Topics in the Foundations of Artificial Intelligence

A Reader

Draft from June 4, 2025.
Comments welcome!

Levin Hornischer
Levin.Hornischer@lmu.de
https://levinhornischer.github.io/FoundAI/

Contents

Preface

This is the reader for the course *Advanced Topics in the Foundations of AI*. Editions of the course were given during the summer semesters 2023, 2024, and 2025 at *LMU Munich* as part of the *Master in Logic and Philosophy of Science*. The reader is still updated as the course progresses. A website for the course—with the most up to date version of this reader—is found at

https://levinhornischer.github.io/FoundAI/.

Comments I'm happy about any comments: spotting typos, finding mistakes, pointing out confusing parts, or simply questions triggered by the material. Just send an informal email to Levin.Hornischer@lmu.de.

Content In recent years, artificial intelligence and, in particular, machine learning made great—but also disconcerting—progress. However, their foundations are, unlike other areas of computer science, less well understood. This situation is sometimes compared to being able to build steam engines without having a theory of thermodynamics. This course is about the foundation of AI. After an introduction to AI, we first review computability theory as the established theory of symbolic AI. This serves as a benchmark for what a theory should deliver when we turn to modern deep-learning-based AI. Here, a general theory is not yet discovered, but the topic of much recent research. We overview several approaches: statistical learning theory, statistical mechanics, scientific computing, dynamical systems, etc. We discuss what questions these approaches can answer and what a successful theory should still deliver. While this theory describes the abilities of AI systems as a whole, we may also ask what one specific AI system is doing. So, at the end, we turn to interpretable AI: explaining what an AI system is doing in human-understandable terms.

Objectives In terms of content, the course aims to convey an overview of the foundations of AI—including both classic material and cutting-edge research. In terms of skills, the course aims to teach the ability to both mathematically and philosophically assess the different approaches to the foundations of AI.

Prerequisites In order to appreciate the literature, the course requires basic familiarity with mathematics (calculus, linear algebra, probability theory), logic (including, ideally, computability theory), and AI (neural networks). Some papers also use more advanced concepts from topology, probability theory, or category theory, so you should also be prepared to read up on those. But they are not assumed: the seminar sessions are, among others, meant to get clearer on these concepts. Programming skills will of course be useful, but will not be assumed.

Schedule and organization The course is organized as a seminar. Hence, for each session, we have assigned readings, which we then discuss during the session. The reading for each week is announced in the schedule on the course's website. The readings are roughly organized by topic, forming the chapters of this reader.

Background material Some helpful short explainer videos on AI are found here. An excellent series on (the mathematics of) neural networks is found here.

Moreover, the material of my companion course on the Philosophy and Theory of AI might also be helpful. You can take the present course independently of that companion course and vice versa, but they do complement each other. The companion course is more introductory and looks at a broader range of philosophical issues connected to AI and how to theorize about them, while the present course focuses specifically on the more mathematical foundations of neural networks.

A recent edited collection on the mathematical foundations of AI is Grohs and Kutyniok (2022).

Layout These notes are informal and partially still under construction. For example, there are margin notes to convey more casual comments that you'd rather find in a lecture but usually not in a book. Todo notes indicate, well, that something needs to be done. References are found at the end.

This is a margin note.

This is a todo note

Notation Throughout, 'iff' abbreviates 'if and only if'.

Part I.

Introduction to AI

1. Introduction

Summary

We give an outlook of the course and of this document.

The field of AI is typically characterized along the lines of aiming to build "machines that can compute how to act effectively and safely in a wide variety of novel situations" (Russell and Norvig 2021, p. 19). In the present part I, and especially in chapter 2, we briefly collect basic terminology and concepts in AI, to make sure we're all on the same page.

Regardless of the definition, it is helpful to distinguish two main traditions in AI. They go by varying names, with different connotations depending on the community that uses them, including the following.

1. *Symbolic AI*: classicist, logic-based, Good Old-Fashioned AI (GOFAI), etc.

   Example: An algorithm or computer program that, given as input a position in a game of chess, outputs the next best move. This algorithm was written by a programmer.

2. *Subsymbolic AI*: connectionist, non-logicist, machine learning, deep learning, etc.

   Example: A neural network that, given as input a pixel image of a handwritten digit, outputs the digit depicted on the image. The neural network was trained to become better at this mapping using thousands of data points, i.e., input images labeled with the digit depicted on them.

(Some might distinguish a third tradition—*statistical AI*—which, in a sense, sits between the two preceding traditions: Like symbolic AI it typically has 'interpretable' variables, but they are now continuous random variables, and like subsymbolic AI it typically processes the information in a continuous way.)

For symbolic AI, we have a well established theory due to logic and computability theory. We cover this in part II. It gives a good idea of which questions a theory of AI should answer.

For subsymbolic AI—which is most of modern AI—, a general theory has not yet been discovered. In part III, we overview different approaches to such a theory.

In chapter 5, we review the main results that the standard theory of machine learning—i.e., statistical learning theory—can deliver. But, in chapter 6, we also look at what is still missing for the concrete case of the neural networks that modern AI is built on.

Chapter 7 uses results from the field of scientific computing to unveil impossibilities for neural networks. Chapter 8 looks at how statistical mechanics bridged the gap between a micro-level description of a system to a human-understandable macro-level description, and it discusses how this also can be done for neural networks. Chapter 9 views neural networks as dynamical systems and shows how the theory of dynamical systems yields insights into what neural networks can and cannot compute.

Finally, chapter 10 lists further approaches toward a theory of deep learning: we can pick topics based on time and interest.

In part IV, we 'zoom in' from a general theory of AI systems to the behavior of specific AI systems. This is the field of interpretable AI: explaining what an AI system is doing in human-understandable terms. Specifically, we will look at mechanistic interpretability in chapter 11. Depending on time and interest, we may look at further topics listed in chapter 12.

2. Background

Readings

- A textbook introduction to the field of AI: Russell and Norvig (2021, ch. 1).

Key concepts

- History of AI: Ada Lovelace, Alan Turing, McCulloch & Pitts, Logic Theorist, Dartmouth workshop, summers and winters, big data, deep learning revolution.
- Types of AI: symbolic, subsymbolic, statistical
- Definitions of AI: acting humanly (Turing test), thinking humanly (cognitive modeling), thinking rationally (logic, probability), acting rationally (rational agent; perfect vs limited rationality) See figure 2.1.
- Types of learning tasks: Supervised learning, unsupervised learning, reinforcement learning. Machine learning pipeline (conceptualization, data, model, deployment).
- Key concepts of artificial neural networks: neurons, layers, feedforward/recurrent, weights, activation function, loss function, backpropagation, learning rate, local/global minima (equilibrium), regularization, overfitting/underfitting.

In this session, we discuss the main reading to get an understanding of each of the key concepts—the basic AI terminology—mentioned above. These key concepts are further illuminated in the additional material mentioned below.

The field of AI has a rich history that is worth knowing, also to put recent developments into perspective (first bullet point above). As already mentioned in the previous chapter, it divides into symbolic and subsymbolic (and statistical) approaches (second bullet point). The goal of AI is typically defined as building intelligent machines (third bullet point).

	empiricist	*rationalist*
thought process	thinking humanly (cognitive modeling)	thinking rationally (logic & probability)
behavior	acting humanely (Turing test)	acting rationally (decision theory)

Figure 2.1.: The four definitions of artificial intelligence of Russell and Norvig (2021, ch. 1) according to whether an AI system should realize thought processes (first row) or behavior (second row) and whether the benchmark is human (left column) or ideal (right column) performance.

Depending on how 'intelligent' and 'machine' is understood, this yields different more precise goals, as illustrated in figure 2.1.

