ARTICLE IN PRESS

Physics Reports xxx (xxxx) xxx

Contents lists available at ScienceDirect

Physics Reports

journal homepage: www.elsevier.com/locate/physrep

Data science applications to string theory

Fabian Ruehle *

CERN, Theoretical Physics Department, 1 Esplanade des Particules, Geneva 23, CH-1211, Switzerland
Rudolf Peierls Centre for Theoretical Physics, University of Oxford, Department of Physics, Clarendon Laboratory, Parks Road, Oxford OX1 3PU, UK

ARTICLE INFO

Article history:
Received 1 June 2019
Received in revised form 26 September 2019
Accepted 30 September 2019
Available online xxxx
Editor: S. Stieberger

ABSTRACT

We first introduce various algorithms and techniques for machine learning and data science. While there is a strong focus on neural network applications in unsupervised, supervised and reinforcement learning, other machine learning techniques are discussed as well. These include various clustering and anomaly detection algorithms, support vector machines, and decision trees. In addition, we review data science techniques such as genetic algorithms and topological data analysis. This first part of the review makes some reference to concepts in physics, but the explanations and examples do not assume any knowledge of string theory and should therefore be accessible to a wide variety of readers with a physics background. After that, we illustrate applications to string theory. We give an overview of existing string theory data sets and describe how they can be studied using data science techniques. We also explain the computational complexity involved in the investigation of string vacua. Example codes that illustrate the techniques introduced in this review are available from Fabian Ruehle (0000).

Contents

* Correspondence to: CERN, Theoretical Physics Department, 1 Esplanade des Particules, Geneva 23, CH-1211, Switzerland.
E-mail address: fabian.ruehle@cern.ch.

https://doi.org/10.1016/j.physrep.2019.09.005
0370-1573/

Please cite this article as: F. Ruehle, Data science applications to string theory, Physics Reports (2019), https://doi.org/10.1016/j.physrep.2019.09.005.

1. Introduction and motivation

In this review, we provide an introduction to data science techniques that are used to study large data sets, and outline how these methods can be applied to problems encountered in string theory. We will mainly present machine learning algorithms, but other techniques such as topological data analysis will be discussed as well. By choosing simple examples with dummy data, the data science part will be kept accessible to all physicists. Example applications to string theory data will be discussed after all techniques have been introduced.

1.1. What is data science and machine learning?

The goal of data science is to develop methods to extract knowledge and insights from a typically very large set of data. Within the recent decade, the ability to monitor, generate, and store data has increased drastically, and data science has become extremely important in order to make sense out of this gigantic amount of data. Since the amount of data is usually too large to go through item by item, one needs some more effective way for extracting information.

Machine learning is a branch of data science which uses heuristics and approximations in order to perform tasks whose exact execution would take too long. One distinguishes (at least) three branches of machine learning (cf. Fig. 1 for an overview):

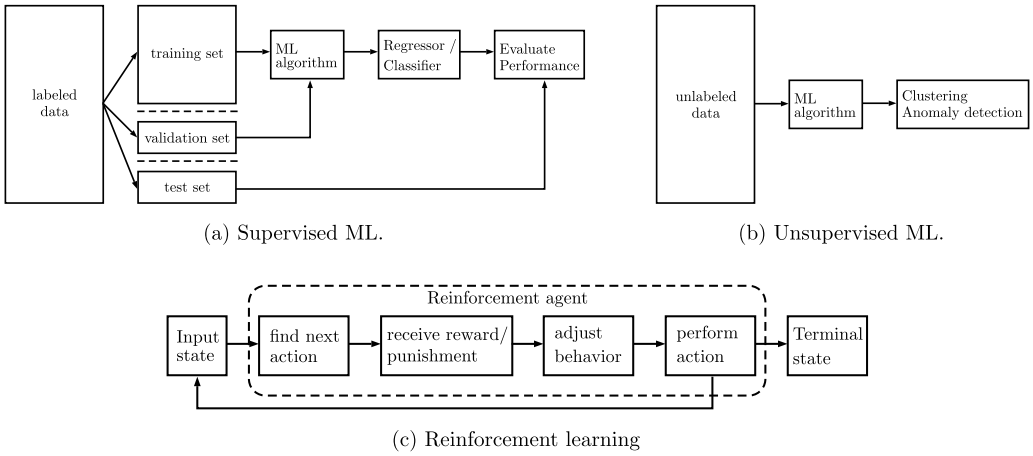
- **Unsupervised machine learning:** Often, one is interested in identifying clusters and/or outliers within a given data set. This is a typical field of application for unsupervised machine learning. The algorithm uses some method to identify data that is similar by some measure (usually the distance of data points) in order to group data into clusters, or to detect anomalies or outliers. This is done without a human telling the algorithm what to look for.
- **Supervised machine learning:** In other cases, the desired result is known for some of the data, and the algorithm should learn from this data in order to produce results for data for which the results are unknown. Typical applications are classification and regression tasks. In classification tasks, the algorithm is taught via examples which type of input data belongs to the same class. In regression, the algorithm is taught which input data produces which output data. In either case, the algorithm learns by adjusting internal parameters in order to learn which output the human wants to assign to a given input based on the examples provided.
- **Reinforcement learning:** In some cases, one knows what one is looking for but does not know how to find it. Such search tasks are well-suited for reinforcement learning. The idea is that the algorithm is not told what to do exactly, but it receives feedback based on its performance, encouraging it to perform actions that lead to the desired result.

Another data science technique that can be used to search through a large space of possibilities in order to find feasible or interesting solutions are genetic algorithms. All that is required for their application is some function that can quantify the feasibility of a solution; the algorithm then simply maximizes this feasibility function. Similar to reinforcement learning, genetic algorithms can be used to find solutions in large configuration spaces.

