Deep Learning – Học Sâu

Nguyễn Quản Bá Hồng*

Ngày 13 tháng 8 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:

- *Deep Learning – Học Sâu.*
  PDF: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/deep_learning/NQBH_deep_learning.pdf.
  TEX: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/deep_learning/NQBH_deep_learning.tex.
- .
  PDF: URL: .pdf.
  TEX: URL: .tex.

Mục lục

1 Basic Deep Learning

1.1 [BB24]. Christopher M. Bishop, Hugh Bishop. Deep Learning: Foundations & Concepts

[130 Amazon ratings]

*A scientist- & creative artist wannabe, a mathematics & computer science lecturer of Department of Artificial Intelligence & Data Science (AIDS), School of Technology (SOT), UMT Trường Đại học Quản lý & Công nghệ TP.HCM, Hồ Chí Minh City, Việt Nam.
E-mail: nguyenquanbahong@gmail.com & hong.nguyenquanba@umt.edu.vn. Website: https://nqbh.github.io/. GitHub: https://github.com/NQBH.

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 $\boxed{\text{almost entirely non-Bayesian}}$.

  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

$$y(x, \mathbf{w}) = \sum_{i=0}^{M} w_i x^i = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M, \tag{1}$$

  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

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

  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

$$E_{\rm RMS} = \sqrt{\frac{1}{N}\sum_{n=1}^N (y(x_n,\mathbf{w}) - t_n)^2} \tag{3}$$

in which division by N allows us to compare different sizes of data sets on an equal footing, & square root ensures: $E_{\rm RMS}$ 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 $3 \le M \le 8$ 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)

$$\widetilde{E}(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^N (y(x_n,\mathbf{w}) - t_n)^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2, \tag{4}$$

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

Fig. 1.9: Plots of $M = 9$ polynomials fitted to data set shown in Fig. 1.4 using regularized error function (1.4) for 2 values of regularization parameter λ corresponding to $\ln\lambda = -18$ & $\ln\lambda = 0$. Case of no regularizer, i.e., $\lambda = 0$, corresponding to $\ln\lambda = -\inf$, is shown at bottom right of Fig. 1.6. shows results of fitting polynomial of order $M = 9$ to same data set as before but now using regularized error function given by (1.4). See: for a value of $\ln\lambda = -18$, overfitting has been suppressed & now obtain a much closer representation of underlying function $\sin 2\pi x$. If, however, use too large a value for λ then again obtain a poor fit, as shown in Fig. 1.9 for $\ln\lambda = 0$. Corresponding coefficients from fitted polynomials

are given in Table 1.2: Table of coefficients $\mathbf{w}^\star$ for $M = 9$ polynomials with various values for regularization parameter λ. Note $\ln \lambda = -\inf$ corresponds to a model with no regularization, i.e., to graph at bottom right in Fig. 1.6. See: as value of λ increases, magnitude of a typical coefficient gets smaller., showing: regularization has desired effect of reducing magnitude of coefficients.

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

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

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

1.2 [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 [1 of 1st documented case of *stochastic music*, long before computers, is Musikalisches Wurfelspiel (Dice Music), attributed to Wolfgang Amadeus Mozart. It was designed for using dice to generate music by concatenating randomly selected predefined music segments composed in a given style (Austrian waltz in a given key).] [208], in his composition named "Atrées" in 1962. Idea involved using computer fast computations to calculate various possibilities from a set of probabilities designed by computer in order to generate samples of musical pieces to be selected. In another approach following initial direction of "The Illiact Suite", grammars & rules were used to specify style of a given corpus or more generally tonal music theory. E.g.: generation in 1980s by Ebcioğlu's composition software named CHORAL of a 4-part chorale in style of Johann Sebastian Bach, according to > 350 handcrafted rules [41]. In late 1980s David Cope's system named Experiments in Musical Intelligence (EMI) extended that approach with capacity to learn from a corpus of scores of a composer to create its own grammar & database of rules [26].

    Since then, computer music has continued developing for general public, if consider, e.g., GarageBand music composition & production application for Apple platforms (computers, tablets, & cellphones), as an offspring of initial Cubase sequencer software, released by Steinberg in 1989.

    For more details about history & principles of computer music in general, see, e.g., book by Roads [159]. For more details about history & principles of algorithmic composition, see, e.g., [127] & books by Cope [26] or Dean & McLean [32].
  - 1.1.2. Autonomy vs. Assistance. When talking about computer-based music generation, there is actually some ambiguity about whether objective is:
    - to design & construct *autonomous* music-making systems – 2 recent examples being DL based Amper & Jukedeck systems/companies aimed at creation of original music for commercials & documentary; or
    - to design & construct computer-based environment to *assist* human musicians (composers, arrangers, producers, etc.) – 2 examples being FlowComposer environment developed at Sony CSL-Paris [152] (introduced in Sect. 6.11.4) & OpenMusic environment developed at IRCAM [3].

    Quest for autonomous music-making systems may be an interesting perspective for exploring process of composition [As Richard Feynman coined it: "What I cannot create, I do not understand."] & it also serves as an evaluation method. An example of a musical Turing test [Initially codified (Ban đầu được mã hóa) in 1950 by Alan Turing & named by him "imitation game" [190], "Turing test" is a test of ability for a machine to exhibit intelligent behavior equivalent to (& more precisely, indistinguishable from) behavior of a human. In his imaginary experimental setting, Turing proposed test to be a natural language conversation between a human (evaluator) & a hidden actor (another human or a machine). If evaluator cannot reliably tell machine from human, machine is said to have passed test.] will be introduced in Sect. 6.14.2. It consists in presenting to various members of public (from beginners to experts) chorales composed by J. S. Bach or generated by a DL system & played by human musicians [This is to avoid bias (synthetic flavor) of a computer rendered generated music.]. DL techniques turn out to be very efficient at succeeding in such tests, due to their capacity to learn musical style from a given corpus & to generate new music that fits into this style. I.e., consider: such a test is more a means than an end.

    A broader perspective is in assisting human musicians during various steps of music creation: composition, arranging, orchestration, production, etc. Indeed, to compose or to improvise [Improvisation is a form of real time composition. (Ngẫu hứng là 1 hình thức sáng tác thời gian thực.)], a musician rarely creates new music from scratch. S/he reuses & adapts, consciously or unconsciously, features from various music that s/he already knows or has heard, while following some principles & guidelines, e.g. theories about harmony & scales. A computer-based musician assistant may act during different stages of composition, to initiate, suggest, provoke &/or complement inspiration of human composer.

    I.e., majority of current DL based systems for generating music are still focused on autonomous generation, although more & more systems are addressing issue of human-level control & interaction.
  - 1.1.3. Symbolic vs. Sub-Symbolic AI. AI is often divided into 2 main streams [With some precaution, as this division is not that strict.]:

- · symbolic AI – dealing with high-level symbolic representations (e.g., chords, harmony . . .) & processes (harmonization, analysis, . . .)
- · sub-symbolic AI – dealing with low-level representations (e.g., sound, timbre . . .) & processes (pitch recognition, classification . . .).

Examples of symbolic models used for music are rule-based systems or grammars to represent harmony. Examples of sub-symbolic models used for music are ML algorithms for automatically learning musical styles from a corpus of musical pieces. These models can then be used in a generative & interactive manner, to help musicians in creating new music, by taking advantage of this added "intelligent" memory (associative, inductive, & generative) to suggest proposals, sketches, extrapolations, mappings, etc. This is now feasible because of growing availability of music in various forms, e.g., sound, scores, & MIDI files, which can be automatically processed by computers.

A recent example of an integrated music composition environment is FlowComposer [152], introduced in Sect. 6.11.4. It offers various symbolic & sub-symbolic techniques, e.g., Markov chains for modeling style, a constraint solving module for expressing constraints, a rule-based module top produce harmonic analysis; & an audio mapping module to produce rendering. Another example of an integrated music composition environment is OpenMusic [3].

However, a deeper integration of sub-symbolic techniques, e.g. DL, with symbolic techniques, e.g. constraints & reasoning, is still an open issue [General objective of integrating sub-symbolic & symbolic levels into a complete AI system is among "Holyy Grails" of AI.], although some partial integrations in restricted contexts already exist (see, e.g., Markov constraints in [148,7] & an example of use for FlowComposer in Sect. 6.11.4).

∗ 1.1.4. DL. Motivation for using DL (& more generally ML techniques) to generate musical content is its *generality*. As opposed to handcrafted models, e.g. grammar-based [176] or rule-based music generation systems [41], a ML-based generation system can be agnostic (người theo thuyết bất khả tri), as it learns a model from an arbitrary corpus of music. As a result, same system may be used for various musical genres.

Therefore, as more large scale musical datasets are made available, a ML-based generation system will be able to automatically learn a musical style from a corpus & to generate new musical content. As stated by Fiebrink & Caramiaux [51], some benefits:

- · it can make creation feasible when desired application is too complex to be described by analytical formulations or manual brute force design
- · learning algorithms are often less brittle (giòn, mỏng manh) than manually designed rule sets & learned rules are more likely to generalize accuracy to new contexts in which inputs may change.

Moreover, as opposed to structured representations like rules & grammars, DL is good at processing raw unstructured data, from which its hierarchy of layers will extract higher level representations adapted to task.

∗ 1.1.5. Present & Future. Research domain in DL-based music generation has turned hot recently, building on initial work using ANNs to generate music (e.g., pioneering experiments by Todd in 1989 [189] & CONCERT system developed by Mozer in 1994 [138]), while creating an active stream of new ideas & challenges made possible thanks to progress of DL. Note: growing interest by some private big actors of digital media in computer-aided generation of artistic content, with creation by Google in Jun 2016 of Magenta research project [47] & creation by Spotify in Sep 2017 of Creator Technology Research Lab (CTRL) [175]. This is likely to contribute to blurring line between music creation & music consumption through personalization of musical content [2].

○ 1.2. This Book. lack (to our knowledge) of a comprehensive survey & analysis of this active research domain motivated writing of this book, build in a *bottom-up* way from analysis of numerous recent research works. Objective: provide a comprehensive description of issues & techniques for using DL to generate music, illustrated through analysis of various architectures, systems & experiments presented in literature. Also propose a conceptual framework & typology aimed at a better understanding of design decisions for current as well as future systems.

∗ 1.2.1. Other Books & Sources. To our knowledge, there are only a few partial attempts at analyzing use of DL for generating music. In [14], a very preliminary version of this work, Briot et al. proposed a 1st survey of various systems through a multicriteria analysis (considering as dimensions objective, representation, architecture, & strategy). Have extended & consolidated this study by integrating as an additional dimension challenge (after having analyzed it in [15]).

In [64], Graves presented an analysis focusing on RNNs & text generation. In [89], Humphrey et al. presented another analysis, sharing some issues about music representation (Sect. 4) but dedicated to music information retrieval (MIR) tasks, e.g. chord recognition, genre recognition & mood estimation. On MIR applications of DL, see also recent tutorial paper by Choi et al. [20].

One could also consult proceedings of some recently created international workshops on topic, e.g.

- · Workshop on Constructive Machine Learning (CML 2016), held during 30th Annual Conference on Neural Information Processing Systems (NIPS 2016) [28];
- · Workshop on Deep Learning for Music (DLM), held during the International Joint Conference on Neural Networks (IJCNN 2017) [74], followed by a special journal issue [75];
- · on deep challenge of *creativity*, related Series of International Conferences on Computational Creativity (ICCC) [185].

For a more general survey of computer-based techniques to generate music, reader can refer to general books e.g.

- · Roads book about computer music [159];
- · Cope's [26], Dean & McLean's [32] and/or Nierhaus' books [143] about algo- rithmic composition;
- · a recent survey about AI methods in algorithmic composition [50]

· Cope's book about models of musical creativity [27].

About ML in general, some examples of textbooks:

· textbook by Mitchell [131]

· a nice introduction & summary by Domingos [38]

· a recent, complete & comprehensive book about DL by Goodfellow et al. [62].

∗ 1.2.2. Other Models. Have to remember: there are various other models & techniques for using computers to generate music, e.g. rules, grammars, automata, Markov models & graphical models. These models are either *manually* defined by experts or are automatically *learnt* from examples by using various ML techniques. They will not be addressed in this book as concerned here with DL techniques. However, in following sect make a quick comparison of DL & Markov models.

∗ 1.2.3. DL vs. Markov Models. DL models are not only models able to learn musical style from examples. Markov chain models are also widely used, see, e.g., [145]. A quick comparison (inspired by analysis of Mozer in [138] [Note: he made his analysis in 1994, long before DL wave.]) of pros (+) & cons (-) of deep neural network models & Markov chain models is as follows:

+ Markov models are conceptually simple.

+ Markov models have a simple implementation & a simple learning algorithm, as model is a transition probability table [Statistics are collected from dataset of examples in order to compute probabilities.].

- Neural network models are conceptually simple but optimized implementations of current deep network architectures may be complex & need a lot of tuning.

- Order 1 Markov models (i.e., considering only prev state) do not capture long-term temporal structures.

- Order n Markov models (considering n prev states) are possible but require an explosive training set size [See discussion in [138, p. 249].] & can lead to plagiarism [By recopying too long sequences from corpus. Some promising solution: consider a variable order Markov model & to constrain generation (through min order & max order constraints) on some sweet spot between junk & plagiarim [151].].

+ Neural networks can capture various types of relations, contexts, & regularities.

+ Deep networks can learn long-term & high-order dependencies.

+ Markov models can learn from a few examples.

- Neural networks need a lot of examples in order to be able to learn well.

- Markov models do not generalize very well.

+ Neural networks generalize better through use of distributed representations [81].

+ Markov models are operational models (automata) on which some control on generation could be attached [Examples: Markov constraints [148] & factor graphs [147].].

- Deep networks are generative models with a distributed representation & therefore with no direct control to be attached [This issue as well as some possible solutions will be discussed in Sect. 6.10.1.].

As DL implementations are now mature & a large number of examples are available, DL-based models are in high demand for their characteristics. I.e., other models (e.g. Markov chains, graphical models, etc.) are still useful & used & choice of a model & its tuning depends on characteristics of problem.

∗ 1.2.4. Requisites & Roadmap. This book does not require prior knowledge about DL & neural networks nor music.

· *Chap. 1: Introduction* introduces purpose & rationale (lý lẽ) of book.

· *Chap. 2: Method* introduces method of analysis (conceptual framework) & 5 dimensions at its basis (objective, representation, architecture, challenge, & strategy), dimensions discussed within next 4 chaps.

· *Chap. 3: Objective* concerns different types of musical content that we want to generate (e.g. a melody or an accompaniment to an existing melody) [Our proposed typology of possible objectives will turn out to be useful for our analysis because different objectives can lead to different architectures & strategies.], as well as their expected use (by a human &/or a machine).

· *Chap. 4: Representation* provides an analysis of different types of representation & techniques for encoding musical content (e.g. notes, durations or chords) for a DL architecture. This chap may be skipped by a reader already expert in computer music, although some of encoding strategies are specific to neural network & DL architectures.

· Chap. 5: Architecture summarizes most common DL architectures (e.g. feedforward, recurrent or autoencoder) used for generation of music. This includes a short reminder of very basics of a simple neural network. This chap may be skipped by a reader already expert in ANNs & DL architectures.

· Chap. 6: Challenge & Strategy provides an analysis of various challenges that occur when applying DL techniques to music generation, as well as various strategies for addressing them. Ground our study in analysis of various systems & experiments surveyed from literature. This chap is core of book.

· Chap. 7: Analysis summarizes survey & analysis conducted in Chap. 6 through some tables as a way to identify design decisions & their interrelations for different system surveyed [& hopefully also for future ones. If draw analogy (at some meta-level) with expected ability for a model learnt from a corpus by a machine to be able to generalize to future examples (Sect. 5.5.9), hope: conceptual framework presented in this book, (manually) inducted from a corpus of scientific & technical literature about DL-based music generation systems, will also be able to help in design & understanding of future systems.].

- · Chap. 8: Discussion & Conclusion revisits some of open issue that were touched in during analysis of challenges & strategies presented in Chap. 6, before concluding this book.

  Supplementary material is provided at the following companion web site: www.briot.info/dlt4mg/.

- ∗ 1.2.5. Limits. This book does not intend to be a general introduction to DL – a recent & broad spectrum book on this topic is [62]. Do not intend to get into all technical details of implementation, like engineering & tuning, as well as theory [E.g., not develop probability theory & information theory frameworks for formalizing & interpreting behavior of neural networks & DL. However, Sect. 5.5.6 will introduce intuition behind notions of entropy & cross-entropy, used for measuring progress made during learning.], as wish to focus on conceptual level, whilst providing a sufficient degree of precision. Also, although having a clear pedagogical objective, do not provide some end-to-end tutorial with all steps & details on how to implement & tune a complete DL-based music generation system.

  Last, as this book is about a very active domain & as our survey & analysis is based on existing systems, our analysis is obviously not exhaustive. Have tried to select most representative proposals & experiments, while new proposals are being presented at time of our writing. Therefore, encourage readers & colleagues to provide any feedback & suggestions for improving this survey & analysis which is a still ongoing project.

- 2. Method. In our analysis, consider 5 main *dimensions* to characterize different ways of applying DL techniques to generate musical content. This typology is aimed at helping analysis of various perspectives (& elements) leading to design of different DL-based music generation systems [In this book, *systems* refers to various proposals (architectures, systems, & experiments) about DL-based music generation that have surveyed from literature.]

  - ○ 2.1. Dimensions. 5 dimensions that have consider are as follows.
    - ∗ 2.1.1. Objective. *Objective* [Could have used term *task* in place of *objective*. However, as task is a relatively well-defined & common term in ML community (see Sect. 1 & [62, Chap. 5]), preferred an alternative term.] consists in:
      - · Musical *nature* of content to be generated. E.g.: a melody, a polyphony or an accompaniment
      - · *Destination & use* of content generated. E.g.: a musical score to be performed by some human musician(s) or an audio file to be played.
    - ∗ 2.1.2. Representation. *Representation* is nature & format of information (data) used to *train* & to *generate* musical content. E.g.: signal, transformed signal (e.g., a spectrum, via a Fourier transform), piano roll, MIDI or text.
    - ∗ 2.1.3. Architecture. *Architecture* is nature of assemblage of processing *units* (artificial neurons) & their *connections*. E.g.: a feedforward architecture, a recurrent architecture, an autoencoder architecture & generative adversarial networks.
      – *Kiến trúc* là bản chất của tập hợp các đơn vị xử lý (nơ-ron nhân tạo) & *kết nối* của chúng. E.g.: kiến trúc truyền thẳng, kiến trúc tuần hoàn, kiến trúc tự mã hóa & mạng đối nghịch tạo sinh.
    - ∗ 2.1.4. Challenge. A *challenge* is 1 of qualities (requirements) that may be desired for music generation. E.g.: content variability, interactivity, & originality.
    - ∗ 2.1.5. Strategy. *Strategy* represents way architecture will process representations in order to *generate* [Note: consider here strategy relating to *generation phase* & not strategy relating to training phase, as they could be different.] objective while matching desired requirements. E.g.: single-step feedforward, iterative feedforward, decoder feedforward, sampling, & input manipulation.
  - ○ 2.2. Discussion. Note: these 5 dimensions are not orthogonal. Choice of representation is partially determined by objective & it also constraints input & output (interfaces) of architecture. A given type of architecture also usually leads to a default strategy of use, while new strategies may be designed in order to target specific challenges.

    Exploration of these 5 different dimensions & of their interplay is actually at core of our analysis [Remember: our proposed typology has been constructed in a *bottom-up* manner from survey & analysis of numerous systems retrieved from literature, most of them being very recent.]. Each of 1st 3 dimensions (objective, representation, & architecture) will be analyzed with its associated typology in a specific chap, with various illustrative examples & discussion. Challenge & strategy dimensions will be jointly analyzed within same chap (Chap. 6) in order to jointly illustrate potential issues (challenges) & possible solutions (strategies). Same strategy may relate to > 1 challenge & vice versa.

    Last, do not expect our proposed conceptual framework (& its associated 5 dimensions & related typologies (các loại hình)) to be a final result, but rather a 1st step towards a better understanding of design decisions & challenges for DL-based music generation. I.e., it is likely to be further amended & refined, but hope: it could help bootstrap what believe to be a necessary comprehensive study (nó có thể giúp khởi động những gì được cho là 1 nghiên cứu toàn diện cần thiết).

- 3. Objective. 1st dimension, *objective*, is nature of musical content to be generated.

  - ○ 3.1. Facets. May consider 5 main *facets* of an objective:
    - ∗ *Type*. Musical nature of generated content. E.g.: a melody, a polyphony or an accompaniment.
    - ∗ *Destination*. Entity aimed at using (processing) generated content. E.g.: a human musician, a software or an audio system.
    - ∗ *Use*: Way destination entity will process generated content. E.g.: playing an audio file or performing a music score.
    - ∗ *Mode*: way *generation* will be conducted, i.e., with some human intervention (*interaction*) or without any intervention (*automation*).
    - ∗ *Style*: Musical style of content to be generated. E.g.: JOHANN SEBASTIAN BACH chorales, WOLFGANG AMADEUS MOZART sonatas, COLE PORTER songs or WAYNE SHORTER music. Style will actually be set though choice of dataset of musical examples (corpus) used as training examples.

- 3.1.1. Type. Main examples of musical types:
  1. *Single-voice monophonic melody*, abbr. *Melody*. It is a sequence of notes for a single instrument or vocal, with *at most* 1 note at same time. E.g.: music produced by a monophonic instrument like a flute [Although there are non-standard techniques to produce > 1 note, simplest one being to sing simultaneously as playing. There are also non-standard diphonic techniques for voice.].
  2. *Single-voice polyphony* (also named *Single-track polyphony*), abbr. as *Polyphony*. It is a sequence of notes for a single instrument, where > 1 note can be played at same time. E.g.: music produced by a polyphonic instrument e.g. a piano or guitar.
  3. *Multivoice polyphony* (also named *Multitrack polyphony*), abbr. *Multivoice* or *Multitrack*. It is a set of multiple *voices/tracks*, which is intended for > 1 voice or instrument. E.g.: a chorale with soprano, alto, tenor, & bass voices or a jazz trio with piano, bass, & drums.
  4. *Accompaniment* to a given melody. E.g.:
     (a) *Counterpoint*, composed of 1 or more melodies (voices), or
     (b) *Chord progression*, which provides some associated *harmony*.
  5. *Association of a melody with a chord progression*: E.g.: what is named a *lead sheet* [Fig. 4.13: Lead sheet of "Very Late" (PACHET & D'INVERNO.) in Chap. 4 Representation will show an example of a lead sheet.] & is common in jazz. It may also include *lyrics* [Note: lyrics could be generated too. Although this target is beyond scope of this book, see in Sect. 4.7.3: in some systems, music is encoded as a text. Thus, a similar technique could be applied to lyric generation.].

  Note: *type* facet is actually most important facet, as it captures musical nature of objective for content generation. In this book, will frequently identify an objective according to its *type*, e.g., a melody, as a matter of simplification. Next 3 facets – *destination, use, & mode* – will turn out important when regarding dimension of *interaction* of human user(s) with process of content generation.
- 3.1.2. Destination & Use. Main examples of destination & use are as follows:
  1. *Audio system*: which will *play* generated content, as in case of generation of an audio file.
  2. *Sequencer software*: which will *process* generated content, as in case of generation of a MIDI file.
  3. *Human(s)*: who will perform & *interpret* generated content, as in case of generation of a music score.
- 3.1.3. Mode. There are 2 main modes of music generation:
  1. *Autonomous & Automated*: Without any human intervention
  2. *Interactive* (to some degree): With some control interface for human user(s) to have some interactive control over process of generation.

  As DL for music generation is recent & basic neural network techniques are non-interactive, majority of systems that have analyzed are not yet very interactive [Some examples of interactive systems will be introduced in Sect. 6.15.]. Therefore, an important goal appears to be design of fully interactive support systems for musicians (for composing, analyzing, harmonizing, arranging, producing, mixing, etc.), as pioneered by FlowComposer prototype [152] introduced in Sect. 6.11.4.
- 3.1.4. Style. As stated previously, musical style of content to be generated will be governed by choice of dataset of musical examples that will be used as training examples. As discussed further in Sect. 4.12, see: choice of a dataset, notably properties like *coherence, coverage* (vs. *sparsity*) & *scope* (specialized vs. large breadth), is actually fundamental for good music generation.
  – Như đã nêu trước đó, phong cách âm nhạc của nội dung được tạo ra sẽ được điều chỉnh bởi sự lựa chọn tập dữ liệu các ví dụ âm nhạc sẽ được sử dụng làm ví dụ đào tạo. Như đã thảo luận thêm trong Phần 4.12, hãy xem: sự lựa chọn tập dữ liệu, đặc biệt là các thuộc tính như tính mạch lạc, độ phủ (so với độ thưa thớt) & phạm vi (chuyên biệt so với độ rộng lớn), thực sự là nền tảng cho việc tạo ra âm nhạc tốt.

- 4. Representation. 2nd dimension of our analysis, *representation*, is about way musical content is represented. Choice of representation & its encoding is tightly connected to configuration of input & output of architecture, i.e., number of input & output variables as well as their corresponding types.

  See: although a DL architecture can automatically extract significant *features* from data, choice of representation may be significant for accuracy of learning & for quality of generated content.

  E.g., in case of an audio representation, could use a spectrum representation (computed by a Fourier transform) instead of a raw waveform representation. In case of a symbolic representation, could consider (as in most systems) enharmony (sự hòa hợp), i.e., A♯ ⇔ B♭ & C♭ ⇔ B, or instead preserve distinction in order to keep harmonic &/or voice leading meaning.

  - 4.1. Phases & Types of Data. Before getting into choices of representation for various data to be processed by a DL architecture, important to identify 2 main phases related to activity of a DL architecture: *training phase* & *generation phase*, as well as related 4 [There may be more types of data depending on complexity of architecture, which may include *intermediate* processing steps.] main types of data to be considered:
    - *Training phase*
      - *Training data*: Set of examples used for training DL system

    - *Validation data* (also [Actually, a difference could be made, explained in Sect. 5.5.9.] named *Test data*): set of examples used for testing DL system [Motivation is introduced in Sect. 5.5.9.]
  - *Generation phase.*
    - *Generation (input) data*: data that will be used as input for generation, e.g., a melody for which system will generate an accompaniment, or a note that will be 1st note of a generated melody
    - *Generated (output) data*: Data produced by generation, as specified by objective.

  Depending on objective [As stated in Sect. 3.1.1, identify an objective by its type as a matter of simplification.], these 4 types of data may be equal or different. E.g.:

  - in case of generation of a melody (e.g., in Sect. 6.6.12), both training data & generated data are melodies; whereas
  - in case of generation of a counterpoint accompaniment (e.g., in Sect. 6.2.2), generated data is a set of melodies.

- 4.2. Audio vs. Symbolic. A big divide in terms of choice of representation (both for input & output) is *audio vs. symbolic*. This also corresponds to divide between *continuous & discrete* variables. Their respective raw material is very different in nature, as are types of techniques for possible processing & transformation of initial representation [Initial representation may be transformed, through, e.g., data compression or extraction of higher-level representations, in order to improve learning &/or generation.]. They in fact correspond to different scientific & technical communities, namely *signal processing & knowledge representation.*

  However, actual processing of these 2 main types of representation by a DL architecture is basically *same* [Indeed, at level of processing by a deep network architecture, initial distinction between audio & symbolic representation boils down, as only *numerical* values & operations are considered.]. Therefore, actual audio & symbolic architectures for music generation may be pretty similar. E.g., WaveNet audio generation architecture (introduced in Sect. 6.10.3.2) has been transposed into MidiNet symbolic music generation architecture (in Sect. 6.10.3.3). This polymorphism (possibility of multiple representations leading to genericity) is an additional advantage of DL approach.

  I.e., focus in this book on *symbolic* representations & on DL techniques for generation of *symbolic* music. There are various reasons for this choice:

  - grand majority of current DL systems for music generation are symbolic
  - believe: essence of music (as opposed to sound [Without minimizing importance of orchestration & production.]) is in compositional process, which is exposed via symbolic representations (like musical scores or lead sheets) & is subject to analysis (e.g., harmonic analysis)
  - covering details & variety of techniques for processing & transforming audio representations (e.g., spectrum, cepstrum, MFCC [Mel-frequency cepstral coefficients (Hệ số cepstral tần số Mel).], etc.) would necessitate an additional book [An example entry point: recent review by Wyse of audio representations for deep convolutional networks [207].]
  - as stated previously, independently of considering audio or symbolic music generation, principles of DL architectures as well as encoding techniques used are actually pretty similar.

  Last, mention a recent DL-based architecture which combines audio & symbolic representations. In this proposal from Manzelli et al. [125], a symbolic representation is used as a conditioning input [Conditioning is introduced in Sect. 6.10.3.] in addition to audio representation main input, in order to better guide & structure generation of (audio) music (see more details in Sect. 6.10.3.2).

- 4.3. Audio. 1st type of representation of musical content is audio *signal*, either in its raw form (waveform) or transformed.
  - 4.3.1. Waveform. Most direct representation: raw audio signal: *waveform*. Visualization of a waveform is shown in Fig. 4.1: Example of a waveform. & another one with a finer grain resolution (độ phân giải hạt mịn hơn) is shown in Fig. 4.2: Example of a waveform with a fine grain resolution. Excerpt from a waveform visualization (sound of a guitar) by Michael Jancsy. In both figures, x axis represents time & y axis represents amplitude of signal.
    Advantage of using a waveform is in considering raw material untransformed, with its full initial resolution. Architectures that process raw signal are sometimes named *end-to-end* architectures [Term *end-to-end* emphasizes: a system learns all features from raw unprocessed data – without any pre-processing, transformation of representation, or extraction of features – to produce final output.]. Disadvantage is in computational load: low level raw signal is demanding in terms of both memory & processing.
  - 4.3.2. Transformed Representations. Using transformed representations of audio signal usually leads to data compression & higher-level information, but as noted previously, at cost of losing some information & introducing some bias.
  - 4.3.3. Spectrogram. A common transformed representation for audio is *spectrum*, obtained via a *Fourier transform* [Objective of Fourier transform (which could be continuous or discrete) is decomposition of an arbitrary signal into its elementary components (sinusoidal waveforms). As well as compressing information, its role is fundamental for musical purposes as it reveals *harmonic* components of signal.]. Fig. 4.3: Example of a spectrogram of spoken words "19th century". Reproduced from Aquegg's original image at https://en.wikipedia.org/wiki/Spectrogram. shows an example of a *spectrogram*, a visual representation of a spectrum, where x axis represents time (in secs), y axis represents frequency (in kHz) & 3rd axis in color represents intensity of sound (in dBFS [Decibel relative to full scale, a unit of measurement for amplitude levels in digital systems.]).
  - 4.3.4. Chromagram. A variation of spectrogram, discretized onto tempered scale & independent of octave, is a *chromagram*. It is restricted to *pitch classes* [A *pitch class* (also named a *chroma*) represents name of corresponding note independently of octave position. Possible pitch classes are C, C♯ (or D♭), D, . . ., A♯ (or B♭) & B.]. Chromagram of C major scale played on a

piano is illustrated in Fig. 4.4: Examples of chromagrams. (a) Musical score of a C-major scale. (b) Chromagram obtained from score. (c) Audio recording of C-major scale played on a piano. (d) Chromagram obtained from audio recording. Reproduced from MEINARD MUELLER's original image at https://en.wikipedia.org/wiki/Chromafeature.. x axis common to 4 subfigures (a–d) represents time (in secs). y axis of score (a) represents note, y axis of chromagrams (b & d) represents chroma (pitch class) & y axis of signal (c) represents amplitude. For chromagrams (b & d), 3rd axis in color represents intensity.

- ◦ 4.4. Symbolic. Symbolic representations are concerned with concepts like notes, duration, & chords, which will be introduced in following sects.
- ◦ 4.5. Main Concepts.
  - ∗ 4.5.1. Note. In a symbolic representation, a note is represented through following main features, & for each future there are alternative ways of specifying its value:
    - · *Pitch ([singular, uncountable] how high or low a sound is, especially a musical note) – cao độ* – specified by
      1. *frequency*, in Hertz (Hz)
      2. *vertical position (height)* on a score, or
      3. *pitch notation* [Also named *international pitch notation* or *scientific pitch notation.*], which combines a musical note name, e.g., A, A$\sharp$, B, etc. – actually its pitch class – & a number (usually notated in subscript) identifying pitch class octave which belongs to $[-1, 9]$ discrete interval. An example is A_4, which corresponds to A440 – with a frequency of 440 Hz – & serves as a general pitch tuning standards.
    - · *Duration* – specified by
      1. *absolute value*, in milliseconds (ms), or
      2. *relative value*, notated as a division or a multiple of a reference note duration, i.e., whole note. E.g.: a quarter note [Named a *crotchet* in British English.] & an 8th note [Named a *quaver* in British English.]
    - · *Dynamics* – specified by
      1. *absolute & quantitative value*, in decibels (dB), or
      2. *qualitative value*, an annotation on a score about how to perform note, which belongs to discrete set $\{ppp, pp, p, f, f, ff, fff\}$, from pianissimo to fortissimo.
  - ∗ 4.5.2. Rest. Rests are important in music as they represent intervals of silence allowing a pause for breath [As much for appreciation of music as for respiration by human performer(s)! – Cũng giống như việc thưởng thức âm nhạc cũng như việc hô hấp của người biểu diễn vậy!]. A *rest* can be considered as a special case of a note, with only 1 feature, its duration, & no pitch or dynamics. Duration of a rest may be specified by
    - · *absolute value*, in milliseconds (ms); or
    - · *relative value*, notated as a division or a multiple of a reference rest duration, whole rest having same duration as a whole note. Examples are a quarter rest & an 8th rest, corresponding resp. to a quarter note & an 8th note.
  - ∗ 4.5.3. Interval. An interval is a relative transition between 2 notes. E.g.: a major 3rd (which includes 4 semitones – nửa cung), a minor 3rd (3 semitones) & a (perfect) 5th (7 semitones). Intervals are basis of chords. E.g., 2 main chords in classical music are major (with a major 3rd & a 5th) & minor (with a minor 3rd & a 5th).

    In pioneering experiments described in [189], TODD discusses an alternative way for representing pitch of a note. Idea: not to represent it in an *absolute* way as in Sect. 4.5.1, but in a *relative* way by specifying relative transition (measured in semitones), i.e., interval, between 2 successive notes. E.g., melody C_4, E_4, G_4 would be represented as $C_4, +4, +3$.

    In [189], TODD points out as 2 advantages fact: there is no fixed bounding of pitch range & fact: it is independent of a given key (tonality). However, he also points out: this 2nd advantage may also be a major drawback, because in case of an error in generation of an interval (resulting in a change of key), wrong tonality (because of a wrong index) will be maintained in rest of melody generated. Another limitation: this strategy applies only to specification of a monophonic melody & cannot directly represent a single-voice polyphony, unless separating parallel intervals into different voices. Because of these pros & cons, an interval-based representation is actually rarely used in DL-based music generation systems.

    – Trong [189], TODD chỉ ra 2 lợi thế thực tế: không có ranh giới cố định của phạm vi cao độ & thực tế: nó độc lập với 1 khóa nhất định (âm điệu). Tuy nhiên, ông cũng chỉ ra: lợi thế thứ 2 này cũng có thể là 1 nhược điểm lớn, bởi vì trong trường hợp xảy ra lỗi khi tạo ra 1 khoảng (dẫn đến thay đổi khóa), âm điệu sai (do chỉ số sai) sẽ được duy trì trong phần còn lại của giai điệu được tạo ra. Một hạn chế khác: chiến lược này chỉ áp dụng cho việc chỉ định 1 giai điệu đơn âm & không thể biểu diễn trực tiếp 1 phức điệu đơn âm, trừ khi tách các khoảng song song thành các giọng khác nhau. Do những ưu & nhược điểm này, biểu diễn dựa trên khoảng thực sự hiếm khi được sử dụng trong các hệ thống tạo nhạc dựa trên DL.
  - ∗ 4.5.4. Chord. A representation of a *chord*, which is a set of at least 3 notes (a triad) [Modern music extends original major & minor triads into a huge set of richer possibilities (diminished, augmented, dominant 7th, suspended, 9th, 13th, etc.) by adding &/or altering intervals/components.], could be
    - · *implicit & extensional*, enumerating exact notes composing it. This permits specification of precise octave as well as position (voicing) for each note, see, e.g., in Fig. 4.5: C major chord with an open position/voicing: 1-5-3 (root, 5th, & 3rd), or
    - · *explicit & intensional*, by using a chord symbol combining
      1. pitch class of its root note, e.g., C, &

2. *type*, e.g., major, minor, dominant 7th, or diminished [There are some abbreviated notations, frequent in jazz & popular music, e.g. C minor = Cmin = Cm = C-; C major 7th = CM7 = Cmaj7 = C∆, etc.]