When it comes to actually building such 'intelligent machine', no matter how made precise, there are two options. One is to directly hand-craft the ready machine, while the other is to build an initial machine and then train it to become better at its task. The former approach comes naturally with the symbolic approach: The programmer thinks of a good algorithm to solve the task at hand and then build a program that implements this algorithm. The second approach comes naturally with the subsymbolic approach: First pick a neural network architecture and then update its parameters using training data to get better performance. This hence also goes by the name of *machine learning*. Russell and Norvig (2021) write:

> In a nutshell, AI has focused on the study and construction of agents that *do the right thing*. What counts as the right thing is defined by the objective that we provide to the agent. This general paradigm is so pervasive that we might call it the *standard model*. ... [Though,] the standard model assumes that we will supply a fully specified objective to the machine. ... The problem of achieving agreement between our true preferences and the objective we put into the machine is called the *value alignment problem* ... Ultimately, we want agents that are *provably beneficial* to humans (Russell and Norvig 2021, 22–21, emphasis altered).

In class, we go trough the construction of a neural network for the standard MNIST task (classifying handwritten digits). We use this to explain the terms above (neurons, layers, activation function, loss function,

backpropagation). As homework, you can do the coding exercise from the companion course (link below in 'further material'). This shows how such a neural network is *actually* implemented on a computer (using Python as a programming language).

Further material

- A very accessible overview, written at the beginning of the deep learning revolution: Boden (2016, ch. 1 and 4).
- A great introduction to AI: M. Mitchell et al. (2019). "Artificial intelligence: A guide for thinking humans." In.
- A concise introduction to deep learning and its philosophical aspects: C. Buckner (2019). "Deep learning: A philosophical introduction." In: *Philosophy Compass* 14.10, e12625. DOI: https://doi.org/10.1111/phc3.12625.
- The Stanford Encyclopedia of Philosophy entry on artificial intelligence: Bringsjord and Govindarajulu (2024).
- Also see the background material mentioned in the preface: e.g., explainer videos on AI here and an excellent series on (the mathematics of) neural networks here.
- You can also take a look at the introduction chapter of my companion course on the Philosophy and Theory of AI. This also contains an introduction to actually coding a neural network (without assuming any coding knowledge).
- A great interactive visualization of neural networks is found here.

Part II.

Theory of symbolic AI

3. Computability theory

Readings

- A short overview of symbolic AI: Flasiński (2016, ch. 2)
- An overview of computability and complexity theory: Immerman (2021)

Key concepts

- Turing machine
- Church–Turing thesis
- Halting problem
- Entscheidungsproblem
- Tiling problems
- Gödel's incompleteness theorems
- Computational complexity theory: P vs NP

What's characteristic to symbolic AI is the following fundamental assumptions about how to build intelligent machines cf. Flasiński 2016, p. 15.

- The model/machine that describes/realizes intelligent behavior is given in an *explicit* way. In particular, the knowledge in the model is represented in a *symbolic* way (e.g., logic formulas, finite graphs, etc.).
- The information processing operations of the model can be described as formal operations (logical inference rules, grammars of a formal language, etc.) on the symbolic representations.

For example, consider a model that plays chess. The states of the game can symbolically be described by noting the positions of each piece on the board and who is to play. Thus, the model's knowledge about the state of the game can be represented symbolically. There is an initial state (the

starting position) and many goal states (where the machine checkmates the opponent). The formal operations of the model should be such that they describe which move the model should make given a current state. These could be given as a list of rules of the form "If the current state of the game is such that these pieces are in this position, then do so-and-so". Taken together, these rules determine the strategy of the model, and if it is a good model, this strategy often results in a win.

When it comes to theorizing about symbolic AI, the main question that we would like to answer is:

> Which problems can be solved by a symbolic model/computer?

After all, before embarking on solving a particular task with a new AI model, we would like to know whether there is a solution in the first place or if we will just waste our time trying to find one.

This question is answered by computability theory. But to give a rigorous answer, the vague terms first have to be made precise (i.e., be *explicated*, as philosophers would say). This will be done as follows.

1. A *problem* will be formally analyzed as a partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$: an input n encodes an instance of the (symbolic) problem and $f(n)$ encodes its solution, if it exists.

2. A *computer* will be formally analyzed as a Turing machine.

3. A computer *solving* a problem will be formally analyzed as the Turing machine computing a function which is identical to the function describing the problem.

We will now develop this in detail. Once we have formalized the answer, we can start developing mathematical tools to answer it.

Turing machines In a remarkable feat of mathematical philosophy, Alan Turing explicated in 1936 the intuitive notion of (symbolic) computation with an abstract machine, now known as the *Turing machine*. It processes symbols, that are written on a tape, according to its program, which is simple a table of instructions. At any given time step, the machine is in some internal state and positioned at some cell of the tape. Based on its internal state and the symbol written in that cell, the program says (a) in which internal state the machine should go into next, (b) which symbol the machine should write into the present cell instead, and (c) whether it should move to the left or the right cell next. The machine has one

designated internal state known as the halting state: if it enters that, it stops its computation. The Turing machine then computes the following function: It takes as input a string of symbols (consisting only of symbols that also the machine uses). They are written on the otherwise blank tape. Then the machine start processing according to its program. If it enters its halting state, then the string of symbols that is written on the tape at that moment is the output of the function. If the machine never enters a halting state and 'loops forever', then no output is ever given.

The formal definition of a Turing machine varies, but—as we mention below—all of them are equivalent. Here we pick the simplest one e.g. Soare 2016, p. 7. A *Turing machine* is a pair $M = (Q, \delta)$ where

- $Q = \{q_0, q_1, \ldots, q_n\}$ with $n \geqslant 1$ is a finite set whose elements are called *internal states*.
- Here q_1 is called the *starting state* and q_0 is called the *halting state*.
- $\delta : Q \times S \to Q \times S \times \{R, L\}$ is a partial function, where $S := \{1, B\}$ is called the set of *symbols*. Here 'B' stands for blank, 'R' for right, and 'L' for left.

The machine computes the following partial function $f_M : \mathbb{N} \rightharpoonup \mathbb{N}$. Given the input n, we write $n+1$ consecutive 1's on the otherwise blank tape (i.e., all other cells of the tape contain 'B'). We start the machine in the starting state and positioned on the leftmost cell containing a 1, called the starting cell. Then we operate according to δ: given the present state q and read symbol s, if $\delta(q, s)$ is defined, say as (q', s', X), then the machine changes into state q', replaces s by s', and moves on position to the right if $X = R$ or to the left if $X = L$. If the machine at some point reaches the halting state q_0, then the machine halts and the output $f_M(n)$ is the number of 1's written on the tape; otherwise the machine does not halt and $f_M(n)$ is not defined.

Exercise: Can you figure out which function the Turing machine $M = (\{q_0, q_1, q_2\}, \delta)$ computes, where δ is given as follows:

$$(q_1, 1) \mapsto (q_1, 1, R)$$
$$(q_1, B) \mapsto (q_2, 1, R)$$
$$(q_2, B) \mapsto (q_0, 1, R)$$

If you'd like to know the answer, take a look at the footnote.[1]

Problems and solutions A *computational problem* is formalized as a partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$. This may seem artificially restricted. Why should all computational problems only have to do with natural numbers. Our motivating chess example, for instance, was not about natural numbers.

The insight, though, is that every symbolic computational problem can be *encoded* into a partial function on the natural numbers. In the chess example, we can encode every chess position and every possible move with a natural number, so our problem of "solving chess" is formally identified with the partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ which, on input n, is defined iff n encodes a game position, and then outputs the number $f(n)$ which encode the best possible next move. This is a well-defined mathematical function, and the question of whether our chess problem is solvable becomes the formal question whether there is a Turing machine that computes that function f.

In general, we say that a partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is *Turing-computable* (informally: the problem f is solvable) if there is a Turing machine M such that $f_M = f$.

The expected functions indeed turn out to be computable. For example, the addition function that maps a code for a pair of numbers (n, m) to the number $n + m$. Similarly for multiplication, exponentiation, ordering, etc.