A further field of data science that is concerned with studying the structure of data is called topological data analysis. The idea is to assign a topology to data (i.e. to a set of points), which can be used to identify structure in data.

In order to be able to judge which problems can benefit from applying the above-mentioned data science techniques, it is useful to have a notion of how hard a problem actually is. If one is trying to solve a problem for which a dedicated, efficient, exact solution algorithm is known, it is usually better to use this algorithm rather than machine learning. Complexity theory in computer science provides a measure for how hard a problem is. Very roughly, it assigns problems to different classes. The easy problems are in the class P and the hard problems are in the class NP\P. We will make this more precise later, but roughly, for P problems a fast algorithm (polynomial running time) exists, while for problems in NP\P, only very slow algorithms (exponential running time or worse) are known. Most likely, no fast algorithms to solve these problems exist, i.e. P$\neq$NP and NP\P is non-empty.

Please cite this article as: F. Ruehle, Data science applications to string theory, Physics Reports (2019), https://doi.org/10.1016/j.physrep.2019.09.005.



(a) Supervised ML. (b) Unsupervised ML.

(c) Reinforcement learning

Fig. 1. Diagrams showing the workflow of supervised ML, unsupervised ML, and Reinforcement learning.

While the field of data science is not very new, and has already been successfully applied in many branches of science (biology, chemistry, experimental particle physics, cosmology, ...) and society (social networks, digital assistants, games, self-driving cars, ...), it had not been applied to string theory until June 2017, when four groups [1–4] published independently work that applied machine learning to string theory. Since then, many machine learning and data science techniques have been used in order to

- speed up computations in string theory, predominantly in algebraic (or toric) geometry,
- identify mathematical structures in string theory,
- study the structure of string vacua,
- find new string vacua.

1.2. What is string theory?

String theory (see e.g. [5–8] for a textbook introduction) aims at unifying quantum field theory with general relativity. Such a theory is ultimately needed if one wants to study energy scales where all four (known) fundamental forces are equally strong. String theory starts with the innocent sounding postulate that particles are not point-like but extended (i.e. one-dimensional strings). These strings can either be open, i.e. an interval, or closed, i.e (topologically) a circle. This has far-reaching consequences: for example, (critical) string theory requires 26 spacetime dimensions, and superstring theory requires 10 spacetime dimensions for consistency. This is just one of the examples where mathematical consistency of string theory imposes stringent constraints on the space of possible solutions. In fact, there are only 5 known consistent supersymmetric 10D string theories. These all seem to be related via dualities and can be understood as limiting cases of an overarching 11D theory known as M-Theory [9], which can in turn be formulated as a twelve-dimensional theory called F-Theory [10]. In order to make contact with the 4D world as we perceive it, the extra 6 dimensions of string theory (or 7 of M-Theory or 8 of F-Theory) are assumed to be compact and small [11], such that they evade detection.

The space of consistent vacua of string theory is called the **string landscape** [12], while the theories that are apparently consistent (or plainly inconsistent, but these are not very interesting) but cannot be derived from string theory are called the **swampland** [13]. In order to understand the physics that can or cannot arise from string constructions, one needs to understand the string landscape or swampland, respectively. Identifying such physics can in turn lead to proposed signatures to test or falsify the theory.

The problem with the string landscape is that it is unfathomably big and finding solutions to its consistency conditions is computationally very hard. While it is essentially unique in 11D and only permits a few formulations in 10D, the space of string vacua in 4D is gigantic. The reason is twofold: first there are a huge number of different choices for the compact space, and second there is a huge number of additional data or boundary conditions, known as fluxes and branes, that are necessary to uniquely specify string theory in 4D. Unfortunately, we lack a selection mechanism for the compactification space, the fluxes, and the branes, which means we somehow need to deal with the huge number of possibilities. Just to give an idea of the numbers: early estimates argue that there are $\mathcal{O}(10^{500})$ boundary data choices for any (typical) 6D compactification space [14]. Other results showed that there are $\mathcal{O}(10^{755})$ inequivalent compactification geometries for F-Theory [15]. Estimates on the entire landscape are even much larger, $\mathcal{O}(10^{272,000})$ [16].

1.3. Why do we need data science techniques in string theory?

As we have seen in the previous section, understanding the string landscape is a formidable task. On top of the data set being unfathomably big, one faces the problem that finding mathematically consistent and phenomenologically viable background configurations requires solving problems which are generically NP-complete, NP-hard, or even undecidable.

Let us motivate this point in more detail. Usually, problems are hard to solve either because they need a lot of computational steps or because they need a lot of memory (or both). Sometimes, one can speed up computation time by using more memory (to store intermediate results). We will be mainly concerned with the number of computation steps needed to solve a problem. Usually, one is interested in the scaling behavior with the number of computational steps for large problems and thus only lists the leading order. As a simple example, take the computational complexity of multiplying two $N \times N$ matrices. This comes down to computing N^2 scalar products, and each scalar product involves N multiplications and N additions. The total complexity for matrix multiplication of two $N \times N$ matrices is thus $\mathcal{O}(N^3)$ (there exist clever ways of speeding this up and performing matrix multiplication in $\mathcal{O}(N^{2.376})$ steps using the Coppersmith–Winograd algorithm [17]). Note that we have only counted the number of multiplication and addition operations and neglected the number of steps necessary to actually perform the multiplications and additions, which grows quadratically and logarithmically in the number of digits, respectively. Since running time scales polynomially in the input size N, the problem is in P. There exist different algorithms with different complexities ($\mathcal{O}(N^3)$ vs. $\mathcal{O}(N^{2.376})$), but this does not change membership in the complexity class P.

