a conceptual understanding of multilayer NNs
  - Implementing fundamental backpropagation algorithm for NN training from scratch
  - Training a basic multilayer NN for image classification
  - Modeling complex functions with artificial neural networks. At beginning of this book, started journey through ML algorithms with artificial neurons in Chap. 2. Artificial neurons represent building blocks of multilayer artificial NNs. Basic concept behind artificial NNs was built upon hypotheses & models of how human brain works to solve complex problem tasks. Although artificial NNs have gained a lot of popularity in recent years, early studies of NNs go back to 1940s, when Warren McCulloch & Walter Pitts 1st described how neurons could work. *A logical calculus of the ideas immanent in nervous activity* W. S. McCulloch & W. Pitts, The Bulletin of Mathematical Biophysics, 5(4):115–133, 1943.

    However, in decades that followed 1st implementation of *McCulloch-Pitts neuron* model – Rosenblatt's perceptron in 1950s – many researchers & ML practitioners slowly began to lose interest in NNs since no one had a good solution for training an NN with multiple layers. Eventually, interest in NNs was rekindled in 1986 when D.E. Rumelhart, G.E. Hinton, R.J. Williams were involved in (re)discovery & popularization of backpropagation algorithm to train NNs more efficiently: *Learning representations by backpropagating errors*, by D.E. Rumelhart, G.E. Hinton, & R.J. Williams, Nature, 323 (6088): 533–536, 1986. Readers who are interested in history of AI, ML, & NNs are also encouraged to read Wikipedia article on so-called *AI winters*, which are periods of time where a large portion of research community lost interest in study of NNs https://en.wikipedia.org/wiki/AI_winter.

    However, NNs are more popular today than ever thanks to many breakthroughs that have been made in prev decade, which resulted in what now call DL algorithms & architectures – NNs that are composed of many layers. NNs are a hot topic not only in academic research but also in big technology companies, e.g. Facebook, Microsoft, Amazon, Uber, Google, & many more that invest heavily in artificial NNs & DL research.

    As of today, complex NNs powered by DL algorithms are considered state-of-art solutions for complex problem solving e.g. image & voice recognition. Some of recent applications include:

    * Predicting COVID-19 resource needs from a series of X-rays https://arxiv.org/abs/2101.04909
    * Modeling virus mutations https://science.sciencemag.org/content/371/6526/284
    * Leveraging data from social media platforms to manage extreme weather events https://onlinelibrary.wiley.com/doi/abs/10.1111/1468-5973.12311
    * Improving photo descriptions for people who are blind or visually impaired https://tech.fb.com/how-facebook-is-using-ai-
    * Single-layer neural network recap. This chap is all about multilayer NNs, how they work, & how to train them to solve complex problems. However, before dig deeper into a particular multilayer NN architecture, briefly reiterate some of concepts of single-layer NNs introduced in Chap. 2, namely, *ADAptive LInear NEuron (Adaline)* algorithm, shown in Fig. 11.1: Adaline algorithm.

      In Chap. 2, implemented Adaline algorithm to perform binary classification, & used gradient descent optimization algorithm to learn weight coefficients of model. In every epoch (pass over training dataset), updated weight vector $\mathbf{w}$ & bias unit b using following update rule: $\mathbf{w} := \mathbf{w} + \Delta\mathbf{w}, b := b + \Delta b$ where $\Delta w_i = -\eta\frac{\partial L}{\partial w_i}, \Delta b = -\eta\frac{\partial L}{\partial b}$ for bias unit & each weight w_i in weight vector $\mathbf{w}$.

I.e., computed gradient based on whole training dataset & updated weights of model by taking a step in opposite direction of loss gradient $\nabla L(\mathbf{w})$. (For simplicity, focus on weights & omit bias unit in following paragraphs; however, as remember from Chap. 2, same concepts apply.) In order to find optimal weights of model, optimized objective function defined as *mean of square errors (MSE)* loss function $L(\mathbf{w})$. Furthermore, multiplied gradient by a factor, *learning rate* η, which had to choose carefully to balance speed of learning against risk of overshooting global minimum of loss function.
In gradient descent optimization, updated all weights simultaneously after each epoch, & defined partial derivative for each weight w_i in weight vector $\mathbf{w}$ as follows:

$$\frac{\partial L}{\partial w_j}\frac{1}{n}\sum_i (y^{(i)} - a^{(i)})^2 = -\frac{2}{n}\sum_i (y^{(i)} - a^{(i)})x_j^{(i)}. \tag{228}$$

Here $y^{(i)}$: target class label of a particular sample $x^{(i)}$, & $a^{(i)}$: activation of neuron, which is a linear function in special case of Adaline.
Furthermore, defined activation function $\sigma(\cdot)$ as follows $\sigma(\cdot) = z = a$. Here, net input z is a linear combination of weights that are connecting input layer to output layer: $z = \sum_i w_i x_i + b = \mathbf{w}^\top \mathbf{x} + b$. While used activation $\sigma(\cdot)$ to compute gradient update, implemented a threshold function to squash continuous-valued output into binary class labels for prediction:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{229}$$

Remark 44 (Single-layer naming convention)**.** *Although Adaline consists of 2 layer, 1 input layer & 1 output layer, it is called a single-layer network because of its single link between input & output layers.*
Also, learned about a certain *trick* to accelerate model learning, so-called *stochastic gradient-descent (SGD)* optimization. SGD approximates loss from a single training sample (online learning) or a small subset of training examples (mini-batch learning). Make use of this concept later in this chap when implement & train a *multilayer perceptron (MLP)*. Apart from faster learning – due to more frequent weight updates compared to gradient descent – its noisy nature is also regarded as beneficial when training multilayer NNs with nonlinear activation functions, which do not have a convex loss function. Here, added noise can help to escape local loss minima.

∗ Introducing multilayer neural network architecture. In this sect, learn how to connect multiple single neurons to a multilayer feedforward NN, this special type of *fully connected* network is also called *MLP*.
Fig. 11.2: A 2-layer MLP illustrates concept of an MLP consisting of 2 layers. Next to data input, MLP depicted in Fig. 11.2 has 1 hidden layer & 1 output layer. Units in hidden layer are fully connected to input features, & output layer is fully connected to hidden layer. If such a network has > 1 hidden layer, also call it a *deep NN*. (Note: in some contexts, inputs are also regarded as a layer. However, in this case, it would make Adaline model, which is a single-layer neural network, a 2-layer neural network, which may be counterintuitive.)

Remark 45 (Adding additional hidden layers)**.** *Can add any number of hidden layers to MLP to create deeper network architectures. Practically, can think of number of layers & units in an NN as additional hyperparameters that we want to optimize for a given problem task using cross-validation technique, discussed in Chap. 6.*
However, loss gradients for updating network's parameters, which will calculate later via backpropagation, will become increasingly small as more layers are added to a network. This vanishing gradient problem makes model learning more challenging. Therefore, special algorithms have been developed to help train such DNN structures; known as deep learning.

As shown in Fig. 11.2, denote ith activation unit in lth layer as $a_i^{(l)}$. To make math & code implementations a bit more intuitive, will not use numerical indices to refer to layers, but will use *in* superscript for input features, h superscript for hidden layer, & *out* superscript for output layer. E.g., $x_i^{(in)}$ refers to ith input feature value, $a_i^{(h)}$ refers to ith unit in hidden layer, & $a_i^{(out)}$ refers to ith unit in output layer. Note: $\mathbf{b}$'s in Fig. 11.2 denote bias units. In fact, $\mathbf{b}^{(h)}, \mathbf{b}^{(out)}$ are vectors with number of elements = number of nodes in layer they correspond to. E.g., $\mathbf{b}^{(h)}$ stores d bias units, where d: number of nodes in hidden layer. If this sounds confusing, don't worry. Looking at code implementation later, where initialize weight matrices & bias unit vectors, will help clarify these concepts.
Each node in layer l is connected to all nodes in layer $l+1$ via a weight coefficient. E.g., connection between kth unit in layer l to jth unit in layer $l+1$ will be written as $w_{i,k}^{(l+1)}$. Referring back to Fig. 11.2, denote weight matrix that connects input to hidden layer as $\mathbf{W}^{(h)}$, & write matrix that connects hidden layer to output layer as $\mathbf{W}^{(out)}$.
While 1 unit in output layer would suffice for a binary classification task, saw a more general form of an NN in preceding figure, which allows us to perform multiclass classification via a generalization of *1-vs-all (OvA)* technique. To better understand how this works, remember *1-hot* representation of categorical variables introduced in Chap. 4.
E.g., can encode 3 class labels in familiar Iris dataset (0=Setosa, 1 = Versicolor, 2=Virginica) as follows:

$$0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \; 1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \; 2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{230}$$

This 1-hot vector representation allows us to tackle classification tasks with an arbitrary number of unique class labels present in training dataset.
If new to NN representations, indexing notation (subscripts & superscripts) may look a little bit confusing at 1st. What may seem overly complicated at 1st will make much more sense in later sects when vectorize NN representation. Summarize

weights that connect input & hidden layers by a $d \times m$ dimensional matrix $\mathbf{W}^{(h)}$, where d: number of hidden units & m: number of input units.

* Activating a neural network via forward propagation. Describe process of *forward propagation* to calculate output of an MLP model. To understand how it fits into context of learning an MLP model, summarize MLP learning procedure in 3 simple steps:

  1. Starting at input layer, forward propagate patterns of training data through network to generate an output.
  2. Based on network's output, calculate loss that want to minimize using a loss function described later.
  3. Backpropagate loss, find its derivative w.r.t. each weight & bias unit in network, & update model.

  After repeat these 3 steps for multiple epochs & learn weight & bias parameters of MLP, use forward propagation to calculate network output & apply a threshold function to obtain predicted class labels in 1-hot representation, described in prev sect.

  Walk through individual steps of forward propagation to generate an output from patterns in training data. Since each unit in hidden layer is connected to all units in input layers, 1st calculate activation unit of hidden layer $a_1^{(h)}$ as follows:

$$z_1^{(h)} = x_1^{(in)} w_{1,1}^{(h)} + x_2^{(in)} w_{1,2}^{(h)} + \cdots + x_m^{(in)} w_{1,m}^{(h)},$$
$$a_1^{(h)} = \sigma(z_1^{(h)}).$$

  Here $z_1^{(h)}$: net input & $\sigma(\cdot)$: activation function, which has to be differentiable to learn weights that connect neurons using a gradient-based approach. To be able to solve complex problems e.g. image classification, need nonlinear activation functions in our MLP model, e.g., sigmoid (logistic) activation function that remember from sect about logistic regression in Chap. 3:

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{231}$$

  Sigmoid function is an S-shaped curve that maps net input z onto a logistic distribution in range 0 to 1, which cuts y axis at $z = 0$, as shown in Fig. 11.3: Sigmoid activation function.

  MLP is a typical example of a feedforward artificial NN. Term *feedforward* refers to fact: each layer serves as input to next layer without loops, in contrast to recurrent NNs – an architecture discussed in this chap & discussed in more detail in Chap. 15. Term *multilayer perceptron* may sound a little bit confusing since artificial neurons in this network architecture are typically sigmoid units, not perceptrons. Can think of neurons in MLP as logistic regression units that return values in continuous range between 0 & 1.

  For purposes of code efficiency & readability, now write activation in a more compact form using concepts of basic linear algebra, which will allow us to vectorize our code implementation via NumPy rather than writing multiple nested & computationally expensive Python `for` loops:

$$\mathbf{z}^{(h)} = \mathbf{x}^{(in)} \mathbf{W}^{(h)\top} + \mathbf{b}^{(h)}, \tag{232}$$
$$a^{(h)} = \sigma(\mathbf{z}^{(h)}). \tag{233}$$

  Here $\mathbf{x}^{(in)}$: our $1 \times m$ dimensional feature vector. $\mathbf{W}^{(h)}$: a $d \times m$ dimensional weight matrix where d: number of units in hidden layer; consequently, transposed matrix $\mathbf{W}^{(h)\top}$: $m \times d$ dimensional. Bias vector $\mathbf{b}^{(h)}$ consists of d bias units (1 bias unit per hidden node).

  After matrix-vector multiplication, obtain $1 \times d$ dimensional net input vector $\mathbf{z}^{(h)}$ to calculate activation $\mathbf{a}^{(h)} \in \mathbb{R}^{1 \times d}$.

  Furthermore, can generalize this computation to all n examples in training dataset:

$$Z^{(h)} = \mathbf{X}^{(in)} \mathbf{W}^{(h)\top} + \mathbf{b}^{(h)}. \tag{234}$$

  Here $\mathbf{X}^{(in)}$ is now an $n \times m$ matrix, & matrix multiplication will result in an $n \times d$ dimensional net input matrix $Z^{(h)}$.

  Finally, apply activation function $\sigma(\cdot)$ to each value in net input matrix to get $n \times d$ activation matrix in next layer (here, output layer):

$$A^{(h)} = \sigma(Z^{(h)}). \tag{235}$$

  Similarly, can write activation of output layer in vectorized form for multiple examples:

$$Z^{(out)} = A^{(h)} W^{(out)\top} + b^{(out)}. \tag{236}$$

  Here, multiply transpose of $t \times d$ matrix $W^{(out)}$ (t: number of output units) by $n \times d$ dimensional matrix $A^{(h)}$ & add t dimensional bias vector $\mathbf{b}^{(out)}$ to obtain $n \times t$ dimensional matrix $Z^{(out)}$. (Rows in this matrix represent outputs for each example.)

  Lastly, apply sigmoid activation function to obtain continuous-valued output of network:

$$A^{(out)} = \sigma(Z^{(out)}). \tag{237}$$

  Similar to $Z^{(out)}$, $A^{(out)}$ is an $n \times t$ dimensional matrix.

* Classifying handwritten digits. In prev sect, covered a lot of theory around NNs, which can be a little bit overwhelming if new to this topic. Before continue with discussion of algorithm for learning weights of MLP model, backpropagation, take a short break from theory & see an NN in action.

Remark 46 (Additional resources on backpropagation)**.** *NN theory can be quite complex; thus, want to provide readers with additional resources that cover some of topics discussed in this chap in more detail or from a different perspective:*

* *Chap. 6: Deep Feedforward Networks, DL, by* I. Goodffellow, Y. Bengio, A. Courville*, MIT Press, 2016 (manuscripts freely accessible at http://www.deeplearningbook.org).*
* *Pattern Recognition & ML by* C. M. Bishop*, Springer New York, 2006.*
* *Lecture video slides from* Sebastian Raschka*'s DL course: https://sebastianraschka.com/blog/2021/dl-course.html#l08-multinomial-logistic-regression--softmax-regression https://sebastianraschka.com/blog/2021/dl-course.html#l09-multilayer-perceptrons-and-backpropration*

In this sect, implement & train 1st multilayer NN to classify handwritten digits from popular *Mixed National Institute of Standards & Technology (MNIST)* dataset that has been constructed by Yann LeCun & others & serves as a popular benchmark dataset for ML algorithms *Gradient-Based Learning Applied to Document Recognition* by Y. LeCun, L. Bottou, Y. Bengio, & P. Haffner Proceedings of the IEEE, 86(11): 2278-2324, 1998.

* Obtaining & preparing MNIST dataset. MNIST dataset is publicly available at http://yann.lecun.com/exdb/mnist/ & consists of following 4 parts:
  1. *Training dataset images*: `train-images-idx3-ubyte.gz` (9.9 MB, 47 MB unzipped, & 60,000 examples)
  2. *Training dataset labels*: `train-labels-idx1-ubyte.gz` (29 KB, 60 KB unzipped, & 60,000 labels)
  3. *Test dataset images*: `t10k-images-idx3-ubyte.gz` (1.6 MB, 7.8 MB unzipped, & 10,000 examples)
  4. *Test dataset labels*: `t10k-labels-idx1-ubyte.gz` (5 KB, 10 KB unzipped, & 10,000 labels)

  MNIST dataset was constructed from 2 datasets of US *National Institute of Standards & Technology (NIST)*. Training dataset consists of handwritten digits from 250 different people, 50% high school students, & 50% employees from Census Bureau. Note: test dataset contains handwritten digits from different people following same split.
  Instead of downloading abovementioned dataset files & preprocessing them into NumPy arrays ourselves, use scikit-learn's new `fetch_openml` function, which allows us to load MNIST dataset more conveniently:

```
>>> from sklearn.datasets import fetch_openml
>>> X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
>>> X = X.values
>>> y = y.astype(int).values
```

  In scikit-learn, `fetch_openml` function downloads MNIST dataset from OpenML https://www.openml.org/d/554 as pandas `DataFrame` & Series objects, which is why use `.values` attribute to obtain underlying NumPy arrays. (If using a scikit-learn version older than 1.0, `fetch_openml` downloads NumPy arrays directly so can omit using `.values` attribute.) $n \times m$ dimensional `X` array consists of 70000 images with 784 pixels each, & `y` array stores corresponding 70000 class labels, which can confirm by checking dimensions of arrays as follows:

```
>>> print(X.shape)
(70000, 784)
>>> print(y.shape)
(70000,)
```

  Images in MNIST dataset consist of 28×28 pixels, & each pixel is represented by a grayscale intensity value. Here `fetch_openml` already unrolled 28×28 pixels into 1D row vectors, which represent rows in our `X` array (784 per row or image) above. 2nd array `y` returned by `fetch_openml` function contains corresponding target variable, & class labels (integers 0-9) of handwritten digits.
  Next, normalize pixels values in MNIST to range -1 to 1 (originally 0 to 255) via following code line:

```
>>> X = ((X / 255.) - .5) * 2
```

  Reason behind this: gradient-based optimization is much more stable under these conditions, as discussed in Chap. 2. Note: scaled images on a pixel-by-pixel basis, which is different from feature-scaling approach that took in prev chaps. Previously, derived scaling parameters from training dataset & used these to scale each column in training dataset & test dataset. However, when working with image pixels, centering them at 0 & rescaling them to a $[-1, 1]$ range is also common & usually works well in practice. To get an idea of how those images in MNIST look, visualize examples of digits 0-9 after reshaping 784-pixel vectors from our feature matrix into original 28×28 image that we can plot via Matplotlib's `imshow` function:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(nrows=2, ncols=5, sharex=True, sharey=True)
>>> ax = ax.flatten()
>>> for i in range(10):
...     img = X[y == i][0].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
```

```
>>> plt.show()
```

Should now see a plot of 2×5 subfigures showing a representative image of each unique digit: Fig. 11.4: A plot showing 1 randomly chosen handwritten digit from each class.
In addition, also plot multiple examples of same digit to see how different handwriting for each really is:

```
>>> fig, ax = plt.subplots(nrows=5, ncols=5,  sharex=True, sharey=True)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = X[y == 7][i].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

After executing code, should see 1st 25 variants of digit 7: Fig. 11.5: Different variants of handwritten digit 7.
Finally, divide dataset into training, validation, & test subsets. Following code will split dataset s.t. 55000 images are used for training, 5000 images for validation, & 10000 images for testing:

```
>>> from sklearn.model_selection import train_test_split
>>> X_temp, X_test, y_temp, y_test = train_test_split(X, y, test_size=10000, random_state=123, stratif
>>> X_train, X_valid, y_train, y_valid = train_test_split(X_temp, y_temp, test_size=5000, random_state
```

∗ Implementing a multilayer perceptron. Now implement an MLP from scratch to classify images in MNIST dataset. To keep things simple, implement an MLP with only 1 hidden layer. Since approach may seem a little bit complicated at 1st, encouraged to download sample code for this chap from Packt Publishing website or from GitHub https://github.com/rasbt/machine-learning-book so that can view this MLP implementation annotated with comments & syntax highlighting for better readability.
If not running code from accompanying Jupyter Notebook file or don't have access to internet, copy `NeuralNetMLP` code from this chap into a Python script file in current working directory (e.g., `neuralnet.py`), which can then import into current Python session via following command:

```
from neuralnet import NeuralNetMLP
```

Code will contain parts that we have not talked about yet, e.g. backpropagation algorithm. Do not worry if not all code makes immediate sense; will follow up on certain parts later in this chap. However, going over code at this stage can make it easier to follow theory later.
Look at following implementation of an MLP, starting with 2 helper functions to compute logistic sigmoid activation & to convert integer class label arrays to 1-hot encoded labels:

```
import numpy as np
def sigmoid(z):
    return 1. / (1. + np.exp(-z))
def int_to_onehot(y, num_labels):
    ary = np.zeros((y.shape[0], num_labels))
    for i, val in enumerate(y):
        ary[i, val] = 1
    return ary
```

Below, implement main class for our MLP, called `NeuralNetMLP`. There are 3 class methods `.__init__()`, `.forward()`, `.backwa` will be discussed 1 by 1, starting with `__init__` constructor [code]. `__init__` constructor instantiates weight matrices & bias vectors for hidden & output layer. Next, see how these are used in `forward` method to make predictions:

```
def forward(self, x):
    # Hidden layer
    # input dim: [n_examples, n_features] dot [n_hidden, n_features].T
    # output dim: [n_examples, n_hidden]
    z_h = np.dot(x, self.weight_h.T) + self.bias_h
    a_h = sigmoid(z_h)

    # Output layer
    # input dim: [n_examples, n_hidden] dot [n_classes, n_hidden].T
    # output dim: [n_examples, n_classes]
    z_out = np.dot(a_h, self.weight_out.T) + self.bias_out
    a_out = sigmoid(z_out)
    return a_h, a_out
```

`forward` method takes in 1 or more training examples & returns predictions. In fact, it returns both activation values from hidden layer & output layer `a_h,a_out`. While `a_out` represents class-membership probabilities that can convert to class labels, which we care about, also need activation values from hidden layer `a_h` to optimize model parameters; i.e., weight & bias units of hidden & output layers.
Finally, talk about `backward` method, which updates weight & bias parameters of neural network `[code]`.
`backward` method implements so-called *backpropagation* algorithm, which calculates gradients of loss w.r.t. weight & bias parameters. Similar to Adaline, these gradients are then used to update these parameters via gradient descent. Note: multilayer NNs are more complex than their single-layer siblings, & will go over mathematical concepts of how to compute gradients in a later sect after discussing code. For now, just consider `backward` method as a way for computing gradients that are used for gradient descent updates. For simplicity, loss function this derivation is based on is same MSE loss used in Adaline. In later chaps, look at alternative loss functions, e.g. multi-category cross-entropy loss, which is a generalization of binary logistic regression loss to multiple classes.
Looking at this code implementation of `NeuralNetMLP` class, may have noticed: this object-oriented implementation differs from familiar scikit-learn API that is centered around `.fit(), .predict()` methods. Instead, main methods of `NeuralNetMLP` class are `.forward(), .backward()` methods. 1 of reasons behind this: it makes a complex neural network a bit easier to understand in terms of how information flows through networks.
Another reason: this implementation is relatively similar to how more advanced DL libraries e.g. PyTorch operate, which will introduce & use in upcoming chaps to implement more complex neural networks.
After have implemented `NeuralNetMLP` class, use following code to instantiate a new `NeuralNetMLP` object:

```
>>> model = NeuralNetMLP(num_features=28*28, num_hidden=50, num_classes=10)
```

`model` accepts MNIST images reshaped into 784-dimensional vectors (in format of `X_train, X_valid, X_test`, defined previously) for 10 integer classes (digits 0-9). Hidden layer consists of 50 nodes. Also, as may be able to tell from looking at previously defined `.forward()` method, use a sigmoid activation function after 1st hidden layer & output layer to keep things simple. In later chaps, learn about alternative activation functions for both hidden & output layers.
Fig. 11.6: NN architecture for labeling handwritten digits summarizes neural network architecture instantiated above. In next subsect, implement training function that we can use to train network on mini-batches (lô nhỏ) of data via backpropagation.

  - Coding neural network training loop.
  - Evaluating neural network performance.
- Training an artificial neural network.
  - Computing loss function.
  - Developing your understanding of backpropagation.
  - Training neural networks via backpropagation.
- About convergence in neural networks.
- A few last words about neural network implementation.
- Summary. In this chap, have learned basic concepts behind multilayer artificial NNs – currently hottest topic in ML research. In Chap. 2, started our journey with simple single-layer NN structures & now have connected multiple neurons to a powerful NN architecture to solve complex problems e.g. handwritten digit recognition. Demystified popular backpropagation algorithm, which is 1 of building blocks of many NN models used in DL. After learning about backpropagation algorithm in this chap, we are well equipped for exploring more complex DNN architectures. In remaining chaps, will cover more advanced DL concepts & `PyTorch`, an open source library that allows us to implement & train multilayer NNs more efficiently.

- Chap. 12: Parallelizing Neural Network Training with PyTorch. Move on from mathematical foundations of ML & DL focus on PyTorch. PyTorch is 1 of most popular DL libraries currently available, & it lets us implement *neural networks (NNs)* much more efficiently than any of our prev NumPy implementations. In this chap, start using PyTorch & see how it brings significant benefits to training performance.

  This chap will begin next stage of our journey into ML & DL, & will explore following topics:

  - How PyTorch improves training performance
  - Working with PyTorch's `Dataset, DataLoader` to build input pipelines & enable efficient model training
  - Working with PyTorch to write optimized ML code
  - Using `torch.nn` module to implement common DL architectures conveniently
  - Choosing activation functions for artificial NNs

  - PyTorch & training performance. PyTorch can speed up our ML tasks significantly. To understand how it can do this, begin by discussing some of performance challenges typically run into when execute expensive calculations on our hardware. Then, take a high-level look at what PyTorch is & what our learning approach will be in this chap.
    - Performance challenges. Performance of computer processors has, of course, been continuously improving in recent years. That allows us to train more powerful & complex learning systems, i.e., can improve predictive performance of our ML models. Even cheapest desktop computer hardware that's available right now comes with processing units that have multiple cores.

In prev chaps, saw: many functions in scikit-learn allow us to spread those computations over multiple processing units. However, by default, Python is limited to execution on 1 core due to *global interpreter lock (GIL)*. So, although indeed take advantage of Python's multiprocessing library to distribute our computations over multiple cores, still have to consider: most advanced desktop hardware rarely comes with > 8 or 16 such cores.
Recall from Chap. 11: implemented a very simple *multilayer perceptron (MLP)* with only 1 hidden layer consisting of 100 units. Had to optimize ≈ 80000 weight parameters $[784 \cdot 100 + 100] + [100 \cdot 10] + 10 = 79510$ for a very simple image classification task. Images in MNIST are rather small 28×28, & can only imagine explosion in number of parameters if wanted to add additional hidden layers or work with images that have higher pixel densities. Such a task would quickly become unfeasible (không khả thi) for a single processing unit. Question then becomes, how can we tackle such problems more effectively.
Obvious solution to this problem: use *graphics processing units (GPUs)*, which ar real workhorses. Can think of a graphics card as a small computer cluster inside your machine. Another advantage: modern GPUs are great value compared to state-of-state *central processing units (CPUs)*, as you can see in following overview: Fig. 12.1: Comparison of a state-of-art CPU & GPU. Sources for information in Fig. 12.1 are following websites:

- https://ark.intel.com/content/www/us/en/ark/products/215570/intel-core-i9-11900kb-processor-24m-cache-up html
- https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3080-3080ti/

At 2.2 times price of a modern CPU, can get a GPU that has 640 times more cores & is capable of around 46 times more floating-point calculations per sec. So, what is holding us back from utilizing GPUs for our ML tasks? Challenge: writing code to target GPUs is not as simple as executing Python code in our interpreter. There are special packages, e.g. CUDA & OpenCL, that allow us to target GPU. However, writing code in CUDA or OpenCL is probably not most convenient way to implement & run ML algorithms. Good news: this is what PyTorch was developed for!

* What is PyTorch? PyTorch is a scalable & multiplatform programming interface for implementing & running ML algorithms, including convenience wrappers for DL. PyTorch was primarily developed by researchers & engineers from *Facebook AI Research (FAIR)* lab. Its development also involves many contributions from community. PyTorch was initiated released in Sep 2016 & is free & open source under modified BSD license. Many ML researchers & practitioners from academia & industry have adapted PyTorch to develop DL solutions, e.g. Tesla Autopilot, Uber's Pyro, & Hugging Face's Transformers https://pytorch.org/ecosystem/.
To improve performance of training ML models, PyTorch allows execution on CPUs, GPUs, & XLA devices e.g. TPUs. However, its greatest performance capabilities can be discovered when using GPUs & XLA devices. PyTorch supports CUDA-enabled & ROCm GPUs officially. PyTorch's development is based on Torch library www.torch.ch. As its name implies, Python interface is primary development focus of PyTorch.
PyTorch is built around a computation graph composed of a set of nodes. Each node represents an operation that may have 0 or more inputs or outputs. PyTorch provides an imperative programming environment that evaluates operations, executes computation, & returns concrete values immediately. Hence, computation graph in PyTorch is defined implicitly, rather than constructed in advance & executed after.
Mathematically, tensors can be understood as a generalization of scalars, vectors, matrices, & so on. More concretely, a scalar can be defined as a rank-0 tensor, a vector can be defined as a rank-1 tensor, a matrix can be defined as a rank-2 tensor, & matrices stacked in a 3rd dimension can be defined as rank-3 tensors. Tensors in PyTorch are similar to NumPy's arrays, except: tensors are optimized for automatic differentiation & can run on GPUs.
To make concept of a tensor clearer, consider Fig. 12.2: Different types of tensor in PyTorch, which represents tensors of ranks 0 & 1 in 1st row, & tensors of ranks 2 & 3 in 2nd row. Now know what PyTorch is, see how to use it.
* How we will learn PyTorch. 1st, cover PyTorch's programming model, in particular, creating & manipulating tensors. Then, see how to load data & utilize `torch,utils.data` module, which will allow us to iterate through a dataset efficiently. In addition, discuss existing, ready-to-use datasets in `torch.utils.data.Dataset` submodule & learn how to use them.
After learning about these basics, PyTorch neural network `torch.nn` module will be introduced. Then, move forward to building ML models, learn how to compose & train models, & learn how to save trained models on disk for future evaluation.

- 1st steps with PyTorch. Take 1st steps in using low-level PyTorch API. After installing PyTorch, cover how to create tensors in PyTorch & different ways of manipulating them, e.g. changing their shape, data type, & so on.

* Installing PyTorch. To install PyTorch, recommend consulting latest instructions on official https://pytorch.org website. Below, outline basic steps that will work on most systems.
Depending on how your system is set up, can typically just use Python's `pip` installer & install PyTorch from PyPI by executing following from terminal:

```
pip install torch torchvision
```

This will install latest *stable* version, which is 1.9.0 at time of writing. To install 1.9.0 version, which is guaranteed to be compatible with following code examples, can modify preceding command as follows:

```
pip install torch==1.9.0 torchvision==0.10.0
```

If want to use GPUs (recommended), need a compatible NVIDIA graphics card that supports CUDA & cuDNN. If machine satisfies these requirements, can install PyTorch with GPU support, as follows:

```
pip install torch==1.9.0+cu111 torchvision==0.10.0+cu111 -f https://download.pytorch.org/whl/torch_sta
```

for CUDA 11.1 or:

```
pip install torch==1.9.0 torchvision==0.10.0\
whl/torch_stable.html
```

for CUDA 10.2 as of time of writing.
As macOS binaries don't support CUDA, can install from source https://pytorch.org/get-started/locally/#mac-from-s
For more information about installation & setup process, see official recommendations at https://pytorch.org/get-started/locally/.
Note: PyTorch is under active development; therefore, every couple of months, new versions are released with significant changes. Can verify PyTorch version from your terminal as follows:

```
python -c 'import torch; print(torch.__version__)'
```

Remark 47 (Troubleshooting installation of PyTorch)**.** *If experience problems with installation procedure, read more about system- & platform-specific recommendations provided at https://pytorch.org/get-started/locally/. Note: all code in this chap can be run on your CPU; using a GPU is entirely optional but recommended if want to fully enjoy benefits of PyTorch. E.g., while training some NN models on a CPU could take a week, same models could be trained in just a few hours on a modern GPU. If have a graphics card, refer to installation page to set it up appropriately. In addition, may find this setup guide helpful, which explains how to install NVIDIA graphics card drivers, CUDA, & cuDNN on Ubuntu (not required but recommended requirements for running PyTorch on a GPU): https://sebastianraschka.com/pdf/books/dlb/appendix_h_cloud-computing.pdf. As in Chap. 17, can also train models using a GPU for free via Google Colab.*

∗ Creating tensors in PyTorch. Consider a few different ways of creating tensors, & then see some of their properties & how to manipulate them. 1stly, can simply create a tensor from a list or a NumPy array using `torch.tensor` or `torch.from_numpy` function as follows:

```
>>> import torch
>>> import numpy as np
>>> np.set_printoptions(precision=3)
>>> a = [1, 2, 3]
>>> b = np.array([4, 5, 6], dtype=np.int32)
>>> t_a = torch.tensor(a)
>>> t_b = torch.from_numpy(b)
>>> print(t_a)
>>> print(t_b)
tensor([1, 2, 3])
tensor([4, 5, 6], dtype=torch.int32)
```

This resulted in tensors `t_a,t_b`, with their properties, `shape=(3,)` & `dtype=int32`, adopted from their source. Similar to NumPy arrays, can also see these properties:

```
>>> t_ones = torch.ones(2, 3)
>>> t_ones.shape
torch.Size([2, 3])
>>> print(t_ones)
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

Finally, creating a tensor of random values can be done as follows:

```
>>> rand_tensor = torch.rand(2,3)
>>> print(rand_tensor)
tensor([[0.1409, 0.2848, 0.8914],
        [0.9223, 0.2924, 0.7889]])
```

∗ Manipulating data type & shape of a tensor. Learning ways to manipulate tensors is necessary to make them compatible for input to a model or an operation. In this sect, learn how to manipulate tensor data types & shapes via several PyTorch functions that cast, reshape, transpose, & squeeze (remove dimensions).
`torch.to()` function can be used to change data type of a tensor to a desired type:

```
>>> t_a_new = t_a.to(torch.int64)
>>> print(t_a_new.dtype)
torch.int64
```

See https://pytorch.org/docs/stable/tensor_attributes.html for all other data types.
Certain operations require: input tensors have a certain number of dimensions (i.e., rank) associated with a certain number of elements (shape). Thus, might need to change shape of a tensor, add a new dimension, or squeeze an unnecessary dimension. PyTorch provides useful functions (or operations) to achieve this, e.g. `torch.transpose()`, `torch.reshape()`, `torch.squeeze()`. Some examples:

· Transposing a tensor:

```
>>> t = torch.rand(3, 5)
>>> t_tr = torch.transpose(t, 0, 1)
>>> print(t.shape, ' --> ', t_tr.shape)
torch.Size([3, 5]) --> torch.Size([5, 3])
```

· Reshaping a tensor (e.g., from a 1D vector to a 2D array):

```
>>> t = torch.zeros(30)
>>> t_reshape = t.reshape(5, 6)
>>> print(t_reshape.shape)
torch.Size([5, 6])
```

· Removing unnecessary dimensions (dimensions that have size 1, which are not needed):

```
>>> t = torch.zeros(1, 2, 1, 4, 1)
>>> t_sqz = torch.squeeze(t, 2)
>>> print(t.shape, ' --> ', t_sqz.shape)
torch.Size([1, 2, 1, 4, 1]) --> torch.Size([1, 2, 4, 1])
```

∗ Applying mathematical operations to tensors. Applying mathematical operations, in particular linear algebra operations, is necessary for building most ML models. In this subsect, will cover some widely used linear algebra operations, e.g. element-wise product, matrix multiplication, & computing norm of a tensor.
1st, instantiate 2 random tensors, one with uniform distribution in range $[-1, 1]$ & the other with a standard normal distribution:

```
>>> torch.manual_seed(1)
>>> t1 = 2 * torch.rand(5, 2) - 1
>>> t2 = torch.normal(mean=0, std=1, size=(5, 2))
```

Note: `torch.rand` returns a tensor filled with a random numbers from a uniform distribution in range of $[0, 1)$.
Notice: `t1, t2` have same shape. To compute element-wise product of `t1, t2`, can use following:

```
>>> t3 = torch.multiply(t1, t2)
>>> print(t3)
tensor([[ 0.4426, -0.3114],
        [ 0.0660, -0.5970],
        [ 1.1249,  0.0150],
        [ 0.1569,  0.7107],
        [-0.0451, -0.0352]])
```

To compute mean, sum, & standard derivation along a certain axis (or axes), can use `torch.mean()`, `torch.sum()`, `torch.std()`. E.g., mean of each column in `t1` can be computed as follows:

```
>>> t4 = torch.mean(t1, axis=0)
>>> print(t4)
tensor([-0.1373, 0.2028])
```

Matrix-matrix product between `t1, t2` (i.e., $t_1 \times t_2^\top$, where superscript $\top$ is for transpose) can be computed using `torch.matmul()` function as follows:

```
>>> t5 = torch.matmul(t1, torch.transpose(t2, 0, 1))
>>> print(t5)
```

Computing $t_1^\top \times t_2$ is performed by transposing `t1`, resulting in an array of size 2×2:

```
>>> t6 = torch.matmul(torch.transpose(t1, 0, 1), t2)
>>> print(t6)
tensor([[ 1.7453,  0.3392],
        [-1.6038, -0.2180]])
```

Finally, `torch.linalg.norm()` function is useful for computing L^p norm of a tensor. E.g., can calculate L^2 norm of `t1` as follows:

```
>>> norm_t1 = torch.linalg.norm(t1, ord=2, dim=1)
>>> print(norm_t1)
tensor([0.6785, 0.5078, 1.1162, 0.5488, 0.1853])
```

To verify: this code snippet computes L^2 norm of `t1` correctly, can compare results with following NumPy function `np.sqrt(np.sum(np.square(t1.numpy()), axis=1))`.

* Split, stack, & concatenate tensors. In this subsect, will cover PyTorch operations for splitting a tensor into multiple tensors, or reverse: stacking & concatenating multiple tensors into a single one.
  Assume: have a single tensor, & want to split it into 2 or more tensors. For this, PyTorch provides a convenient `torch.chunk()` function, which divides an input tensor into a list of equally sized tensors. Can determine desired number of splits as an integer using `chunks` argument to split a tensor along desired dimension specified by `dim` argument. In this case, total size of input tensor along specified dimension must be divisible by desired number of splits. Alternatively, can provide desired sizes in a list using `torch.split()` function. Have a look at an example of both these options:
  - Provide number of splits:

```
>>> torch.manual_seed(1)
>>> t = torch.rand(6)
>>> print(t)
tensor([0.7576, 0.2793, 0.4031, 0.7347, 0.0293, 0.7999])
>>> t_splits = torch.chunk(t, 3)
>>> [item.numpy() for item in t_splits]
[array([0.758, 0.279], dtype=float32),
 array([0.403, 0.735], dtype=float32),
 array([0.029, 0.8  ], dtype=float32)]
```

  In this example, a tensor of size 6 was divided into a list of 3 tensors each with size 2. If tensor size is not divisible by `chunks` value, last chunk will be smaller.
  - Providing sizes of different splits: Alternatively, instead of defining number of splits, can also specify sizes of output tensors directly. Here, splitting a tensor of size 5 into tensors of sizes 3 & 2:

```
>>> torch.manual_seed(1)
>>> t = torch.rand(5)
>>> print(t)
tensor([0.7576, 0.2793, 0.4031, 0.7347, 0.0293])
>>> t_splits = torch.split(t, split_size_or_sections=[3, 2])
>>> [item.numpy() for item in t_splits]
[array([0.758, 0.279, 0.403], dtype=float32),
 array([0.735, 0.029], dtype=float32)]
```

  Sometimes, working with multiple tensors & need to concatenate or stack them to create a single tensor. In this case, PyTorch functions e.g. `torch.stack(), torch.cat()` come in handy. E.g., create a 1D tensor `A`, containing 1s with size 3, & a 1D tensor `B`, containing 0s with size 2, & concatenate them into a 1D tensor `C` of size 5:

```
>>> A = torch.ones(3)
>>> B = torch.zeros(2)
>>> C = torch.cat([A, B], axis=0)
>>> print(C)
tensor([1., 1., 1., 0., 0.])
```

  If create 1D tensors `A, B`, both with size 3, then can stack them together to form a 2D tensor `S`:

```
>>> A = torch.ones(3)
>>> B = torch.zeros(3)
>>> S = torch.stack([A, B], axis=1)
>>> print(S)
tensor([[1., 0.],
        [1., 0.],
        [1., 0.]])
```

  PyTorch API has many operations that can use for building a model, processing data, & more. However, covering every function is outside scope of book, where will focus on most essential ones. For full list of operations & functions, can refer to documentation page of PyTorch at https://pytorch.org/docs/stable/index.html.

◦ Building input pipelines in PyTorch. When training a deep NN model, usually train model incrementally using an iterative optimization algorithm e.g. stochastic gradient descent, as have seen in prev chaps.

`torch.nn` is a module for building NN models. In cases where training dataset is rather small & can be loaded as a tensor into memory, can directly use this tensor for training. In typical use cases, however, when dataset is too large to fit into

computer memory, will need to load data from main storage device (e.g., hard derive or solid-state drive) in chunks, i.e., batch by batch. (Note: use of term "batch" instead of "mini-batch" in this chap to stay close to PyTorch terminology.) In addition, may need to construct a data-processing pipeline to apply certain transformations & preprocessing steps to our data, e.g. mean centering, scaling, or adding noise to augment training procedure & to prevent overfitting.

Applying preprocessing functions manually every time can be quite cumbersome. Luckily, PyTorch provides a special class for constructing efficient & convenient preprocessing pipelines. In this sect, see an overview of different methods for constructing a PyTorch `Dataset, DataLoader` & implementing data loading shuffling, & batching.

* Creating a PyTorch DataLoader from existing tensors. If data already exists in form of a tensor object, a Python list, or a NumPy array, can easily create a dataset loader using `torch.utils.data.DataLoader()` class. It returns an object of `DataLoader` class, which can use to iterate through individual elements in input dataset. As a simple example, consider following code, which creates a dataset from a list of values from 0 to 5:

```
>>> from torch.utils.data import DataLoader
>>> t = torch.arange(6, dtype=torch.float32)
>>> data_loader = DataLoader(t)
```

Can easily iterate through a dataset entry by entry as follows:

```
>>> for item in data_loader:
...     print(item)
tensor([0.])
tensor([1.])
tensor([2.])
tensor([3.])
tensor([4.])
tensor([5.])
```

If want to create batches from this dataset, with a desired batch size of 3, can do this with `batch_size` argument as follows:

```
>>> data_loader = DataLoader(t, batch_size=3, drop_last=False)
>>> for i, batch in enumerate(data_loader, 1):
...    print(f'batch {i}:', batch)
batch 1: tensor([0., 1., 2.])
batch 2: tensor([3., 4., 5.])
```

This will create 2 batches from this dataset, where 1st 3 elements go into batch #1, & remaining elements go into batch #2. Optimal `drop_last` argument is useful for cases when number of elements in tensor is not divisible by desired batch size. Can drop last non-full batch by setting `drop_last` to `True`. Default value for `drop_last` is `False`.
Can always iterate through a dataset directly, but as just saw, `DataLoader` provides an automatic & customizable batching to a dataset.

* Combining 2 tensors into a joint dataset. Often, may have data in 2 (or possibly more) tensors. E.g., could have a tensor for features & a tensor for labels. In such cases, need to build a dataset that combines these tensors, which will allow us to retrieve elements of these tensors in tuples.
Assume: have 2 tensors `t_x,t_y`. Tensor `t_x` holds our feature values, each of size 3, & `t_y` stores class labels. For this example, 1st create these 2 tensors as follows:

```
>>> torch.manual_seed(1)
>>> t_x = torch.rand([4, 3], dtype=torch.float32)
>>> t_y = torch.arange(4)
```

Now, want to create a joint dataset from these 2 tensors. 1st need to create a `Dataset` class as follows: `[code]`.
A custom `Dataset` class must contain following methods to be used by data loader later on:

· `__init__()`: This is where initial logic happens, e.g. reading existing arrays, loading a file, filtering data, & so forth.

· `__getitem__()`: This returns corresponding sample to given index.

Then create a joint dataset of `t_x,t_y` with custom `Dataset` class as follows:

```
>>> from torch.utils.data import TensorDataset
>>> joint_dataset = TensorDataset(t_x, t_y)
```

Finally, can print each example of joint dataset as follows:

```
>>> for example in joint_dataset:
...     print('x: ', example[0], ' y: ', example[1])
```

Can also simply utilize `torch.utils.data.TensorDataset` class, if 2nd dataset is a labeled dataset in form of tensors. So, instead of using our self-defined `Dataset` class, `JointDataset`, can create a joint dataset as follows:

```
>>> joint_dataset = TensorDataset(t_x, t_y)
```

Note: a common source of error could be: element-wise correspondence between original features x & labels y might be lost (e.g., if 2 datasets are shuffled separately). However, once they are merged into 1 dataset, safe to apply these operations. If have a dataset created from list of image filenames on disk, can define a function to load images from these filenames. See an example of applying multiple transformations to a dataset later in this chap.

* Shuffle, batch, & repeat. As was mentioned in Chap. 2, when training an NN model using stochastic gradient descent optimization, important to feed training data as randomly shuffled batches. Have already seen how to specify batch size using `batch_size` argument of a data loader object. Now, in addition to creating batches, will see how to shuffle & reiterate over datasets. Will continue working with prev joint dataset.
1st, create a shuffled version data loader from `joint_dataset` dataset:

```
>>> torch.manual_seed(1)
>>> data_loader = DataLoader(dataset=joint_dataset, batch_size=2, shuffle=True)
```

Here, each batch contains 2 data records x & corresponding labels y. Now iterate through data loader entry by entry as follows:

```
>>> for i, batch in enumerate(data_loader, 1):
...     print(f'batch {i}:', 'x:', batch[0], '\n y:', batch[1])
```

Rows are shuffled without losing 1-to-1 correspondence between entries in `x, y`.
In addition, when training a model for multiple epochs, need to shuffle & iterate over dataset by desired number of epochs. So, iterate over batched dataset twice:

```
>>> for epoch in range(2):
>>>     print(f'epoch {epoch+1}')
>>>     for i, batch in enumerate(data_loader, 1):
...         print(f'batch {i}:', 'x:', batch[0],
'\n          y:', batch[1])
```

This results in 2 different sets of batches. In 1st epoch, 1st batch contains a pair of values `[y=1, y=2]`, & 2nd batch contains a pair of valeus `[y=3, y=0]`. In 2nd epoch, 2 batches contain a pair of values, `[y=2, y=0] & [y=1, y=3]` resp. For each iteration, elements within a batch are also shuffled.

* Creating a dataset from files on local storage disk. Will build a dataset from image files stored on disk. There is an image folder associated with online content of this chap. After downloading folder, should be able to see 6 images of cats & dogs in JPEG format.
This small dataset will show how building a dataset from stored files generally works. To accomplish this, going to use 2 additional modules: `Image in PIL` to read image file contents & `transforms in torchvision` to decode raw contents & resize images.

Remark 48. `PIL.Image & torchvision.transforms` *modules provide a lot of additional & useful functions, which are beyond scope of book. Encouraged to browse through official documentation to learn more about these functions:*

· *https://pillow.readthedocs.io/en/stable/reference/Image.html for* `PIL.Image`

· *https://pytorch.org/vision/stable/transforms.html for* `torchvision.transforms`.

Before start, take a look at content of these files. use `pathlib` library to generate a list of image files:

```
>>> import pathlib
>>> imgdir_path = pathlib.Path('cat_dog_images')
>>> file_list = sorted([str(path) for path in
... imgdir_path.glob('*.jpg')])
>>> print(file_list)
['cat_dog_images/dog-03.jpg', 'cat_dog_images/cat-01.jpg', 'cat_dog_images/cat-
02.jpg', 'cat_dog_images/cat-03.jpg', 'cat_dog_images/dog-01.jpg', 'cat_dog_
images/dog-02.jpg']
```

Next, visualize these image examples using Matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> import os
>>> from PIL import Image
>>> fig = plt.figure(figsize=(10, 5))
>>> for i, file in enumerate(file_list):
...     img = Image.open(file)
...     print('Image shape:', np.array(img).shape)
...     ax = fig.add_subplot(2, 3, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
```

```
...     ax.imshow(img)
...     ax.set_title(os.path.basename(file), size=15)
>>> plt.tight_layout()
>>> plt.show()
Image shape: (900, 1200, 3)
Image shape: (900, 1200, 3)
Image shape: (900, 1200, 3)
Image shape: (900, 742, 3)
Image shape: (800, 1200, 3)
Image shape: (800, 1200, 3)
```

Fig. 12.3: Images of cats & dogs show example images.
TECHNICAL p. 383: COME BACK LATER WHEN NEEDED

  - Fetching available datasets from `torchvision.datasets` library.
- Building an NN model in PyTorch.
  - PyTorch neural network module `torch.nn`.
  - Building a linear regression model.
  - Model training via `torch.nn & torch.optim` modules.
  - Building a multilayer perceptron for classifying flowers in Iris dataset.
  - Evaluating trained model on test dataset.
  - Saving & reloading trained model.
- Choosing activation functions for multilayer neural networks.
  - Logistic function recap.
  - Estimating class probabilities in multiclass classification via softmax function.
  - Broadening output spectrum using a hyperbolic tangent.
  - Rectified linear unit activation.
- Summary.

- Chap. 13: Going Deeper – Mechanics of PyTorch.
  - Key features of PyTorch.
  - PyTorch's computation graphs.
  - PyTorch tensor objects for storing & updating model parameters.
  - Computing gradients via automatic differentiation.
  - Simplifying implementations of common architectures via `torch.nn` module.
  - Project 1 – predicting fuel efficiency of a car.
  - Project 2 – classifying MNIST handwritten digits.
  - Higher-level PyTorch APIs: a short introduction to PyTorch-Lightning.
  - Summary.

- Chap. 14: Classifying Images with Deep Convolutional Neural Networks. In prev chap, looked in depth at different aspects of PyTorch neural network & automatic differentiation modules, became familiar with tensors & decorating functions, & learned how to work with `torch.nn`. In this chap, will now learn about *convolutional neural networks (CNNs)* for image classification. Start by discussing basic building blocks of CNNs, using a bottom-up approach. Then, will take a deeper dive into CNN architecture & explore how to implement CNNs in PyTorch. Covered topics:
  - Convolution operations in 1D & 2D.
  - Building blocks of CNN architectures
  - Implementing deep CNNs in PyTorch
  - Data augmentation techniques for improving generalization performance
  - Implementing a facial CNN classifier for recognizing if someone is smiling or not
  - Building blocks of CNNs. CNNs are a family of models that were originally inspired by how visual cortex of human brain works when recognizing objects. Development of CNNs goes back to 1990s, when Yann LeCun & his colleagues proposed a novel NN architecture for classifying handwritten digits from images: *Handwritten Digit Recognition with a Back-Propagation Network* by Y. LeCun, & colleagues, 1989, published at NeurIPS conference.

Remark 49 (Human visual cortex). *Original discovery of how visual cortex of our brain functions was made by* David H. Hubel & Torsten Wiesel *in 1959, when they inserted a microelectrode into primary visual cortex of an anesthetized cat (khi họ đưa một vi điện cực vào vỏ não thị giác chính của một con mèo bị gây mê). They observed: neurons respond differently after projecting different patterns of digit in front of cat. This eventually led to discovery of different layers of visual cortex. While primary layer mainly detects edges & straight lines, higher-order layers focus more on extracting complex shapes & patterns.*

Due to outstanding performance of CNNs for image classification tasks, this particular type of feedforward NN gained a lot of attention & led to tremendous improvements in ML for computer vision. Several years later, in 2019, Yann LeCun received Turing award (most prestigious award in CS) for his contributions to field of AI, along with 2 other researchers, Yoshua Bengio & Geoffrey Hinton, whose names you encountered in prev chaps.

In following sects, discuss broader concepts of CNNs & why convolutional architectures are often described as "feature extraction layers". Then, will delve into (đào sâu vào) theoretical def of type of convolution operation that is commonly used in CNNs & walk through examples of computing convolutions in 1D & 2D.

* Understanding CNNs & feature hierarchies. Successfully extracting *salient (relevant) features* is key to performance of any ML algorithm, & traditional ML models rely on input features that may come from a domain expert or are based on computational feature extraction techniques.
  Certain types of NNs, e.g. CNNs, can automatically learn features from raw data that are most useful for a particular task. For this reason, common to consider CNN layers as feature extractors: early layers (those right after input layer) extract *low-level features* from raw data, & later layers (often *fully connected layers*, as in a *multilayer perceptron (MLP)*) use these features to predict a continuous target value or class label.
  Certain types of multilayer NNs, & in particular, deep CNNs, construct a so-called *feature hierarchy* by combining low-level features in a layer-wise fashion to form high-level features. E.g., if dealing with images, then low-level features, e.g. edges & blobs (đốm), are extracted from earlier layers, which are combined to form high-level features. These high-level features can form more complex shapes, e.g. general contours of objects like buildings, cats, or dogs.
  As can see in Fig. 14.1: Creating feature maps from an image., a CNN computes *feature maps* from an input image, where each element comes from a local patch of pixels in input image.
  This local patch of pixels is referred to as *local receptive field* (trường tiếp nhận cục bộ). CNNs will usually perform very well on image-related tasks, & that's largely due to 2 important ideas:
  - *Sparse connectivity*: A single element in feature map is connected to only a small patch (miếng vá) of pixels. (This is very different from connecting to whole input image, as in case of MLPs. May find it useful to look back & compare how implemented a fully connected network that connected to whole image in Chap. 11.)
  - *Parameter sharing*: Same weights are used for different patches of input image.

  As a direct consequence of these 2 ideas, replacing a conventional, fully connected MLP with a convolution layer substantially decreases number of weights (parameters) in network, & see an improvement in ability to capture *salient* features (đặc điểm nổi bật). In context of image data, it makes sense to assume: nearby pixels are typically more relevant to each other than pixels that are far away from each other.
  Typically, CNNs are composed of several *convolutional* & subsampling layers that are followed by 1 or more fully connected layers at end. Fully connected layers are essentially an MLP, where every input unit i is connected to every output unit j with weight w_{ij} (covered in Chap. 11).
  Note: subsampling layers, commonly known as *pooling layers*, do not have any learnable parameters; e.g., there are no weights or bias units in pooling layers. However, both convolutional & fully connected layers have weights & biases that are optimized during training. In following sects, study convolutional & pooling layers in more detail & see how they work. To understand how convolution operations work, start with a convolution in 1D, which is sometimes used for working with certain types of sequence data, e.g. text. After discussing 1D convolutions, work through typical 2D ones that are commonly applied to 2D images.
* Performing discrete convolutions. A *discrete convolution* (or simply *convolution*) is a fundamental operation in a CNN. Therefore, important to understand how this operation works. In this sect, cover mathematical def & discuss some of *naive* algorithms to compute convolutions of 1D tensors (vectors) & 2D tensors (matrices).
  Note: formulas & descriptions in this sect are solely for understanding how convolution operations in CNNs work. Indeed, much more efficient implementations of convolutional operations already exist in packages e.g. PyTorch.

  Remark 50 (Mathematical notation). *In this chap, use subscript to denote size of multidimensional array (tensor); e.g., $A_{n_1 \times n_2}$ is a 2D array of size $n_1 \times n_2$. Use brackets [] to denote indexing of a multidimensional array. E.g., $A[9, j]$ refers to element at index i, j of matrix A. Furthermore, note: use a special symbol * to denote convolution operation between 2 vectors or matrices, which is not to be confused with multiplication operator * in Python.*
* Discrete convolutions in 1D. Start with some basic defs & notations that we are going to use. A discrete convolution for 2 vectors $\mathbf{x}, \mathbf{w}$ is denoted by $\mathbf{y} = \mathbf{x} * \mathbf{w}$, in which vector $\mathbf{x}$: our input (sometimes called *signal*) & $\mathbf{w}$ is called *filter* or *kernel*. A discrete convolution is mathematically defined as follows:

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i-k]w[k]. \tag{238}$$

  As mentioned earlier, brackets [] are used to denote indexing for vector elements. Index i runs through each element of

output vector $\mathbf{y}$. There are 2 odd things in preceding formula needed to clarify: $-\infty$ to $+\infty$ indices & negative indexing for $\mathbf{x}$.
COMPUTATION TECHNIQUES

    - Padding inputs to control size of output feature maps.
    - Determining size of convolution output.
    - Performing a discrete convolution in 2D.
    - Subsampling layers.
  - Putting everything together – implementing a CNN.
  - Implementing a deep CNN using PyTorch.
  - Smile classification from face images using a CNN.
  - Summary.

- Chap. 15: Modeling Sequential Data Using Recurrent Neural Networks.
  - Introducing sequential data.
  - RNNs for modeling sequences.
  - Implementing RNNs for sequence modeling in PyTorch.
  - Summary.

- Chap. 16: Transformers – Improving Natural Language Processing with Attention Mechanisms.
  - Adding an attention mechanism to RNNs.
  - Introducing self-attention mechanism.
  - Attention is all we need: introducing original transformer architecture.
  - Building large-scale language models by leveraging unlabeled data.
  - Fine-tuning a BERT model in PyTorch.
  - Summary.

- Chap. 17: Generative Adversarial Networks for Synthesizing New Data.
  - Introducing generative adversarial networks.
  - Implementing a GAN from scratch.
  - Improving quality of synthesized images using a convolutional & Wasserstein GAN.
  - Other GAN applications.
  - Summary.

- Chap. 18: Graph Neural Networks for Capturing Dependencies in Graph Structured Data.
  - Introduction to graph data.
  - Understanding graph convolutions.
  - Implementing a GNN in PyTorch from scratch.
  - Implementing a GNN using PyTorch Geometric library.
  - Other GNN layers & recent developments.
  - Summary.

- Chap. 19: Reinforcement Learning for Decision Making in Complex Environments.
  - Introduction – learning from experience.
  - Theoretical foundations of RL.
  - Reinforcement learning algorithms.
  - Implementing 1st RL algorithm.
  - A glance at deep Q-learning.
  - Chap & book summary.

2.12 [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:

$$\nabla \cdot \mathbf{v} = 0, \tag{239}$$

$$(\mathbf{v} \cdot \nabla)\mathbf{v} + \nabla p - \frac{1}{\mathrm{Re}}\Delta \mathbf{v} = 0, \tag{240}$$

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)

$$\lambda(\mathbf{W}, \mathbf{b}) = \|\lambda_c\|_\Omega + \|\lambda_u\|_\Omega + \|\lambda_v\|_\Omega + \|\lambda_w\|_\Omega, \tag{241}$$

  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.

$$\int (\nabla \cdot \mathbf{v}) \, \mathrm{d}\mathbf{x} = \sum_i^M v_i n_i A_i \tag{242}$$

  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):

$$u_e = \frac{u_i + u_{i+1}}{2} + \frac{1}{a_p}(p_i + \nabla P_i \cdot \bar{r}_i - p_{i+1} - \nabla P_{i+1} \cdot \bar{r}_{i+1}), \tag{243}$$

  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:

$$\left(\frac{du}{dx}\right)_G = \frac{u_B + u_G}{2\Delta x_B} - \frac{u_B + u_I}{2\Delta x_I}, \tag{244}$$

$$u_G = 2u_b - u_B, \tag{245}$$

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 Deep Learning

3.1 [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 Recognition & 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 $\mathrm{var}[f(x)]$, & for vector variables, covariance is written $\mathrm{cov}[\mathbf{x}, \mathbf{y}]$. Will also use $\mathrm{cov}[\mathbf{x}]$ as a shorthand notation for $\mathrm{cov}[\mathbf{x}, \mathbf{x}]$.

  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

$$y(x, \mathbf{w}) = \sum_{i=0}^{M} w_i x^i = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M, \tag{246}$$

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

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y(x_n, \mathbf{w}) - t_n)^2, \tag{247}$$

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

$$E_{\mathrm{RMS}} = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(y(x_n, \mathbf{w}) - t_n)^2} \tag{248}$$

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)

$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}(y(x_n, \mathbf{w}) - t_n)^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2, \tag{249}$$

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 $\mathbf{w}$.

3.2 [BJP20]. Jean-Pierre Briot, Gaëtan Hadjeres, François-David Pachet. Deep Learning Techniques for Music Generation. 2020

[2 Amazon ratings][324–454 citations]

Amazon review. This book is a survey & analysis of how DL can be used to generate musical content. Authors offer a comprehensive presentation of foundations of DL techniques for music generation. They also develop a conceptual framework used to classify & analyze various types of architecture, encoding models, generation strategies, & ways to control generation. 5 dimensions of this framework are: objective (kind of musical content to be generated, e.g., melody, accompaniment); representation (musical elements to be considered & how to encode them, e.g., chord, silence, piano roll, 1-hot encoding); architecture (structure organizing neurons, their connections, & flow of their activations, e.g., feedforward, recurrent, variational autoencoder); challenge (desired properties & issues, e.g., variability, incrementality, adaptability); & strategy (way to model & control process of generation, e.g., single-step feedforward, iterative feedforward, decoder feedforward, sampling). To illustrate possible design decisions & to allow comparison & correlation analysis, analyze & classify $>$ 40 systems, & discuss important open challenges e.g. interactivity, originality, & structure.

Authors have extensive knowledge & experience in all related research, technical, performance, & business aspects. Book is suitable for students, practitioners, & researchers in AI, ML, & music creation domains. Reader does not require any prior knowledge abut ANNs, DL, or computer music. Text is fully supported with a comprehensive table of acronyms, bibliography, glossary, & index, & supplementary material is available from author's website.

Series: Computational Synthesis & Creative Systems. (Tổng hợp tính toán & Hệ thống sáng tạo) Creativity has become motto of modern world: everyone, every situation, & every company is exhorted to create, to innovate, to think out of box. This calls for design of a new class of technology, aimed at assisting humans in tasks that are deemed creative.

Developing a machine capable of synthesizing completely novel instances from a certain domain of interest is a formidable challenge for CS, with potentially ground-breaking applications in fields e.g. biotechnology, design, & art. Creativity & originality are major requirements, as is ability to interact with humans in a virtuous loop of recommendation & feedback. Problem calls for an interdisciplinary perspective, combining fields e.g. ML, AI, engineering, design, & experimental psychology. Related questions & challenges include design of systems that effectively explore large instance spaces; evaluating automatic generation systems, notably in creative domains; designing systems that foster creativity in humans; formalizing (aspects of) notions of creativity & originality; designing productive collaboration scenarios between humans & machines for creative tasks; & understanding dynamics of creative collective systems.

This book series intends to publish monographs, textbooks & edited books with a strong technical content, & focuses on approaches to computational synthesis that contribute not only to specific problem areas, but more generally introduce new problems, new data, or new well-defined challenges to CS. More information about this series at http://www.springer.com/series/15219.

- Preface. This book is a survey & an analysis of different ways of using DL (deep ANNs) to generate musical content. Propose a methodology based on 5 dimensions for analysis:
  - Objective:
    - What musical content is to be generated? E.g.: melody, polyphony, accompaniment or counterpoint.
    - For what destination & for what use? To be performed by a human(s) (in case of a musical score), or by a machine (in case of an audio file).
  - Representation
    - What are concepts to be manipulated? E.g.: waveform, spectrogram, note, chord, meter & beat.
    - What format is to be used? E.g.: MIDI, piano roll or text.
    - How will representation be encoded? E.g.: scalar, 1-hot or many-hot.
  - Architecture:
    - What type(s) of deep neural network is (are) to be used? E.g.: feedforward network, recurrent network, autoencoder or generative adversarial networks.
  - Challenge
    - What are limitations & open challenges? E.g.: variability, interactivity, & creativity.
  - Strategy
    - How do we model & control process of generation? E.g.: single-step feedforward, iterative feedforward, sampling or input manipulation.

  For each dimension, conduct a comparative analysis of various models & techniques & propose some tentative multidimensional typology (loại hình đa chiều thử nghiệm). This typology is *bottom-up*, based on analysis of many existing DL based systems for music generation selected from relevant literature. These systems are described in this book & are used to exemplify various choices of objective, representation, architecture, challenge, & strategy. Last part of this book includes some discussion & some prospects. Supplementary material is provided at companion website: www.briot.info/dlt4mg/.

- 1. Introduction. DL has recently become a fast growing domain & is now used routinely for classification & prediction tasks, e.g. image recognition, voice recognition or translation. It became popular in 2012, when a DL architecture significantly outperformed standard techniques relying on handcrafted features in an image classification competition, see Sect. 5.

  May explain this success & reemergence of ANN techniques by combination of:

- availability of *massive data*
- availability of *efficient & affordable computing power* [Notably, thanks to graphics processing units (GPU), initially designed for video games, which have now 1 of their biggest markets in DS & DL applications.]
- *technical advances*, e.g.:
  - *pre-training*, which resolved initially inefficient training of neural networks with many layers [79] [Although nowadays it has being replaced by other techniques, e.g. batch normalization [91] & deep residual learning [73].]
  - *convolutions*, which provide motif translation invariance [110]
  - LSTM (long short-term memory), which resolved initially inefficient training of RNNs [82].

There is no consensual definition (định nghĩa đồng thuận) for DL. It is a repertoire (tiết mục) of ML techniques, based on ANNs. Key aspect & common ground is term *deep*. I.e., there are multiple layers processing multiple hierarchical levels of abstractions, which are automatically extracted from data [That said, although DL will automatically extract significant features from data, manual choices of input representation, e.g., spectrum vs. raw wave signal for audio, may be very significant for accuracy of learning & for quality of generated content, see Sect. 4.9.3.] Thus a deep architecture can manage & decompose complex representations in terms of simpler representations. Technical foundation is mostly ANNs (Chap. 5) with many extensions, e.g.: convolutional networks, recurrent networks, autoencoders, & restrictive Boltzmann machines. For more information about history & various facets of DL, see, e.g., a recent comprehensive book on domain [62].

Driving applications of DL are traditional ML tasks [Tasks in ML are types of problems & may also be described in terms of how ML system should process an example [62, Sect. 5.1.1]. E.g.: classification, regression, & anomaly detection (phát hiện bất thường).]: *classification* (e.g., identification of images) & *prediction* [As a testimony of initial DNA of neural networks: *linear regression* & *logistic regression*, Sect. 5.1.] (e.g. of weather) & also more recent ones e.g. *translation*.

But a growing area of application of DL techniques is *generation of content*. Content can be of various kinds: images, t ext & music, the latter being focus of our analysis. Motivation is in using now widely available various corpora (tập đoàn) to automatically learn musical *styles* & to generate *new* musical content based on this.

- 1.1. Motivation.
  - 1.1.1. Computer-Based Music Systems. 1st music generated by computer appeared in 1957. It was a 17 secs long melody named "The Silver Scale" by its author Newman Guttman & was generated by a software for sound synthesis named Music I, developed by Mathews at Bell Laboratories. The same year, "The Illiac Suite" was 1st score composed by a computer [77]. It was named after ILLIAC I computer at University of Illinois at Urbana-Champaign (UIUC) in US. Human "meta-composers" were Lejaren A. Hiller & Leonard M. Isaacson, both musicians & scientists. It was an early example of algorithmic composition, making use of stochastic models (Markov chains) for generation as well as rules to filter generated material according to desired properties.
    In domain of sound synthesis, a landmark was release in 1983 by Yamaha of DX synthesizer, building on groundwork by Chowning on a model of synthesis based on frequency modulation (FM). Same year, MIDI [Musical instrument digital interface, introduced in Sect. 4.7.1.] interface was launched, as a way to interoperate (tương tác) various software & instruments (including Yamaha DX 7 synthesizer). Another landmark was development by Puckette at IRCAM of Max/MSP real-time interactive processing environment, used for real-time synthesis & for interactive performances.
    Regarding algorithmic composition, in early 1960s Iannis Xenakis explored idea of stochastic composition p. 3

- 2. Method.
- 3. Objective.
- 4. Representation.
- 5. Architecture. Deep networks are a natural evolution of neural networks, themselves being an evolution of Perceptron, proposed by Rosenblatt in 1957 [165]. Historically speaking [See, e.g., [62, Sect. 1.2] for a more detailed analysis of key trends in history of DL.], Perceptron was criticized by Minsky & Papert in 1969 [130] for its inability to classify *nonlinearly separable domains* [A simple example & a counterexample of linear separability (of a set of 4 points within a 2D space & belonging to green cross or red circle classes) are shown in Fig. 5.1: Example & counterexample of linear separability. Elements of 2 classes are linearly separable if there is at least 1 straight line separating them. Note: discrete version of counterexample corresponds to case of exclusive or (XOR) logical operator, which was used as an argument by Minsky & Papert in [130].] Their criticism also served in favoring an alternative approach of AI, based on symbolic representations & reasoning.

  Neural networks reappeared in 1980s, thanks to idea of *hidden layers* joint with nonlinear units, to resolve initial linear separability limitation, & to *backpropagation* algorithm, to train such multilayer neural networks [166].

  In 1990s, neural networks suffered declining interest [Meanwhile, convolutional networks started to gain interest, notably though handwritten digit recognition applications [111]. As Goodfellow et al. in [62, Sect. 9.11] put it: "In many ways, they carried torch for rest of DL & paved way to acceptance of neural networks in general."] because of difficulty in training efficiently neural networks with many layers [Another related limitation, although specific to case of recurrent networks, was difficulty in training them efficiently on very long sequences. This was solved in 1997 by Hochreiter & Schmidhuber with LSTM architecture [82], Sect. 5.8.3.] & due to competition fro SVM [196], which were efficiently designed to maximize *separation margin* & had a solid formal background.

An importance advance was invention of *pre-training* technique [Pre-training consists in prior training in *cascade* (1 layer at a time, also named *greedy layer-wise unsupervised training*) of each hidden layer [79] [62, page 528]. It turned out to be a significant improvement for accurate training of neural networks with several layers [46]. I.e., pre-training is now rarely used & has been replaced by other more recent techniques, e.g. *batch normalization* & *deep residual learning*. But its underlying techniques are useful for addressing some new concerns like *transfer learning*, which deals with issue of *reusability* (of what has been learnt, Sect. 8.3).] by Hinton et al. in 2006 [79], which resolved this limitation. In 2012, an image recognition competition (ImageNet Large Scale Visual Recognition Challenge [167]) was won by a deep neural network algorithm named AlexNet [AlexNet was designed by SuperVision team headed by HINTON & composed of ALEX KRIZHEVSKY, ILYA SUTSKEVER, & GEOFFREY E. HINTON [103]. AlexNet is a deep convolutional neural network with 60 million parameters & 650000 neurons, consisting of 5 convolutional layers, some followed by max-pooling layers, & 3 globally-connected layers.], with a stunning margin [On 1st task, AlexNet won competition with a 15% error rate whereas other teams did not achieve $> 26\%$ error rate.] over other algorithms which were using handcrafted features. This striking victory was event which ended prevalent opinion that neural networks with many hidden layers could not be efficiently trained [Interesting, title of Hinton et al.'s article about pre-training [79] is about "deep belief nets" & does not mention term "neural nets" because, as HINTON remembers it in [105]: "At that time, there was a strong belief: deep neural networks were no good & could *never* be trained & that ICML (International Conference on Machine Learning) should not accept papers about neural networks."].

- 5.1. Introduction to Neural Networks. Purpose: review, or to introduce, basic principles of ANNs. Objective: define key *concepts & terminology* used when analyzing various music generation systems. Then, will introduce concepts & basic principles of various derived architectures, like autoencoders, recurrent networks, RBMs, etc., which are used in musical applications. Will not describe extensively techniques of neural networks & DL, e.g. covered in recent book [62].
  - 5.1.1. Linear Regression. Although bio-inpsired (biological neurons), foundation of neural networks & DL is *linear regression*. In statistics, linear regression is an approach for modeling (assumed linear) relationship between a scalar variable $y \in \mathbb{R}$ & 1 [Case of 1 explanatory variable is called *simple linear regression*, otherwise it is named *multiple linear regression*] or > 1 *explanatory variable(s)* $x_1, \ldots, x_n$, with $x_i \in \mathbb{R}$, jointly noted as vector $\mathbf{x}$. A simple example: predict value of a house, depending on some factors (e.g., size, height, location, ...). Eqn (5.1)

$$h(\mathbf{x}) = b + \sum_{i=1}^{n} \theta_i x_i$$

    gives general model of a (multiple) linear regression, where
    - h: *model*, also named *hypothesis*, as this is hypothetical best model to be discovered, i.e. learn
    - b: *bias* [it could be also noted as θ_0, Sect. 5.1.5.], representing *offset*
    - $\theta_1, \ldots, \theta_n$: *parameters* of model, *weights*, corresponding to explanatory variables $x_1, \ldots, x_n$.
  - 5.1.2. Notations. Use following simple notation conventions
    - a *constant* is in roman (straight) font, e.g., integer 1 & note C_4.
    - a *variable* of a model is in roman font, e.g., input variable x & output variable y (possibly vectors).
    - a *parameter* of a model is in italics, e.g., bias b, weight parameter θ_1, model function h, number of explanatory variables n & index i of a variable x_i.
    - a *probability* as well as a *probability distribution* are in italics & upper case, e.g., probability $P(\text{note} = A_4)$ that value of variable note is A_4 & probability distribution $P(\text{note})$ of variable note over all possible notes (outcomes).
  - 5.1.3. Model Training. Purpose of training a linear regression model: find values for each weight θ_i & bias b that best fit actual training data/examples, i.e., various pairs of values (x, y). I.e., want to find parameters & bias values s.t. for all values of x, $h(x)$ is *as close as possible* [Actually, for neural networks that are more complex (nonlinear models) than linear regression & that will be introduced in Sect. 5.5, best fit to training data is not necessarily best hypothesis because it may have a low *generalization*, i.e., a low ability to predict *yet unseen data*. This issue, named *overfitting*, will be introduced in Sect. 5.5.9.] to y, according to some measure named *cost*. This measure represents *distance* between $h(x)$ (prediction, also notated as $\hat{y}$) & y (actual ground value), for *all* examples.
    Cost, also named *loss*, is usually [or also $J(\theta), \mathcal{L}_\theta, \mathcal{L}(\theta)$] notated $J_\theta(h)$ & could be measured, e.g., by a mean squared error (MSE), which measures average squared difference, as shown in (5.2)

$$J_\theta(h) = \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - h(x^{(i)}))^2 = \frac{1}{m} \sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)})^2.$$

    An example is shown in Fig. 5.2: Example of simple linear regression. for case of simple linear regression, i.e., with only 1 explanatory variable x. Training data are shown as blue solid dots. Once model has been trained, values of parameters are adjusted, illustrated by blue solid bold line which mostly fits examples. Then, model can be used for *prediction*, e.g., to provide a good estimate $\hat{y}$ of actual value of y for a given value of x by computing $h(x)$.
  - 5.1.4. Gradient Descent Training Algorithm. Basic algorithm for training a linear regression model, using simple *gradient descent* method, is actually pretty simple [See, e.g., [ANDREW NG lecture notes] for more details.]:
    - initialize each parameter θ_i & bias b to a random or some heuristic value [Pre-training led to a significant advance, as it improved initialization of parameters by using actual training data, via sequential training of successive layers [46].]

- compute values of model h for all examples [Computing cost for all examples is best method but also computationally costly. There are numerous heuristic alternatives to minimize computational cost, e.g., *stochastic gradient descent* (SGD), where 1 example is randomly chosen, & *minibatch gradient descent*, where a subset of examples is randomly chosen. See, e.g., [62, Sects. 5.9 & 8.1.3] for more details.]
- compute *cost* $J_\theta(h)$, e.g., by (5.2)
- compute *gradients* $\frac{\partial J_\theta(h)}{\partial \theta_i}$ which are *partial derivatives* of cost function $J_\theta(h)$ w.r.t. each θ_i, as well as to bias b
- *update simultaneously* [A simultaneous update is necessary for algorithm to behave correctly.] all parameters θ_i & bias according to update rule [Update rule may also be notated as $\theta \coloneqq \theta - \alpha \nabla_\theta J_\theta(h)$, where $\nabla_\theta J_\theta(h)$ is vector of gradients $\frac{\partial J_\theta(h)}{\partial \theta_i}$.] shown in (5.3)

$$\theta_i \coloneqq \theta_i - \alpha \frac{\partial J_\theta(h)}{\partial \theta_i},$$

  with α being *learning rate*. This represents an update in opposite direction of gradients in order to decrease cost $J_\theta(h)$, as illustrated in Fig. 5.3: Gradient descent.
- *iterate* until error reaches a *minimum* [If cost function is *convex* (case for linear regression), there is only 1 *global minimum*, & thus there is a guarantee of finding *optimal* model.], or after a certain number of iterations.