Church-Turing thesis As indicated, other versions of Turing machines also exists, where, e.g., more than just two symbols are allowed, or several tapes are used, or several halting states can be defined. Moreover, many other models of computation have been developed (e.g., Church's lambda calculus or Gödel's recursive functions). Remarkably, though, they all turned out to be equivalent: a function is computable in one model if and only if it is computable in the other model. This has lead to the conviction that the formal concept of Turing computability really coincides with the intuitive concept of (symbolic) computabilility. This is known as the *Church–Turing thesis*: that a function is Turing computable if and only if it is computable in the intuitive sense. Note that this is a *philosophical* and not a mathematical claim. It is a claim about an informal philosophical concept: namely, that of computability. It cannot be proven by formal mathematical means.

[1] The function that is computed is $f_M(n) = n + 3$, because running the machine adds two new 1's to the tape to the $n + 1$ many 1's that are written there initially, so there are $n + 3$ many 1's on the tape after halting.

Methods to establish (non-) computability Having established a formalization of our initial question—namely, which problems are solvable by a computer?—, the next question is how we can establish whether or not a problem is computable. We have mentioned some problems that are computable, but Turing already was also asking: are there functions that are not computable?[2] In other words, are there things that no computer can ever solve? He considered the so-called *halting problem*: Given a Turing machine e and an input n, decide if the Turing machine on this input halts or not. A Turing machine that solves this problem is a 'meta-program' that checks whether an inputted programs ever gets stuck in a loop. So it would be very desirable to have such a meta-program, but Turing showed that this cannot exist.

Theorem 3.1 (Turing)**.** *There is no Turing machine that solves the halting problem. In other words, the* h *function that maps a (code of a) Turing machine* e *and an input* n *to*

$$h(e,n) := \begin{cases} 1 & \textit{Turing machine } e \textit{ halts on input } n \\ 0 & \textit{otherwise} \end{cases}$$

is not computable.

Proof. Toward a contradiction, assume h is computable. Then we can define the Turing machine e that, on input n, computes $h(n,n)$ and if this is 0 it outputs 0 and otherwise loops forever. Now, we plug e into h. Since h always outputs 0 or 1, there are two cases:

- Case 1: $h(e,e) = 0$. Then, by definition, the Turing machine e with input e outputs 0, i.e., halts, so $h(e,e) = 1$.
- Case 2: $h(e,e) = 1$. Then, by definition, the Turing machine e with input e runs forever, i.e., does not halt, so $h(e,e) = 0$.

Thus, we indeed get a contradiction. □

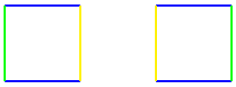
(This establishes that there is no Turing machine computing the halting problem on every input. But even the problem of computing the halting function on a specific, even very small input is very difficult: this is the *busy beaver problem*. A popular science introduction to it is found here.)

[2] Turing machines are finite objects, there are only countably many Turing machines, so there are also only countably many computable functions. On the other hand, there are uncountably many function $\mathbb{N} \to \mathbb{N}$ (by Cantor's diagonalization theorem). So there must be *some* uncomputable functions. But explicitly providing one is more difficult.

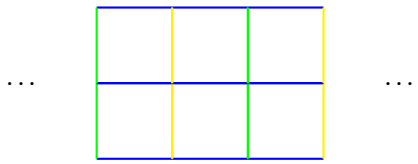
Knowing the non-computability of the halting problem is a stepping stone for other non-computability results. A common method to prove a given problem to be non-computable is a *reduction argument*: assume there is a Turing machine solving the given problem and the use this to build a Turing machine that solves the halting problem. Since the latter is impossible, the former is, too.

This is how the historically important *Enscheidungsproblem* was answered in the negative: there is no algorithm to decide whether a given formula in first-order logic is logically valid (i.e., true in all interpretations). If there was such an algorithm, we could solve the halting problem: given Turing machine e and input n, there is a clever way to build a formula φ of first-order logic such that φ is valid iff e halts on input n, so if we can decide validity we can also decide halting.

Another nicely visual problem that can be shown to be non-computable is the *tiling problem*. As an input one gets finitely many square tiles (all of the same size) and each of their edges has some color. For example:



Then one should decide if it is possible to tile the plane with them, given infinitely many copies of each tile. So one has to put them together so they match colors at adjacent edges. For example, with the above tiles, once can do it:



But with other sets of tiles, one cannot tile the plane. Again one can show with a reduction argument that there is no Turing machine that, given a set of tiles as input, output 'yes' or 'no' according to whether or not the set of tiles tiles the plane.

In fact, one of the most famous theorems of 20th century mathematics can also be proven with a reduction argument: Gödel's first incompleteness theorem. (Gödel's original proof was different: Turing machines did not exist yet at that time.)

The formulation that we state here uses the concept of a computable enumeration. A set of string is *computably enumerable* if there is a Turing machine that, as it keeps processing, writes on the tape strings separated

by a blank cell such that all written strings are in the set and each string of the set eventually is written on the tape. This is weaker than the set of string being *decidable*: for that we require a Turing machine that take a string as input and outputs 'yes' or 'no' depending on whether or not the string is in the set. In the computably enumerable case, we only get confirmation if the string is in the set. If the string is not in the set, then, as we follow the enumeration, we never know if the string actually is in the set but has not yet appeared or actually is not in the set and because of that did not appear. On the other hand, in the decidable case, we also get confirmation if the string is not in the set.

Theorem 3.2 (Gödel's first incompleteness theorem)**.** *There is no computably enumerable list of axioms that prove all and only the true statements of elementary mathematics (including, say, basic arithmetic, finite combinatorics, etc.).*

This version is discussed, e.g., in this online lecture.

Proof. For a reduction argument, assume there is such a list of axioms. By systematically applying all the deduction rules, we can computably enumerate all and only the theorems of the theory (not just the axioms). But then we can solve the halting problem: Given a Turing machine e and an input n, consider the statement 'Turing machine e halts on input n'. This is a statement of elementary mathematics. So either this statement or its negation shows up on the enumeration of all theorems (one of the two has to). So we start the enumeration and wait until one of the two shows up, and depending on which, we say the Turing machine halts or does not halt. This hence would be a computable procedure to solve the halting problem, which is impossible. □

Just arithmetic is enough, because other 'elementary mathematics' can be encoded with natural numbers, so questions there reduce to questions about arithmetic.

This theorem is so important because it shattered Hilbert's dream of writing down, in a systematic—and hence computably enumerable—way all the axioms of mathematics, so that every mathematical truth can be immutably established by deriving it from the axioms. But it also played a big role in the philosophy of AI in the 1980s and 90s, when Roger Penrose revived an argument of Lucas with the conclusion that no machine can ever reach human-level intelligence. The idea of the argument is that any such machine would have to do mathematical reasoning and, qua machine, it would do it in a computable way, but, by Gödel's incompleteness result, there would be statements (e.g., about the halting behavior of some Turing machine) that we, with our human intelligence, can see to be true, but the machine cannot establish. This argument had a controversial discussion: for a short overview, see here.

Computational complexity So far, we have talked about the distinction between computable functions (or problems) and non-computable ones. However, even if one knows that that the problem one attempts to solve is computable, it is still important to know just how *complex* it is, i.e., how long (or more generally, how many resources) the best possible Turing machine would take to solve it. This is studied by *complexity theory*. It has a whole zoo of classes of problems that have a similar complexity. But the two most famous classes are:

- The class P of polynomial-time decision problems: the time to compute 'yes' or 'no' for a given input n takes at most $p(n)$ time steps, for some polynomial p like, e.g., n^2 (rather than exponentially long, i.e., 2^n).
- The class NP of problems where it can at least be verified in polynomial time that the provided answer for a given input is indeed correct.

Intuitively one might expect that $\mathsf{P} \neq \mathsf{NP}$, i.e., that there are problems where one cannot quickly find the correct answer, but for a suggested answer one can at least quickly verify that it is correct (think, e.g., of solving a puzzle). And it is indeed widely believed that $\mathsf{P} \neq \mathsf{NP}$—a lot of modern cryptography is even built on this belief. However, despite the simplicity of this question, it is one Millenium Prize Problems whether $\mathsf{P} = \mathsf{NP}$ or not.

Further material

- A popular science introduction to the busy beaver problem: here.
- S. Aaronson (2011). *Why Philosophers Should Care About Computational Complexity*. arXiv: 1108.1791 [cs.CC]. URL: https://arxiv.org/abs/1108.1791

Part III.