Next, let us compare a model in P (with polynomial running time) to an NP-hard problem. Assume that we can perform 10^9 operations per second on our computer. Then multiplying two $N \times N$ matrices with $N = 1,000$ would take one second. Multiplying two $10,000 \times 10,000$ matrices takes half an hour and two $10^5 \times 10^5$ matrices take 11.5 days. If one cannot (or does not want to) wait that long, one could use a 10 times faster computer, which would cut the computation time down to one day, or one could parallelize the problem and use 10^5 computers to compute 1 scalar product each and be done within a nano second.

To contrast this with an NP problem, we look at the so-called traveling salesman problem (TSP). The task is to find a route for a traveling salesman to visit N cities while traveling the shortest distance possible. The naive solution is to just try all $N!$ possibilities and take the one that requires the least amount of traveling, which has a complexity of $\mathcal{O}(N!)$, i.e. it grows faster than any polynomial in the number of cities. A slight variation of the problem is to ask whether there exists a route that has length at most ℓ_* and visits every city. Even for this version of the problem there is no known efficient algorithm, so one has to try different routes to see whether or not one has length at most ℓ_*. However, given a proposed solution, one can verify quickly (i.e. in polynomial time) that the proposal is indeed a solution. Problems for which solutions can be verified in polynomial time (but one need not be able to find this solution in polynomial time) are in the complexity class NP, which is a short form for non-deterministic polynomial running time. While this is of course true for all problems in P, there exist also problems for which there is indeed no known fast way of finding solutions, such as TSP. In fact, if one were to find a polynomial time solution for TSP, it has been shown that any problem in NP could be solved quickly; however, most people believe that P$\neq$NP which implies that such a solution will not exist. This is why TSP is NP-complete. We shall discuss these concepts in more detail in Section 10.2.

The problem is rather well-known, since many problems in optimization and graph theory can be mapped onto TSP. Finding an optimal route for 10 cities on the same computer used above takes 3 ms, 15 cities takes half an hour and 25 cities takes the lifetime of the universe. So if one has solved the problem for $N = 17$ in 4 days and wants to go to $N = 20$, which takes 22 years, there is not much to be gained by using computers that are a factor 120 faster or even with massive parallelization. Unless there exists an algorithm which solves this problem in polynomial time, one is bound to eventually hit a wall beyond which one cannot solve the problem, even with a lot faster or a lot more computers. This will happen eventually when trying to solve any NP-hard and NP complete problems.

Even worse than that, there are problems which are undecidable. The most famous undecidable problem is the halting problem, which asks to design an algorithm that takes as input any computer program and decides whether or not the program will finish (halt) or run indefinitely. Interestingly, such an algorithm cannot exist, i.e. it is impossible to decide the problem of whether a computer program halts. Another example for an undecidable problem, which is more relevant to string theory, is the solution of a coupled system of non-linear Diophantine equations.

For many formulations of string theory, we need to solve nested problems that are NP-hard and undecidable in general [18,19]. A typical workflow for finding a consistent and physically viable string vacuum involves the following steps:

1. Find a set of integers (fluxes, brane winding numbers, other topological quantities that specify the compactification space) such that tadpole and anomaly cancellation constraints are fulfilled. These lead to a coupled system of non-linear Diophantine equations, whose solution is undecidable.
2. Within the set of mathematically consistent models, identify those with viable physics:
    - Find solutions with a small cosmological constant (NP-complete if solved via flux contributions).
    - Compute the spectrum of massless particles (double exponential running time if solved via Groebner bases).
    - Compute the superpotential (NP-hard) .
    - Find critical points of the scalar potential (NP-hard) and check whether they are minima (co-NP-hard).

So, can machine learning or data science beat complexity theory or even decide undecidable problems? Certainly not, they are just algorithms and as such are subject to the complexity constraints raised above. The reason why they can nevertheless speed up computations and help with studying the string landscape is twofold:

- The algorithm only finds approximate rather than exact solutions to a problem.
- The algorithm might identify patterns or heuristics it can use to solve the problem more efficiently as compared to the solution techniques required to solve a generic instance of the problem. In this sense, ML algorithms act probabilistically.

Thus, when applying ML to tackle problems with exponential running time, the reason why the ML or data science algorithm produces solutions in polynomial running time (which means an exponential speedup) is that they are answering a different question.

Let us illustrate these two points. First, finding an exact minimum of a function is in general in NP, but finding an (arbitrarily) good approximation is in P. This approximation can be good enough since in many cases, the quantities computed in string theory are topological (i.e. integers). If the algorithm approximates a solution, it will return a real number, which can be rounded to the nearest integer. In many cases (for all NP problems), it can be verified that this rounded integer indeed solves the problem. Similarly to complexity theory, one also defines different classes of problems according to how well they can be solved approximately. For example, the above minimization problem allows for a **polynomial time approximation scheme** (PTAS), since a polynomial time algorithm exists to solve the problem up to any given accuracy $\varepsilon > 0$. Similarly, a **polynomial time randomized approximation scheme** (PRAS) produces a solution which is to a high probability (usually 75%, but the exact value often does not matter) within a distance ε of the optimal solution. We will not discuss approximation schemes in this review in more detail. The interested reader can consult e.g. [20] for a textbook introduction.

Second, solving for example a general system of Diophantine equations is undecidable. However, solving a single quadratic Diophantine equation is decidable [21], but already the simple form $ax_1^2+bx_2+c = 0$ is NP-complete [22]. Finally, solving linear Diophantine equations is in P by using the LLL lattice basis reduction algorithm [23]. Thus, it can happen that all the input relevant to the problem we want to solve has a specific pattern or belongs to a certain subclass of NP (or even undecidable) problems, which can be solved (or decided) in polynomial time. Since there is a lot of structure in the mathematics of string theory, and since many physical problems are severely constrained by e.g. consistency conditions or symmetries, the hope is that some patterns emerge and can be picked up by machine learning algorithms to cleverly solve the problem quickly and efficiently, while it might be hard for a human to spot such simplifications. Evidence towards this actually happening is presented in [24].