Extensional approach (explicitly listing all component notes) is more common for DL-based music generation systems, but there are some examples of systems representing chords explicitly with intensional approach, as e.g. MidiNet system introduce in Sect. 6.10.3.3.

– Phương pháp mở rộng (liệt kê rõ ràng tất cả các nốt thành phần) phổ biến hơn đối với các hệ thống tạo nhạc dựa trên DL, nhưng có 1 số ví dụ về các hệ thống biểu diễn hợp âm rõ ràng bằng phương pháp mở rộng, chẳng hạn như hệ thống MidiNet được giới thiệu trong Phần 6.10.3.3.

∗ 4.5.5. Rhythm. *Rhythm* is fundamental to music. It conveys pulsation as well as stress on specific beats, indispensable for dance! Rhythm introduces pulsation, cycles & thus structure in what would otherwise remain a flat linear sequence of notes.

· 4.5.5.1. Beat & Meter. A *beat* is unit of pulsation in music. Beats are grouped into measures, separated by *bars* [Although (& because) a bar is actually *graphical entity* – line segment "|" – separating measures, term bar is also often used, specially in US, in place of measure. In this book, stick to term *measure*.]. Number of beats in a measure as well as duration between 2 successive beats constitute rhythmic signature of a measure & consequently of a piece of music [For more elaborate music, meter may change within different portions of music.]. This *time signature* is also often named *meter*. It is expressed as fraction *numberOfBeats*/*BeatDuration*, where

1. *numberOfBeats*: number of beats within a measure, &
2. *beatDuration*: duration between 2 beats. As with relative duration of a note (Sect. 4.5.1) or of a rest, it is expressed as a division of duration of a whole note.

More frequent meters are `2/4, 3/4, 4/4`. E.g., `3/4` means 3 beats per measure, each one with duration of a quarter note. It is rhythmic signature of a Waltz. Stress (or accentuation – sự nhấn mạnh) on some beats or their subdivisions may form actual style of a rhythm for music as well as for a dance, e.g., ternary jazz vs. binary rock.

· 4.5.5.2. Levels of Rhythm Information. May consider 3 different levels in terms of amount & granularity (độ chi tiết) of information about rhythm to be included in a musical representation for a DL architecture:

1. *None*: only notes & their durations are represented, without any explicit representation of measures. This is case for most systems.
2. *Measures*: measures are explicitly represented. An example is system described in Sect. 6.6.1.2 [Interesting to note: as pointed out by Sturm et al. in [178], generated music format also contains bars separating measures & there is no guarantee: number of notes in a measure will always fit to a measure. However, errors rarely occurs, indicating: this representation is sufficient for architecture to learn to count, see [59] & Sect. 6.6.1.2.].
3. *Beats*: information abut meter, beats, etc. is included. E.g.: C-RBM system described in Sect. 6.10.5.1, which allows us to impose a specific meter & beat stress for music to be generated.

◦ 4.6. Multivoice/Multitrack. A *multivoice* representation, also named *multitrack*, considers independent various voices, each being a different vocal range (e.g., soprano, alto, . . .) or a different instrument (e.g., piano, bass, drums, . . .). Multivoice music is usually modeled as parallel tracks, each one with a distinct sequence of notes [With possibly simultaneous notes for a given voice, see Sect. 3.1.1.], sharing same meter but possibly with different strong (stressed) beats [Dance music is good at this, by having some syncopated bass &/or guitar not aligned on strong drum beats, in order to create some bouncing pulse. – Nhạc dance rất hay ở điểm này, bằng cách sử dụng 1 số âm trầm &hoặc tiếng guitar không đồng bộ với nhịp trống mạnh, nhằm tạo ra nhịp đập mạnh mẽ.].

Note: in some cases, although there are simultaneous notes, representation will be a single-voice polyphony, as introduced in Sect. 3.1.1. Common examples are polyphonic instruments like a piano or a guitar. Another example is a drum or percussion kit – bộ gõ, where each of various components, e.g., snare, hi-hat, ride cymbal, kick, etc., will usually be considered as a distinct note for same voice.

– Lưu ý: trong 1 số trường hợp, mặc dù có các nốt nhạc đồng thời, nhưng cách thể hiện sẽ là 1 hợp âm đơn âm, như đã giới thiệu trong Phần 3.1.1. Ví dụ phổ biến là các nhạc cụ hợp âm như piano hoặc guitar. Một ví dụ khác là bộ trống hoặc bộ gõ – bộ gõ, trong đó mỗi thành phần khác nhau, ví dụ như snare, hi-hat, ride cymbal, kick, v.v., thường được coi là 1 nốt riêng biệt cho cùng 1 giọng.

Different ways to encode single-voice polyphony & multivoice polyphony are further discussed in Sect. 4.11.2.

◦ 4.7. Format. Fromat is language (i.e., grammar & syntax) in which a piece of music is expressed (specified) in order to be interpreted by a computer [Standard format for humans is a musical score – âm phổ.].

∗ 4.7.1. MIDI. Musical Instrument Digital Interface (MIDI) is a technical standard that describes a protocol, a digital interface & connectors for interoperability between various electronic musical instruments, softwares, & devices [132]. MIDI carries event messages that specify real-time note performance data as well as control data. Only consider here 2 most important messages for our concerns:

· *Note on*: to indicate: a note is played. It contains

1. a *channel number*, which indicates instrument or track, specified by an integer within set $\{0, 1, \dots, 15\}$
2. a MIDI *note number*, which indicates note *pitch*, specified by an integer within set $\{0, 1, \dots, 127\}$
3. a *velocity*, which indicates how loud note is played [For a keyboard, it means speed of pressing down key & therefore corresponds to volume.], specified by an integer within set $\{0, 1, \dots, 127\}$.

E.g.: "Note on, 0. 60, 50" which means "On channel 1, start playing a middle C with velocity 50"

- *Note off*: to indicate : a note ends. In this situation, velocity indicates how fast note is released. E.g.: "Note off, 0, 60, 20" which means "On channel 1, stop playing a middle C with velocity 20".

Each note even is actually embedded into a track chunk, a data structure containing a delta-time value which specifies timing information & event itself. A *delta-time value* represents time position of event & could represent

- a *relative metrical* time – number of *ticks* from beginning. A reference, named *division* & defined in file header, specifies number of ticks per quarter note; or
- an *absolute time* – useful for real performances, not detailed here, see [132].

An example of an excerpt from a MIDI file (turned into readable ascii) & its corresponding score are shown Fig. 4.6: Excerpt from a MIDI file & Fig. 4.7:Score corresponding to MIDI excerpt. Division has been set to 384, i.e. 384 ticks per quarter note (which corresponds to 96 ticks for a 16th note).

```
2, 96, Note_on, 0, 60, 90
2, 192, Note_off, 0, 60, 0
2, 192, Note_on, 0, 62, 90
2, 288, Note_off, 0, 62, 0
2, 288, Note_on, 0, 64, 90
2, 384, Note_off, 0, 64, 0
```

In [86], Huang & Hu claim: 1 drawback of encoding MIDI messages directly: it does not effectively preserve notion of multiple notes being played at once through use of multiple tracks. In their experiment, they concatenate tracks end-to-end & thus posit: it will be difficult for such a model to learn: multiple notes in same position across different tracks can really be played at same time. Piano roll does not have this limitation but at cost another limitation.

– Trong [86], Huang & Hu tuyên bố: 1 nhược điểm của việc mã hóa trực tiếp các thông điệp MIDI: nó không bảo toàn hiệu quả khái niệm về nhiều nốt nhạc được chơi cùng 1 lúc thông qua việc sử dụng nhiều bản nhạc. Trong thí nghiệm của họ, họ nối các bản nhạc từ đầu đến cuối & do đó đưa ra giả thuyết: sẽ rất khó để 1 mô hình như vậy học được: nhiều nốt nhạc ở cùng 1 vị trí trên các bản nhạc khác nhau thực sự có thể được chơi cùng 1 lúc. Piano roll không có hạn chế này nhưng lại có 1 hạn chế khác.

∗ 4.7.2. Piano Roll. *Piano roll* representation of a melody (monophonic or polyphonic) is inspired from automated pianos (Fig. 4.8: Automated piano & piano roll. Yaledmot's post https://www.youtube.com/watch?v=QrcwR7eijyc.). This was a continuous roll of paper with perforations (holes) punched into it. Each perforation represents a piece of *note control information*, to trigger a given note. *Length* of perforation corresponds to duration of a note. In other dimensions, *localization* of a perforation corresponds to its pitch.

An example of a modern piano roll representation (for digital music systems) is shown in Fig. 4.9: Example of symbolic piano roll [70] permission of Hao Staff Music Publishing (Hong Kong) Co Ltd. x axis represents time & y axis pitch.

There are several music environments using piano roll as a basic visual representation, in place of or in *complement* to a score, as more intuitive than tradition score notation [Another notation specific to guitar or string instruments is a *tablature*, in which 6 lines represent chords of a guitar (4 lines for a bass) & note is specified by number of fret used to obtain it.]. E.g.: Hao Staff piano roll sheet music [70], shown in Fig. 4.9 with time axis being horizontal rightward & notes represented as green cells. E.g.: tabs, where melody is represented in a piano roll-like format [84], in complement to chords & lyrics. Tabs are used as an input by MidiNet system, introduced in Sect. 6.10.3.3.

Piano roll is 1 of most commonly used representations, although it has some limitations. An important one, compared to MIDI representation: there is no note off information. As a result, there is no way to distinguish between a long note & a repeated short note [Actually, in original mechanical paper piano roll, distinction is made: 2 holes are different from a longer single hole. End of hole is encoding of end of note.]. In Sect. 4.9.1, look at different ways to address this limitation. For a more detailed comparison between MIDI & piano roll, see [86,199].

∗ 4.7.3. Text.

- 4.7.3.1. Melody. A melody can be encoded in a textual representation & processed as a *text*. A significant example: ABC notation [201], a *de facto* standard for folk & traditional music [Note: ABC notation has been designed *independently* of computer music & ML concerns.]. Fig. 4.10: Score of "A Cup of Tea" (Traditional), reproduced from The Session [98]. & Fig. 4.11: ABC notation of "A Cup of Tea" reproduced from The Session [98]. show original score & its associated ABC notation for a tune named "A Cup of Tea", from repository & discussion platform The Session [98].

  1st 6 lines are header & represent *metadata*: T: title of music, M: meter, L: default note length, K: key, etc. Header is followed by main text representing melody. Some basic principles of encoding rules of ABC notation are as follows [Refer to [201] for more details]:

  1. pitch class of a note is encoded as letter corresponding to its English notation, e.g., `A` for A or La
  2. its pitch is encoded as following: A corresponds to A_4, `a` to an A one octave up & `a'` to an A 2 octaves up
  3. duration of a note is encoded as following: if default length is marked as `1/8` (i.e. an 8th note, case for "A Cup of Tea" example), `a` corresponds to an 8th note, `a/2` to a 16th note & `a2` to a quarter note [Note: rests may be expressed in ABC notation through `z` letter. Their durations are expressed as for notes, e.g., `z2` is a double length rest.]
  4. measure are separated by | (bars).

  Note: ABC notation can only represent monophonic melodies.

In order to be processed by a DL architecture, ABC text is usually transformed from a character vocabulary text into a *token* vocabulary text in order to properly consider concepts which could be noted on > 1 character, e.g., `g2`. Sturm et al.'s experiment, described in Sect. 6.6.1.2, uses a token-based notation named folk-rnn notation [178]. A tune is enclosed within a `<s>` begin mark & an `<\s>` end mark. Last, all examples melodies are transposed to same C root base, resulting in notation of tune "A Cup of Tea" shown in Fig. 4.12: Folk-rnn notation of a "A Cup of Tea" [178].

    - 4.7.3.2. Chord & Polyphony. When represented extensionally, chords are usually encoded with simultaneous notes as a vector. An interesting alternative extensional representation of chords, named Chord2Vec [Chord2Vec is inspired by Word2Vec model for natural language processing [129].], has recently been proposed in [121] [For information, there is another similar model, also named Chord2Vec, proposed in [87].]. Rather than thinking of chords (vertically) as vectors, it represents chords (horizontally) as sequences of constituent notes. More precisely,

      1. a chord is represented as an arbitrary length-ordered sequence of notes
      2. chords are separated by a special symbol, as with sentence markers in natural language processing.

      When using this representation for predicting neighboring chords, a specific compound architecture is used, named RNN Encoder-Decoder described in Sect. 6.10.2.3.

      Note: a somewhat similar model is also used for polyphonic music generation by BachBot system [118] introduced in Sect. 6.17.1. In this model, for each time step, various notes (ordered in a descending pitch) are represented as a sequence & a special delimiter symbol | | | indicates next time frame.

  - 4.7.4. Markup Language. Mention case of general text-based structured representations based on markup languages (famous examples are HTML & XML). Some markup languages have been designed for music applications, like e.g. open standard MusicXML [61]. Motivation: provide a common format to facilitate sharing, exchange & storage of scores by musical software systems (e.g. score editors & sequencers). MusicXML, as well as similar languages, is not intended for direct use by humans because of its verbosity (sự dài dòng), which is down side of its richness & effectiveness as an interchange language. Furthermore, not very appropriate as a direct representation for ML tasks for same reasons, as its verbosity & richness would create too much overhead as well as bias.

    – Ngôn ngữ đánh dấu. Đề cập đến trường hợp biểu diễn có cấu trúc dựa trên văn bản chung dựa trên ngôn ngữ đánh dấu (ví dụ nổi tiếng là HTML & XML). Một số ngôn ngữ đánh dấu đã được thiết kế cho các ứng dụng âm nhạc, chẳng hạn như chuẩn mở MusicXML [61]. Động lực: cung cấp 1 định dạng chung để tạo điều kiện chia sẻ, trao đổi & lưu trữ bản nhạc của các hệ thống phần mềm âm nhạc (ví dụ: trình chỉnh sửa bản nhạc & trình sắp xếp). MusicXML, cũng như các ngôn ngữ tương tự, không dành cho con người sử dụng trực tiếp vì tính dài dòng của nó, đây là nhược điểm của tính phong phú & hiệu quả của nó như 1 ngôn ngữ trao đổi. Hơn nữa, không thực sự phù hợp để biểu diễn trực tiếp cho các tác vụ ML vì những lý do tương tự, vì tính dài dòng & phong phú của nó sẽ tạo ra quá nhiều chi phí cũng như thiên vị.

  - 4.7.5. Lead Sheet. Lead sheets are an important representation format for popular music (jazz, pop, etc.). A *lead sheet* conveys in upto a few pages score of a melody & its corresponding chord progression via an intentional notation (Sect. 4.5.4). Lyrics may also be added. Some important information for performer, e.g. composer, author, style & tempo, is often also present. An example of lead sheet is shown in Fig. 4.13: Lead sheet of "Very Late" (Pachet & d'Inverno).

    Paradoxically, few systems & experiments use this rich & concise representation, & most of time they focus on notes. Note: Eck & Scmidhuber's Blues generation system, introduced in Sect. 6.5.1.1, outputs a combination of melody & chord progression, although not as an *explicit* lead sheet. A notable contribution: systematic encoding of lead sheets done in Flow Machines project [48], resulting in Lead Sheet Data Base (LSBD) repository [146], which includes > 12000 lead sheets.

    Note: there are some alternative notations, notably tabs [84], where melody is represented in a piano roll-like format (Sect. 4.7.2) & complemented with corresponding chords. An example of use of tabs: MidiNet system to be analyzed in Sect. 6.10.3.3.

- 4.8. Temporal Scope & Granularity. Representation of time is fundamental for musical processes.

  - 4.8.1. Temporal Scope. An initial design decision concerns *temporal scope* of representation used for generation data & for generated data, i.e. way representation will be interpreted by architecture w.r.t. time, as illustrated in Fig. 4.14:

    - *Global*: in this 1st case, temporal scope of representation is *whole* musical piece. Deep network architecture (typically a feedforward or an autoencoder architecture, see Sects. 5.5–5.6) will process input & produce output within a *global single step* [In Chap. 6, name it *single-step feedforward strategy*, Sect. 6.2.1.]. E.g.: MiniBach & DeepHear systems introduced in Sects. 6.2.2 & 6.4.1.1, resp.
    - *Time step* (or *time step*) – in this 2nd case, most frequent one, temporal scope of representation is a *local time slice* of musical piece, corresponding to a specific temporal moment (time step). Granularity of processing by deep network architecture (typically a recurrent network) is a *time step* & generation is iterative [In Chap. 6, name it *interactive feedforward strategy*, see Sect. 6.5.1.]. Note: time step is usually set to *shortest note duration* (see more details in Sect. 4.8.2), but it may be larger, e.g., set to a measure in system as discussed in [189].
    - *Note step*: this 3rd case was proposed by Mozer in [138] in his CONCERT system [138], see Sect. 6.6.1.1. In this approach there is *no fixed time step*. Granularity of processing by deep network architecture is a *note*. This strategy uses a distributed encoding of duration that allows to process a note of any duration in a single network processing step. Note: by considering as a single processing step a note rather than a time step, number of processing steps to be bridged by network is greatly reduced. Approach proposed later on by Walder in [199] is similar.

      Note: a global temporal scope representation actually also considers time steps (separated by dash lines in Fig. 4.14: Temporal scope for a piano role-like representation.). However, although time steps are present at representation level,

they will not be interpreted as distinct processing steps by neural network architecture. Basically, encoding of successive time slices will be concatenated into a global representation considered as a whole by network, as shown in Fig. 6.2: MiniBach architecture & encoding of an example to be introduced in Sect. 6.2.2.
Note: in case of a global temporal scope musical content generated has a *fixed length* (number of time steps), whereas in case of a time step or a note step temporal scope musical content generated has an *arbitrary length*, because generation is iterative as see in Sect. 6.5.1.

∗ 4.8.2. Temporal Granularity. In case of a global or a time step temporal scope, granularity of time step, corresponding to granularity of time *discretization*, must be defined. There are 2 main strategies:

· Most common strategy: set time step to a *relative duration*, smallest duration of a note in corpus (training examples/dataset), e.g., a 16th note. To be more precise, as stated by TODD in [189], time step should be *greatest common factor* of durations of all notes to be learned. This ensures: duration of every note will be properly represented with a whole number of time steps. 1 immediate consequence of this "leveling down": number of processing steps necessary, independent of duration of actual notes.

· Another strategy: set time step to a fixed *absolute duration*, e.g., 10 milliseconds. This strategy permits us to capture expressiveness in timing of each note during a human performance, as see in Sect. 4.10.

Note: in case of a note step temporal scope, there is no uniform discretization of time (no fixed time step) & no need for.

◦ 4.9. Metadata. In some systems, additional information from score may also be explicitly represented & used as *metada*, e.g.

∗ note tie [A tied note on a music score specifies how a note duration extends across a single measure. In our case, issue is how to specify: duration extends across a single *time step*. Therefore, consider it as metadata information, as it is specific to representation & its processing by a neural network architecture.]

∗ fermata

∗ harmonics

∗ key

∗ meter

∗ instrument associated as a voice.

This extra information may lead to more accurate learning & generation.

∗ 4.9.1. Note Hold/Ending. An important issue: how to represent if a note is held, i.e., tied to prev note. This is actually ⇔ issue of how to represent ending of a note.
In MIDI representation format, end of a note is explicitly stated (via a "Note off" event [Note: in MIDI, a "Note on" message with a null (0) velocity is interpreted as a "Note off" message.]). In piano roll format discussed in Sect. 4.7.2, there is no explicit representation of ending of a note &, as a result, one cannot distinguish between 2 repeated quarter notes & a half note.
Main possible techniques:

· introduce a *hold/replay* representation, as a dual representation of sequence of notes. This solution is used, e.g., by Mao et al. in their DeepJ system [126] (to be analyzed in Section 6.10.3.4), by introducing a replay matrix similar to piano roll-type matrix of notes

· divide size of time step [See Sect. 4.8.2 for details of how value of time step is defined.] by 2 & always mark a *note ending* with a special tag, e.g., 0. This solution is used, e.g., by ECK & SCHMIDHÜBER in [42], & analyzed in Sect. 6.5.1.1

· divide size of time step as before but instead mark a *new note beginning*. This solution is used by TODD in [189] or

· use a special *hold* symbol __ in place of a note to specify when prev note is held. This solution was proposed by Hadjeres et al. in their DeepBach system [69] to be analyzed in Sect. 6.14.2.

This last solution considers hold symbol as a note, see an example in Fig. 4.15: (a) Extract from a J. S. BACH chorale & (b) its representation using hold symbol __ [69]. Advantages of hold symbol technique:

· simple & uniform as hold symbol is considered as a note

· there is no need to divide value of time step by 2 & mark a note ending or beginning.

Authors of DeepBach also emphasize: good results they obtain using Gibbs sampling rely exclusively on their choice to integrate hold symbol into list of notes (see [69] & Sect. 6.14.2). An important limitation: hold symbol only applies to case of a monophonic melody, i.e. it cannot directly express held notes in an unambiguous way in case of a single-voice polyphony. In this case, single-voice polyphony must be reformulated into a multivoice representation with each voice being a monophoniic melody; then a hold symbol is added separately for each voice. Note: in case of replay matrix, additional information (matrix row) is for each possible note & not for each voice.
Discuss in Sect. 4.11.7 how to encode a hold symbol.

∗ 4.9.2. Note Denotation (vs. Enharmony). Most systems consider *enharmony*, i.e., in tempered system A♯ is *enharmonically equivalent* to (i.e., has same pitch as) B♭, although harmonically & in composer's intention they are different. An exception: DeepBach system, described in Sect. 6.14.2, which encodes notes using their real names & not their MIDI note numbers. Authors of DeepBach state: this additional information leads to a more accurate model & better results [69].

∗ 4.9.3. Feature Extraction. Although DL is good at processing raw unstructured data, from which its hierarchy of layers will extract higher-level representations adapted to task (Sect. 1.1.4), some systems include a preliminary step of automatic *feature extraction*, in order to represent data in a more compact, characteristic & discriminative form. 1 motivation could be to gain efficiency & accuracy for training & for generation. Moreover, this *feature-based representation* is also useful for indexing data, in order to control generation through compact labeling (see, e.g., DeepHear system in Sect. 6.4.1.1), or for indexing musical units to be queried & concatenated (see Sect. 6.10.7.1).

Set of *features* can be defined *manually (handcrafted)* or *automatically* (e.g., by an autoencoder, Sect. 5.6). In case of handcrafted features, bag-of-words (BOW) model is a common strategy for natural language text processing, which may also be applied to other types of data, including musical data, as see in Sect. 6.10.7.1. It consists in transforming original text (or arbitrary representation) into a "bag of words" (vocabulary composed of all occurring words, or more generally speaking, all possible tokens); then various measures can be used to characterize text. Most common is *term frequency*, i.e., number of times a term appears in text [Note: this bag-of-words representation is a *lossy representation* (i.e., without effective means to perfectly reconstruct original data representation).]

Sophisticated methods have been designed for neural network architectures to automatically compute a vector representation which preserves, as much as possible, relations between items. Vector representations of texts are named *word embeddings* [Term *embedding* comes from analogy with *mathematical embedding*, which is an objective & structure-preserving mapping. Initially used for natural language processing, it is now often used in DL as a general term for *encoding* a given representation into a vector representation. Note: term embedding, which is an abstract model representation, is often also used (we think, abusively) to define a specific instance of an embedding (which may be better named, e.g., a *label*, see [179] & Sect. 6.4.1.1).]. A recent reference model for NLP is Word2Vec model [129]. It has recently been transposed to Chord2Vec model for vector encoding of chords, as described in [121] (Sect. 4.5.4).

- 4.10. Expressiveness.
  - 4.10.1. Timing. If training examples are processed from conventional scores or MIDI-format libraries, there is a good chance: music is perfectly *quantized* – i.e., note onsets [An onset refers to beginning of a musical note (or sound).] are exactly aligned onto tempo – resulting in a mechanical sound without *expressiveness*. 1 approach: consider symbolic records – in most cases recorded directly in MIDI – from real human *performances*, with musician interpreting tempo. An example of a system for this purpose is PerformanceRNN [173], analyzed in Sect. 6.7.1. It follows *absolute time duration* quantization strategy, presented in Sect. 4.8.2.
  - 4.10.2. Dynamics. Another common limitation: many MIDI-format libraries do not include *dynamics* (volume of sound produced by an instrument), which stays fixed throughout whole piece. 1 option: take into consideration (if present on score) annotations made by composer about dynamics, from pianissimo ppp to fortissimo fff, see Sect. 4.5.1. As for tempo expressiveness, addressed in Sect. 4.10.1, another option: use real human performances, recorded with explicit dynamics variation – velocity field in MIDI.
  - 4.10.3. Audio. Note: in case of an audio representation, expressiveness as well as tempo & dynamics are entangled within whole representation. Although easy to control global dynamics (global volume), less easy to separately control dynamics of a single instrument or voice [More generally speaking, audio source separation, often coined as *cocktail party effect*, has been known for a long time to be a very difficult problem, see original article in [18]. Interestingly, this problem has been solved in 2015 by DL architectures [45], opening up ways for disentangling instruments or voices & their relative dynamics as well as tempo (by using audio time stretching techniques).]
- 4.11. Encoding. Once format of a representation has been chosen, issue still remains of how to *encode* this representation. *Encoding* of a representation (of a musical content) consists in *mapping* of representation (composed of a set of *variables*, e.g., pitch or dynamics) into a set of *inputs* (also named *input nodes* or *input variables*) for neural network architecture [See Sect. 5.4 for more details about input nodes of a neural network architecture.].
  - 4.11.1. Strategies. At 1st, consider 3 possible types for a variable:
    - *Continuous* variables – an example is pitch of a note defined by its frequency in Hertz, that is a real value within $(0, \infty)$ interval.

      Straightforward way: directly encode variable [In practice, different variables are also usually scaled & normalized, in order to have similar domains of values ($[0, 1]$ or $[-1, 1]$) for all input variables, in order to ease learning convergence.] as a *scalar* whose domain is real values. Call this strategy *value encoding*.
    - *Discrete integer* variables – e.g.: pitch of a note defined by its MIDI note number, i.e., an integer value within $\{0, 1, \ldots, 127\}$ discrete set [See summary of MIDI specification in Sect. 4.7.1.].

      Straightforward way: encode variable as a real value *scalar*, by casting integer into a real. This is another case of *value encoding*.
    - *Boolean (binary)* variables – e.g.: specification of a note ending (see Sect. 4.9.1).

      Straightforward way: encode variable as a real value *scalar*, with 2 possible values: 1 (for true) & 0 (for false).
    - *Categorical* variables [In statistics, a *categorical variable* is a variable that can take 1 of a limited – & usually fixed – number of possible values. In CS it is usually referred as an *enumerated type*.] – an example is a component of a drum kit; an element within a set of possible values: {snare, high-hat, kick, middle-tom, ride-cymbal, etc.}.

      Usual strategy: encode a categorical variable as a *vector* having as its length number of possible elements, i.e., cardinality of set of possible values. Then, in order to represent a given element, corresponding element of encoding vector is set to 1 & all other elements to 0. Therefore, this encoding strategy is usually called *1-hot encoding* [Name comes from digital circuits, *1-hot* referring to a group of bits among which only legal (possible) combinations of values are those with a single *high* (hot!) (1) bit, all others being *low* (0).]. This frequently used strategy is also often employed for encoding discrete integer variables, e.g. MIDI note numbers.
  - 4.11.2. From 1-Hot to Many-Hot & to Multi-1-Hot. Note: a 1-hot encoding of a note corresponds to a time slice of piano roll representation (Fig. 4.9), with as many lines as there are possible pitches. Note: while a 1-hot encoding of a piano roll

representation of a *monophonic* melody (with 1 note at a time) is straightforward, a 1-hot encoding of a *polyphony* (with simultaneous notes, as for a guitar playing a chord) is not. One could then consider

- *many-hot encoding*: where all elements of vector corresponding to notes or to active components are set to 1
- *multi-1-hot encoding*: where different voices or tracks are considered (for multivoice representation, Sect. 4.6) & a 1-hot encoding is used for each different voice/track, or
- *multi-many-hot encoding*: which is a multivoice representation with simultaneous notes for at least 1 or all of voices.

∗ 4.11.3. Summary. Various approaches for encoding are illustrated in Fig. 4.16: Various types of encoding, showing from left to right

- a scalar continuous value encoding of A_4 (A440), real number specifying its frequency in Hertz
- a scalar discrete integer value encoding [Note: because processing level of an ANN only considers real values, an integer value will be casted into a real value. Thus, case of a scalar integer value encoding boils down to prev case of a scalar continuous value encoding.] of A_4, integer number specifying its MIDI note number
- a 1-hot encoding of A_4
- a many-hot encoding of a D minor chord D_4, F_4, A_4
- a multi-1-hot encoding of a 1st voice with A4 & a 2nd voice with D3
- a multi-many-hot encoding of a 1st voice with a D minor chord D_4, F_4, A_4 & a 2nd voice with C_3 (corresponding to a minor 7th on bass).

∗ 4.11.4. Binning. In some cases, a continuous variable is transformed into a discrete domain. A common technique, named *binning*, or also *bucketing*, consists of

- dividing original domain of values into smaller interval [This can be automated through a learning process, e.g., by automatic construction of a decision tree.], named *bins*
- replacing each bin (& values within it) by a *value representative*, often central value.

Note: this binning technique may also be used to reduce cardinality of discrete domain of a variable. E.g.: Performance RNN system described in Sect. 6.7.1, for which initial MIDI set of 127 values for note dynamics is reduced into 32 bins.

∗ 4.11.5. Pros & Cons. In general, value encoding is rarely used except for audio, whereas 1-hot encoding is most common strategy for symbolic representation [Remind (as pointed out in Sect. 4.2): at level of encoding of a representation & its processing by a deep network, distinction between audio & symbolic representation boils down to nothing, as only numerical values & operations are considered. In fact general principles of a DL architecture are independent of that distinction & this is 1 of vectors of generality of approach. See also in [125] example of an architecture (introduced in Sect. 6.10.3.2) which combines audio & symbolic representations.].

A counterexample is case of DeepJ symbolic generation system described in Sect. 6.10.3.4, which is, in part, inspired by WaveNet audio generation system. DeepJ's authors state: "Keep track of dynamics of every note in an $N \times T$ dynamics matrix that, for each time step, stores values of each note's dynamics scaled between 0 & 1, where 1 denotes loudest possible volume. In preliminary work, also tried an alternate representation of dynamics as a categorical value with 128 bins as suggested by Wavenet [193]. Instead of predicting a scalar value, our model would learn a multinomial distribution of note dynamics. Would then randomly sample dynamics during generation from this multinomial distribution. Contrary to Wavenet's results, our experiments concluded: scalar representation yielded results that were more harmonious." [126].

Advantage of value encoding is its compact representation, at cost of sensibility because of numerical operations (approximations). Advantage of 1-hot encoding is its robustness (discrete vs. analog), at cost of a high cardinality & therefore a potentially large number of inputs.

Also important to understand: choice of 1-hot encoding at *output* of network architecture is often (albeit not always) associated to a *softmax* function [Introduced in Sect. 5.5.3.] in order to compute probabilities of each possible value, e.g. probability of a note being an A, or an A♯, a B, a C, etc. This actually corresponds to a *classification task* between possible values of categorical variable, further analyzed in Sect. 5.5.3.

∗ 4.11.6. Chords. 2 methods of encoding chords, corresponding to 2 main alternative representations discussed in Sect. 4.5.4, are

- *implicit & extensional*: enumerating exact notes composing chord. Natural encoding strategy is many-hot. E.g.: RBM-based polyphonic music generation system described in Sect. 6.4.2.3
- *explicit & intensional* – using a chord symbol combining a pitch class & a type (e.g., D minor). Natural encoding strategy is multi-1-hot, with an initial 1-hot encoding of pitch class & a 2nd 1-hot encoding of class type (major, minor, dominant 7th, etc.). E.g.: MidiNet system [In MidiNet, possible chord types are actually reduced to only major & minor. Thus, a boolean variable can be used in place of 1-hot encoding.] described in Sect. 6.10.3.3.

∗ 4.11.7. Special Hold & Rest Symbols. Have to consider case of special symbols for hold ("hold prev note", Sect. 4.9.1) & rest ("no note", Sect. 4.5.2) & how they relate to encoding of actual notes.

1st, note: there are some rare cases where rest is actually *implicit*:

- in MIDI format – when there is no "active" "Note on", i.e., when they al have been "closed" by a corresponding "Note off"
- in 1-hot encoding – when all elements of vector encoding possible notes are = 0 (i.e., a "0-hot" encoding, i.e., none of possible notes is currently selected). This is e.g. case in experiments by Todd (described in Sect. 6.8.1) [This may appear at 1st as an economical encoding of a rest, but at cost of some ambiguity when interpreting probabilities (for each possible note) produced by softmax output of network architecture. A vector with low probabilities for each note

may be interpreted as a rest or as an equiprobability between notes. See threshold trick proposed in Sect. 6.8.1 in order to discriminate between 2 possible interpretations.]

Consider how to encode hold & rest depending on how a note pitch is encoded:

- *value encoding*: in this case, one needs to add 2 extra boolean variables (& their corresponding input nodes) *hold & rest*. This must be done for each possible independent voice in case of a polyphony, or
- *1-hot encoding*: In that case (most frequent & manageable strategy), one just needs to extend vocabulary of 1-hot encoding with 2 additional possible values: *hold & rest*. They will be considered at same level, & of same nature, as possible notes (e.g., A_3 or C_4) for input as well as for output.

∗ 4.11.8. Drums & Percussion. (Trống & Bộ gõ) Some systems explicitly consider drums &/or percussion. A drum or percussion kit is usually modeled as a single-track polyphony by considering distinct simultaneous "notes", each "note" corresponding to a drum or percussion component (e.g., snare, kick, bass tom, hi-hat, ride cymbal, etc.), i.e. as a many-hot encoding.

An example of a system dedicated to rhythm generation is described in Sect. 6.10.3.1. It follows single-track polyphony approach. In this system, each of 5 components is represented through a binary value, specifying whether or not there is a related event for current time step. Drum events are represented as a binary word [In this system, encoding is made in text, similar to format described in Sect. 4.7.3 & more precisely following approach proposed in [21].] of length 5, where each binary value corresponds to 1 of 5 drum components; e.g., 10010 represents simultaneous playing of kick (bass drum) & high-hat, following a many-hot encoding.

Note: this system also includes – as an additional voice/track – a condensed representation of bass line part & some information representing meter, see more details in Sect. 6.10.3.1. Authors [122] argue: this extra explicit information ensures: network architecture is aware of beat structure at any given point.

E.g.: MusicVAE system (Sect. 6.12.1), where 9 different drum/percussion components are considered, which gives 2^9 possible combinations, i.e., $2^9 = 512$ different tokens.

○ 4.12. Dataset. Choice of a dataset is fundamental for good music generation. At 1st, a dataset should be of sufficient size (i.e., contain a sufficient number of examples) to guarantee accurate learning [Neural networks & DL architectures need lots of examples to function properly. However, 1 recent research area is about learning from scarce data (dữ liệu khan hiếm).]. As noted by Hadjeres in [66]: "believe: this tradeoff between size of a dataset & its coherence is 1 of major issues when building deep generative models. If dataset is very heterogeneous (rất không đồng nhất), a good generative model should be able to distinguish different subcategories & manage to generalize well. On contrary, if there are only slight differences between subcategories, important t know if "averaged model" can produce musically-interesting results."