Theory of deep learning

4. Desiderata for a theory of machine learning

Now that we have seen the theory for symbolic computation, we can ask how do those insights about classical computers carry over to machine learning? We first consider similarities and dissimilarities between those two paradigms. Then we consider—based on knowing a successful theory for symbolic computation—what a theory of machine learning should deliver. In the remainder of the course, we then consider various theories that deliver on these desiderata.

From Turing machines to machine learning First, there is a clear similarity. One can think of the weights of a neural network as its program. The weights determine how the neural network computes the output for a given input. This is just like the program determines how the Turing machine computes the output for a given input. Thus, the question about computability for classical computers becomes the following question for machine learning: Are there weights for the neural network with which it computes the given function? This problem is known as determining the expressive power of neural network architectures.

Second, there are also dissimilarities between classical computers and machine learning. In fact, Turing (1950) already pointed them out:

> An important feature of a learning machine is that its teacher will often be very largely ignorant of quite what is going on inside, although he may still be able to some extent to predict his pupil's behaviour. ... This is in clear contrast with normal procedure when using a machine to do computations: one's object is then to have a clear mental picture of the state of the machine at each moment in the computation. (Turing 1950, pp. 458–9)

In other words, in classical programming, someone has to come up with a Turing machine or program code, based on a clear intuitive algorithm. In machine learning, in contrast, we do not hand-craft the weights based on our task understanding (since we are 'largely ignorant' of what these weights represent). Rather, the machine should 'program itself' through

	Classical computation	*Deep learning*
Standard architecture	Turing machine	Feed-forward neural net
Other architectures	λ-calculus, automata, ...	CNNs, transformers, ...
Instructions	machine program	neural network weights
Input/output	discrete	continuous and/or discrete
Problem	ground-truth function	ground-truth function
Solving problem	invent machine program	train from data
Interpretability	yes	no
Correctness	identity (all-or-nothing)	closeness (degree-based)
Complexity	time and space resources	required data and epochs

Figure 4.1.: Analogies and disanalogies between classical computation and (supervised) machine learning.

training with data. So we not only care about whether the machine learning model can compute the given function with some weights. We also care about whether we can find these parameters via training.[3]

Third, Turing continues with another difference between classical computation and machine learning:

> Intelligent behaviour presumably consists in a departure from the completely disciplined behaviour involved in computation, but a rather slight one ... Processes that are learnt do not produce a hundred per cent. certainty of result; if they did they could not be unlearnt. (Turing 1950, p. 459)

In other words, in classical programming, we work with discrete symbols representing inputs and outputs and have a 'completely disciplined' relation between the two. In machine learning, in contrast, we often have continuous inputs and/or outputs and there is some uncertainty about the relation between the two. Hence, in classical programming, correctness is an all-or-nothing matter (qualitative): for every input, the Turing machine outputs the very same symbol as the given function. In machine learning, on the other hand, correctness is a matter of degrees (quantitative): given an input, how likely is the machine to output something that is close—in the sense of a continuous metric—to the desired output.

What a theory for neural networks should deliver The preceding subsection is summarized as follows (see figure 4.1). Problems in classical computability are identified with functions that have discrete inputs and

[3] In classical computability theory, one cares about whether there *is* a Turing machine that computes a given function. One typically does not care about *how* one finds this Turing machine. Complexity theory offers answers to how 'complicated' even the simplest Turing machine must be that computes the given function. But we may also ask how 'difficult' it is to find such a Turing machine (cf. Friedberg 1958).

outputs. A solution is a Turing machine which computes the ground-truth function. Finding this Turing machine is done by programming, i.e., hand-crafting the program of the machine. Classical computability theory describes which problems are solvable and which are not. In machine learning, we, too, want to find a machine—e.g., a deep neural network—that computes the ground-truth function. However, the machine should find the program—i.e., the correct weights—*automatically*, by training on data from the ground-truth function. Moreover, the ground-truth function can have continuous input and outputs, so we no longer require the model to find an identical function, but one that is very close.

In analogy with the successful theory for symbolic computation, we can now formulate what a theory for neural networks should deliver (items 1–4 below). Like in the symbolic case, we start with the notion of a problem.

1. *Problem/solution (non-uniform).* A problem in machine learning is finding the ground-truth function f, given some data D about this function. A solution is a neural network that, when trained on this data, updates its weights so that it computes f to high accuracy.

Of course, one can also pose versions of this problem. For example, we can pose it for machine learning models other than neural networks. We can ask it for various learning algorithms, i.e., backpropagation with different choices of training hyperparameters (e.g., learning rate, number of epochs, batch size, etc.) and even for learning algorithms beyond backpropagation. We can also ask to find the right weights only with high probability instead of always (since the training usually is stochastic). Finally, we can also require the discovered function to not just be accurate but also robust (e.g., not easily susceptible to adversarial attacks).

In practice, we are also interested in a uniform solution, i.e., that a single neural network can be used to solve a whole set of machine learning problems.[4] This is because, in practice, we of course do not know the ground-truth function f (otherwise we do not need to bother trying to learn it). We only know that the actual ground-truth function is one in a set F of possible functions. But since we do not know which, we want our neural network to work whatever function in F turns out to be the actual ground-truth function.

[4]The distinction between uniform and non-uniform solutions is also well-known in computability theory. A set $\{f_0, f_1, f_2, \ldots\}$ of problems has a *non-uniform* solution if, for each n there is a Turing machine M_n that computes f_n (i.e., $f_n = f_{M_n}$). The set has a *uniform* solution if there is a Turing machine M that takes as input n and outputs a (code for a) Turing machine M_n such that $f_n = f_{M_n}$.

2. *Problem/solution (uniform).* A uniform problem in machine learning is a set of pairs (f, D) of ground-truth functions f with data D about that function. A solution is a neural network such that, for each (f, D), if the network is trained on D, it computes f to high accuracy.

Once the problem has been identified, it is clear what the theory should deliver:

3. *Computability.* A computability theory for neural networks should describe which machine learning problems (uniform or non-uniform) are solvable and which are not.

Further, like in the symbolic case, once we know what constitutes finding a solution, we can ask how complex it is to find, if solvable at all.

4. *Complexity.* A complexity theory for neural networks describes the minimal amount of data and neural network size—and maybe other hyperparamters (like learning rate, number of epochs, batch size)—needed to solve the problem.

Finally, let's turn to one of the dissimilarities between the symbolic and the subsymbolic case mentioned by Turing: namely, interpretability. For Turing machines, we have a clear idea of the algorithm that it implements: so interpretability is a non-issue and hence it does not need to be dealt with by the theory. However, for machine learning it is an issue. We can observe the behavior of the trained machine, but it is difficult to understand the algorithm that it implements in human-understandable terms. Hence we may pose as a further problem:

5. *Interpretability.* Describe in human-understandable terms the procedure that the trained machine implements to solve the problem.

We may consider this problem as separate from the other ones: one question is to ask if there is any machine that solves the problem and another question is whether we can also interpret that machine.

Now let's see what answers are already known.

5. Statistical learning theory

Readings

- On statistical learning theory and the No-Free-Lunch theorem: Shalev-Shwartz and Ben-David (2014), chapters 2–3 (basic set up), chapter 5 (No-Free-Lunch), sections 6.1–4 (fundamental theorem of statistical learning theory).
- On the approximation theorem: One of the classics, Hornik et al. (1989).

Key concepts

- Statistical learning theory
- PAC learnability
- No-Free-Lunch theorem
- Bias-complexity tradeoff
- VC-dimension and the fundamental theorem of PAC learning
- Universal approximation theorem

Statistical learning theory was developed as the theory of machine learning. If any, it is the theory of non-symbolic AI, even though, as we discuss at the end of the lecture, there is some disconnect between theory and practice. It aims to provide a framework that is as general as possible to ask and answer questions about what is and what is not learnable.