∗ 5.1.5. From Model to Architecture. Now introduce in Fig. 5.4: Architectural model of linear regression. a graphical representation of a linear regression model, as a precursor of a neural network. *Architecture* represented is actually computational representation of model [Mostly use term *architecture* as, in this book, concerned with way to implement & compute a given model & also with relation between an architecture & a representation.].
Weighted sum is represented as a *computational unit* [Use term *node* for any component of a neural network, whether it is just an *interface* (e.g., an input node) or a *computational unit* (e.g., a weighted sum or a function). Use term *unit* only in case of a computational node. Term *neuron* is also often used in place of unit, as a way to emphasize inspiration from biological neural networks.], drawn as a squared box with a Σ, taking its inputs from x_i nodes, drawn as circles.
In example shown, there are 4 explanatory variables: x_1, x_2, x_3, x_4. Note: there is some convention of considering bias as a special case of weight (thus alternatively notated as θ_0) & having a corresponding input node named *bias node*, which is *implicit* [However, as will be explained in Sect. 5.5, bias nodes rarely appear in illustrations of non-toy neural networks.] & has a constant value notated as $+1$. This actually corresponds to considering an implicit additional explanatory variable x_0 with constant value $+1$, as shown in (5.4)

$$h(x) = \theta_0 + \theta_1 x_1 + \cdots + \theta_n x_n = \sum_{i=0}^{n} \theta_i x_i,$$

alternative formulation of linear regression initially defined in (5.1).

∗ 5.1.6. From Model to Linear Algebra Representation. Initial linear regression equation (5.1) may also be made more compact thanks to a linear algebra notation leading to (5.5)

$$h(\mathbf{x}) = b + \boldsymbol{\theta}^\top \mathbf{x},$$

where

1. $b, h(\mathbf{x})$: scalars
2. θ: a vector consisting of n elements $\boldsymbol{\theta} = [\theta_1, \ldots, \theta_n]^\top$
3. $\mathbf{x} = [x_1, \ldots, x_n]$.

∗ 5.1.7. From Simple to Multivariate Model. Linear regression can be generalized to *multivariable linear regression*, case when there are multiple variables $y_1, \ldots, y_p$ to be predicted, as illustrated in Fig. 5.5: Architectural model of multivariate linear regression with 3 predicted variables: y_1, y_2, y_3, each subnetwork represented in a different color.
Corresponding linear algebra equation:

$$h(\mathbf{x}) = b + W\mathbf{x}$$

where

1. b bias vector is a column vector of dimension $p \times 1$, with b_j representing weight of connection between bias input node & jth sum operation corresponding to jth output node
2. W weight matrix is a matrix of dimension $p \times n$, with $W_{i,j}$ representing weight of connection between jth input node & ith sum operation corresponding to ith output node
3. n: number of input nodes (without considering bias node)
4. p: number of output nodes.

For architecture shown in Fig. 5.5, $n = 4$ (number of input nodes & of columns of W) & $p = 3$ (number of output nodes & of rows of W). Corresponding b bias vector & W weight matrix are shown in (5.7)–(5.8) [Indeed, b, W are generalizations of b, θ for case of univariate linear regression (as shown in Sect. 5.1.6) to case of multivariate & thus to multiple rows, each row corresponding to an output node.] [By showing only connections to 1 of output node, in order to keep readability.] & Fig. 5.6: Architectural model of multivariate linear regression showing bias & weights corresponding to connections to 3rd output.

* 5.1.8. Activation Function. Now also apply an *activation function* (AF) to each weighted sum unit, as shown in Fig. 5.7: Architectural model of multivariate linear regression with activation function. This activation function allows us to introduce arbitrary *nonlinear functions*.
  1. From an *engineering* perspective, a nonlinear function is necessary to overcome linear separability limitation of single layer Perceptron.
  2. From a *biological inspiration* perspective, a nonlinear function can capture *threshold* effect for activation of a neuron through its incoming signals (via its dendrites), determining whether it fires along its output (axone).
  3. From a *statistical* perspective, when activation function is sigmoid function, a model corresponds to *logistic regression*, which models probability of a certain class or event & thus performs binary classification [For each output node/variable. See more details in Sect. 5.5.3.].

  Historically speaking, sigmoid function (which is used for *logistic regression*) is most common. Sigmoid function (usually written σ) is defined in (5.9)
$$\text{sigmoid}(z) = \sigma(z) = \frac{1}{1+e^{-z}},$$
  & is shown in Fig. 5.8: Sigmoid function, further analyzed in Sect. 5.5.3.
  An alternative is hyperbolic tangent, often noted tanh, similar to sigmoid but having $[-1, 1]$ as its domain interval ($[0, 1]$ for sigmoid). Tanh is defined in (5.10)
$$\tanh z = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$
  shown in Fig. 5.9: Tanh function.
  But ReLU is now widely used for its simplicity & effectiveness. ReLU, which stands for *rectified linear unit*, is defined as
$$\text{ReLU}(z) = \max(0, z)$$
  shown in Fig. 5.10: ReLU function. Note: as some notation convention use z as name of variable of an activation function, as x is usually reserved for input variables.

◦ 5.2. Basic Building Block. Architectural representation (of multivariate linear regression with activation function) shown in Fig. 5.7 is an instance (with 4 input nodes & 3 output nodes) of a *basic building block* of neural networks & DL architectures. Although simple, this basic building block is actually a working neural network.

It has 2 layers [Although, saw in Sect. 5.5.2, it will be considered as a single-layer neural network architecture. As it has no hidden layer, it still suffers from linear separability limitation of Perceptron.]:

* *Input layer*, on left of figure, is composed of *input nodes* x_i & *bias node* which is an *implicit* & specific input node with a constant value of 1, therefore usually denoted as $+1$.
* *Output layer*, on right of figure, is composed of *output nodes* y_j.

Training a basic building block is essentially the same as training a linear regression model, described in Sect. 5.1.3.

* 5.2.1. Feedforward Computation. After it has been trained, can use this basic building block neural network for prediction. Therefore, simply *feedforward* network, i.e. provide input data to network (*feed in*) & compute output values. This corresponds to (5.12)
$$\hat{y} = h(\mathbf{x}) = \text{AF}(b + W\mathbf{x}).$$
  Feedforward computation of prediction (for architecture shown in Fig. 5.5) is illustrated in (5.13),
$$\hat{y} = h(\mathbf{x}) = \text{AF}(b + W\mathbf{x}) = \cdots = [h_1(\mathbf{x}), h_2(\mathbf{x}), h_3(\mathbf{x})]^\top = [\hat{y}_1, \hat{y}_2, \hat{y}_3]^\top,$$
  where $h_j(\mathbf{x})$ (i.e., $\hat{y}_j$) is prediction of jth variable y_j.
* 5.2.2. Computing Multiple Input Data Simultaneously. Feedforwarding simultaneously a set of examples is easily expressed is easily expressed as a matrix by matrix multiplication, by substituting single vector example $\mathbf{x}$ in (5.12) with a matrix of examples (usually notated as X), leading to (5.14).
$$h(X) = \text{AF}(b + WX).$$
  Successive columns of matrix of elements X correspond to different examples. Use a superscript notation $X^{(k)}$ to denote kth example, kth column of X matrix, to avoid confusion with subscript notation x_i which is used to denote ith input variable. Therefore, $X_i^{(k)}$ denotes ith input value of kth example. Feedforward computation of a set of examples is illustrated in (5.15)
$$h(X) = \cdots = \text{AF}(b + WX) = \cdots = [h(X^{(1)}) \ \cdots \ h(X^{(m)})],$$
  with predictions $h(X^(k))$ being successive columns of resulting output matrix.
  Note: main computation taking place [Apart from computation of AF activation function. In case of ReLU this is fast.] is a product of matrices. This can be computed very efficiently, by using linear algebra vectorized implementation libraries & furthermore with specialized hardware like graphics processing units (GPUs).

◦ 5.3. ML.

* 5.3.1. Definition. Reflect a bit on meaning of training a model, whether it is a linear regression model (Sect. 5.1.1) or basic building block architecture presented in Sect. 5.2. Therefore, consider what ML actually means. Our starting point: following concise & general definition of ML provided by MITCHELL in [131]: "A computer program is said to learn from experience E w.r.t. some class of tasks T & performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."
  At 1st, note: word *performance* actually covers different meanings, specifically regarding computer music context of book:
  1. *execution* of (action to perform) an action, notably an artistic act e.g. a musician playing a piece of music
  2. a *measure* (criterium of evaluation) of that action, notably for a computer system its *efficiency* in performing a task, in terms of time & memory [With corresponding analysis measurements, time complexity & space complexity, for corresponding algorithms.] measurements; or
  3. a measure of *accuracy* in performing a task, i.e. ability to predict or classify with minimal errors.

  In remainder of book, in order to try to minimize ambiguity, use terms as following:
  - *performance* as an act by a musician
  - *efficiency* as a measure of computational ability
  - *accuracy* as a measure of quality of a prediction or a classification [In fact, accuracy may not be a pertinent metric (số liệu liên quan) for a classification task with *skewed* classes, i.e. with 1 class being vastly more represented in data than other(s), e.g., in case of detection of a rare disease. Therefore a confusion matrix & additional metrics like *precision & recall*, & possible combinations like F-score, are used (see, e.g., [62, Sect. 11.1] for details). Will not address them in book, because primarily concerned with content generation & not in pattern recognition (classification).].

  Thus, could rephrase definition as: "A computer program is said to learn from experience E w.r.t. some class of tasks T & accuracy measure A, if its accuracy at tasks in T, as measured by A, improves with experience E."
* 5.3.2. Categories. May now consider 3 main categories of ML w.r.t. nature of experience conveyed by examples:
  - *supervised learning* – dataset is fixed & a correct (expected) answer [usually named a *label* in case of a *classification* task & a *target* in case of a *prediction/regression* task.] is associated to each example, general objective being to *predict answers* for new examples. Examples of tasks are regression (prediction), classification & translation
  - *unsupervised learning* – dataset is fixed & general objective is in *extracting information*. Examples of tasks are feature extraction, data compression (both performed by *autoencoders*, Sect. 5.6), probability distribution learning (performed by RBMs, Sect. 5.7), series modeling (performed by *recurrent* networks, Sect. 5.8), clustering & anomaly detection
  - *reinforcement learning* (Sect. 5.12) – experience is *incremental* through successive actions of an *agents* within an *environment*, with some feedback (*reward*) providing information about *value* of action, general objective being to learn a near optimal *policy* (strategy), i.e. a suite of actions maximizing its cumulated rewards (its *gain*). Examples of tasks are game playing & robot navigation.
* 5.3.3. Components. In his introduction to ML [38], DOMINGOS describes ML algorithms through 3 components:
  1. *representation* – way to represent model – in our case, a *neural network*, as it has been introduced & will be further developed in following sects
  2. *evaluation* – way to evaluate & compare models – via a *cost function*, analyzed in Sect. 5.5.4
  3. *optimization* – way to identify (search among models for) a best model.
* 5.3.4. Optimization. Searching for values (of parameters of a model) that minimize cost function is indeed an *optimization* problem. 1 of most simple optimization algorithms is gradient descent.
  There are various more sophisticated algorithms, e.g. stochastic gradient descent (SGD), Nesterov accelerated gradient (NAG), Adagrad, BFGS, etc. (see, e.g., [62, Chap. 9] for more details).

◦ 5.4. Architectures. From this basic building block, describe in following sects main *types* of *DL architectures* used for music generation (as well as for other purposes):

* feedforward
* autoencoder
* restricted Boltzmann machine (RBM)
* recurrent (RNN).

Also introduce *architectural patterns* (Sect. 5.13.1) which could be applied to them:

* convolutional
* conditioning
* adversarial.

◦ 5.5. Multilayer Neural Network aka Feedforward Neural Network. A *multilayer neural network*, also named a *feedforward neural network*, is an assemblage of successive layers of basic building blocks (1 tập hợp các lớp liên tiếp của các khối xây dựng cơ bản):

* *1st layer*, composed of input nodes, is called *input layer*
* *last layer*, composed of output nodes, is called *output layer*
* any layer *between* input layer & output layer is named a *hidden layer*.

An example of a multilayer neural network with 2 hidden layers is illustrated in Fig. 5.11: Example of a feedforward neural network (detailed).

Combination of a hidden layer & a nonlinear activation function makes neural network a *universal approximator*, able to overcome *linear separability* limitation [Universal approximation theorem [85] states: a feedforward network with a single hidden layer containing a finite number of neurons can approximate a wide variety of interesting functions when giving appropriate parameters (weights). However, there is no guarantee: neural network will be able to learn them!].

∗ 5.5.1. Abstract Representation. Note: in case of practical (non-toy) illustrations of neural network architectures, in order to simplify figures, bias nodes are very rarely illustrated. With a similar objective, sum units & activation function units are also almost always omitted, resulting in a more abstract view e.g. that shown in Fig. 5.12: Example of feedforward neural network (simplified)..

Can further abstract each layer by representing it as an oblong form (hình thuôn dài) (by hiding its nodes) [Sometimes pictured as a rectangle, see Fig. 5.14: (left) GoogLeNet 27-layer deep network architecture. Reproduced from [181]. (right) ResNet 34-layer deep network architecture. Reproduced from [73]. or even as a circle, notably in case of recurrent networks, see Fig. 5.31: RNN (folded)] as shown in Fig. 5.13: Example of a feedforward neural network (abstract).

∗ 5.5.2. Depth. Architecture illustrated in Fig. 5.13 is called a 3-layer neural network architecture, also indicating: *depth* of architecture is 3. Note: number of layers (depth) is indeed 3 & not 4, irrespective of fact: summing up input layer, output layer & 2 hidden layers gives 4 & not 3. This is because, by convention, only layers with weights (& units) are considered when counting number of layers in a multilayer neural network; therefore, input layer is not counted. Indeed, input layer only acts as an input interface, without any weight or computation.

In this book, use a superscript (power) notation [Set of compact notations for expressing dimension of an architecture or a representation will be introduced in Sect. 6.1.] to denote number of layers of a neural network architecture. E.g., architecture illustrated in Fig. 5.13 could be denoted as Feedforward.

Depth of 1st neural network architectures was small. Original Perceptron [165], ancestor of neural networks, has only an input layer & an output layer without any hidden layer, i.e., it is a single-layer neural network. In 1980s, conventional neural networks were mostly 2-layer or 3-layer architectures.

For modern deep networks, depth can indeed by very large, deserving name of *deep* (or even *very deep*) networks. 2 recent examples, both illustrated in Fig. 5.14: (left) GoogLeNet 27-layer deep network architecture. Reproduced from [181]. (right) ResNet 34-layer deep network architecture. Reproduced from [73]., are

· 27-layer GoogLeNet architecture [181]

· 34-layer (up to 152-layer!) ResNet architecture [73] [It introduces technique of *residual learning*, reinjecting input between levels & estimating residual function $h(\mathbf{x}) - \mathbf{x}$, a technique aimed at very deep networks, see [73] for more details.]

Note: depth *does* matter. A recent theorem [43] states: there is a simple radial function [A *radial function* is a function whose value at each point depends only on distance between that point & origin. More precisely, it is radial iff it is invariant under all rotations while leaving origin fixed.] on $\mathbb{R}$, expressible by a 3-layer neural network, which cannot be approximated by any 2-layer network to more than a constant accurcy unless its width is exponential in dimension d. Intuitively, i.e., reducing depth (removing a layer) means exponentially augmenting width (number of units) of layer left. On this issue, interested reader may also wish to review analyses in [4,192].

Note: for both networks pictured in Fig. 5.14, flow of computation is vertical, upwawrd for GoogLeNet & downward for ResNet. These are different usages than convention for flow of computation that have introduced & used so far, which is horizontal, from left to right. Unfortunately, there is no consensus in literature about notation for flow of computation.

Note: in specific case of recurrent networks, introduced in Sect. 5.8, consensus notation is vertical, upward.

∗ 5.5.3. Output Activation Function. Have seen in Sect. 5.2: in modern neural networks, activation function AF chosen for introducing nonlinearity at output of each hidden layer is often ReLU function. But output layer of a neural network has a special status. Basically, there are 3 main possible types of activation function for output layer, named in following, *output activation function* [A shorthand for output layer activation function.]:

· identity – case or a prediction (regression) task. It has continuous (real) output values. Therefore, do not need & do not *want* a nonlinear transformation at last layer

· sigmoid – case of a binary classification task, as in logistic regression [For details about logistic regression, see, e.g., [62, p. 137] or [72, Sect. 4.4]. For this reason, sigmoid function is also called *logistic function*.]. Sigmoid function (usually written σ) has been defined in (5.9) & shown for Fig. 5.8. Note its specific shape, which provides a "separation" effect, used for binary decision between 2 options represented by values 0 & 1

· softmax – most common approach for a classification task with > 2 classes but with only 1 label to be selected [A very common example: estimation by a neural network architecture of next note, modeled as a classification task of a single note label within set of possible notes.] (& where a 1-hot encoding is generally used, see Sect. 4.11).

Softmax function actually represents a *probability distribution* over a discrete output variable with n possible values (i.e., probability of occurrence of each possible value v, knowing input variable x, i.e. $P(y = v|x)$). Therefore, softmax ensures: sum of probabilities for each possible value = 1. Softmax function is defined in (5.16)

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{i=1}^{n} e^{z_i}},$$

& an example of its use is shown in (5.17). Note: σ is used for softmax function, as for sigmoid function, because softmax is actually generalization of sigmoid to case of multiple values, being a variadic function, that is one which accepts a variable number of arguments.

For a classification or prediction task, can simply select value with highest probability (i.e. via *argmax* function, indice of 1-hot vector with highest value). But distribution produced by softmax function can also be used as basis for *sampling*, in order to add nondeterminism & thus content variability to generation (detailed in Sect. 6.6).
– Đối với nhiệm vụ phân loại hoặc dự đoán, có thể chỉ cần chọn giá trị có xác suất cao nhất (tức là thông qua hàm *argmax*, chỉ số của vectơ 1-hot có giá trị cao nhất). Nhưng phân phối được tạo ra bởi hàm softmax cũng có thể được sử dụng làm cơ sở cho *lấy mẫu*, để thêm tính không xác định & do đó là tính biến thiên nội dung vào quá trình tạo (chi tiết trong Phần 6.6).

∗ 5.5.4. Cost Function. Choice of a cost (loss) function is actually correlated to choice of output activation function & to choice of encoding of target y (true value). Table 5.1: Relation between output activation function & cost (loss) function. [Inspired by RONAGHAN's concise pedagogical presentation in [164].] summarizes main cases.
A cross-entropy function measures difference between 2 probability distributions, in our case (of a classification task) between target (true value) distribution y & predicted distribution $\hat{y}$. Note: there are 2 types of cross-entropy cost functions:

· binary cross-entropy, when classification is binary (Boolean),

· categorical cross-entropy, when classification is multiclass with a single label to be selected.

In case of a classification with multiple labels, binary cross-entropy must be chosen joint with sigmoid (because in such cases want to compare distributions independently, class per class [In case of multiple labels, probability of each class is independent from other class probabilities – sum > 1.]) & costs for each class are summed up.
In case of multiple simultaneous classifications (multi multiclass single label), each classification is now independent from other classification, thus have 2 approaches: apply sigmoid & binary cross-entropy for each element & sum up costs, or apply softmax & categorical cross-entropy *independently* for each classification & sum up costs.

∗ 5.5.5. Interpretation. Take some examples to illustrate these subtle but important differences, starting with cases of real & binary values in Fig. 5.15: Cost functions & interpretation for real & binary values.. They also include basic interpretation of results [Interpretation is actually part of *strategy* of generation of music content. It will be explored in Chap. 6. E.g., sampling from probability distribution may be used in order to ensure content generation variability, as will be explained in Sect. 6.6.].

· An example of use of *multiclass single label* type is a classification among a set of possible notes for a monophonic melody, therefore with only 1 single possible note choice (single label), as shown in Fig. 5.16: Cost function & interpretation for a multiclass single label. See, e.g., Blues$_C$ system in Sect. 6.5.1.1.

· An example of use of *multiclass multilabel* type is a classification among a set of possible notes for a single-voice polyphonic melody, therefore with several possible note choices (several labels), as shown in Fig. 5.17: Cost function & interpretation for a multiclass multilabel. See, e.g., Bi-Axial LSTM system in Sect. 6.9.3.

· An example of use of *multi multiclass single label* type is a multiple classification among a set of possible notes for multivoice monophonic melodies, therefore with only 1 single possible note choice for each voice, as shown in Fig. 5.18: Cost function & interpretation for a multi multiclass single label. See, e.g., Blues$_{MC}$ system in Sect. 6.5.1.2.

· Another example of use of *multi multiclass single label* type is a multiple classification among a set of possible notes for a set of time steps (in a piano roll representation) for a monophonic melody, therefore with only 1 single possible note choice for each time step. See, e.g., DeepHear$_M$ system in Sect. 6.4.1.1.

· An example of use of a *multi multiclass single label* type is a 2-level multiple classification among a set of possible notes for a set of time steps for a multivoice set of monophonic melodies. See, e.g., MiniBach system in Sect. 6.2.2.

3 main interpretations used [In various systems to be analyzed in Chap. 6] are

· argmax (index of output vector with largest value), in case of a 1-hot multiclass single label (in order to select most likely note)

· *sampling* from probability represented by output vector, in case of a 1-hot multiclass single label (in order to select a note sorted along its likelihood)

· argsort [argsort is a numpy library Python function.] (indexes of output vector sorted according to their diminishing values), in case of a many-hot multiclass multi label, filtered by some thresholds (in order to select most likely notes above a probability threshold & under a maximum number of simultaneous notes).

∗ 5.5.6. Entropy & Cross-Entropy. Mean squared error has been defined in (5.2) in Sect. 5.1.3. Without getting into details about information theory, now introduce notion & formulation of cross-entropy [With some inspiration from Preiswerk's introduction in [155].].
Intuition behind information theory: information content about an event with a likely (expected) outcome is low, while information content about an event with an unlikely (unexpected, i.e., a surprise) outcome is high.
Take example of a neural network architecture used to estimate next note of a melody. Suppose: outcome is note = B & it has a probability $P(\text{note} = B)$. Can then introduce *self-information* (notated I) of that event in

$$I(\text{note} = B) = \log \frac{1}{P(\text{note} = B)} = -\log P(\text{note} = B)$$

A probability is by def within $[0, 1]$ interval. If look at $-\log$ function in Fig. 5.19: $-\log$ function, could see: its value is high for a low probability value (unlikely outcome) & its value is null for a probability value = 1 (certain outcome), which corresponds to objective introduced above. Note: use of a logarithm also makes self-information additive for independent events, i.e., $I(P_1 P_2) = I(P_1) + I(P_2)$.

Consider all possible outcomes note = Note_i, each outcome having $P(\text{note} = \text{Note}_i)$ as its associated probability, & $P(\text{note})$ being probability distribution for all possible outcomes. Intuition: define *entropy* (notated H) of probability distribution for all possible outcomes as sum of self-information for each possible outcome, weighted by probability of outcome. This leads to

$$H(P) = \sum_{i=0}^{n} P(\text{note} = \text{Note}_i) I(\text{note} = \text{Note}_i) = -\sum_{i=0}^{n} P(\text{note} = \text{Note}_i) \log P(\text{note} = \text{Note}_i).$$

Note: can further rewrite def by using notion of expectation [As expectation, or expected value, of some function $f(x)$ w.r.t. a probability distribution $P(x)$, usually notated as $\mathbb{E}_{X\sim P}[f(x)]$, is average (mean) value that f takes on when x is drawn from P, i.e., $\mathbb{E}_{X\sim P}[f(x)] = \sum_x P(x)f(x)$ (here considering case of discrete variables, which is case for classification within a set of possible notes).], which leads to

$$H(P) = \mathbb{E}_{\text{note}\sim P}[I(\text{note})] = -\mathbb{E}_{\text{note}\sim P}[\log P(\text{note})].$$

Introduce in

$$D_{\text{KL}}(P \parallel Q) = \mathbb{E}_{\text{note}\sim P}\left[\log \frac{P(\text{note})}{Q(\text{note})}\right] = \mathbb{E}_{\text{note}\sim P}[\log P(\text{note}) - \log Q(\text{note})] = \mathbb{E}_{\text{note}\sim P}[\log P(\text{note})] - \mathbb{E}_{\text{note}\sim P}[\log Q(\text{note})]$$

Kullback-Leibler divergence (often abbr. as *KL-divergence*, & notated D_{KL}), as some measure [Note: it is not a true distance measure as it not symmetric.] of how different are 2 separate probability distribution P, Q over a same variable (note). D_{KL} may be rewritten as (5.22) [By using $H(P)$ def in (5.20)]

$$D_{\text{KL}}(P \parallel Q) = -H(P) + H(P, Q)$$

where $H(P, Q)$, named *categorical cross-entropy*, is defined in (5.23)

$$H(P, Q) = -\mathbb{E}_{\text{note}\sim P}[\log Q(\text{note})].$$

Note: categorical cross-entropy is similar to KL-divergence [just like Kl-divergence, it is not symmetric.], while lacking $H(P)$ term. But minimizing $D_{\text{KL}}(P \parallel Q)$ or minimizing $H(P, Q)$, w.r.t. Q, are equivalent, because omitted term $H(P)$ is a constant w.r.t. Q.
Now, remember [Sect. 5.5.4]: objective of neural network: predict $\hat{y}$ probability distribution, which is an estimation of y true ground probability distribution, by minimizing difference between them. This leads to

$$D_{\text{KL}}(y \parallel \hat{y}) = \mathbb{E}_y[\log y - \log \hat{y}] = \sum_{i=0}^{n} y_i(\log y_i - \log \hat{y}_i),$$

$$H(y, \hat{y}) = -\mathbb{E}_y[\log \hat{y}] = -\sum_{i=0}^{n} y_i \log \hat{y}_i.$$

Minimizing $D_{\text{KL}}(y \parallel \hat{y})$ or minimizing $H(y, \hat{y})$, w.r.t. $\hat{y}$, are equivalent, because omitted term $H(y)$ is a constant w.r.t. $\hat{y}$.
Last, deriving *binary cross-entropy* (notated H_{B}) is easy, as there are only 2 possible outcomes, which leads to

$$H_{\text{B}}(y, \hat{y}) = -(y_0 \log \hat{y}_0 + y_1 \log \hat{y}_1).$$

Because $y_1 = 1 - y_0$ & $\hat{y}_1 = 1 - \hat{y}_0$ (as sum of probabilities of 2 possible outcomes is 1), this ends up into

$$H_{\text{B}}(y, \hat{y}) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y})).$$

More details & principles for cost functions [Underlying principle of *maximum likelihood estimation*, not explained here.] can be found, e.g., in [62, Sect. 6.2.1] & [62, Sect. 5.5], resp. In addition, information theory foundation of cross-entropy as number of bits needed for encoding information is introduced, e.g., in [36].

∗ 5.5.7. Feedforward Propagation. Feedforward propagation in a multilayer neural network consists in injecting input data [x part of an example, for generation phase as well as for training phase.] into input layer & propagating computation through its successive layers until output is produced. This can be implemented very efficiently because it consists in a pipelined computation of successive vectorized matrix products (intercalated with AF activation function calls).
Each computation from layer $k-1$ to layer k is processed as (5.28)

$$\text{output}^{[k]} = \text{AF}(b^{[k]} + W^{[k]}\text{output}^{[k-1]}),$$

which is a generalization of (5.12) [Feedforward computation for 1 layer has been introduced in Sect. 5.2.1], where $b^{[k]}, W^{[k]}$ [Use a superscript notation with brackets $^{[k]}$ to denote kth layer, to avoid confusion with superscript notation with parentheses $^{(k)}$ to denote kth example & subscript notation $_i$ to denote ith input variable.] are resp. bias & weight matrix between layer $k-1$ & layer k, & where $\text{output}^{[0]}$ is input layer, as shown in Fig. 5.20: Example of a feedforward neural network (abstract) pipelined computation.

Multilayer neural networks are therefore often also named *feedforward neural networks* or *multilayer Perceptron* (MLP) [Original Perceptron was a neural network with no hidden layer, & thus equivalent to our basic building block, with only 1 output node & with step function as activation function.]

Note: neural networks are *deterministic*. I.e., same input will deterministically *always* produce *same* output. This is a useful guarantee for prediction & classification purposes but may be a limitation for generating new content. However, this may be compensated by *sampling* from resultant probability distribution (see Sects. 5.5.3 & 6.6).

∗ 5.5.8. Training. For training phase [Remember: this is a case of supervised learning (Sect. 5.2).], computing derivatives becomes a bit more complex than for basic building block (with no hidden layer) presented in Sect. 5.1.4. *Backpropagation* is standard method of estimating derivatives (gradients) for a multilayer neural network. Based on *chain rule* principle [166], in order to estimate contribution of each weight to final prediction error, i.e. cost. See, e.g., [62, Chap. 6] for more details.

Note: in most common case, cost function of a multilayer neural network is *not convex*, i.e., there may be *multiple local minima*. Gradient descent, as well as other more sophisticated heuristic optimization methods, does not guarantee global optimum will be reached. But in practice a clever configuration of model (notably, its *hyperparameters, see Sect. 5.5.11*) & well-tuned optimization heuristics, e.g. stochastic gradient descent (SGD), will lead to accurate solutions [On this issue, see [23], which shows: (1) local minima are located in a well-defined band, (2) SGD converges to that band, (3) reaching global minimum becomes harder as network size increase & (4) in practice this is irrelevant as global minimum often leads to overfitting.]

∗ 5.5.9. Overfitting. A fundamental issue for neural networks (& more generally speaking for ML algorithms) is their *generalization* ability, i.e. their capacity to perform well on *yet unseen data*. I.e., do not want a neural network to just perform well on training data [Otherwise, best & simpler algorithm would be a memory-based algorithm, which simply *memorizes* all (x, y) pairs. It has best fit to training data but it does not have any generalization ability.] but also on future data [Future data is not yet known but that does not mean that it is *any kind* of (random) data, otherwise a ML algorithm would not be able to learn & generalize well. There is indeed a fundamental assumption of regularity of data corresponding to a task (e.g., images of human faces, jazz chord progressions, etc.) that neural networks will exploit.] This is actually a fundamental dilemma, 2 opposing risks being

· *underfitting* – when *training error* (error measure on *training data*) is large

· *overfitting* – when *generalization error* (expected error on *yet unseen data*) is large.

A simple illustrative example of underfit, good fit, & overfit models for same training data (green solid dots) is shown in Fig. 5.21: Underfit, good fit, & overfit models.

In order to be able to estimate potential for generalization, dataset is actually divided into 2 portions, with a ratio of $\approx 70/30$:

· *training set* – which will be used for training neural network

· *validation set*, also named *test set*[19] – which will be used to estimate capacity of model for generalization.

∗ 5.5.10. Regularization. There are various techniques to control overfitting, i.e., to improve generalization. They are usually named *regularization* & some examples of well-known techniques are:

· *weight decay* (also known as L^2), by penalizing over-predonderant weights

· *dropout*, by introducing random disconnections

· *early stopping*, by storing a copy of model parameters every time error on validation set reduces, then terminating after an absence of progress during a pre-specified number of iterations, & returning these parameters

· *dataset augmentation*, by data synthesis (e.g., by mirroring, translation & rotation for images; by transposition for music, see Sect. 4.12.1), in order to augment number of training examples.

Will not further detail regularization techniques, see, e.g., [62, Sect. 7].

∗ 5.5.11. Hyperparameters. In addition to *parameters* of model, which are weights of connections between nodes, a model also includes *hyperparameters*, which are parameters at an *architectural meta-level*, concerning both *structure & control*. Examples of *structural* hyperparameters, mainly concerned with architecture, are

· number of layers

· number of nodes

· nonlinear activation function.

Examples of *control* hyperparameters, mainly concerned with learning process, are

· optimization procedure

· learning rate

· regularization strategy & associated parameters.

[19] Actually, a difference could (should) be made, as explained by Hastie et al. in [72, p. 222]: "It is important to note: there are in fact 2 separate goals that might have in mid:

1. Model selection: estimating performance of different models in order to choose best one.

2. Model assessment: having chosen a final model, estimating its prediction error (generalization error) on new data.

If we are in a data-rich situation, best approach for both problems is to randomly divided dataset into 3 parts: a training set, a validation set, & a test set. Training set is used to fit models; validation set is used to estimate prediction error for model selection; test set is used for assessment of generalization error of final chosen model." However, as a matter of simplification, will not consider that difference in book.

Choosing proper values for (tuning) various hyperparameters is fundamental both for efficiency & accuracy of neural networks for a given application. There are 2 approaches for exploring & tuning hyperparameters: *manual tuning* or *automated tuning* – by algorithmic exploration of multidimensional space of hyperparameters & for each sample evaluating generalization error. 3 main strategies for automated tuning are:

· *random search* – by defining a distribution for each hyperparameter, sampling configurations, & evaluating them

· *grid search* – as opposed to random search, exploration is systematic on a small set of values for each hyperparameter

· *model-based optimization* – by building a model of generalization error & running an optimization algorithm over it.

Challenge of automated tuning is its computational cost, although trials may be run in parallel. Will not detail these approaches here; however, further information can be found in [62, Sect. 11.4].

Note: this tuning activity is more objective for conventional tasks e.g. prediction & classification because evaluation measure is objective, being error rate for validation set. When task is generation of new musical content, tuning is more subjective because there is no preexisting evaluation measure. It then turns out to be more *qualitative*, e.g. through a manual evaluation of generated music by musicologists. This evaluation issue will be addressed in Sect. 8.6.

∗ 5.5.12. Platforms & Libraries. Various platforms [See, e.g., survey in [153].], e.g. CNTK, MXNet, PyTorch & TensorFlow, are available as a foundation for developing & running DL systems [There are also more general libraries for ML & data analysis, e.g. SciPy library for Python language, or language R & its libraries.]. They include libraries of

· basic architectures, e.g. ones presented in this chap

· components, e.g. optimization algorithms

· runtime interfaces for running models on various hardware, including GPUs or distributed Web runtime facilities

· visualization & debugging facilities.

Keras is an example of a higher-level framework to simplify development, with CNTK, TensorFlow, & Theano as possible backends. ONNX is an open format for representing DL models & was designed to ease transfer of models between different platforms & tools.

◦ 5.6. Autoencoder. An *autoencoder* is a neural network with 1 hidden layer & with an additional *constraint*: number of output nodes = number of input nodes [Bias is not counted/considered here as it is an implicit additional input node.] Output layer actually *mirrors* input layer. It is shown in Fig. 5.22: Autoencoder architecture, with its peculiar symmetric diabolo (or sand-timer) shape aspect.

p. 83

∗ 5.6.1. Sparse Autoencoder.

∗ 5.6.2. Variational Autoencoder.

• 6. Challenge & Strategy. Core of this book. This chap will analyze in depth how to apply architectures presented in Chap. 5 to learn & generate music. 1st start with a naive, straightforward strategy, using basic prediction task of a neural network to generate an accompaniment for a melody.

– Chương này sẽ phân tích sâu về cách áp dụng các kiến trúc được trình bày trong Chương 5 để học & tạo ra âm nhạc. Đầu tiên, hãy bắt đầu với một chiến lược đơn giản, dễ hiểu, sử dụng nhiệm vụ dự đoán cơ bản của mạng nơ-ron để tạo ra phần đệm cho một giai điệu.

See: although this simple direct strategy does work, it suffers from some limitations. Then study these limitations, some relatively simple to solve, some more difficult & profound – challenges. Analyze various strategies [Remember, & this will be important for following sects, as stated in Chap. 2, consider here strategy related to *generation phase* & not training phase (which could be different).] for each challenge, & illustrate them though different systems [As proposed in Chap. 2, use term *systems* for various proposals – architectures, models, prototypes, systems, & related experiments – for DL-based music generation, collected from related literature.] taken from relevant literature. This also provides an opportunity to study possible relationships between architectures & strategies.

◦ 6.1. Notations for Architecture & Representation Dimensions. At 1st, introduce some compact notations for dimension of an architecture & for size of a representation:

∗ *Architecture-type*^{n} for a n-layer architecture [This notation has actually been introduced in Sect. 5.5.2.], e.g., Feedforward^2 for 2-layer feedforward architecture of MiniBach system introduced in Sect. 6.2.2

∗ *Architecture-type*$\times n$ for a n-instance compound architecture, e.g., RNN$\times$2 for double RNN compound architecture of RL-Tuner introduced in Sect. 6.10.6.1

∗ 1-hot$\times n$ for a multi-1-hot encoding representation, e.g.:

· a n-time steps 1-hot encoding, e.g., 1-hot$\times$64 for 64-time steps representation of DeepHear$_M$ system introduced in Sect. 6.4.1.1

· a n-voice 1-hot encoding, e.g., 1-hot$\times$2 for melody+chords representation of Blues$_{MC}$ system introduced in Sect. 6.5.1.2, or

· a combination of a multi-time steps encoding & a multivoice encoding, e.g., 1-hot$\times 64 \times (1+3)$ for 64-time steps 1-voice input & 3-voices output representation of MiniBach system introduced in Sect. 6.2.2.

An example of a combination of 2 notations is LSTM$^2 \times 2$ for double 2-layer RNN compound architecture of Anticipation-RNN system introduced in Sect. 6.10.3.5.