∗ 4.12.1. Transposition & Alignment. (Chuyển vị & Căn chỉnh) A common technique in ML: generate *synthetic data* as a way to artificially augment size of dataset (number of training examples) [This is named *dataset augmentation*.], in order to improve accuracy & generalization of learnt model (Sect. 5.5.10). In musical domain, a natural & easy way is *transposition*, i.e., to transpose all examples in all keys. In addition to artificially augmenting dataset, this provides a key (tonality) invariance of all examples & thus makes examples more generic (chung chung hơn). Moreover, this also reduces sparsity in training data. This transposition technique is, e.g., used in C-RBM system [108] described in Sect. 6.10.5.1.

An alternative approach: transpose (align) all examples into a *single common key*. This has been advocated for RNN-RBM system [11] to facilitate learning, see Sect. 6.9.1.

∗ 4.12.2. Datasets & Libraries. A practical issue: availability of datasets for training systems & also for evaluating & comparing systems & approaches. There are some reference datasets in image domain (e.g., MNIST [MNIST stands for Modified National Institute of Standards & Technology.] dataset about handwritten digits [112]), but none yet in music domain. However, various datasets or libraries [Difference between a dataset & a library: a dataset is almost ready for use to train a neural network architecture, as all examples are encoded within a single file & in same format, although some extra data processing may be needed in order to adapt format to encoding of representation for architecture or vice-versa, whereas a library is usually composed of a set of files, 1 for each example.] have been made public, with some examples listed below:

- Classical piano MIDI database [104]
- JSB Chorales dataset [Note: this dataset uses a quarter note quantization, whereas a smaller quantization at level of a 16th note should be used in order to capture smallest note duration (8th note), Sect. 4.9.1.] [1].
- LSDB (Lead Sheet Data Base) repository [146], with > 12000 lead sheets (including from all jazz & bossa nova song books), developed within Flow Machines project [48]
- MuseData library, an electronic library of classical music > 800 pieces, from CCARH in Stanford University [76]
- MusicNet dataset [187], a collection of 330 freely-licensed classical music recordings together with > 1 million annotated labels (indicating timing & instrumental information)
- Nottingham database, a collection of 1200 folk tunes in ABC notation [53], each tune consisting of a simple melody on top of chords, i.e. an ABC equivalent of a lead sheet
- Session [98], a repository & discussion platform for Celtic music in ABC notation containing > 15000 songs
- Symbolic Music dataset by Walder [200], a huge set of cleaned & preprocessed MIDI files
- TheoryTab database [84], a set of songs represented in a tab format, a combination of a piano roll melody, chords & lyrics, i.e. a piano roll equivalent of a lead sheet
- Yamaha e-Piano Competition dataset, in which participants MIDI performance records are made available [209].

- 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 $P(\mathrm{note} = A_4)$ that value of variable note is A_4 & probability distribution $P(\mathrm{note})$ 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 $\boldsymbol{\theta} = [\theta_1, \dots, \theta_n]^\top$
3. $\mathbf{x} = [x_1, \dots, x_n]$.

* 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)
$$\mathrm{sigmoid}(z) = \sigma(z) = \frac{1}{1 + e^{-z}},$$
  & 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
$$\mathrm{ReLU}(z) = \max(0, z)$$
  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)
$$\hat{y} = h(\mathbf{x}) = \mathrm{AF}(b + W\mathbf{x}).$$
  Feedforward computation of prediction (for architecture shown in Fig. 5.5) is illustrated in (5.13),
$$\hat{y} = h(\mathbf{x}) = \mathrm{AF}(b + W\mathbf{x}) = \cdots = [h_1(\mathbf{x}), h_2(\mathbf{x}), h_3(\mathbf{x})]^\top = [\hat{y}_1, \hat{y}_2, \hat{y}_3]^\top,$$
  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).

  $$h(X) = \mathrm{AF}(b + WX).$$

  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)

  $$h(X) = \cdots = \mathrm{AF}(b + WX) = \cdots = [h(X^{(1)}) \ \cdots \ h(X^{(m)})],$$

  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., $I(P_1P_2) = I(P_1) + I(P_2)$.

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 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 ≈ 70/30:
  - · *training set* – which will be used for training neural network
  - · *validation set*, also named *test set*[1] – 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].

[1] 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.

- 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.

  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.

Training an autoencoder represents a case of *unsupervised learning*, as examples do not contain any additional label information (effective value or class to be predicted). But trick: this is implemented using conventional supervised learning techniques, by presenting output data = input data [This is sometimes called *self-supervised* learning [109].]. In practice, autoencoder tries to learn identity function. As hidden layer usually has fewer nodes than input layer, *encoder* component must *compress* information while *decoder* has to *reconstruct*, as accurately as possible, initial information [Compared to traditional dimension reduction algorithms, e.g. principal component analysis (PCA), this approach has 2 advantages: (1) feature extraction is nonlinear (case of *manifold learning*, see [62, Sect. 5.11.3] & Sect. 5.6.2) & (2) in case of a sparse autoencoder (see next sect), number of features may be arbitrary (& not necessarily smaller than number of input parameters).]. This forces autoencoder to *discover* significant (discriminating) *features* to encode useful information into hidden layer nodes (also named *latent variables* [In statistics, *latent variables* are variables that are not directly observed but are rather inferred (through a mathematical model) from other variables that are observed (directly measured). They can serve to reduce dimensionality of data.]). Therefore, autoencoders may be used to automatically extract high-level *features* [109]. Set of features extracted are often named an *embedding* [See def of embedding in Sect. 4.9.3.]. Once trained, in order to extract features from an input, one just needs to feedforward input data & gather activations of hidden layer (values of latent variables).

Another interesting use of decoders: high-level control of content generation. Latent variables of an autoencoder constitute a compact representation of common features of learnt examples. By instantiating these latent variables & decoding embedding, can generate a new musical content corresponding to values of latent variables. Explore this strategy in Sect. 6.4.1.

- 5.6.1. Sparse Autoencoder. A *sparse autoencoder* is an autoencoder with a *sparsity* constraint, s.t. its hidden layer units are inactive most of time. Objective: enforce *specialization* of each unit in hidden layer as a specific *feature detector*.

  E.g., a sparse autoencoder with 100 units in its hidden layer & trained on 10 × 10 pixel images will learn to detect edges at different positions & orientations in images, as shown in Fig. 5.23: Visualization of input image motives that maximally activate each of hidden units of a sparse autoencoder architecture [140]. When applied to other input domains, e.g. audio or symbolic music data, this algorithm will learn useful features for those domains too.

Sparsity constraint is implemented by adding an additional term to cost function to be minimized, see more details in [140] or [62, Sect. 14.2.1].

∗ 5.6.2. Variational Autoencoder. A *variational autoencoder* (VAE) [101] has added constraint that encoded representation, latent variables, by convention denoted by variable z, follow some prior probability distribution $P(z)$. Usually, a *Gaussian distribution* [Also named *normal distribution*.] is chosen for its generality.

This constraint is implemented by adding a specific term to cost function, by computing cross-entropy between values of latent variables & prior distribution [Actual implementation is more complex & has some tricks (e.g., encoder actually generates a mean vector & a standard deviation vector) not detailed here.]. For more details about VAEs, an example of tutorial could be found in [37] & there is a nice introduction of its application to music in [162].

As with an autoencoder, a VAE will learn identity function, but furthermore decoder part will learn relation between a Gaussian distribution of latent variables & learnt examples. As a result, sampling from VAE is immediate, one just needs to

· sample a value for latent variables $z \sim P(z)$, i.e., z following distribution $P(z)$

· input it into decoder

· feedforward decoder to generate an output corresponding to distribution of examples, following $P(x|z)$ conditional probability distribution learnt by decoder.

This is in contrast to need for indirect & computationally expensive strategies e.g. Gibbs sampling for other architectures e.g. RBM, introduced in Sect. 5.7.

By construction, a variational autoencoder is representative of dataset that it has learnt, i.e., for any example in dataset, there is at least 1 setting of latent variables which causes model to generate something very similar to that example [37].

A very interesting characteristic of variational autoencoder architecture for generation purposes – therefore often considered as 1 type of a class of models named *generative models* – is in meaningful exploration of latent space, as a variational autoencoder is able to learn a "smooth" [I.e., a small change in latent space will correspond to a small change in generated examples, without any discontinuity or jump. For a more detailed discussion about which (& how) interesting effects (smoothness, parsimony & axis-alignment between data & latent variability) a VAE has on latent representation (vector of latent variables) learnt, see, e.g., [205].] latent space mappping to realistic examples. Note: this general objective is named *manifold learning* & more generally *representation learning* [8], i.e., learning of a representation capturing topology of a set of examples. As defined in [62, Sect. 5.11.3], a *manifold* is a connected set of points (examples) that can be approximated by a small number of dimensions, each one corresponding to a local direction of variation. An intuitive example is a 2D map capturing topology of cities dispersed (phân tán) on 3D earth, where a movement on map corresponds to a movement on earth.

To illustrate possibilities, train a VAE with only 2 latent variables on MNIST handwritten digits database dataset [112] (with 60000 examples, each one being an image of 28×28 pixels). Then, scan latent 2D plane, sampling latent values for 2 latent variables (i.e., sampling points within 2D latent space) at regular intervals & generating corresponding artificial digits by decoding latent points [As proposed & implemented in [22].]. Fig. 5.24: Various digits generated by decoding sampled latent points at regular intervals on MNIST handwritten digits database. shows examples of artificial digits generated.

Note: training VAE has forced it to compress information about actual examples by splitting (though encoder) information in 2 subsets:

· *specific* (discriminative) part, encoded within latent variables, &

· *common* part, encoded within weights of decoder, in order to be able to reconstruct as close as possible each original data [Indeed, there is no magic here, reversible compression from 28×28 variables to 2 variables must have extracted & stored missing information somewhere.].

VAE actually has been forced to find out dimensions of variations for dataset examples. By looking at figure, can guess: 2D *could* be:

· from angular to round elements for 1st variable (z_1, horizontally represented),

· size of compound element (circle or angle) for 2nd latent variable (z_2, vertically represented).

Note: cannot expect/force VAE towards semantics (meaning) of specific dimensions, as VAE will automatically extract them (this depends on dataset as well as on training configuration), & can only try to interpret them *a posteriori* [However, can construct arbitrary characteristic attributes from a subset of examples & impose them on other examples, by doing *attribute vector arithmetics*, as defined in immediately following list, & as will be illustrated in Fig. 6.73: Example of a melody generated by MusicVAE by adding a "high note density" attribute vector to latent space of an existing melody [161]. in Sect. 6.12.1.]. Examples of possible dimensions for music generation could be: number of notes [illustrated in Fig. 6.33: Examples of 2 measures long melodies (separated by double bar lines) generated by GLSR-VAE [68]. in Sect. 6.10.2.3.], range (distance from lowest to highest pitch), etc.

Once learnt by a VAE, latent representation (a vector of latent variables) can be used to explore latent space with various operations to control/vary generation of content. Some examples of operations on latent space, as proposed in [161,162] for MusicVAE system described in Sect. 6.12.1, are

· *translation*

· *interpolation*

· *averaging* of some points

· *attribute vector arithmetics*, by addition or subtraction of an attribute vector capturing a given characteristics [This attribute vector is composed as average latent vector for a collection of examples sharing that attribute (characteristic).]

Fig. 5.25: Comparison of interpolations between top & bottom melodies by (left) interpolating in data (melody) space & (right) interpolating in latent space & decoding it into melodies [161]. shows an interesting comparison of melodies resulting from

· interpolation in *data space*, i.e. space of representation of melodies

· interpolation in *latent space*, which is then decoded into corresponding melodies.

Interpolation in latent space produces more meaningful & interesting melodies than interpolation in data space (which basically just varies ratio of notes from 2 melodies), as can be heard in [160,163]. More details about these experiments will be provided in Sect. 6.12.1.

Variational autoencoders are therefore elegant & promising models, & as a result they are currently among hot approaches explored for generating content with controlled variations. Application to music generation is illustrated in Sects. 6.10.2.3 & 6.12.1.

∗ 5.6.3. Stacked Autoencoder. Idea of a *stacked autoencoder*: hierarchically nest successive autoencoders with decreasing numbers of hidden layer units. An example of a 2-layer stacked autoencoder [Note: convention in this case: count & notate number of nested autoencoders, i.e., number of hidden layers. This is different from depth of *whole* architecture, which is double. E.g., a 2-layer stacked autoencoder results in a 4-layer whole architecture, as shown in Fig. 5.26: A 2-layer stacked autoencoder architecture, resulting in a 4-layer full architecture.], i.e. 2 nested autoencoders that we could notate as Autoencoder^2, is illustrated in Fig. 5.26.

Chain of encoders will increasingly compress data & extract higher-level features. Stacked autoencoders, which are indeed deep networks, are therefore used for feature extraction (an example will be introduced in Sect. 6.10.7.1). They are also useful for music generation, as see in Sect. 6.4.1. This is because *innermost hidden layer*, sometimes named *bottleneck hidden layer*, provides a compact & high-level encoding (embedding) as a seed for generation (by chain of decoders).

○ 5.7. Restricted Boltzmann Machine (RBM). A *restricted Boltzmann machine* (RBM) [80] is a *generative stochastic* ANN that can learn a *probability distribution* over its set of inputs. Its name comes from fact: it is restricted (constrained) form [Which actually makes RBM practical, as opposed to general form, which besides its interest suffers from a learning scalability limitation.] of a (general) *Boltzmann machine* [81], named after *Boltzmann distribution* in statistical mechanics, which is used in its sampling function. Architectural restrictions of an RBM (Fig. 5.27: Restricted Boltzmann machine (RBM) architecture) are that:

∗ it is organized in *layers*, just as for a feedforward network or an autoencoder, & more precisely 2 layers:

· *visible* layer (analog to both input layer & output layer of an autoencoder)

· *hidden* layer (analog to hidden layer of an autoencoder)

∗ as for a standard neural network, there cannot be connections between nodes within same layer.

An RBM bears some similarity in spirit & objective to an autoencoder. However, there are some important differences:

∗ an RBM has *no output* – input also acts as output

∗ an RBM is *stochastic* (& therefore *not deterministic*, as opposed to a feedforward network or an autoencoder)

∗ an RBM is trained in an *unsupervised learning* manner, with a specific algorithm (named *contrastive divergence*, see Sect. 5.7.1), whereas an autoencoder is trained using a standard supervised learning method, with same data as input & output

∗ values manipulated are *booleans* [Although there are extensions with multinoulli (categorical) or continuous values, Sect. 5.7.3.].

RBMs became popular after Hinton designed a specific fast learning algorithm for them, named *contrastive divergence* [78], & used them for *pre-training* DNNs [46] (Sect. 5).

An RBM is an architecture dedicated to learning distributions. Moreover, it can learn efficiently from only a few examples. For musical applications, this is interesting for learning (& generating) chords, as combinatorial nature of possible notes forming a chord is large & number of examples is usually small. See an example of such an application in Sect. 6.4.2.3.

∗ 5.7.1. Training. Training an RBM has some similarity to training an autoencoder, with practical difference that, because there is no decoder part, RBM will alternate between 2 steps:

· *feedforward step*: to encode input (visible layer) into hidden layer, by making predictions about hidden layer node activations

· *backward step*: to decode/reconstruct input (visible layer), by making predictions about visible layer node activations.

Not detail here learning technique behind RBMs, see, e.g., [62, Sect. 20.2]. Note: reconstruction process is an example of *generative learning* (& not *discriminative learning*, as for training autoencoders which is based on regression) [See, e.g., aa nice introduction to generative learning (& difference with discriminative learning) in [142].].

∗ 5.7.2. Sampling. After training phase has been completed, in *generation* phase, a *sample* can be drawn from model by randomly initializing visible layer vector $\mathbf{v}$ (following a standard uniform distribution) & running *sampling* [More precisely *Gibbs sampling (GS)*, see [106]. Sampling will be introduced in Sect. 6.4.2.1.] until convergence. To this end, hidden nodes & visible nodes are alternately updated (as during training phase).

In practice, convergence is reached when energy stabilizes. *Energy* of a *configuration* (pair of visible & hidden layers) is expressed [For more details, see, e.g., [62, Sect. 16.2.4].] in

$$E(\mathbf{v},\mathbf{h}) = -\mathbf{a}^\top\mathbf{v} - \mathbf{b}^\top\mathbf{h} - \mathbf{v}^\top W\mathbf{h},$$

where

        - $\mathbf{v}, \mathbf{h}$, resp.: column vectors representing visible & hidden layers
        - W: matrix of weights associated with connections between visible & hidden nodes
        - $\mathbf{a}, \mathbf{b}$, resp.: column vectors representing bias weights for visible & hidden nodes, with $\mathbf{a}^\top, \mathbf{b}^\top$ being their respective transpositions into row vectors
        - $\mathbf{v}^\top$: transposition of $\mathbf{v}$ into a row vector.
    - 5.7.3. Types of Variables. Note: there are actually 3 possibilities for nature of RBM variables (units, visible or hidden):
        - *Boolean* or *Bernoulli*: this is case of standard RBMs, in which units (visible & hidden) are Boolean, with a *Bernoulli distribution* (see [62, Sect. 3.9.2])
        - *multinoulli*: an extension with *multinoulli* units [As explained by Goodfellow et al. in [62, Sect. 3.9.2]: ""Multinoulli" is a term that was recently coined by GUSTAVO LACERDO & popularized by MURPHY in [139]. Multinoulli distribution is a special case of multinomial distribution. A multinomial distribution: distribution over vectors in $\{0, \dots, n\}^k$ representing how many times each of k categories is visited when n samples are drawn from a multinoulli distribution. Many texts use term "multinomial" to refer to multinoulli distributions without clarifying that they refer only to $n = 1$ case."], i.e., with > 2 possible discrete values
        - *continuous*: another extension with continuous units, taking arbitrary real values (usually within $[0, 1]$ range). E.g.: C-RBM architecture analyzed in Sect. 6.10.5.1.

- 5.8. RNN. A RNN is a feedforward neural network extended with *recurrent connections* in order to learn series of items (e.g., a melody as a sequence of notes). Input of RNN is an element x_t [This x_t notation – or sometimes s_t to stress fact: it is a sequence – is very common but unfortunately introduces possible confusion with notation of x_i as ith input variable. Context – recurrent vs. nonrecurrent network – usually helps to discriminate, as well as use of letter t (for time) as index. An example of an exception: RNN-RBM system analyzed in Sect. 6.9.1, which uses $x^{(t)}$ notation.] of sequence, where t represents *index* or *time*, & expected output is next element x_{t+1}. I.e., RNN will be trained to predict next element of a sequence.

  In order to do so, output of hidden layer *reenters* itself as an additional input (with a specific corresponding weight matrix). This way, RNN can learn, not only based on *current* item but also on its *previous* own state, & thus, recursively, on whole of prev sequence. Therefore, an RNN can learn sequences, notably *temporal sequences*, as in case of musical content.

  An example of RNN (with 2 hidden layers) is shown in Fig. 5.28: RNN (folded). Recurrent connections are signaled with a solid square, in order to distinguish them from standard connections [Actually, there are some variations of this basic architecture, depending on exact nature & location of recurrent connections. Most standard case is a recurrent connection for each hidden unit, as shown in Fig. 5.28. But there are some other cases, see e.g. in [62, Sect. 10.2]. An example of a music generation architecture with recurrent connections from output to a special context input is introduced in Sect. 6.8.2.]. Unfolded version of visual representation is in Fig. 5.29: RNN (unfolded)., with a new diagonal axis representing time dimension, in order to illustrate prev step value of each layer (in thinner & lighter color). Note: as for standard connections, recurrent connections fully connect (with a specific weight matrix) all nodes corresponding to prev step nodes to nodes corresponding to current step, as illustrated in Fig. 5.30: Standard connections. vs. recurrent connections (unfolded).

  An RNN can learn a probability distribution over a sequence by being trained to predict next element at time step t in a sequence as being conditional probability distribution $P(s_t|s_{t-1}, \dots, s_1)$, also notated as $P(s_t|s_{<t})$, i.e., probability distribution $P(s_t)$ given all prev elements generated $s_1, s_2, \dots, s_{t-1}$. In summary, recurrent networks (RNNs) are good at learning sequences & therefore are routinely used for natural text processing & for music generation.

    - 5.8.1. Visual Representation. A more frequent visual representation for an RNN is actually showing flow upwards & time rightwards, see folded version (of an RNN with only 1 hidden layer) in Fig. 5.31: RNN (folded) & unfolded version in Fig. 5.32: RNN (unfolded), with h_t being value of hidden layer at step t, & x_t, y_t being values of input & output at step t.
    - 5.8.2. Training. A recurrent network is not trained in exactly same manner as a feedforward network. Idea: present an example element of a sequence (e.g., a note within a melody) as input x_t & next element of sequence (next note) x_{t+1} as output y_t. This will train recurrent network to predict next element of sequence. In practice, an RNN is rarely trained element by element but with a sequence as an input & same sequence shifted left by 1 step/item as output. See an example in Fig. 5.33 [End of sequence is marked by a special symbol.]. Therefore, recurrent network will learn to predict [Pictured as dashed arrows.] next element for all successive elements of sequence.

      Backpropagation algorithm to compute gradients for feedforward networks, introduced in Sect. 5.5.8, has been extended into a *backpropagation through time* (BPTT) algorithm for recurrent networks. Intuition is in unfolding RNN through time & considering an ordered sequence of input-output pairs, but with every unfolded copy of network sharing same parameters, & then applying standard backpropagation algorithm. More details may be found, e.g., in [62, Sect. 10.2.2].

      Note: a RNN has usually an output layer identical to its input layer [RNNs are actually more general & there are actually some rare cases of an RNN with an arbitrary output different from input (as for a feedforward network). E.g.: a RNN-based architecture to generate a chord-based accompaniment, analyzed in Sect. 6.8.3. As KARPATHY puts it in [97]: "Depending on your background you might be wondering: What makes Recurrent Networks so special? Aa glaring limitation of Vanilla Neural Networks (& also Convolution Networks): their API is too constrained: they accept a fixed-sized vector as input (e.g. an image) & produce a fixed-size vector as output (e.g. probabilities of different classes). Not only: These models perform this mapping using a fixed amount of computational steps (e.g. number of layers in model). Core reason: recurrent nets are more exiting: they allow us to operate over sequences of vectors: Sequences in input, output, or in most general case both."], as a recurrent network predicts next item, which will be used iteratively as next input in a recursive way in order to produce a sequence.

Note: training a recurrent network is usually considered as a case of supervised learning as, for each item, next item is presented as expected prediction, although it is not an additional label information (effective value or class to be predicted) but only recurrent information about next item (*intrinsically* present within a sequence).

∗ 5.8.3. LSTM. Recurrent networks suffered from a training problem caused by difficulty of estimating gradients because in backpropagation through time recurrence brings repetitive multiplications, & could thus lead to over *amplify* or *minimize* effects [This has been coined as *vanishing or exploding gradient problem* & also as *challenge of long-term dependencies* (see, e.g., [62, Sect. 10.7]).]. This problem has been addressed & resolved by *long short-term memory* (LSTM) architecture, proposed by Hochreiter & Schmidhuber in 1997 [82]. As solution has been quite effective, LSTM has become de facto standard for recurrent networks [Although, there are a few subsequent but similar proposals, e.g. *gated recurrent units* (GRUs). See a comparative analysis of LSTM & GRU in [24].].

Idea behind LSTM: secure information in memory *cells*, within a *block* [Cells within same block *share* input, output, & forget gates. Therefore, although each cell might hold a different value in its memory, all cell memories within a block are read, written or erased *all at once* [82].], protected from standard data flow of recurrent network. Decisions about *writing* to, *reading* from & *forgetting (erasing)* values of cells within a block are performed by opening or closing of *gates* & are expressed at a distinct control level (*meta-level*), while being learnt during training process. Therefore, each gate is modulated by a *weight* parameter, & thus is suitable for backpropagation & standard training process. I.e., each LSTM block learns how to maintain its memory as a function of its input in order to minimize loss.

See a conceptual view of an LSTM cell in Fig. 5.34: LSTM architecture (conceptual). Not further detail here inner mechanism of an LSTM cell (& block) because may consider it here as a *black box* (refer to original article [82]).

Note: a more general model of memory with access customized through training has recently been proposed: neural Turing machines (NTM) [65]. In this model, memory is global & has *read & write operations* with differentiable controls, & thus is subject to learning through backpropagation. Memory to be accessed, specified by *location* or by *content*, is controlled via an *attention* mechanism.

∗ 5.8.4. Attention Mechanism. Motivation for an *attention mechanism* has been inspired by human visual system ability to efficiently track & recognize objects by focusing its *attention.* It has therefore been 1st introduced into neural network architectures for image recognition, as e.g. for object tracking [34]. It has then been adapted to recurrent architectures for NLP (& more specifically for translation tasks) & has showed significant improvement for management of long-term dependencies.

Idea of an attention mechanism: focus at each time step on some specific elements of input sequence. This is modeled by weighted connections onto sequence elements (or onto sequence of hidden units). Therefore it is differentiable & subject to backpropagation-based learning at a meta-level, as with LSTM gate control described in prev sect. For more details, see, e.g., [62, Sect. 12.4.5.1].

Interestingly, a novel architecture for translation of sequences, named *Transformer* & which is *solely* based on an attention mechanism [Architecture introduces *multi-head attention* which allows model to jointly attend to information from different representation subspaces at different positions [197].], has recently being proposed & shows promising results [197]. Its very recent application to music generation is shortly discussed in Sect. 8.2.

○ 5.9. Convolutional Architectural Pattern. *Convolutional neural network* (CNN or ConvNet) architectures for DL have become common place for image applications. Concept was originally inspired by both a model of human vision & *convolution* mathematical operator [In mathematics, a convolution is a mathematical operation on 2 functions sharing same domain (usually noted $f * g$) that produces a 3rd function which is integral (or sum in discrete case – case of images made of pixels) of pointwise multiplication fo 2 functions varying within domain in an opposing way. In case of a continuous domain [*low high*]:

$$(f * g)(x) = \int_{\mathrm{low}}^{\mathrm{high}} f(x-t)g(t)\,\mathrm{d}t.$$

In discrete case:

$$(f * g)(n) = \sum_{m=\mathrm{low}}^{\mathrm{high}} f(n-m)g(m).$$

]. It has been carefully adapted to neural networks & improved by LeCun, at 1st for handwritten character & object recognition [110]. This resulted in efficient & accurate architectures for pattern recognition, exploiting spatial local *correlation* present in natural images.

∗ 5.9.1. Principles. Basic idea [Inspired by nice intuitive explanation provided by Karn in [96]. For more technical details see, e.g., [116] or [62, Chap. 9].] is to

- *slide* a matrix (named a *filter*, a *kernel* or a *feature detector*) through entire image (seen as input matrix)
- for each mapping position:
  1. compute dot product of filter with each mapped portion of image &
  2. then sum up all elements of resulting matrix
- resulting in a new matrix (composed of different sums for each sliding/mapping position), named *convolved feature*, or also *feature map.*

Size of feature map is controlled by 3 hyperparameters:

- *depth*: number of filters used

- · *stride*: number of pixels by which slide filter matrix over input matrix
- · *0-padding*: padding of input matrix with 0s around its border [0-padding allows mapping of filter up to borders of image. It also avoids shrinking representation, which otherwise would be problematic when using multiple consecutive convolutional layers [62, Sect. 9.5].].

An example is illustrated in Fig. 5.35: Convolution, filter & feature map. Inspired by Karn's DS blog post [96]. with some simple settings: depth = 1, stride = 1 & no 0-padding. Various filter matrices can be used with different objectives, e.g. detection of different features (e.g., edges or curves) or other operations e.g. sharpening or blurring.

Parameter sharing used by convolution (because of shared fixed filter) brings important property of *equivariance* to translation, i.e. a motif in an image can be detected independently of its location [62, Chap. 9].

- ∗ 5.9.2. Stages. A convolution usually consists of 3 successive *stages*:
  - · a *convolution stage*, as described in Sect. 5.9.1
  - · a *nonlinear rectification stage*, sometimes named detector stage, which applies a nonlinear operation, usually ReLU
  - · a *pooling stage*, also named *subsampling*, to reduce dimensionality.
- ∗ 5.9.3. Pooling. Motivation for *pooling*: reduce dimensionality of each feature map while retaining significant information. Operations used for pooling are, e.g., max, average & sum. In addition to reducing dimensionality of data, pooling brings important property of *invariance* to small transformations, distortions & translations in input image. This provides an overall robustness to processing [96]. Like convolution, pooling has hyperparameters to control process. A simple example of max pooling with stride = 2 is illustrated in Fig. 5.36: Pooling. Inspired by Karn's DS blog post [96].
- ∗ 5.9.4. Multilayer Convolutional Architecture. A typical example of a convolutional architecture with successive layers – each one including 3 stages of convolution, nonlinearity & pooling – is illustrated in Fig. 5.37: Convolutional DNN architecture. Inspired by Karn's DS blog post [96]. Final layer is a fully connected layer, like in standard feedforward networks, & typically ends up in a softmax in order to classify image types.

  Note: a convolution is an *architectural pattern*, as it may be applied internally to almost any architecture listed.
- ∗ 5.9.5. Convolution over Time. For musical applications, it could be interesting to apply convolutions to *time dimension* [This approach is actually basis for *time-delay neural networks* [107].], in order to model temporally invariant motives. Therefore, convolution operation will share parameters across time [62, p. 374], like for RNNs [Indeed, RNNs are invariant in time, as remarked in [93].]. However, sharing of parameters is *shallow*, as it applies only to a small number of temporal neighboring members of input, in contrast to RNNs that share parameters in a *deep* way, for *all* time steps. RNNs are indeed much more frequent than convolutional networks for musical applications.

  I.e., have noticed recent occurrence of some convolutional architectures as an alternative to RNN architectures, following pioneering WaveNet architecture for audio [193], described in Sect. 6.10.3.2. WaveNet presents a stack of causal convolutional layers, somewhat analogous to recurrent layers. Another example: C-RBM architecture, described in Sect. 6.10.5.1.

  If consider *pitch dimension*, in most cases pitch intervals are not considered invariants, & thus convolutions should not a *priori* apply to pitch dimension [An example: Johnson's architecture [93], analyzed in Sect. 6.9.2, which explicitly looks for invariance in pitch (although this seems to be a rare choice) & accordingly uses an RNN over pitch dimension.].

  This issue of convolution vs. recurrence (recurrent networks) for musical applications will be further discussed in Sect. 8.2.

○ 5.10. Conditioning Architectural Pattern. Idea of a *conditioning* (sometimes also named *conditional*) architecture: parameterize architecture based on some extra *conditioning* information, which could be arbitrary, e.g., a class label or data from other modalities. Objective: have some control over data generation process. Examples of conditioning information:

- ∗ a *bass line* or a *beat structure* in rhythm generation system to be described in Sect. 6.10.3.1
- ∗ a *chord progression* in MidiNet system to be described in Sect. 6.10.3.3
- ∗ some *positional constraints on notes* in Anticipation-RNN system to be described in Sect. 6.10.3.5
- ∗ a *musical genre* or an *instrument* in WaveNet system to be described in Sect. 6.10.3.2.

In practice, conditioning information is usually fed into architecture as an additional & specific input layer, shown in purple in Fig. 5.38: Conditioning architecture.

Conditioning layer could be

- ∗ a simple input layer. E.g.: a tag specifying a musical genre or an instrument in WaveNet system to be described in Sect. 6.10.3.2; or
- ∗ some output of some architecture, being
  - · same architecture, as a way to condition architecture on some history [This is close in spirit to a recurrent architecture (RNN).]. E.g.: MidiNet system to be described in Sect. 6.10.3.3, in which history information from prev measure(s) is injected back into architecture; or
  - · another architecture. E.g.: DeepJ system to be described in Sect. 6.10.3.4, in which 2 successive transformation layers of a style tag produce an embedding used as conditioning input.

  In case of *conditioning* a time-invariant architecture – recurrent or convolutional over time – there are 2 options
  - · *global conditioning*: if conditioning input is shared for all time steps
  - · *local conditioning*: if conditioning input is specific to each time step.

WaveNet architecture, which is convolutional over time (Sect. 5.9.5), offers 2 options, as analyzed in Sect. 6.10.3.2.

- 5.11. Generative Adversarial Networks (GAN) Architectural Pattern. A significant conceptual & technical innovation was introduced in 2014 by Goodfellow et al. with concept of *generative adversarial networks* (GAN) [63]. Idea: train simultaneously 2 neural networks [In original version, 2 feedforward networks are used. But other networks may be used, e.g., recurrent networks in C-RNN-GAN architecture (Sect. 6.10.2.4) & convolutional feedforward networks in MidiNet architecture (Sect. 6.10.3.3).], as illustrated in Fig. 5.39: Generative adversarial networks (GAN) architecture [157]:

  * a *generative model* (or *generator*) G, whose objective: transform a random noise vector into a synthetic (faked) *sample*, which resembles real samples drawn from a distribution of real images
  * a *discriminative model* (or *discriminator*) D, which estimates probability that a sample came from real data rather than from generator G [In some ways, a GAN represents an automated Turing test setting, with discriminator being evaluator & generator being hidden actor.].

  This corresponds to a *minimax* 2-player game, with 1 unique (final) solution [It corresponds to Nash equilibrium of game. In game theory, intuition of a Nash equilibrium is a solution where no player can benefit by changing strategies while other plays keep theirs unchanged, see, e.g., [144].]: G recovers training data distribution & D outputs $\frac{1}{2}$ everywhere. Generator is then able to produce user-appealing synthetic samples from noise vectors. Discriminator may then be discarded.

  Minimax relationship is defined in (5.30)

$$\min_G \max_D V(G,D) = \mathbb{E}_{x\sim P_{\text{Data}}}[\log D(x)] + \mathbb{E}_{z\sim P_z(z)}[\log(1-D(G(z)))].$$

  * $D(x)$ represents probability that x came from real data (i.e., *correct* estimation by D)
  * $\mathbb{E}_{x\sim P_{\text{Data}}}[\log D(x)]$: expectation of $\log D(x)$ w.r.t. x being drawn from real data.

  It is thus D's objective to estimate correctly *real data*, i.e. maximize $\mathbb{E}_{x\sim P_{\text{Data}}}[\log D(x)]$ term.

  * $D(G(z))$ represents probability that $G(z)$ came from real data (i.e., *uncorrect* estimation by D)
  * $1-D(G(z))$ represents probability that $G(z)$ did not come from real data, i.e., that it was generated by G (i.e., *correct* estimation by D)
  * $\mathbb{E}_{z\sim P_z(z)}[\log(1-D(G(z)))]$: expectation of $\log(1-D(G(z)))$ w.r.t. $G(z)$ being produced by G from z random noise.

  It is thus also D's objective to estimate correctly *synthetic data*, i.e., maximize $\mathbb{E}_{z\sim P_z(z)}[\log(1-D(G(z)))]$ term.

  In summary, it is D's objective to estimate correctly both *real data & synthetic data* & thus to maximize both $\mathbb{E}_{x\sim P_{\text{Data}}}[\log D(x)]$ & $\mathbb{E}_{z\sim P_z(z)}[\log(1-D(G(z)))]$ terms, i.e. to maximize $V(G,D)$. On opposite side, G's objective: minimize $V(G,D)$. Actual training is organized with successive turns between training of generator & training of discriminator.

  1 of initial motivations for GAN was for classification tasks to prevent adversaries from manipulating deep networks to force misclassification of inputs (this vulnerability is analyzed in detail in [182]). However, from perspective of content generation (which is our interest), GAN improves generation of samples, which became hard to distinguish from actual corpus examples.

  To generate music, random noise is used as an input to generator G, whose goal: transform random noises into objective, e.g., melodies [In that respect, generation from a GAN has some similarity with generation by decoding hidden layer variables of a variational autoencoder (Sect. 5.6.2), as in both cases generation is done from latent variables. An important difference: by construction, a variational encoder is representative of whole dataset that it has learnt, i.e., for any example in dataset, there is at least 1 setting of latent variables which causes model to generate sth very similar to that example [37]. A GAN does not offer such guarantee & does not offer a smooth generation control interface over latent space (by, e.g., interpolation or attribute arithmetics, see Sect. 5.6.2), but it can usually generate better quality (better resolution) images than a variational autoencoder [124]. Note: resolution limitation for a VAE may be a problem too for audio generation of music, but it appears *a priori* less a direct concern in case of symbolic generation of music.]. An example of us of GAN for generating music: MidiNet system, described in Sect. 6.10.3.3.