1.4. Organization of the review

This review consists of two parts. In the first part (Sections 2 to 9), we introduce concepts of data science that are relevant for string theory studies. This introduction will be general and not make reference to string theory concepts, in order to keep them accessible to a wider class of readers that have a physics background, but not necessarily a string theory background. In Section 10, we review the existing machine learning literature and illustrate applications of the techniques explained before to problems that arise in string theory.

In more detail, the review is organized as follows:

- Sections 2 to 4 introduce neural networks (NNs). We start with a simple example network in Section 2.1, which illustrates the different pieces and concepts entering a NN and its training. After that, each section deals with one of these aspects in detail:
  - Section 2 introduces different layers and activation functions of NNs, which form the basic building blocks of NNs.
  - In Section 3, we explain how a NN is trained. Many concepts, such as the generation of training sets as explained in Section 3.2, the definition of different loss functions introduced in Section 3.5, the problem of over- and underfitting examined in Section 3.6, and methods of performance evaluation detailed in Section 3.8, are not specific to NNs and can be applied to other machine learning techniques as well.
  - Section 4 introduces the most common NN architectures and explains for which type of data they can be used. After explaining why NNs are so powerful in Section 4.1, the reviewed architectures include common feed-forward neural network architectures for classification or regression of data in Section 4.2, adaptive resonance theory networks and autoencoders for clustering in Sections 4.3 and 4.6, various convolutional neural networks for image recognition in Section 4.4, Boltzmann machines and deep belief networks in Section 4.5, and Generative adversarial networks (GANs) and variational autoencoders for data generation in Sections 4.7 and 4.8.
- Section 5 describes genetic algorithms. This class of data science algorithms is different from the machine learning and NN algorithms discussed so far. GAs can be utilized to solve complicated extremization problems by making use of the concept of survival of the fittest to evolve solutions to a given problem.

- Section 6 describes persistent homology as an example for topological data analysis. The idea is to define a notion of homology for (discrete) data points and to analyze the structure of data by analyzing their topology.
- Section 7 describes machine learning algorithms other than NNs that can be used in unsupervised machine learning to cluster data or detect outliers and anomalies in a data set. After explaining a general problem that occurs in all these algorithms in Section 7.1, we introduce common algorithms such as principal component analysis in Section 7.3, K-means clustering in Section 7.4, mean shift clustering in Section 7.5, Gaussian expectation–maximization clustering in Section 7.6, clustering with BIRCH in Section 7.7, and with DBSCAN in Section 7.8. In the last part, in Section 7.9, we compare the various clustering algorithms.
- Section 8 introduces reinforcement learning (RL), to search for solutions in a large space of possibilities.
- Section 9 discusses classification and regression algorithms besides NNs that can be used in supervised machine learning. The algorithms discussed are the k-nearest neighbor algorithm in Section 9.1, decision trees and random forests in Section 9.2, and support vector machines (SVMs) in Section 9.3.
- In Section 10, we discuss how all the techniques introduced above have been applied to problems in string theory. We briefly introduce string theory data sets in Section 10.1. In Section 10.2, we review the problems we are facing in string theory and how hard they are from a computational point of view. In Section 10.3, we describe how NNs and clustering plus regression can be used to compute cohomologies of line bundles over Calabi–Yau manifolds. Section 10.4 describes how NNs and SVMs were used to study various topological quantities in the Kreuzer–Skarke and Complete Intersection Calabi–Yau data sets. Section 10.5 presents examples where linear regression, equation learning, and decision trees were used in the context of intelligible AI to generate and prove conjectures based on observations made by the AI in some data set. Section 10.5.3 uses decision trees to predict the types of non-Higgsable gauge groups that appear in F-Theory on toric, elliptically fibered Calabi–Yau fourfolds. Section 10.6 explains how generative adversarial networks can be used to generate superpotentials for $4D$ $\mathcal{N}=1$ theories. Section 10.7 reviews how persistent homology and autoencoders with decision trees were used to study the structure of string vacua. Section 10.8 illustrates how RL can be used to search through the landscape of string vacua to identify viable models. Section 10.9 illustrates the use of genetic algorithms to distinguish high-scale SUSY breaking models and to find string vacua. Section 10.10 illustrates the use of convolutional neural networks for toric diagrams to predict volumes of Sasaki–Einstein manifolds. Section 10.11 introduces the idea of using NNs to approximate the bulk metric in AdS/CFT. Lastly, in Section 10.12, deep Boltzmann machines and their relation to AdS/CFT and Riemann Theta functions are discussed.

1.5. Programs and libraries for machine learning

There are many machine learning libraries that automate the algorithms discussed in this review. They are open source (i.e. free to use). Most of them are written in Python, so if one wants to do machine learning seriously, one should probably learn Python. Some string theorists (if they are using Sage) are already familiar with Python, which should make it easy to get started. Those who are not familiar with and do not want to learn Python might want to look at Mathematica [25], which offers machine learning algorithms such as neural networks, but also clustering and regression.

For all libraries listed below, a wealth of examples and tutorials can be found on the web. I have also made code used in examples of this review publicly available [26]. This should provide a starting point to get familiar with the different libraries and allow for successively creating more advanced applications.

Neural network libraries

There are several neural network libraries available. I have used all of the ones listed below and have not noticed a big difference. So the reader is encouraged to look at them and to see which one they like best. All of the ones listed below are Python libraries.