- 6.2. An Introductory Example.
  - 6.2.1. Single-Step Feedforward Strategy. Most direct strategy is using prediction or classification task of a neural network in order to generate musical content. Consider following objective: for a given melody want to generate an accompaniment, e.g., a counterpoint. Consider a dataset of examples, each one being a pair (melody, counterpointmelody(ies)). Then train a feedforward neural network architecture in a supervised learning manner on this dataset. Once trained, can choose an arbitrary melody & feedforward it into architecture in order to produce a corresponding counterpoint accompaniment, in style of dataset. Generation is completed in a single-step of feedforward processing. Therefore, have named this strategy *single-step feedforward strategy*.
  - 6.2.2. Example: MiniBach Chorale Counterpoint Accompaniment Symbolic Music Generation System. Consider following objective: generating a counterpoint accompaniment to a given melody for a soprano voice, through 3 matching parts, corresponding to alto, tenor, & bass voices. Use as a corpus set of J. S. Bach's polyphonic chorales [5]. As want this 1st introductory system to be simple, consider only 4 measures long excerpts from corpus. Dataset is constructed by extracting all possible 4 measures long excerpts from original 352 chorales, also transposed in all possible keys. Once trained on this dataset, system may be used to generate 3 counterpoint voices corresponding to an arbitrary 4 measures long melody provided as an input. Somehow, it does capture practice of J. S. Bach, who chose various melodies for a soprano & composed 3 additional voices melodies (for alto, tenor, & bass) in a counterpoint manner.
    1st, need to decide input as well as output representations. Represent 4 measures of 4/4 music. Both input & output representations are symbolic, of piano roll type, with 1-hot recording for each voice, i.e. a multi-1-hot encoding for output representation. 3 1st voices (soprano, alto, & tenor) have a scope of 20 possible notes plus an additional token to encode a hold [see Sect. 4.9.1. Note: as a simplification, MiniBach does not consider rests.], while last voice (bass) has a scope of 27 possible notes plus hold symbol. Time quantization (value of time step) is set at 16th note, which is minimal note duration used in corpus. Input representation has a size of 21 possible notes $\times 16$ time steps $\times 4$ measures, i.e., $21 \times 16 \times 4 = 1344$, while output representation has a size of $(21 + 21 + 28) \times 16 \times 4 = 4480$.
    Architecture, a feedforward network, is shown in Fig. 6.1: MiniBach architecture. As explained previously & because of mapping between representation & architecture, input layer has 1344 nodes & output layer 4480. There is a single hidden layer with 200 units [This is an arbitrary choice.]. Nonlinear activation function used for hidden layer is ReLU. Output layer activation function is sigmoid & cost function used is binary cross-entropy (this is a case of multi^2 multiclass single label, see Sect. 5.5.4).
    Detail of architecture & encoding is shown in Fig. 6.2: MiniBach architecture & encoding. It shows encoding of successive music time slices into successive 1-hot vectors directly mapped to input nodes (variables). In figure, each blackened vector element as well as each corresponding blackened input node element illustrate specific encoding (1-hot vector index) of a specific note time slice, depending of its actual pitch (or a hold in case of a longer note, shown with a bracket). Dual process happens at output. Each grey output node element illustrates chosen note (the one with highest probability), leading to a corresponding 1-hot index, leading ultimately to a sequence of notes for each counterpoint voice.
    Characteristics of this system, named MiniBach [MiniBach is actually a strong simplification – but with same objective, corpus, & representation principles – of DeepBach system introduced in Sect. 6.14.2.], are summarized in multidimensional conceptual framework (as defined in Chap. 2 Method) in Table 6.1: MiniBach summary. Notation [These notations, introduced in Sect. 6.1, will be summarized in Sect. 7.2.] 1-hot$\times 64 \times (1 + 3)$ means an encoding with 1 input + 3 output voices, each with 64 (for 4 measures of 16 time steps each) 1-hot encodings of notes. Notation Feedforward^2 means a 2-layer feedforward architecture (with 1 hidden layer). An example of a chorale counterpoint generated from a soprano melody is shown in Fig. 6.3: Example of a chorale counterpoint generated by MiniBach from a soprano melody.
  - 6.2.3. A 1st Analysis. Chorales produced by Minibach look convincing at 1st glance. But, independently of a qualitative musical evaluation, where an expert could detect some defects, objective limitations of MiniBach appear:
    - A structural limitation: music produced (as well as input melody) has a *fixed size* (one cannot produce a longer or shorter piece of music).
    - The same melody will always produce exactly *same* accompaniment because of *deterministic* nature of a feedforward neural network architecture.
    - Generated accompaniment is produced with a *single atomic step*, without any possibility of human intervention (i.e., without *incrementality* & *interactivity*).
- 6.3. A Tentative List of Limitations & Challenges. Introduce a tentative list of limitations (in most cases, properties not fulfilled) & challenges [Our shallow distinction between a limitation & a challenge: *limitations* have relatively well-understood solutions, whereas *challenges* are more profound & still subject of open research.]:
  - *Ex nihilo* generation (vs accompaniment)
  - Length variability (vs fixed length)
  - Content variability (vs determinism)
  - Expressiveness (vs mechanization)
  - Melody-harmony consistency
  - Control (e.g., tonality conformance, maximum number of repeated notes ...)
  - Style transfer
  - Structure
  - Originality (vs imitation)

* Incrementality (vs 1-hot generation)
* Interactivity (vs automation)
* Adaptability (vs no improvement through usage)
* Explainability (vs black box).

Analyze them with possible matching solutions & illustrate them through various examples systems.

- 6.4. *Ex Nihilo* Generation. MiniBach system is good at generating an accompaniment (a counterpoint composed of 3 distinct melodies) matching an input melody. This is an example of supervised learning, as training examples include both an input (a melody) & a corresponding output (accompaniment).

  Now suppose: our objective: generate a melody on its own – not as an accompaniment of some input melody – while being based on a style learn from a corpus of melodies. A standard feedforward architecture & its companion single-step feedforward strategy, e.g. those used in MiniBach (described in Sect. 6.2.2), are not appropriate for such an objective.

  Introduce some strategies to generate new music content *ex nihilo* or from minimal *seed* information, e.g. a starting note or a high-level description.

  * 6.4.1. Decoder Feedforward. 1st strategy is based on an autoencoder architecture. As explained in Sect. 5.6, through training phase an autoencoder will specialize its hidden layer into a detector of features characterizing type of music learnt & its variations [To enforce this specialization, sparse autoencoders are often used (Sect. 5.6.1).]. One can then use these features as an *input inference* to *parameterize* generation of musical content. Idea is then to:
    · *choose a seed* as a vector of values corresponding to hidden layer units
    · *insert* it in hidden layer
    · *feedforward* it through decoder.

    This strategy, named *decoder feedforward*, will produce a *new* musical content corresponding to features, in same format as training examples.

    In order to have a minimal & high-level vector of features, a stacked autoencoder (Sect. 5.6.3) is often used. Seed is then inserted at *bottleneck hidden layer* of stacked autoencoder [I.e., at exact middle of encoder/decoder stack, as shown in Fig. 6.6: Generation in DeepHear. Extension of a figure reproduced from [179].] & feedforwarded through chain of decoders. Therefore, a simple seed information can generate an arbitrarily long, although fixed-length, musical content.

    · 6.4.1.1. #1 Example: DeepHear Ragtime Melody Symbolic Music Generation System. An example of this strategy: DeepHear system by Sun [179]. Corpus used is 600 measures of Scott Joplin's ragtime music, split into 4 measures long segments. Representation used is piano roll with a multi-1-hot encoding. Quantization (time step) is a 16th note, thus representation includes $4 \times 16 = 64$ time steps (notated as 1-hot$\times$64). Number of input nodes is round 5000, which provides a vocabulary of about 80 possible note values. Architecture is shown in Fig. 6.4: DeepHear stacked autoencoder architecture. Extension of a figure reproduced from [179]. & is a 4-layer stacked autoencoder (notated as Autoencoder^4) with a decreasing number of hidden units, down to 16 units.

      After a pre-training phase [Do not detail pre-training here, refer to, e.g., [62, p. 528].], final training is performed, with each provided example used both as an input & as an output, in self-supervised learning manner (see Sect. 5.6) shown in Fig. 6.5: Training DeepHear. [179].

      Generation is performed by inputting random data as seed into 16 bottleneck hidden layer units [units of hidden layer represent an embedding (Sect. 4.9.3), of which an arbitrary instance is named by Sun a *label*.] (shown within a red rectangle) & then by feedforwarding it into chain of decoders to produce an output (in same 4 measures long format as training examples), as shown in Fig. 6.6: Generation in DeepHear. Extension of a figure reproduced from [179].. Summarize characteristics of DeepHear$_M$ [Notate DeepHear$_M$ this DeepHear melody generation system, where M stands for melody, because another experiment with same DeepHear architecture but with a different objective will be presented later on in Sect. 6.10.4.1.] in Table 6.2: DeepHear$_M$ summary.

      In [179], Sun remarks: system produces a certain amount of plagiarism. Some generated music is almost recopied from corpus. He states: this is because of small size of bottleneck hidden layer (only 16 nodes) [179]. Measured similarity (defined as percentage of notes in a generated piece that are also in 1 of training pieces) & found: on average, it is 59.6%, which is indeed quite high, although it does not prevent most of generated pieces from sounding different.

    · 6.4.1.2. #2 Example: deepAutoController Audio Music Generation System. deepAutoController system, by Sarroff & Casey [169], is similar to DeepHear (Sect. 6.4.1.1) in that it also uses a stacked autoencoder. But representation is *audio*, more precisely a spectrum generated by Fourier transform, see [169] for more details. Dataset is composed of 8000 songs of 10 musical genres, leading to 70000 frames of magnitude Fourier transforms [As authors state in [169]: "Chose to use frames of magnitude FFs (Fast Fourier transforms) for our models because they may be reconstructed exactly into original time domain signal when phase information is preserved, Fourier coefficients are not altered, & appropriate windowing & overlap-add is applied. It was thus easier to subjectively evaluate quality of reconstructions that had been processed by autoencoding models."]. Entire data is normalized to $[0, 1]$ range. Cost function used is mean squared error. Architecture is a 2-layer stacked autoencoder, bottleneck hidden layer having 256 units & input & output layers having 1000 nodes. Authors report: increasing number of hidden units does not appear to improve model performance.

      System, summarized in Table 6.3: deepAutoController summary, also provides a user interface, analyzed in Sect. 6.15, to interactively control generation, e.g., selecting a given input (to be inserted at bottleneck hidden layer), generating a random input, & controlling (by scaling or muting) activation of a given unit.

  * 6.4.2. Sampling. Another strategy is based on sampling. *Sampling* is action of generating an element (a *sample*) from a *stochastic* model according to a *probability distribution*.

· 6.4.2.1. Sampling Basics. Main issue for sampling: ensure: samples generated match a given distribution. Basic idea: generate a sequence of sample values in such a way that, as more & more sample values are generated, distribution of values more closely approximates target distribution. Sample values are thus produced *iteractively*, with distribution of next sample being dependent only on current sample value. Each successive sample is generated through a *generate-&-test* strategy, i.e., by generating a prospective candidate, accepting or rejecting it (based on a defined *probability density*) &, if needed, regenerating it. Various sampling strategies have been proposed: Metropolis-Hastings algorithm, Gibbs sampling (GS), block Gibbs sampling, etc. See, e.g. [62, Chap. 17] for more details about sampling algorithms.

· 6.4.2.2. Sampling for Music Generation. For musical content, may consider 2 different levels of probability distribution (& sampling):

1. *item-level* or *vertical* dimension – at level of a compound musical item, e.g., a chord. In this case, distribution is about relations between components of chord, i.e., describing probability of notes to occur together
2. *sequence*-level or *horizontal* dimension – at level of a sequence of items, e.g., a melody composed of successive notes. In this case, distribution is about sequence of notes, i.e., it describes probability of occurrence of a specific note after a given note.

An RBM (restricted Boltzmann machine) architecture is generally [A counterexample is C-RBM convolutional RBM architecture, introduced in Sect. 6.10.5.1, which models both vertical dimension (simultaneous notes) & horizontal dimension (sequence of notes) for single-voice polyphonies.] used to model vertical dimension, i.e. which notes should be played together. As noted in Sect. 5.7, an RBM architecture is dedicated to learning distributions & can learn efficiently from few examples. This is particularly interesting for learning & generating chords, as combinatorial nature of possible notes forming a chord is large & number of examples is usually small. An example of a *sampling strategy* applied on an RBM for horizontal dimension will be presented in Sect. 6.4.2.3.

An RNN architecture is often used for horizontal dimension, i.e. which note is likely to be played after a given note, described in Sect. 6.5.1. As in Sect. 6.6.1, a sampling strategy may be also added to enforce variability.

See in Sect. 6.9.1: a compound architecture named RNN-RBM may combine & *articulate* [This issue of how to articulate vertical & horizontal dimensions, i.e. harmony with melody, will be further analyzed in Sect. 6.9.] these 2 different approaches:

1. an RBM architecture with a sampling strategy for vertical dimension
2. an RNN architecture with an iterative feedforward strategy for horizontal dimension.

An alternative approach: use sampling as *unique* strategy for both dimensions, as witnessed by DeepBach system to be analyzed in Sect. 6.14.2.

· 6.4.2.3. Example: RBM-based Chord Music Generation System. In [11], Boulanger-Lewandowski et al. propose to use a restricted Boltzmann machine (RBM) [80] to model polyphonic music. Their objective is actually to improve transcription of polyphonic music from audio. But prior to that, authors discuss generation of samples from model that has been learnt as a qualitative evaluation & also for music generation [12]. In their 1st experiment, RBM learns from corpus distribution of possible simultaneous notes, i.e., repertoire of chords.

Corpus is set of J. S. Bach's chorales (as for MiniBach, described in Sect. 6.2.2). Polyphony (number of simultaneous notes) varies from 0 to 15 & average polyphony is 3.9. Input representation has 88 binary visible units that span whole range of piano from A_0 to C_8, following a many-hot encoding. Sequences are aligned (transposed) onto a single common tonality (e.g., C major/minor) to ease learning process.

One can sample from RBM through block Gibbs sampling, by performing alternative steps of sampling hidden layer nodes (considered as variables) from visible layer nodes (Sect. 5.7). Fig. 6.7: Samples generated by RBM trained on J. S. Bach chorales. [11] shows various examples of samples. Vertical axis represents successive possible notes. Each column represents a specific sample composed of various simultaneous notes, with name of chord written below when analysis is unambiguous. Table 6.4: RBM_C summary. summarizes this RBM-based chord generation system, which notate RBM_C where C stands for chords.

◦ 6.5. Length Variability. An important limitation of single-step feedforward strategy (Sect. 6.2.1) & of decoder feedforward strategy (Sect. 6.4.1): length of music generated (more precisely number of times steps or measures) is *fixed*. It is actually fixed by architecture, namely number of nodes of output layer [In case of an RBM, number of nodes of input layer (which also has role of an output layer).]. To generate a longer (or shorter) piece of music, one needs to reconfigure architecture & its corresponding representation.

∗ 6.5.1. Iterative Feedforward. Standard solution to this limitation is to use a RNN. Typical usage, as initially described for text generation by Graves in [64], is to

· select some *seed* information as *1st* item (e.g., 1st note of a melody)

· *feedforward* it into recurrent network in order to produce *next* item (e.g., next note)

· use this next item as next input to produce *next next* item

· repeat this process iteratively until a *sequence* (e.g., of notes, i.e., a melody) of desired length is produced.

Note: *iterative* aspect of generation, processed element by element. Therefore, name this approach *iterative time step feedforward* strategy, abbr. as *iterative feedforward* strategy. Actually, a *recursion* – current output reenters as next input – is also often present. However, there are a few rare exceptions, as will see, e.g., in Sequential (Sect. 6.8.2) & in BLSTM (Sect. 6.8.3) architectures, where there is an iteration but *no* recusion.

Note: iterative feedforward straetgy, as decoder feedforward strategy (Sect. 6.4.1), is 1 kind of *seed-based generation* (Sect. 6.4), as full sequence (e.g., a melody) is generated iteratively from an initial seed item (e.g., a starting note).

· 6.5.1.1. #1 Example: Blues Chord Sequence Symbolic Music Generation System. In [42], ECK & SCHMIDHUBER describe a double experiment undertaken with a recurrent network architecture using LSTMs [This was actually 1st experiment in using LSTMs to generate music.] In their 1st experiment, objective: learn & generate chord sequences. Format of representation is piano roll, with 2 types of sequences: melody & chords, although chords are represented as notes. Melodic range as well as chord vocabulary is strongly constrained, as corpus consists of 12 measures long blues & is handcrafted (melodies & chords). 13 possible notes extend from middle C C_4 to tensor C C_5. 12 possible chords extend from C to B.

A 1-hot encoding is used. Time quantization (time step) is set at 8th note, half of minimal note duration used in corpus, which is a quarter note. With 12 measures long music this equates to 96 time steps. An example of chord sequence training example is shown in Fig. 6.8: A chord training example for blues generation [42].

Architecture for this 1st experiment is: an input layer with 12 nodes (corresponding to a 1-hot encoding of 12 chord vocabulary), a hidden layer with 4 LSTM blocks containing 2 cells each [see Sect. 5.8.3 for difference between LSTM cells & blocks.] & an output layer with 12 nodes (identical to input layer).

Generation is performed by presenting a *seed* chord (represented by a note) & by iteratively feedforwarding network, producing prediction of next time step chord, using it as next input & so on, until a sequence of chords has been generated. Architecture & iterative generation is illustrated in Fig. 6.9: Blues chord generation architecture. This system, which notate Blues$_C$, where C stands for chords, is summarized in Table 6.5: Blues$_C$ summary.

· 6.5.1.2. #2 Example: Blues Melody & Chords Symbolic Music Generation System. In ECK & SCHMIDHUBER's 2nd experiment [42], objective: simultaneously generate melody & chord sequences. New architecture is an extension of prev one: it has an input layer with 25 nodes (corresponding to a 1-hot encoding of 12 chord vocabulary & to a 1-hot encoding of 13 melody note vocabulary), a hidden layer with 8 LSTM blocks (4 chord blocks & 4 melody blocks), containing 2 cells each, & an output layer with 25 nodes (identical to input layer).

Separation between chords & melody is ensured as follows:

1. chord blocks are fully connected to input nodes & to output nodes corresponding to chords
2. melody blocks are fully connected to input nodes & to output nodes corresponding to melody
3. chord blocks have recurrent connections to themselves & to melody blocks, &
4. melody blocks have recurrent connections *only* to themselves.

Generation is performed by presenting a seed (note & chord) & by recursively feedforwarding it into network, producing prediction of next time step note & chord, & so on, until a sequence of notes with chords is generated. Fig. 6.10: Example of blues generated (excerpt). shows an example of melody & chords generated. Table 6.6: Blues$_{MC}$ summary summarizes this 2nd system, which notate Blues$_{MC}$ (where MC stands for melody & chords).

This 2nd experiment is interesting in that it *simultaneously* generates melody & chords. Note: in this 2nd architecture, recurrent connections are *asymmetric* as authors wanted to ensure preponderant role of chords. Chord blocks have recurrent connections to themselves but also to melody blocks, whereas melody blocks do not have recurrent connections to chord blocks. I.e., chord blocks will receive previous step information about chords & melody, whereas melody blocks cannot use previous step information about chords. This somewhat ad hoc configuration of recurrent connections in architecture is a way to control interaction between harmony & melody in a master-slave manner. Control of interaction & consistency between melody & harmony is indeed an effective issue & it will be further addressed in Sect. 6.9 where analyze alternative approaches.

○ 6.6. Content Variability. A limitation of iterative feedforward strategy on an RNN, as illustrated by blues generation experiment described in Sect. 6.5.1.2: generation is *deterministic*. Indeed, a neural network is deterministic [There are stochastic versions of ANNs – an RBM is an example – but they are not mainstream.]. As a consequence, feedfowarding *same input* will always produce *same output*. As generation of next note, next next note, etc., is deterministic, *same* seed note will lead to *same* generated series of notes [Actual length of melody generated depends on number of iterations.]. Moreover, as there are only 12 possible input values (12 pitch classes), there are only 12 possible melodies.

∗ 6.6.1. Sampling. Fortunately, usual solution is quite simple. Assumption: output representation of melody is 1-hot encoded. I.e., output representation is of a piano roll type, output activation layer is softmax & generation is modeled as a classification task. See an example in Fig. 6.11: Sampling softmax output, where $P(x_t = C|x_{<t})$ represents conditional probability for element (note) x_t at step t to be a C given previous elements $x_{<t}$ (melody generated so far).

Default *deterministic* strategy consists in choosing class (note) with *highest probability*, i.e. $\arg\max_{x_t} P(x_t|x_{<t})$, i.e. Ab in Fig. 6.11. Can then easily switch to a *nondeterministic* strategy, by *sampling* output which corresponds (through softmax function) to a probability distribution between possible notes. By sampling a note following distribution generated [Chance of sampling a given class/note is its corresponding probability. In example shown in Fig. 6.11, Ab has around 1 chance in 2 of being selected & Bb 1 chance in 4.], introduce *stochasticity* in process & thus *variability* in generation.

· 6.6.1.1. #1 Example: CONCERT Bach Melody Symbolic Music Generation System. CONCERT (an acronym for CONnectionist Composer of ERudite Tunes) developed by Mozer [138] in 1994, was actually 1 of 1st systems for generating music based on recurrent networks (& before LSTM). It is aimed at generating melodies, possibly with some chord progression as an accompaniment.

Input & output representation includes 3 aspects of a note: pitch, duration, & *harmonic chord accompaniment*. Representation of a pitch, named PHCCCH, is inspired by psychological pitch representation space of Shepard [172], & is based on 5 dimensions, as illustrated in Fig. 6.12: CONCERT PHCCCH pitch representation. Inspired by [172] & [138].

3 main components are as follows:

1. pitch height (PH)
2. (modulo) chroma circle (CC) cartesian coordinates
3. (harmonic) circle of 5ths (CH) cartesian coordinates.

Motivation is in having a more musically meaningful representation of pitch by capturing similarity of octaves & also harmonic similarity between a note & its 5th. Proximity of 2 pitches is determined by computing Euclidean distance between their PHCCCH representations, that distance being invariant under transposition. Encoding of pitch height is through a scalar variable scaled to range from -9.798 for C_1 to $+9.798$ for C_5. Encoding of chroma circle & of circle of 5ths is through a 6 binary value vector, for reasons detailed in [138]. Resulting encoding includes 13 input variables, with some examples shown in Table 6.7: Examples of PHCCCF pitch representation. Note: a rest is encoded as a pitch with a unique code.

Durations are considered at a very fine-grain level, each beat (a quarter note) being divided into 12ths, thus having a duration of 12/12. This choice allows to represent binary (2 or 4 divisions) as well as ternary (3 divisions) rhythms. In a similar way to representation of pitch, a duration is represented through a scalar & 2 circle coordinates, for 1/4 & 1/3 beat cycles, as illustrated in Fig. 6.13: CONCERT duration representation. Inspired by [138], resulting in 5 dimensions directly encoded through a 5 binary value vector see more details in [138]). Temporal scope is a *note step*, i.e., granularity of processing by architecture is a *note* [& not a fixed time step as for most of recurrent architectures, e.g., in Sect. 6.5.1.1. Various types of temporal scope have been introduced in Sect. 4.8.1.]

Chords are represented in an exceptional way as a triad or a tetrachord, through root, 3rd (major or minor) & 5th (perfect, augmented or diminished), with possible addition of a 7th component (minor or major). To represent next note to be predicted, CONCERT system actually uses both this rich & distributed representation (named next-node-distributed, see Fig. 6.14: CONCERT architecture [138].) & a more concise & traditional representation (named next-node-local), in order to be more intelligible. Activation function is sigmoid function resccaled to $[-1, 1]$ range & cost function is mean squared error.

In generation phase, output is interpreted as a probability distribution over a set of possible notes as a basis for deciding next node in a nondeterministic way, following *sampling* strategy.

CONCERT has been tested on different examples, notably after training with melodies of J. S. Bach. Fig. 6.15: Example of melody generation by CONCERT based on J. S. Bach training set [138]. shows an example of a melody generated based on Bach training set. Although now a bit dated, CONCERT has been a pioneering model & discussion in article about representation issues is still relevant.

Note also: CONCERT, summarized in Table 6.8: CONCERT summary, is a representative of early generation systems, before advent of DL architectures, when representations were designed with rich handcrafted features. 1 of benefits of using DL architectures: this kind of rich & deep representation may be automatically extracted & managed by architecture.

· 6.6.1.2. #2 Example: Celtic Melody Symbolic Music Generation System. Another representative example: system by Sturm et al. to generate Celtic music melodies [178]. Architecture used is a recurrent network with 3 hidden layers, which could notate [Note: as explained in Sect. 5.5.2, notate number of hidden layers without considering input layer.] as LSTM^3, with 512 LSTM cells in each layer.

Corpus comprises folk & Celtic monophonic melodies retrieved from a repository & discussion platform named The Session [98]. Pieces that were too short, too complex (with varying meters) or contained errors were filtered out, leaving a dataset of 23636 melodies. All melodies are aligned (transposed) onto single C key. 1 of specificities: representation chosen is *textual*, namely token-based *folk-rnn* notation, a transformation of character-based ABC notation (Sect. 4.7.3). Number of input & output nodes = number of tokens in vocabulary (i.e. with a 1-hot encoding), in practice = 137. Output of network is a probability distribution over its vocabulary.

Training recurrent network is done in an iterative way, as network learns to predict next item. Once trained, generation is done iteratively by inputting a random token or a specific token (e.g., corresponding to a specific starting note), feedforwarding it to generate output, sampling from this probability distribution, & recursively using selected vocabulary element as a subsequent input, in order to produce a melody element by element.

Final step: decode folk-rnn representation generated into a MIDI format melody to be played. See in Fig. 6.16: Score of "The Mal's Copporim" automatically generated [178]. for an example of a melody generated. One may also see & listen to results on [177]. Results are very convincing, with melodies generated in a clear Celtic style. System is summarized in Table 6.9: Celtic system summary.

As observed in [66]: "Interesting to note: in this approach bar lines & repeat bar lines are given explicitly & are to be predicted as well. This can cause some issues, since there is no guarantee: output sequence of tokens would represent a valid song in ABC format. There could be too many notes in 1 bar e.g., but according to authors, this rarely occurs. This would tend to show: such an architecture is able to learn to count." [On this issue, see also [59].]

◦ 6.7. Expressiveness. 1 limitation of most existing systems: they consider fixed dynamics (amplitude) for all notes as well as an exact quantization (a fixed tempo), which makes music generated too mechanical, without *expressiveness* or *nuance*.

A natural approach resides in considering representations recorded from real performances & not simply scores, & therefore with musically grounded (by skilled human musicians) variations of tempo & of dynamics, discussed in Sect. 4.10.

Note: an alternative approach: automatically *augment* generated music information (e.g., a standard MIDI piece) with slight transformations on amplitude &/or tempo. E.g.: Cyber-João system [29], which performs bossa nova guitar accompaniment with expressiveness, through automatic retrieval [By a mixed use of production rules & case-based reasoning (CBR).] &

application of rhythmic patterns [These patterns have been manually extracted from a corpus of performances by guitarist & single João Gilberto, 1 of inventors of Bossa nova style. One could also consider automatic extraction, as, e.g., in [31].]. As noted in Sect. 4.10.3, in case of an audio representation, expressiveness is implicit to representation. However, difficult to separately control expressiveness (dynamics or tempo) of a single instrument or voice as representation is global.

∗ 6.7.1. Example: Performance RNN Piano Polyphony Symbolic Music Generation System. +++

- 7. Analysis.
- 8. Discussion & Conclusion.

3.3 [HJE18]. Jiequn Hana , Arnulf Jentzen, Weinan E. Solving High-Dimensional PDEs Using Deep Learning

[2102 citations]

- Abstract. Developing algorithms for solving high-dimensional PDEs has been an exceedingly difficult task for a long time, due to notoriously difficult problem known as "curse of dimensionality". This paper introduces a DL-based approach that can handle general high-dimensional parabolic PDEs. To this end, PDEs are reformulated using backward stochastic differential equations & gradient of unknown solution is approximated by neural networks, very much in spirit of deep reinforcement learning with gradient acting as policy function. Numerical results on examples including nonlinear Black-Scholes equation, Hamilton–Jacobi–Bellman equation, & Allen–Cahn equation suggest: proposed algorithm is quite effective in high dimensions, in terms of both accuracy & cost. This opens up possibilities in economics, finance, operational research, & physics, by considering all participating agents, assets, resources, or particles together at same time, instead of making ad hoc assumptions on their interrelationships.
- Keywords. PDEs, backward stochastic differential equations, high dimension, DL, Feynman–Kac
- Intro. PDEs are among most ubiquitous tools used in modeling problems in nature. Some of most important ones are naturally formulated as PDEs in high dimensions. Well-known examples include following:

  1. Schrödinger equation in quantum many-body problem. In this case dimensionality of PDE is roughly 3 times number of electrons or quantum particles in system.
  2. Nonlinear Black–Scholes equation for pricing financial derivatives, in which dimensionality of PDE is number of underlying financial assets under consideration.
  3. Hamilton–Jacobi–Bellman equation in dynamic programming. In a game theory setting with multiple agents, dimensionality goes up linearly with number of agents. Similarly, in a resource allocation problem, dimensionality goes up linearly with number of devices & resources.

  As elegant as these PDE models are, their practical use has proved to be very limited due to curse of dimensionality (1): Computational cost for solving them goes up exponentially with dimensionality.

  Another area where curse of dimensionality has been an essential obstacle is ML & data analysis, where complexity of nonlinear regression models, e.g., goes up exponentially with dimensionality. In both cases essential problem we face is how to represent or approximate a nonlinear function in high dimensions. Traditional approach, by building functions using polynomials, piecewise polynomials, wavelets, or other basis functions, is bound to run into curse of dimensionality problem.

  In recent years a new class of techniques, deep neural network model, has shown remarkable success in AI, e.g., [2–6]. Neural network is an old idea but recent experience has shown: deep networks with many layers seem to do a surprisingly good job in modeling complicated datasets. In terms of representing functions, neural network model is compositional (thành phần): It uses compositions of simple functions to approximate complicated ones. It contrast, approach of classical approximation theory is usually additive. Mathematically, there are universally approximation theorems stating: a single hidden-layer network can approximate a wide class of functions on compact subsets (see, e.g., survey in [7] & refs therein), even though still lack a theoretical framework for explaining seemingly unreasonable effectiveness of multilayer neural networks, which are widely used nowadays. Despite this, practical success of deep neural networks in AI has been very astonishing & encourages applications to other problems where curse of dimensionality has been a tormenting issue (vấn đề đau khổ).

  In this paper, extend power of deep neural networks to another dimension by developing a strategy for solving a large class of high-dimensional nonlinear PDEs using DL. Class of PDEs dealed with is (nonlinear) parabolic PDEs. Special cases include Black–Scholes equation & Hamilton–Jacobi–Bellman equation. To do so, make use of reformulation of these PDEs as backward stochastic differential equations (BSDEs) (e.g., [8, 9]) & approximate gradient of solution using deep neural networks. Methodology bears some resemblance to deep reinforcement learning with BSDE playing role of model-based reinforcement learning (or control theory models) & gradient of solution playing role of policy function. Numerical examples manifest: proposed algorithm is quite satisfactory in both accuracy & computational cost.

  Remark 51 (Significance)**.** *PDEs are among most ubiquitous tools used in modeling problems in nature. However, solving high-dimensional PDEs has been notoriously difficult due to "curse of dimensionality". This paper introduces a practical algorithm for solving nonlinear PDEs in very high (hundreds & potentially thousands of) dimensions. Numerical results suggest: proposed*

algorithm is quite effective for a wide variety of problems, in terms of both accuracy & speed. Believe: this opens up a host of possibilities in economics, finance, operational research, & physics, by considering all participating agents, assets, resources, or particles together at same time, instead of making ad hoc assumptions on their interrelationships.

Due to curse of dimensionality, there are only a very limited number of cases where practical high-dimensional algorithms have been developed in literature. For linear parabolic PDEs, one can use Feynman–Kac formula & Monte Carlo methods to develop efficient algorithms to evaluate solutions at any given space-time locations. For a class of inviscid Hamilton–Jacobi equations, Darbon & Osher [10] recently developed an effective algorithm in high-dimensional case, based on Hopf formula for Hamilton–Jacobi equations. A general algorithm for nonlinear parabolic PDEs based on multilevel decomposition of Picard iteration is developed in [11] & has been shown to be quite efficient on a number of examples in finance & physics. Branching diffusion method is proposed in [12, 13], which exploits fact: solutions of semilinear PDEs with polynomial nonlinearity can be represented as an expectation of a functional of branching diffusion processes. This method does not suffer from curse of dimensionality, but still has limited applicability due to blow-up of approximated solutions in finite time.

Starting point of present paper is DL. Stressed: even though DL has been a very successful tool for a number of applications, adapting it to current setting with practical success is still a highly nontrivial task. Here by using reformulation of BSDEs, able to cast problem of solving PDEs as a learning problem & design a DL framework that fits naturally to that setting. This has proved to be quite successful in practice.

- Methodology. Consider a general class of PDEs known as semilinear parabolic PDEs. These PDEs can be represented as (1)

$$\partial_t u(t,x) + \frac{1}{2}\mathrm{Tr}(\sigma\sigma^\top(t,x)(\mathrm{Hess}_x u)(t,x)) + \nabla u(t,x)\cdot\mu(t,x) + f(t,x,u(t,x),\sigma^\top(t,x)\nabla u(t,x)) = 0$$

with some specified terminal condition $u(T,x) = g(x)$. Here t, x represent time & d-dimensional space variable, resp., μ: a known vector-valued function, σ: a known $d\times d$ matrix-valued function, $\sigma^\top$ denotes transpose associated to σ, ∇u & $\mathrm{Hess}_x u$ denote gradient & Hessian of function u w.r.t. x, Tr denotes trace of a matrix, & f: a known nonlinear function. To fix ideas, interested in solution at $t = 0, x = \xi$ for some vector $\xi\in\mathbb{R}^d$.

Let $\{W_t\}_{t\in[0,T]}$: a d-dimensional Brownian motion & $\{X_t\}_{t\in[0,T]}$: a d-dimensional stochastic process which satisfies (2)

$$X_t = \xi + \int_0^t \mu(s,X_s)\,\mathrm{d}s + \int_0^t \sigma(s,X_s)\,\mathrm{d}W_s.$$

Then solution of (1) satisfies following BSDE (cf., e.g., [8, 9]): (3)

$$u(t,X_t) - u(0,X_0) = -\int_0^t f(s,X_s,u(s,X_s),\sigma^\top(s,X_s)\nabla u(s,X_s))\,\mathrm{d}s + \int_0^t [\nabla u(s,X_s)]^\top\sigma(s,X_s)\,\mathrm{d}W_s.$$

To derive a numerical algorithm to compute $u(0,X_0)\approx\theta_{u_0}, \nabla u(0,X_0)\approx\theta_{\nabla u_0}$ as parameters in model & view (3) as a way of computing values of u at terminal time T, knowing $u(0,X_0)$ & $\nabla u(t,X_t)$. Apply a temporal discretization to (2)–(3). Given a partition of time interval $[0,T]$: $0 = t_0 < t_1 < \cdots < t_N = T$, consider simple Euler scheme for $n = 1,\ldots,N-1$: (4)–(5)

$$\begin{aligned} X_{t_{n+1}} - X_{t_n} &\approx \mu(t_n,X_n)\Delta t_n + \sigma(t_n,X_{t_n})\Delta W_n,\\ u(t_{n+1},X_{t_{n+1}}) - u(t_n,X_{t_n}) &\approx -f(t_n,X_{t_n},u(t_n,X_{t_n}),\sigma^\top(t_n,X_{t_n})\nabla u(t_n,X_{t_n}))\Delta t_n + [\nabla u(t_n,X_{t_n})]^\top\sigma(t_n,X_{t_n})\Delta W_n, \end{aligned}$$

where $\Delta t_n = t_{n+1} - t_n$, $\Delta W_n = W_{t_{n+1}} - W_{t_n}$. Given this temporal discretization, path $\{X_{t_n}\}_{0\le n\le N}$ can be easily sampled using (4). Key step next: approximate function $x\mapsto\sigma^\top(t,x)\nabla u(t,x)$ at each time step $t = t_n$ by a multilayer feedfoward neural network (7)

$$\sigma^\top(t_n,X_{t_n})\nabla u(t_n,X_{t_n}) = (\sigma^\top\nabla u)(t_n,X_{t_n}) \approx (\sigma^\top\nabla u)(t_n,X_{t_n}|\theta_n),\ n = 1,\ldots,N-1,$$

where θ_n denotes parameters of neural network approximating $x\mapsto\sigma^\top(t,x)\nabla u(t,x)$ at $t = t_n$.

Therefore, stack all of subnetworks in (7) together to form a deep neural network as a whole, based on summation of (5) over $n = 1,\ldots,N-1$. Specifically, this network takes paths $\{X_{t_n}\}_{0\le n\le N}, \{W_{t_n}\}_{0\le n\le N}$ as input data & gives final output, denoted by $\hat{u}(\{X_{t_n}\}_{0\le n\le N}, \{W_{t_n}\}_{0\le n\le N})$, as an approximation of $u(t_N, X_{t_N})$. Refer to *Materials & Methods* for more details on architecture of neural network. Difference in matching of a given terminal condition can be used to define expected loss function (8)

$$l(\theta) = \mathbb{E}\left[|g(X_{t_N}) - \hat{u}(\{X_{t_n}\}_{0\le n\le N}, \{W_{t_n}\}_{0\le n\le N})|^2\right].$$

Total set of parameters is $\theta = \{\theta_{u_0}, \theta_{\nabla u_0}, \theta_1, \ldots, \theta_{N-1}\}$.

Can now use a stochastic gradient descent-type (SGD) algorithm to optimize parameter θ, just as in standard training of deep neural networks. In numerical examples, use Adam optimizer [14]. See *Materials & Methods* for more details on training of deep neural networks. Since BSDE is used as an essential tool, call methodology introduced above deep BSDE method.

- Examples.
  - Nonlinear Black–Scholes Equation with Default Risk.

- Hamilton–Jacobi–Bellman Equation.
- Allen–Cahn Equation.

- Conclusions. Algorithm proposed in this paper opens up a host of possibilities in several different areas. E.g., in economics one can consider many different interacting agents at same time, instead of using "representative agent" model. Similarly in finance, one can consider all of participating instruments at same time, instead of relying on ad hoc assumptions about their relationships. In operational research, one can handle cases with hundreds & thousands of participating entities directly, without need to make ad hoc approximations.

  Note: although methodology presented here is fairly general, so far not able to deal with quantum many-body problem due to difficulty in dealing with Pauli exclusion principle.

- Materials & Methods.

3.4 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.