  * 5.11.1. Challenges. Training based on a minimax objective is known to be challenging to optimize [210], with a risk of nonconverging oscillations. Thus, careful selection of model & its hyperparameters is important [62, p. 701]. There are also some newer techniques, e.g. *feature matching* [Feature matching changes objective for generator (& accordingly its cost function) to minimize statistical difference between features of real data & generated samples, see more details in [168].], among others, to improve training [168].

    A recent proposed alternative both to GANs & to autoencoders is *generative latent optimization* (GLO) [9]. It is an approach to train a generator without need to learn a discriminator, by learning a mapping from noise vectors to images. GLO can thus be viewed both as an encoder-less autoencoder, & as a discriminator-less GAN. It can also be used, as for a VAE (variational autoencoder) introduced in Sect. 5.6.2, to control generation by exploring latent space. GLO has been tested on images but not yet on music & needs more evaluation.

- 5.12. Reinforcement Learning. *Reinforcement learning (RL)* may appear at 1st glance to be outside of our interest in DL architectures, as it has distinct objectives & models. However, 2 approaches have recently been combined. 1st move, in 2013, was to use DL architectures to efficiently implement reinforcement learning techniques, resulting in *deep reinforcement learning* [133]. 2nd move, in 2016, is directly related to our concerns, as it explored use of reinforcement learning to control music generation, resulting in RL-Tuner architecture [92] to be described in Sect. 6.10.6.1.

  Start with a reminder of basic concepts of reinforcement learning, illustrated in Fig. 5.40: Reinforcement learning – conceptual model [39]

- an *agent* within an *environment* sequentially selects & performs *actions* in an environment
- where each action performed brings it to a new *state*
- agent receives a *reward (reinforcement signal)*, which represents *fitness* of action to environment (current situation)
- objective of agent being to learn a near optimal *policy* (sequence of actions) in order to maximize its *cumulated rewards* (named its *gain*).

Note: agent does not know beforehand model of environment & reward, thus it needs to balance between *exploring* to learn more & *exploiting* (what it has learned) in order to improve its gain – this is *exploration exploitation dilemma.*

There are many approaches & algorithms for reinforcement learning (for a more detailed presentation, refer, e.g., [95]). Among them, *Q-learning* [202] turned out to be a relatively simple & efficient method, & thus is widely used. Name comes from objective to learn (estimate) Q function $Q^*(s,a)$, which represents expected gain for a given pair (s,a), where s is a state & a an action, for an agent choosing actions optimally (i.e., by following optimal policy π^*). Agent will manage a table, called *Q-table*, with values corresponding to all possible pairs. As agent explores environment, table is incrementally updated, with estimates becoming more accurate.

A recent combination of reinforcement learning (more specifically Q-learning) & DL, named *deep reinforcement learning*, has been proposed [133] in order to make learning more efficient. As Q-table could be huge [Because of high combinatorial natural when number of possible states & possible actions is huge.], idea: use a DNN in order to approximate expected values of Q-table through learning of many replaced experiences.

A further optimization, named *double Q-learning* [195] *decouples action selection* from *evaluation*, in order to avoid value overestimation. Task of 1st network, named Target Q-Network: estimate gain Q, while task of Q-Network: select next action.

Reinforcement learning appears to be a promising approach for incremental adaptation of music to be generated, e.g., based on *feedback* from listeners (this issue is addressed in Sect. 6.16). Meanwhile, a significant move has been made in using reinforcement learning to inject some control into generation of music by DL architectures, through reward mechanism, as described in Sect. 6.10.6.

- 5.13. Compound Architectures. Often *compound* architectures are used. Some cases are *homogeneous* compound architectures, combining various instances of same architecture, e.g., a stacked autoencoder (see Sect. 5.6.3), & most cases are *heterogeneous* compound architectures, combining various types of architectures, e.g., an RNN Encoder-Decoder which combines an RNN & autoencoder, see Sect. 5.13.3.
  - 5.13.1. Composition Types. From an architectural point of view, various types of composition [We are taking inspiration from concepts & terminology in programming languages & software architectures [171], e.g. *refinement, instantiation, nesting & pattern* [54].] may be used:
    - *Composition*: at least 2 architectures, of same type or of different types, are combined, e.g.
      1. a bidirectional RNN (Sect. 5.13.2) combining 2 RNNs, forward & backward in time
      2. RNN-RBM architecture (Sect. 5.13.5) combining an RNN architecture & an RBM architecture.
    - *Refinement*: 1 architecture is *refined & specialized* through some additional constraint(s0 [Both cases are refinements of standard autoencoder architecture through additional constraints, in practice adding an extra term onto cost function.]), e.g.
      1. a sparse autoencoder architecture (Sect. 5.6.1)
      2. a variational autoencoder (VAE) architecture (Sect. 5.6.2).
    - *Nested*: 1 architecture is nested into other one, e.g.
      1. a stacked autoencoder architecture (Sect. 5.6.3)
      2. RNN Encoder-Decoder architecture (Sect. 5.13.3), where 2 RNN architectures are nested within encoder & decoder parts of an autoencoder, which could therefore also notate as Autoencoder(RNN, RNN).
    - *Pattern instantiation*: an architectural pattern is instantiated onto a given architecture(s), e.g.
      1. C-RBM architecture (Sect. 6.58) that instantiates convolutional architectural pattern onto an RBM architecture, which could notate as Convolutional(RBM)
      2. C-RNN-GAN architecture (Sect. 6.10.2.4), where GAN architectural pattern is instantiated onto an RNN architecture, which we could notate as GAN(RNN, RNN)
      3. Anticipation-RNN architecture (Sect. 6.10.3.5) that instantiates conditioning architectural pattern onto an RNN with output of another RNN as conditioning input, which could notate as Conditioning(RNN, RNN).
  - 5.13.2. Bidirectional RNN. Bidirectional RNNs were introduced by Schuster & Paliwal [170] to handle case when prediction depends not only on prev elements but also on *next* elements, as e.g. with speech recognition. In practice, a bidirectional RNN combines [See more details in [170].]
    - a 1st RNN that moves *forward* through time & begins from *start* of sequence
    - a 2nd symmetric RNN that moves *backward* through time & begins from *end* of sequence.

    Output y_t of bidirectional RNN at step t combines
    - output h_t^f at step t of hidden layer of "forward RNN"
    - output h_{N-t+1}^b at step $N-t+1$ of hidden layer of "backward RNN".

    As illustration is in Fig. 5.41: Bidirectional RNN architecture. Examples of use:
    - BLSTM architecture (Sect. 6.8.3)

- · C-RNN-GAN architecture (Sect. 6.10.2.4) that encapsulates a bidirectional RNN into discriminator of a GAN
- · MusicVAE architecture (Sect. 6.12.1) that encapsulates a bidirectional RNN into encoder of a VAE (variational autoencoder).

∗ 5.13.3. RNN Encoder-Decoder. Idea of encapsulating 2 identical recurrent networks (RNNs) into an autoencoder, named *RNN Encoder-Decoder* [Could also notate it as Autoencoder(RNN, RNN).], was initially proposed in [19] as a technique to encode a variable length sequence learnt by a recurrent network into another variable length sequence produced by another recurrent network [This is named *sequence-to-sequence learning* [180].]. Motivation & application target is translation from 1 language to another, resulting inn sentences of possibility different lengths.

Idea: use a fixed-length vector representation as a *pivot* representation between an encoder & a decoder architecture, see illustration in Fig. 5.42: RNN Encoder-Decoder architecture. Inspired from [19]. Hidden layer(s) h_t^e of encoder will act as a memory which

- · iteratively accumulates information about some input sequence of length N, while reading its successive x_t elements [End of sequence is marked by a special symbol, as when training an RNN, Sect. 5.8.2.], resulting in a final state h_N^e
- · which is passed to decoder as summary c of whole input sequence
- · decoder then iteratively generates output sequence of length M, by predicting next item y_t given its hidden state h_t^d & summary (as a conditioning additional input) c [As noted by Goodfellow et al. in [62, Section 10.4], an alternative: use summary c only to initialize initial hidden state of decoder h_0^d, i.e., e.g., strategy chosen in GLSR-VAE architecture described in Sect. 6.10.2.3.]

2 components of RNN Encoder-Decoder are jointly trained to minimize cross-entropy between input & output. See in Fig. 5.43: RNN Encoder-Decoder audio Word2Vec architecture [25]. example of Audio Word2Vec architecture for processing audio phonetic structures [25].

1 limitation of RNN Encoder-Decoder approach: difficulty for summary to memorize very long sequences [In text translation applications, sentences have a limited size.]. 2 possible directions:

- · using an *attention mechanism* [Introduced in Sect. 5.8.4.]
- · using a *hierarchical model*, as proposed in MusicVAE architecture, to be introduce in Sect. 6.12.1.

∗ 5.13.4. Variational RNN Encoder-Decoder. An interesting development is a *variational* version of RNN Encoder-Decoder, i.e., a variational autoencoder (VAE) encapsulating 2 RNNs. Could notate it as Variational(Autoencoder(RNN, RNN)). Objective: combine

- · *variational* property of VAE for controlling generation (Sect. 5.6.2)
- · *sequence* generation property of RNN.

Examples of its application to music generation are introduced in Sect.6.10.2.3.

∗ 5.13.5. Polyphonic Recurrent Networks. RNN-RBM architecture, introduced in Sect. 6.9.1, combines an RBM architecture & a recurrent (LSTM) architecture by *coupling* them to associate vertical perspective (simultaneous notes) with horizontal perspective (temporal sequences of notes) of a polyphony to be generated.

∗ 5.13.6. Further Compound Architectures. Possible to further combine architecture that are already compound, e.g.

- · WaveNet architecture (Sect. 6.10.3.2), which is a conditioning convolutional feedforward architecture with some tag as conditioning input, which could notate as Conditioning(Convolutional(Feedforward), Tag)
- · VRASH architecture (Sect. 6.10.3.6), which is a variational autoencoder encapsulating RNNs with decoder being conditioned on history, which could notate as Variational(Autoencoder(RNN, Conditioning(RNN, History))).

There are also some more specific (ad hoc) compound architectures, e.g.

- · Johnson's Hexahedria architecture (Sect. 6.9.2), which combines 2 layers recurrent on time dimension with 2 other layers recurrent on pitch dimension, as an integrated alternative to RNN-RBM architecture
- · DeepBach architecture (Sect. 6.14.2), which combines 2 feedforward architectures with 2 recurrent architectures.

∗ 5.13.7. Limits of Composition. There is a natural tendency to explore possible combinations of different architectures with hope of combining their respective features & merits. An example of a sophisticated compound architecture: VRASH architecture (Sect. 6.10.3.6), which combines

- · variational autoencoder
- · recurrent networks
- · conditioning (on decoder).

However, note:

- · not all combinations make sense. E.g., recurrence & convolution over time dimension would complete, as discussed in Sect. 5.9
- · there is no guarantee that combining a maximal variety of types will make a sound & accurate architecture [As in case of a good cook, whose aim is not to simply mix *all* possible ingredients but to discover original successful combinations.].

See in Chap. 6: an important additional design dimension is *strategy*, which governs how an architecture will process representations in order to reach a given objective with some expected properties (*challenges*).

- 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 1 chiến lược đơn giản, dễ hiểu, sử dụng nhiệm vụ dự đoán cơ bản của mạng nơ-ron để tạo ra phần đệm cho 1 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×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×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×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 ×16 time steps ×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 [But not impossible to achieve, regarding recent achievements made on audio source separation through DL techniques, as has been pointed out in Sect. 4.10.3.] 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. In [173], SIMON & OORE present their architecture & methodology named Performance RNN – an LSTM-based RNN architecture. 1 of specificities is in dataset characteristics, as corpus is composed of recorded human performances, with records of exact timing as well as dynamics for each note played. Corpus used is Yamaha e-Piano Competition dataset, whose participants MIDI performance records are made available to public [209]. It captures > 1400 performances by skilled pianists. To create additional training examples, some time stretching (up to 5% faster or slower) as well as some transposition (up to a major 3rd) is applied.

    Representation is adapted to objective. At 1st look, it resembles a piano roll with MIDI note numbers but it is actually a bit different. Each time slice is a multi-1-hot vector of possible values for each of following possible events:

    - start of a new note – with 128 possible values (MIDI pitches).
    - end of a note – with 128 possible values (MIDI pitches)
    - time shift – with 100 possible values (from 10 milisecs to 1 sec)
    - dynamics – with 32 possible values (128 MIDI velocities quantized into 32 bins [See description of binning transformation in Sect. 4.11.]).

    An example of a performance representation is shown in Fig. 6.17: Example of Performance RNN representation [173].

    Some control is made available to user, referred to as *temperature*, which controls randomness of generated events in following way:

    - a temperature of 1.0 uses exact distribution predicted
    - a value < 1.0 reduces randomness & thus increases repetition of patterns
    - a larger value increases randomness & decreases repetition of patterns.

    Examples are available on web page [173]. Performance RNN is summarized in Table 6.10: Performance RNN summary.

- 6.8. RNN & Iterative Feedforward Revisited. As saw in prev examples, iterative feedforward strategy is based on idea of RNN architecture to iteratively generate successive item of a sequence. It looks like a RNN architecture & iterative feedforward strategy are strongly coupled. Indeed, almost all RNN-based systems use an iterative feedforward strategy & recursively reenter output produced (next time step generated) into input. But will introduce in this sect some exceptions.

  - 6.8.1. #1 Example: Time-Windowed Melody Symbolic Music Generation System. Experiments by TODD in [189] were 1 of very 1st attempts (in 1989) at exploring how to use ANNs to generate music. Although architectures he proposed are not directly used nowadays, his experiments & discussion were pioneering & are still an important source of information.

    Todd's objective was to generate a monophonic melody in some iterative way. He has experimented with different choices for representing notes (Sect. 4.5.3) & durations, but finally had decided to use a conventional pitch note representation with a 1-hot encoding & a time step temporal scope approach. Time step is set at duration of an 8th note. In most of experiments, input melodies used for training are 34 time steps long (i.e., 4 measures & a half long), padded at end with rests. A note begin is represented with a specific token & is encoded as an additional value encoding node (Sects. 4.9.1 & 4.11.7). Rests are not encoded explicitly but as absence of a note, i.e., as note 1-hot encoding being all filled with null values (Sect. 4.11.7).

1st experiment is what author named Time-Windowed architecture, where a sliding window of successive time-periods of fixed size is considered. In practice this sliding window of a melody segment is 1 measure long, i.e. 8 time steps. Its representation may be considered as a piano roll, like in MiniBach architecture (Sect. 6.2.2), with successive 1-hot encodings of notes for 8 successive time steps, notated as 1-hot×8.

Architecture is a feedforward network (& not an RNN), with a melody segment as its input, next melody segment as its output & with a single hidden layer. Generation is conducted iteratively (& recursively), melody segment by melody segment. Architecture is illustrated in Fig. 6.18: Time-Windowed architecture. Inspired from [189].

For each time step of melody segment, predicted note is the one with highest probability. Because of 0-hot encoding of a rest (i.e. as all values being null), there is an ambiguity between case of every possible note has a low probability & case of a rest (Sect. 4.11.7). For that reason, a probability threshold is introduced, namely 0.5. Thus, predicted note is the one with highest probability if it is > 0.5 & is a rest otherwise.

Network is trained in a supervised way by presenting a melody segment as an input & its corresponding next segment as output, & repeating this for various segments. Note: as architecture is not recurrent, although network will learn pairwise correlations between 2 successive melody segments [In that respect, Time-Windowed model is analog to an order 1 Markov model (considering only prev state) at level of a melody measure.], there is no explicit memory for learning long term correlations e.g. in case of recurrent network architecture. Thus, although author does not show a comparison with its next experiment (see next sect), architecture appears to have a low ability to learn long term correlations. Time-Windowed architecture is summarized in Table 6.11: Time-Windowed summary.

- 6.8.2. #2 Example: Sequential Melody Symbolic Music Generation System. In [189], Todd proposed another architecture, named Sequential, as notes are generated in a sequence. It is illustrated in Fig. 6.19: Sequential architecture. Inspired from [189].

  Input layer is divided in 2 parts, named *context & plan*. Context is actual memory (of melody generated so far) & consists in units corresponding to each note (D_4 to C_6), plus a unit about note begin information (notated as "nb" in Fig. 6.19). Therefore, it receives information from output layer which produces next note, with a reentering connection corresponding to each unit [Note: output layer is isomorphic to context layer.]. In addition, as Todd explains it: "A memory of more than just single prev output (note) is kept by having a self-feedback connection on each individual context unit." [This is a peculiar characteristic of this architecture, as in a standard recurrent network architecture recurrent connections are encapsulated within hidden layer (see Fig. 5.30: Standard connections vs. recurrent connections (unfolded). & Fig. 5.34: LSTM architecture (conceptual).). Argument by Todd in [189]: context units are more interpretabel than hidden units: "Since hidden units typically compute some complicated, often uninterpretable function of their inputs, memory kept in context units will likely also be uninterpretable. This is in contrast to [this] design, where, as described earlier, each context unit keeps a memory of its corresponding output unit, which is interpretable."]

  Plan represents a melody (among many): network has learnt. Todd has experimented with various encodings, 1-hot or distributed (through a many-hot embedding).

  Training is done by selecting a plan (melody) to be learnt. Activations of context units are initialized to 0 in order to begin with a clean empty context. Network is then feedforward & its output, corresponding to 1st time step note, is compared to 1st tie step note of melody to be learnt, resulting in adjustment of weights. Output values [Actually, as an optimization, Todd proposes in following of his description to pass back target values & not output values.] are passed back to current context. & then, network is feedforwarded again, leading to next time step note, again compared to melody target, & so on until last time step of melody. This process is then repeated for various plans (melodies).

  Generation of new melodies is conducted by feedforwarding network with a new plan embedding, corresponding to a new melody (not part of training plans/melodies). Activations of context units are initialized to 0 in order to begin with a clean empty context. Generation takes place iteratively, time step after time step. Note: as opposed to most cases of iterative feedforward strategy (Sect. 6.5.1), in which output is explicitly reentered (recursively) into input of architecture, in Todd's Sequential architecture reentrance is implicit because of specific nature of recurrent connections: output is reentered into context units while input – plan melody – is constant.

  After having trained network on a plan melody, various melodies may be generated by extrapolation by inputting new plans, as shown in Fig. 6.20: Examples of melodies generated by Sequential architecture. (o) Original plan melody learnt. (e_1, e_2) Melodies generated by extrapolating from a new plan melody. Inspired from [189]. A repeat sign **:** indicates when network output goes into a fixed loop.

  One could also do interpolation between several (2 or more) plans melodies that have been learnt [Note: this way of doing is actually some precursor of doing interpolation on embeddings of melodies to be generated by combining a decoder feedforward strategy & an iterative feedforward strategy, e.g. in VRAE or Music VAE systems, described in Sects. 6.10.2.3 & 6.12.1, resp.]. Examples are shown in Fig. 6.21: Examples of melodies generated by Sequential architecture. (o_A, o_B) Original plan melodies learnt. (i_1, i_2) Melodies generated by interpolating between o_A plan & o_B plan melodies. Inspired from [189].. Sequential architecture is summarized in Table 6.12: Sequential architecture summary.

- 6.8.3. #3 Example: BLSTM Chord Accompaniment Symbolic Music Generation System. BLSTM (Bidirectional LSTM) chord accompaniment system by Lim et al. [119] is a rare & interesting case [As noted in Sects. 5.8.2 & 6.5.1.] of an accompaniment system based on a recurrent architecture. Objective: generate a progression (sequence) of chords as an accompaniment to a melody (specified symbolically).

Corpus is imported from a now defunct lead sheet public data base. Authors selected 2252 selected lead sheets of various western modern music (rock, pop, country, jazz, folk, R&B, children's song, etc.), all in major key & majority with a single chord per measure (otherwise only 1st chord is considered). This results in a training set of 1,802 songs (making a total of 72,418 measures) & a test set of 450 songs (17768 measures). All songs are transposed (aligned) to C major key.

Desired characteristics are extracted from original XML files & converted to a CSV [CSV stands for Comma-separated values.] (spreadsheet) matrix format, as shown in Fig. 6.22: Example of extracted data from a single measure [119]. Specificities (simplifications) of representation are as follows:

- for melody [& obviously also for chords.], only pitch classes are considered (& octaves are not), resulting in a 12 notes 1-hot encoding (named 12-semitone-vector) plus the rest
- for chords, only their primary triads are considered, with only 2 types: major & minor, resulting in a 24 chords 1-hot encoding.

Architecture is a bidirectional LSTM with 2 LSTM layers, each one with 128 unis. Motivation: provide network with musical context backward & also forward in time. Time step considered by architecture is 4 measures long, as shown in Fig. 6.23: BLSTM architecture [119]. Tanh function is used as nonlinear activation function for hidden layers & softmax is used as output layer activation function, with categorical cross-entropy as its associated cost function.

Training is done with various 4 measures long samples as input & their associated 4 chords as output, generated by sliding a 4 measures long window over each training song. Generation is done by iteratively feedforwarding successive 4 measures long melody fragments (time slices) of a song & concatenating resulting 4 measures long chord progression fragments.

Architecture is peculiar (riêng) in that, although recurrent, generation is not recursive & output data has a different nature & structure (chords) than input data (notes). Furthermore, note: although strategy is iterative & architecture is recurrent, granularity (độ chi tiết) of each iterative step is quite coarse as it is 4 measures long, as opposed to most of systems based on recurrent architectures & iterative feedforward strategy which consider time step at level of smallest notre duration (see, e.g., system analyzed in Sect. 6.5.1.1). This kind of mixed architecture/strategy between forward/single step & recurrent/iterative may have been motivated by objective of capturing sufficiently history of horizontal correlations (between notes of melody & between chords of accompaniment) as LSTM cells focus on capturing history of vertical correlations (between notes & chords).

System has been evaluated by comparing to some hidden Markov model (HMM) model & to some deep neural network–HMM hybrid model (named DNN-HMM, see details in [198]), both quantitatively (by comparing accuracies & through confusion matrices), & qualitatively (through a web-based survey of 25 musically untrained participants). Results are showing a better accuracy & preference for BLSTM model, see a simple example in Fig. 6.24: Comparison of generated chord progressions (HMM, DNN-HMM, BLSTM & original) [119]. Authors note: evaluation also shows: when songs are unknown, preference for BLSTM model is weaker. They conjecture: this is because BLSTM often generates a more diverse chord sequence than original. BLSTM system is summarized in Table 6.13: BLSTM summary.

∗ 6.8.4. Summary. In summary, have seen: an RNN architecture is usually coupled to an iterative feedforward strategy, which allows a recursive seed-based variable length generation, as discussed in Sect. 6.5. However, there are some exceptions:

- Time-Windowed system by TODD (Sect. 6.8.1) uses an iterative feedforward strategy on a feedforward architecture in order to generate a melody,
- BLSTM system (Sect. 6.8.3) uses an iterative feedforward strategy on a recurrent architecture in order to generate a chord accompaniment to a melody.

See further (with VRAE system to be described in Sect. 6.10.2.3) use of an RNN Encoder-Decoder compound architecture (Sect. 5.13.3), as a way to decouple length of input sequence with length of output sequence, by combining decoder feedforward strategy with iterative feedforward strategy.

Some other examples of couplings between architectures & strategies, or between challenges, will be discussed in Sect. 6.18. Before that, continue to analyze challenges & possible solutions or directions.

○ 6.9. Melody–Harmony Interaction. When objective: generate simultaneously a melody with an accompaniment, expressed through some harmony or counterpoint [Harmony & counterpoint are dual approaches of accompaniment with different focus & priorities. Harmony focuses on *vertical* relations between simultaneous notes, as objects on their own (*chords*), & then considers horizontal relations between them (e.g., harmonic cadences). Conversely, counterpoint focuses on *horizontal* relations between successive notes for each simultaneous melody (a *voice*), & then considers vertical relations between their progression (e.g., to avoid parallel 5ths). Note: although their perspectives are different, analysis & control of relations between vertical & horizontal dimensions are their shared objectives.], an issue: musical consistency between melody & harmony. Although a general architecture e.g. MiniBach (Sect. 6.2.2) is supposed to have learnt correlations, interactions between vertical & horizontal dimensions are not explicitly considered.

Have analyzed in Sect. 6.5.1.2 an example of a specific architecture to generate simultaneously melody & chords, with explicit relations between them (i.e. chords can use prev step information about melody but not opposite). However, this architecture is a bit ad hoc. In following sects, will analyze some more general architectures having in mind interactions between melody & harmony.

∗ 6.9.1. #1 Example: RNN-RBM Polyphony Symbolic Music Generation System. In [11], BOULANGER-LEWANDOWSKI et al. have associated to RBM-based architecture introduced in Sect. 6.4.2.3 a RNN in order to represent temporal sequence of notes. Idea: *couple* RBM to a deterministic RNN with a single hidden layer, s.t.

- RNN models *temporal sequence* to produce successive outputs, corresponding to successive time steps,

- which are *parameters*, more precisely *biases*, of an RBM that models *conditional probability distribution* of *accompaniment notes*, i.e., which notes should be played together.

I.e., objective: combine a *horizontal view* (temporal sequence) & a *vertical view* (combination of notes for a particular time step). Resulting architecture named RNN-RBM is shown in Fig. 6.25: RNN-RBM architecture [12], & can be interpreted as follows:

- bottom line represents temporal sequence of RNN hidden units $u^{(0)}, u^{(0)}, \ldots, u^{(t)}$, where $u^{(t)}$ notation means [Note: usual notation would be u_t, as $u^{(t)}$ notation is usually reserved to index dataset examples (tth example), Sect. 5.8.] value of RNN hidden layer u at time (index) t
- upper part represents sequence of each RBM instance at time t, which could notate RBM$^{(t)}$, with
  1. $v^{(t)}$ its visible layer with $b_v^{(t)}$ its bias
  2. $h^{(t)}$ its hidden layer with $b_h^{(t)}$ its bias,
  3. W: weight matrix of connections between visible layer $v^{(t)}$ & hidden layer $h^{(t)}$.

  There is a specific training algorithm, not detailed here, see [11]. During generation, each t time step of processing is as follows:
  1. compute biases $b_v^{(t)}$ & $b_h^{(t)}$ of RBM$^{(t)}$, via (6.1)–(6.2) resp.
  2. sample from RBM$^{(t)}$ by using block Gibbs sampling to produce $v^{(t)}$,
  3. feedforward RNN with $v^{(t)}$ as input, using RNN hidden layer value $u^{(t-1)}$, in order to produce RNN new hidden layer value $u^{(t)}$ via (6.3), where
     (a) W_{vu}: weight matrix & b_u: bias for connections between input layer of RBM & hidden layer of RNN
     (b) W_{uu}: weight matrix for recurrent connections of hidden layer of RNN.

$$\begin{aligned} b_v^{(t)} &= b_v + W_{uv}u^{(t-1)}, \\ b_h^{(t)} &= b_h + W_{uh}u^{(t-1)}, \\ u^{(t)} &= \tanh(b_u + W_{uu}u^{(t-1)} + W_{vu}v^{(t)}). \end{aligned}$$

Note: biases $b_v^{(t)}, b_h^{(t)}$ of RBM$^{(t)}$ are variable for each time step, i.e., they are *time dependent*, whereas weight matrix W for connections between visible & hidden layer of RBM$^{(t)}$ is *shared* for all time steps (∀ RBMs), i.e., it is *time independent* [$W_{uv}, W_{uh}, W_{uu}, W_{vu}$ weight matrices are also shared & thus time independent.].

4 different corpora have been used in experiments: classical piano, folk tunes, orchestral classical music & J. S. Bach chorales. Polyphony varies from 0 to 15 simultaneous notes, with an average value of 3.9. A piano roll representation is used with many-hot encoding of 88 units representing pitches from A_0 to C_8. Discretization (time step) is a quarter note. All examples are aligned onto a single common tonality (âm giai): C major or minor. An example of a sample generated in a piano roll representation is shown in Fig. 6.26: Example of a sample generated by RNN-RBM trained on J. S. Bach chorales [12].. Quality of model has made RNN-RBM, summarized at Table 6.14: RNN-RBM summary, 1 of reference architectures for polyphonic music generation.

6.9.1.1. Other RNN-RBM Systems. There have been a few systems following on & extending RNN-RBM architecture, but they are not significantly different & furthermore they have not been thoroughly evaluated. However, worth mentioning following:

1. RNN-DBN architecture [This is apparently state of art for J. S. Bach Chorales dataset in terms of cross-entropy loss.], using multiple hidden layers [60]
2. LSTM-RTRBM architecture, using an LSTM instead of an RNN [120].

* 6.9.2. #2 Example: Hexahedria Polyphony Symbolic Music Generation Architecture. System for polyphonic music proposed by Johnson in his Hexahedria blog [93] is hybrid & original in that it *integrates* into same architecture

1. a 1st part made of 2 RNNs (actually LSTM) layers, each with 300 hidden units, recurrent over *time dimension*, which are in charge of *temporal* horizontal aspect, i.e., relations between notes in a sequence. Each layer has connections across time steps, while being independent across notes
2. a 2nd part made of 2 other RNN (LSTM) layers, with 100 & 50 hidden units, recurrent over *note dimension*, which are in charge of *harmony* vertical aspect, i.e., relations between simultaneous notes within same time step. Each layer is independent between time steps but has transversal directed connections between notes.

Can notate this architecture as LSTM^{2+2} in order to highlight 2 successive 2-level recurrent layers, recurrent in 2 different dimensions (time & note). Architecture is actually a kind of integration within a single architecture [See in Sect. 6.9.3 an alternative architecture, named Bi-Axial LSTM, where each of 2-level time-recurrent layers is encapsulated into a different architectural module.] of RNN-RBM architecture described in prev sect. Main originality is in using recurrent networks not only on time dimension but also on note dimension, more precisely on pitch class dimension. This latter type of recurrence is used to model occurrence of a simultaneous note based on other simultaneous notes. Like for time relation, which is oriented towards future, pitch class relation is oriented towards higher pitch, from C to B.

Resulting architecture is shown in its folded from Fig. 6.27: Hexahedria architecture (folded) [93]. & in its unfolded form [Our unfolded pictorial representation of an RNN shown in Fig. 6.29: Bi-Axial LSTM architecture [126]. was actually inspired by Johnson's Hexahedria pictorial representation.] in Fig. Fig. 6.28: Hexahedria architecture (unfolded) [93]., with 3 axes represented:

1. *flow axis*, shown horizontally & directed from left to right, represents flow of (feedforward) computation through architecture, from input layer to output layer
2. *note axis*, shown vertically & directed from top to bottom, represents connections between units corresponding to successive notes of each of 2 last (note-oriented) recurrent hidden layers
3. *time axis*, only in unfolded in Fig. 6.28, shown diagonally & directed from top left to bottom right, represents time steps & propagation of memory within a same unit of 2 1st (time-oriented) recurrent hidden layers.

Dataset is constructed by extracting 8 measures long parts from MIDI files from Classical piano MIDI database [104]. Input representation used is piano roll, with pitch represented as MIDI note number. More specific information is added: pitch class, prev note played (as a way to represent a possible hold), how many times a pitch class has been played in prev time step & beat (position within measure, assuming a 4/4 time signature). Output representation is also a piano roll, in order to represent possibility of > 1 note at same time. Generation is done in an iterative way (i.e., following iterative feedforward strategy), as for most recurrent networks. System is summarized in Table 6.15: Hexahedria summary.

∗ 6.9.3. #3 Example: Bi-Axial LSTM Polyphony Symbolic Music Generation Architecture. Johnson recently proposed an evolution of his original Hexahedria architecture, described in Sect. 6.9.2, named Bi-Axial LSTM (or BALSTM) [94].

Representation used is piano roll, with note hold & rest tokens added to vocabulary. Various corpora are used: JSB Chorales dataset, a corpus of 382 4-part chorales by J. S. Bach [1]; MuseData library, an electronic classical music library from CCARH in Stanford [76]; Nottingham database, a collection of 1200 folk tunes in ABC notation [53]; & Classical piano MIDI database [104]. Each dataset is transposed (aligned) into key of C major or C minor.

Probability of playing a note depends on 2 types of information:

1. all notes at prev time steps – this is modeled by *time-axis module*
2. all notes within current time step that have already been generated (order being lowest to highest) – this is modeled by *note-axis module*.

There is an additional front end layer, named "Note Octaves", which transforms each note into a vector of all its possible corresponding octave notes (i.e., an extensional version of pitch classes). Resulting architecture is illustrated in Fig. 6.29: Bi-Axial LSTM architecture [126] [This figure comes from description of another system based on Bi-Axial LSTM architecture, named DeepJ, described in Sect. 6.10.3.4.]. "x2" represents fact: each module is stacked twice (i.e., has 2 layers).

Time-axis module is recurrent in time (as for a classical RNN), LSTM weights being shared across notes in order to gain note transposition invariance. Note-axis module [Note: as opposed to Johnson's 1st architecture (refer to as Hexahedria, introduced in Sect. 6.9.2), which integrates 2-level time-recurrent layers with 2-level note-recurrent layers within a single architecture & therefore notated as LSTM^{2+2}, Bi-Axial LSTM architecture explicitly separates each 2-level time-recurrent layers into distinct architecture modules & is therefore notated as LSTM$^2 \times 2$.] is recurrent in note. For each note input of note-axis module, $\oplus$ represents concatenation of corresponding output from time-axis module with already predicted lower notes. Sampling (into a binary value, by using a coin flip) is applied to each note output probability in order to compute final prediction (whether that note is played or not).

As pointed out by Johnson [94], during training phase, as all notes at all time steps are known, training process may be accelerated by processing each layer independently (e.g., on a GPU), by running input through 2 time-axis layers in parallel across all notes, & using 2 note-axis layers to compute probabilities in parallel across all time steps.

Generation phase is sequential for each time step (by following both iterative feedforward strategy & sampling strategy). An excerpt of music generated is shown in Fig. 6.30: Example of Bi-Axial LSTM generated music (excerpt) [94].

Bi-Axial LSTM system, summarized in Table 6.16: Bi-Axial LSTM summary, has been evaluated & compared to some other architectures. Author reports noticeably better results with Bi-Axial LSTM, greatest improvements being on MuseData [76] & Classical piano MIDI database [104] datasets, & states in [94]: "It is likely due to fact: those datasets contain many more complex musical structures in different keys, which are an ideal case for a translation-invariant architecture." Note: an extension of Bi-Axial LSTM architecture with conditioning, named DeepJ, introduced in Sect. 6.10.3.4.

∘ 6.10. Control. A deep architecture generates musical content matching style learnt from corpus. This capacity of induction from a corpus without any explicit modeling or programming is an important ability, as discussed in Chap. 1 & also in [51]. However, like a fast car that needs a good steering wheel, control is also needed as musicians usually want to *adapt* ideas & patterns *borrowed* from other contexts to their own objective & context, e.g., transposition to another key, minimizing number of notes, finishing with a given note, etc.

∗ 6.10.1. Dimensions of Control Strategies. Arbitrary control is a difficult issue for DL architectures & techniques because neural networks have not been designed to be controlled. In case of Markov chains, they have an operational model on which on can attach constraints to control generation [2 examples: Markov constraints [148] & factor graphs [147].]. However, neural networks do not offer such an operational entry point & distributed nature of their representation does not provide a clear relation to structure of content generated. Therefore, most of strategies for controlling DL generation rely on *external* intervention at various *entry points* (hooks) & *levels*:

1. input
2. output
3. input & output
4. encapsulation/reformulation.

Various control *strategies* can be employed:

1. sampling
2. conditioning
3. input manipulation
4. reinforcement
5. unit selection.