Statistical learning theory framework A learner gets input x from some domain X (e.g., a particular papaya) and they need to label this input with a label $h(x) = y$ from the label space Y, here the only labels are 0 (e.g., not tasty) and 1 (e.g., tasty). The learner will see finitely many training data $(x_1, y_1), \ldots, (x_m, y_m)$ of input-output pairs, and based on that suggest a general rule $h : X \to Y$. We now formalize this idea.

- The *domain set* X. Typically the elements are vectors, e.g., $x = (0.1, 0.7)$ saying that the object x is described completely by feature 0 (e.g., the papayas color) having value 0.1 and feature 1 (e.g., the papayas softness) having value 0.7.
- The *label set* Y. Typically $Y = \{0, 1\}$.
- The *training data* $S = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ which is a finite subset of $X \times Y$. These are the examples that the learner has access to (e.g., the m-many papayas they have bought, checked the color and softness, and then tasted them to determine whether they were tasty or not). Elements of S are also called data points or training examples, and S is also called training set.
- The *learner's output*: Based on the training data, the learner has to output a prediction rule, i.e., a rule for how to label points from the domain set. This rule is described as a function $h : X \to Y$, which is also called predictor, hypothesis, or classifier.
- The *learning algorithm* A: Typically, the learner will follow a general learning algorithm that works not just for the specific training set S but also for other ones. So it is a function A that takes as input finite subsets of $X \times Y$ and outputs a predictor $A(S) : X \to Y$. One learning algorithm that we introduce below is empirical risk minimization.
- *Sampling*: We assume there is a probability distribution D on the domain set X which describes how likely it is that we see a particular point x (e.g., how likely it is that the learner gets papaya x when going to the market). One says D is a (probabilistic) data-generation model. Importantly, the learner has *no* access to this probability distribution. Rather D describes how the world actually is.
- *True risk.* The error of a classifier h is the probability that it does not predict the correct label. Formally, we assume there is a true labeling function $f : X \to Y$ (that the learner aims to find) and the error of h is

  $$L_{D,f}(h) := D(\{x \in X : h(x) \neq f(x)\}).$$

  This is also called generalization error, risk, or true error (to distinguish it from the empirical error/risk that we introduce later). (Below, in the NFL Theorem, we generalize the assumption of a true function f and instead work with a probability distribution over $X \times Y$.)

- *Empirical risk*. The true risk is defined with respect to the distribution D and the true labeling function f, both of which the learner has no access to. The learner can only calculate the error of their predictions on the training dataset:

$$L_S(h) := \frac{1}{m}\left|\{i \in \{1,\dots,m\} : h(x_i) \neq y_i\}\right|.$$

- *Empirical Risk Minimization* (ERM) is the learning paradigm of coming up with a predictor h that minimizes the empirical risk $L_S(h)$. This has to be restricted though: If the learner is allowed to pick any predictor h, they can pick the h which, on input x, predicts y if (x, y) is in the training set and 0 otherwise. This minimizes the empirical risk (it is 0), but it generalizes badly to points x outside of the training set (it all assigns them the same label): one says h *overfits* the data. To avoid this, one fixes a set H of allowed predictors (and the just mentioned h wouldn't usually be allowed). This H is called the *hypothesis class*. The learner has to choose this in advance, before seeing the data. This choice of H is the *inductive bias* (or, positively, prior knowledge) of the learner: they are biased to certain predictors before seeing data (or know a priori that they will better fit the data). (This will lead to the bias-complexity trade-off that we'll discuss later.)

PAC learning:

- What would it mean for a learner—with their choice of hypothesis class H—to be 'good', i.e., to produce correct prediction rules? We want that, no matter what the true distribution D and labeling function f are, given a required confidence parameter δ and accuracy parameter ϵ, there is a number of samples $m = m(\delta, \epsilon)$ such that, if we sample m-many examples from D labeled with f, then with confidence $1 - \delta$ the learner knows that they are correct up to ϵ, provided there is a correct hypothesis in the first place. One says: we are *probably approximately correct* (PAC). The formal definition is:

Though one might ask: why quantify over all D and f and not just those that are likely for the task? E.g., for the papayas some distributions and labelings are more likely than others. (Cf. margins theory *discussed by Belkin (2021, sec. 3.3).*

- A hypothesis class H is *PAC-learnable* if there is a function $m_H : (0,1) \times (0,1) \to \mathbb{N}$ and a learning algorithm A such that: For every $\epsilon, \delta \in (0,1)$, for every probability distribution D over X, and for every labeling function $f : X \to \{0,1\}$, if the realizability assumption holds (i.e., there is $h^* \in H$ with $L_{D,f}(h^*) = 0$), then, when running A on $m \geqslant m(\epsilon, \delta)$ i.i.d. (independently and identically distributed)

examples sampled with D and labeled with f, the algorithm return a hypothesis h such that, with probability of at least $1 - \delta$ (over the choice of the examples), $L_{D,f}(h) \leqslant \epsilon$.

- This can be generalized by dropping the realizability assumption, going beyond the binary label case (multiclass and regression, using more general loss functions).

The No-Free-Lunch and the PAC Learning theorems No-Free-Lunch Theorem (NFL Theorem):

- We avoided the overfitting problem of the ERM learning paradigm by making explicit the inductive bias/prior knowledge of the learner in the form of a hypothesis class H. One may ask: is this really necessary, i.e., can there be a learner who is successful without using any task-specific prior knowledge and can thus solve any task? One way to interpret the NFL theorem is that it answers 'no': there cannot be such a *universal learner*.

For a discussion of interpretations of the NFL theorem, see the further reading below (Sterkenburg and Grünwald 2021).

- A bit more precisely, the NFL Theorem takes a learning task to be given by an unknown distribution P over $X \times Y$, and the goal of the learner is to find a predictor $h : X \to Y$ whose risk $L_P(h)$ is small. As mentioned, this generalizes the framework so far: So far we had a distribution D on X and a true label function f. This determines the distribution P on $X \times Y$ according to which the probability of (x, y) is 0 if $y \neq f(x)$ and otherwise the probability of x according to D. Now we allow any distribution P on $X \times Y$, so we don't assume there is a single true label function, but only a conditional probability of how likely a label is given the input. Accordingly, the loss is

$$L_P(h) = P(\{(x, y) \in X \times Y : h(x) \neq y\},$$

  which is also known as *0-1 loss*. The NFL Theorem then says: It is not the case that there is a learning algorithm A and a training set size m such that, for every distribution P over $X \times Y$, if A receives m-many i.i.d. samples from P, there is a high chance it outputs a predictor h that has low risk. In other words, for every learner, there is a task on which it fails, even though another learner succeeds.

- Formally, the NFL theorem is stated as follows. Let A be any learning algorithm for the task of binary classification with respect to the 0-1 loss over a domain X. Let m be any number smaller than $|X|/2$,

representing a training set size. Then, there exists a distribution P over $X \times \{0,1\}$ such that:

1. There exists a function $f : X \to \{0,1\}$ with $L_P(f) = 0$, but
2. With probability of at least $1/7$ over the choice of $S \sim P^m$ we have that $L_P(A(S)) \geqslant 1/8$.

So A fails on this task while the ERM learner with hypothesis class $H = \{f\}$ succeeds.

Bias-complexity tradeoff

- Since we don't have a universal learner—who performs best possible on any task—, we cannot get around analyzing a learner on a specific task. So we want to get some guarantee how bad the true error of the learner is. As we'll describe now, we can split this error up into two parts, but there is a tradeoff in that improving one part tends to worsen the other, and vice versa.

- *Error decomposition*. We can analyze the error $L_P(A(S))$ of our learning algorithm/ERM predictor A on a given dataset S as $\epsilon_{app} + \epsilon_{est}$ with

$$\epsilon_{app} := \min_{h \in H} L_P(h) \qquad \epsilon_{est} := L_P(A(S)) - \epsilon_{app},$$

  where ϵ_{app} is the *approximation error* (the minimal risk achievable by a predictor from the hypothesis class) and ϵ_{est} is the *estimation error* (the difference between the empirical minimal risk $L_P(A(S))$ and the true minimal risk ϵ_{app}).