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)

$$y_m = \mathcal{E}(\mathbf{x}_m). \tag{250}$$

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

$$(\mathbf{x}_1, y_1) = (\mathbf{x}_1, \mathcal{E}(\mathbf{x}_1)), \ldots, (\mathbf{x}_M, y_M) = (\mathbf{x}_M, \mathcal{E}(\mathbf{x}_M)) \in \mathbb{R}^d \times \mathbb{R}. \tag{251}$$

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

$$\mathfrak{L}(\phi) = \frac{1}{M} \left(\sum_{i=1}^{M} |\phi(\mathbf{x}_i) - y_m|^2 \right), \ \forall \phi \in C(\mathbb{R}^d, \mathbb{R}). \tag{252}$$

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

$$\mathfrak{L} : C(\mathbb{R}^d, \mathbb{R}) \to [0, \infty). \tag{253}$$

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

$$\mathcal{L} = \mathfrak{L} \circ \psi. \tag{254}$$

Think of set (6)

$$\{\psi_\theta : \theta \in \mathbb{R}^\mathfrak{d}\} \subseteq C(\mathbb{R}^d, \mathbb{R}) \tag{255}$$

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

$$\mathbb{R}^\mathfrak{d} \ni \theta \mapsto \psi_\theta \in C(\mathbb{R}^d, \mathbb{R}) \tag{256}$$

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: $\forall \theta = (\theta_1, \ldots, \theta_{\mathfrak{d}}) \in \mathbb{R}^{\mathfrak{d}}, x \in \mathbb{R}$ it holds: (8)

$$\psi_\theta(x) = \sum_{k=0}^{\mathfrak{d}-1} \theta_{k+1} x^k \tag{257}$$

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)

$$\{\psi_\theta : \theta \in \mathbb{R}^{\mathfrak{d}}\} \ni \phi \mapsto \mathfrak{L}(\phi) = \frac{1}{M} \left(\sum_{m=1}^{M} |\phi(\mathbf{x}_m) - y_m|^2 \right) \in [0, \infty). \tag{258}$$

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

$$\mathbb{R}^{\mathfrak{d}} \ni \theta \mapsto \mathcal{L}(\theta) = \mathfrak{L}(\psi_\theta) = \frac{1}{M} \left(\sum_{m=1}^{M} |\psi_\theta(\mathbf{x}_m) - y_m|^2 \right) \in [0, \infty), \tag{259}$$

& 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)

$$\psi_\vartheta(\mathbf{x}_{M+1}) \approx \mathcal{E}(\mathbf{x}_{M+1}) \tag{260}$$

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 9 (Affine functions)**.** *Let $\mathfrak{d}, m, n \in \mathbb{N}, s \in \mathbb{N}_0, \theta = (\theta_1, \dots, \theta_{\mathfrak{d}}) \in \mathbb{R}^{\mathfrak{d}}$ 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 $\forall \mathbf{x} = (x_1, \dots, x_n) \in \mathbb{R}^n$:*

$$\mathcal{A}^{\theta,s}_{m,n}(\mathbf{x}) = \left([\sum_{k=1}^{n} x_k \theta_{s+k}] + \theta_{s+mn+1}, [\sum_{k=1}^{n} x_k \theta_{s+n+k}] + \theta_{s+mn+2}, \dots, [\sum_{k=1}^{n} x_k \theta_{s+(m-1)n+k}] + \theta_{s+mn+m} \right), \tag{261}$$

& 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 10 (Vectorized description of fully-connected feedforward ANNs)**.** *Let $\mathfrak{d}, L \in \mathbb{N}, l_0, l_1, \dots, l_L \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ satisfy*

$$\mathfrak{d} \geq \sum_{k=1}^{L} l_k (l_{k-1} + 1) \tag{262}$$

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

$$(\mathcal{N}^{\theta,l_0}_{\Psi_1,\dots,\Psi_L})(\mathbf{x}) = (\Psi_L \circ \mathcal{A}^{\theta,\sum_{k=1}^{L-1} l_k(l_{k-1}+1)}_{l_L,l_{L-1}} \circ \Psi_{L-1} \circ \mathcal{A}^{\theta,\sum_{k=1}^{L-2} l_k(l_{k-1}+1)}_{l_{L-1},l_{L-2}} \circ \Psi_2 \circ \mathcal{A}^{\theta,l_1(l_0+1)}_{l_2,l_1} \circ \Psi_1 \circ \mathcal{A}^{\theta,0}_{l_1,l_0})(\mathbf{x}), \tag{263}$$

& call $\mathcal{N}^{\theta,l_0}_{\Psi_1,\dots,\Psi_L}$ realization function *or* realization *of fully-connected feedforward ANN associated to θ with $L+1$ layers with dimensions $(l_0, l_1, \dots, l_L)$ & activation functions $(\Psi_1, \dots, \Psi_L)$.*

∗ 1.1.3. Weight & bias parameters of fully-connected feedforward ANNs.

Remark 52 (Weights & biases for fully-connected feedforward ANNs)**.** *Let $L \in \{2, 3, \dots\}, v_0, v_1, \dots, v_{L-1} \in \mathbb{N}_0, l_0, l_1, \dots, l_L, \mathfrak{d} \in \mathbb{N}, \theta = (\theta_1, \dots, \theta_{\mathfrak{d}}) \in \mathbb{R}^{\mathfrak{d}}$ satisfy $\forall k \in \{0, 1, \dots, L-1\}$:*

$$\mathfrak{d} \geq \sum_{i=1}^{L} l_i (l_i + 1), \ v_k = \sum_{i=1}^{k} l_i (l_{i-1} + 1), \tag{264}$$

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

$$W_k = \textit{weight parameters}, \ b_k = \textit{bias parameters}, \tag{265}$$

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

· *(i) it holds*

$$\mathcal{N}^{\theta,l_0}_{\Psi_1,\dots,\Psi_L} = \Psi_L \circ \mathcal{A}^{\theta,v_{L-1}}_{l_L,l_{L-1}} \circ \Psi_{L-1} \circ \mathcal{A}^{\theta,v_{L-2}}_{l_{L-1},l_{L-2}} \circ \Psi_{L-2} \circ \cdots \circ \mathcal{A}^{\theta,v_1}_{l_2,l_1} \circ \Psi_1 \circ \mathcal{A}^{\theta,v_0}_{l_1,l_0}, \tag{266}$$

· *(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 11 (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*

$$\mathfrak{M}_{\psi,d_1,\dots,d_T} : \mathbb{R}^{d_1 \times \cdots \times d_T} \to \mathbb{R}^{d_1 \times \cdots d_T} \tag{267}$$

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

$$\mathfrak{M}_{\psi,d_1,\dots,d_T}(\mathbf{x}) = \mathbf{y}, \tag{268}$$

& 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 $\mathcal{I}, \mathcal{H} \in \mathbb{N}, \theta = (\theta_1, \ldots, \theta_{\mathcal{H}\mathcal{I}+2\mathcal{H}+1}) \in \mathbb{R}^{\mathcal{H}\mathcal{I}+2\mathcal{H}+1}, \mathbf{x} = (x_1, \ldots, x_{\mathcal{I}}) \in \mathbb{R}^{\mathcal{I}}$ & let $\psi : \mathbb{R} \to \mathbb{R}$ be a function. Then*

$$\mathcal{N}_{\mathfrak{M}_\psi, \mathcal{H}, \mathrm{id}_{\mathbb{R}}}(\mathbf{x}) = \left[\sum_{k=1}^{\mathcal{H}} \theta_{\mathcal{H}\mathcal{I}+\mathcal{H}+k} \psi\left(\left[\sum_{i=1}^{\mathcal{I}} x_i \theta_{(k-1)\mathcal{I}+i}\right] + \theta_{\mathcal{H}\mathcal{I}+k}\right)\right] + \theta_{\mathcal{H}\mathcal{I}+2\mathcal{H}+1}. \tag{269}$$

∗ 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 12 (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 13 (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 $\forall x \in \mathbb{R}$:*

$$x = W_1 \max\{w_1 x + b_1, 0\} + W_2 \max\{w_2 x + b_2, 0\} + B. \tag{270}$$

Problem 13 (Real identity). *Prove or disprove following statement: 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. $\forall x \in \mathbb{R}$: $(\mathcal{N}^{\theta,1}_{\mathfrak{R}_{l_1}, \ldots, \mathfrak{R}_{l_H}, \mathrm{id}_{\mathbb{R}}})(x) = x$.*

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 $\mathfrak{d}, H \in \mathbb{N}, l_1, \ldots, l_H \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ with $\mathfrak{d} \geq 2l_1 + [\sum_{k=2}^{H} l_k(l_{k-1}+1)] + l_H + 1$ s.t. $\forall x \in \mathbb{R}$, $(\mathcal{N}^{\theta,1}_{\mathfrak{R}_{l_1}, \ldots, \mathfrak{R}_{l_H}, \mathrm{id}_{\mathbb{R}}})(x) = |x|$.*

Problem 15 (Exponential). *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. $\forall x \in \mathbb{R}$, $(\mathcal{N}^{\theta,1}_{\mathfrak{R}_{l_1}, \ldots, \mathfrak{R}_{l_H}, \mathrm{id}_{\mathbb{R}}})(x) = e^x$.*

Problem 16 (2D maximum). *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 $\mathfrak{d} \geq 3l_1 + [\sum_{k=2}^{H} l_k(l_{k-1}+1)] + l_H + 1$ s.t. $\forall x, y \in \mathbb{R}$, $(\mathcal{N}^{\theta,2}_{\mathfrak{R}_{l_1}, \ldots, \mathfrak{R}_{l_H}, \mathrm{id}_{\mathbb{R}}})(x, y) = \max\{x, y\}$.*

Problem 17 (Real identity with 2 hidden layers). *Prove or disprove: There exist $\mathfrak{d}, H \in \mathbb{N}, l_1, \ldots, l_H \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ with $\mathfrak{d} \geq 2l_1 + l_1 l_2 + 2l_2 + 1$ 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 $\mathfrak{d}, H \in \mathbb{N}, l_1, \ldots, l_H \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ with $\mathfrak{d} \geq 4l_1 + [\sum_{k=2}^{H} l_k(l_{k-1}+1)] + l_H + 1$ s.t. $\forall x, y, z \in \mathbb{R}$, $(\mathcal{N}^{\theta,3}_{\mathfrak{R}_{l_1}, \ldots, \mathfrak{R}_{l_H}, \mathrm{id}_{\mathbb{R}}})(x, y, z) = \max\{x, y, z\}$.*

Problem 19 (Multidimensional maxima). *Prove or disprove: For every $k \in \mathbb{N}$, there exists $\mathfrak{d}, H \in \mathbb{N}, l_1, \ldots, l_H \in \mathbb{N}, \theta \in \mathbb{R}^{\mathfrak{d}}$ with $\mathfrak{d} \geq (k+1)l_1 + [\sum_{k=2}^{H} l_k(l_{k-1}+1)] + l_H + 1$ s.t. $\forall x, y, z \in \mathbb{R}$, $(\mathcal{N}^{\theta,k}_{\mathfrak{R}_{l_1}, \ldots, \mathfrak{R}_{l_H}, \mathrm{id}_{\mathbb{R}}})(x_1, \ldots, x_k) = \max\{x_1, \ldots, x_k\}$.*

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 14 (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 2. *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*

$$\mathbb{P}(\mathcal{T} \in A) = \int_A \mathfrak{t}(t)\,\mathrm{d}t, \ \mathbb{P}(\mathcal{X} \in B) = \int_B \mathfrak{x}(\mathbf{x})\,\mathrm{d}\mathbf{x}, \tag{271}$$

    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

$$\mathfrak{L}(v) = \mathbb{E}[|v(0,\mathcal{X}) - g(\mathcal{X})|^2 + |(\partial_t v)(\mathcal{T},\mathcal{X}) - (\Delta_{\mathbf{x}} v)(\mathcal{T},\mathcal{X}) - f(v(\mathcal{T},\mathcal{X}))|^2]. \tag{272}$$

    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* $u(0,\mathbf{x}) = g(\mathbf{x})$ *&*

$$\partial_t u(t,\mathbf{x}) = (\Delta_{\mathbf{x}} u)(t,\mathbf{x}) + f(u(t,\mathbf{x})). \tag{273}$$

- 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 $u(0,\mathbf{x}) = g(\mathbf{x})$ &

$$\partial_t u(t,\mathbf{x}) = (\Delta_{\mathbf{x}} u)(t,\mathbf{x}) + f(u(t,\mathbf{x})). \tag{274}$$

  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

$$\mathbb{P}(\mathcal{T} \in A) = \int_A \mathfrak{t}(t)\,\mathrm{d}t, \ \mathbb{P}(\mathcal{X} \in B) = \int_B \mathfrak{x}(\mathbf{x})\,\mathrm{d}\mathbf{x}, \tag{275}$$

  & let $\mathfrak{L} : C^{1,2}([0,T] \times \mathbb{R}^d, \mathbb{R}) \to [0,\infty]$ satisfy $\forall v = (v(t,\mathbf{x}))_{(t,\mathbf{x}) \in [0,T] \times \mathbb{R}^d} \in C^{1,2}([0,T] \times \mathbb{R}^d, \mathbb{R})$:

$$\mathfrak{L}(v) = \mathbb{E}[|v(0,\mathcal{X}) - g(\mathcal{X})|^2 + |\partial_t v(\mathcal{T},\mathcal{X}) - (\Delta_{\mathbf{x}} v)(\mathcal{T},\mathcal{X}) - f(v(\mathcal{T},\mathcal{X}))|^2]. \tag{276}$$

  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 $h \in \mathbb{N}, l_1, \ldots, l_h, \mathfrak{d} \in \mathbb{N}$ 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 $\forall \theta \in \mathbb{R}^{\mathfrak{d}}$:

$$\mathcal{L}(\theta) = \ldots \tag{277}$$

  (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 $\mathcal{N}_{\mathfrak{M}_{a,l_1}, \mathfrak{M}_{a,l_2}, \ldots, \mathfrak{M}_{a,l_h}, \mathrm{id}_{\mathbb{R}}}$ 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 $\xi \in \mathbb{R}^{\mathfrak{d}}, J \in \mathbb{N}, (\gamma_n)_{n \in \mathbb{N}} \subseteq [0,\infty), \forall n \in \mathbb{N}, j \in \{1,2,\ldots,J\}$ let $\mathfrak{T}_{n,j} : \Omega \to [0,T]$ & $\mathfrak{X}_{n,j} : \Omega \to \mathbb{R}^d$ be random variables, assume $\forall n \in \mathbb{N}$, $j \in \{1,\ldots,J\}$, $A \in \mathcal{B}([0,T]), B \in \mathcal{B}(\mathbb{R}^d)$:

$$\mathbb{P}(\mathcal{T} \in A) = \mathbb{P}(\mathfrak{T}_{n,j} \in A), \ \mathbb{P}(\mathcal{X} \in B) = \mathbb{P}(\mathfrak{X}_{n,j} \in B), \tag{278}$$

  [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.

3.5 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

$$\mathbb{R}^d \ni \mathbf{x} \mapsto \nu(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b), \tag{279}$$

      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 $\mathbb{R}_+ := [0, \infty)$, 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

$$\mathbb{R}^d \ni \mathbf{x} \mapsto \Phi(\mathbf{x}) = T_1 \circ \sigma \circ T_0(\mathbf{x}), \tag{280}$$

      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

$$\mathbb{R}^d \ni \mathbf{x} \mapsto \Phi(\mathbf{x}) = T_{L+1} \circ \sigma \circ \cdots T_1 \circ \sigma \circ T_0(\mathbf{x}), \tag{281}$$

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)

$$\Phi(\mathbf{x}_i) \approx \mathbf{y}_i, \ \forall i = 1, \ldots, m, \tag{282}$$

in a meaningful sense. E.g., could interpret "$\approx$" 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

$$\widehat{\mathcal{R}}_S(\Phi) \coloneqq \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\Phi(\mathbf{x}_i), \mathbf{y}_i). \tag{283}$$

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 $\Phi_1, \Phi_2, \ldots$, for which empirical risk is expected to decrease. Fig. 1.3: A sequence of 1D neural networks $\Phi_1, \ldots, \Phi_4$ 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

$$\mathcal{R}(\Phi_*) = \mathbb{E}_{(\mathbf{x}_{\text{new}}, \mathbf{y}_{\text{new}}) \sim \mathcal{D}}[\mathcal{L}(\Phi_*(\mathbf{x}_{\text{new}}), \mathbf{y}_{\text{new}})], \tag{284}$$

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. $\Phi(\mathbf{x}_i) \approx \mathbf{y}_i$, $\forall i = 1, \ldots, m$. 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 (279) & Fig. 1.1. A neural network is constructed by connecting multiple neurons. Make precise this connection procedure:

    Definition 15 (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)*

$$\mathbf{x}^{(0)} \coloneqq \mathbf{x}, \tag{285}$$

$$\mathbf{x}^{(l)} \coloneqq \sigma(\mathbf{W}^{(l-1)}\mathbf{x}^{(l-1)} + b^{(l-1)}), \ \forall l \in \{1, \dots, L\}, \tag{286}$$

$$\mathbf{x}^{(L+1)} \coloneqq \mathbf{W}^{(L)}\mathbf{x}^{(L)} + \mathbf{b}^{(L)} \tag{287}$$

    holds

$$\Phi(\mathbf{x}) = \mathbf{x}^{(L+1)}, \ \forall \mathbf{x} \in \mathbb{R}^{d_0}. \tag{288}$$

    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 53. *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

$$\mathbf{w} = ((\mathbf{W}^{(0)}, \mathbf{b}^{(0)}), \dots, (\mathbf{W}^{(L)}, \mathbf{b}^{(L)})). \tag{289}$$

      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 (289) 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 16 (Size of neural network)**.** *Let Φ be as in Def. 2.1. Then* size *of Φ is*

$$\text{size}(\Phi) := \left| \left(\{(i,k,l) | W_{k,l}^{(i)} \neq 0\} \cup \{(i,k) | b_k^{(i)} \neq 0\} \right) / \sim \right|. \tag{290}$$

◦ 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

$$\sigma_{\text{sig}}(x) = \frac{1}{1+e^{-x}}, \ \forall x \in \mathbb{R}. \tag{291}$$

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

$$\frac{d}{db}\Phi_n(x) = \sigma'(\Phi_{n-1}(x))\frac{d}{db}\Phi_{n-1}(x). \tag{292}$$

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

$$\Phi(x) = \sigma_{\mathrm{ReLU}}(b - \sigma_{\mathrm{ReLU}}(x)) \ \forall x \in \mathbb{R} \tag{293}$$

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

$$\sigma_{\mathrm{SiLU}}(x) \coloneqq x\sigma_{\mathrm{sig}}(x) = \frac{x}{1 + e^{-x}}, \ \forall x \in \mathbb{R}. \tag{294}$$

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

$$\sigma_a(x) = \max\{x, ax\}, \ \forall x \in \mathbb{R}, \tag{295}$$

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. $\forall l = 0, \dots, L$, $\|\mathbf{W}^{(l)} - \widetilde{\mathbf{W}}^{(l)}\| \leq \varepsilon$, $\|\mathbf{b}^{(l)} - \widetilde{\mathbf{b}}^{(l)}\| \leq \varepsilon$ 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 17. *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 18. *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 3 (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 25 (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 19. *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*

$$\mathcal{N}_d^m(\sigma; L, n) \coloneqq \{\Phi : \mathbb{R}^d \to \mathbb{R}^m | \Phi \text{ as in Def. 2.1},\ \operatorname{depth}(\Phi) \leq L,\ \operatorname{width}(\Phi) \leq n\}. \tag{296}$$

Furthermore,

$$\mathcal{N}_d^m(\sigma; L) \coloneqq \bigcup_{n\in\mathbb{N}} \mathcal{N}_d^m(\sigma; L, n). \tag{297}$$

In sequel, require activation function σ to belong to set of piecewise continuous & locally bounded functions

$$\mathcal{M} \coloneqq \{\sigma \in L^\infty_{\text{loc}}(\mathbb{R}) | \text{there exists intervals } I_1, \dots, I_M \text{ partitioning } \mathbb{R},\ \text{s.t. } \sigma \in C^0(I_i),\ \forall i = 1, \dots, M\}. \tag{298}$$

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 26. *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 $x \mapsto \mathbf{1}_{\mathbf{x}>\mathbf{0}}$ (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 4. *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 54. *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 $C^0(\mathbb{R}) \subseteq \overline{\mathcal{N}_1^1(\sigma; 1)}^{\text{cc}}$ then $C^0(\mathbb{R}^d) \subseteq \overline{\mathcal{N}_d^1(\sigma; 1)}^{\text{cc}}$,

(ii) if $\sigma \in C^\infty(\mathbb{R})$ is not a polynomial then $C^0(\mathbb{R}) \subseteq \overline{\mathcal{N}_1^1(\sigma; 1)}^{\text{cc}}$,

(iii) if $\sigma \in \mathcal{M}$ is not a polynomial then there exists $\widetilde{\sigma} \in C^\infty(\mathbb{R}) \cap \overline{\mathcal{N}_1^1(\sigma; 1)}^{\text{cc}}$ 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.

3.6 [Zha+23]. Aston Zhang, Zachary C. Lipton, Mu Li, Alexander J. Smola. Dive into Deep Learning

[51 Amazon ratings]

Amazon review. DL has revolutionized pattern recognition, introducing tools that power a wide range of technologies in such diverse fields as computer vision, natural language processing, & automatic speech recognition. Applying DL requires you to simultaneously understand how to cast a problem, basic mathematics of modeling, algorithms for fitting your models to data, & engineering techniques to implement it all. This book is a comprehensive resource that makes DL approachable, while still providing sufficient technical depth to enable engineers, scientists, & students to use DL in their own work. No previous background in ML or DL is required – every concept is explained from scratch & appendix provides a refresher on mathematics needed. Runnable code is featured throughout, allowing you to develop your own intuition by putting key ideas into practice.

Editorial Reviews.

- "In < a decade, AI revolution has swept from research labs to broad industries to every corner of our daily life. Dive into DL is an excellent text on DL & deserves attention from anyone who wants to learn why DL has ignited AI revolution: most powerful technology force of our time." – Jensen Huang, Founder & CEO, NVIDIA
- "This is a timely, fascinating book, providing not only a comprehensive overview of DL principles but also detailed algorithms with hands-on programming code, & moreover, a state-of-art introduction to DL in computer vision & natural language processing. Dive into this book if want to dive in DL!" – Jiawei Han, Michael Aiken Chair Professor, University of Illinois at Urbana-Champaign
- "This is a highly welcome addition to ML literature, with a focus on hands-on experience implemented via integration of Jupyter notebooks. Students of DL should find this invaluable to become proficient in this field." – Bernhard Schölkopf, Director, Max Planck Institute for Intelligent System

Book Description. An approachable text combining depth & quality of a textbook with interactive multi-framework code of a hands-on tutorial.

About Author. Aston Zhang is Senior Scientist at Amazon Web Services.
Zachary C. Lipton is Assistant Professor of Machine Learning and Operations Research at Carnegie Mellon University.
Mu Li is Senior Principal Scientist at Amazon Web Services.
Alexander J. Smola is VP/Distinguished Scientist for Machine Learning at Amazon Web Services.

- 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:

1. Start off with a randomly initialized model that cannot do anything useful.
2. Grab some of your data (e.g., audio snippets & corresponding {yes, no} labels).
3. Tweak knobs to make model perform better as assessed on those examples.
4. 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:

  1. *Data* that we can learn from.
  2. A *model* of how to transform data.
  3. An *objective function* that quantifies how well (or badly) model is doing.
  4. 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.[20] When have more data, can train more powerful models & rely

[20] NQBH: Really? Or more illusion, delusion?

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 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.

1. **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").
2. **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.
3. **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.
4. **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 oppo-

nents' 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.
  1. 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?
  2. 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.
  3. Describe relationships between algorithms, data, & computation. How do characteristics of data & current available computational resources influence appropriateness of various algorithms?
  4. 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:

1. techniques for storing & manipulating data
2. libraries for ingesting & preprocessing data from a variety of sources
3. knowledge of basic linear algebraic operations that we apply to high-dimensional data elements
4. just enough calculus to determine which direction to adjust each parameter in order to decrease loss function
5. ability to automatically compute derivatives so that you can forget much of calculus you just learned
6. some basic fluency in probability, our primary language for reasoning under uncertainty
7. 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.

3.7 [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.

3.8 Son N. T. Tu, Thu Nguyen. FinNet: Finite Difference Neural Network for Solving Differential Equations. 2022. arXiv

[2 citations]

- Abstract. DL approaches for PDEs have received much attention in recent years due to their mess-freeness & computational efficiency. However, most of works so far have concentrated on time-dependent nonlinear differential equations. In this work, analyze potential issues with well-known Physics Informed Neural Network for differential equations with little constraints on boundary (i.e., constraints are only on a few points). This analysis motivates us to introduce a novel technique called FinNet, for solving differential equations by incorporating FD into DL. Even though use a mesh during training, prediction phase is mesh-free. Illustrate effectiveness of our method through experiments on solving various equations, which shows: FinNet can solve PDEs with low error rates & may work even when PINNs cannot.

- 1. Introduction. Differential equations play a crucial role in many aspects of modern world, from technology to supply chain, economics, operational research, & finance [1]. Solving these equations numerically has been an extensive area of research since 1st conception of modern computer. Yet, there are some potential drawbacks of classical methods, e.g. FD & FE. 1stly, *curse of dimensionality*, i.e., computational cost, increases exponentially with dimension of equation [2]. 2ndly, classical methods usually need a mesh [3, 4]. With advacement of DL, there have been many works on using neural networks to solve differential equations that potentially can shed light on resolving above difficulties [5, 6].

  1 of foundational works in DL for solving PDEs is PINNs [7]. Here, a neural network is trained to solve supervised learning tasks w.r.t. given laws of physics described by nonlinear PDEs. Various variants or extension of this method exist. E.g., XPINNs [8] is a generalized space-time domain decomposition framework for PINNs to solve nonlinear PDEs in arbitrary complex-geometry domains. Another example is PhyGeoNet [9], a CNN-based variant of PINNs for solving PDEs in an irregular domain.

  In another work [1], authors try to address curse of dimensionality in high-dimensional semilinear parabolic PDEs by reformulating PDEs using backward stochastic differential equations & approximating gradient of unknown solution by deep reinforcement learning with gradient acting as policy function. Further notable work on high-dimensional PDEs is Deep Galerkin Method [10], in which solution is approximated by a neural network trained to satisfy differential operator, initial condition, & boundary conditions using batches of randomly sampled time & space points. In addition, authors in [11] consider using deep neural network for high-dimensional elliptic PDEs with boundary conditions.

  Furthermore, SPINN [12] is a recently developed method that uses an interpretable sparse network architecture for solving PDEs & authors in [13] propose a deep ReLU neural network approximation of parametric & stochastic elliptic PDEs with lognormal inputs.

  However, most of works in field of DL for differential equations are for time-dependent PDEs [7, 8, 10, 1]. Therefore, it would be interesting to explore how DL techniques can be used in other scenarios. In this work, illustrate via examples that applying PINNs to certain PDEs may not give desirable results. Investigate potential reasons for such problems & propose a novel method, namely Finite Difference Network (FinNet), that uses neural networks & FD to solve such equations.

  Main contributions of this work:

  1. Show examples PINNs fails to work for PDEs with very few constraints on boundary & analyze potential reason
  2. Propose FinNet, a method based on FD & neural network to solve PDE with little constraints on boundary
  3. Illustrate via various examples that FinNet can solve PDEs efficiently, even when PINNs cannot
  4. Discuss open problems for future research.

  Rest of paper is organized as follows:

  - Sect. 2 gives some preliminaries on PINNs for solving time-dependent nonlinear PDEs.
  - Sect. 3 explore potential issues with applying PINNs for some differential equations that are not time-dependent nonlinear, analyze examples, & give motivations to FinNet approach.
  - Sect. 4 details FinNet method
  - Sect. 5 gives various examples on applying FinNet to solve differential equations.
  - Sect. 6: end with a conclusion of this work & open questions.

- 2. Preliminaries: Physics Informed Neural Networks. PINNs [7] considers parameterized & nonlinear PDEs of form $u_t+\mathcal{N}[u;\lambda]=0$, where $u(t,x)$: latent solution (giải pháp tiềm ẩn), & $\mathcal{N}[\cdot;\lambda]$ is a nonlinear operator parameterized by λ. It defines $f \coloneqq u_t+\mathcal{N}[u]$, & approximates $u(t,x)$ by a neural network. then, parameters of neural network & $f(t,x)$ can be learned by minimizing mean squared error (MSE) loss

$$L=\frac{1}{N_u}\sum_{i=1}^{N_u}|u(t_u^i,x_u^i)-u^i|^2+\frac{1}{N_f}\sum_{i=1}^{N_f}|f(t_f^i,x_f^i)|^2,$$

where $\{t_u^i,x_u^i,u^i\}_{i=1}^N$: initial & boundary training data on $u(t,x)$ & $\{t_f^i,x_f^i\}_{i=1}^{N_f}$: collocations points for $f(t,x)$.
E.g., consider solving Burgers equation with Dirichlet boundary conditions

$$\begin{cases} u_t+uu_x-\dfrac{0.01}{\pi}u_{xx}=0,\ x\in[-1,1],\ t\in[0,1], \\ u(0,x)=\sin\pi x, \\ u(t,-1)=u(t,1)=0. \end{cases}$$

Then PINNs defines

$$f=u_t+uu_x-\frac{0.01}{\pi}u_{xx},$$

& approximate $u(t,x)$ by a neural network. Next, parameters of neural network $u(t,x)$ can be learned by minimizing MSE:

$$L=\frac{1}{N_u}\sum_{i=1}^{N_u}|u(t_u^i,x_u^i)-u^i|^2+\frac{1}{N_f}\sum_{i=1}^{N_f}|f(t_f^i,x_f^i)|^2,$$

where $\{t_u^i,x_u^i,u^i\}_{i=1}^N$: initial & boundary training data on $u(t,x)$ & $\{t_f^i,x_f^i\}_{i=1}^{N_f}$: collocations points for $f(t,x)$.

- 3. Motivation. In this sect, 1st illustrate via examples: in some cases, applying PINNs to solve differential equations may not lead to convergence towards desired solution. Attempt to explain potential reasons why such an issue can arise & by this, provides motivation for our approach.
  - Example 1. Consider equation (5)

$$\begin{cases} u'(x)+u(x)=x,\ x\in(0,1), \\ u(0)=1. \end{cases}$$

Exact solution: $u^*(x)=x-1+2e^{-x}$. To solve this equation by PINNs, approximate u by a neural network with 4 layers, each layer has 32 neurons, & tanh as activation function. Train network with 5000 epochs & following loss function

$$L=\frac{1}{99}\sum_{i=1}^{99}\left(\frac{d\hat{u}}{dx_i}+\hat{u}-x_i\right)^2+|\hat{u}(0)-1|^2.$$

Here, $x_1=0.01,x_2=0.02,\ldots,x_{99}=0.99$: training data.
After 5000 epochs, loss becomes as low as $5.15\cdot10^{-5}$. Yet Fig. 1: Left: Approximation by PINNs compared to true solution for (5). Right: true solution vs. neural network solution for (8). shows: approximation from neural network is not close to true solution. Examining gradients shows: $u'(x)\approx0.0125$ at all interior points (mean of $u'(x_i)$, $i=1,\ldots,n$ is 0.0125 & variance is 0.0001).
  - 3.2. Example 2: 2nd order static equation. Attempted to solve following initial boundary equation

$$\begin{cases} u''(x)+u(x)=e^{-x},\ x\in(0,1) \\ u(0)=1,\ u(1)=\dfrac{1}{2}\cos1+\dfrac{1}{2}\sin1+\dfrac{1}{2e}. \end{cases}$$

Exact solution (viscosity solution) is

$$u^*(x)=\frac{\sin x+\cos x+e^{-x}}{2}.$$

In an attempt to solve this equation by PINNs, approximate u using a neural network with 4 layers, where each layer has 32 neurons & a tanh activation function. Train network with 5000 epochs & following loss function

$$L=\frac{1}{99}\sum_{i=1}^{99}\left(\frac{d^2\hat{u}}{dx_i^2}+\hat{u}(x_i)-e^{-x_i}\right)^2+\frac{1}{2}\left(|\hat{u}(0)-1|^2+\left|\hat{u}(1)-\frac{1}{2}\cos1+\frac{1}{2}\sin1+\frac{1}{2e}\right|^2\right).$$

Here $x_1=0.01,x_2=0.02,\ldots,x_{99}=0.99$: training data. Approximated solution produced by PINNs is provided in Fig. 1 right.
After 50 epochs, loss reduces to 2.88 & then stays approximately same throughout epoch 51 to epoch 5000. From Fig. 1, can see: approximation from neural network is almost constant rather than being close to true solution. Examining gradients shows: $u''(x)\approx0$ at all interior points (mean of $u'(x_i)$, $i=1,\ldots,n$ is $-7.86\cdot10^{-5}$ & variance is $9.53\cdot10^{-9}$). Hence, can say: neural network get stuck at a local minima in this case.

- 3.3. Analysis & Motivation for FinNet. By *Universal approximation theorem* for neural network [14,15], PINNs' approximation is always possible given enough parameters. However, from examples above, see: applying PINNs to certain kinds of differential equations may not give a desirable result, & network may get stuck at a local minimum. However, note: training in this manner does not involve any label, & PINNs seem to work well for nonlinear time-dependent PDEs as studied in [7]. Furthere, without boundary constraints, a PDE fails to have a unique solution. In addition, when training a neural network to solve a differential equation, need to inform network about constraint on boundary. Next, recall: in (4) on L formula, constraints on boundary is informed to network via term $\frac{1}{N_f}\sum_{i=1}^{N_f}|f(t_f^i, x_f^i)|^2$, which is based on N_f points. For a time-dependent equation, N_f can be reasonably large & feed into network enough information for convergence to a desirable result. However, for PDEs in (5) & (8) boundary consists of only 2 points.

  This motivates to provide more instructions for neural network learning process by incorporating FD mechanism into network, which informs network: data points should satisfy conditions stated by FD. In addition, $u(x, y)$ is known at boundary. E.g., in (18), boundary is known to be

$$u(0) = 1,\ u(1) = \frac{1}{2}\cos 1 + \frac{1}{2}\sin 1 + \frac{1}{2e}.$$

  Therefore, instead of minimizing MSE as in (1), use this information along with FD to estimate derivative terms. This helps estimate derivatives at boundary more accurately & provides learning process with better instructions on what network should satisfy.

- 4. Finite Difference Network (FinNet). This sect details FD network (FinNet) approach. Assume: have a function $f : \mathbb{R} \to \mathbb{R}$, & a (uniform) mesh $\ldots, x_{i-2}, x_{i-1}, x_i, x_{i+1}, x_{i+2}, \ldots$ with $h = x_{i+1} - x_i$. Then, recall: by using FD, 1st order derivative $f'(x_i)$ can be computed approximately by 1 of following 3 formulas

$$f'(x_i) \approx \frac{f(x_{i+1}) - f(x_i)}{h},\ f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{h},\ f'(x_i) \approx \frac{f(x_{i+1}) - f(x_{i-1})}{2h},$$

  & 2nd order derivative can be approximated by

$$\frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2},$$

  & for general case where $f : \mathbb{R}^n \to \mathbb{R}$ then derivative terms are estimated by using above univariate FD scheme to partial derivatives of f.

  Next, define some defs & notations in Table 1: Table of Notations

  1. Ω: an open subset of $\mathbb{R}^n$
  2. $\partial\Omega$: boundary of Ω
  3. G: a set of meshgrid points
  4. B: a set of boundary points, $B \subset G$
  5. u^*: true solution
  6. v: a neural network that approximates u^*
  7. L: loss function
  8. N: mesh grid size
  9. MSE$(\mathbf{a}, \mathbf{b})$: mean squared error between vectors $\mathbf{a}, \mathbf{b}$.

  For a continuous operator F, to solve problem:

$$\begin{cases} F(x, u, Du, D^2u) = 0 & \text{in } \Omega, \\ u = g & \text{on } \partial\Omega. \end{cases}$$

  Discretize $[a, b] = \{x_1, \ldots, x_N\}$ & for simplicity use uniform mesh $\Delta = \frac{b-a}{N+1}$ as distance between 2 consecutive points.

  Question 2 (FDMs on nonuniform meshes)**.** *Possible to generalize this framework to non-uniform mesh? Read [LeV07].*

  Finnet strategy for solving differential equations is as given in Algorithm 1: FinNet. Given a neural network model v, train network as following: For each epoch, 1st compute $\hat{u} \leftarrow v(G), \hat{u}_B \leftarrow v(B)$. Note: $B \subset G$ so computation of $\hat{u}_B \leftarrow v(B)$ is already done in $\hat{u} \leftarrow v(G)$ operation. Though, write it down to clarity of $\hat{u}_B$ notation. Then, initialize loss L with MSE loss at boundary: $L \leftarrow \text{MSE}(\hat{u}_B, g(B))$. This is to ensure: constraint $u = g$ on $\partial\Omega$ is satisfied. Next, update boundary values of $\hat{u}$ with already known exact values based on $u = g$ on $\partial\Omega$ as in (13). This is done by assigning $\hat{u}_B \leftarrow g(B)$. Based on this newly updated $\hat{u}$, estimate derivatives in F by finite difference. This later allows us to estimate F based on approximated terms. Then, update loss: $L \leftarrow L + \text{MSE}(F(x, u, \hat{D}u, \hat{D}^2u), 0)$. This is to ensure: condition $F(x, u, Du, D^2u) = 0$ in Ω is satisfied. After getting loss, update weights of neural network v.

  Note: Step "update boundary values of $\hat{u}$ with already known exact values based on $u = g$ on $\partial\Omega$ as in (13). This is done by assigning $\hat{u}_B \leftarrow g(B)$." is crucial. Estimating derivative terms by FD using this is more accurate than using predicted values of network on boundary. Another noteworthy point: since use FD during training phase, a mesh is needed at this stage. However, similar to PINNs, prediction phase is mesh-free.

- 5. Examples. Provide various examples on how FinNet can be used to solve differential equations. Source code for examples will be made available upon acceptance of paper.
  - 5.1. Example 1: Linear 1st-order equation. Consider (5) in Sect. 3. True solution: $u^*(x) = x - 1 + 2e^{-x}$. For this equation, let $F = u'(x) + u(x) - x$. Used a neural network of 2 hidden layers with 16 neurons/layer & hyperbolic tangent activation functions to approximate true solution. To learn parameters, use Adam optimizer with learning rate 0.01. In this case, $G = \{0, 0.01, 0.02, \ldots, 0.99, 1\}$, $B = \{0, 1\}$.