Some strategies (e.g. sampling, Sect. 6.10.2) are more *bottom-up* & others (e.g. structure imposition, Sect. 6.10.5.1, a unit selection, Sect. 6.10.7) are more *top-down*. Lastly, there is also a continuum between *partial* solutions (e.g. conditioning/parametrization, Sect. 6.10.3) & more *general* approaches (e.g. reinforcement, Sect. 6.10.6).

∗ 6.10.2. Sampling. Sampling from a stochastic architecture (e.g. a restricted Boltzmann machine (RBM), Sect. 6.4.2), or from a deterministic architecture (in order to introduce *variability*, Sect. 6.6.1), may be an entry point for control if introduce *constraints* into sampling process. This is called *constrained sampling*, see e.g. C-RBM system in Sect. 6.10.5.1. Constrained sampling is usually implemented by a *generate-&-test* approach, where valid solutions are picked from a set of random samples generated from model. But this could be a very costly process &, moreover, with no guarantee of success. A key & difficult issue is therefore how to *guide* sampling process in order to fulfill constraints.

· 6.10.2.1. Sampling for Iterative Feedforward Generation. In case of an iterative feedforward strategy on a recurrent network, some refinements in sampling produce can be made.

In Sect. 6.6.1, introduced technique of sampling softmax output of a recurrent network in order to introduce content variability. However, this may sometimes lead to generation of an unlikely note (with a low probability). Moreover, as noted in [66], generating such a "wrong" note can have a cascading effect on remaining of generated sequence.

A counter measure consist in adjusting a learnt RNN model (conditional probability distribution $P(s_t|s_{<t})$, as defined in Sect. 5.8) by not considering notes with a probability under a certain threshold. New model, with a probability distribution $P_{\text{threshold}}(s_t|s_{<t})$, is defined in

$$P_{\text{threshold}}(s_t|s_{<t}) \coloneqq \begin{cases} 0 & \text{if } \dfrac{P(s_t|s_{<t})}{\max_{s_t} P(s_t|s_{<t})} < \text{threshold}, \\ \dfrac{P(s_t|s_{<t})}{z} & \text{otherwise.} \end{cases}$$

1. $\max_{s_t} P(s_t|s_{<t})$: note maximum probability
2. *threshold*: threshold hyperparameter
3. z: a normalization constant.

A slightly more sophisticated version interpolates between original distribution $P(s_t|s_{<t})$ & $\arg\max_{s'_t} P(s'_t|s_{<t})$ deterministic variant (Sect. 6.6.1), with some temperature user control hyperparameter (see more details in [66, Sect. 4.1.1.3]).

This technique will be further generalized & combined with conditioning strategy in order to control generation of notes at specific positions via positional constraints. This will be exemplified by Anticipation-RNN system to be introduced in Sect. 6.10.3.5.

· 6.10.2.2. Sampling for Incremental Generation. In case of an incremental generation (introduced in Sect. 6.14), user may select

1. on which part (e.g., a given part of a melody &/or a given voice) sampling will occur (or reoccur),
2. interval of possible values on which sampling will occur.

In case of DeepBach system (introduced in Sect. 6.15.2), this will be basis for introducing user control on generation, notably to regenerate only some parts of a music, to restrict note range, & to impose some basic rhythm.

· 6.10.2.3. Sampling for Variational Decoder Feedforward Generation. Another interesting case: use of sampling for *generative models*, e.g. variational autoencoders (VAEs) & generative adversarial networks (GANs), introduced in Sect. 6.10.2.4. Some nice control of sampling, e.g., to produce an interpolation, averaging or attribute modification, will produce meaningful variations in content generated by decoder feedforward strategy. Moreover, as has been discussed in Sect. 5.6.2, a variational autoencoder (VAE) is interesting for its ability for controlling generation over significant dimensions that have been learnt.

Example 1 (VRAE Video Game Melody Symbolic Music Generation System)**.** *In [49],* Fabius & van Amersfoort *propose extension of RNN Encoder-Decoder architecture to case of a variational autoencoder (VAE), which is therefore named a variational recurrent autoencoder (VRAE). Both encoder & decoder encapsulate an RNN (actually an LSTM), as has been explained in Sect. 5.13.3. In terms of strategy, VRAE combines iterative feedforward strategy with decoder feedforward strategy & sampling strategy.*

Corpus used in experiment is a set of MIDI files of 8 video game songs from 1980s & 1990s (Sponge Bob, Super Mario, Tetris, . . .), which are divided into various shorter parts of 50 time steps. A 1-hot encoding of 49 possible pitches is used (pitches with too few occurrences of notes were not considered). Experiments have been conducted with 2 or 20 hidden layer units (latent variables). Training takes place as for training recurrent networks, i.e. for each input note presenting next note as output.

After training phase, latent space vector can be sampled & used by RNN encapsulated within decoder to generate iteratively a melody. This could be done by random sampling or also by interpolating between values of latent variables corresponding to different songs that have been learnt, creating a sort of "medley" of these songs. Fig. 6.31: Visualization of VRAE latent space encoded data. Extended from [49]. *visualizes organization of encoded data in latent space, each color representing data points from 1 song. Result is positive, but low musical quality of corpus hampers a careful evaluation. VRAE system is summarized in* Table 6.17: VRAE summary.

Example 2 (GLSR-VAE Melody Symbolic Music Generation System)**.** *Architecture proposed by* Hadjeres & Nielsen *in [68] is based on a variational autoencoder (VAE) architecture (Sect. 5.6.2), but it proposes an improvement in control of variation in generation, named geodesic latent space regularization (GLSR), with a system named GLSR-VAE.*
Starting point: a straight line between 2 points in latent space will not necessarily produce best *interpolation in generated content domain space. Idea: introduce a regularization to relate variations in latent space to variations in attributes of decoded elements. Details of definition of added cost term may be found in [68].*
Experiment consists in generating chorale melodies in style of J. S. Bach. *Dataset comprises monophonic soprano voices from* J. S. Bach *chorales corpus [5].*
GLSR-VAE shares principle of representation initiated by DeepBach system (Sect. 6.14.2), i.e.

1. *1-hot encoding of a note*
2. *with addition to vocabulary of hold symbol __ & rest symbol to specify, resp., a note repetition & a rest (Sect. 4.11.7)*
3. *using names of notes (with no enharmony, e.g., ♯ & G♭ are considered to be different, Sect. 4.9.2).*

Quantization is at level of a 16th note. Latent variable space is set to 12 dimensions (12 latent variables).
In experiments conducted, regularization is executed on a 1st dimension which has been found (Sect. 5.6.2) to represent number of notes (named z_1). Fig. 6.32: Visualization of GLSR-VAE latent space encoded data [68]: With & Without geodesic latent space regularization shows organization of encoded data in latent space, with number of notes z_1 being abscissa axis, with from left to right an effective progressive increase in number of notes (shown with scales of colors). Fig. 6.33: Examples of 2 measures long melodies (separated by double bar lines) generated by GLSR-VAE [68]. shows examples of melodies generated (each 2 measures long, separated by double bar lines) while increasing z_1, showing a progressive correlated densification of melodies generated.
GLSR-VAE is summarized in Table 6.18: GLSR-VAE summary. More examples of sampling from variational autoencoders are described in Sect. 6.12.1.

- 6.10.2.4. Sampling for Adversarial Generation. Another example of a generative model is a generative adversarial networks (GAN) architecture. In such an architecture, after having trained generator in an adversarial way, generation of content is done by sampling latent random variables.

Example 3 (Mogren's C-RNN-GAN Classical Polyphony Symbolic Music Generation System)**.** *Objective of Mogren's C-RNN-GAN [134] system: generation of single voice polyphonic music. Representation chosen is inspired by MIDI & models each musical event (note) via 4 attributes: duration, pitch, intensity & time elapsed since prev event, each attribute being encoded as a real value scalar. This allows representation of simultaneous notes (in practice up to 3). Musical genre of corpus is classical music, retrieved in MIDI format from Web & contains 3697 pieces from 160 composers.*
C-RNN-GAN is based on a generative adversarial networks (GAN) architecture, with both generator & discriminator being recurrent networks [This generative GAN architecture encapsulates 2 recurrent networks, in same spirit that generative VRAE variational autoencoder architecture encapsulates 2 recurrent network as explained in Sect. 6.10.2.3.], more precisely each having 2 LSTM layers with 350 units each. A specificity: discriminator (but not generator) has a bidierctional recurrent architecture, in order to take context from both past & future for its decisions. Architecture is shown in Fig. 6.34: C-RNN-GAN architecture [134]. *& summarized in* Table 6.19: C-RNN-GAN summary.
Discriminator is trained, in parallel to generator, to classify if a sequence input is coming from real data. Similar to case of encoder part of RNN Encoder-Decoder, which summarizes a musical sequence into values of hidden layer (see Sect. 5.13.3), bidirectional RNN decoder part of C-RNN-GAN summarizes sequence input into values of 2 hidden layers (forward sequence & backward sequence) & then classifies them.
An example of generated music is shown in Fig. 6.35: C-RNN-GAN generated example (excerpt) [134]. *Author conducted a number of measurements on generated music. He states: model trained with feature matching [A regularization technique for improving GANs, see Sect. 5.11.1.] achieves a better trade-off between structure & surprise than other variants. Note: this is consonant with use of feature matching regularization technique to control creativity in Midinet (introduced in Sect. 6.10.3.3). C-RNN-GAN is summarized in* Table. 6.19.

- 6.10.2.5. Sampling for Other Generation Strategies. Sampling may also be combined with other strategies for content generation, as e.g.

  1. *Conditioning*, as a way to *parameterize* generation with constraints, in Sect. 6.10.3.5, or
  2. *Input manipulation*, as a way to *correct* manipulation performed in order to *realign* samples with learnt distribution, in Sect. 6.10.5.

∗ 6.10.3. Conditioning. Idea of *conditioning* (sometimes also named *conditional architecture*): condition architecture on some extra information, which could be arbitrary, e.g., a class label or data from other modalities. E.g.

- a *bass line* or a *beat structure*, in rhythm generation architecture (Sect. 6.10.3.1)
- a *chord progression*, in MidiNet architecture (Sect. 6.10.3.3)
- *previously generated note*, in VRASH architecture (Sect. 6.10.3.6)
- some *positional constraints on notes*, in Anticipation-RNN architecture (Sect. 6.10.3.5)
- a *musical genre* or an *instrument*, in WaveNet architecture (Sect. 6.10.3.2)
- a *musical style*, in DeepJ architecture (Sect. 6.10.3.4).

In practice, conditioning information is usually fed into architecture as an additional input layer (e.g., see Fig. 6.38: Example of a rhythm pattern generated with a specific bass line as conditioning input [122].). This distinction between *standard input & conditioning input* follows a good architectural modularity principle [Note: do not consider conditioning as a strategy because consider: essence of conditioning relates to *conditioning architecture*. Generation uses a conventional strategy (e.g., single-step feedforward, iterative feedforward ...) depending on type of architecture (e.g., feedforward, recurrent ...).]. Conditioning is a way to have some degree of parameterized control over generation process.

Conditioning layer could be

- a simple input layer. E.g.: a tag specifying a musical genre or an instrument in WaveNet system (Sect. 6.10.3.2)
- some output of some architecture, being
  1. same architecture, as a way to condition architecture on some history [This is close in spirit to a recurrent architecture (RNN).] – e.g.: Midinet system (Sect. 6.10.3.3) in which history information from prev measure(s) is injected back into architecture, or
  2. another architecture – e.g.: rhythm generation system (Sect. 6.10.3.1) in which a feedforward network in charge of bass line & metrical structure information produces conditioning input, & DeepJ system (Sect. 6.10.3.4) in which 2 successive transformation layers of a style tag produce an embedding used as conditioning input.

If architecture is time-invariant – i.e., recurrent or convolutional over time – there are 2 options

- *global conditioning*: if conditioning input is shared for all time steps, or
- *local conditioning*: if conditioning input is specific to each time step.

WaveNet architecture, which is convolutional over time (Sect. 5.9.5), offers 2 options, as analyzed in Sect. 6.10.3.2.

- 6.10.3.1. #1 Example: Rhythm Symbolic Music Generation System.
- 6.10.3.2. #2 Example: WaveNet Speech & Music Audio Generation System.
- 6.10.3.3. #3 Example: MidiNet Pop Music Melody Symbolic Music Generation System.

- 6.10.4. Input Manipulation.
- 6.10.5. Input Manipulation & Sampling.
- 6.10.6. Reinforcement.
- 6.10.7. Unit Selection.

- 6.11. Style Transfer.
- 6.12. Structure.
- 6.13. Originality.

- 7. Analysis.
- 8. Discussion & Conclusion.
- Glossary.

1. **ABC notation.** A text-based musical notation for folk & traditional music.
   – **Ký hiệu ABC.** Ký hiệu âm nhạc dựa trên văn bản dành cho nhạc dân gian & nhạc truyền thống.
2. **Accompaniment.** Musical part which provides rhythm &/or harmonic support for melody or main themes of a song or instrumental piece. There are many different styles & types of accompaniment in different genres & styles of music. Examples: an harmony accompaniment through a progression (sequence) of chords to be played by a polyphonic instrument e.g. a piano or guitar, & a counterpoint accompaniment through a sequence of melodic voices to be played by human voices or by instruments.
   – **Đệm.** Phần âm nhạc cung cấp nhịp điệu &hoặc hỗ trợ hòa âm cho giai điệu hoặc chủ đề chính của 1 bài hát hoặc bản nhạc không lời. Có nhiều phong cách & loại đệm khác nhau trong các thể loại & phong cách âm nhạc khác nhau. Ví dụ: đệm hòa âm thông qua 1 chuỗi (chuỗi) hợp âm được chơi bằng 1 nhạc cụ đa âm, ví dụ như piano hoặc guitar, & đệm đối âm thông qua 1 chuỗi các giọng hát giai điệu được chơi bằng giọng người hoặc nhạc cụ.
3. **Activation function.** Function associated to a neural network layer. In case of hidden layers, its purpose: add nonlinearity. Standard examples are sigmoid, tanh, & ReLU. IN case of output layer, its purpose: organize result in order to be able to interpret it. Examples of output layer activation function: softmax for computing associated probabilities in case of a categorical classification task with a single label to be selected, & identity in case of a prediction task.
   – **Hàm kích hoạt.** Hàm liên kết với 1 lớp mạng nơ-ron. Trong trường hợp các lớp ẩn, mục đích của nó: thêm phi tuyến tính. Các ví dụ tiêu chuẩn là sigmoid, tanh, & ReLU. Trong trường hợp lớp đầu ra, mục đích của nó: sắp xếp kết quả để có thể diễn giải kết quả đó. Các ví dụ về hàm kích hoạt lớp đầu ra: softmax để tính toán xác suất liên kết trong trường hợp tác vụ phân loại theo danh mục với 1 nhãn duy nhất cần chọn, & danh tính trong trường hợp tác vụ dự đoán.
4. **Algorithmic composition.** Use of algorithms & computers to generate music compositions (symbolic form) or music pieces (audio form). Examples of models & algorithms are: grammars, rules, stochastic processes (e.g., Markov chains), evolutionary methods & ANNs.
5. **Architecture.** An (ANN) architecture is structure of organization of computational units (neurons), usually grouped in layers, & their weighted connections. Examples of types of architecture: feedforward (aka multilayer Perceptron), recurrent (RNN), autoencoder & generative adversarial networks. Architectures process encoded representations (in our case of a musical content) which have been encoded.

6. **ANN.** A family of bio-inspired ML algorithms whose model is based on weighted connections between computing units (neurons). Weights are incrementally adjusted during training phase in order for model to fit data (examples).

7. **Attention mechanism.** A mechanism inspired by human visual system which focuses at each time step on some specific elements of input sequence. This is modeled by weighted connections onto sequence elements (or onto sequence of hidden units) which are subject to be learned.

8. **Autoencoder.** A specific case of ANN architecture with an output layer mirroring input layer & with a hidden layer. Autoencoders are good at extracting features.

9. **Backpropagation.** A short hand for "backpropagation of errors", algorithm used to compute gradients (partial derivatives w.r.t. each weight parameter & to bias) of cost function. Gradients will be used to guide minimization of cost function in order to fit data.

10. **Bag-of-words (BOW).** consist in transforming original text (or arbitrary representation) into a vocabulary composed to all occurring tokens (items). Then, various measures can be used to characterize text, most common being term frequency, i.e., number of times a term appears in text. Mainly used for feature extraction, e.g., to characterize & compare texts.

11. **Bias.** b offset term of a simple linear regression model $h(\mathbf{x}) = b + \boldsymbol{\theta} \cdot \mathbf{x}$ & by extension of a neural network layer.

12. **Bias node.** Node of a neural network layer corresponding to a bias. Its constant value is 1 & is usually notated as +1.

13. **Binning.** A technique to discretize a continuous interval or to reduce dimensionality of a discrete interval. It consists of dividing original domain of values into smaller intervals & replacing each bin (& values within it) by a value representative, often central value.

14. **Bottleneck hidden layer (aka Innermost hidden layer).** Innermost hidden layer of a stacked autoencoder. It provides a compact & high-level embedding of input data & may be used as a seed for generation (by chain of decoders).

15. **Challenge.** 1 of qualities (requirements) that may be desired for music generation. Examples of challenges: incrementality, originality, & structure.

16. **Chromagram (aka Chroma).** A discretized version of a spectrogram. It is discretized onto tempered scale & is independent of octave.

17. **Classification.** A ML task about attribution of an instance to a class (from a set of possible classes). E.g.: determine if next note is a C_4, a $C\sharp_4$, etc.

18. **Conditioning architecture.** Parameterization of an ANN architecture by some conditioning information (e.g., a bass line, a chord progression ...) represented via a specific extra input, in order to guide generation.

19. **Connection.** A relation between a neuron & another neuron representing a computational flow from output of 1st neuron to an input of 2nd neuron. A connection is modulated by a weight which will be adjusted during training phase.

20. **Convolution.** In mathematics, a mathematical operation on 2 functions sharing same domain that produces a 3rd function which is integral (or sum in discrete case – case of images made of pixels) of pointwise multiplication of 2 functions varying within domain in an opposing way. Inspired both by mathematical convolution & by a model of human visions, it has been adapted to ANNs & it improves pattern recognition accuracy by exploiting spatial local correlation present in natural images. Basic principle: slide a matrix (named a filter, a kernel or a feature detector) through entire image (seen as input matrix), & for each mapping position to compute dot product of filter with each mapped portion of image & then sum up all elements of resulting matrix. Results are named feature maps.

21. **Correlation.** Any statistical relationship, whether causal or not, between 2 random variables. ANNs are good at extracting correlations between variables, e.g. between input variables & output variables & also between input variables.

22. **Cost function (aka Loss function).** Function used for measuring distance between prediction by an ANN architecture $\hat{y}$ & actual target (true value y). Various cost functions may be used, depending on task (prediction or classification) & encoding of output, e.g., mean squared error, binary cross-entropy & categorical cross entropy.

23. **Counterpoint.** In musical theory, an approach for accompaniment of a melody through a set of other melodies (voices). An example is a chorale with 3 voices (alto, tenor, & bass) matching a soprano melody. Counterpoint focuses on horizontal relations between successive notes for each simultaneous melody (voice) & then considers vertical relations between their progression (e.g., to avoid parallel 5ths).

    – **Đối âm.** Trong lý thuyết âm nhạc, 1 cách tiếp cận để đệm 1 giai điệu thông qua 1 tập hợp các giai điệu khác (giọng hát). Một ví dụ là 1 hợp xướng với 3 giọng hát (alto, tenor, & bass) phù hợp với 1 giai điệu soprano. Đối âm tập trung vào các mối quan hệ theo chiều ngang giữa các nốt nhạc liên tiếp cho mỗi giai điệu đồng thời (giọng hát) & sau đó xem xét các mối quan hệ theo chiều dọc giữa các tiến trình của chúng (ví dụ, để tránh song song 5).

24. **Cross-entropy.** A function measuring dissimilarity between 2 probability distributions. It is used as a cost (loss) function for a classification task to measure difference between prediction by an ANN architecture $\hat{y}$ & actual target (true value y). There are 2 types of cross-entropy cost functions: binary cross-entropy when classification is binary & categorical cross-entropy when classification is multiclass with a single label to be selected.

25. **Data synthesis.** A ML technique to generate synthesic data as a way to artificially augment size of dataset (number of training examples), in order to improve accuracy & generalization of learnt model. In musical domain, a natural & easy way is transportation, i.e., to transpose all examples in all keys.

26. **Dataset.** Set of examples used for training an ANN architecture. Dataset is usually divided into 2 subsets: training set used during training phase & validation set used to estimate ability for generalization by model learnt.

27. **Decoder.** Decoding component of an autoencoder which reconstructs compressed representation (an embedding) from hidden layer into a representation at output layer as close as possible to initial data representation at input layer.

28. **Decoder feedforward strategy.** A strategy for generating content based on an autoencoder architecture in which values are assigned onto latent variables of hidden layer & forwarded into decoder component of architecture in order to generate a musical content corresponding to abstract description inserted.

29. **DL (aka Deep neural network).** An ANN architecture with a significant number of successive layers.

30. **Discriminator.** Discriminative model component of generative adversarial networks (GAN) which estimates probability that a sample came from real data rather from generator.

31. **Embedding.** In mathematics, an injective & structure-preserving mapping. Initially used for natural language processing, it is now often used in DL as a general term for encoding a given representation into a vector representation.

32. **Encoder.** Encoding component of an autoencoder which transforms data representation from input layer into a compressed representation (an embedding) at hidden layer.

33. **Encoding.** Encoding of a representation consists in mapping of representation (composed of a set of variables, e.g., pitch or dynamics) into a set of inputs (also named input nodes or input variables) for neural network architecture. Examples of encoding strategies: value encoding, 1-hot encoding & many-hot encoding.

34. **End-to-end architecture.** An ANN architecture that processes raw unprocessed data – without any pre-processing, transformation of representation, or extraction of features – to produce a final output.

35. **Enharmony.** In tempered system, equivalence of notes with a same pitch, e.g. A♯ with B♭, although harmonically they are distinct.

36. **Feature map.** Also named a convolved feature, this is result of applying a filter matrix (also named a feature detector) at a specific position of an image & summing up all dot products. This represents basic operation of a convolutional ANN architecture.

37. **Feedforward.** Basic way for a neural network architecture to process an input by feedforwarding input data into successive layers of neurons of architecture until producing output. A feedforward neural architecture (also named multilayer neural network or multilayer Perceptron, MLP) is composed of successive layers, with at least 1 hidden layer.

38. **Fourier transform.** A transformation (which could be continuous or discrete) of a signal into decomposition into its elementary components (sinusoidal waveforms). As well as compressing information, its role is fundamental for musical purposes as it reveals harmonic components of signal.

39. **Generative adversarial networks (GAN).** A compound architecture composed of 2 component architectures, generator & discriminator, who are trained simultaneously with opposed objectives. Generator objective: generate synthetic samples resembling real data while discriminator objective: detect synthetic samples.

40. **Generator.** Generative model component of generative adversarial networks (GAN) whose objective: transform a random noise vector into a synthetic (faked) sample which resembles real samples drawn from a distribution of real data.

41. **Gradient.** A partial derivative of cost function w.r.t. a weight parameter or a bias.

42. **Gradient descent.** A basic algorithm for training a linear regression model & an ANN. It consists in an incremental update of weight parameters guided by gradients of cost function until reaching a minimum.

43. **Harmony.** In musical theory, a system for organizing simultaneous notes. Harmony focuses on vertical relations between simultaneous notes, as objects on their own (chords), & then considers horizontal relations between them (e.g., harmonic cadences).

44. **Hidden layer.** Any neuron layer located between input layer & output layer of a neural network architecture.

45. **Hold.** Information about a note that extends its duration over a single time step.

46. **Hyperparameter.** Higher-order parameters about configuration of a neural network architecture & its behavior. E.g.: number of layers, number of neurons for each layer, learning rate & stride (for a convolutional architecture).

47. **Input layer.** 1st layer of a neural network architecture. It is an interface consisting in a set of nodes without internal computation.

48. **Input manipulation strategy.** A strategy for generating content based on incremental modification of a representation to be processed by an ANN architecture.

49. **Iterative feedforward strategy.** A strategy for generating content by generating its successive time slices.

50. **Latent variable.** In statistics, a variable which is not directly observed. In DL architectures, variables within a hidden layer. By sampling a latent variable(s), one may control generation, e.g., in case of a variational autoencoder.

51. **Layer.** A component of a neural network architecture composed of a set of neurons with no direct connections between them.

52. **Linear regression.** In statistics, linear regression is an approach for modeling (assumed linear) relationship between a scalar variable & 1 for several explanatory variable(s).

53. **Linear separability.** Ability to separate by a line or a hyperplane elements of 2 different classes represented in an Euclidean space.

54. **Long short-term memory (LSTM).** A type of RNN architecture with capacity for learning long term correlations & not suffering from vanishing or exploding gradient problem during training phase. Idea: secure information in memory cells protected from standard data flow of recurrent network. Decisions about writing to, reading from & forgetting values of cells are performed by opening or closing of gates & are expressed at a distinct control level, while being learnt during training process.

55. **Many-hot encoding.** Strategy used to encode simultaneously several values of a categorical variable, e.g., a triadic chord composed of 3 note pitches. As for a 1-hot encoding, it is based on a vector having as its length number of possible values (e.g., from C_4 to B_4). Each occurrence of a note is represented with a corresponding 1 with all other elements being 0.

56. **Markov chain.** A stochastic model describing a sequence of possible states. Chance to change from current state to a state or to another state is governed by a probability & does not depend on prev states.

57. **Melody.** Abbreviation of a single-voice monophonic melody, i.e. a sequence of notes for a single instrument with at most 1 note at same time.

58. **Musical instrument digital interface (MIDI).** A technical standard that describes a protocol, a digital interface & connectors for interoperability between various electronic musical instruments, softwares & devices.

59. **Multilayer Perceptron (MLP).** A feedforward neural architecture composed of successive layers, with at least 1 hidden layer.

60. **Multivoice (aka Multitrack).** Abbreviation of a multivoice polyphony, i.e. a set of sequences of notes intended for > 1 voice or instrument.

61. **Neuron.** Atomic processing element (unit) of an ANN architecture, inspired by biological model of a neuron. A neuron has several input connections, each one with an associated weight, & 1 output. A neuron will compute weighted sum of all its input values & then apply its associated activation function in order to compute its output value. Weights will be adjusted during training phase of neural network architecture.

62. **Node.** Atomic structural element of an ANN architecture. A node could be a processing unit (a neuron) or a simple interface element for a value, e.g., in case of input layer or a bias node.

63. **Nonlinear function.** A function used as an activation function in an ANN architecture in order to introduce nonlinearity & to address linear separability limitation.

64. **Objective.** Nature & destination of musical content to be generated by a neural network architecture. Examples of objectives: a monophonic melody to be played by a human flutist & a polyphonic accompaniment played by a synthesizer.

65. **1-hot encoding.** Strategy used to encode a categorical variable (e.g., a note pitch) as a vector having as its length number of possible values (e.g., from C_4 to B_4). A given element (e.g., a note pitch) is represented with a corresponding 1 with all other elements being 0. Name comes from digital circuits, 1-hot referring to a group of bits among which only legal (possible) combinations of values are those with a single high (hot) (1) bit, all others being low (0).

66. **Output layer.** Last layer of a neural network architecture. It includes output activation function which could be e.g. a sigmoid or a softmax in case of a classification task.

67. **Overfitting.** Situation for an ANN architecture (& more generally speaking for a ML algorithm) when model learnt is well fit to training data but not to evaluation data. I.e., inability of model to generalize well.

68. **Parameter.** Parameters of an ANN architecture are weights associated to each connection between neurons as well as biases associated to each layer.

69. **Perceptron.** 1 of 1st ANN architecture, created by ROSENBLATT in 1957. It had not hidden layer & suffered from linear separability limitation.

70. **Piano roll.** Representation of a melody (monophonic or polyphonic) inspired from automated pianos. Each "perforation" represents a note control information, to trigger a given note. Length of perforation corresponds to duration of a note. In other dimension, localization (height) of a perforation corresponds to its pitch.

    – **Piano roll.** Biểu diễn giai điệu (đơn âm hoặc đa âm) lấy cảm hứng từ đàn piano tự động. Mỗi "lỗ thủng" biểu diễn thông tin điều khiển nốt nhạc, để kích hoạt 1 nốt nhạc nhất định. Chiều dài của lỗ thủng tương ứng với thời lượng của 1 nốt nhạc. Ở chiều không gian khác, vị trí (chiều cao) của lỗ thủng tương ứng với cao độ của nó.

71. **Pitch class.** Name of corresponding note (e.g., C) independently of octave position. Also named chroma.

72. **Polyphony.** Abbreviation of a single-voice polyphony, i.e., a sequence of notes for a single instrument (e.g., a guitar or a piano) with possibly simultaneous notes.

73. **Pooling.** For a convolutional architecture, a data dimensinality reduction operation (by max, average or sum) for each feature map produced by a convolutional stage, while retaining significant information. Pooling brings important property of invariance to small transformations, distortions & translations in input image.

    – **Pooling.** Đối với kiến trúc tích chập, 1 hoạt động giảm chiều dữ liệu (theo max, average hoặc sum) cho mỗi bản đồ đặc điểm được tạo ra bởi 1 giai đoạn tích chập, trong khi vẫn giữ lại thông tin quan trọng. Pooling mang lại tính chất bất biến quan trọng cho các phép biến đổi nhỏ, biến dạng & phép tịnh tiến trong hình ảnh đầu vào.

74. **Pre-training.** A technique, also named greedy layer-wise unsupervised training, consisting in prior training in cascade (1 layer at a time) of each hidden layer. It turned out to be a significant improvement for accurate training of ANNs with several layers by initializing weights based on learnt data.

75. **Q-learning.** An algorithm for reinforcement learning based on an incremental refinement of action value function Q which represents cumulated rewards for a given state & a given action.

76. **Recurrent connection.** A connection from an output of a node to its input. By extension, a layer recurrent connection fully connects all layer nodes outputs to all nodes inputs. This is basis of a RNN architecture.

77. **RNN.** A type of ANN architecture with recurrent connections, used to learn sequences.

78. **Reinforcement learning.** An area of ML concerned with an agent making successive decisions about an action in an environment while receiving a reward (reinforcement signal) after each action. Objective for agent: find best policy maximizing its cumulated rewards.

79. **Reinforcement strategy.** A strategy for content generation by modeling generation of successive notes as a reinforcement learning problem while using an RNN as a reference for modeling of reward. Therefore, one may introduce arbitrary control objectives (e.g., adherence to current tonality, maximum number of repetitions, etc.) as additional reward terms.

80. **ReLU.** Rectified linear unit function, which may be used as a hidden layer nonlinear activation function, specially in case of convolutions.

81. **Representation.** Nature & format of information (data) used to train & to generate musical content. Examples of types of representation: signal, spectrum, piano roll, & MIDI.

82. **Rest.** Information about absence of a note (silence) during 1 (or more) time step(s).

83. **Restricted Boltzmann machine (RBM).** A specific type of ANN that can learn a probability distribution over its set of inputs. It is stochastic, has no output & uses a specific learning algorithm.

84. **Sampling.** Action of producing an item (a sample) according to a given probability distribution over possible values. As more & more samples are generated, their distribution should more closely approximate given distribution.

85. **Sampling strategy.** A strategy for generating content where variables of a content representation are incrementally instantiated & refined according to a target probability distribution which has been previously learnt.

86. **Seed-based generation.** An approach to generate arbitrary content (e.g., a long melody) with a minimal (seed) information (e.g., a 1st note).

87. **Self-supervised learning.** A category of ML when output value of example (target value of supervision) = input value. An example: training of an autoencoder.

88. **Sigmoid.** Also named logistic function, it is used as an output layer activation function for binary classification tasks & it may also be used as a hidden layer nonlinear activation function.

89. **Single-step feedforward strategy.** A strategy for generating content where a feedforward architecture processes in a single processing step a global temporal scope representation which includes all time slices.

90. **Softmax.** Generalization of sigmoid (logistic) function to case of multiple classes. Used as an output activation function for multiclass single-label classification.

91. **Sparse autoencoder.** An autoencoder with a sparsity constraint s.t. its hidden layer units are inactive most of time. Objective: enforce specialization of each unit in hidden layer as a specific feature detector.

92. **Spectrogram.** A visual representation of a spectrum of an audio signal obtained via a Fourier transform.

93. **Stacked autoencoder.** A set of hierarchically nested autoencoders with decreasing numbers of hidden layer units.

94. **Strategy.** Way architecture will process representations in order to generate objective while matching desired requirements. Examples of types of strategy: single-step feedforward, iterative feedforward & decoder feedforward.

95. **Stride.** For a convolutional architecture, number of pixels by which slide filter matrix over input matrix.

96. **Style transfer.** Technique for capturing a style (e.g., of a given painting, by capturing correlations between neurons for each layer) & applying it onto another content.

97. **Supervised learning.** A category of ML where for each training example a target information (a scalar value in case of a regression & a class in case of a classification) is provided.

98. **Support vector machine (SVM).** A class of supervised ML models for linear classification with optimization of separation margin. A kernel method is usually associated to a SVM in order to transform initial nonlinear classification problem into a linear classification problem within a higher dimension space.

99. **Tanh (aka Hyperbolic tangent).** Hyperbolic tangent function, which may be used as a hidden layer nonlinear activation function.

100. **Test set (aka Validation set).** Subset of examples (dataset) which are used for evaluating ability of learnt model to generalize, i.e., predict or to classify properly in presence of yet unseen data.

101. **Time slice.** Time interval considered as an atomic portion (grain) of temporal representation used by an ANN architecture.

102. **Time step.** Atomic increment of time considered by an ANN architecture.

103. **Training set.** Subset of examples (dataset) which are used for training ANN architecture.

104. **Transfer learning.** An area of ML concerned with ability to reuse what has been learnt & apply (transfer) it to related domains or tasks.

105. **Turing test.** Initially codified in 1950 by Alan Turing & named by him "imitation game", "Turing test" is a test of ability for a machine to exhibit intelligent behavior equivalent to (& more precisely, indistinguishable from) behavior of a human. In his imaginary experimental setting, Turing proposed test to be a natural language conversation between a human (evaluator) & a hidden actor (another human or a machine). If evaluator cannot reliably tell machine from human, machine is said to have passed test.

106. **Unit.** see neuron.

107. **Unit selection strategy.** A strategy for content generation about querying successive musical units (e.g., 1 measure long melody segments) from a database & concatenating them in order to generate a sequence according to some user characteristics.

108. **Unsupervised learning.** A category of ML which extracts information from data without any added label or class information.

109. **Variational autoencoder (VAE).** An autoencoder with added constraint that encoded representation (its latent variables) follow some prior probability distribution, usually a Gaussian distribution. Variational autoencoder is therefore able to learn a "smooth" latent space mapping to realistic examples which provides interesting ways to control variation of generation.

110. **Value encoding.** Direct encoding of a numerical value as a scalar.

111. **Vanishing or exploding gradient problem.** A known problem when training a RNN caused by difficulty of estimating gradients, because, in backpropagation through time, recurrence brings repetitive multiplications & could thus lead to over amplify or minimize effects (numerical errors). LSTM architecture solved problem.

112. **Waveform.** Raw representation of a signal as evolution of its amplitude in time.

113. **Weight.** A numerical parameter associated to a connection between a node (neuron or not) & a unit (neuron). A neuron will compute weighted sum of activations of its connections & then apply its associated activation function. Weights will be adjusted during training phase.

114. **Zero-padding.** For a convolutional architecture, padding of input matrix with 0s around its border.

1.3 Chaitanya K. Joshi. **Transformers are Graph Neural Networks**

- Abstract. Establish connections between Transformer architecture, originally introduced for NLP, & Graph Neural Networks (GNNs) for representation learning on graphs. Show how Transformers can be viewed as message passing GNNs operating on fully connected graphs of tokens, where self-attention mechanism capture relative importance of all tokens w.r.t. each-other, & positional encodings provide hints about sequential ordering or structure. Thus, Transformers are expressive set processing networks that *learn* relationships among input elements without being constrained by apriori graphs. Despite this mathematical connection to GNNs, Transformers are implemented via dense matrix operations that are significantly more efficient on modern hardware than sparse message passing. This leads to perspective: Transformer are GNNs currently winning hardware lottery.