- PyTorch [27]: A very popular neural network library that offers all the standard ML tools with CPU and GPU support.
- TensorFlow [28]: Another very popular library developed by Google. It provides more functionality beyond just neural networks and comes with Tensorboard, which can be used to visualize the networks. It implements all usual neural network functionality, including CPU and GPU support.
- Keras [29]: This is a high-level API for other machine learning libraries such as TensorFlow or Theano. What this means is that Keras uses its own syntax, which bundles functionality of TensorFlow or Theano to makes it easier to build, train and save neural networks.
- Chainer [30] and ChainerRL [31]: This library offers reinforcement learning support. ChainerRL includes implementations of several reinforcement learning algorithms and uses the Chainer library for its neural network implementation. Of course, Chainer can also be used as a standalone neural network library. It offers an interface to the OpenAI gym [32], which provides a simple way to separate the implementation of the physical search space (e.g. the string landscape) and the computer science implementation of the algorithm.

Other clustering, classification, and regression algorithms

While neural networks are often used to perform clustering, classification, and regression, many other algorithms have been developed for these tasks. Scikit learn [33] is a Python library that collects many of these algorithms. It is very easy to use and I recommend trying it out for machine learning algorithms that are not neural networks.

Genetic algorithms

Genetic algorithms are mostly very simple to program oneself. Of course, there also exist genetic algorithm libraries. Two popular implementations are:

- DEAP [34]: Distributed Evolutionary Algorithms in Python (DEAP) is a Python library for genetic algorithms. It implements most aspects discussed in this review.
- PIKAIA [35,36]: This is a FORTRAN-based library for genetic algorithms, which also implements most standard operations of genetic algorithms.

Persistent homology

There are different persistent homology libraries, which differ in the functionality they provide. For a comprehensive overview see also [37].

- JAVAPLEX [38]: Javaplex is, as the name suggests, a JAVA library for persistent homology. It offers the most functionality of all the libraries, including Witness complexes, a barcode plotter, support for non-Euclidean distances, and many more. It can be integrated into Mathematica, Matlab, Jython, or can be run directly via JAVA.
- RIPSER [39]: Ripser is an open-source C++ program. After compiling, it computes persistent homology given a distance matrix or a point cloud. According to the author, it outperforms many other implementations of Vietoris–Rips computations.
- GUDHI [40]: Gudhi is another library for topological data analysis. It is written in C++, but also comes with a Python interface.

2. Common layers in neural networks

Artificial neural networks, or neural networks (NNs) for short, are modeled after the human brain. Just like synapses connect neurons and forward electrical or chemical signals for processing, information in NNs is processed in connected layers, producing an output from an input. Each layer consists of a set of nodes and the connections are modeled by linear transformations (i.e. matrix multiplications). When the input or signal reaches a node, the node is **activated**, meaning that some **activation function** is applied to the signal. After that, the activated signal is passed on to the next set of nodes that are connected to the current one. While details vary among different NNs, this basic idea and architecture remain the same for all NNs. We will start by illustrating the concept using a simple neural network and then discuss the different ingredients in detail.

2.1. Example: A simple neural network

To illustrate how neural networks work, let us study in a simple example how they approximate the function

$$\begin{aligned} &f : \mathbb{R}^2 \to \mathbb{R}\,, \\ &(x_1, x_2) \mapsto y = \begin{cases} 1 & \text{if } x_1 x_2 < 0 \\ 0 & \text{else} \end{cases}\,. \end{aligned} \tag{1}$$

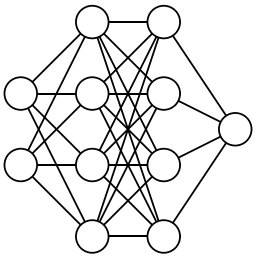
A slight variation of this example was discussed as an illustration in [3], where it is explained that the problem corresponds to finding stable regions of a line bundle in Kähler moduli space of a Complete Intersection Calabi–Yau (which, in turn, corresponds to the absence of global sections of the line bundle). Note that we have simply written down the function we want to approximate in Eq. (1); in usual applications, this function is not known and shall be "found" by the NN based on some input–output pairs which the NN learns from.

For this example, the input is a vector in $\mathbb{R}^2$ and the output is a scalar. We set up the neural network with four layers, see Fig. 2a:

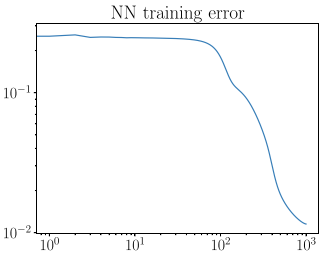
- The first layer is the input layer which consists of two nodes corresponding to the two entries of the input vector $x = (x_1, x_2)^T$.
- The second layer, also called the first hidden layer (it is hidden since its output is never shown to the user but just used internally by the network), consists of four nodes, which are connected to the input nodes by matrix multiplication with a 4×2 matrix $w^{(1)}$ and addition of a constant vector $b^{(1)}$. The activation function $a(\cdot)$ of each node of the second layer is chosen to be the so-called logistic sigmoid function,

$$a(x) = 1/(1 + e^{-x})\,. \tag{2}$$

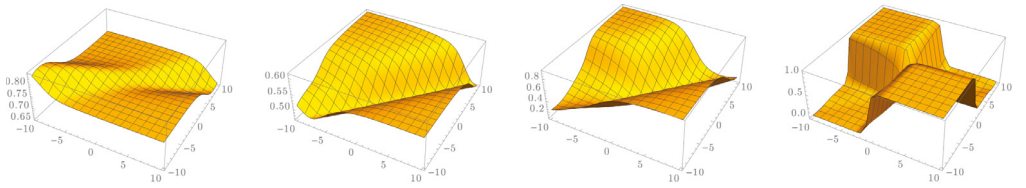
  We could have chosen other functions as well, as discussed in the remainder of this section.
- The third layer, which is again a hidden layer, has the same architecture as the second layer: four nodes and a logistic sigmoid function. The layer connections are modeled by matrix multiplication with a 4×4 matrix $w^{(2)}$ and a vector $b^{(2)}$.