- *Bias-complexity tradeoff*. To reduce the error $L_P(A(S))$ we hence want to reduce both the approximation error and the estimation error. However, this comes at a tradeoff:

  1. To reduce the approximation error, we want a large, more *complex* hypothesis class; but, as we saw, this may lead to *overfitting* (empirical risk is low, but true risk is high), so the the estimation error can be high.
  2. To reduce the estimation error, we might rather choose a small, more *biased* hypothesis class, but this results in a high approximation error, because the predictors are now *underfitting* the data.

So learning theory studies how to find rich hypothesis classes H for which we still have reasonable estimation errors. We will further discuss this for neural networks in chapter 6 (the universal approximation theorem will show that—as the name suggests—their approximation error is zero).

VC-dimension and the fundamental theorem of PAC learning:

- How can we ensure our learner has a low error? Reducing the approximation error is a matter of picking the right prior knowledge encapsulated as a hypothesis class. For the estimation error, the key is to realize that PAC learnability bounds the estimation error. So we would like to have a guarantee for PAC learnability. This is provided by the so-called VC-dimension, which is defined as follows.

Under the realizability assumption, the approximation error is zero, so the estimation error can be bounded by ϵ with probability $1-\delta$ when sampling a dataset of size $\geqslant m(\delta, \epsilon)$.

- First, some helpful terminology: If H is a hypothesis class and $C \subseteq X$, then H *shatters* C if $\{h \restriction C : h \in H\} = Y^C$, where $h \restriction C$ is the restriction of the function $h : X \to Y$ to the set C, and Y^C is the set of all functions from C to Y.

- The *VC-dimension* of a hypothesis class H is the maximal size of a finite set $C \subseteq X$ that can be shattered by H. If H shatters sets of arbitrarily large size, it has infinite VC-dimension.

The idea is: If H shatters a set C of size $2m$, then we cannot learn H using m examples: all ways of labeling the other m instances in C are possible according to some hypothesis in H, i.e., prior knowledge doesn't exclude any of those.

- The *fundamental theorem*: Let H be a hypothesis class of functions from X to $\{0, 1\}$, and let the loss function be the 0-1 loss. Then the following are equivalent

  1. H is PAC learnable
  2. Any ERM rule is a successful PAC learner for H
  3. H has a finite VC-dimension.

  In fact, if the VC-dimension of H is $d < \infty$, there are constants C_1 and C_2 such that H is PAC learnable with sample complexity

$$C_1 \frac{d + \log(1/\delta}{\epsilon} \leqslant m(\epsilon, \delta) \leqslant C_2 \frac{d \log(1/\epsilon) + \log(1/\delta)}{\epsilon}.$$

The full version of the theorem has some further equivalent conditions.

The Universal Approximation Theorem There are several universal approximation theorems. Here we discuss an early one by Hornik et al. (1989).[5] (For a modern textbook version including a proof sketch, see

[5] Interestingly, Hornik et al. (1989, p. 360) mention Kolmogorov's superposition theorem which has a similar form of the approximation theorem but would require a possibly

Berner et al. (2022, thm. 1.16 on p. 16); and the further reading Kratsios 2021 below.)

It considers the set Σ^r of functions $\mathbb{R}^r \to \mathbb{R}$ that can be realized by a feed-forward neural network with r-many input neurons, one output neuron, and one hidden layer using an activation function $G : \mathbb{R} \to \mathbb{R}$ that is non-decreasing and converges to 0 (resp., 1) as its argument goes to $-\infty$ (resp., $+\infty$).

Intuitively, the universal approximation theorem should say that the functions in Σ can approximate any non-pathological function $f : \mathbb{R}^r \to \mathbb{R}$ arbitrarily well. This is made precise as follows.

The 'non-pathological' functions are the measurable functions. We don't state the precise definition here, since this would take us a bit astray. For us it suffices to know that basically any function that naturally occurs is measurable. So let M^r be the set of all measurable functions $\mathbb{R}^r \to \mathbb{R}$. Any function realized by a neural network is measurable, so $\Sigma^r \subseteq M^r$.

Now how to say that Σ^r can approximate any function in M^r arbitrarily well? For that, we will define a metric ρ on M^r, i.e., a way to measure the distance $d = \rho(f, g) \in \{x \in \mathbb{R} : x \geqslant 0\}$ between two functions $f, g \in M^r$. Then to say that Σ^r can approximate any function in M^r is to say that Σ^r is *dense* in M^r, i.e., for every $f \in M^r$ and for every $\epsilon > 0$, there is $g \in \Sigma^r$ such that $\rho(f, g) < \epsilon$.

So it remains to define the metric ρ on M^r. The idea is that two functions $f, g \in M^r$ are close if "there is only a small probability that they differ significantly" (Hornik et al. 1989, p. 361). This is made precise as by fixing a probability measure μ on $\mathbb{R}^r$ and defining

$$\rho_\mu(f, g) := \inf\Big\{\epsilon > 0 : \mu\big(\{x : |f(x) - g(x)| > \epsilon\}\big) < \epsilon\Big\}.$$

So for a given 'significance level' $\epsilon > 0$, we check on how many inputs $x \in \mathbb{R}^r$ our functions f and g differ by more than ϵ. If the probability of encountering such an x is low, i.e., smaller than ϵ, then our functions pass the test and are at least ϵ-close; otherwise they are more than ϵ far apart. Now we look for the smallest ϵ's for which our functions are at least ϵ-close. The infimum of all of them then is the distance between f and g.

Now the universal approximation theorem says (Hornik et al. 1989, thm. 2.4 on p. 362):

different activation function for every neuron (rather than a single one for all neurons as it is done in feed-forward neural networks). This idea has been taken up very recently (Liu et al. 2024) investigating a neural network architecture with different, learnable activation functions.

- For any probability measure μ on $\mathbb{R}^r$, the set Σ^r of functions realized by a neural network is ρ_μ-dense in M^r.

Further material

- A discussion of the no-free-lunch theorem: Sterkenburg and Grünwald (2021).
- A modern, more general approach to the universal approximation theorem: Kratsios (2021).

6. Refining statistical learning theory

Readings

- Berner et al. (2022), pages 1–31.
- Belkin (2021), sections 1–3.

Key concepts

- Approximation error, generalization error, and optimization error.
- The generalization puzzle/problem
- Interpolation and over-parametrization
- Double descent generalization curve

Berner et al. (2022) describe in section 1.1.2 (i.e., pages 5–23) the insights that statistical learning theory can provide when it is applied to neural networks. On page 13, they analyze the true error of a neural network into three components. (The first two are closely related to the approximation and estimation error that we considered in the general setting; the third is new.)

- The approximation error: how far the best possible prediction rule of the network is away from the truly best possible prediction rule, i.e., the Bayes-optimal function.
- The (uniform) generalization error: among the prediction rules of the network, the maximal difference between the true error of the prediction rule and the empirical error of the prediction rule.
- The optimization error: the difference in empirical error between the prediction rule found by the neural network after training and the prediction rule that minimizes the empirical risk.

The third type of error is added because with neural networks we do *algorithmic* empirical risk minimization (in the words of Belkin (2021, p. 217)):

backpropagation (i.e., the implementation of stochastic gradient descent for neural networks) is an algorithm that aims to find the prediction rule (i.e., the set of weights) that minimizes empirical risk (i.e., the error on the training set). The general theory that we covered previously is *algorithm-independent* in the sense that it only assumes *that* an (ERM-) learner outputs a prediction rule with minimal empirical risk but doesn't say so *how* this rule is found.

On pages 13–23, the then provide theorems to bound these errors.

- The approximation error is bound by the universal approximation theorem, which we already discussed.
- The generalization error is bound—using the VC-dimension—with high probability by

  $$\sqrt{\frac{p \log(p)}{m}},$$

  where p is the number of parameters of the neural network and m is the size of the training set. (But: when there are more parameters than training samples, as is common for neural networks, this bound is vacuous. This is picked up as an open question below.)
- The optimization error is bound by convergence guarantees of stochastic gradient descent. (But: the assumptions of these guarantees are typically some form of convexity that, however, is usually not satisfied for neural networks. Again this is picked up as an open question.)