  – Thiết lập kết nối giữa kiến trúc Transformer, ban đầu được giới thiệu cho NLP, & Mạng nơ-ron đồ thị (GNN) để học biểu diễn trên đồ thị. Hiển thị cách Transformer có thể được xem như GNN truyền tin hoạt động trên đồ thị được kết nối đầy đủ của các mã thông báo, trong đó cơ chế tự chú ý nắm bắt tầm quan trọng tương đối của tất cả các mã thông báo so với nhau, & mã hóa vị trí cung cấp gợi ý về thứ tự hoặc cấu trúc tuần tự. Do đó, Transformer là mạng xử lý tập hợp biểu đạt *học* các mối quan hệ giữa các phần tử đầu vào mà không bị ràng buộc bởi đồ thị apriori. Bất chấp kết nối toán học này với GNN, Transformer được triển khai thông qua các hoạt động ma trận dày đặc hiệu quả hơn đáng kể trên phần cứng hiện đại so với truyền tin thưa thớt. Điều này dẫn đến quan điểm: Transformer là GNN hiện đang chiến thắng xổ số phần cứng.

- 1. Transformers for NLP. Discussion focuses on *representation learning*, which is foundation for any ML task, be it predictive or generative modeling [Olah2014].

  At a high level, deep neural networks compress statistical & semantic information about data into a list of numbers, called a *latent representation* or *embedding*. Models are trained by optimizing a loss function that measures how well model's representations perform on a task of interest, e.g. predicting some properties of input data. E.g., if give a model a dataset of sentences & train it to predict next word in each sentence, it will learn to build representations of each word that capture its meaning & context in sentence [Graves2013].

  – Ở cấp độ cao, mạng nơ-ron sâu nén thông tin thống kê & ngữ nghĩa về dữ liệu thành 1 danh sách các số, được gọi là *biểu diễn tiềm ẩn* hoặc *nhúng*. Các mô hình được đào tạo bằng cách tối ưu hóa hàm mất mát đo lường mức độ biểu diễn của mô hình thực hiện tốt như thế nào đối với 1 nhiệm vụ quan tâm, e.g., dự đoán 1 số thuộc tính của dữ liệu đầu vào. E.g., nếu cung cấp cho 1 mô hình 1 tập dữ liệu các câu & đào tạo nó để dự đoán từ tiếp theo trong mỗi câu, nó sẽ học cách xây dựng các biểu diễn của mỗi từ nắm bắt được ý nghĩa của nó & ngữ cảnh trong câu [Graves2013].

  When trained on diverse but interconnected data sources, models learn to build general-purpose representations that capture underlying structure of data. Good representations enable generalization to new tasks via knowledge transfer across related domains [Radford et al., 2019, Raffel et al., 2020]. E.g., training a model on mathematical tasks may also improve it for programming related tasks, as both domains require abstract problem-solving.

– Khi được đào tạo trên các nguồn dữ liệu đa dạng nhưng có sự kết nối, các mô hình học cách xây dựng các biểu diễn mục đích chung nắm bắt cấu trúc cơ bản của dữ liệu. Các biểu diễn tốt cho phép khái quát hóa các nhiệm vụ mới thông qua việc chuyển giao kiến thức trên các miền liên quan [Radford và cộng sự, 2019, Raffel và cộng sự, 2020]. E.g., việc đào tạo 1 mô hình về các nhiệm vụ toán học cũng có thể cải thiện mô hình đó cho các nhiệm vụ liên quan đến lập trình, vì cả hai miền đều yêu cầu giải quyết vấn đề trừu tượng.

A key ingredient for building good representations is a highly expressive & scalable model architecture. This can be understood by studying at transformation of architectures in context of NLP.

– 1 thành phần chính để xây dựng các biểu diễn tốt là kiến trúc mô hình có khả năng biểu đạt cao & có thể mở rộng. Điều này có thể được hiểu bằng cách nghiên cứu chuyển đổi kiến trúc trong bối cảnh NLP.

- **From RNNs to Transformers.** RNNs were a class of widely used DL architectures for NLP [Hochreiter and Schmidhuber, 1997, Sutskever et al., 2014]. RNNs build representations of each word in a sentence in a sequential manner, i.e., 1 word at a time. Intuitively, can imagine an RNN layer as a conveyor belt, with words being processed on it *autoregressively* from left to right. At end, get a latent embedding for each word in sentence, which we pass to next RNN layer or use for our tasks of choice.

  – RNN là 1 lớp kiến trúc DL được sử dụng rộng rãi cho NLP [Hochreiter và Schmidhuber, 1997, Sutskever và cộng sự, 2014]. RNN xây dựng các biểu diễn của từng từ trong 1 câu theo cách tuần tự, tức là, 1 từ tại 1 thời điểm. Theo trực giác, có thể tưởng tượng 1 lớp RNN như 1 băng chuyền, với các từ được xử lý trên đó *tự hồi quy* từ trái sang phải. Cuối cùng, có được 1 nhúng tiềm ẩn cho mỗi từ trong câu, mà chúng ta chuyển đến lớp RNN tiếp theo hoặc sử dụng cho các tác vụ mà chúng ta lựa chọn.

  However, sequential nature of RNNs means they struggle with long input contexts, as they compress entire sentence into a single fixed-length representation at end of conveyor belt. This lead to design of *attention mechanism* [Bahdanau et al., 2015], which allows RNNs to focus on different parts of input sentence when building representations at each step, rather than just representation of last word processed.

  – Tuy nhiên, bản chất tuần tự của RNN có nghĩa là chúng gặp khó khăn với các ngữ cảnh đầu vào dài, vì chúng nén toàn bộ câu thành 1 biểu diễn có độ dài cố định duy nhất ở cuối băng chuyền. Điều này dẫn đến thiết kế cơ chế chú ý [Bahdanau et al., 2015], cho phép RNN tập trung vào các phần khác nhau của câu đầu vào khi xây dựng biểu diễn ở mỗi bước, thay vì chỉ biểu diễn từ cuối cùng được xử lý.

  Fig. 1: **Representation Learning for NLP.** RNNs build representations 1 token at a time, which captures sequential nature of language. Transformers build representations in parallel via attention mechanisms, which capture relative importance of words w.r.t. each-other. – **Học biểu diễn cho NLP.** RNN xây dựng các biểu diễn 1 mã thông báo tại 1 thời điểm, nắm bắt được bản chất tuần tự của ngôn ngữ. Transformers xây dựng các biểu diễn song song thông qua các cơ chế chú ý, nắm bắt được tầm quan trọng tương đối của các từ liên quan đến nhau.

  Transformer networks [Vaswani et al., 2017: Attention is all you need], which are built on top of attention mechanism, take this idea further by allowing model to build representations of each word in parallel, rather than sequentially. This is done by computing importance of each word in sentence w.r.t. each other word, & then updating representation of each word based on this importance. This allows model to capture long-range dependencies & relationships between words, leading to more expressive representations.

  – Mạng lưới biến áp [Vaswani và cộng sự, 2017: Attention is all you need], được xây dựng trên cơ chế attention, đưa ý tưởng này đi xa hơn bằng cách cho phép mô hình xây dựng các biểu diễn của từng từ song song, thay vì tuần tự. Điều này được thực hiện bằng cách tính toán tầm quan trọng của từng từ trong câu so với từng từ khác, & sau đó cập nhật biểu diễn của từng từ dựa trên tầm quan trọng này. Điều này cho phép mô hình nắm bắt các mối quan hệ phụ thuộc & tầm xa giữa các từ, dẫn đến các biểu diễn biểu cảm hơn.

  Initially introduced for machine translation [1 of motivations for Transformers was: it was inefficient to train RNNs on large internal datasets that Google had for language translation. They need to develop an expressive architecture that could scale better than RNNs.], Transformers have replaced RNNs as architecture of choice across NLP [Achiam et al., 2023] & wide DL applications [Dosovitskiy et al., 2021, Radford et al., 2023] due to their expressivity & scalability.

  – Ban đầu được giới thiệu cho dịch máy [1 trong những động lực của Transformers là: việc đào tạo RNN trên các tập dữ liệu nội bộ lớn mà Google có để dịch ngôn ngữ là không hiệu quả. Họ cần phát triển 1 kiến trúc biểu đạt có thể mở rộng tốt hơn RNN.], Transformers đã thay thế RNN làm kiến trúc được lựa chọn trên NLP [Achiam và cộng sự, 2023] & các ứng dụng DL rộng rãi [Dosovitskiy và cộng sự, 2021, Radford và cộng sự, 2023] do khả năng biểu đạt & khả năng mở rộng của chúng.

- **Attention mechanism.** Central component of Transformer is attention mechanism, which allows representations of words in sentences to capture their relative importance w.r.t. each-other; see Weng [2018] for an intuitive introduction.

  – **Cơ chế chú ý.** Thành phần trung tâm của Transformer là cơ chế chú ý, cho phép biểu diễn các từ trong câu nắm bắt được tầm quan trọng tương đối của chúng so với nhau; xem Weng [2018] để biết phần giới thiệu trực quan.

  Formally, given a sentence $\mathcal{S}$ consisting of an ordered set of n words (or tokens, more generally). For each token i, initialize its representation $h_i^{l=0} \in \mathbb{R}^d$ as an initial token embedding [Mikolov et al., 2013]. Token representations for each token i are then updated via an attention mechanism from any arbitrary layer l to layer $l+1$ as follows: (1)–(2)

$$h_i^{l+1} = \text{Attention}(Q = W_Q^l h_i^l,\ K = \{W_K^l h_j^l,\ \forall j \in \mathcal{S}\},\ V = \{W_V^l h_j^l,\ \forall j \in \mathcal{S}\}) = \sum_{j \in \mathcal{S}} w_{ij} \cdot W_V^l h_j^l,$$

where $j \in \mathcal{S}$ denotes set of all tokens in sentence (including token i itself), & $W_Q^l, W_K^l, W_V^l \in \mathbb{R}^{d \times d}$ are learnable linear transformations denoting Query, Key, & Value fro attention computation, resp. Attention weights $w_{ij} \in \mathbb{R}$, which captures relative importance between each pair of tokens (i,j), are computed via a dot-product of linearly transformed representations, followed by a softmax normalization across all tokens $j \in \mathcal{S}$: (3)–(4)

$$w_{ij} = \text{softmax}_{j \in \mathcal{S}}(W_Q^l h_i^l \cdot W_K^l h_j^l) = \frac{\exp\left(W_Q^l h_i^l \cdot W_K^l h_j^l\right)}{\sum_{j' \in \mathcal{S}} \exp\left(W_Q^l h_i^l \cdot W_K^l h_{j'}^l\right)}.$$

Fig. 2: **A simple attention mechanism.** Taking as input representations of token h_i^l & set of other tokens in sentence $\{h_j^l; \forall j \in \mathcal{S}\}$, compute attention weights w_{ij} denoting relative importance for each pair (i,j) through dot-product followed by a softmax normalization. Finally, produce updated token representation h_i^{l+1} illustrates described attention mechanism, which is a slightly simplified version of one used in Transformers. Attention computation in (1) is performed in parallel for each token in sentence to obtain updated representations in *1 shot*. This is a key advantage of Transformers over RNNs, which update representations token-by-token.

– Về mặt hình thức, cho 1 câu $\mathcal{S}$ bao gồm 1 tập hợp có thứ tự gồm n từ (hoặc các mã thông báo, nói chung hơn). Đối với mỗi mã thông báo i, hãy khởi tạo biểu diễn $h_i^{l=0} \in \mathbb{R}^d$ của nó như 1 nhúng mã thông báo ban đầu [Mikolov et al., 2013]. Biểu diễn mã thông báo cho mỗi mã thông báo i sau đó được cập nhật thông qua cơ chế chú ý từ bất kỳ lớp l tùy ý nào đến lớp $l+1$ như sau: (1)–(2)

$$h_i^{l+1} = \text{Attention}(Q = W_Q^l h_i^l,\ K = \{W_K^l h_j^l,\ \forall j \in \mathcal{S}\},\ V = \{W_V^l h_j^l,\ \forall j \in \mathcal{S}\}) = \sum_{j \in \mathcal{S}} w_{ij} \cdot W_V^l h_j^l,$$

trong đó $j \in \mathcal{S}$ biểu thị tập hợp tất cả các mã thông báo trong câu (bao gồm cả mã thông báo i), & $W_Q^l, W_K^l, W_V^l \in \mathbb{R}^{d \times d}$ là các phép biến đổi tuyến tính có thể học được biểu thị Truy vấn, Khóa, & Giá trị cho tính toán chú ý, tương ứng. Trọng số chú ý $w_{ij} \in \mathbb{R}$, nắm bắt tầm quan trọng tương đối giữa mỗi cặp mã thông báo (i,j), được tính toán thông qua tích vô hướng của các biểu diễn được biến đổi tuyến tính, theo sau là chuẩn hóa softmax trên tất cả các mã thông báo $j \in \mathcal{S}$: (3)–(4)

$$w_{ij} = \text{softmax}_{j \in \mathcal{S}}(W_Q^l h_i^l \cdot W_K^l h_j^l) = \frac{\exp\left(W_Q^l h_i^l \cdot W_K^l h_j^l\right)}{\sum_{j' \in \mathcal{S}} \exp\left(W_Q^l h_i^l \cdot W_K^l h_{j'}^l\right)}.$$

Hình 2: **Một cơ chế chú ý đơn giản.** Lấy các biểu diễn đầu vào của token h_i^l & tập hợp các token khác trong câu $\{h_j^l; \forall j \in \mathcal{S}\}$, tính toán trọng số chú ý w_{ij} biểu thị tầm quan trọng tương đối cho mỗi cặp (i,j) thông qua tích vô hướng theo sau là chuẩn hóa softmax. Cuối cùng, tạo ra biểu diễn token được cập nhật h_i^{l+1} minh họa cho cơ chế chú ý được mô tả, đây là phiên bản đơn giản hóa 1 chút của cơ chế được sử dụng trong Transformers. Tính toán chú ý trong (1) được thực hiện song song cho mỗi token trong câu để có được các biểu diễn được cập nhật trong *1 shot*. Đây là 1 lợi thế chính của Transformers so với RNN, nơi cập nhật các biểu diễn token-by-token.

- **Multihead attention.** In practice, using a single attention head can be limiting, as it forces model to learn only 1 set of weights that capture relationships among tokens. However, tokens can interact in multiple ways, & we may want to simultaneously capture different aspects of their relationships, e.g. syntactic & semantic dependencies as well as contextual connections.

  – **Chú ý nhiều đầu.** Trong thực tế, việc sử dụng 1 đầu chú ý duy nhất có thể bị hạn chế, vì nó buộc mô hình phải học chỉ 1 tập trọng số nắm bắt mối quan hệ giữa các mã thông báo. Tuy nhiên, các mã thông báo có thể tương tác theo nhiều cách, & chúng ta có thể muốn nắm bắt đồng thời các khía cạnh khác nhau của mối quan hệ của chúng, ví dụ như sự phụ thuộc cú pháp & ngữ nghĩa cũng như các kết nối theo ngữ cảnh.

  To improve expressivity of attention mechanism, Transformers introduced *multihead attention*, which computes multiple attention weights in parallel. Each attention head learns its own set of query, key, & value transformations, allowing model to attention to different types of relationships simultaneously & 'hedge its bets' on most relevant representations for each token.

  – Để cải thiện khả năng biểu đạt của cơ chế chú ý, Transformers đã giới thiệu chú ý đa đầu, tính toán nhiều trọng số chú ý song song. Mỗi đầu chú ý học tập bộ truy vấn, khóa, & chuyển đổi giá trị của riêng mình, cho phép mô hình chú ý đến các loại mối quan hệ khác nhau cùng lúc & 'bảo vệ cược của mình' trên các biểu diễn có liên quan nhất cho mỗi mã thông báo.

  Formally, can define K attention heads, where each head k computes its own set of query, key, & value transformations when updating representation of token i at layer l:

$$Q^k = W_Q^{l,k} h_i^l,\ K^k = \{W_K^{l,k} h_j^l,\ \forall j \in \mathcal{S}\},\ V^k = \{W_V^{l,k} h_j^l,\ \forall j \in \mathcal{S}\}$$

where $W_Q^{l,k}, W_K^{l,k}, W_V^{l,k} \in \mathbb{R}^{d\times\frac{d}{k}}$ are learnable linear transformations for kth attention head at layer l. Output of each attention head is then computed as

$$\mathrm{head}_i^k = \sum_{j\in\mathcal{S}} w_{ij}^k \cdot W_V^{l,k} h_j^l,$$
$$w_{ij}^k = \mathrm{softmax}_{j\in\mathcal{S}}\left(W_Q^{l,k} h_i^l \cdot W_K^{l,k} h_j^l\right),$$

where $w_{ij}^k \in \mathbb{R}$ are attention weights for kth head, computed in same way as before in (3), but using query & key transformations specific to that head. Outputs of all attention heads are concatenated & projected to produce updated representation for token i as (8)

$$\tilde{h}_i^l = \mathrm{Concat}(\mathrm{head}_i^1, \ldots, \mathrm{head}_i^K) O^l,$$

where $O^l \in \mathbb{R}^{d\times d}$ is a learnable linear transformation that projects concatenated outputs to original representation dimension d. In practice, computation for all heads is done in parallel via batched matrix multiplication.

– Về mặt hình thức, có thể định nghĩa K đầu chú ý, trong đó mỗi đầu k tính toán tập hợp các truy vấn, khóa, & chuyển đổi giá trị riêng của nó khi cập nhật biểu diễn của mã thông báo i tại lớp l:

$$Q^k = W_Q^{l,k} h_i^l,\ K^k = \{W_K^{l,k} h_j^l,\ \forall j \in \mathcal{S}\},\ V^k = \{W_V^{l,k} h_j^l,\ \forall j \in \mathcal{S}\}$$

trong đó $W_Q^{l,k}, W_K^{l,k}, W_V^{l,k} \in \mathbb{R}^{d\times\frac{d}{k}}$ là các chuyển đổi tuyến tính có thể học được đối với đầu chú ý thứ k tại lớp l. Đầu ra của mỗi đầu chú ý sau đó được tính như sau

$$\mathrm{head}_i^k = \sum_{j\in\mathcal{S}} w_{ij}^k \cdot W_V^{l,k} h_j^l,$$
$$w_{ij}^k = \mathrm{softmax}_{j\in\mathcal{S}}\left(W_Q^{l,k} h_i^l \cdot W_K^{l,k} h_j^l\right),$$

trong đó $w_{ij}^k \in \mathbb{R}$ là trọng số chú ý cho đầu thứ k, được tính theo cùng cách như trước trong (3), nhưng sử dụng truy vấn & chuyển đổi khóa cụ thể cho đầu đó. Đầu ra của tất cả các đầu chú ý được nối lại & chiếu để tạo ra biểu diễn được cập nhật cho mã thông báo i như (8)

$$\tilde{h}_i^l = \mathrm{Concat}(\mathrm{head}_i^1, \ldots, \mathrm{head}_i^K) O^l,$$

trong đó $O^l \in \mathbb{R}^{d\times d}$ là 1 phép biến đổi tuyến tính có thể học được chiếu các đầu ra được nối lại tới chiều biểu diễn ban đầu d. Trong thực tế, tính toán cho tất cả các đầu được thực hiện song song thông qua phép nhân ma trận theo lô.

p. 3+++

- 2. Graph Neural Networks for Representation Learning on Graphs. Turn attention to sets with relational structure, i.e., graphs. Graphs are used to model complex & interconnected systems in real-world, ranging from knowledge graphs to social networks & molecular structures. Formally, an attributed graph $\mathcal{G} = (\mathbf{A}, \mathbf{H})$ is a set $\mathcal{V}$ of n nodes connected by edges. $\mathbf{A}$ denotes an $n \times n$ adjacency matrix where each entry $a_{ij} \in \{0, 1\}$ indicates presence or absence of an edge connecting nodes i, j. Additionally, can define $\mathcal{N}_i$ as set of neighbors of node i, which are nodes connected to i by an edge, i.e., $\mathcal{N}_i = \{j \in \mathcal{V}; a_{ij} = 1\}$. Matrix of initial representations $\mathbf{H} \in \mathbb{R}^{n\times d}$ stores attributes $\mathbf{h}_i \in \mathbb{R}^d$ associated with each node i. E.g., in molecular graphs, each node contains information about atom type & edges can represent interactions among atoms.

  – Mạng nơ-ron đồ thị để biểu diễn học tập trên đồ thị. Chuyển sự chú ý sang các tập hợp có cấu trúc quan hệ, tức là đồ thị. Đồ thị được sử dụng để mô hình hóa các hệ thống phức tạp & được kết nối trong thế giới thực, từ đồ thị kiến thức đến mạng xã hội & cấu trúc phân tử. Về mặt hình thức, 1 đồ thị được gán $\mathcal{G} = (\mathbf{A}, \mathbf{H})$ là 1 tập $\mathcal{V}$ gồm n nút được kết nối bởi các cạnh. $\mathbf{A}$ biểu thị 1 ma trận kề $n \times n$ trong đó mỗi mục $a_{ij} \in \{0, 1\}$ biểu thị sự có mặt hoặc không có mặt của 1 cạnh kết nối các nút i, j. Ngoài ra, có thể định nghĩa $\mathcal{N}_i$ là tập hợp các lân cận của nút i, là các nút được kết nối với i bằng 1 cạnh, tức là $\mathcal{N}_i = \{j \in \mathcal{V}; a_{ij} = 1\}$. Ma trận biểu diễn ban đầu $\mathbf{H} \in \mathbb{R}^{n\times d}$ lưu trữ các thuộc tính $\mathbf{h}_i \in \mathbb{R}^d$ liên kết với mỗi nút i. Ví dụ, trong đồ thị phân tử, mỗi nút chứa thông tin về loại nguyên tử & các cạnh có thể biểu diễn các tương tác giữa các nguyên tử.

  Typically, nodes in a graph have no canonical or fixed ordering & can be shuffled arbitrarily, resulting in an equivalent shuffling of rows & columns of adjacency matrix $\mathbf{A}$. Thus, accounting for permutation symmetry is a critical consideration when designing ML models for graphs [Bronstein et al., 2021]. Formally, a permutation σ of nodes acts on graph $\mathcal{G}$ by permuting rows & columns of adjacency matrix $\mathbf{A}$ & representations $\mathbf{H}$: (10)

$$\mathbf{P}_\sigma \mathcal{G} = (\mathbf{P}_\sigma \mathbf{A} \mathbf{P}_\sigma^\top, \mathbf{P}_\sigma \mathbf{H}),$$

  where $\mathbf{P}_\sigma$ is a permutation matrix corresponding to σ, & $\mathbf{P}_\sigma \in \mathbb{R}^{n\times n}$ has exactly 1 number 1 in every row & column, & 0 elsewhere.

  – Thông thường, các nút trong đồ thị không có thứ tự chuẩn hoặc cố định & có thể được xáo trộn tùy ý, dẫn đến việc xáo trộn tương đương các hàng & cột của ma trận kề $\mathbf{A}$. Do đó, việc tính đến tính đối xứng hoán vị là 1 cân nhắc quan trọng khi thiết kế các mô hình ML cho đồ thị [Bronstein và cộng sự, 2021]. Về mặt hình thức, 1 hoán vị σ của các nút tác động lên đồ thị $\mathcal{G}$ bằng cách hoán vị các hàng & cột của ma trận kề $\mathbf{A}$ & biểu diễn $\mathbf{H}$: (10)

$$\mathbf{P}_\sigma \mathcal{G} = (\mathbf{P}_\sigma \mathbf{A} \mathbf{P}_\sigma^\top, \mathbf{P}_\sigma \mathbf{H}),$$

  trong đó $\mathbf{P}_\sigma$ là ma trận hoán vị tương ứng với σ, & $\mathbf{P}_\sigma \in \mathbb{R}^{n\times n}$ có đúng 1 số 1 trong mỗi hàng & cột, & 0 ở những nơi khác.

- **Message Passing Graph Neural Networks.** GNNs are a class of DL architectures designed to operate on graph-structured data. GNNs leverage graph topology to propagate & aggregate information between connected nodes. They have emerged as architecture of choice for representation learning on graph data across domains ranging from molecular modeling [Stokes et al., 2020, Batzner et al., 2022] to recommendation systems [Ying et al., 2018] & transportation networks [Derrow-Pinion et al., 2021].

  – **Mạng nơ-ron đồ thị truyền tin.** GNN là 1 lớp kiến trúc DL được thiết kế để hoạt động trên dữ liệu có cấu trúc đồ thị. GNN tận dụng cấu trúc đồ thị để truyền bá & tổng hợp thông tin giữa các nút được kết nối. Chúng đã nổi lên như 1 kiến trúc được lựa chọn để học biểu diễn trên dữ liệu đồ thị trên nhiều miền, từ mô hình phân tử [Stokes và cộng sự, 2020, Batzner và cộng sự, 2022] đến các hệ thống đề xuất [Ying và cộng sự, 2018] & mạng lưới giao thông [Derrow-Pinion và cộng sự, 2021].

  GNNs are based on principle of *message passing*, where each node iteratively updates its representations by aggregating from its local neighbors [Battaglia et al., 2018]. Formally, node representations $\mathbf{h}_i$ for each node $i \in \mathcal{V}$ are updated from layer l to $l+1$ through a 3-step process:

  1. *Message construction*: For each node i & its neighbors $j \in \mathcal{N}_i$, construct a message $\mathbf{m}_{ij}^l$ that captures relationship between representations of nodes i, j. (11)
  $$\mathbf{m}_{ij}^l = \psi(\mathbf{h}_i^l, \mathbf{h}_j^l),\ \forall j \in \mathcal{N}_i,$$
  where $\psi : \mathbb{R}^{2\times d} \to \mathbb{R}^d$ is an MLP that learns to construct message based on representations of nodes i, j.
  2. *Aggregation*: Combine all messages from neighbors of node i to produce a single aggregated message $\mathbf{m}_i^l$.
  $$\mathbf{m}_i^l = \bigoplus_{j\in\mathcal{N}_i} \mathbf{m}_{ij}^l,$$
  where $\bigoplus$ is a permutation-invariant operator (e.g., sum, mean, max) that aggregates messages from all neighbors $j \in \mathcal{N}_i$. Thus, a change in order of neighbors does not affect aggregated message, preserving graph's symmetry (eqn (10)).
  3. *Update*: Update representations of node i using aggregated message $\mathbf{m}_i^l$ & its previous representations $\mathbf{h}_i^l$. (13)
  $$\mathbf{h}_i^{l+1} = \phi(\mathbf{h}_i^l, \mathbf{m}_i^l),$$
  where $\phi : \mathbb{R}^d \to \mathbb{R}^d$ is another MLP.

  – GNN dựa trên nguyên tắc *message passing*, trong đó mỗi nút cập nhật lặp lại các biểu diễn của nó bằng cách tổng hợp từ các nút lân cận cục bộ của nó [Battaglia et al., 2018]. Về mặt hình thức, các biểu diễn nút $\mathbf{h}_i$ cho mỗi nút $i \in \mathcal{V}$ được cập nhật từ lớp l đến $l+1$ thông qua quy trình 3 bước:

  1. *Xây dựng thông điệp*: Đối với mỗi nút i & các nút lân cận $j \in \mathcal{N}_i$, hãy xây dựng 1 thông điệp $\mathbf{m}_{ij}^l$ nắm bắt mối quan hệ giữa các biểu diễn của các nút i, j. (11)
  $$\mathbf{m}_{ij}^l = \psi(\mathbf{h}_i^l, \mathbf{h}_j^l),\ \forall j \in \mathcal{N}_i,$$
  trong đó $\psi : \mathbb{R}^{2\times d} \to \mathbb{R}^d$ là 1 MLP học cách xây dựng thông điệp dựa trên các biểu diễn của các nút i, j.
  2. *Tổng hợp*: Kết hợp tất cả các thông điệp từ các nút lân cận của nút i để tạo ra 1 thông điệp tổng hợp duy nhất $\mathbf{m}_i^l$.
  $$\mathbf{m}_i^l = \bigoplus_{j\in\mathcal{N}_i} \mathbf{m}_{ij}^l,$$
  trong đó $\bigoplus$ là toán tử bất biến hoán vị (ví dụ: tổng, trung bình, tối đa) tổng hợp các thông điệp từ tất cả các hàng xóm $j \in \mathcal{N}_i$. Do đó, việc thay đổi thứ tự của các hàng xóm không ảnh hưởng đến thông điệp tổng hợp, bảo toàn tính đối xứng của đồ thị (phương trình (10)).
  3. *Cập nhật*: Cập nhật các biểu diễn của nút i bằng cách sử dụng thông điệp tổng hợp $\mathbf{m}_i^l$ & các biểu diễn trước đó của nó $\mathbf{h}_i^l$. (13)
  $$\mathbf{h}_i^{l+1} = \phi(\mathbf{h}_i^l, \mathbf{m}_i^l),$$
  trong đó $\phi : \mathbb{R}^d \to \mathbb{R}^d$ là 1 MLP khác.

  This general formulation encompasses most commonly used GNN architectures including Graph Convolutional Networks [Kipf and Welling, 2017], Graph Isomorphism Networks [Xu et al., 2019], & Message Passing Neural Networks [Gilmer et al., 2017]. By stacking multiple message passing layers, GNNs can propagate information beyond immediate neighbors & capture complex multi-hop relationships in graph structure Fig. 4: **Representation learning on graphs with message passing.** (a) Graphs model complex systems via a set of nodes connected by edges. (b) GNNs build latent representations of graph data via message passing, where each node learns to aggregate representations from its local neighborhood. (c) Stacking L message passing layers enables GNNs to send & aggregate information from L-hop subgraphs around each node.

  – Công thức chung này bao gồm các kiến trúc GNN được sử dụng phổ biến nhất bao gồm Mạng tích chập đồ thị [Kipf và Welling, 2017], Mạng đẳng cấu đồ thị [Xu và cộng sự, 2019], & Mạng nơ-ron truyền tin nhắn [Gilmer và cộng sự, 2017]. Bằng cách xếp chồng nhiều lớp truyền tin nhắn, GNN có thể truyền thông tin vượt ra ngoài các lân cận liền kề & nắm bắt các mối quan hệ đa bước nhảy phức tạp trong cấu trúc đồ thị Hình 4: **Học biểu diễn trên đồ thị với truyền tin nhắn.** (a) Đồ thị mô hình hóa các hệ thống phức tạp thông qua 1 tập hợp các nút được kết nối bởi các cạnh. (b) GNN xây dựng các biểu diễn tiềm ẩn của dữ liệu đồ thị thông qua truyền tin nhắn, trong đó mỗi nút học cách tổng hợp các biểu diễn từ vùng lân cận cục bộ của nó. (c) Xếp chồng L lớp truyền tin nhắn cho phép GNN gửi & tổng hợp thông tin từ các đồ thị con L-bước nhảy xung quanh mỗi nút.

- ◦ **Graph Attention Networks.** A particularly interesting class of GNNs uses attention mechanisms to weight importance of different neighbors during aggregation [Veličković et al., 2018]. In Graph Attention Networks (GATs), message from neighbor j to node i is computed as (14)–(15)

$$\psi(\mathbf{h}_i^l, \mathbf{h}_j^l) = \mathrm{LocalAttention}(W_Q^l \mathbf{h}_i^l, \{W_K^l \mathbf{h}_j^l,\ \forall j \in \mathcal{N}_i\}, \{W_V^l \mathbf{h}_j^l,\ \forall j \in \mathcal{N}_i\}) = \frac{\exp\left(W_Q^l \mathbf{h}_i^l \cdot W_K^l \mathbf{h}_j^l\right)}{\sum_{j' \in \mathcal{N}_i} \exp\left(W_Q^l \mathbf{h}_i^l \cdot W_K^l \mathbf{h}_{j'}^l\right)} \cdot W_V^l \mathbf{h}_j^l,$$

  where $W_Q^l, W_K^l, W_V^l \in \mathbb{R}^{d\times d}$. Local attention mechanism allows GATs to learn which neighbors are more important for each node during aggregation step. Updated representation for node i is computed by aggregating messages from all its neighbors: (16)

$$\mathbf{h}_i^{l+1} = \mathbf{h}_i^l + \sum_{j\in\mathcal{N}_i} \psi(\mathbf{h}_i^l, \mathbf{h}_j^l).$$

  In practice, GATs also use multihead attention to compute multiple sets of attention weights in parallel, allowing model to learn different aspects of relationships between nodes.

  – **Mạng chú ý đồ thị.** Một lớp GNN đặc biệt thú vị sử dụng các cơ chế chú ý để đánh giá tầm quan trọng của các láng giềng khác nhau trong quá trình tổng hợp [Veličković et al., 2018]. Trong Mạng chú ý đồ thị (GAT), thông điệp từ hàng xóm j đến nút i được tính như sau (14)–(15)

$$\psi(\mathbf{h}_i^l, \mathbf{h}_j^l) = \mathrm{LocalAttention}(W_Q^l \mathbf{h}_i^l, \{W_K^l \mathbf{h}_j^l,\ \forall j \in \mathcal{N}_i\}, \{W_V^l \mathbf{h}_j^l,\ \forall j \in \mathcal{N}_i\}) = \frac{\exp\left(W_Q^l \mathbf{h}_i^l \cdot W_K^l \mathbf{h}_j^l\right)}{\sum_{j' \in \mathcal{N}_i} \exp\left(W_Q^l \mathbf{h}_i^l \cdot W_K^l \mathbf{h}_{j'}^l\right)} \cdot W_V^l \mathbf{h}_j^l,$$

  trong đó $W_Q^l, W_K^l, W_V^l \in \mathbb{R}^{d\times d}$. Cơ chế chú ý cục bộ cho phép GAT tìm hiểu xem những người hàng xóm nào quan trọng hơn đối với mỗi nút trong bước tổng hợp. Biểu diễn cập nhật cho nút i được tính bằng cách tổng hợp các thông điệp từ tất cả những người hàng xóm của nó: (16)

$$\mathbf{h}_i^{l+1} = \mathbf{h}_i^l + \sum_{j\in\mathcal{N}_i} \psi(\mathbf{h}_i^l, \mathbf{h}_j^l).$$

  Trong thực tế, GAT cũng sử dụng sự chú ý đa đầu để tính toán nhiều bộ trọng số chú ý song song, cho phép mô hình tìm hiểu các khía cạnh khác nhau của mối quan hệ giữa các nút.

  These equations should look very familiar. In fact, they are almost identical to Transformer's attention mechanism for computing relative importance of words in a sentence.

  – Những phương trình này trông rất quen thuộc. Trên thực tế, chúng gần giống với cơ chế chú ý của Transformer để tính toán tầm quan trọng tương đối của các từ trong 1 câu.

- 3. Transformers are GNNs over Fully Connected Graphs. Establish a formal equivalence between multihead attention in Transformers & message passing in Graph Attention Networks (GATs).

  – Bộ chuyển đổi là GNN trên Đồ thị được kết nối đầy đủ. Thiết lập sự tương đương chính thức giữa sự chú ý đa đầu trong Bộ chuyển đổi & truyền tin nhắn trong Mạng chú ý đồ thị (GAT).

  Transformers can be viewed as GNNs operating on complete graphs, where self-attention models relationships between all input tokens in a sentence $\mathcal{S}$. Multihead attention in (8) can be directly instantiated in message passing framework as follows (for each head): (17)–(18)

$$\psi(\mathbf{h}_i^l, \mathbf{h}_j^l) = \mathrm{GlobalAttention}(W_Q^l \mathbf{h}_i^l, \{W_K^l \mathbf{h}_j^l,\ \forall j \in \mathcal{S}\}, \{W_V^l \mathbf{h}_j^l,\ \forall j \in \mathcal{S}\}) = \frac{\exp\left(W_Q^l \mathbf{h}_i^l \cdot W_K^l \mathbf{h}_j^l\right)}{\sum_{j' \in \mathcal{S}} \exp\left(W_Q^l \mathbf{h}_i^l \cdot W_K^l \mathbf{h}_{j'}^l\right)} \cdot W_V^l \mathbf{h}_j^l.$$

- 4. Transformers are GNNs Winning Hardware Lottery.