(a) NN architecture.



(b) NN training progress.



(c) NN output after 0, 100, 200, 1000 training steps (note the different y-scales).

Fig. 2. We show the NN architecture, the training progress, and the output of the NN after 0, 100, 200, 1000 training steps.

- The fourth layer is the output layer. It consists of a single node whose value is the output $\hat{y}$ of the neural network. It is connected to the previous layer via matrix multiplication by a 1×4 matrix $w^{(3)}$ and addition of a vector $b^{(3)}$. It is again activated with a logistic sigmoid function, since the result should be in the interval $[0, 1]$.

The entries of the matrices $w^{(i)}$ and the vectors $b^{(i)}$ are not fixed at this point and correspond to parameters of the network, which we collectively denote by θ. For the network to learn, it needs to be trained. Initially, the parameters are set to random numbers, as explained in Section 3.4. During training, the NN is presented with many input–output pairs of the type $((x_1, x_2), y)$, and it adjusts its parameters θ such that its output $\hat{y} = h_\theta(x_1, x_2)$, where h_θ is the function applied by the NN, approximates the target output y as well as possible. The procedure of training NNs is explained in great detail in Section 3. We show the evolution of $\hat{y}$ over many training steps in Fig. 2c. Fig. 2b shows how the error is reducing during training. For this example, we can see from the plot that the NN learned the function in Eq. (1) to great accuracy. We discuss how to evaluate NN performance for more complicated cases in Section 3.8. To give some perspective about the time scale, the full training takes order one second on a normal laptop.

Let us explain in detail how the fully trained network operates, using an example input of $(x_1, x_2) = (1, 2)$. As a first step, this vector is passed on to the four nodes of the second layer by matrix multiplication and addition of a vector. After this linear map, the logistic sigmoid activation function is applied to each component of the vector. The training process has fixed the matrix and vector entries, such that the process reads

$$\begin{pmatrix}1\\2\end{pmatrix} \xrightarrow{w^{(1)}\cdot x+b^{(1)}} \begin{pmatrix}1.196 & -1.422\\ 0. & 0.\\ 3.098 & -0.225\\ 0.22 & -3.104\end{pmatrix}\cdot\begin{pmatrix}1\\2\end{pmatrix}+\begin{pmatrix}-0.065\\-1.932\\-0.117\\-0.13\end{pmatrix} \xrightarrow{a(\cdot)} \begin{pmatrix}0.153\\0.126\\0.926\\0.002\end{pmatrix}.$$

Similarly, the process of the third layer reads

$$\begin{pmatrix}0.153\\0.126\\0.926\\0.002\end{pmatrix} \xrightarrow{w^{(2)}\cdot x+b^{(2)}} \begin{pmatrix}-0.274 & -0.753 & -0.522 & -0.569\\ -1.445 & -0.115 & -2.788 & -3.104\\ 0.082 & 1.921 & 1.516 & 1.138\\ -1.218 & -0.865 & 0.442 & -0.964\end{pmatrix}\cdot\begin{pmatrix}0.153\\0.126\\0.926\\0.002\end{pmatrix}+\begin{pmatrix}0.292\\1.525\\-1.829\\0.071\end{pmatrix}$$
$$\xrightarrow{a(\cdot)} \begin{pmatrix}0.418\\0.214\\0.458\\0.545\end{pmatrix}.$$

Finally, this is mapped to the output layer, which consists of a single node, via the linear map

$$\begin{pmatrix}0.418\\0.214\\0.458\\0.545\end{pmatrix} \xrightarrow{w^{(3)}\cdot x+b^{(3)}} \begin{pmatrix}-1.273 & 3.33 & 3.129 & -1.389\end{pmatrix}\cdot\begin{pmatrix}0.418\\0.214\\0.458\\0.545\end{pmatrix}+\begin{pmatrix}-0.857\end{pmatrix} \xrightarrow{a(\cdot)} \begin{pmatrix}0.000\end{pmatrix}.$$

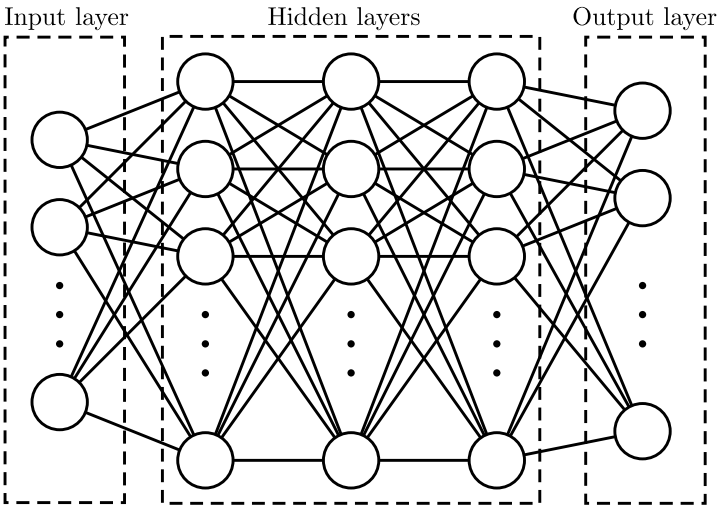


Fig. 3. Architecture overview of a feed-forward neural network.

By combining linear maps with element-wise logistic sigmoid functions, we have approximated the original function. In fact, as we shall see in Section 4.1, one can approximate any (bounded, continuous) function in this way. This is why NNs work so well.