    SKIPPED DETAILS After getting loss, update weights of neural network v.

    After 5000 epochs, loss goes down to $3.34 \cdot 10^{-5}$, & mean square error between true solution & predicted values is $1.15 \cdot 10^{-7}$. Plot of true solution vs. neural network's approximated solution is shown in Fig. 2: Left: True solution vs. neural network's approximated solution for (15). Right: True solution vs. neural network's approximated solution for (18).
  - 5.2. Example 2: 2nd-order linear equation. Consider following initial boundary equation, which have tried to solve by PINNs in Sect. 3. Exact solution (viscosity solution) is $u^*(x) = \frac{\sin x + \cos x + e^{-x}}{2}$. In this case, $F(x) = u''(x) + u(x) - e^{-x}$. Used a neural network consisting of 2 hidden layers with 16 neurons per layer & hyperbolic tangent activation functions to approximate true solution. To learn parameters, use Adam optimizer with learning rate 0.01. In this case,

    $$G = \{0, 0.01, 0.02, \ldots, 0.99, 1\},\ B = \{0, 1\}.$$
  - Example 3: Laplace equation in 2D. Let $\Omega = (-1, 1)^2$, problem:

    $$\begin{cases} u_{xx} + u_{yy} = 0 & \text{in } (-1,1)^2, \\ u(x,y) = xy & \text{on } \partial\Omega. \end{cases}$$

    Exact solution: $u^*(x, y) = xy$. Used a neural network v of 2 hidden layers with 8 neurons per layer & hyperbolic tangent activation functions to approximate true solution. To learn parameters, use Adam optimizer with learning rate 0.01.

    SKIPPED SIMULATION DETAILS

    After 8000 epochs, loss goes down to 0.088, MSE is between true solution & predicted values is $2.74 \cdot 10^{-4}$. Note: MSE between true solution & predicted values is much smaller than loss of neural network. This is reasonable sine using FD to estimate derivatives using a relatively coarse mesh grid with $N = 32$. Plot of true solution vs. neural network's approximated solution is shown in Fig. 3. True solution vs. neural network's approximated solution for Laplace equation.
  - Example 4: Eikonal equation in 2D. An Eikonal equation is a nonlinear PDE of 1st-order, which is commonly encountered in problems of wave propagation. Let $\Omega = (-1, 1)^2$, consider equation

    $$\begin{cases} |Du(x,y)| = 1 + \epsilon\Delta(x,y) & \text{in } (-1,1)^2, \\ u(x,y) = 1 - \sqrt{x^2+y^2} & \text{on } \partial\Omega. \end{cases}$$

    Here use $\epsilon = 0.0001$. Exact solution: $u^*(x, y) = 1 - \sqrt{x^2 + y^2}$.

    Used a neural network of 4 hidden layers with 64 neurons per layer & hyperbolic tangent activation functions to approximate true solution. To learn parameters, use Adam optimizer with learning rate 0.001. Mesh size used is $N = 32$.

    SKIPPED SIMULATION DETAILS

    After 5000 epochs, loss goes down to 0.01, MSE is between true solution & predicted value is $7.4 \cdot 10^{-5}$. Plot of true solution vs. neural network's approximated solution is as shown in Fig. 4: True solution vs. neural network's approximated solution for Eikonal equation.
- 6. Discussion & Conclusions. In this work, analyzed potential issues when applying PINNs for differential equations & introduced a novel technique, namely FinNet, for solving differential equations by incorporating FD into DL. Even though training phase in mesh-dependent, prediction phase is mesh-free. Illustrated effectiveness of our methods through experiments on solving various equations, which shows: approximation provided by FinNet is very close to true solution in terms of MSE & may work even when PINNs do not.

  For further work, various questions remain that are interesting to be addressed. Those can be questions on hyperparameters for FinNet, e.g. how to choose number of layers, activation function & mesh grid size. Furthermore, it would be interesting to compare FinNet with other approaches for nonlinear time-dependent PDEs or high-dimensional PDEs e.g. high-dimensional Hamilton–Jacobi–Bellman equation, or Burgers' equation.

4 Neural Network

4.1 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:

$$\text{find } u \in K : \langle Au - f, u - v\rangle_{V^*,V} \leq 0, \ \forall v \in K, \tag{299}$$

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 (299) is assumed to be Lipschitz & coercive, i.e., there exist constant $C_a, C_b > 0$ s.t. (3)

$$\|Au - Av\|_{V^*} \leq C_b\|u - v\|_V, \ \forall u, v \in K, \tag{300}$$

$$\langle Au - Av, u - v\rangle_{V^*,V} \geq C_a\|u - v\|_V^2, \ \forall u, v \in K, \tag{301}$$

& for simplicity, focus attention on linear differential operators of form

$$\langle Au, u - v\rangle_{V^*,V} = \int_\Omega \nabla u \cdot \nabla(u - v) + \sum_{i=1}^n b_i \partial_{x_i} u(u - v) + ku(u - v), \tag{302}$$

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 $A = -\Delta + \sum_{i=1}^n b_i\partial_{x_i} + k\text{Id}$ 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 (299) 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)

$$0 \leq (Au - f) \perp (u - \psi) \geq 0 \text{ a.e. on } \Omega, \tag{303}$$

$$u = h \text{ a.e. on } \partial\Omega, \tag{304}$$

where $a \perp b$ stands for $ab = 0$. This formulation $\Leftrightarrow$ (299) 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 (299) 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**[21]: 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 (299) 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 (299) $\Leftrightarrow$ 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, (299) $\Leftrightarrow$ minmax problem

$$\min_{u\in K}\max_{v\in K}\langle Au - f, u - v\rangle - \frac{1}{2\gamma}\|u - v\|_V^2 \tag{305}$$

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 (305), 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.

[21] có thêm lợi thế là không phụ thuộc hoàn toàn vào lưới.

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 $i = 1, \dots, n$.

- 2. Analysis of continuous problem. Some theoretical results concerning VI (299).
  - 2.1. Basic properties & saddle points. Address existence & uniqueness for (299). Since A is coercive & Lipschitz on $H^1(\Omega)$ & K is nonempty[22] 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 (299) 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 (299) 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!

4.2 Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser. Attention Is All You Need. 2017

[152738 citations]

- Abstract. Dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder & a decoder. Best performing models also connect encoder & decoder through an attention mechanism. Propose a new simple network architecture, Transformer, based solely on attention mechanisms, dispensing with recurrence & convolutions entirely. Experiments on 2 machine translation tasks show these models to be superior in quality while being more parallelizable & requiring significantly less time to train. Our model achieves 28.4 BLEU on WMT 2014 English-to-German translation task, improving over existing best results, including ensembles, by over 2 BLEU. On WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-art BLEU score of 41.0 after training for 3.5 days on 8 GPUs, a small fraction of training costs of best models from literature.
- Equal contribution. Listing order is random. Jakob proposed replacing RNNs with self-attention & started effort to evaluate this idea. Ashish, Illia, designed & implemented 1st Transformer models & has been crucially involved in every aspect of this work. Noam proposed scaled dot-product attention, multi-head attention & parameter-free position representation & became other person involved in nearly every detail. Niki designed, implemented, tuned, & evaluated countless model variants in our original codebase & tensor2tensor. Llion also experimented with novel model variants, was responsible for our initial codebase, & efficient inference & visualizations. ßLukasz & Aidan spent countless long days designing various parts of & implementing tensor2tensor, replacing our earlier codebase, greatly improving results & massively accelerating our research.
- 1. Introduction. RNNs, long short-term memory [12], & gated recurrent [7] neural networks in particular, have been firmly established as state of art approaches in sequence modeling & transduction problems e.g. language modeling & machine translation [29,2,5]. Numerous efforts have since continued to push boundaries of recurrent language models & encoder-decoder architectures [31,21,13].

  Recurrent models typically factor computation along symbol positions of input & output sequences. Aligning positions to steps in computation time, they generate a sequence of hidden states h_t, as a function of previous hidden state h_{t-1} & input for position t. This inherently sequential nature precludes (ngăn cản) parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples. Recent work has achieved significant improvements in computational efficiency through factorization tricks [18] & conditional computation [26], while also improving model performance in case of the latter. Fundamental constraint of sequential computation, however, remains.

  – Các mô hình truy hồi thường phân tích tính toán theo vị trí ký hiệu của chuỗi đầu vào & đầu ra. Căn chỉnh vị trí theo các bước trong thời gian tính toán, chúng tạo ra một chuỗi các trạng thái ẩn h_t, như một hàm của trạng thái ẩn trước đó h_{t-1} & đầu vào cho vị trí t. Bản chất tuần tự vốn có này ngăn cản (ngăn cản) việc song song hóa trong các ví dụ đào tạo, điều này trở nên quan trọng ở các chuỗi dài hơn, vì các ràng buộc về bộ nhớ giới hạn việc xử lý hàng loạt trên các ví dụ. Các công trình gần đây đã đạt được những cải tiến đáng kể về hiệu quả tính toán thông qua các thủ thuật phân tích [18] & tính toán

[22] 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.

có điều kiện [26], đồng thời cũng cải thiện hiệu suất mô hình trong trường hợp sau. Tuy nhiên, vẫn còn tồn tại ràng buộc cơ bản của tính toán tuần tự.

Attention mechanisms have become an integral part of compelling sequence modeling & transduction models in various tasks, allowing modeling of dependencies without regard to their distance in input or output sequences [2,16]. In all but a few cases [22], however, such attention mechanisms are used in conjunction with a recurrent network.

– Các cơ chế chú ý đã trở thành một phần không thể thiếu của mô hình hóa trình tự hấp dẫn & các mô hình chuyển đổi trong nhiều tác vụ khác nhau, cho phép mô hình hóa các mối phụ thuộc mà không cần quan tâm đến khoảng cách của chúng trong các trình tự đầu vào hoặc đầu ra [2,16]. Tuy nhiên, trong hầu hết các trường hợp ngoại trừ một số ít trường hợp [22], các cơ chế chú ý như vậy được sử dụng kết hợp với một mạng lưới tuần hoàn.

In this work propose Transformer, a model architecture eschewing recurrence & instead relying entirely on an attention mechanism to draw global dependencies between input & output. Transformer allows for significantly more parallelization & can reach a new state of art in translation quality after being trained for as little as 12 hours on 8 P100 GPUs.

– Trong tác phẩm này, đề xuất Transformer, một kiến trúc mô hình tránh lặp lại & thay vào đó hoàn toàn dựa vào cơ chế chú ý để rút ra các mối phụ thuộc toàn cục giữa đầu vào & đầu ra. Transformer cho phép song song hóa nhiều hơn đáng kể & có thể đạt đến trạng thái nghệ thuật mới về chất lượng dịch sau khi được đào tạo trong thời gian ít nhất là 12 giờ trên 8 GPU P100.

- 2. Background. Goal of reducing sequential computation also forms foundation of Extended Neural GPU [20], ByteNet [15] &7 ConvS2S [8], all of which use CNNs as basic building block, computing hidden representations in parallel for all input & output positions. In these models, number of operations required to relate signals from 2 arbitrary input or output positions grows in distance between positions, linearly for ConvS2S & logarithmically for ByteNet. This makes it more difficult to learn dependencies between distant positions [11]. In Transformer this is reduced to a constant number of operations, albeit at cost of reduced effective resolution due to averaging attention-weighted positions, an effect we counteract with Multi-Head Attention as described in Sect. 3.2.

  Self-attention, sometimes called intra-attention is an attention mechanism relating different positions of a single sequence in order to compute a representation of sequence. Self-attention has been used successfully in a variety of tasks including reading comprehension, abstractive summarization, textual entailment & learning task-independent sentence representations [4, 22, 23, 19].

  End-to-end memory networks are based on a recurrent attention mechanism instead of sequence-aligned recurrence & have been shown to perform well on simple-language question answering & language modeling tasks [28].

  To best of our knowledge, however, Transformer is 1st transduction model relying entirely on self-attention to compute representations of its input & output without using sequence-aligned RNNs or convolution. In following sects, describe Transformer, motivate self-attention & discuss its advantages over models e.g. [14,15,8].

- 3. Model Architecture. Most competitive neural sequence transduction models have an encoder-decoder structure [5, 2, 29]. Here, encoder maps an input sequence of symbol representations $(x_1, \ldots, x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, \ldots, z_n)$. Given $\mathbf{z}$, decoder then generates an output sequence $(y_1, \ldots, y_m)$ of symbols 1 element at a time. At each step model is auto-regressive [9], consuming previously generated symbols as additional input when generating text.

  Transformer follows this overall architecture using stacked self-attention & point-wise, fully connected layers for both encoder & decoder, shown in Fig. 1: Transformer - model architecture.

  - 3.1. Encoder & Decoder Stacks. Encoder is composed of a stack of $N = 6$ identical layers. Each layer hs 2 sub-layers. 1st is a multi-head self-attention mechanism, & 2nd is a simple, position-wise fully connected feed-forward network. Employ a residual connection [10] around each of 2 sub-layers, followed by layer normalization [1]. I.e., output of each sub-layer is $\text{LayerNorm}(x + \text{Sublayer}(x))$, where $\text{Sublayer}(x)$: function implemented by sub-layer itself. To facilitate these residual connections, all sub-layers in model, as well as embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

    Decoder. Decoder is also composed of a stack of $N = 6$ identical layers. In addition to 2 sub-layers in each encoder layer, decoder inserts a 3rd sub-layer, which performs multi-head attention over output of encoder stack. Similar to encoder, employ residual connections around each of sub-layers, followed by layer normalization. Also modify self-attention sub-layer in decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact: output embeddings are offset by 1 position, ensures: predictions for position i can depend only on known outputs at positions $< i$.

  - 3.2. Attention. An attention function can be described as mapping a query & a set of key-value pairs to an output, where query, keys, values, & output are all vectors. Output is computed as a weighted sum of values, where weight assigned to each value is computed by a compatibility function of query with corresponding key.

    * 3.2.1. Scaled Dot-Product Attention. Call our particular attention "Scaled Dot-Product Attention" (Fig. 2: Left: Scaled Dot-Product Attention. Right: Multi-Head Attention consists of several attention layers running in parallel.). Input consists of queries & keys of dimension d_k, & values of dimension d_v. Compute dot products of query with all keys, divide each by $\sqrt{d_k}$, & apply a softmax function to obtain weights on values.
      In practice, compute attention function on a set of queries simultaneously, packed together into a matrix Q. Keys & values are also packed together intro matrices K, V. Compute matrix of output as:

$$\text{Attention}(Q, K, V) = \text{softmax}$$

    - 3.2.2. Multi-Head Attention.
    - 3.2.3. Applications of Attention in our Model.
  - 3.3. Position-wise Feed-Forward Networks.
  - 3.4. Embeddings & Softmax.
- 4. Why Self-Attention.
- 5. Training.
- 6. Results.
- 7. Conclusion.

5 Recurrent Neural Network

5.1 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 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.[23] 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

[23] 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 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 $\hat{\mathbf{y}}^{(t)}$.

    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):

$$v_j = l_j\left(\sum_{j'} w_{jj'} \cdot v_{j'}\right). \tag{306}$$

    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

$$\hat{y}_k = \frac{e^{a_k}}{\sum_{k'=1}^{K} e^{a_{k'}}}, \ \forall k = 1, \ldots, K. \tag{307}$$

    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

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} F_i, \tag{308}$$

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

$$\delta_k = \frac{\partial \mathcal{L}(\hat{y}_k, y_k)}{\partial \hat{y}_k} \cdot l'_k(a_k). \tag{309}$$

Given these values δ_k, for each node in immediately prior layer, calculate

$$\delta_j = l'(a_j) \sum_k \delta_k \cdot w_{kj}. \tag{310}$$

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

$$\partial_{w_{jj'}} \mathcal{L} = \delta_j v'_j. \tag{311}$$

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 $t+1$.:

$$\mathbf{h}^{(t)} = \sigma(W_{\text{hx}}\mathbf{x}^{(t)} + W_{\text{hh}}\mathbf{h}^{(t-1)} + \mathbf{b}_h), \tag{312}$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(W_{\text{yh}}\mathbf{h}^{(t)} + \mathbf{b}_y). \tag{313}$$

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.[24] 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

[24] [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".

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 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:

$$\mathbf{s}^{(t)} = \mathbf{g}^{(t)} \odot \mathbf{i}^{(t)} + \mathbf{f}^{(t)} \odot \mathbf{s}^{(t-1)}. \tag{314}$$

* *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: ***

5.2 [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'[25]. 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 $w_1, \dots, w_N$, & N inputs $x_1, \dots, x_N$, & 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 $\mathbf{x} = [x_1, \dots, x_N, 1]^\top$, whereas $\mathbf{w} = [w_1, \dots, w_N, w_0]^\top$ 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 : $\mathbb{R}^N \to \mathbb{R}^M$.

[25] **perceptron** [n] (*computing*) an artificial network which is intended to copy the brain's ability to recognize things & see the differences between things.

- 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.

  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.[26] 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 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

[26] 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.

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:

$$\hat{\mathbf{w}}(k+1) = \hat{\mathbf{w}}(k) + \eta\Gamma(k)(y(k) - \mathbf{x}^\top(k)\hat{\mathbf{w}}(k)), \tag{315}$$

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*:

$$\breve{\mathbf{w}}(k+1) = (I - \eta\Sigma(k))\breve{\mathbf{w}}(k) - \eta\Gamma(k)\nu(k) + \mathbf{n}(k+1). \tag{316}$$

For different gains $\Gamma(k)$, 3 well-known algorithms can be obtained from (315). 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.

1. Least mean square (LMS) algorithm: $\Gamma(k) = \mathbf{x}(k)$.
2. Recursive least-squares (RLS) algorithm:

$$\Gamma(k) = P(k)\mathbf{x}(k), \tag{317}$$

$$P(k) = \frac{1}{1-\eta}\left[P(k-1) - \eta\frac{P(k-1)\mathbf{x}(k)\mathbf{x}^\top(k)P(k-1)}{1-\eta+\eta\mathbf{x}^\top(k)P(k-1)\mathbf{x}(k)}\right]. \tag{318}$$

3. Kalman filter (KF) algorithm (Guo & Ljung 1995; Kay 1993):

$$\Gamma(k) = \frac{P(k-1)\mathbf{x}(k)}{R + \eta\mathbf{x}^\top(k)P(k-1)\mathbf{x}(k)}, \tag{319}$$

$$P(k) = P(k-1) - \frac{\eta P(k-1)\mathbf{x}(k)\mathbf{x}^\top(k)P(k-1)}{R + \eta\mathbf{x}^\top(k)P(k-1)\mathbf{x}(k)} + \eta Q. \tag{320}$$

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

$$\Delta E(\mathbf{w}) = E(\mathbf{w} + \Delta\mathbf{w}) - E(\mathbf{w}) \approx (\nabla_{\mathbf{w}} E(\mathbf{w}))^\top \Delta\mathbf{w} + \frac{1}{2}\Delta\mathbf{w}^\top \mathbf{H}\Delta\mathbf{w}. \tag{321}$$

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: $\boxed{\text{adaptive algorithms mainly differ in their form of adaptation gains.}}$ 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,

$$J(\mathbf{w}(k)) = [d(k) - F(\mathbf{x}(k), \mathbf{w}(k))]^2, \tag{322}$$

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.,

$$J_{\text{Wiener}}(\mathbf{w}(k)) = E[[d(k) - \mathbf{x}^\top(k)\mathbf{w}(k)]^2], \tag{323}$$

& 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

$$\partial_{\mathbf{w}} J_{\text{Wiener}} = -2E[e(k)\mathbf{x}(k)], \tag{324}$$

where $e(k) = d(k) - \mathbf{x}^\top(k)\mathbf{w}(k)$.
At Wiener solution, this gradient equals null vector $\mathbf{0}$. Solving (324) 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

$$\partial_{\mathbf{w}(k)} e(k) = -\mathbf{x}(k), \ \partial_{\mathbf{w}(k)} J(k) = -e(k)\mathbf{x}(k). \tag{325}$$

Set of equations that describes LMS algorithm is given by

$$\begin{cases} y(k) = \sum_{i=1}^{N} x_i(k) w_i(k) = \mathbf{x}^\top(k)\mathbf{w}(k), \\ e(k) = d(k) - y(k), \\ \mathbf{w}(k+1) = \mathbf{w}(k) + \eta e(k)\mathbf{x}(k). \end{cases} \tag{326}$$

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

$$e(k) = d(k) - \Phi(\mathbf{x}^\top(k)\mathbf{w}(k)), \tag{327}$$

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \eta\nabla_{\mathbf{w}}J(k), \tag{328}$$

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

$$\partial_{\mathbf{w}(k)}J(k) = e(k)\partial_{\mathbf{w}(k)}e(k) = -e(k)\Phi'(\mathbf{x}^\top(k)\mathbf{w}(k))\mathbf{x}(k), \tag{329}$$

where $\Phi'(\cdot)$ represents 1st derivative of nonlinearity $\Phi(\cdot)$ & weight update equation (328) can be rewritten as

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \eta\Phi'(\mathbf{x}^\top(k)\mathbf{w}(k))e(k)\mathbf{x}(k). \tag{330}$$

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:

$$J(k) = \frac{1}{N}\sum_{i=1}^{N}(e^2(k-i+1) + G(\|\mathbf{w}(k-i+1)\|_2^2)), \tag{331}$$

    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.
    1. Initialize weights
    2. 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.
    1. Initialize weights
    2. 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

$$\mathbf{y}(k) = \boldsymbol{\sigma}(\mathbf{A}\mathbf{x}(k)), \tag{332}$$

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 (332) 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 (332). 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.

1. Normalization, which in this context means dividing each element of input vector $\mathbf{x}(k)$ by its squared norm, i.e. $x_i(k) \in \mathbf{x}(k) \to \frac{x_i(k)}{\|\mathbf{x}(k)\|_2^2}$.
2. Rescaling, which means transforming input data in manner that we multiply/divide them by a constant & also add/subtract a constant from data.[27]
3. 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

$$\text{mean} = \frac{\sum_i X_i}{N}, \ \text{std} = \sqrt{\frac{\sum_i (X_i - \text{mean}^2)}{N-1}}. \tag{333}$$

   Standardized quantity becomes $S_i = \frac{X_i - \text{mean}}{\text{std}}$.
   · Standardize X to midrange 0 & range 2. This can be achieved by

$$\text{midrange} = \frac{1}{2}(\max_i X_i + \min_i X_i), \ \text{range} = \max_i X_i - \min_i X_i, \ S_i = \frac{X_i - \text{midrange}}{\text{range}/2}. \tag{334}$$

[27] In real life a typical rescaling is transforming temperature from Celcius into Fahrenheit scale.

4. 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.
- 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[28] $O(\eta^2)$.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[29] (Qin et al. 1992). However, difference could also be controlled by learning rate η. Difference will converge to 0 as quickly as $\eta \to 0$[30] (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

$$H(z) = \frac{\sum_{k=0}^{M} b(k) z^{-k}}{1 + \sum_{k=1}^{N} a(k) z^{-k}}. \tag{335}$$

Can rearrange this function either in a cascaded manner as

$$H(z) = A \prod_{k=1}^{\max\{M,N\}} \frac{1 - \beta_k z^{-1}}{1 - \alpha_k z^{-1}}, \tag{336}$$

or in a parallel manner as

$$H(z) = \sum_{k=1}^{N} \frac{A_k}{1 - \alpha_k z^{-1}}, \tag{337}$$

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

$$F(W, X) = \Phi\left(\sum_n w_n \Phi\left(\sum_i v_i \Phi\left(\cdots \Phi\left(\sum_j u_j X_j\right)\cdots\right)\right)\right), \tag{338}$$

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

$$F(W, X) = \Phi\left(\sum_n w_n \Phi\left(\sum_j u_j X_j\right)\right) \tag{339}$$

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.

[28] 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.

[29] (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.

[30] In which case, have a very slow learning process.

- 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[31] 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.

  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.[32]

- 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

$$e(k) = y(k) - \hat{y}(k) = y(k) - \sum_{i=1}^{p} a_i y(k-i). \tag{340}$$

[31] 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.

[32] ARMA models also have a canonical (up to an invariant) representation.

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,[33] with zero mean (Papoulis 1984). A fundamental advantage of mean square error criterion is so-called *orthogonality condition*, which implies

$$E[e(k), y(k-j)] = 0,\ j = 1, 2, \ldots, p, \tag{341}$$

is satisfied only when $a_i, i = 1, \ldots, p$, take on their optimal values. As a consequence of of (341) & 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 $r_{yy}(0) - \sum_{i=1}^{p} a_i r_{yy}(i)$.

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

$$\hat{r}_{yy}(\tau) = \frac{1}{N}\sum_{k=0}^{N-\tau-1} y(k)y(k+\tau),\ \tau = 0, 1, \ldots, p. \tag{342}$$

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[34] (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:

$$\Phi(v(k)) = \begin{cases} 0 & v(k) < 0, \\ 1 & v(k) \geq 0, \end{cases} \tag{343}$$

  * Piecewise-linear:

$$\Phi(v(k)) = \begin{cases} 0 & v(k) \leq -\frac{1}{2}, \\ v(k) & -\frac{1}{2} < v(k) < \frac{1}{2}, \\ 1 & v(k) \geq \frac{1}{2}, \end{cases} \tag{344}$$

  * Logistic:

$$\Phi(v(k)) = \frac{1}{1+e^{-\beta v(k)}},\ \beta \geq 0. \tag{345}$$

  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

[33] Wide sense stationarity implies that mean is constant, autocorrelation function is only a function of time lag & variance is finite.
[34] z^{-1} operator is a delay operator s.t. $\mathcal{Z}(y(k-1)) = z^{-1}\mathcal{Z}(y(k))$.

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[35] 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 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

$$y(k) = \sum_{i=1}^{p} a_i y(k-i) + \sum_{j=0}^{q} b_j e(k-j), \tag{346}$$

where $y(k)$: output, $e(k)$: input,[36] $a_i, i = 1, \ldots, p$, are (AR) feedback coefficients & $b_j, j = 0, 1, \ldots, q$, are (MA) feedforward coefficients. In causal systems, (346) 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 (346) 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 (346) is simplified by removing feedback terms to yield

$$y(k) = \sum_{j=0}^{q} b_j e(k-j). \tag{347}$$

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, (347) is simplified to yield an autoregressive AR(p) filter

$$y(k) = \sum_{i=1}^{p} a_i y(k-i) + e(k), \tag{348}$$

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).[37] Most straightforward way to test stability is to exploit $\mathcal{Z}$-domain representation of transfer function of filter represented by (3.10):

$$H(z) = \frac{Y(z)}{E(z)} = \frac{b_0 + b_1 z^{-1} + \cdots + b_q z^{-q}}{1 - a_1 z^{-1} - \cdots - a_p z^{-p}} = \frac{N(z)}{D(z)}. \tag{349}$$

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

[35] Term 'artificial neuron' will be replaced by 'neuron' in sequel.
[36] Notice $e(k)$ is used as filter input, rather than $x(k)$, for consistency with later sections on prediction error filtering.
[37] This type of stability is commonly denoted as BIBO stability in contrast to other types of stability, e.g. global asymptotic stability (GAS).

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 $\sigma_{\rm e}^2$, represent resp., ARMA(p,q), MA(q), AR(p) signal models. Autocorrelation function of input $e(k)$ is given by $\sigma_{\rm e}^2\delta(k) \Rightarrow$ its power spectral density (PSD) is $P_{\rm e}(f) = \sigma_{\rm e}^2$ for all f. PSD of an ARMA model is therefore:

$$P_y(f) = |H(f)|^2 P_{\rm e}(f) = \sigma_{\rm e}^2 |H(f)|^2, \ f \in \left(-\frac{1}{2}, \frac{1}{2}\right], \tag{350}$$

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).

- 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 $\{y(k)\}$

$$\hat{y}(k) = E[y(k)|y(k-1), y(k-2), \ldots, y(0)] \tag{351}$$

is given by

$$\hat{y}(k) = \sum_{i=1}^{p} a_i y(k-i) + \sum_{j=1}^{q} b_j \hat{e}(k-j), \tag{352}$$

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,

$$y(k) = \Theta\left(\sum_{i=1}^{p} a_i y(k-i) + \sum_{j=1}^{q} b_j e(k-j)\right) + e(k), \tag{353}$$

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

$$\hat{y}(k) = \Theta\left(\sum_{i=1}^{p} a_i y(k-i) + \sum_{j=1}^{q} b_j \hat{e}(k-j)\right), \tag{354}$$

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

$$y(k) = \Theta\left(\sum_{i=1}^{p} a_i y(k-i)\right) + e(k) \tag{355}$$

& its associated predictor is

$$\hat{y}(k) = \Theta\left(\sum_{i=1}^{p} a_i y(k-i)\right). \tag{356}$$

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

$$y(k) = \Theta\left(\sum_{i=1}^{p} a_i y(k-i) + \sum_{j=1}^{q} b_j e(k-j) + \sum_{s=1}^{r} c_s u(k-s)\right) + e(k), \tag{357}$$

& is termed a nonlinear autoregressive moving average with exogeneous inputs model, NARMAX(p,q,r), with associated predictor

$$\hat{y}(k) = \Theta\left(\sum_{i=1}^{p} a_i y(k-i) + \sum_{j=1}^{q} b_j e(k-j) + \sum_{s=1}^{r} c_s u(k-s)\right), \tag{358}$$

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

$$\hat{y}(k) = \Phi(y(k-1), y(k-2), \ldots, y(k-p)), \tag{359}$$

  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

$$\tilde{y}_{\mu,j}(k) = \mu \tilde{y}_{\mu,j}(k-1) + (1-\mu)\tilde{y}_{\mu,j-1}(k-1), \tag{360}$$

  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

$$\hat{y}(k) = \Phi(y(k-1), y(k-2), \ldots, y(k-p), \hat{e}(k-1), \ldots, \hat{e}(k-q)), \tag{361}$$

  where again $\Phi(\cdot)$ represents nonlinear mapping of neural network &

$$\hat{e}(k-j) = y(k-j) - \hat{y}(k-j), \;\; j = 1, \ldots, q. \tag{362}$$

  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

  $$\mathbf{s}(k) = \varphi(\mathbf{s}(k-1), \mathbf{y}(k-1), \hat{y}(k-1)), \tag{363}$$
  $$\hat{y}(k) = \psi(\mathbf{s}(k-1), \mathbf{y}(k-1), \hat{y}(k-1)), \tag{364}$$

  where φ, Ψ represent general classes of nonlinearities. Particular choice of N minimal state variables is not unique, therefore several canonical forms[38] 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.

[38] 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.

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,[39] 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 & compositions of single-variable functions?[40] 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

$$f(x_1, \ldots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \psi_{pq}(x_p) \right), \tag{365}$$

  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,[41] then finite sums of form

$$g(\mathbf{x}) = \sum_{i=1}^{N} w_i \sigma(\mathbf{a}_i^\top \mathbf{x} + b_i), \tag{367}$$

  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

[39] 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.'

[40] 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).

[41] $\sigma(\cdot)$ 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

$$\sigma(t) = \begin{cases} 0 & t \to -\infty, \\ 1 & t \to +\infty. \end{cases} \tag{366}$$

This justifies use of logistic function $\sigma(x) = \frac{1}{(1+e^{-\beta x})}$ in neural network applications.

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).

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.[42] 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[43] functions having following desirable properties[44]

  * $\sigma_i \in S$ 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 $x_1, x_2 \in \mathbb{R} \Rightarrow \sigma_i(x_1) \leq \sigma_i(x_2)$

[42] 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.
[43] Sigmoid means S-shaped.
[44] Constraints we impose on sigmoidal functions are stricter than ones commonly employed.

* 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 $L < 1$.

Surveys of neural transfer functions can be found in Duch & Jankowski (1999) & Cichocki & Unbehauen (1993). Examples of sigmoidal functions are

$$\sigma_1(x) = \frac{1}{1 + e^{-\beta x}}, \ \beta \in \mathbb{R}, \tag{368}$$

$$\sigma_2(x) = \tanh(\beta x) = \frac{e^{\beta x} - e^{-\beta x}}{e^{\beta x} + e^{-\beta x}}, \ \beta \in \mathbb{R}, \tag{369}$$

$$\sigma_3(x) = \frac{2}{\pi} \arctan(\frac{\pi}{2}\beta x), \ \beta \in \mathbb{R}, \tag{370}$$

$$\sigma_4(x) = \frac{x^2}{1 + x^2}\text{sgn}x, \tag{371}$$

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*,[45] 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

$$\sigma_1'(x) = \beta\sigma_1(x)(1 - \sigma_1(x)), \tag{372}$$

$$\sigma_2'(x) = \beta\text{sech}^2 x = \beta(1 - \sigma_2^2(x)). \tag{373}$$

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 $R_{\min}, R_{\max}$.[46] 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.[47] 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.

[45] 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.

[46] To normalize input data to $\mu = 0, \text{std} = 1$, calculate $\mu = \frac{\sum_{i=1}^N x_i}{N}, \text{std} = \sqrt{\frac{\sum_{i=1}^N (x_i - \mu)^2}{N}}$, & perform standardization of input data as $\tilde{x}_i = \frac{x_i - \mu}{\text{std}}$. To translate data to midrange 0 & standardize to range R, perform

$$Z = \frac{\max_i\{x_i\} + \min_i\{x_i\}}{R}, \ S_x = \max_i\{x_i\} - \min_i\{x_i\}, \ x_i^n = \frac{x_i - Z}{S_x/R}. \tag{374}$$

[47] Rational transfer functions (Leung & Haykin 1993) & Gaussian transfer functions also allow NNs to implement universal approximators.

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).

$$A(\mathbf{x},\mathbf{w}) = w_0 + \sum_{i=1}^{N} w_i x_i + w_{N+1}\sum_{i=1}^{N} x_i^2. \tag{375}$$

- Other Activation Functions. By universal approximation theorems, there are many choices of nonlinear activation function $\Rightarrow$ 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

$$H(x) = \begin{cases} 0 & x \le \theta, \\ 1 & x > \theta, \end{cases} \tag{376}$$

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 function*. Semilinear function is defined as

$$H_{\mathrm{SL}}(x) = \begin{cases} 0 & x \le \theta_1, \\ \dfrac{x-\theta_1}{\theta_2-\theta_1} & \theta_1 < x \le \theta_2, \\ 1 & x > \theta_2, \end{cases} \tag{377}$$

Functions (376) & (377) 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

$$\sigma(x) = \frac{1}{1+e^{-\beta x}} \tag{378}$$

degenerates into step function (376) 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)

$$f(x) = \frac{1}{1+e^{-(x^2+\mathrm{bias})}}. \tag{379}$$

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

$$f(x) = \lambda\sigma(x) + (1-\lambda)H(x), \tag{380}$$

where $\sigma(x)$ is a sigmoid function, $H(x)$ is a hard-limiting function & $0 \le \lambda \le 1$, 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

$$f(x) = \sum_{n=1}^{N} h_n\sigma_n = \sum_{n=1}^{N} \frac{h_n}{1+e^{-\beta_n x+b_n}}. \tag{381}$$

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[48] (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)

$$\varphi(x) = \begin{cases} |x| & |x| \le 1, \\ 1 & |x| > 1. \end{cases} \tag{382}$$

[48] Splines are piecewise polynomials (often cubic) that are smooth & can retain 'squashing property'.

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 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

$$f(x) = \begin{cases} 0 & x \le -1, \\ 0.5 + x\left(1 - \dfrac{|x|}{2}\right) & -1 < x < 1, \\ 1 & x \ge 1. \end{cases} \tag{383}$$

  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)

$$\sigma(x) = \frac{x}{1+|x|},\ \sigma'(x) = \frac{1}{(1+|x|)^2} = (1-|\sigma|)^2, \tag{384}$$

  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

$$f(\mathbf{x}) = \sum_{i=1}^{N} c_i G(\mathbf{x}; \mathbf{t}_i), \tag{385}$$

  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.[49] 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.

[49] 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}.

If again consider approximation, (4.17)

$$f(x) = \sum_{i=1}^{N} c_i \sigma(x - a_i), \tag{386}$$

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.[50]

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.[51] 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)

$$f(z) = \frac{1}{1 + e^{-z}} : \mathbb{C}^N \to \mathbb{C}, \tag{387}$$

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)

$$\sigma(z) = \sigma(x) + \mathrm{i}\sigma(y), \tag{388}$$

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)

$$\sigma(z) = \frac{2c_1}{1 + e^{-2c_2 z}} - c_1, \tag{389}$$

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)

$$\sigma(z) = \frac{(\cos\theta + \mathrm{i}\sin\theta)(z - s)}{1 - \alpha^\star z}, \tag{390}$$

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 $z = \frac{1}{\alpha}$,

[50] 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.

[51] 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.

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)

$$f(z) = \frac{z}{c + \frac{|z|}{r}}, \tag{391}$$

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.
  - 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[52] & 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[53] 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[54] 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

$$y(k) = h_0 + \sum_{i=0}^{N} h_1(i)x(k-i) + \sum_{i=0}^{N}\sum_{j=0}^{N} h_2(i,j)x(k-i)x(k-j) + \cdots \tag{392}$$

[52] System identification, e.g., consists of choice of model, model parameter estimation & model validation.

[53] Technically, notions of *system* & *process* are equivalent (Pearson 1995; Sjöberg et al. 1995).

[54] 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.