For such convergence guarantees also see the Polyak–Łojasiewicz condition in Belkin (2021, sec. 4.1).

In section 1.1.3 (i.e., pages 23–31), Berner et al. (2022) overview the questions that are left open by classical statistical learning theory when applied to neural networks. In particular, this includes:

- The *generalization puzzle* (as Berner et al. (2022, p. 25) call it): Neural networks are typically *over-parametrized*, i.e., have more parameters than training samples. This means that they can typically *interpolate* (or overfit) the training data, i.e., fit exactly to the training data (hence have zero empirical risk). Hence classical bounds on the generalization error are vacuous, and indeed the bias-complexity tradeoff of classical statistical learning theory would predict neural networks to perform poorly. Nonetheless, they perform well. Explaining this is the generalization puzzle.

- The *wonders of optimization* (as Belkin (2021, sec. 4) calls it): Stochastic gradient descent, as it is performed by neural networks, typically converges to good local minima, despite typically being non-convex. Hence the classical convergence guarantees from optimization theory cannot be given. Why is it that neural networks still successfully optimize?

Belkin (2021, sec. 3.7) extends the classical picture of the U-shape trade-off between bias (i.e., small hypothesis class) and complexity (i.e., complex/interpolating hypothesis class). Belkin extends this 'classical regime' further to the 'modern regime' with an even more complex hypothesis class that is over-parametrized. The result is the *double decent generalization curve* (for the figure, see the top of page 3 here): Not only can we descent from the interpolation threshold (where we have a hypothesis class that can just overfit/interpolate the training data) into the classical regime by reducing the complexity of the hypothesis class. We can also descent into the modern regime by making the hypothesis class even more complex.

Belkin (2021, sec. 3.6) provides a heuristic for why this works: In this modern, interpolating regime, there are many prediction rules of the learner which interpolate the training data. Among those, the over-parametrized neural network learner tends to pick the *smoothest* one. (A smooth function is one with, intuitively speaking, no sharp edges.) Being smooth is a sense of being simple, so we can understand this inductive bias of the neural network as an instance of Occam's razor: among the explanation that are consistent with the evidence (i.e., the prediction rules with zero empirical risk) pick the simplest one (i.e., the smoothest one).

Further material

- Rest of the papers.

7. Scientific computing

Readings

- Colbrook et al. (2022)

Key concepts

- Inverse problems
- Smale's 18th problem: the limits of AI
- Stability vs accuracy tradeoff
- Adversarial attacks

With the universal approximation theorems, we know that neural networks can approximate any (measurable) function arbitrarily well. But knowing that a neural network must exist that approximates a given function is one thing. Another thing is to *actually find* this network. The paper by Colbrook et al. (2022) shows that these two things really can come apart: that although there must an approximating network, it is impossible for us to find it.

An analogous difference is described by intuitionistic logic: between truth/existence and provability/construction.

Concretely, they consider neural networks aiming to solve *inverse problem*. Some reality x determines, via a linear map A and some random noise e, the sensor readings $y = Ax + e$. The task is to reconstruct, given the sensor readings y, what the most minimal reality x must have been giving rise to these sensor readings.

Their result (theorem 1 and 2 combined) then states: There are mappings that map training data of an inverse problem to a neural network solving it, but no training algorithm (like, e.g., stochastic gradient descent) can compute such a mapping (producing neural nets with reasonable accuracy).

The broader context is the question: Why do neural networks tend to be unstable (adversarial attacks, hallucinations, etc.), even in situations—like inverse problems—where stable and accurate neural nets exist (since the universal approximation theorem guarantees that some neural network can approximate the stable solution)?

This suggest developing for neural networks an analogous theory like computability and complexity theory for symbolic computation.

Further material

- Related papers to the above: Bastounis et al. (2022), Boche et al. (2022)
- On decidability of learnability: Caro (2023)
- An older overview of computability theory of neural networks: Šíma and Orponen (2003).
- On formal language theory and neural networks: Delétang et al. (2023), Strobl et al. (2024)

8. Using statistical mechanics

Readings

- Roberts and Yaida (2022)
- Bahri et al. (2020)

Key concepts

- Analogy between statistical mechanics in physics and neural networks in machine learning

Further material

- The book by Roberts and Yaida (2022) has been presented in a course (`https://deeplearningtheory.com/lectures/`).

9. Dynamical systems

Readings

- Overview: Bournez and Pouly (2021)

Key concepts

- Dynamical system
- Monoid action, differential equation
- Classification of models of computation into space-discrete vs space-continuous and time-discrete vs time-continuous.
- Characterizing the expressive power of different neural network architectures in terms of symbolic models of computation
- General purpose analog computer and the equivalence with computable analysis

The key take-away from this chapter is that basically all forms of computation, be it symbolic or not, can be regarded as dynamical systems. A dynamical system consists out of a state space (consisting of the states that the system can be in) and a dynamics (describing how the system evolves over time from the current state to the next states). And, indeed, any computing machine is, at any given time, in some state, and to compute, it updates its state step-by-step according to some procedure or algorithm.

Thus, dynamical systems theory provides a rich and all-encompassing framework to talk about different models of computation. The paper provides an overview of how this perspective furthers our understanding of specific models of computation, focusing on the analog ones (including neural networks) rather than the digital ones (like Turing machines).

Further material

- Further reading: Saxe et al. (2014) and E et al. (2022).
- Blog post: `https://www.eigentales.com/NTK/`

- Popular science videos on analog computation: here and here

10. Further approaches

Topological data analysis

Readings

- Naitzat et al. (2020)
- Overview: Hensel et al. (2021)

Key concepts

- TBA

Further material

- For a short explanation of persistent homology, see here.
- For a popular science application to neuroscience, see here.

Geometric deep learning

Readings

- Bronstein et al. (2021)

Key concepts

- Applying the Erlangen Program to Machine Learning

Further material

- A website of the project is found here.

Category theory as a language of machine learning

Readings

- A categorical framework to describe how Large Language Models move from next-word-probability distributions to syntax and semantics of language: Bradley et al. (2021)
- Overview interactions of category theory and machine learning: Shiebler et al. (2021)

Key concepts

- TBA

Further material

- The paper by Bradley et al. (2021) is also covered on the first author's great blog here.
- More on categories for AI in this course (`https://cats.for.ai/`) and this list of papers on the topic (`https://github.com/bgavran/Category_Theory_Machine_Learning`).
- A(nother) language for design patterns of AI architectures: van Bekkum et al. (2021).

Part IV.

Interpretable AI

11. Mechanistic interpretability

Readings

- Geiger et al. (2024)

Key concepts

- TBA

Further material

- TBA

12. Further topics

Verification of neural networks

Readings

- Albarghouthi (2021)

Key concepts

- TBA

Further material

- TBA

Post-hoc explainability

Readings

- Bilodeau et al. (2024)

Key concepts

- TBA

Further material

- TBA

Bibliography

Aaronson, S. (2011). *Why Philosophers Should Care About Computational Complexity*. arXiv: 1108.1791 [cs.CC]. URL: https://arxiv.org/abs/1108.1791 (cit. on p. 17).

Albarghouthi, A. (2021). *Introduction to Neural Network Verification*. arXiv: 2109.10317 [cs.LG] (cit. on p. 43).

Bahri, Y. et al. (2020). "Statistical Mechanics of Deep Learning." In: *Annual review of condensed matter physics* 11.1, pp. 501–528. DOI: 10.1146/annurev-conmatphys-031119-050745 (cit. on p. 36).

Bastounis, A., A. C. Hansen, and V. Vlačić (2022). *The extended Smale's 9th problem – On computational barriers and paradoxes in estimation, regularisation, computer-assisted proofs and learning*. arXiv: 2110.15734 [math.OC] (cit. on p. 35).

Belkin, M. (2021). "Fit without fear: remarkable mathematical phenomena of deep learning through the prism of interpolation." In: *Acta Numerica* 30, pp. 203–248 (cit. on pp. 25, 31 sqq.).