1.4 [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 1 (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)

$$X_{t_{n+1}} - X_{t_n} \approx \mu(t_n, X_n)\Delta t_n + \sigma(t_n, X_{t_n})\Delta W_n,$$
$$u(t_{n+1}, X_{t_{n+1}}) - u(t_n, X_{t_n}) \approx -f(t_n, X_{t_n}, u(t_n, X_{t_n}), \sigma^\top(t_n, X_{t_n})\nabla u(t_n, X_{t_n}))\Delta t_n + [\nabla u(t_n, X_{t_n})]^\top \sigma(t_n, X_{t_n})\Delta W_n,$$

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.

1.5 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)

$$y_m = \mathcal{E}(\mathbf{x}_m). \tag{5}$$

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

$$(\mathbf{x}_1, y_1) = (\mathbf{x}_1, \mathcal{E}(\mathbf{x}_1)), \ldots, (\mathbf{x}_M, y_M) = (\mathbf{x}_M, \mathcal{E}(\mathbf{x}_M)) \in \mathbb{R}^d \times \mathbb{R}. \tag{6}$$

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

$$\mathfrak{L}(\phi) = \frac{1}{M}\left(\sum_{i=1}^{M} |\phi(\mathbf{x}_i) - y_m|^2\right), \ \forall \phi \in C(\mathbb{R}^d, \mathbb{R}). \tag{7}$$

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

$$\mathfrak{L} : C(\mathbb{R}^d, \mathbb{R}) \to [0, \infty). \tag{8}$$

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

$$\mathcal{L} = \mathfrak{L} \circ \psi. \tag{9}$$

Think of set (6)

$$\{\psi_\theta : \theta \in \mathbb{R}^\mathfrak{d}\} \subseteq C(\mathbb{R}^d, \mathbb{R}) \tag{10}$$

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

$$\mathbb{R}^\mathfrak{d} \ni \theta \mapsto \psi_\theta \in C(\mathbb{R}^d, \mathbb{R}) \tag{11}$$

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)

$$\psi_\theta(x) = \sum_{k=0}^{\mathfrak{d}-1} \theta_{k+1} x^k \tag{12}$$

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)

$$\{\psi_\theta : \theta \in \mathbb{R}^\mathfrak{d}\} \ni \phi \mapsto \mathfrak{L}(\phi) = \frac{1}{M}\left(\sum_{m=1}^{M} |\phi(\mathbf{x}_m) - y_m|^2\right) \in [0, \infty). \tag{13}$$

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

$$\mathbb{R}^\mathfrak{d} \ni \theta \mapsto \mathcal{L}(\theta) = \mathfrak{L}(\psi_\theta) = \frac{1}{M}\left(\sum_{m=1}^{M} |\psi_\theta(\mathbf{x}_m) - y_m|^2\right) \in [0, \infty), \tag{14}$$

& 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)

$$\psi_\vartheta(\mathbf{x}_{M+1}) \approx \mathcal{E}(\mathbf{x}_{M+1}) \tag{15}$$

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 1 (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 $\mathfrak{d} \ge s + mn + m$. Then denote by $\mathcal{A}^{\theta,s}_{m,n} : \mathbb{R}^n \to \mathbb{R}^m$ function which satisfies $\forall \mathbf{x} = (x_1, \dots, x_n) \in \mathbb{R}^n$:*

      $$\mathcal{A}^{\theta,s}_{m,n}(\mathbf{x}) = \left([\sum_{k=1}^n x_k\theta_{s+k}] + \theta_{s+mn+1}, [\sum_{k=1}^n x_k\theta_{s+n+k}] + \theta_{s+mn+2}, \dots, [\sum_{k=1}^n x_k\theta_{s+(m-1)n+k}] + \theta_{s+mn+m}\right), \tag{16}$$

      & 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 2 (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*

      $$\mathfrak{d} \ge \sum_{k=1}^L l_k(l_{k-1} + 1) \tag{17}$$

      & $\forall k \in \{1, 2, \dots, L\}$ 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 $\forall \mathbf{x} \in \mathbb{R}^{l_0}$:

      $$(\mathcal{N}^{\theta,l_0}_{\Psi_1,\dots,\Psi_L})(\mathbf{x}) = (\Psi_L \circ \mathcal{A}^{\theta,\sum_{k=1}^{L-1} l_k(l_{k-1}+1)}_{l_L,l_{L-1}} \circ \Psi_{L-1} \circ \mathcal{A}^{\theta,\sum_{k=1}^{L-2} l_k(l_{k-1}+1)}_{l_{L-1},l_{L-2}} \circ \Psi_2 \circ \mathcal{A}^{\theta,l_1(l_0+1)}_{l_2,l_1} \circ \Psi_1 \circ \mathcal{A}^{\theta,0}_{l_1,l_0})(\mathbf{x}), \tag{18}$$

& 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 2 (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* $\forall k\in\{0,1,\dots,L-1\}$*:*

$$\mathfrak{d}\ge\sum_{i=1}^{L}l_i(l_i+1),\ v_k=\sum_{i=1}^{k}l_i(l_{i-1}+1), \tag{19}$$

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* $\forall k=1,\dots,L$*:*

$$W_k=\textit{weight parameters},\ b_k=\textit{bias parameters}, \tag{20}$$

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

· *(i) it holds*

$$\mathcal{N}^{\theta,l_0}_{\Psi_1,\dots,\Psi_L}=\Psi_L\circ\mathcal{A}^{\theta,v_{L-1}}_{l_L,l_{L-1}}\circ\Psi_{L-1}\circ\mathcal{A}^{\theta,v_{L-2}}_{l_{L-1},l_{L-2}}\circ\Psi_{L-2}\circ\cdots\circ\mathcal{A}^{\theta,v_1}_{l_2,l_1}\circ\Psi_1\circ\mathcal{A}^{\theta,v_0}_{l_1,l_0}, \tag{21}$$

· *(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 3 (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*

$$\mathfrak{M}_{\psi,d_1,\dots,d_T}:\mathbb{R}^{d_1\times\cdots\times d_T}\to\mathbb{R}^{d_1\times\cdots d_T} \tag{22}$$

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*

$$\mathfrak{M}_{\psi,d_1,\dots,d_T}(\mathbf{x})=\mathbf{y}, \tag{23}$$

& call $\mathfrak{M}_{\psi,d_1,\dots,d_T}$ $d_1\times d_2\times\cdots\times d_T$-dimensional version *of* ψ.

∗ 1.2.2. Single hidden layer fully-connected feedforward ANNs. Fig. 1.3: Graphical illustration of a fully-connected feedforward ANN consisting 2 affine transformations (i.e., consisting of 3 layers: 1 input layer, 1 hidden layer, & 1 output layer) with $\mathcal{I}\in\mathbb{N}$ neurons on input layer (i.e., with $\mathcal{I}$-dimensional input layer), with $\mathcal{H}\in\mathbb{N}$ neurons on hidden layer (i.e., with $\mathcal{H}$-dimensional hidden layer), & with 1 neuron in output layer (i.e., with 1D output layer).

Lemma 1 (Fully-connected feedforward ANN with 1 hidden layer)**.** *Let* $\mathcal{I},\mathcal{H}\in\mathbb{N},\theta=(\theta_1,\dots,\theta_{\mathcal{HI}+2\mathcal{H}+1})\in\mathbb{R}^{\mathcal{HI}+2\mathcal{H}+1},\mathbf{x}=(x_1,\dots,x_{\mathcal{I}})\in\mathbb{R}^{\mathcal{I}}$ *& let* $\psi:\mathbb{R}\to\mathbb{R}$ *be a function. Then*

$$\mathcal{N}_{\mathfrak{M}_{\psi,\mathcal{H}},\mathrm{id}_{\mathbb{R}}}(\mathbf{x})=\left[\sum_{k=1}^{\mathcal{H}}\theta_{\mathcal{HI}+\mathcal{H}+k}\psi\left(\left[\sum_{i=1}^{\mathcal{I}}x_i\theta_{(k-1)\mathcal{I}+i}\right]+\theta_{\mathcal{HI}+k}\right)\right]+\theta_{\mathcal{HI}+2\mathcal{H}+1}. \tag{24}$$

∗ 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 4 (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 5 (Multidimensional ReLU activation functions)**.** *Let* $d\in\mathbb{N}$*. Then denote by* $\mathfrak{R}_d:\mathbb{R}^d\to\mathbb{R}^d$ *function given by* $\mathfrak{R}^d=\mathfrak{M}_{\mathfrak{r},d}$ *& call* $\mathfrak{R}^d$ d-dimensional ReLU activation function *(call* $\mathfrak{R}^d$ d-dimensional rectifier function*).*

Lemma 2 (An ANN with ReLU activation function as activation function)**.** *Let* $W_1=w_1=1,W_2=w_2=-1,b_1=b_2=B=0$*. Then it holds* $\forall x\in\mathbb{R}$*:*

$$x=W_1\max\{w_1x+b_1,0\}+W_2\max\{w_2x+b_2,0\}+B. \tag{25}$$

Problem 1 (Real identity)**.** *Prove or disprove following statement: There exist* $\mathfrak{d},H\in\mathbb{N},l_1,\dots,l_H\in\mathbb{N},\theta\in\mathbb{R}^{\mathfrak{d}}$ *with* $\mathfrak{d}\ge 2l_1+[\sum_{k=2}^{H}l_k(l_{k-1}+1)]+l_H+1$ *s.t.* $\forall x\in\mathbb{R}$*:* $(\mathcal{N}^{\theta,1}_{\mathfrak{R}_{l_1},\dots,\mathfrak{R}_{l_H},\mathrm{id}_{\mathbb{R}}})(x)=x$.

A partial answer:

Lemma 3 (Real identity)**.** *Let* $\theta=(1,-1,0,0,1,-1,0)\in\mathbb{R}^7$*. Then* $(\mathcal{N}^{\theta,1}_{\mathfrak{R}_2,\mathrm{id}_{\mathbb{R}}})(x)=x$.

Problem 2 (Absolute value)**.** *Prove or disprove: There exist* $\mathfrak{d},H\in\mathbb{N},l_1,\dots,l_H\in\mathbb{N},\theta\in\mathbb{R}^{\mathfrak{d}}$ *with* $\mathfrak{d}\ge 2l_1+[\sum_{k=2}^{H}l_k(l_{k-1}+1)]+l_H+1$ *s.t.* $\forall x\in\mathbb{R}$*,* $(\mathcal{N}^{\theta,1}_{\mathfrak{R}_{l_1},\dots,\mathfrak{R}_{l_H},\mathrm{id}_{\mathbb{R}}})(x)=|x|$.

Problem 3 (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 4 (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 5 (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 6 (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 7 (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 8. *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 9 (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 6 (Clipping (Cắt xén) activation function)**.** *Let* $u \in [-\infty, \infty), v \in (u, \infty]$*. Then denote by* $\mathfrak{c}_{u,v} : \mathbb{R} \to \mathbb{R}$ *function which satisfies* $\forall x \in \mathbb{R}$, $\mathfrak{c}_{u,v}(x) = \max\{u, \min\{x, v\}\}$ *& call* $\mathfrak{c}_{u,v}$ (u, v)-clipping activation function.

Fig. 1.5: A plot of $(0, 1)$-clipping activation function & ReLU activation function.

- ○ 1.3. Fully-connected feedforward ANNs (structured description).
- ○ 1.4. Convolutional ANNs (CNNs).
- ○ 1.5. Residual ANNs (ResNets).
- ○ 1.6. Recurrent ANNs (RNNs).
- ○ 1.7. Further types of ANNs.

- 2. ANN calculus.
    - ○ 2.1. Compositions of fully-connected feedforward ANNs.
    - ○ 2.2. Parallelizations of fully-connected feedforward ANNs.
    - ○ 2.3. Scalar multiplications of fully-connected feedforward ANNs.
    - ○ 2.4. Sums of fully-connected feedforward ANNs with same length.

PART II. APPROXIMATION.

- 3. 1D ANN approximation results.
- 3. Multi-dimensional ANN approximation results.

PART III. OPTIMIZATION.

- 5. Optimization through gradient flow (GF) trajectories.
- 6. Deterministic gradient descent (GD) optimization methods.
- 7. Stochastic gradient descent (SGD) optimization methods.
- 8. Backpropagation.
- 9. Kurdyka–Łojasiewicz (KL) inequalities.
- 10. ANNs with batch normalization.
- 11. Optimization through random initializations.

PART IV. GENERALIZATION.

- 12. Probabilistic generalization error estimates.
- 13. Strong generalization error estimates.

PART V. COMPOSED ERROR ANALYSIS.

- 14. Overall error decomposition.
- 15. Composed error estimates.

PART VI. DL FOR PDES.

- 16. Physics-informed neural networks (PINNs). DL methods have not only become very popular for data-driven learning problems, but are nowadays also heavily used for solving mathematical equations e.g. ODEs & PDEs (cf., e.g., [119, 187, 347, 379]). In particular, refer to overview articles [24, 56, 88, 145, 237, 355] & refs therein for numerical simulations & theoretical investigations for DL methods for PDEs.

  Often DL methods for PDEs are obtained, 1st, by reformulating PDE problem under consideration as an infinite dimensional stochastic optimization problem, then, by approximating infinite dimensional stochastic optimization problem through finite dimensional stochastic optimization problems involving deep ANNs as approximations for PDE solution &/or its derivatives, & therefore, by approximately solving resulting finite dimensional stochastic optimization problems through SGD-type optimization methods.

  Among most basic schemes of such DL learning methods for PDEs are PINNs & DGMs; see [347, 379]. In this chapter present in Thm. 16.1.1 in Sect. 16.1 a reformulation of PDE problems as stochastic optimization problems, use theoretical considerations from Sect. 16.1 to briefly sketch in Sect. 16.2 a possible derivation of PINNs & DGMs, & present in Sects. 16.3–16.4 numerical simulations for PINNs & DGMs. For simplicity & concreteness, restrict in this chap to case of semilinear heat PDEs. Specific presentation of this chap is based on Beck et al. [24].

  - 16.1. Reformulation of PDE problems as stochastic optimization problems. Both PINNs & DGMs are based on reformulations of considered PDEs as suitable infinite dimensional stochastic optimization problems. Present theoretical result behind this reformulation in special case of semilinear heat PDEs.

    Theorem 1. *Let $T \in (0,\infty), d \in \mathbb{N}, g \in C^2(\mathbb{R}^d,\mathbb{R}), u \in C^{1,2}([0,T]\times\mathbb{R}^d,\mathbb{R}), \mathfrak{t} \in C([0,T],(0,\infty)), \mathfrak{x} \in C(\mathbb{R}^d,(0,\infty))$, assume that g has at most polynomially growing partial derivatives, let $(\Omega,\mathcal{F},\mathbb{P})$ be a probability space, let $\mathcal{T}:\Omega\to[0,T]$ & $\mathcal{X}:\Omega\to\mathbb{R}^d$ be independent random variables, assume $\forall A\in\mathcal{B}([0,T]), B\in\mathcal{B}(\mathbb{R}^d)$ that*

    $$\mathbb{P}(\mathcal{T}\in A) = \int_A \mathfrak{t}(t)\,\mathrm{d}t,\ \mathbb{P}(\mathcal{X}\in B) = \int_B \mathfrak{x}(\mathbf{x})\,\mathrm{d}\mathbf{x}, \tag{26}$$

    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

    $$\mathfrak{L}(v) = \mathbb{E}[|v(0,\mathcal{X}) - g(\mathcal{X})|^2 + |(\partial_t v)(\mathcal{T},\mathcal{X}) - (\Delta_{\mathbf{x}} v)(\mathcal{T},\mathcal{X}) - f(v(\mathcal{T},\mathcal{X}))|^2]. \tag{27}$$

    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 $u(0,\mathbf{x}) = g(\mathbf{x})$ &

    $$\partial_t u(t,\mathbf{x}) = (\Delta_{\mathbf{x}} u)(t,\mathbf{x}) + f(u(t,\mathbf{x})). \tag{28}$$

  - 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 $u(0,\mathbf{x}) = g(\mathbf{x})$ &

    $$\partial_t u(t,\mathbf{x}) = (\Delta_{\mathbf{x}} u)(t,\mathbf{x}) + f(u(t,\mathbf{x})). \tag{29}$$

    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

    $$\mathbb{P}(\mathcal{T}\in A) = \int_A \mathfrak{t}(t)\,\mathrm{d}t,\ \mathbb{P}(\mathcal{X}\in B) = \int_B \mathfrak{x}(\mathbf{x})\,\mathrm{d}\mathbf{x}, \tag{30}$$

    & let $\mathfrak{L}: C^{1,2}([0,T]\times\mathbb{R}^d,\mathbb{R})\to[0,\infty]$ satisfy $\forall v = (v(t,\mathbf{x}))_{(t,\mathbf{x})\in[0,T]\times\mathbb{R}^d} \in C^{1,2}([0,T]\times\mathbb{R}^d,\mathbb{R})$:

    $$\mathfrak{L}(v) = \mathbb{E}[|v(0,\mathcal{X}) - g(\mathcal{X})|^2 + |\partial_t v(\mathcal{T},\mathcal{X}) - (\Delta_{\mathbf{x}} v)(\mathcal{T},\mathcal{X}) - f(v(\mathcal{T},\mathcal{X}))|^2]. \tag{31}$$

    Observe: Thm. 16.1.1 assures: unknown function u satisfies $\mathfrak{L}(u) = 0$ & is thus a minimizer of optimization problem associated to (16.12). Motivated by this, consider aim to find approximations of u by computing approximate minimizers of function $\mathfrak{L} : C^{1,2}([0,T]\times\mathbb{R}^d,\mathbb{R}) \to [0,\infty]$. Due to its infinite dimensionality this optimization problem is however not yet amenable to numerical computations.

For this reason, in 2nd step, reduce this infinite dimensional stochastic optimization problem to a finite dimensional stochastic optimization problem involving ANNs. Specifically, let $a : \mathbb{R} \to \mathbb{R}$ be differentiable, let $h \in \mathbb{N}, l_1, \dots, l_h, \mathfrak{d} \in \mathbb{N}$ satisfy $\mathfrak{d} = l_1(d+2) + [\sum_{k=2}^h l_k(l_{k-1}+1)] + l_h + 1$, & let $\mathcal{L} : \mathbb{R}^\mathfrak{d} \to [0,\infty)$ satisfy $\forall\theta \in \mathbb{R}^\mathfrak{d}$:

$$\mathcal{L}(\theta) = \dots \tag{32}$$

(cf. Defs. 1.1.3 & 1.2.1). Can now compute an approximate minimizer of function $\mathcal{L}$ by computing an approximate minimizer $\vartheta \in \mathbb{R}^\mathfrak{d}$ of function $\mathcal{L}$ & employing realization $\mathcal{N}_{\mathfrak{M}_{a,l_1},\mathfrak{M}_{a,l_2},\dots,\mathfrak{M}_{a,l_h},\mathrm{id}_\mathbb{R}}$ of ANN associated to this approximate minimizer as an approximate minimizer of $\mathcal{L}$.

3rd & last step of this derivation is to approximately compute such an approximate minimizer of $\mathcal{L}$ by means of SGD-type optimization methods. Now sketch this in case of plain-vanilla SGD optimization method (cf. Def. 7.2.1). Let $\xi \in \mathbb{R}^\mathfrak{d}, J \in \mathbb{N}, (\gamma_n)_{n\in\mathbb{N}} \subseteq [0,\infty)$, $\forall n \in \mathbb{N}, j \in \{1,2,\dots,J\}$ let $\mathfrak{T}_{n,j} : \Omega \to [0,T]$ & $\mathfrak{X}_{n,j} : \Omega \to \mathbb{R}^d$ be random variables, assume $\forall n \in \mathbb{N}$, $j \in \{1,\dots,J\}$, $A \in \mathcal{B}([0,T]), B \in \mathcal{B}(\mathbb{R}^d)$:

$$\mathbb{P}(\mathcal{T} \in A) = \mathbb{P}(\mathfrak{T}_{n,j} \in A),\ \mathbb{P}(\mathcal{X} \in B) = \mathbb{P}(\mathfrak{X}_{n,j} \in B), \tag{33}$$

[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.

1.6 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

    $$\mathbb{R}^d \ni \mathbf{x} \mapsto \nu(\mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x} + b), \tag{34}$$

    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 $(x_1,\ldots,x_6) = \mathbf{x}$ 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 $w_1,\ldots,w_d, b \in \mathbb{R}$.

    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

    $$\mathbb{R}^d \ni \mathbf{x} \mapsto \Phi(\mathbf{x}) = T_1 \circ \sigma \circ T_0(\mathbf{x}), \tag{35}$$

    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 $(x_1,\ldots,x_6) = \mathbf{x} \mapsto \mathbf{W}\mathbf{x} + \mathbf{b}$, 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

    $$\mathbb{R}^d \ni \mathbf{x} \mapsto \Phi(\mathbf{x}) = T_{L+1} \circ \sigma \circ \cdots T_1 \circ \sigma \circ T_0(\mathbf{x}), \tag{36}$$

    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)

    $$\Phi(\mathbf{x}_i) \approx \mathbf{y}_i,\ \forall i = 1,\ldots,m, \tag{37}$$

    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

    $$\widehat{\mathcal{R}}_S(\Phi) \coloneqq \frac{1}{m}\sum_{i=1}^m \mathcal{L}(\Phi(\mathbf{x}_i), \mathbf{y}_i). \tag{38}$$

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 $(\mathbf{x}_{\rm new}, \mathbf{y}_{\rm new})$. 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 $\mathcal{D}$, & $(\mathbf{x}_{\rm new}, \mathbf{y}_{\rm new})$ also sampled independently from $\mathcal{D}$. If want Φ_* to perform well on average, then this amounts to controlling expression

$$\mathcal{R}(\Phi_*) = \mathbb{E}_{(\mathbf{x}_{\rm new}, \mathbf{y}_{\rm new}) \sim \mathcal{D}}[\mathcal{L}(\Phi_*(\mathbf{x}_{\rm new}), \mathbf{y}_{\rm new})], \tag{39}$$

  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 $\boxed{\text{universal approximation theorem}}$, 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 $\boxed{\textit{curse of dimensionality}}$. 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 (34) & Fig. 1.1. A neural network is constructed by connecting multiple neurons. Make precise this connection procedure:

    Definition 7 (Neural network). *Let $L \in \mathbb{N}, d_0, \ldots, d_{L+1} \in \mathbb{N}$, & 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 $\mathbf{b}^{(l)} \in \mathbb{R}^{d_{l+1}}$, $l = 0, \ldots, L$, s.t. with (2.1.1)*

$$\mathbf{x}^{(0)} := \mathbf{x}, \tag{40}$$

$$\mathbf{x}^{(l)} := \sigma(\mathbf{W}^{(l-1)}\mathbf{x}^{(l-1)} + b^{(l-1)}), \ \forall l \in \{1, \ldots, L\}, \tag{41}$$

$$\mathbf{x}^{(L+1)} := \mathbf{W}^{(L)}\mathbf{x}^{(L)} + \mathbf{b}^{(L)} \tag{42}$$

holds

$$\Phi(\mathbf{x}) = \mathbf{x}^{(L+1)},\ \forall \mathbf{x} \in \mathbb{R}^{d_0}. \tag{43}$$

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 3. *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
$$\mathbf{w} = ((\mathbf{W}^{(0)}, \mathbf{b}^{(0)}), \dots, (\mathbf{W}^{(L)}, \mathbf{b}^{(L)})). \tag{44}$$
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 (44) 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 1. *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 8 (Size of neural network)**.** *Let Φ be as in Def. 2.1. Then* size *of Φ is*

$$\mathrm{size}(\Phi) := \left|\left(\{(i,k,l)|W_{k,l}^{(i)} \neq 0\} \cup \{(i,k)|b_k^{(i)} \neq 0\}\right) / \sim\right|. \tag{45}$$

- 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

    $$\sigma_{\mathrm{sig}}(x) = \frac{1}{1+e^{-x}},\ \forall x \in \mathbb{R}. \tag{46}$$

    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

    $$\frac{d}{db}\Phi_n(x) = \sigma'(\Phi_{n-1}(x))\frac{d}{db}\Phi_{n-1}(x). \tag{47}$$

    If $\sup_{x\in\mathbb{R}} |\sigma'(x)| \le 1-\delta$, then by induction, $|\frac{d}{db}\Phi_n(x)| \le (1-\delta)^n$. 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

    $$\Phi(x) = \sigma_{\mathrm{ReLU}}(b - \sigma_{\mathrm{ReLU}}(x))\ \forall x \in \mathbb{R} \tag{48}$$

    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

    $$\sigma_{\mathrm{SiLU}}(x) := x\sigma_{\mathrm{sig}}(x) = \frac{x}{1+e^{-x}},\ \forall x \in \mathbb{R}. \tag{49}$$

    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

$$\sigma_a(x) = \max\{x, ax\},\ \forall x \in \mathbb{R}, \tag{50}$$

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 10. *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 $\sigma_{\rm ReLU}$ 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 $\Phi_2(x) = \sigma_{\rm ReLU}(x)$, $\forall x \in \mathbb{R}$. (iii) Conclude: every ReLU neural network can be expressed as a leaky ReLU neural network & vice versa.*

Problem 11. *Show: for sigmoid activation functions, dead-neuron-like behavior is very rare. Let Φ be a neural network with sigmoid activation function. Assume: Φ is a constant function. Show: $\forall\varepsilon > 0$, there is a non-constant neural network $\widetilde{\Phi}$ with same architecture as Φ s.t. $\forall l = 0,\ldots,L$, $\|\mathbf{W}^{(l)} - \widetilde{\mathbf{W}}^{(l)}\| \le \varepsilon$, $\|\mathbf{b}^{(l)} - \widetilde{\mathbf{b}}^{(l)}\| \le \varepsilon$ 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 9. *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 10. *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 $\overline{\mathcal{H}}^{\rm cc}$ its closure w.r.t. compact convergence.

Relationship between a universal approximator & closure w.r.t. compact convergence is established in:

Proposition 2. *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 $C^0(\mathbb{R}^d) \subseteq \overline{\mathcal{H}}^{\rm cc}$.*

A key tool to show that a set is a universal approximator is Stone–Weierstrass theorem, see, e.g., [Rud91, Sect. 5.7].

Theorem 2 (Stone–Weierstrass)**.** *Let $d \in \mathbb{N}$, let $K \subseteq \mathbb{R}^d$ be compact, & let $\mathcal{H} \subseteq C^0(K, \mathbb{R})$ satisfy that*

(a) $\forall\mathbf{x} \in K$, there exists $f \in \mathcal{H}$ s.t. $f(\mathbf{x}) \neq 0$,

(b) $\forall\mathbf{x} \neq \mathbf{y} \in K$ there exists $f \in \mathcal{H}$ s.t. $f(\mathbf{x}) \neq f(\mathbf{y})$,

(c) $\mathcal{H}$ is an algebra of functions, i.e., $\mathcal{H}$ is closed under addition, multiplication, & scalar multiplication.

Then $\mathcal{H}$ is dense in $C^0(K)$.

Example 4 (Polynomials are dense in $C^0(\mathbb{R}^d)$)**.** *For a multiindex $\boldsymbol{\alpha} = (\alpha_1,\ldots,\alpha_d) \in \mathbb{N}_0^d$ & a vector $\mathbf{x} = (x_1,\ldots,x_d) \in \mathbb{R}^d$ denote $\mathbf{x}^{\boldsymbol{\alpha}} := \prod_{j=1}^d x_j^{\alpha_j}$. In the following, with $|\boldsymbol{\alpha}| := \sum_{i=1}^d \alpha_i$, write $\mathcal{P}_n := {\rm span}\{\mathbf{x}^{\boldsymbol{\alpha}} | \boldsymbol{\alpha} \in \mathbb{N}_0^d,\ |\boldsymbol{\alpha}| \le n\}$, i.e., $\mathcal{P}_n$ is space of polynomials of degree $\le n$ (with real coefficients). Easy to check: $\mathcal{P} := \bigcup_{n\in\mathbb{N}} \mathcal{P}_n(\mathbb{R}^d)$ 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 11. *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*

  $$\mathcal{N}_d^m(\sigma; L, n) \coloneqq \{\Phi : \mathbb{R}^d \to \mathbb{R}^m | \Phi \textit{ as in Def. 2.1}, \ \mathrm{depth}(\Phi) \le L, \ \mathrm{width}(\Phi) \le n\}. \tag{51}$$

  Furthermore,

  $$\mathcal{N}_d^m(\sigma; L) \coloneqq \bigcup_{n \in \mathbb{N}} \mathcal{N}_d^m(\sigma; L, n). \tag{52}$$

  In sequel, require activation function σ to belong to set of piecewise continuous & locally bounded functions

  $$\mathcal{M} \coloneqq \{\sigma \in L^\infty_{\mathrm{loc}}(\mathbb{R}) | \text{there exists intervals } I_1, \ldots, I_M \text{ partitioning } \mathbb{R}, \text{ s.t. } \sigma \in C^0(I_i), \ \forall i = 1, \ldots, M\}. \tag{53}$$

  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 5. *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 3. *Let $d \in \mathbb{N}$ & $\sigma \in \mathcal{M}$. Then $\mathcal{N}_d^1(\sigma, 1)$ is a universal approximator of $C^0(\mathbb{R}^d)$ iff σ is not a polynomial.*

  Remark 4. *Exercise 3.26 & Corollary 3.18: neural networks can also arbitrarily well approximate non-continuous functions w.r.t. suitable norms.*

  Universal approximation theorem by Leshno, Lin, Pinkus & Schocken [132] – of which Thm. 3.8 is a special case – is even formulated for a much larger set $\mathcal{M}$, which allows for activation functions that have discontinuities at a (possibly non-finite) set of Lebesgue measure 0. Instead of proving theorem in this generality, resort to simpler case stated above. This allows to avoid some technicalities, but main ideas remain same. Proof strategy: verify 3 claims:

  (i) if $C^0(\mathbb{R}) \subseteq \overline{\mathcal{N}_1^1(\sigma; 1)}^{\mathrm{cc}}$ then $C^0(\mathbb{R}^d) \subseteq \overline{\mathcal{N}_d^1(\sigma; 1)}^{\mathrm{cc}}$,

  (ii) if $\sigma \in C^\infty(\mathbb{R})$ is not a polynomial then $C^0(\mathbb{R}) \subseteq \overline{\mathcal{N}_1^1(\sigma; 1)}^{\mathrm{cc}}$,

  (iii) if $\sigma \in \mathcal{M}$ is not a polynomial then there exists $\widetilde{\sigma} \in C^\infty(\mathbb{R}) \cap \overline{\mathcal{N}_1^1(\sigma; 1)}^{\mathrm{cc}}$ which is not a polynomial.

- ○ 3.2. Superexpressive activations & Kolmogorov's supersolution theorem.

- • 4. Splines.
  - ○ 4.1. B-splines & smooth functions.
  - ○ 4.2. Reaproximation of B-splines with sigmoidal activations.

- • 5. ReLU neural networks.
  - ○ 5.1. Basic ReLU calculus.
  - ○ 5.2. Continuous piecewise linear functions.
  - ○ 5.3. Simplicial pieces.
  - ○ 5.4. Convergence rates for Hölder continuous functions.

- • 6. Affine pieces for ReLU neural networks.
  - ○ 6.1. Upper bounds.
  - ○ 6.2. Tightness of upper bounds.
  - ○ 6.3. Depth separation.
  - ○ 6.4. Number of pieces in practice.

- • 7. Deep ReLU neural networks.
  - ○ 7.1. Square function.
  - ○ 7.2. Multiplication.
  - ○ 7.3. $C^{k,s}$ functions.

- 8. High-dimensional approximation.
    - 8.1. Barron class.
    - 8.2. Functions with compositionality structure.
    - 8.3. Functions on manifolds.
- 9. Interpolation.
    - 9.1. Universal interpolation.
    - 9.2. Optimal interpolation & reconstruction.
- 10. Training of neural networks.
    - 10.1. Gradient descent.
    - 10.2. Stochastic gradient descent (SGD).
    - 10.3. Backpropagation.
    - 10.4. Acceleration.
    - 10.5. Other methods.
- 11. Wide neural networks & neural tangent kernel.
    - 11.1. Linear least-squares.
    - 11.2. Kernel least-squares.
    - 11.3. Tangent kernel.
    - 11.4. Convergence to global minimizers.
    - 11.5. Training dynamics for LeCun initialization.
    - 11.6. Normalized initialization.
- 12. Loss landscape analysis.
    - 12.1. Visualization of loss landscapes.
    - 12.2. Spurious valleys.
    - 12.3. Saddle points.
- 13. Shape of neural network spaces.
    - 13.1. Lipschitz parameterizations.
    - 13.2. Convexity of neural network spaces.
    - 13.3. Closedness & best-approximation property.
- 14. Generalization properties of deep neural networks.
    - 14.1. Learning setup.
    - 14.2. Empirical risk minimization.
    - 14.3. Generalization bounds.
    - 14.4. Generalization bounds from covering numbers.
    - 14.5. Covering numbers of deep neural networks.
    - 14.6. Approximate-complexity trade-off.
    - 14.7. PAC learning from VC dimension.
    - 14.8. Lower bounds on achievable approximation rates.
- 15. Generalization in overparameterized regime.
    - 15.1. Double descent phenomenon.
    - 15.2. Size of weights.
    - 15.3. Theoretical justification.
    - 15.4. Double descent for neural network learning.
- 16. Robustness & adversarial examples.
    - 16.1. Adversarial examples.

- 16.2. Bayes classifier.
- 16.3. Affine classifiers.
- 16.4. ReLU neural networks.
- 16.5. Robustness.

- A. Probability theory.
  - A.1. Sigma-algebras, topologies, & measures.
  - A.2. Random variables.
  - A.3. Conditionals, marginals, & independence.
  - A.4. Concentration inequalities.
- B. Functional analysis.
  - B.1. Vector spaces.
  - B.2. Fourier transform.

1.7 [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 & 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.[2] When have more data, can train more powerful models & rely less heavily on preconceived assumptions. Regime (chế độ) change from (comparatively) small to big data is a major contributor to success of modern DL. To drive point home, many of most exciting models in DL do not work without large datasets. Some others might work in small data regime, but are no better than traditional approaches.

    Finally, not enough to have lots of data & to process it cleverly. Need *right* data. If data is full of mistakes, or if chosen features are not predictive of target quantity of interest, learning is going to fail. Situation is captured well by cliché: *garbage in, garbage out*. Moreover, poor predictive performance is not only potential consequence. In sensitive applications of ML, like predictive policing, resume screening, & risk models used for lending, must be especially alert to consequences of garbage data. 1 commonly occurring failure mode concerns datasets where some groups of people are unrepresented in training data. Imagine applying a skin cancer recognition system that had never been black skin before. Failure can also occur when data does not only under-represent some groups but reflects societal prejudices. E.g., if past hiring decisions are used to train a predictive model that will be used to screen resumes then ML models could inadvertently capture & automate historical injustices. Note: this can all happen without data scientist actively conspiring, or even being aware.

  - 1.2.2. Models. Most ML involves transforming data in some sense. Might want to build a system that ingests photos & predicts smiley-ness. Alternatively, might want to ingest a set of sensor readings & predict how normal vs. anomalous (bất thường) readings are. By *model*, denote computational machinery for ingesting data of 1 type, & spitting out predictions of a possibly different type. In particular, interested in *statistical models* that can be estimated from data. While simple models are perfectly capable of addressing appropriately simple problems, problems that we focus on in this book stretch limits of classical methods. DL is differentiated from classical approaches principally by set of powerful models that it focuses on. These models consist of many successive transformations of data chained together top to bottom, thus name *deep learning*. On our way to discussing deep models, also discuss some more traditional methods.

[2] NQBH: Really? Or more illusion, delusion?

- 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-theshelf 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 opponents' cards). Libratus exceeded human performance in Poker using efficiently structured strategies (Brown & Sandholm, 2017).
- Another indication of progress in AI: advent of self-driving vehicles (sự ra đời của xe tự lái). While full autonomy is not yet within reach, excellent progress has been made in this direction, with companies e.g. Tesla, NVIDIA, & Waymo shipping products that enable partial autonomy. What makes full autonomy so challenging: proper driving requires ability to perceive, to reason & to incorporate rules into a system. At present, DL is used primarily in visual aspect of these problems. The rest is heavily tuned by engineers.