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 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* ⇒ 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 ⇒ 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 $y = F(u_{\text{PAST}}, u)$. 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)

$$\Phi(v(k)) = \frac{1}{1+e^{-\beta v(k)}} = \frac{1}{2} + \frac{\beta}{4}v(k) - \frac{\beta^3}{48}v^3(k) + \frac{\beta^5}{480}v^5(k) - \frac{17\beta^7}{80640}v^7(k) + \cdots \tag{393}$$

  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

$$y(k) = \Phi(w_1 x(k-1) + w_2 y(k-1) + w_0). \tag{394}$$

  Expanding this using (5.7):

$$y(k) = \frac{1}{2} + \frac{1}{4}[w_1 x(k-1) + w_2 y(k-1) + w_0] - \frac{1}{48}[w_1 x(k-1) + w_2 y(k-1) + w_0]^3 + \cdots, \tag{395}$$

  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)

$$y(k) = c_0 + c_1 x(k-1) + c_2 y(k-1) + c_3 x^2(k-1) + c_4 y^2(k-1) + c_5 x(k-1)y(k-1) + c_6 x^3(k-1) + c_7 y^3(k-1) + c_8 x^2(k-1)y(k-1) + \cdots. \tag{396}$$

  (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)

$$z(k) = 0.5z(k-1) + 0.1x(k-1). \tag{397}$$

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[55] (5.11) as

$$y(k) = 0.5y(k-1) + 0.1x(k-1) + e(k) - 0.5e(k-1). \tag{398}$$

An equivalent FIR (MA) representation of same filter (5.11), using method of long division, gives

$$z(k) = 0.1x(k-1) + 0.05x(k-2) + 0.025x(k-3) + 0.0125x(k-4) + \cdots \tag{399}$$

& representation of a noisy system now becomes (5.15)

$$y(k) = 0.1x(k-1) + 0.05x(k-2) + 0.025x(k-3) + 0.0125x(k-4) + \cdots + e(k). \tag{400}$$

Clearly, noise in (5.15) is not correlated with previous outputs & estimates are unbiased.[56] 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,[57] 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
    1. Wiener system
    $$y(k) = g(H(z^{-1})u(k)), \tag{401}$$
    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
    2. Hammerstein system
    $$y(k) = H(z^{-1})g(u(k)) \tag{402}$$
    3. Uryson system, defined by
    $$y(k) = \sum_{i=1}^{M} H_i(z^{-1})g_i(u(k)). \tag{403}$$

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)

$$y(k) = \Phi_N(H_N(z^{-1})\Phi_{N-1}(H_{N-1}(z^{-1})\cdots\Phi_1(H_1(z^{-1})u(k)))), \tag{404}$$
$$y(k) = H_N(z^{-1})\Phi_N(H_{N-1}(z^{-1})\Phi_{N-1}\cdots\Phi_1(H_1(z^{-1}u(k)))) \tag{405}$$

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[58] 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

$$y(k) = \Phi[u(k-1)] + \sum_{i=2}^{\infty} h_i \Phi[u(k-i)] + \nu(k), \tag{406}$$

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.

[55] 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.
[56] Under usual assumption that external additive noise $e(k)$ is not correlated with input signal $x(k)$.
[57] A finite degree polynomial steady-state characteristic.
[58] By Weierstrass theorem, polynomials can approximate arbitrarily well any nonlinear function, including sigmoid functions.

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

$$y(k) = \Phi(v(k)), \tag{407}$$

$$v(k) = \sum_{i=0}^{\deg N(z)} n_i(k)x(k-i) + 1 + \sum_{j=1}^{\deg D(z)} d_j(k)v(k-j), \tag{408}$$

$$x(k) = \sum_{l=1}^{p} w_l(k)u_l(k), \tag{409}$$

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 27. *Wiener model*

$$w(k) = 0.8w(k-1) + 0.4u(k-1), \tag{410}$$

$$y(k) = w(k) + w^3(k) + e(k), \tag{411}$$

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

$$v(k) = \sum_{i=0}^{M} w_{u,i}(k)u(k-i) + \sum_{j=1}^{N} w_{v,j}(k)v(k-j), \tag{412}$$

$$y(k) = \Phi(v(k)), \tag{413}$$

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

$$v(k) = \sum_{i=0}^{M} w_{u,i}(k)u(k-i) + \sum_{j=1}^{N} w_{y,j}(k)y(k-j), \tag{414}$$

$$y(k) = \Phi(v(k)), \tag{415}$$

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,

$$y_i(k) = \Phi(v_i(k)),\ i = 1, \ldots, N, \tag{416}$$

$$v_i(k) = \sum_{l=1}^{p+N+1} w_{i,l}(k) u_l(k), \tag{417}$$

$$\mathbf{u}_i^\top = [s(k-1), \ldots, s(k-p), 1, y_1(k-1), y_2(k-1), \ldots, y_N(k-1)], \tag{418}$$

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:

$$y_{i,n}(k) = \Phi(v_{i,n}(k)), \tag{419}$$

$$v_{i,n}(k) = \sum_{l=1}^{p+N+1} w_{n,l}(k) u_{i,l}(k), \tag{420}$$

$$\mathbf{u}_i^\top(k) = [s(k-i), \ldots, s(k-i-p+1), 1, y_{i+1,1}(k), y_{i,2}(k-1), \ldots, y_{i,N}(k-1)] \text{ for } 1 \le i \le M-1, \tag{421}$$

$$\mathbf{u}_M^\top(k) = [s(k-M), \ldots, s(k-M-p+1), 1, y_{M,1}(k-1), y_{M,2}(k-1), \ldots, y_{M,N}(k-1)] \text{ for } i = M. \tag{422}$$

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)

$$e_i(k) = s(k-i+1) - y_{i,1}(k),\ i = 1, \ldots, M. \tag{423}$$

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)

$$\hat{x}(k) = \Phi\left(\sum_{i=1}^{p} w_{1,i}(k) x(k-i) + w_{1,p+1}(k) + \sum_{j=p+2}^{p+q+1} w_{1,j}(k) \hat{e}(k+j-2-p-q) + \sum_{l=p+q+2}^{p+q+N} w_{1,l}(k) y_{l-p-q}(k-1)\right). \tag{424}$$

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

$$x(k-1),\ldots,x(k-p),+1,\hat{e}(k-1),\ldots,\hat{e}(k-q),y_2(k-1),\ldots,y_N(k-1). \tag{425}$$

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 $\{\hat{e}\}$ ⇒ 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

$$\hat{x}(k)=\Phi\left(\sum_{i=1}^{p}w_{1,i}(k)x(k-i)+w_{1,p+1}(k)+\sum_{j=p+2}^{p+q+1}w_{1,j}(k)\hat{x}(k+j-2-p-q)+\sum_{l=p+q+2}^{p+q+N}w_{1,l}(k)y_{l-p-q}(k-1)\right) \tag{426}$$

or, bearing in mind (5.39), notation used earlier (Haykin & Li 1995; Mandic et al. 1998) for examples on prediction of speech, i.e., $x(k)=s(k)$, & $y_1(k)=\hat{s}(k)$,

$$\hat{s}(k)=\Phi\left(\sum_{i=1}^{p}w_{1,i}(k)s(k-i)+w_{1,p+1}(k)+\sum_{j=p+2}^{p+q+1}w_{1,j}(k)y_1(k+j-2-p-q)+\sum_{l=p+q+2}^{p+q+N}w_{1,l}(k)y_{l-p-q}(k-1)\right), \tag{427}$$

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[59] 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.

[59] 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 $h_{i,j,\ldots,z}\neq 0\Leftrightarrow i=j=\cdots=z$.

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.[60] 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 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

$$y(k) = c_0 + c_1 u(k-1) + c_2 u(k-2) + c_3 u^2(k-1) + c_4 u^2(k-2) + c_5 u(k-1)u(k-2) + c_6 u^3(k-1) + c_7 u^3(k-2) + \cdots , \quad (428)$$

which has form of a general Volterra series, given by (6.2)

$$y(k) = h_0 + \sum_{i=0}^{N} h_1(i)x(k-i) + \sum_{i=0}^{N}\sum_{j=0}^{N} h_2(i,j)x(k-i)x(k-j) + \cdots . \quad (429)$$

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

$$\boxed{y(k) = \sum_{j=1}^{N-1} c_j y(k-j) + \sum_{i=0}^{N-1}\sum_{j=1}^{N-1} b_{i,j} y(k-j)x(k-i) + \sum_{i=0}^{N-1} a_i x(k-i).} \quad (430)$$

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.*

[60] E.g. nonsmooth functions e.g. $|x|$.

Solution. Functional description of recurrent network depicted in Fig. 6.1 is given by

$$y(k) = c_1 y(k-1) + b_{0,1} x(k) y(k-1) + b_{1,1} x(k-1) y(k-1) + a_0 x(k) + a_1 x(k-1), \tag{431}$$

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

$$\text{net}_i(k) = \sum_{j=0}^{M} w_{i,j} x(k-j), \tag{432}$$

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[61] as

$$\Phi(\text{net}_i(k)) = \sum_{l=0}^{L} \xi_{il} \text{net}_i^l(k), \tag{433}$$

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[62] (6.8)

$$\hat{y} = F(y(k-1), \ldots, y(k-N), u(k-1), \ldots, u(k-M)), \tag{434}$$

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.

[61] Using Weierstrass theorem, this expansion can be arbitrarily accurate. However, in practice resort to a moderate order of this polynomial expansion.

[62] This model is referred to as NARMAX model (nonlinear ARMAX), since it resembles linear model $\hat{y}(k) = a_0 + \sum_{j=1}^{N} a_j y(k-j) + \sum_{i=1}^{M} b_i u(k-i)$.

5.3 John Miller, Moritz Hardt. Stable Recurrent Models. 2019

[209 citations]

- Abstract. Stability is a fundamental property of dynamical systems, yet to this date it has had little bearing on practice of RNNs. In this work, conduct a thorough investigation of stable recurrent models. Theoretically, prove stable RNNs are well approximated by feed-forward networks for purpose of both inference & training by gradient descent. Empirically (theo kinh nghiệm), demonstrate stable recurrent models often perform as well as their unstable counterparts on benchmark sequence tasks. Taken together, these findings shed light on effective power of recurrent networks & suggest much of sequence learning happens, or can be made to happen, in stable regime. Moreover, our results help to explain why in many cases practitioners succeed in replacing recurrent models by feed-forward models.

- 1. Introduction. RNNs are a popular modeling choice for solving sequence learning problems arising in domains e.g. speech recognition & natural language processing. At outset, RNNs are nonlinear dynamical systems commonly trained to fit sequence data via some variant of gradient descent.

  Stability is of fundamental importance in study of dynamical system. Surprisingly, however, stability has had little impact on practice of RNNs. Recurrent models trained in practice do not satisfy stability in an obvious manner, suggesting: perhaps training happens in a chaotic regime. Difficulty of training recurrent models has compelled practitioners to successfully replace recurrent models with non-recurrent, feed-forward architectures.

  – Tính ổn định có tầm quan trọng cơ bản trong nghiên cứu hệ thống động. Tuy nhiên, điều đáng ngạc nhiên là tính ổn định có ít tác động đến việc thực hành RNN. Các mô hình hồi quy được đào tạo trong thực hành không đáp ứng được tính ổn định theo cách rõ ràng, cho thấy: có lẽ việc đào tạo diễn ra trong chế độ hỗn loạn. Khó khăn trong việc đào tạo các mô hình hồi quy đã buộc các học viên phải thay thế thành công các mô hình hồi quy bằng các kiến trúc không hồi quy, truyền thẳng.

  This state of affairs raises important unresolved questions.

  > *Is sequence modeling in practice inherently unstable? When & why are recurrent models really needed?*
  > *– Mô hình trình tự trong thực tế có thực sự không ổn định không? Khi nào & tại sao các mô hình tuần hoàn thực sự cần thiết?*

  In this work, shed light on both of these questions through a theoretical & empirical investigation of stability in recurrent models.

  1st prove stable recurrent models can be approximated by feed-forward networks. In particular, not only are models equivalent for *inference*,they are also equivalent for *training* via gradient descent. While easy to contrive (chế tạo) nonlinear recurrent models that are some input sequence cannot be approximated by feed-forward models, our result implies such models are inevitably unstable. I.e., in particular they must have exploding gradients, which is in general an impediment to learnability via gradient descent.

  – đầu tiên chứng minh các mô hình hồi quy ổn định có thể được xấp xỉ bằng các mạng truyền thẳng. Đặc biệt, các mô hình không chỉ tương đương với *suy luận*, mà còn tương đương với *đào tạo* thông qua giảm dần độ dốc. Trong khi các mô hình hồi quy phi tuyến tính dễ chế tạo (chế tạo) có một số chuỗi đầu vào không thể được xấp xỉ bằng các mô hình truyền thẳng, kết quả của chúng tôi ngụ ý rằng các mô hình như vậy chắc chắn không ổn định. Tức là, cụ thể chúng phải có độ dốc bùng nổ, nói chung là trở ngại đối với khả năng học thông qua giảm dần độ dốc.

  2nd, across a variety of different sequence tasks, show how *recurrent models can often be made stable without loss in performance*. Also show models that are nominally unstable often operate in stable regime on data distribution. Combined with our 1st result, these observation helps to explain why an increasingly large body of empirical research succeeds in replacing recurrent models with feed-forward models in important applications, including translation [6,26], speech synthesis [25], & language modeling [5]. While stability does not always hold in practice to begin with, often possible to generate a high-performing stable model by *imposing stability* during training.

  Our results also shed light on effective representational properties of recurrent networks trained in practice. In particular, stable models cannot have long-term memory. Therefore, when stable & unstable models achieve similar results, either task does not require long-term memory, or unstable model does not have it.

  - 1.1. Contributions. In this work, make following contributions.

    1. Present a generic definition of stable recurrent models in terms of nonlinear dynamical systems & show how to ensure stability of several commonly used models. Previous work establishes stability for vanilla recurrent neural networks. Give new sufficient conditions for stability of long short-term memory (LSTM) networks. These sufficient conditions come with an efficient projection operator that can be used at training time to enforce stability.
    2. Prove, under stability assumption, feed-forward networks can approximate recurrent networks for purposes of both inference & training by gradient descent. While simple in case of inference, training result lies on non-trivial stability properties of gradient descent.
    3. Conduct extensive experimentation on a variety of sequence benchmarks, show stable models often have comparable performance with their unstable counterparts, & discuss when, if ever, there is an intrinsic performance price to using stable models.

- 2. Stable Recurrent Models. Define *stable recurrent models* & illustrate concept for various popular model classes. From a pragmatic perspective, stability roughly corresponds to criterion that gradients of training objective do not *explode* over time. Common recurrent models can operate in both stable & unstable regimes, depending on their parameters. To study stable variants of common architectures, give sufficient conditions to ensure stability & describe how to efficiently enforce these conditions during training.

  - 2.1. Defining Stable Recurrent Models. A *recurrent model* is a nonlinear dynamical system given by a differentiable *state-transition map* $\phi_{\mathbf{w}} : \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}^n$, parameterized by $\mathbf{w} \in \mathbb{R}^m$. Hidden state $h_t \in \mathbb{R}^n$ evolves in discrete time steps according to update rule

$$\boxed{h_t = \phi_w(h_{t-1}, x_t)},$$

    where vector $x_t \in \mathbb{R}^d$ is an arbitrary input provided to system at time t. This general formulation allows us to unify many examples of interest. E.g., for a RNN, given weight matrices W, U, state evolves according to

$$h_t = \phi_{W,U}(h_{t-1}, x_t) = \tanh(Wh_{t-1} + Ux_t).$$

    Recurrent models are typically trained using some variant of gradient descent. 1 natural – even if not strictly necessary – requirement for gradient descent to work: gradients of training objective do not explode over time. *Stable recurrent models* are precisely class of models where gradients cannot explode. They thus constitute a natural class of models where gradient descent can be expected to work. In general, define a stable recurrent model as follows.

    Definition 20. *A recurrent model $\phi_{\mathbf{w}}$ is* stable *if there exists some $\lambda < 1$ s.t., for any weights $\mathbf{w} \in \mathbb{R}^m$, states $\mathbf{h}, \mathbf{h}' \in \mathbb{R}^n$, & input $\mathbf{x} \in \mathbb{R}^d$,*

$$\|\phi_{\mathbf{w}}(\mathbf{h}, \mathbf{x}) - \phi_{\mathbf{w}}(\mathbf{h}', \mathbf{x})\| \leq \lambda \|\mathbf{h} - \mathbf{h}'\|.$$

    Equivalently, a recurrent model is stable if map $\phi_{\mathbf{w}}$ is λ-contractive in $\mathbf{h}$. If $\phi_{\mathbf{w}}$ is λ-stable, then $\|\nabla_{\mathbf{h}} \phi_{\mathbf{w}}(\mathbf{h}, \mathbf{x})\| < \lambda$, & for Lipschitz loss p, $\|\nabla_{\mathbf{w}} p\|$ is always bounded [21].

    Stable models are particularly well-behaved & well-justified from a theoretical perspective. E.g., at present, only stable linear dynamical systems are known to be learnable via gradient descent [7]. In unstable models, gradients of objective can explode, & it is a delicate matter to even show: gradient descent converges to a stationary point. Following proposition offers 1 such example. Proof is provided in appendix.

    Proposition 11. *There exists an unstable system $\phi_{\mathbf{w}}$ where gradient descent does not converge to a stationary point, & $\|\nabla_{\mathbf{w}} p\| \to \infty$ as number of iterations $N \to \infty$.*

  - 2.2. Examples of Stable Recurrent Models. Provide sufficient conditions to ensure stability for several common recurrent models. These conditions offer a way to require learning happens in stable regime – after each iteration of gradient descent, one imposes corresponding stability condition via projection.

    * Linear dynamical systems & RNNs. Given a Lipschitz, pointwise nonlinearity ρ & matrices $W \in \mathbb{R}^{n \times n}, U \in \mathbb{R}^{n \times d}$, state-transition map for a RNN is

$$h_t = \rho(Wh_{t-1} + Ux_t).$$

      If ρ is identity, then system is a linear dynamical system, [10] show if ρ is L_ρ-Lipschitz, then model is stable provided $\|W\| < \frac{1}{L_\rho}$. Indeed, for any states $\mathbf{h}, \mathbf{h}'$, & any $\mathbf{x}$

$$\|\rho(W\mathbf{h} + U\mathbf{x}) - \rho(W\mathbf{h}' + U\mathbf{x})\| \leq L_\rho \|W\mathbf{h} + U\mathbf{x} - W\mathbf{h}' - U\mathbf{x}\| \leq L_\rho \|W\| \|\mathbf{h} - \mathbf{h}'\|.$$

      In case of a linear dynamical system, model is stable provided $\|W\| < 1$. Similarly, for 1-Lipschitz tanh-nonlinearity, stability obtains provided $\|W\| < 1$. In appendix, verify assumptions required by theorems given in next sect for this example. Imposing this condition during training corresponds to projecting onto spectral norm ball.

    * Long short-term memory networks. Long Short-Term Memory (LSTM) networks are another commonly used class of sequence models [9]. State is a pair of vectors $\mathbf{s} = (c, h) \in \mathbb{R}^{2d}$, & model is parameterized by 8 matrices, $W_\square \in \mathbb{R}^{d \times d}, U_\square \in \mathbb{R}^{d \times n}$, for $\square \in \{i, f, o, z\}$. State-transition map ϕ_{LSTM} is given by

$$\begin{aligned}
f_t &= \sigma(W_f \mathbf{h}_{t-1} + U_f \mathbf{x}_t), \\
i_t &= \sigma(W_i \mathbf{h}_{t-1} + U_i \mathbf{x}_t), \\
o_t &= \sigma(W_o \mathbf{h}_{t-1} + U_o \mathbf{x}_t), \\
z_t &= \tanh(W_z \mathbf{h}_{t-1} + U_z \mathbf{x}_t), \\
c_t &= i_t \circ z_t + f_t \circ c_{t-1}, \\
\mathbf{h}_t &= o_t \cot \tanh c_t,
\end{aligned}$$

      where $\circ$ denotes elementwise multiplication, & σ: logistic function.
      Provide conditions under which iterated system $\phi^r_{\text{LSTM}} = \phi_{\text{LSTM}} \circ \cdots \circ \phi_{\text{LSTM}}$ is stable. Let $\|f\|_\infty = \sup_t \|f_t\|_\infty$. If weights W_f, U_f & inputs $\mathbf{x}_t$ are bounded, then $\|f\|_\infty < 1$ since $|\sigma| < 1$ for any finite input. I.e., next state c_t must "forget" a nontrivial portion of c_{t-1}. Leverage this phenomenon to give sufficient conditions for ϕ_{LSTM} to be contractive in l_∞ norm, which in turn implies iterated system ϕ^r_{LSTM} is contractive in l_2 norm for $r = O(\log d)$. Let $\|W\|_\infty$ denote induced l_∞ matrix norm, which corresponds to maximum absolute row sum $\max_i \sum_j |W_{ij}|$.

Proposition 12. *If* $\|W_i\|_\infty, \|W_o\|_\infty < 1 - \|f\|_\infty, \|W_z\|_\infty \leq \frac{1}{4}(1 - \|f\|_\infty), \|W_f\|_\infty < (1 - \|f\|_\infty)^2$, *&* $r = O(\log d)$, *then iterated system* ϕ^r_{LSTM} *is stable.*

Proof is given in appendix. Conditions given in Prop. 2 are fairly restrictive. Somewhat surprisingly show in experiments models satisfying these stability conditions still achieve good performance on a number of tasks. Leave it as an open problem to find different parameter regimes where system is stable, as well as resolve whether original system ϕ_{LSTM} is stable. Imposing these conditions during training & corresponds to simple row-wise normalization of weight matrices & inputs.

- 3. Stable Recurrent Models Have Feed-forward Approximations. Prove stable recurrent models can be well-approximated by feed-forward networks for purposes of both inference & training by gradient descent. From a memory perspective, stable recurrent models are *equivalent* to feed-forward networks – both models use same amount of context to make predictions. This equivalence has important consequences for sequence modeling in practice. When a stable recurrent model achieves satisfactory performance on some tasks, a feedforward network can achieve similar performance. Consequently, if sequence learning in practice is inherently stable, then recurrent models may not be necessary. Conversely, if feed-forward models cannot match performance of recurrent models, then sequence learning in practice is in unstable regime.

  – Chứng minh các mô hình hồi quy ổn định có thể được xấp xỉ tốt bằng các mạng truyền thẳng cho mục đích suy luận & đào tạo bằng phương pháp giảm dần độ dốc. Theo quan điểm về bộ nhớ, các mô hình hồi quy ổn định *tương đương* với các mạng truyền thẳng – cả hai mô hình đều sử dụng cùng một lượng ngữ cảnh để đưa ra dự đoán. Sự tương đương này có những hậu quả quan trọng đối với mô hình chuỗi trong thực tế. Khi một mô hình hồi quy ổn định đạt được hiệu suất thỏa đáng trên một số tác vụ, một mạng truyền thẳng có thể đạt được hiệu suất tương tự. Do đó, nếu việc học chuỗi trong thực tế vốn đã ổn định, thì các mô hình hồi quy có thể không cần thiết. Ngược lại, nếu các mô hình truyền thẳng không thể khớp với hiệu suất của các mô hình hồi quy, thì việc học chuỗi trong thực tế đang ở chế độ không ổn định.

  - 3.1. Truncated recurrent models. For our purposes, salient distinction (sự phân biệt nổi bật) between a recurrent & feed-forward model is the latter has *finite-context*. Therefore, say a model is *feed-forward* if prediction made by model at step t is a function only of inputs $x_{t-k}, \ldots, x_t$ for some finite k.

    While there are many choices for a feed-forward approximation, consider simplest one – truncation of system to some finite context k. I.e., feed-forward approximation moves over input sequence with a sliding window of length k producing an output every time sliding window advances by 1 step. Formally, for context length k chosen in advance, define *truncated model* via update rule

    $$\mathbf{h}^k_t = \phi_{\mathbf{w}}(\mathbf{h}^k_{t-1}, \mathbf{x}_t),\ \mathbf{h}^k_{t-k} = \mathbf{0}.$$

    Note: $\mathbf{h}^k_t$ is a function only of previous k inputs $\mathbf{x}_{t-k}, \ldots, \mathbf{x}_t$. While this definition is perhaps an abuse of term "feed-forward", truncated model can be implemented as a standard autoregressive, depth-k feed-forward network, albeit with significant weight sharing.

    Let f denote a prediction function that maps a state $\mathbf{h}_t$ to outputs $f(\mathbf{h}_t) = y_t$. Let y^k_t denote predictions from truncated model. To simplify presentation, prediction function f is not parameterized. This is w.l.o.g. because always possible to fold parameters into system $\phi_{\mathbf{w}}$ itself. In sequel, study $\|y_t - y^k_t\|$ both during & after training.

  - 3.2. Approximation during inference. Suppose train a full recurrent model $\phi_{\mathbf{w}}$ & obtain a prediction y_t. For an appropriate choice of context k, truncated model makes essentially same prediction y^k_t as full recurrent model. To show this result, 1st control difference between hidden states of both models.

    Lemma 5. *Assume* $\phi_{\mathbf{w}}$ *is* λ*-contractive in* $\mathbf{h}$ *&* L_x*-Lipschitz in* x. *Assume input sequence* $\|x_t\| \leq B_x\ \forall t$. *If truncation length* $k \geq \log_{\frac{1}{\lambda}} \frac{L_x B_x}{(1-\lambda)\varepsilon}$, *then difference in hidden states* $\|\mathbf{h}_t - \mathbf{h}^k_t\| \leq \varepsilon$.

    Lemma 1 effectively says stable models do not have long-term memory – distant inputs do not change states of system. If prediction function is Lipschitz, Lemma 1 immediately implies recurrent & truncated model make nearly identical predictions.

    Proposition 13. *If* $\phi_{\mathbf{w}}$ *is a* L_x*-Lipschitz &* λ*-contractive map, &* f *is* L_f *Lipschitz, & truncation length* $k \geq \log_{\frac{1}{\lambda}} \frac{L_x B_x}{(1-\lambda)\varepsilon}$, *then* $\|y_t - y^k_t\| \leq \varepsilon$.

  - 3.3. Approximation during training via gradient descent. Equipped with our inference result, return towards optimization. Show gradient descent for stable recurrent models at same point & track divergence in weights throughout course of gradient descent. Roughly, show if $k \approx O(\log \frac{N}{\varepsilon})$, then after N steps of gradient descent, difference in weights between recurrent & truncated models is at most ε. Even if gradients are similar for both models at same point, it is a priori possible: slight differences in gradients accumulate over time & lead to divergent weights where no meaningful comparison is possible. Building on similar techniques as [8], show: gradient descent itself is stable, & this type of divergence cannot occur.

    Our gradient descent result requires 2 essential lemmas. 1st bounds difference in gradient between full & truncated model. 2nd establishes gradient map of both full & truncated models is Lipschitz. Defer proofs of both lemmas to appendix.

    Let p_T denote loss function evaluated on recurrent model after T time steps, & define p^k_T similarly for truncated model. Assume there compact, convex domain $\Phi \subset \mathbb{R}^n$ so that map $\phi_{\mathbf{w}}$ is stable $\forall$ choices of parameters $\mathbf{w} \in \Theta$.

    Lemma 6. *Assume* p *(& therefore* p^k*) is Lipschitz & smooth. Assume* $\phi_{\mathbf{w}}$ *is smooth,* λ*-contractive, & Lipschitz in* x *&* $\mathbf{w}$. *Assume inputs satisfy* $\|\mathbf{x}_t\| \leq B_x$, *then*

    $$\|\nabla_{\mathbf{w}} p_T - \nabla_{\mathbf{w}} p^k_T\| = \gamma k \lambda^k,$$

    where $\gamma = O(B_{\mathbf{x}}(1-\lambda)^{-2})$, *suppressing dependence on Lipschitz & smoothness parameters.*

Lemma 7. *For any $w, w' \in \Theta$, suppose $\phi_{\mathbf{w}}$ is smooth, λ-contractive, & Lipschitz in $\mathbf{w}$. If p is Lipschitz & smooth, then*

$$\|\nabla_{\mathbf{w}} p_T(\mathbf{w}) - \nabla_{\mathbf{w}} p_T(\mathbf{w}')\| \leq \beta \|\mathbf{w} - \mathbf{w}'\|,$$

where $\beta = O((1-\lambda)^{-3})$, suppressing dependence on Lipschitz & smoothness parameters.

Let w^i_{recurr}: weights of recurrent model on step i & define w^i_{trunc} similarly for truncated model. At initialization, $w^0_{\text{recurr}} = w^0_{\text{trunc}}$. For k sufficiently large, Lemma 2 guarantees difference between gradient of recurrent & truncated models is negligible. Therefore, after a gradient update, $\|w^1_{\text{recurr}} - w^1_{\text{trunc}}\|$ is small. Lemma 3 then guarantees: this small difference in weights does not lead to large differences in gradient on subsequent time step. For an appropriate choice of learning rate, formalizing this argument leads to following proposition.

Proposition 14. *Under assumptions of Lemmas 2–3, for compact, convex Θ, after N steps of projected gradient descent with step size $\alpha_t = \frac{\alpha}{t}$, $\|w^N_{\text{recurr}} - w^N_{\text{trunc}}\| \leq \alpha\gamma k \lambda^k N^{\alpha\beta+1}$.*

Decaying step size in our theorem is consistent with regime in which gradient descent is known to be stable for non-convex training objectives [8]. While decay is faster than many learning rates encountered in practice, classical results nonetheless show: with this learning rate gradient descent still converges to a stationary point; see p. 119 in [4] & references there. In appendix, give empirical evidence $O(\frac{1}{t})$ rate is necessary for our theorem & show examples of stable systems trained with constant or $O(\frac{1}{\sqrt{t}})$ rates that do not satisfy our bound.

Critically, bound in Prop. 4 goes to 0 as $k \to \infty$. In particular, if take $\alpha = 1$ & $k \geq \Omega \log \frac{\gamma N^\beta}{\varepsilon}$, then after N steps of projected gradient descent, $\|w^N_{\text{recurr}} - w^N_{\text{trunc}}\| \leq \varepsilon$. For this choice of k, obtain main theorem. Proof is left to appendix.

Theorem 5. *Let p be Lipschitz & smooth. Assume $\phi_{\mathbf{w}}$ is smooth, λ-contractive, Lipschitz in x & $\mathbf{w}$. Assume inputs are bounded, & prediction function f is L_f-Lipschitz. If $k \geq \Omega(\log \frac{\gamma N^\beta}{\varepsilon})$, then after N steps of projected gradient descent with step size $\alpha_t = \frac{1}{t}$, $\|y_T - y^k_T\| \leq \varepsilon$.*

- 4. Experiments. In experiments, show stable recurrent models can achieve solid performance on several benchmark sequence tasks. Namely, show unstable recurrent models can often be made stable without a loss in performance. In some cases, there is a small gap between performance between unstable & stable models. Analyze whether this gap is indicative of a "price of stability" & show unstable methods involved are stable in a data-dependent sense.
  - 4.1. Tasks. Consider 4 benchmark sequence problems – word-level language modeling, character-level language modeling, polyphonic music modeling, & slot-filling.
    - **Language modeling.** In language modeling, given a sequence of words or characters, model must predict next word or character. For character-level language modeling, train & evaluate models on Penn Treebank [15]. To increase coverage of our experiments, train & evaluate word-level language models on Wikitext-2 dataset, which is twice as large as Penn Treemark & features a larger vocabulary [17]. Performance is reported using bits-per-character for character-level models & perplexity for word-level models.
    - **Polyphonic music modeling.** In polyphonic music modeling, a piece is represented as a sequence of 88-bit binary codes corresponding to 88 keys on a piano, with a 1 indicating a key that is pressed at a given time. Given a sequence of codes, task: predict next code. Evaluate our models on JSB Chorales, a polyphonic music dataset consisting of 382 harmonized chorales by J.S. Bach [1]. Performance is measured using negative log-likelihood.
    - **Slot-filling.** In slot filling, model takes as input a query like "I want to Boston on Monday" & outputs a class label for each word in input, e.g. Boston maps to `Departure_City` & Monday maps to `Departure_Time`. Use Airline Travel Information Systems (ATIS) benchmark & report F1 score for each model [22].
  - 4.2. Cf. Stable & Unstable Models. For each task, 1st train an unconstrained RNN & an unconstrained LSTM. All hyperparameters are chosen via grid-search to maximize performance of unconstrained model. For consistency with our theoretical results in Sect. 3 & stability conditions in Sect. 2.2, both models have a single recurrent layer & are trained using plain SGD. In each case, resulting model is unstable. However, then retrain best models using projected gradient descent to enforce stability *without retuning hyperparameters*. In RNN case, constrain $\|W\| < 1$. After each gradient update, project W onto spectral norm ball by computing SVD & thresholding singular values to lie in $[0, 1)$. In LSTM case, after each gradient update, normalize each row of weight matrices to satisfy sufficient conditions for stability given in Sect. 2.2.

    Stable & unstable models achieve similar performance. Table 1: Comparison of stable & unstable models on a variety of sequence modeling tasks. For all tasks, stable & unstable RNNs achieve same performance. For polyphonic music & slot-filling, stable & unstable LSTMs achieve same results. On language modeling, there is a small gap between stable & unstable LSTMs. Discuss this in Sect. 4.3. Performance is evaluated on held-out test set. For negative log-likelihood (nll), bits per character (bpc), & perplexity, lower is better. For F1 score, higher is better. gives a comparison of performance between stable & unstable RNNs & LSTMs on each of different tasks. Each of reported metrics is computed on held-out test set. Also show a representative comparison of learning curves for word-level language modeling & polyphonic music modeling in Fig. 1: Stable & unstable variants of common recurrent architectures achieve similar performance across a range of different sequence tasks.

    Across all tasks considered, stable & unstable RNNs have roughly same performance. Stable RNNs & LSTMs achieve results comparable to published baselines on slot-filling [18] & polyphonic music modeling [3]. On word & character level language modeling, both stable & unstable RNNs achieve comparable results to [3].

    On language modeling tasks, however, there is a gap between stable & unstable LSTM models. Given restrictive conditions place on LSTM to ensure stability, surprising they work as well as they do. Weaker conditions ensuring stability of LSTM could reduce this gap. Also possible imposing stability comes at a cost in representational capacity required for some tasks.

- 4.3. What is "price of stability" in sequence modeling? Gap between stable & unstable LSTMs on language modeling raises question of whether there is an intrinsic performance cost for using stable models on some tasks. If measure stability in a data-dependent fashion, then unstable LSTM language models are stable, indicating this gap is illusory (ảo tưởng). However, in some cases with short sequences, instability can offer modeling benefits.
  - **LSTM language models are stable in a "data-dependent" way.** Our notion of stability is conservative & requires stability to hold for every input & pair of hidden states. If instead consider a weak, data-dependent notion of stability, word & character-level LSTM models are stable (in iterated sense of Prop. 2). In particular, compute stability parameter only *using input sequences from data*. Furthermore, only evaluate stability on hidden states *reachable via gradient descent*. More precisely, to estimate λ, run gradient ascent to find worst-case hidden states $\mathbf{h}, \mathbf{h}'$ to maximize $\frac{\|\phi_{\mathbf{w}}(\mathbf{h},\mathbf{x})-\phi_{\mathbf{w}}(\mathbf{h}',\mathbf{x})\|}{\|\mathbf{h}-\mathbf{h}'\|}$ → Appendix.
    Data-dependent definition given above is a useful diagnostic – when sufficient stability conditions fail to hold, data-dependent condition addresses whether model is still operating in stable regime. Moreover, when input representation is fixed during training, our theoretical results go through without modification when using data-dependent definition.
    Using data-dependent measure, in Fig. 2: What is intrinsic "price of stability"? For language modeling, show unstable LSTMs are actually stable in weaker, data-dependent sense. On other hand, for polyphonic music modeling with short sequences, instability can improve model performance. (a) Data-dependent stability of character-level language models. Iterated-LSTM refers to iteration system $\phi^r_{\text{LSTM}} = \phi_{\text{LSTM}} \circ \cdots \circ \phi_{\text{LSTM}}$., show: iterated character-level LSTM ϕ^r_{LSTM} is stable for $r \approx 80$ iterations. A similar result holds for word-level language model for $r \approx 100$. These findings are consistent with experiments in [14] which find LSTM trajectories converge after ≈ 70 steps *only when evaluated on sequences from data*. For language models, "price of stability" is therefore much smaller than gap in Table 1 suggests – even "unstable" models are operating in stable regime on data distribution.
  - **Unstable systems can offer performance improvements for short-time horizons.** When sequences are short, training unstable models is less difficult because exploding gradients are less of an issue. In these case, unstable models can offer performance gains. To demonstrate this, train truncated unstable models on polyphonic music task for various values of truncation parameter k. In Fig. 2.(b): Unstable models can boost performance for short sequences., simultaneously plot performance of unstable model & stability parameter λ for converged model for each k. For short-sequences, final model is more unstable $\lambda \approx 3.5$ & achieves a better test-likelihood. For longer sequence lengths, λ decreases closer to stable regime $\lambda \approx 1.5$, & this improved test-likelihood performance disappears.
- 4.4. Unstable Models Operate in Stable Regime. In prev sect, showed: nominally unstable models often satisfy a data-dependent notion of stability (các mô hình không ổn định về mặt danh nghĩa thường thỏa mãn khái niệm về tính ổn định phụ thuộc vào dữ liệu). In this sect, offer further evidence unstable models are operating in stable regime. These results further help explain why stable & unstable models perform comparably in experiments.
  - **Vanishing gradients.** Stable models necessarily have vanishing gradients, & indeed this ingredient is a key ingredient in proof of our training-time approximation result. For both word & character-level language models, find both *unstable RNNs & LSTMs also exhibit vanishing gradients*. In Fig. 3: Unstable word & character-level language models exhibit vanishing gradients. Plot norm of gradient w.r.t. inputs $\|\nabla_{x_t} p_{t+i}\|$, as distance between input & loss grows, averaged over entire training set. Gradient vanishes for moderate values of i for both RNNs & LSTMs, though decay is slower for LSTMs. (a) Distance i from input x_t to loss p_{t+i}. (b) Character-level language modeling., plot average gradient of loss at time $t+i$ w.r.t. input at time t, $\|\nabla_{x_t} p_{t+i}\|$ as t ranges over training set. For either language modeling task, LSTM & RNN suffer from limited sensitivity to distance inputs at initialization & throughout training. Gradients of LSTM vanish more slowly than those of RNN, but both models exhibit same qualitative behavior.
  - **Truncating Unstable Models.** Results in Sect. 3 show stable models can be truncated without loss of performance. In practice, unstable models can also be truncated without performance loss. In Fig. 4: Effect of truncating unstable models. On both language & music modeling, RNNs & LSTMs exhibit diminishing returns for large values of truncation parameter k. In LSTMs, larger k doesn't affect performance, whereas for unstable RNNs, large k slightly decreases performance. (a) Word-level language modeling. (b) Polyphonic music modeling., show: performance of both LSTMs & RNNs for various values of truncation parameter k on word-level language modeling & polyphonic music modeling. Initially, increasing k increases performance because model can use more context to make predictions. However, in both cases, there is diminishing returns to larger values of truncation parameter k. LSTMs are unaffected by longer truncation lengths, whereas performance of RNNs slight degrades as k becomes very large, possibly due to training instability. In either case, diminishing returns to performance for large values of k means truncation & therefore feed-forward approximation is possible even for these unstable models.
  - **Prop. 4 holds for unstable models.** In stable models, Prop. 4 in Sect. 3 ensures distance between weight matrices $\|\mathbf{w}_{\text{recurr}} - \mathbf{w}_{\text{trunc}}\|$ grows slowly as training progresses, & this rate decreases as k becomes large. In Fig. 5: Qualitative version of Prop. 4 for unstable, word-level language models. Assume $k = 65$ well-captures full-recurrent model & plot $\|\mathbf{w}_{\text{recurr}} - \mathbf{w}_{\text{trunc}}\| = \|W_k - W_{65}\|$ as training proceeds, where W denotes recurrent weights. As Pro. 4 suggests, this quantity grows slowly as training proceeds, & rate of growth decreases as k increases. (a) Unstable RNN language model. (b) Unstable LSTM language model., plot $\|W_k - W_{65}\|$ for $k \in \{5, 10, 15, 25, 35, 50, 64\}$ throughout training. As suggested by Prop. 4, after an initial rapid increase in distance, $\|W_k - W_{65}\|$ grows slowly. Moreover, there is a diminishing return to choosing larger values of truncation parameter k in terms of accuracy of approximation.