2.2. Feed-forward, fully connected layers and activation functions

Feed-forward neural networks are the simplest implementation of NNs and probably among the most common. A feed-forward neural network consists of a set of layers. Each layer in turn consists of a set of nodes, whose number typically varies between layers. As the name suggests, feed-forward NNs are directed NNs, and information is passed on from the **input layer** through various intermediate **hidden layers** to the **output layers**, cf. Fig. 3. The name of the hidden layers is derived from the fact that the information that passes through them is never output and thus their states are hidden. NNs with many hidden layers are referred to as **deep** NNs, while NNs with many nodes per layer are referred to as **wide** NNs.

The connection between layers is taken to be a linear map and thus simply modeled by matrix multiplication plus possibly addition of a vector. Denoting the output of the ith layer by $\ell^{(i)}$ and the number of nodes in the ith layer by n_i, we thus get for the value $z^{(i)}_\mu$ of the μth node of the ith layer after matrix multiplication

$$z^{(i)}_\mu = \sum_{\nu=1}^{n_{i-1}} w^{(i)}_{\mu\nu} \ell^{(i-1)}_\nu + b^{(i)}_\mu \,, \qquad \text{for } \mu = 1, 2, \ldots, n_i \,. \tag{3}$$

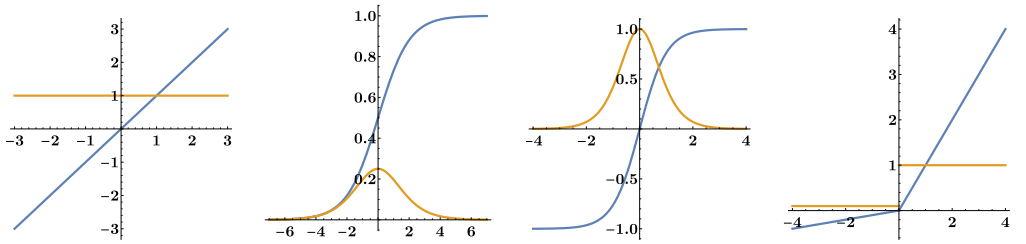
We will use Latin indices to indicate elements of a set and Greek indices to label components of a vector. Notably, in all applications, Greek indices will be raised and lowered with a flat **Euclidean** metric. While this is somewhat unusual, I believe it is still better than using Latin indices for labeling sets and vector components and not use Greek indices at all. We will also often suppress vector indices to talk about the full vector. We will also sometimes use a dot to indicate contraction of Greek indices, or just write out the sum.

The matrix entries $w^{(i)}_{\mu\nu}$ of the $n_i \times n_{i-1}$ matrix $w^{(i)}$ are called **weights** and the vector $b^{(i)}$ is called **bias**. Since a concatenation of linear maps is linear, the non-triviality of a (deep) neural network is derived from applying a non-linear activation function $a(z)$ to each node. With this, the output $\ell^{(i)}_\mu$ of the μth node of the ith layer becomes

$$\ell^{(i)}_\mu = a^{(i)}(z^{(i)}_\mu) \,. \tag{4}$$

Note that the activation function a is the same for all nodes in the layer (hence it only carries an index i and not an index μ). They often only depend only on a single component $z^{(i)}_\mu$ of the vector $z^{(i)}$.

We introduce the following notation. We denote the input to the neural network by x and the output of the neural network by $\hat{y}$. We take the NN to have $s+1$ layers and start counting at zero, so that the input layer is at $i = 0$ and the output layer is at $i = s$. As introduced in (4), we write the output of the ith layer as $\ell^{(i)}$, so that $x = \ell^{(0)}$ and $\hat{y} = \ell^{(s)} = h_\theta(x)$, where h_θ is the map of the NN with parameters θ. These parameters are usually the entries of the matrices $w^{(i)}$ and biases $b^{(i)}$, but sometimes layers that come with additional parameters are used as well. We furthermore denote the activation



(a) Identity (b) Log. Sigmoid (c) Tanh (d) Leaky ReLU

Fig. 4. Overview of some activation functions (blue) and their derivative (orange).

function in the ith layer by $a^{(i)}$. The map h_θ of the NN then maps input vectors (typically in $\mathbb{R}^{n_0}$) to output vectors (typically in $\mathbb{R}^{n_s}$),

$$\begin{aligned} &h_\theta : \mathbb{R}^{n_0} \to \mathbb{R}^{n_s}, \\ &\ell^{(s)}_{\mu_s} = h_\theta(x) \\ &= a^{(s)}\left(\sum_{\mu_{s-1}=1}^{n_s-1} w^{(s)}_{\mu_s\mu_{s-1}} a^{(s-1)}\left(\dots a^{(1)}\left(\sum_{\mu_0=1}^{n_0} w^{(1)}_{\mu_1\mu_0}\ell^{(0)}_{\mu_0} + b^{(1)}_{\mu_1}\right)\dots\right) + b^{(s)}_{\mu_s}\right). \end{aligned} \tag{5}$$

Often in machine learning, the entries of the input vector are called **features** and the corresponding (vector) space $\mathbb{R}^{n_0}$ is called **feature space**. We alternatively also denote the dimension of the feature space by $F = n_0$. The values that we want to compute with the NN (if known) are often called **target** or **ground truth** values, and we denote the dimension of the output space by $T = n_s$.

In order to get some intuition for what the weights and biases do it is useful to consider limiting cases. If the weights are very large (on scales of the input parameter),

$$w \gg b, x, \tag{6}$$

then two input vectors which are close together in feature space are mapped far apart. While this can help to distinguish them, it will also lead to chaotic behavior, since noise is amplified as well. Moreover, when training the NN, we need to compute derivatives. Amplifying small fluctuations can then lead to unstable behavior and exploding gradients. On the other extreme, where the bias is very large,

$$b \gg w, x, \tag{7}$$

the result becomes essentially independent of the input, i.e. the actual value of x is washed away by addition of a large constant. In this case, even input vectors that are far apart can be mapped closely together. This leads to the opposite problem of almost no variation and can lead to vanishing gradients. This implies that the NN will not learn anything during training. Continuity then implies that there is a critical region or critical line in the weight-bias space which is optimal for the NN in the sense that it neither amplifies noise nor washes away the information encoded in the input [41].