Berner, J. et al. (2022). "The Modern Mathematics of Deep Learning." In: *Mathematical Aspects of Deep Learning*. Ed. by P. Grohs and G. Kutyniok. Cambridge: Cambridge University Press, pp. 1–111. DOI: 10.1017/9781009025096.002 (cit. on pp. 29, 31 sq.).

Bilodeau, B. et al. (2024). "Impossibility theorems for feature attribution." In: *Proceedings of the National Academy of Sciences* 121.2, e2304406120 (cit. on p. 43).

Boche, H., A. Fono, and G. Kutyniok (2022). *Inverse Problems Are Solvable on Real Number Signal Processing Hardware*. arXiv: 2204.02066 [eess.SP] (cit. on p. 35).

Boden, M. A. (2016). *AI: Its nature and future*. Oxford: Oxford University Press (cit. on p. 8).

Bournez, O. and A. Pouly (2021). "A Survey on Analog Models of Computation." In: *Handbook of Computability and Complexity in Analysis*. Ed. by V. Brattka and P. Hertling. Cham: Springer International Publishing, pp. 173–226. DOI: 10.1007/978-3-030-59234-9_6 (cit. on p. 37).

Bradley, T.-D., J. Terilla, and Y. Vlassopoulos (2021). *An enriched category theory of language: from syntax to semantics*. arXiv: 2106.07890 [math.CT] (cit. on p. 40).

Bringsjord, S. and N. S. Govindarajulu (2024). "Artificial Intelligence." In: *The Stanford Encyclopedia of Philosophy*. Ed. by E. N. Zalta and U. Nodelman. Summer 2024. Metaphysics Research Lab, Stanford University (cit. on p. 8).

Bronstein, M. M. et al. (2021). *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges*. arXiv: 2104.13478 [cs.LG] (cit. on p. 39).

Buckner, C. (2019). "Deep learning: A philosophical introduction." In: *Philosophy Compass* 14.10, e12625. DOI: https://doi.org/10.1111/phc3.12625 (cit. on p. 8).

Caro, M. C. (12/2023). "From undecidability of non-triviality and finiteness to undecidability of learnability." In: *International Journal of Approximate Reasoning* 163, p. 109057. DOI: 10.1016/j.ijar.2023.109057. URL: https://arxiv.org/abs/2106.01382 (cit. on p. 35).

Colbrook, M. J., V. Antun, and A. C. Hansen (2022). "The difficulty of computing stable and accurate neural networks: On the barriers of deep learning and Smale's 18th problem." In: *Proceedings of the National Academy of Sciences* 119.12, e2107151119. DOI: 10.1073/pnas.2107151119. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.2107151119. URL: https://www.pnas.org/doi/abs/10.1073/pnas.2107151119 (cit. on p. 34).

Delétang, G. et al. (2023). *Neural Networks and the Chomsky Hierarchy*. arXiv: 2207.02098 [cs.LG] (cit. on p. 35).

E, W., J. Han, and Q. Li (2022). "Dynamical Systems and Optimal Control Approach to Deep Learning." In: *Mathematical Aspects of Deep Learning*. Ed. by P. Grohs and G. Kutyniok. Cambridge University Press, pp. 422–438. DOI: 10.1017/9781009025096.011 (cit. on p. 37).

Flasiński, M. (2016). *Introduction to Artificial Intelligence*. Cham: Springer. DOI: https://doi.org/10.1007/978-3-319-40022-8 (cit. on p. 10).

Friedberg, R. M. (1958). "A learning machine: Part I." In: *IBM Journal of Research and Development* 2.1, pp. 2–13 (cit. on p. 20).

Geiger, A. et al. (2024). *Causal Abstraction: A Theoretical Foundation for Mechanistic Interpretability*. arXiv: 2301.04709 [cs.AI]. URL: https://arxiv.org/abs/2301.04709 (cit. on p. 42).

Grohs, P. and G. Kutyniok, eds. (2022). *Mathematical Aspects of Deep Learning*. Cambridge University Press. DOI: 10.1017/9781009025096 (cit. on p. 2).

Hensel, F., M. Moor, and B. Rieck (2021). "A Survey of Topological Machine Learning Methods." In: *Frontiers in Artificial Intelligence* 4. DOI: 10.3389/frai.2021.681108 (cit. on p. 39).

Hornik, K., M. Stinchcombe, and H. White (1989). "Multilayer feedforward networks are universal approximators." In: *Neural Networks* 2.5, pp. 359–366. DOI: https://doi.org/10.1016/0893-6080(89)90020-8 (cit. on pp. 23, 28 sq.).

Immerman, N. (2021). "Computability and Complexity." In: *The Stanford Encyclopedia of Philosophy*. Ed. by E. N. Zalta. Winter 2021. Metaphysics Research Lab, Stanford University (cit. on p. 10).

Kratsios, A. (2021). "The Universal Approximation Property." In: *Annals of Mathematics and Artificial Intelligence* 89.435-469. DOI: https://doi.org/10.1007/s10472-020-09723-1 (cit. on pp. 29 sq.).

Liu, Z. et al. (2024). *KAN: Kolmogorov-Arnold Networks*. arXiv: 2404.19756 [cs.LG] (cit. on p. 29).

Mitchell, M. et al. (2019). "Artificial intelligence: A guide for thinking humans." In: (cit. on p. 8).

Naitzat, G., A. Zhitnikov, and L.-H. Lim (2020). "Topology of Deep Neural Networks." In: *Journal of Machine Learning Research* 21.184, pp. 1–40. URL: http://jmlr.org/papers/v21/20-345.html (cit. on p. 39).

Roberts, D. A. and S. Yaida (2022). *The Principles of Deep Learning Theory: An Effective Theory Approach to Understanding Neural Networks*. Cambridge:

Cambridge University Press. DOI: 10.1017/9781009023405. URL: https://arxiv.org/abs/2106.10165 (cit. on p. 36).

Russell, S. J. and P. Norvig (2021). *Artificial Intelligence: A Modern Approach*. Pearson series in artificial intelligence. Harlow: Pearson (cit. on pp. 4, 6 sq.).

Saxe, A. M., J. L. McClelland, and S. Ganguli (2014). *Exact solutions to the nonlinear dynamics of learning in deep linear neural networks*. arXiv: 1312.6120 (cit. on p. 37).

Shalev-Shwartz, S. and S. Ben-David (2014). *Understanding Machine Learning: From Theory to Algorithms*. A copy for personal use only at https://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/copy.html. New York: Cambridge University Press (cit. on p. 23).

Shiebler, D., B. Gavranović, and P. Wilson (2021). *Category Theory in Machine Learning*. URL: https://arxiv.org/abs/2106.07032 (cit. on p. 40).

Šíma, J. and P. Orponen (2003). "General-Purpose Computation with Neural Networks: A Survey of Complexity Theoretic Results." In: *Neural Computation* 15, pp. 2727–2778 (cit. on p. 35).

Soare, R. I. (2016). "Turing Computability: Theory and Applications." In: *(No Title)*. DOI: https://doi.org/10.1007/978-3-642-31933-4 (cit. on p. 12).

Sterkenburg, T. F. and P. D. Grünwald (2021). "The no-free-lunch theorems of supervised learning." In: *Synthese* 199.3-4, pp. 9979–10015. DOI: 10.1007/s11229-021-03233-1 (cit. on pp. 26, 30).

Strobl, L. et al. (2024). *What Formal Languages Can Transformers Express? A Survey*. arXiv: 2311.00208 [cs.LG] (cit. on p. 35).

Turing, A. M. (1950). "Computing Machinery and Intelligence." In: *Mind* 59.236, pp. 433–460. DOI: https://doi.org/10.1093/mind/LIX.236.433 (cit. on pp. 19 sq.).

Van Bekkum, M. et al. (2021). "Modular design patterns for hybrid learning and reasoning systems: a taxonomy, patterns and use cases." In: *Applied Intelligence* 51, pp. 6528–6546. DOI: 10.1007/s10489-021-02394-3 (cit. on p. 40).