- 5. Are recurrent models truly necessary? Our experiments show recurrent models trained in practice operate in stable regime, & our theoretical results show stable recurrent models are approximable by feed-forward networks. As a consequence, conjecture

recurrent networks trained in practice are always approximable by feed-forward networks. Even with this conjecture, cannot yet conclude recurrent models as commonly conceived are unnecessary. 1st, our present proof techniques rely on truncated versions of recurrent models, & truncated recurrent architectures like LSTMs may provide useful inductive bias on some problems. Moreover, implementing truncated approximation as a feed-forward network increases number of weights by a factor of k over original recurrent model. Declaring recurrent models truly superfluous would require both finding more parsimonious feed-forward approximations & proving natural feed-forward models, e.g., fully connected networks or CNNs, can approximate stable recurrent models during training. This remains an important question for future work.

– Các mô hình hồi quy có thực sự cần thiết không? Các thí nghiệm của chúng tôi cho thấy các mô hình hồi quy được đào tạo trong thực tế hoạt động ở chế độ ổn định, & kết quả lý thuyết của chúng tôi cho thấy các mô hình hồi quy ổn định có thể xấp xỉ được bằng các mạng truyền thẳng. Do đó, phỏng đoán *các mạng truyền thẳng được đào tạo trong thực tế luôn có thể xấp xỉ được bằng các mạng truyền thẳng*. Ngay cả với phỏng đoán này, vẫn chưa thể kết luận các mô hình hồi quy như thường được quan niệm là không cần thiết. Trước tiên, các kỹ thuật chứng minh hiện tại của chúng tôi dựa trên các phiên bản cắt cụt của các mô hình hồi quy, & các kiến trúc hồi quy cắt cụt như LSTM có thể cung cấp độ lệch quy nạp hữu ích cho một số vấn đề. Hơn nữa, việc triển khai phép xấp xỉ cắt cụt như một mạng truyền thẳng làm tăng số lượng trọng số lên gấp k so với mô hình hồi quy ban đầu. Việc tuyên bố các mô hình hồi quy thực sự là thừa sẽ yêu cầu cả việc tìm ra các phép xấp xỉ truyền thẳng tiết kiệm hơn & chứng minh các mô hình truyền thẳng tự nhiên, ví dụ như các mạng hoàn toàn kết nối hoặc CNN, có thể xấp xỉ các mô hình hồi quy ổn định trong quá trình đào tạo. Đây vẫn là một câu hỏi quan trọng cho công việc trong tương lai.

- 6. Related Work. Learning dynamical systems with gradient descent has been a recent topic of interest in ML community. [7] shows gradient descent can efficiently learn a class of stable, linear dynamical systems, [20] shows gradient descent learns a class of stable, nonlinear dynamical systems. Work by [23] gives a moment-based approach for learning some classes of stable nonlinear RNNs. Our work explores theoretical & empirical consequences of stability assumption made in these works. In particular, our empirical results show models trained in practice can be made closer to those currently being analyzed theoretically without large performance penalties.

  For *linear* dynamical systems, [24] exploit connection between stability & truncation to learn a truncated approximation to full stable system. Their approximation result is same as our inference result for linear dynamical systems, & extend this result to nonlinear setting. Also analyze impact of truncation on training with gradient descent. Our training time analysis builds on stability analysis of gradient descent in [8], but increasingly uses it for an entirely different purpose. Results of this kind are completely new to our knowledge.

  For RNNs, link between vanishing & exploding gradients & $\|W\|$ was identified in [21]. For 1-layer RNNs, [10] give sufficient conditions for stability in terms of norm $\|W\|$ & Lipschitz constant of nonlinearity. Our work additionally considers LSTMs & provides new sufficient conditions for stability. Moreover, study consequences of stability in terms of feed-forward approximation.

  A number of recent works have sought to avoid vanishing & exploding gradients by ensuring system is an isometry, i.e. $\lambda = 1$. In RNN case, this amounts to constraining $\|W\| = 1$ [2,11,12,19,28]. [27] observes strictly requiring $\|W\| = 1$ reduces performance on several tasks, & instead proposes maintaining $\|W\| \in [1-\varepsilon, 1+\varepsilon]$. [29] maintains this "soft-isometry" constraint using a parameterization based on SVD that obviates (loại bỏ) need for projection step used in our stable-RNN experiments. [13] sidestep these issues & stabilizes training using a residual parameterization of model. At present, these unitary models have not yet seen widespread use, & our work shows much of sequence learning in practice, even with nominally unstable models, actually occurs in stable regime.

  From an empirical perspective, [14] introduce a non-chaotic recurrent architecture & demonstrate it can perform as well more complex models like LSTMs. [3] conducts a detailed evaluation of recurrent & convolutional, feed-forward models outperform their recurrent counterparts. Their experiments are complimentary to ours; find recurrent models can often be replaced with stable recurrent models, which show are equivalent to feed-forward models.

- A. Proofs from Sect. 2.
  - A.2.1. RNNs.
  - A.2.2. LSTMs.
- B. Proofs from Sect. 3.
  - B.1. Proofs from Sect. 3.3.
    - B.1.1. Gradient difference rule to truncation is negligible. Argue difference in gradient w.r.t. weights between recurrent & truncated models is $O(k\lambda^k)$. Fur sufficiently large k (independent of sequence length), impact of truncation is therefore negligible. Proof leverages "vanishing-gradient" phenomenon – long-term components of gradient of full recurrent model quickly vanish. Remaining challenge: show short-term components of gradient are similar for full & recurrent models.
    - B.1.2. Stable recurrent models are smooth. Prove: gradient map $\nabla_{\mathbf{w}} p_T$ is Lipschitz. 1st, show on forward pass, difference between hidden states $\mathbf{h}_t(\mathbf{w}), \mathbf{h}'_t(\mathbf{w}')$ obtained by running model with weights $\mathbf{w}, \mathbf{w}'$, resp., is bounded in terms of $\|\mathbf{w}-\mathbf{w}'\|$. Using smoothness of ϕ, difference in gradients can be written in terms of $\|\mathbf{h}_t(\mathbf{w}) - \mathbf{h}'_t(\mathbf{w}')\|$, which in turn can be bounded in terms of $\|\mathbf{w} - \mathbf{w}'\|$. Repeatedly leverage this fact to conclude total difference in gradients must be similarly bounded. 1st show small differences in weights don't significantly change trajectory of recurrent model.

Lemma 8. *For some $\mathbf{w}, \mathbf{w}'$, suppose $\phi_{\mathbf{w}}, \phi_{\mathbf{w}'}$ are λ-contractive & $L_{\mathbf{w}}$ Lipschitz in $\mathbf{w}$. Let $\mathbf{h}_t(\mathbf{w}), \mathbf{h}_t(\mathbf{w}')$ be hidden state at time t obtain from running model with weights $\mathbf{w}, \mathbf{w}'$ on common inputs $\{\mathbf{x}_t\}$. If $\mathbf{h}_0(\mathbf{w}) = \mathbf{h}_0(\mathbf{w}')$, then*

$$\|\mathbf{h}_t(\mathbf{w}) - \mathbf{h}_t(\mathbf{w}')\| \leq \frac{L_{\mathbf{w}}\|\mathbf{w} - \mathbf{w}'\|}{1-\lambda}.$$

Proof of Lem. 3 is similar in structure to Lem. 2 & follows from repeatedly using smoothness of ϕ & Lem. 5.

∗ B.1.3. Gradient descent analysis. Equipped with smoothness & truncation lemmas (Lems. 2–3), turn towards proving main gradient descent result.

- C. Experiments.

  - $O(\frac{1}{t})$ **rate may be necessary.** Key result underlying Thm. 1 is bound on parameter difference $\|\mathbf{w}_{\text{trunc}} - \mathbf{w}_{\text{recurr}}\|$ while running gradient descent obtained in Prop. 4. Show this bound has correct qualitative scaling using random instances & training randomly initialized, stable linear dynamical systems & tanh-RNNs. In Fig. 6, plot parameter error $\|\mathbf{w}^t_{\text{trunc}} - \mathbf{w}_{\text{recurr}}\|$ as training progresses for both models (averaged > 10 runs). Error scales comparably with bound given in Prop. 4. Also find for larger step-sizes like $\frac{\alpha}{\sqrt{t}}$ or constant α, bound fails to hold, suggesting $O(\frac{1}{t})$ condition is necessary.

    Concretely, generate random problem instance by fixing a sequence length $T = 200$, sampling input data $x_t \overset{\text{i.i.d}}{\sim} \mathcal{N}(0, 4 \cdot I_{32})$, & sampling $y_T \sim \text{Unif}[-2, 2]$. Next, set $\lambda = 0.75$ & randomly initialize a stable linear dynamical system or RNN with tanh nonlinearity by sampling $U_{ij}, W_{ij} \overset{\text{i.i.d}}{\sim} \mathcal{N}(0, 0.5)$ & thresholding singular values of W so $\|W\| \leq \lambda$. Use squared loss & prediction function $f(\mathbf{h}_t, \mathbf{x}_t) = C\mathbf{h}_t + D\mathbf{x}_t$, wehre $C, D \overset{\text{i.i.d}}{\sim} \mathcal{N}(0, I_{32})$. Fix truncation length to $k = 35$, set learning rate to $\alpha_t = \frac{\alpha}{t}$ for $\alpha = 0.01$, & take $N = 200$ gradient steps. These parameters are chosen so that $\gamma k \lambda^k N^{\alpha\beta+1}$ bound from Prop. 4 does not become vacuous (trống rỗng) – by triangle inequality, always have $\|\mathbf{w}_{\text{trunc}} - \mathbf{w}_{\text{recurr}}\| \leq 2\lambda$.

  - **Stable vs. unstable models.** Word & character level language modeling experiments are based on publically available code from [16]. Polyphonic music modeling code is based on code in [3], & slot-filling model is a reimplementation of [18]: Word-level language modeling code is based on https://github.com/pytorch/examples/tree/master/word_language_model, character-level code is based on https://github.com/salesforce/awd-lstm-lm, & polyphonic music modeling code is based on https://github.com/locuslab/TCN.

    Since sufficient conditions for stability derived in Sect. 2.2. only apply for networks with a single layer, use a single layer RNN or LSTM ∀ experiments. Further, our theoretical results are only applicable for vanilla SGD, & not adaptive gradient methods, so all models are trained with SGD. Table 2: Hyperparameters for all experiments contains a summary of all hyperparameters for each experiment.

    All hyperparameters are shared between stable & unstable variants of both models. In RNN case, enforcing stability is conceptually simple, though computationally expensive. Since tanh is 1-Lipschitz, RNN is stable as long as $\|W\| < 1$. Therefore, after each gradient update, project W onto spectral norm ball by taking SVD & thresholding singular values to lie in $[0, 1)$. In LSTM case, enforcing stability is conceptually more difficult, but computationally simple. To ensure LSTM is stable, appeal to Prop. 2. Enforce following inequalities after each gradient update

    1. Hidden-to-hidden forget gate matrix should satisfy $\|W_f\|_\infty < 0.128$, which is enforced by normalizing l_1-norm of each row to have value at most 0.128.
    2. Input vectors $\mathbf{x}_t$ must satisfy $\|\mathbf{x}_t\|_\infty \leq B_{\mathbf{x}} = 0.75$, which is achieved by thresholding all values to lie in $[-0.75, 0.75]$.
    3. Bias of forget gate b_f, must satisfy $\|b_f\|_\infty \leq 0.25$, which is again achieved by thresholding all values to lie in $[-0.25, 0.25]$.
    4. Input-hidden forget gate matrix U_f should satisfy $\|U_f\|_\infty \leq 0.25$. This is enforced by normalizing l_1-norm of each row to have value at most 0.25.
    5. Given 1–4, forget gate can take value at most $f_\infty < 0.64$. Consequently, enforce $\|W_i\|_\infty, \|W_o\|_\infty \leq 0.36, \|W_z\| \leq 0.091, \|W_f\|_\infty < \min\{0.128, (1 - 0.64)^2\} = 0.128$.

    After 1–5 are enforced, by Prop. 2, resulting (iterated)-LSTM is stable. Although above description is somewhat complicated, implementation boils down to normalizing rows of LSTM weight matrices, which can be done very efficiently in a few lines of PyTorch.

  - **Data-dependent stability.** Unlike RNN, is an LSTM, not clear how to analytically compute stability parameter λ. Instead, rely on a heuristic method to estimate λ. Recall a model is stable if $\forall \mathbf{x}, \mathbf{h}, \mathbf{h}'$, have

$$S(\mathbf{h}, \mathbf{h}', \mathbf{x}) := \frac{\|\phi_{\mathbf{w}}(\mathbf{h}, \mathbf{x}) - \phi_{\mathbf{w}}(\mathbf{h}', \mathbf{x})\|}{\|\mathbf{h} - \mathbf{h}'\|} \leq \lambda < 1.$$

    To estimate $\sup_{\mathbf{h},\mathbf{h}',\mathbf{x}} S(\mathbf{h}, \mathbf{h}', \mathbf{x})$, do: 1st, take $\mathbf{x}$ to be point in training set. In language modeling case, $\mathbf{x}$ is 1 of learned word-vectors. Randomly sample & fix $\mathbf{x}$, & then perform gradient ascent on $S(\mathbf{h}, \mathbf{h}', \mathbf{x})$ to find worst-case $\mathbf{h}, \mathbf{h}'$. In our experiments, initialize $\mathbf{h}, \mathbf{h}' \sim \mathcal{N}(0, 0.1 \cdot I)$ & run gradient ascent with learning rate 0.9 for 1000 steps. This procedure is repeated 20 times, & estimate λ as maximum value of $S(\mathbf{h}, \mathbf{h}', \mathbf{x})$ encountered during any iteration from any of 20 random starting points.

5.4 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 (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.

$$\mathbf{H}_t = \phi_h(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h), \tag{435}$$

$$\mathbf{O}_t = \phi_o(\mathbf{H}_t \mathbf{W}_{ho} + \mathbf{b}_o). \tag{436}$$

  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.

$$\mathbf{H} = \phi_h(\mathbf{X} \mathbf{W}_{xh} + \mathbf{b}_h), \tag{437}$$

$$\mathbf{O} = \phi_o(\mathbf{H} \mathbf{W}_{ho} + \mathbf{b}_o). \tag{438}$$

  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)

$$\mathcal{L}(\mathbf{O}, \mathbf{Y}) = \sum_{t=1}^{T} l_t(\mathbf{O}_t, \mathbf{Y}_t). \tag{439}$$

  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)

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{ho}} = \sum_{t=1}^{T} \frac{\partial l_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \cdot \frac{\partial \phi_o}{\partial \mathbf{W}_{ho}} = \sum_{t=1}^{T} \frac{\partial l_t}{\partial \mathbf{O}_t} \cdot \frac{\partial \mathbf{O}_t}{\partial \phi_o} \mathbf{H}_t. \tag{440}$$

  For partial derivative w.r.t. $\mathbf{W}_{hh}$ get

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{hh}} = \cdots. \tag{441}$$

  For partial derivative w.r.t. $\mathbf{W}_{xh}$ get:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{xh}} = \cdots. \tag{442}$$

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):

$$\mathbf{O}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o), \tag{443}$$

$$\mathbf{I}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i), \tag{444}$$

$$\mathbf{F}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f). \tag{445}$$

  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):

$$\widetilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c). \tag{446}$$

  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)

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \widetilde{\mathbf{C}}_t, \tag{447}$$

  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):

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh \mathbf{C}_t. \tag{448}$$

  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.

$$\mathbf{H}_t^{(1)} = \phi_1(\mathbf{X}_t, \mathbf{H}_{t-1}^{(1)}), \tag{449}$$

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)}, \mathbf{H}_{t-1}^{(l)}). \tag{450}$$

  Output $\mathbf{O}_t \in \mathbb{R}^{n \times o}$ where o: number of outputs is then computed as shown in (21)

$$\mathbf{O}_t = \phi_o(\mathbf{H}_t^{(L)} \mathbf{W}_{ho} + \mathbf{b}_o) \tag{451}$$

  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):

$$\mathbf{Y}_t = \text{softmax}(\text{score}(\mathbf{S}_t, \mathbf{H}_{t'})) = \text{softmax}(\mathbf{v}_a^\top \tanh \mathbf{W}_a[\mathbf{S}_t; \mathbf{H}_{t'}]). \tag{452}$$

  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.

6 Geeksforgeeks

6.1 Introduction to Recurrent Neural Networks

"*Recurrent Neural Networks (RNNs)* were introduced in 1980s by researchers David Rumelhart, Geoffrey Hinton, & Ronald J. Williams. RNNs have laid foundation for advancements in processing sequential data, e.g. natural language & time-series analysis, & continue to influence AI research & applications today.

In this article, explore core principles of RNNs, understand how they function, & discuss why they are essential for tasks where previous inputs in a sequence influence further predictions.

6.1.1 What is RNNs?

In traditional neural networks, inputs & outputs are treated independently. However, tasks like predicting next word in a sentence require information from previous words to make accurate predictions. To address this limitation, RNNs were developed.

RNNs introduce a mechanism where *output from 1 step is fed back as input to next, allowing them to retain information (giữ lại thông tin) from previous inputs*. This design makes RNNs well-suited for tasks where context from earlier steps is essential, e.g. predicting next word in a sentence.

Defining feature of RNNs is their *hidden state* – also called *memory state* – which preserved essential information from previous inputs in sequence. By using same parameters across all steps, RNNs perform consistently across inputs, reducing parameter complexity compared to traditional neural networks. This capability makes RNNs highly effective for sequential tasks.

In simple terms, RNNs apply same network to each element in a sequence, RNNs preserve & pass on relevant information, enabling them to learn temporal dependencies that conventional neural networks cannot.

How RNN differs from Feedforward Neural Networks. Feedforward Neural Networks (FNNs) process data in 1 direction, from input to output, without retaining information from previous inputs. This makes them suitable for tasks with independent inputs, like image classification. However, FNNs struggle with sequential data since they lack memory.

RNNs solve this by incorporating loops that allow information from previous steps to be fed back into network. This feedback enables RNNs to remember prior inputs, making them ideal for tasks where context is important.

6.1.2 Key Components of RNNs

1. 1. Recurrent Neurons. Fundamental processing unit in a RNN is a Recurrent Unit, which is not explicitly called a "Recurrent Neuron". Recurrent units hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.

2. 2. RNN Unfolding. *RNN unfolding* or *"unrolling"* is process of expanding recurrent structure over time steps. During unfolding, each step of sequence is represented as a separate layer in a series, illustrating how information flows across each time step. This unrolling enables *backpropagation through time (BPTT)*, a learning process where errors are propagated across time steps to adjust network's weights, enhancing RNN's ability to learn dependencies within sequential data.

6.1.3 Types of RNNs

There are 4 types of RNNs based on number of inputs & outputs in network:

1. 1. 1-to-1 RNN. *1-to-1 RNN* behaves as *Vanilla Neural Network*, is simplest type of neural network architecture. In this setup, there is a single input & a single output. Commonly used for straightforward classification tasks where input data points do not depend on previous elements.

2. 2. 1-to-many RNN. In a *1-to-many RNN*, network processes a single input to produce multiple outputs over time. This setup is beneficial when a single input element should generate a sequence of predictions.

   E.g., for image captioning task, a single image as input, model predicts a sequence of words as a caption.

3. 3. Many-to-1 RNN. *Many-to-1 RNN* receives a sequence of inputs & generates a single output. This type is useful when overall context of input sequence is needed to make 1 prediction.

   In sentiment analysis (phân tích tình cảm), model receives a sequence of words (like a sentence) & produces a single output, which is sentiment of sentence (positive, negative, or neutral).

4. 4. Many-to-Many RNN. *Many-to-Many RNN* type processes a sequence of inputs & generates a sequence of outputs. This configuration is ideal for tasks where input & output sequences need to align over time, often in a 1-to-1 or many-to-many mapping.

   In language translation task, a sequence of words in 1 language is given as input, & a corresponding sequence is another language is generated as output.

6.1.4 Variants of RNNs

There are several variations of RNNs, each designed to address specific challenges or optimize for certain tasks:

1. 1. Vanilla RNN. This simplest form of RNN consists of a single hidden layer, where weights are shared across time steps. Vanilla RNNs are suitable for learning short-term dependencies but are limited by vanishing gradient problem, which hampers long-sequence learning.

2. 2. Bidirectional RNNs. Bidirectional RNNs process inputs in both forward & backward directions, capturing both past & future context for each time step. This architecture is ideal for tasks where entire sequence is available, e.g. named entity recognition & question answering.

3. 3. Long Short-Term Memory Networks (LSTMs). Long Short-Term Memory Networks (LSTMs) introduced a memory mechanism to overcome vanishing gradient problem. Each LSTM cell has 3 gates:

   - *Input Gate*: Controls how much new information should be added to cell state.
   - *Forget Gate*: Decides what past information should be discarded.
   - *Output Gate*: Regulates what information should be output at current step. This selective memory enables LSTMs to handle long-term dependencies, making them ideal for tasks where earlier context is critical.

4. 4. Gated Recurrent Units (GRUs). Gated Recurrent Units (GRUs) simplify LSTMs by combining input & forget gates into a single update gate & streamlining output mechanism. This design is computationally efficient, often performing similarly to LSTMs, & is useful in tasks where simplicity & faster training are beneficial.

6.1.5 RNN Architecture

RNNs share similarities in input & output structures with other DL architectures but differ significantly in how information flows from input to output. Unlike traditional deep neural networks, where each dense layer has distinct weight matrices, RNNs use shared weights across time steps, allowing them to remember information over sequences.

In RNNs, hidden state H_i is calculated for every input X_i to retain sequential dependencies. Computations follow these core formulas:

- *Hidden State Calculation*: $h = \sigma(U \cdot X + W \cdot h_{t-1} + B)$ where h represents current hidden state, U, W: weight matrices, B: bias.
- *Output Calculation*: $Y = O(V \cdot h + C)$. Output Y is calculated by applying O, an activation function, to weighted hidden state, where V, C represent weights & bias.
- *Overall Function*: $Y = f(X, h, W, U, V, B, C)$. This function defines entire RNN operation, where state matrix S holds each element s_i representing network's state at each time step i.
- *Key Parameters in RNNs*:
  - Weight Matrices: W, U, V
  - Bias Terms: B, C.

  These parameters remain consistent across all time steps, enabling network to model sequential dependencies more efficiently, which is essential for tasks like language processing, time-series forecasting, & more.

6.1.6 How does RNN work?

In a RNN, each time step consists of units with a fixed activation function. Each unit contains an internal hidden state, which acts as memory by retaining information from previous time steps, thus allowing network to store past knowledge. Hidden state h_t is updated at each time step to reflect new input, adapting network's understanding of previous inputs.

Updating Hidden State in RNNs. Current hidden state h_t depends on previous state h_{t-1} & current input x_t, & is calculated using following relations:

1. 1. State Update. $h_t = f(h_{t-1}, x_t)$ where h_t: current state, h_{t-1}: previous state, x_t: input at current time step.
2. 2. Activation Function Application. $h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t)$ where W_{hh}: weight matrix for recurrent neuron, W_{xh}: weight matrix for input neuron.
3. 3. Output Calculation. $y_t = W_{hy} \cdot h_t$ where y_t: output & W_{hy}: weight at output layer.

These parameters are updated using backpropagation. However, since RNN works on sequential data here use an updated backpropagation which is known as *backpropagation through time*.

Backpropagation through time (BPTT) in RNNs. In a Recurrent Neural Network (RNN), data flows sequentially, where each time step's output depends on previous time step. This ordered data structure necessitates applying backpropagation across all hidden states, or time steps, in sequence. This unique approach is called *Backpropagation Through Time (BPTT)*, essential for updating network parameters that rely on temporal dependencies.

BPTT Process in RNNs. SKIPPED MATH.

Training Process in RNNs. Training RNNs involves feeding input data through multiple time steps, capturing dependencies across these steps, & updating model through backpropagation.

Steps in RNN training include:

1. *Input at Each Time Step*: A single time step of input sequence is provided to network.
2. *Calculate Hidden State*: Using current input & prev hidden state, network calculates current hidden state h_t.
3. *State Transition*: Current hidden state h_t then becomes h_{t-1} for next time step.
4. *Sequential Processing*: This process continues across all time steps to accumulate information from prev states.
5. *Output Generation & Error Calculation*: Final hidden state is used to compute network's output, which is then compared to actual target output to generate an error.
6. *Backpropagation Through Time (BPTT)*: This error is backpropagated through each time step to update weights & train RNN.

6.1.7 Implementing a Text Generator Using RNNs

Create a character-based text generator using RNN in TensorFlow & Keras. Implement an RNN that learns patterns from a text sequence to generate new text character-by-character.

- Step 1: Import Necessary Libraries. Start by importing essential libraries for data handling & building neural network.

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense
```

- *Step 2: Define Input Text & Prepare Character Set.* Define input text & identify unique characters in text, which encode for model.

```
text = "This is GeeksforGeeks a software training institute"
chars = sorted(list(set(text)))
char_to_index = {char: i for i, char in enumerate(chars)}
index_to_char = {i: char for i, char in enumerate(chars)}
```

- *Step 3: Create Sequences & Labels.* To train RNN, need sequences of fixed length `seq_length` & character following each sequence as label.

```
seq_length = 3
sequences = []
labels = []

for i in range(len(text) - seq_length):
    seq = text[i:i + seq_length]
```

```
        label = text[i + seq_length]
        sequences.append([char_to_index[char] for char in seq])
        labels.append(char_to_index[label])

    X = np.array(sequences)
    y = np.array(labels)
```

- *Step 4: Convert Sequences & Labels to 1-Hot Encoding.* For training, convert `X, y` into 1-hot encoded tensors.

```
X_one_hot = tf.one_hot(X, len(chars))
y_one_hot = tf.one_hot(y, len(chars))
```

- *Step 5: Build RNN Model.* Create a simple RNN model with a hidden layer of 50 units & a Dense output layer with softmax activation.

```
model = Sequential()
model.add(SimpleRNN(50, input_shape=(seq_length, len(chars)), activation='relu'))
model.add(Dense(len(chars), activation='softmax'))
```

- *Step 6: Compile & Train Model.* Compile model using categorical crossentropy loss & train it for 100 epochs.

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(X_one_hot, y_one_hot, epochs=100)
```

- *Step 7: Generate New Text Using Trained Model.* After training, use a starting sequence to generate new text character-by-character.

```
start_seq = "This is G"
generated_text = start_seq

for i in range(50):
x = np.array([[char_to_index[char] for char in generated_text[-seq_length:]]])
x_one_hot = tf.one_hot(x, len(chars))
prediction = model.predict(x_one_hot)
next_index = np.argmax(prediction)
next_char = index_to_char[next_index]
generated_text += next_char

print("Generated Text:")
print(generated_text)
```

Output:

```
Generated Text: This is Geeks a software training instituteais is is is is
```

NQBH's output: ***WHY?***

```
Generated Text:
This is GeeksforGeeksforGeeksforGeeksforGeeksforGeeksforGee
```

Advantages of RNNs.

- *Sequential Memory*: RNNs retain information from previous inputs, making them ideal for time-series predictions where past data is crucial. This capability is often called *Long Short-Term Memory* (LSTM).
- *Enhanced Pixel Neighborhoods*: RNNs can be combined with convolutional layers to capture extended pixel neighborhoods, improving performance in image & video data processing.

Limitations of RNNs. While RNNs excel at handling sequential data, they face 2 main training challenges, i.e., vanishing gradient & exploding gradient problem:

1. *Vanishing Gradient*: During backpropagation, gradients diminish as they pass through each time step, leading to minimal weight updates. This limits RNN's ability to learn long-term dependencies, which is crucial for tasks like language translation.
2. *Exploding Gradient*: Sometimes, gradients grow uncontrollably, causing excessively large weight updates that destabilize training. Gradient clipping is a common technique to manage this issue.

These challenges can hinder performance (cản trở hiệu suất) of standard RNNs on complex, long-sequence tasks.

Applications of RNNs. RNNs are used in various applications where data is sequential or time-based:

- Time-Series Prediction: RNNs excel in forecasting tasks, e.g. stock market predictions & weather forecasting.
- Natural Language Processing (NLP): RNNs are fundamental in NLP tasks like language modeling, sentiment analysis, & machine translation.
- *Speech Recognition*: RNNs capture temporal patterns in speech data, aiding in speech-to-text & other audio-related applications.
- *Image & Video Processing*: When combined with convolutional layers, RNNs help analyze video sequences, facial expressions, & gesture recognition.

FAQs on RNNs.

- Which type of problem can be solved by RNN? Modeling time-dependent & sequential data problems, like text generation, machine translation, & stock market prediction, is possible with RNNs. Nevertheless, will discover: gradient problem makes RNN difficult to train. Vanishing gradients issue affects RNNs.
- What are types of RNN? 1 to 1, 1 to many, many to 1, many to many.
- What are differences between RNN & CNN? Key distinctions between CNNs & RNNs: CNNs are frequently employed in solution of problems involving spatial data, like images. Text & video data that is temporally & sequentially organized is better analyzed by RNNs. RNNs & CNNs are not designed alike.

" – Geeksforgeeks/introduction to recurrent neural networks

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):

$$r_i = e_i - \sum_{j=1}^{n} k_{ij}[r_j - r_{ij}]^+, \tag{453}$$

where $[w]^+ := \max\{w, 0\}$. 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:

$$r_i(t) = e_i(t) - \sum_{j=1}^{n} k_{ij}\left[\frac{1}{\tau}\int_0^t e^{-\frac{t-s}{\tau}} r_j(s)\,\mathrm{d}s - r_{ij}\right]^+, \tag{454}$$

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:

$$\frac{d}{dt}x_i = -A_i x_i + \sum_{j=1}^{n} f_j(x_j)B_{ji}z_{ji}^{(+)} - \sum_{j=1}^{n} g_j(x_j)C_{ji}z_{ji}^{(-)} + I_i. \tag{455}$$

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:

$$\frac{d}{dt}x_i = -A_i x_i + \sum_{j=1}^{n} f_j(x_j)z_{ji} + I_i. \tag{456}$$

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):

$$\frac{d}{dt}x_i = -A_i x_i + (B - x_i)\left[\sum_{j=1}^{n} f_j(x_j)C_{ji}z_{ji}^{(+)} + I_i\right] - (D + x_i)\left[\sum_{j=1}^{n} g_j(x_j)E_{ji}z_{ji}^{(-)} + J_i\right]. \tag{457}$$

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:

$$\frac{dx_i}{dt} = -Ax_i + (B - Cx_i)\left[\sum_{k=1}^{n} f_k(x_k)D_{ki}y_{ki}z_{ki} + I_i\right] - (E + Fx_i)\left[\sum_{k=1}^{n} g_k(x_k)G_{ki}Y_{ki}Z_{ki} + J_i\right]. \tag{458}$$

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 $\sigma_h = \tanh, \sigma_y = \text{Identity}$. 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:

  $$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h), \ y_t = \sigma_y(W_y h_t + b_y). \qquad \text{(Elman nw)}$$

  Jordan network:

  $$h_t = \sigma_h(W_h x_t + U_h s_t + b_h), \ y_t = \sigma_y(W_y h_t + b_y), \ s_t = \sigma_s(W_{s,s} s_{t-1} + S_{s,y} y_{t-1} + b_s). \qquad \text{(Jordan nw)}$$

  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 $\cdot$ 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/supervised learning

"In supervised learning, training data is labeled with expected answers, while in unsupervised learning, model identifies patterns or structures in unlabeled data. In ML, *supervised learning (SL)* is a paradigm where a statistical model is trained using input objects (e.g., a vector of predictor variables) & desired output values (also known as a *supervisory signal*), which are often human-made labels. Training process builds a function that maps new data to expected output values. An optimal scenario will allow for algorithm to accurately determine output values for unseen instances. This requires learning algorithm to generalize from training data to unseen situations in a "reasonable" way (see inductive bias). This statistical quality of an algorithm is measured via a generalization error.

8.3.1 Steps to follow

To solve a given problem of supervised learning, following steps must be performed:

1. Determine type of training samples. Before doing anything else, user should decide what kind of data is to be used as a training set. In case of handwriting analysis, e.g., this might be a single handwritten character, an entire handwritten word, an entire sentence of handwriting, or a full paragraph of handwriting.

2. Gather a training set. Training set needs to be representative of real-world use of function. Thus, a set of input objects is gathered together with corresponding outputs, either from human experts or from measurements.

3. Determine input feature representation of learned function. Accuracy of learned function depends strongly on how input object is represented. Typically, input object is transformed into a feature vector, which contains a number of features that are descriptive of object. Number of features should not be too large, because of curse of dimensionality; but should contain enough information to accurately predict output.

4. Determine structure of learned function & corresponding learning algorithm. E.g., one may choose to use support-vector machines or decision trees.

5. Complete design. Run learning algorithm on gathered training set. Some supervised learning algorithms require use to determine certain control parameters. These parameters may be adjusted by optimizing performance on a subset (called a validation set) of training set, or via cross-validation.

6. Evaluate accuracy of learned function. After parameter adjustment & learning, performance of resulting function should be measured on a test case that is separate from training set.

8.3.2 Algorithm choice

A wide range of supervised learning algorithms are available, each with its strengths & weaknesses. There is no single learning algorithm that works best on all supervised learning problems (see no free lunch theorem).

There are 4 major issues to consider in supervised learning:

- Bias-variance tradeoff.
- Function complexity & amount of training data.
- Dimensionality of input space.
- Noise in output values.
- Other factors to consider.
- Algorithms.

8.3.3 How supervised learning algorithms work

- Empirical risk minimization.
- Structural risk minimization.

8.3.4 Generative training

8.3.5 Generalizations

8.3.6 Approaches & algorithms

8.3.7 Applications

8.3.8 General issues

" – Wikipedia/supervised learning

8.4 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.

8.4.1 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.

8.4.2 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.

8.4.3 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.

8.4.4 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)

8.5 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.

8.5.1 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.

8.5.2 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.

8.5.3 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.

8.5.4 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.

8.5.5 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.

8.5.6 Modular

- Committee of machines.
- Associative.

8.5.7 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.

8.5.8 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.

8.5.9 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.

8.5.10 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.

8.5.11 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, denoising 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:

$$p(v,\psi) = \frac{1}{Z}\sum_h \exp\left(\sum_{ij} W_{ij}^{(1)} v_i h_j^1 + \sum_{jl} W_{jl}^{(2)} h_j^1 h_l^2 + \sum_{lm} W_{lm}^{(3)} h_l^2 h_m^3\right), \tag{459}$$

  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:

$$p(v,h^1,h^2|h^3) = \frac{1}{Z(\psi,h^3)}\sum_h \exp\left(\sum_{ij} W_{ij}^{(1)} v_i h_j^1 + \sum_{jl} W_{jl}^{(2)} h_j^1 h_l^2 + \sum_{lm} W_{lm}^{(3)} h_l^2 h_m^3\right). \tag{460}$$

- 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 & $m_l \in \{1, \ldots, n_l\}$, 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.

[BHH06] Francis R. Bach, David Heckerman, and Eric Horvitz. "Considering cost asymmetry in learning classifiers". In: *J. Mach. Learn. Res.* 7 (2006), pp. 1713–1741. ISSN: 1532-4435.

[BJP20] Jean-Pierre Briot, Gaëtan Jadjeres, and François-David Pachet Pachet. *Deep Learning Techniques for Music Generation*. Computational Synthesis & Creative Systems. Springer, 2020, p. 284.

[Cho21] François Chollet. *Deep Learning with Python*. 2nd edition. Manning, 2021, p. 478.

[DFO23] Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. 1st edition. Cambridge University Press, 2023, p. 398.

[DG24] Shlomo Dubnov and Ross Greer. *Deep & Shallow: Machine Learning in Music & Audio*. 1st edition. Chapman & Hall/CRC Machine Learning & Pattern Recognition. CRC Press, 2024, p. 328.

[HJE18] Jiequn Han, Arnulf Jentzen, and Weinan E. "Solving high-dimensional partial differential equations using deep learning". In: *Proc. Natl. Acad. Sci. USA* 115.34 (2018), pp. 8505–8510. ISSN: 0027-8424. DOI: 10.1073/pnas.1718942115. URL: https://doi.org/10.1073/pnas.1718942115.

[Kut23] Gitta Kutyniok. "The mathematics of artificial intelligence". In: *ICM—International Congress of Mathematicians. Vol. 7. Sections 15–20*. EMS Press, Berlin, [2023] ©2023, pp. 5118–5139.

[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.

[LeV07] Randall J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. Steady-state and time-dependent problems. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2007, pp. xvi+341. ISBN: 978-0-898716-29-0. DOI: 10.1137/1.9780898717839. URL: https://doi.org/10.1137/1.9780898717839.

[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.

[RLM22] Sebastian Raschka, Yuxi (Hayden) Liu, and Vahid Mirjalili. *Machine Learning with PyTorch & Scikit-Learn: Develop Machine Learning & Deep Learning Models with Python*. 1st edition. Packt Publishing, 2022, p. 741.

[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.