This barely scratches surface of significant applications of ML. E.g., robotics, logistics, computational biology, particle physics, & astronomy owe some of their most impressive recent advances at least in parts to ML, which is thus becoming a ubiquitous tool for engineers & scientists.

Frequently, questions about a coming AI apocalypse & plausibility of a *singularity* have been raised in non-technical articles. Thông thường, các câu hỏi về ngày tận thế sắp tới của AI & khả năng xảy ra *điểm kỳ dị* đã được nêu ra trong các bài viết không mang tính kỹ thuật. Fear: somehow ML systems will become sentient & make decisions, independently of their programmers, that directly impact lives of humans. To some extent, AI already affects livelihood of humans in direct ways: creditworthiness is assessed automatically, autopilots mostly navigate vehicles, decisions about whether to grant bail use statistical data as input. More frivolously, can ask Alexa to switch on coffee machine.

Fortunately, we are far from a sentient AI system that could deliberately manipulate its human creators. 1st, AI systems are engineered, trained, & deployed in a specific, goal-oriented manner. While their behavior might give illusion of general intelligence, it is a combination of rules, heuristics & statistical models that underlie design. 2nd, at present, there are simply no tools for *artificial general intelligence* that are able to improve themselves, reason about themselves, & that are able to modify, extend, & improve their own architecture while trying to solve general tasks.

A much more pressing concern is how AI is being used in our daily lives. Likely: many routine tasks, currently fulfilled by humans, can & will be automated. Farm robots will likely reduce costs for organic farmers but they will also automate harvesting operations. This phase of industrial revolution may have profound consequences for large swaths of society, since menial jobs provide much employment in many countries. Furthermore, statistical models, when applied without care, can lead to racial, gender, or age bias & raise reasonable concerns about procedural fairness if automated to drive consequential decisions. Important to ensure: these algorithms are used with care. With what we know today, this strikes us as a much more pressing concern than potential of malevolent superintelligence for destroying humanity.

- 1.7. Essence of Deep Learning. Thus far, have talked in broad terms about ML. DL is subset of ML concerned with models based on many-layered neural networks. *Deep* in precisely sense that its models learn many *layers* of transformations. While this might sound narrow, DL has given rise to a dizzying (chóng mặt) array of models, techniques, problem formulations, & applications. Many intuitions have been developed to explain benefits of depth. Arguably, all ML has many layers of computation, 1st computing of feature processing steps. What differentiates DL: operations learned at each of many layers of representations are learned jointly from data.

  Problems that have discussed so far, e.g. learning from raw audio signal, raw pixel values of images, or mapping between sentences of arbitrary lengths & their counterparts in foreign languages, are those where DL excels & traditional methods falter (những nơi mà DL vượt trội & các phương pháp truyền thống không hiệu quả). Turn out: these many-layered models are capable of addressing low-level perceptual data in a way that previous tools could not – Thực tế: các mô hình nhiều lớp này có khả năng giải quyết dữ liệu nhận thức cấp thấp theo cách mà các công cụ trước đây không thể làm được. Arguably most significant commonality in DL methods is *end-to-end training.* I.e., rather than assembling a system based on components that are individually tuned, one builds system & then tunes their performance jointly. E.g., in computer vision scientists used to separate process of *feature engineering* from process of building ML models. Canny edge detector (Canny, 1987) & Lowe's SIFT feature extractor (Lowe, 2004) reigned supreme for over a decade as algorithms for mapping images into feature vectors. In bygone days, crucial part of applying Ml to these problems consisted of coming up with manually-engineered ways of transforming data into some form amenable to shallow models. Unfortunately, there is only so much that humans can accomplish by ingenuity in comparison with a consistent evaluation over millions of choices carried out automatically by an algorithm. When DL took over, these feature extractors were replaced by automatically tuned filters that yielded superior accuracy.

  Thus, 1 key advantage of DL: DL replaces not only shallow models at end of traditional learning pipelines, but also labor-intensive process of feature engineering. Moreover, by replacing much of domain-specific preprocessing, DL has eliminated many of boundaries that previously separated computer vision, speech recognition, natural language processing, medical informatics, & other application areas, thereby offering a unified set of tools for tackling diverse problems.

  Beyond end-to-end training, are experiencing a transition from parametric statistical descriptions to fully nonparametric models. When data is scarse, one needs to rely on simplifying assumptions about reality in order to obtain useful models. When data is abundant, these can be replaced by nonparametric models that better fit data. To some extent, this mirrors progress that physics experienced in middle of previous century with availability of computers. Rather than solving by hand parametric approximations of how electrons behave, one can now resort to numerical simulations of associated PDEs. This has led to much more accurate models, albeit often at expense of interpretation.

  Another difference from previous work is acceptance of suboptimal solutions, dealing with nonconvex nonlinear optimization problems, & willingness to try things before proving them. This new-found empiricism in dealing with statistical problems, combined with a rapid influx of talent has led to rapid progress in development of practical algorithms, albeit in many cases at expense of modifying & re-inventing tools that existed for decades.

  In the end, DL community prides itself on sharing tools across academic & corporate boundaries, releasing many excellent libraries, statistical models, & trained networks as open source. In this spirit: notebooks forming this book are freely available for distribution & use. Have worked hard to lower barriers of access for anyone wishing to learn about DL & hope: readers will benefit from this.

- 1.8. Summary. ML studies how computer systems can leverage experience (có thể tận dụng kinh nghiệm) (often data) to improve performance at specific tasks. ML combines ideas from statistics, data mining, & optimization. Often, ML is used as a means of implementing AI solutions. As a class of ML, representational learning focuses on how to automatically find appropriate way to represent data. Considered as multi-level representation learning through learning many layers of transformations, DL replaces not only shallow models at end of traditional ML pipelines, but also labor-intensive process of feature engineering. Much of recent progress in DL has been triggered by an abundance of data rising from cheap sensors & Internet-scale applications, & by significant progress in computation, mostly through GPUs. Furthermore, availability of efficient DL frameworks has made design & implementation of whole system optimization significantly easier & this is a key component in obtaining high performance.

- Exercises.

  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.

1.8 [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.

1.9 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 $5.15 \cdot 10^{-5}$. 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: $u'(x) \approx 0.0125$ 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

    $$\begin{cases} u''(x) + u(x) = e^{-x}, \ x \in (0,1) \\ u(0) = 1, \ u(1) = \frac{1}{2}\cos 1 + \frac{1}{2}\sin 1 + \frac{1}{2e}. \end{cases}$$

    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

    $$L = \frac{1}{99}\sum_{i=1}^{99}\left(\frac{d^2\hat{u}}{dx_i^2} + \hat{u}(x_i) - e^{-x_i}\right)^2 + \frac{1}{2}\left(|\hat{u}(0) - 1|^2 + \left|\hat{u}(1) - \frac{1}{2}\cos 1 + \frac{1}{2}\sin 1 + \frac{1}{2e}\right|^2\right).$$

    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: $u''(x) \approx 0$ at all interior points (mean of $u'(x_i)$, $i = 1, \ldots, n$ is $-7.86 \cdot 10^{-5}$ & variance is $9.53 \cdot 10^{-9}$). 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 1 (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 $u^*(x) = \dfrac{\sin x + \cos x + e^{-x}}{2}$. 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 $\boxed{\text{Adam optimizer}}$ 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.

1.10 Geeks for Geeks/What Is A Neural Network?

Resources – Tài nguyên. Geeks for Geeks/what is a neural network?

Neural networks are ML models that mimic complex functions of human brain. These models consist of interconnected nodes or neurons that process data, learn patterns & enable tasks e.g. pattern recognition & decision-making.

– Mạng nơ-ron là mô hình học máy (ML) mô phỏng các chức năng phức tạp của não người. Các mô hình này bao gồm các nút hoặc nơ-ron được kết nối với nhau để xử lý dữ liệu, học các mẫu & thực hiện các tác vụ như nhận dạng mẫu & ra quyết định.

Explore fundamentals of neural networks, their architecture, how they work & their applications in various fields. Understanding neural networks is essential for anyone interested in advancements of AI.

– Khám phá những kiến thức cơ bản về mạng nơ-ron, kiến trúc, cách thức hoạt động và ứng dụng của chúng trong nhiều lĩnh vực. Hiểu biết về mạng nơ-ron là điều cần thiết cho bất kỳ ai quan tâm đến những tiến bộ của AI.

1.10.1 Understanding neural networks in DL

Neural networks are capable of learning & identifying patterns directly from data without pre-defined rules. These networks are built from several key components:

1. **Neurons.** Basic units that receive inputs, each neuron is governed by a threshold & an activation function.
2. **Connections.** Links between neurons that carry information, regulated by weights & biases.
3. **Weights & Biases.** These parameters determine strength & influence of connections.
4. **Propagation Functions.** Mechanisms that help process & transfer data across layers of neurons.
5. **Learning Rule.** Method that adjusts weights & biases over time to improve accuracy.

– Mạng nơ-ron có khả năng học & nhận dạng các mẫu trực tiếp từ dữ liệu mà không cần các quy tắc được xác định trước. Các mạng này được xây dựng từ một số thành phần chính:

1. **Nơ-ron.** Các đơn vị cơ bản nhận đầu vào, mỗi nơ-ron được điều khiển bởi một ngưỡng & một hàm kích hoạt.
2. **Kết nối.** Các liên kết giữa các nơ-ron mang thông tin, được điều chỉnh bởi trọng số & độ lệch.
3. **Trọng số & Độ lệch.** Các tham số này xác định cường độ & ảnh hưởng của các kết nối.
4. **Hàm truyền.** Các cơ chế giúp xử lý & truyền dữ liệu qua các lớp nơ-ron.
5. **Quy tắc học.** Phương pháp điều chỉnh trọng số & độ lệch theo thời gian để cải thiện độ chính xác.

Learning in neural networks follows a structured, 3-stage process:

1. **Input Computation.** Data is fed into network.
2. **Output Generation.** based on current parameters, network generates an output.
3. **Iterative Refinement.** Network refines its output by adjusting weights & biases, gradually improving its performance on diverse tasks.

– Việc học trong mạng nơ-ron tuân theo một quy trình có cấu trúc gồm 3 giai đoạn:

1. **Tính toán Đầu vào.** Dữ liệu được đưa vào mạng.
2. **Tạo Đầu ra.** Dựa trên các tham số hiện tại, mạng tạo ra đầu ra.
3. **Tinh chỉnh Lặp lại.** Mạng tinh chỉnh đầu ra bằng cách điều chỉnh trọng số & độ lệch, dần dần cải thiện hiệu suất của nó trên các tác vụ đa dạng.

In an adaptive learning environment:

- Neural network is exposed to a simulated scenario or dataset.
- Parameters e.g. weights & biases are updated in response to new data or conditions.
- With each adjustment, network's response evolves allowing it to adapt effectively to different tasks or environments.

Fig: Image illustrates analogy between a biological neuron & an artificial neuron, showing how inputs are received & processed to produce outputs in both systems.

– Trong môi trường học tập thích ứng:

- Mạng nơ-ron được tiếp xúc với một kịch bản hoặc tập dữ liệu mô phỏng.
- Các tham số, ví dụ như trọng số & độ lệch, được cập nhật để đáp ứng với dữ liệu hoặc điều kiện mới.
- Với mỗi lần điều chỉnh, phản ứng của mạng sẽ phát triển, cho phép nó thích ứng hiệu quả với các tác vụ hoặc môi trường khác nhau.

Hình: Hình ảnh minh họa sự tương tự giữa nơ-ron sinh học & nơ-ron nhân tạo, cho thấy cách tiếp nhận & xử lý đầu vào để tạo ra đầu ra trong cả hai hệ thống.

Importance of neural networks. Neural networks are important in identifying complex patterns, solving intricate challenges & adapting to dynamic environments. Their ability to learn from vast amounts of data is transformative, impacting technologies like NLP, self-driving vehicles & automated decision-making.

– **Tầm quan trọng của mạng nơ-ron.** Mạng nơ-ron đóng vai trò quan trọng trong việc xác định các mô hình phức tạp, giải quyết những thách thức phức tạp & thích ứng với môi trường năng động. Khả năng học hỏi từ lượng dữ liệu khổng lồ của chúng mang tính đột phá, tác động đến các công nghệ như NLP, xe tự lái & ra quyết định tự động.

Neural networks streamline processes, increase efficiency & support decision-making across various industries. As a backbone of AI, they continue to drive innovation, shaping future of technology.

– Mạng nơ-ron nhân tạo hợp lý hóa quy trình, tăng hiệu quả & hỗ trợ ra quyết định trong nhiều ngành công nghiệp khác nhau. Là xương sống của AI, chúng tiếp tục thúc đẩy đổi mới, định hình tương lai của công nghệ.

1.10.2 Layers in neural network architecture

1. **Input Layer.** This is where network receives its input data. Each input neuron in layer corresponds to a feature in input data.

   – **Lớp Đầu vào.** Đây là nơi mạng nhận dữ liệu đầu vào. Mỗi nơ-ron đầu vào trong lớp tương ứng với một đặc trưng trong dữ liệu đầu vào.

2. **Hidden Layers.** These layers perform most of computational heavy lifting. A neural network can have 1 or multiple hidden layers. Each layer consists of units (neurons) that transform inputs into something that output layer can use.

   – **Lớp Ẩn.** Các lớp này thực hiện hầu hết các tác vụ tính toán nặng. Một mạng nơ-ron có thể có 1 hoặc nhiều lớp ẩn. Mỗi lớp bao gồm các đơn vị (nơ-ron) biến đổi dữ liệu đầu vào thành thứ mà lớp đầu ra có thể sử dụng.

3. **Output Layer.** Final layer produces output of model. Format of these outputs varies depending on specific tasks like classification, regression.

   – **Lớp Đầu ra.** Lớp cuối cùng tạo ra đầu ra của mô hình. Định dạng của các đầu ra này thay đổi tùy thuộc vào các tác vụ cụ thể như phân loại, hồi quy.

1.10.3 Working of neural networks

1. **Forward Propagation.** When data is input into network, it passes through network in forward direction, from input layer through hidden layers to output layer. This process is known as forward propagation. Here is what happens during this phase:

   (a) **Linear Transformation.** Each neuron in a layer receives inputs which are multiplied by weights associated with connections. These products are summed together & a bias is added to sum. This can be represented mathematically as $z = \mathbf{w} \cdot \mathbf{x} + b = \sum_{i=1}^n w_i x_i + b$ where w: weights, x inputs, b: bias.

   (b) **Activation.** Result of linear transformation, denoted as z, is then passed through an activation function. Activation function is crucial because it introduces nonlinearity into system, enabling network to learn more complex patterns. Popular activation functions include ReLU, sigmoid & tanh.

   – **Truyền thẳng.** Khi dữ liệu được đưa vào mạng, nó truyền qua mạng theo hướng thẳng, từ lớp đầu vào qua các lớp ẩn đến lớp đầu ra. Quá trình này được gọi là truyền thẳng. Dưới đây là những gì xảy ra trong giai đoạn này:

   (a) **Biến đổi tuyến tính.** Mỗi nơ-ron trong một lớp nhận các đầu vào được nhân với trọng số liên quan đến các kết nối. Các tích này được cộng lại với nhau & một độ lệch được thêm vào tổng. Điều này có thể được biểu diễn bằng toán học như sau $z = \mathbf{w} \cdot \mathbf{x} + b = \sum_{i=1}^n w_i x_i + b$ trong đó w: trọng số, x đầu vào, b: độ lệch.

   (b) **Kích hoạt.** Kết quả của phép biến đổi tuyến tính, ký hiệu là z, sau đó được truyền qua một hàm kích hoạt. Hàm kích hoạt rất quan trọng vì nó đưa tính phi tuyến tính vào hệ thống, cho phép mạng học các mẫu phức tạp hơn. Các hàm kích hoạt phổ biến bao gồm ReLU, sigmoid & tanh.

2. **Backpropagation.** After forward propagation, network evaluates its performance using a loss function which measures difference between actual output & predicted output. Goal of training: minimize this loss. This is where backpropagation comes into play:

   (a) **Loss Calculation.** Network calculates loss which provides a measure of error in predictions. Loss function could vary; common choices are mean squared error for regression tasks or cross-entropy loss for classification.

   (b) **Gradient Calculation.** Network computes gradients of loss function w.r.t. each weight & bias in network. This involves applying chain rule of calculus to find out how much each part of output error can be attributed to each weight & bias.

   (c) **Weight Update.** Once gradients are calculated, weights & biases are updated using an optimization algorithm like stochastic gradient descent (SGD). Weights are adjusted in opposite direction of gradient to minimize loss. Size of step taken in each update is determined by learning rate.

   – **Truyền ngược.** Sau khi truyền ngược, mạng đánh giá hiệu suất của nó bằng hàm mất mát, hàm này đo lường sự khác biệt giữa đầu ra thực tế và đầu ra dự đoán. Mục tiêu của quá trình huấn luyện: giảm thiểu tổn thất này. Đây là lúc truyền ngược phát huy tác dụng:

   (a) **Tính toán Mất mát.** Mạng tính toán tổn thất, hàm này cung cấp thước đo sai số trong dự đoán. Hàm mất mát có thể thay đổi; các lựa chọn phổ biến là sai số bình phương trung bình cho các tác vụ hồi quy hoặc mất mát entropy chéo cho phân loại.

   (b) **Tính toán Độ dốc.** Mạng tính toán độ dốc của hàm mất mát đối với mỗi trọng số và độ lệch trong mạng. Điều này liên quan đến việc áp dụng quy tắc chuỗi của phép tính để tìm ra mức độ mà mỗi phần của lỗi đầu ra có thể được quy cho mỗi trọng số và độ lệch.

   (c) **Cập nhật Trọng số.** Sau khi độ dốc được tính toán, các trọng số và độ lệch được cập nhật bằng một thuật toán tối ưu hóa như giảm độ dốc ngẫu nhiên (SGD). Các trọng số được điều chỉnh theo hướng ngược lại của độ dốc để giảm thiểu tổn thất. Kích thước bước thực hiện trong mỗi lần cập nhật được xác định bởi tốc độ học.

3. **Iteration.** This process of forward propagation, loss calculation, backpropagation & weight update is repeated for many iterations over dataset. Over time, this iterative process reduces loss & network's predictions become more accurate.

   – **Lặp lại.** Quá trình lan truyền thuận, tính toán tổn thất, lan truyền ngược & cập nhật trọng số này được lặp lại qua nhiều lần lặp trên tập dữ liệu. Theo thời gian, quá trình lặp lại này làm giảm tổn thất & dự đoán của mạng trở nên chính xác hơn.

Through these steps, neural networks can adapt their parameters to better approximate relationships in data, thereby improving their performance on tasks e.g. classification, regression or any other predictive modeling.

– Thông qua các bước này, mạng nơ-ron có thể điều chỉnh các tham số của chúng để ước tính tốt hơn các mối quan hệ trong dữ liệu, do đó cải thiện hiệu suất của chúng trong các tác vụ như phân loại, hồi quy hoặc bất kỳ mô hình dự đoán nào khác.

1.10.4 Example of Email Classification

Consider a record of an email dataset: Email ID: 1, Email Content: "Get free gift cards now!", Sender `spam@example.com`, Subject Line: "Exclusive Offer", Label: 1. To classify this email, create a feature vector based on analysis of keywords e.g. "free" "win" & "offer". Feature vector of recrod can be presented as: "free": Present (1), "win": Absent (0), "offer": Present (1).

How neurons process data in a neural network. In a neural network, input data is passed through multiple layers, including 1 or more hidden layers. Each neuron in these hidden layers performs several operations, transforming input into a usable output.

1. **Input Layer.** Input layer contains 3 nodes that indicates presence of each keyword.
2. **Hidden Layer.** Input vector is passed through hidden layer. Each neuron in hidden layer performs 2 primary operations: a weighted sum followed by an activation function. Weights: Neuron H1: [0.5, -0.2,0.3] Neuron H2: [0.4,0.1,-0.5]. Input Vector: [1,0,1]. Weighted Sum Calculation: For H1: $1\cdot 0.5+0\cdot -0.2+1\cdot 0.3=0.8, 1\cdot 0.4+0\cdot 0.1+1\cdot -0.5=-0.1$. Activation Function: Here we will use ReLu activation function: H1 Output: ReLU(0.8)= 0.8. H2 Output: ReLu(-0.1) = 0.
3. **Output Layer.** Activated values from hidden neurons are sent to output neuron where they are again processed using a weighted sum & an activation function. Output Weights: [0.7, 0.2]. Input from Hidden Layer: [0.8, 0]. Weighted Sum: $0.8\cdot 0.7+0\cdot 0.2=0.56$. Activation (Sigmoid): $\sigma(0.56)=\frac{1}{1+e^{-0.56}}\approx 0.636$.
4. **Final Classification.** Output value of approximately 0.636 indicates probability of email being spam. Since this value is >0.5, neural network classifies email as spam (1).

– Trong mạng nơ-ron, dữ liệu đầu vào được truyền qua nhiều lớp, bao gồm 1 hoặc nhiều lớp ẩn. Mỗi nơ-ron trong các lớp ẩn này thực hiện một số thao tác, biến đổi đầu vào thành đầu ra có thể sử dụng được.

1. **Lớp Đầu vào.** Lớp đầu vào chứa 3 nút biểu thị sự hiện diện của mỗi từ khóa.
2. **Lớp Ẩn.** Vectơ đầu vào được truyền qua lớp ẩn. Mỗi nơ-ron trong lớp ẩn thực hiện 2 thao tác chính: tổng có trọng số theo sau là hàm kích hoạt. Trọng số: Nơ-ron H1: [0,5, -0,2, 0,3] Nơ-ron H2: [0,4, 0,1, -0,5]. Vectơ đầu vào: [1, 0, 1]. Tính toán Tổng có trọng số: Đối với H1: $1\cdot 0.5+0\cdot -0.2+1\cdot 0.3=0.8, 1\cdot 0.4+0\cdot 0.1+1\cdot -0.5=-0.1$. Hàm kích hoạt: Ở đây, chúng ta sẽ sử dụng hàm kích hoạt ReLu: Đầu ra H1: ReLU(0.8)= 0.8. Đầu ra H2: ReLu(-0.1) = 0.
3. **Lớp đầu ra.** Các giá trị được kích hoạt từ các nơ-ron ẩn được gửi đến nơ-ron đầu ra, tại đó chúng lại được xử lý bằng tổng có trọng số & một hàm kích hoạt. Trọng số đầu ra: [0.7, 0.2]. Đầu vào từ Lớp ẩn: [0.8, 0]. Tổng có trọng số: $0.8\cdot 0.7+0\cdot 0.2=0.56$. Kích hoạt (Sigmoid): $\sigma(0,56)=\frac{1}{1+e^{-0,56}}\approx 0,636$.
4. **Phân loại cuối cùng.** Giá trị đầu ra xấp xỉ 0,636 cho biết xác suất email là thư rác. Vì giá trị này $>0,5$, mạng nơ-ron sẽ phân loại email là thư rác (1).

1.10.5 Learning of a Neural Network

1. **Learning with Supervised Learning.** In supervised learning, a neural network learns from labeled input-output pairs provided by a teacher. Network generates outputs based on inputs & by comparing these outputs to known desired outputs, an error signal is created. Network iteratively adjusts its parameters to minimize errors until it reaches an acceptable performance level.

   – **Học với Học có Giám sát.** Trong học có giám sát, mạng nơ-ron học từ các cặp đầu vào-đầu ra được gán nhãn do giáo viên cung cấp. Mạng tạo ra các đầu ra dựa trên các đầu vào & bằng cách so sánh các đầu ra này với các đầu ra mong muốn đã biết, một tín hiệu lỗi được tạo ra. Mạng điều chỉnh các tham số của nó một cách lặp đi lặp lại để giảm thiểu lỗi cho đến khi đạt được mức hiệu suất chấp nhận được.
2. **Learning with Unsupervised Learning.** Unsupervised learning involves data without labeled output variables. Primary goal: understand underlying structure of input data. Unlike supervised learning, there is no instructor to guide process. Instead, focus in on modeling data patterns & relationships, with techniques like clustering & association commonly used.

   – **Học tập với Học không giám sát.** Học không giám sát liên quan đến dữ liệu không có biến đầu ra được gắn nhãn. Mục tiêu chính: hiểu cấu trúc cơ bản của dữ liệu đầu vào. Không giống như học có giám sát, không có người hướng dẫn để hướng dẫn quy trình. Thay vào đó, hãy tập trung vào việc mô hình hóa các mẫu dữ liệu & các mối quan hệ, với các kỹ thuật như phân cụm & liên kết thường được sử dụng.
3. **Learning with Reinforcement Learning.** Reinforcement learning enables a neural network to learn through interaction with its environment. Network receives feedback in form of rewards or penalties, guiding it to find an optimal policy or strategy that maximizes cumulative rewards over time. This approach is widely used in applications like gaming & decision-making.

   – **Học với Học Tăng cường.** Học tăng cường cho phép mạng nơ-ron học thông qua tương tác với môi trường. Mạng nhận được phản hồi dưới dạng phần thưởng hoặc hình phạt, hướng dẫn nó tìm ra chính sách hoặc chiến lược tối ưu để tối đa hóa phần thưởng tích lũy theo thời gian. Phương pháp này được sử dụng rộng rãi trong các ứng dụng như trò chơi & ra quyết định.

1.10.6 Types of neural networks

There are 7 types of neural networks that can be used so far?

1. Feedforward Networks: A simple ANN architecture in which data moves from input to output in a single direction.
2. Singlelayer Perception: It has 1 layer & it applies weights, sums inputs & uses activation to produce output.
3. Multiplayer Perception (MLP): A type of feedforward neural network with 3 or more layers, including an input layer, 1 or more hidden layers & an output layer. It uses nonlinear activation functions.
4. Convolutional Neural Network (CNN): It is designed for image processing. It uses convolutional layers to automatically learn features from input images, enabling effective image recognition & classification.
5. Recurrent Neural Network (RNN): Handles sequential data using feedback loops to retain context over time.
6. Long Short-Term Memory (LSTM): A type of RNN with memory cells & gates to handle long-term dependencies & avoid vanishing gradients.

– Có 7 loại mạng nơ-ron có thể được sử dụng cho đến nay?

1. Mạng truyền thẳng: Một kiến trúc ANN đơn giản, trong đó dữ liệu di chuyển từ đầu vào đến đầu ra theo một hướng duy nhất.
2. Nhận thức một lớp: Có 1 lớp & áp dụng trọng số, tổng các đầu vào & sử dụng kích hoạt để tạo đầu ra.
3. Nhận thức nhiều người chơi (MLP): Một loại mạng nơ-ron truyền thẳng có 3 lớp trở lên, bao gồm một lớp đầu vào, 1 hoặc nhiều lớp ẩn & một lớp đầu ra. Nó sử dụng các hàm kích hoạt phi tuyến tính.
4. Mạng nơ-ron tích chập (CNN): Được thiết kế để xử lý hình ảnh. Nó sử dụng các lớp tích chập để tự động học các đặc điểm từ hình ảnh đầu vào, cho phép nhận dạng hình ảnh & phân loại hiệu quả.
5. Mạng nơ-ron hồi quy (RNN): Xử lý dữ liệu tuần tự bằng các vòng phản hồi để duy trì ngữ cảnh theo thời gian.
6. Bộ nhớ dài hạn ngắn hạn (LSTM): Một loại RNN với các ô nhớ & cổng để xử lý các phụ thuộc dài hạn & tránh hiện tượng mất dần gradient.

1.10.7 Implementation of Neural Network using TensorFlow

Implement simple feedforward neural network that trains on a sample dataset & makes predictions using following steps:

1. **Import Necessary Libraries.** Import necessary libraries, primarily TensorFlow & Keras, along with other required packages e.g. NumPy & Pandas for data handling.
2. **Create & Load Dataset.** Create or load a dataset. Convert data into a format suitable for training (usually NumPy arrays). Define features X & labels y.
3. **Create a Neural Network.** Instantiate a Sequential model & add layers. Input layer & hidden layers are typically created using Dense layers, specifying number of neurons & activation functions.
4. **Compiling Model.** Compile model by specifying loss function, optimizer & metrics to evaluate during training. Here use binary crossentropy & Adam optimizer.
5. **Train Model.** Fit model on training data, specifying number of epochs & batch size. This step trains neural network to learn from input data.
6. **Make Predictions.** Use trained model to make predictions on new data. Process output to interpret predictions like converting probabilities to binary outcomes.

– Triển khai mạng nơ-ron truyền thẳng đơn giản, huấn luyện trên một tập dữ liệu mẫu & đưa ra dự đoán bằng các bước sau:

1. **Nhập Thư viện Cần thiết.** Nhập các thư viện cần thiết, chủ yếu là TensorFlow & Keras, cùng với các gói cần thiết khác, ví dụ: NumPy & Pandas để xử lý dữ liệu.
2. **Tạo & Tải Tập dữ liệu.** Tạo hoặc tải một tập dữ liệu. Chuyển đổi dữ liệu sang định dạng phù hợp để huấn luyện (thường là mảng NumPy). Xác định các đặc trưng X & nhãn y.
3. **Tạo Mạng Nơ-ron.** Khởi tạo một mô hình Tuần tự & thêm các lớp. Lớp đầu vào & các lớp ẩn thường được tạo bằng cách sử dụng các lớp Dense, chỉ định số lượng nơ-ron & các hàm kích hoạt.
4. **Biên dịch Mô hình.** Biên dịch mô hình bằng cách chỉ định hàm mất mát, trình tối ưu hóa & các số liệu để đánh giá trong quá trình huấn luyện. Ở đây sử dụng crossentropy nhị phân & trình tối ưu hóa Adam.
5. **Mô hình Huấn luyện.** Phù hợp mô hình với dữ liệu huấn luyện, chỉ định số kỷ nguyên & kích thước lô. Bước này huấn luyện mạng nơ-ron để học từ dữ liệu đầu vào.
6. **Đưa ra Dự đoán.** Sử dụng mô hình đã huấn luyện để đưa ra dự đoán trên dữ liệu mới. Xử lý đầu ra để diễn giải các dự đoán, chẳng hạn như chuyển đổi xác suất thành kết quả nhị phân.

1.10.8 Advantages of neural networks

Neural networks are widely used in many different applications because of their many benefits:

- **Adaptability.** Neural networks are useful for activities where link between inputs & outputs is complex or not well defined because they can adapt to new situations & learn from data.
- **Pattern Recognition.** Their proficiency in pattern recognition renders them efficacious in tasks like as audio & image identification, NLP & other intricate data patterns.
- **Parallel Processing.** Because neural networks are capable of parallel processing by nature, they can process numerous jobs at once which speeds up & improves efficiency of computations.
- **Nonlinearity.** Neural networks are able to model & comprehend complicated relationships in data by virtue of nonlinear activation functions found in neurons which overcome drawbacks of linear models.

– Mạng nơ-ron được sử dụng rộng rãi trong nhiều ứng dụng khác nhau nhờ những lợi ích của chúng:

- **Khả năng thích ứng.** Mạng nơ-ron hữu ích cho các hoạt động mà mối liên kết giữa đầu vào & đầu ra phức tạp hoặc chưa được xác định rõ ràng vì chúng có thể thích ứng với các tình huống mới & học hỏi từ dữ liệu.
- **Nhận dạng Mẫu.** Khả năng nhận dạng mẫu của chúng giúp chúng hiệu quả trong các tác vụ như nhận dạng âm thanh & hình ảnh, NLP & các mẫu dữ liệu phức tạp khác.
- **Xử lý Song song.** Do bản chất mạng nơ-ron có khả năng xử lý song song, chúng có thể xử lý nhiều tác vụ cùng lúc, giúp tăng tốc & cải thiện hiệu quả tính toán.
- **Phi tuyến tính.** Mạng nơ-ron có khả năng mô hình hóa & hiểu các mối quan hệ phức tạp trong dữ liệu nhờ các hàm kích hoạt phi tuyến tính được tìm thấy trong nơ-ron, khắc phục được nhược điểm của các mô hình tuyến tính.

1.10.9 Disadvantages of neural networks

Neural networks while powerful, are not without drawbacks & difficulties:

- **Computational Intensity.** Large neural network training can be a laborious & computationally demanding process that demands a lot of computing power.
- **Black box nature.** As "black box" models, neural networks pose a problem in important applications since difficult to understand how they make decisions.
- **Overfitting.** Overfitting is a phenomenon in which neural networks commit training material to memory rather than identifying patterns in data. Although regularization approaches help to alleviate this, problem still exists.
- **Need for Large datasets.** For efficient training, neural networks frequently need sizable, labeled datasets; otherwise, their performance may suffer from incomplete or skewed data.

– Mạng nơ-ron tuy mạnh mẽ, nhưng cũng có những nhược điểm & khó khăn:

- **Cường độ tính toán.** Việc huấn luyện mạng nơ-ron lớn có thể là một quá trình tốn kém & tính toán, đòi hỏi rất nhiều sức mạnh tính toán.
- **Bản chất hộp đen.** Là các mô hình "hộp đen", mạng nơ-ron gây ra một vấn đề trong các ứng dụng quan trọng vì khó hiểu cách chúng đưa ra quyết định.
- **Quá khớp.** Quá khớp là hiện tượng mạng nơ-ron ghi nhớ dữ liệu huấn luyện thay vì xác định các mẫu trong dữ liệu. Mặc dù các phương pháp chính quy hóa giúp giảm thiểu vấn đề này, nhưng vấn đề vẫn tồn tại.
- **Nhu cầu về tập dữ liệu lớn.** Để huấn luyện hiệu quả, mạng nơ-ron thường cần các tập dữ liệu có kích thước lớn và được gắn nhãn; nếu không, hiệu suất của chúng có thể bị ảnh hưởng do dữ liệu không đầy đủ hoặc bị lệch.

1.10.10 Applications of neural networks

Neural networks have numerous applications across various fields:

1. **Image & Video Recognition.** CNNs are extensively used in applications e.g. facial recognition, autonomous driving & medical image analysis.
2. **Natural Language Processing (NLP).** RNNs & transformers power language translation, chatbots & sentiment analysis.
3. **Finance.** Predicting stock prices, fraud detection & risk management.
4. **Healthcare.** Neural networks assist in diagnosing diseases, analyzing medical images & personalizing treatment plans.

5. **Gaming & Autonomous Systems.** Neural networks enable real-time decision-making, enhancing user experience in video games & enabling autonomous systems like self-driving cars.

– Mạng nơ-ron nhân tạo có nhiều ứng dụng trong nhiều lĩnh vực khác nhau:

1. **Nhận dạng hình ảnh & Video.** Mạng nơ-ron nhân tạo (CNN) được sử dụng rộng rãi trong các ứng dụng như nhận dạng khuôn mặt, lái xe tự động & phân tích hình ảnh y tế.

2. **Xử lý ngôn ngữ tự nhiên (NLP).** Mạng nơ-ron nhân tạo & bộ biến đổi hỗ trợ dịch thuật ngôn ngữ, chatbot & phân tích cảm xúc.

3. **Tài chính.** Dự đoán giá cổ phiếu, phát hiện gian lận & quản lý rủi ro.

4. **Chăm sóc sức khỏe.** Mạng nơ-ron nhân tạo hỗ trợ chẩn đoán bệnh, phân tích hình ảnh y tế & cá nhân hóa kế hoạch điều trị.

5. **Trò chơi & Hệ thống tự động.** Mạng nơ-ron nhân tạo cho phép ra quyết định theo thời gian thực, nâng cao trải nghiệm người dùng trong trò chơi điện tử & cho phép các hệ thống tự động như xe tự lái.

2 Miscellaneous

Tài liệu

[BB24] Christopher M. Bishop and Hugh Bishop. *Deep Learning: Foundations & Concepts*. 2024 edition. Springer, 2024, p. 669.

[BJP20] Jean-Pierre Briot, Gaëtan Jadjeres, and François-David Pachet. *Deep Learning Techniques for Music Generation*. Computational Synthesis & Creative Systems. Springer, 2020, p. 284.

[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.

[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.

[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.

[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.