Concerning the activation function, any non-linear function of the nodes suffices. Typically, one chooses **unary functions**, i.e. functions that are applied to the signal (or output) $\ell^{(i)}_\mu$ of each node μ separately. The choice depends on various aspects: some activation functions might lead to a better performance of the NN than others and some might facilitate training of the NN. The choice also depends on the purpose of the NN, e.g. whether it is used for regression or classification, as well as on the nature of the input feature space (data, images, ...). Popular choices for unary[1] activation functions are

$$\begin{array}{lll} \text{Identity:} & a : \mathbb{R} \to \mathbb{R}, & a(x) = x, \\ \text{Logistic sigmoid:} & a : \mathbb{R} \to (0,1), & a(x) = \frac{1}{1+e^{-x}}, \\ \text{Tanh:} & a : \mathbb{R} \to (-1,1), & a(x) = \tanh(x), \\ \text{Leaky ReLU:} & a : \mathbb{R} \to \mathbb{R}, & a(x) = \begin{cases} x & x \geq 0 \\ -cx & x < 0 \end{cases}, \\ \text{Softmax:} & a : \mathbb{R}^{n_i} \to (0,1], & a(x_\mu) = \frac{e^{x_\mu}}{\sum_{\nu=1}^{n_i} e^{x_\nu}}. \end{array} \tag{8}$$

Fig. 4 shows the graphs of some of these functions, as well as their derivatives. The latter will become important later when training the NN. Let us discuss the different activation functions in turn.

Identity: Strictly speaking this is not an activation function. Also, it does not introduce non-linearity and thus causes the layer to collapse with the next; if there is only a linear activation function $a^{(i)} = \mathrm{id}$ in layer i, then the two matrix

[1] The softmax function is strictly speaking not unary, since it normalizes the output with respect to the value of the other nodes.

multiplications that map from layer $i-1$ to i and from layer i to $i+1$ can be written as just a single map, removing layer i from the network,

$$\begin{aligned} z_\mu^{(i+1)} &= \sum_{\nu=1}^{n_i} w_{\mu\nu}^{(i+1)} \ell_\nu^{(i)} + b_\mu^{(i+1)} = \sum_{\nu=1}^{n_i} w_{\mu\nu}^{(i+1)} a^{(i)}(z_\nu^{(i)}) + b_\mu^{(i+1)} \\ &= \sum_{\nu=1}^{n_i} \sum_{\rho=1}^{n_{i-1}} w_{\mu\nu}^{(i+1)} (w_{\nu\rho}^{(i)} z_\rho^{(i)} + b_\nu^{(i)}) + b_\mu^{(i+1)} \\ &= \sum_{\nu=1}^{n_i} \sum_{\rho=1}^{n_{i-1}} (w_{\mu\nu}^{(i+1)} w_{\nu\rho}^{(i)}) z_\rho^{(i)} + (w_{\mu\nu}^{(i+1)} b_\nu^{(i)} + b_\mu^{(i+1)}) . \end{aligned} \tag{9}$$

This is why this layer is usually used in the last (output) layer in regression problems only, where the layer cannot collapse with the next layer and the non-trivial computational steps have been carried out by the non-linear combinations of the previous layers. Note, however, that this factorization is not unique, which can lead to interesting consequences, cf. the discussion on implicit regularization in Section 3.6.

Logistic sigmoid: The logistic sigmoid is closely related to the biological activation of a neuron "firing" upon receiving a signal. The logistic sigmoid function also features in maximum likelihood estimation in logistic regression in statistics. It has the advantage that its derivative can be expressed in terms of the function itself,

$$a'(x) = a(x)(1 - a(x)) . \tag{10}$$

As we shall discuss in Section 3, gradients are used in training neural networks in supervised learning, and this feature means that the function value can be reused for the derivative, which leads to less function evaluations and thus faster training. A disadvantage is that logistic sigmoid can get stuck during training. Since this function maps negative inputs close to zero, both the activation and its gradient are close to zero. As a consequence, subsequent nodes might not get activated. Moreover, the parameter updates during training, which are proportional to the derivatives, are also very small, which makes it hard for the NN to leave this unfeasible region in parameter space.

Tanh: While the tanh is similar in shape to the logistic sigmoid function, it maps its input to $(-1, 1)$ with $a(0) = 0$. Just as for the logistic sigmoid, its derivative can be expressed in terms of the original function,

$$a'(x) = (1 - a(x)^2) , \tag{11}$$

which leads to faster training. This function cannot get stuck as easily as the logistic sigmoid function, as only a small region around zero is mapped close to zero, rather than most of the negative axis.

(Leaky) Rectified Linear Unit (ReLU): Taking Eq. (8) (i.e. the leaky ReLU activation function) and setting $c = 0$ gives probably the simplest non-trivial activation function, called the ReLU activation function. Its piecewise linearity makes it fast to evaluate and the derivative is just a constant (the function is not differentiable at zero, but this does not pose a problem). However, this function has a problem similar to the logistic sigmoid function. All negative values and their derivatives are mapped to zero, which means that the ReLU can "die" and not activate or update any following nodes. This is known as the **dying ReLU problem**. To combat this, the **leaky ReLU** activation function was introduced, which "leaks" a bit for negative values, i.e. does not map them all to zero. The leaky coefficient c is usually chosen small, $c \sim \mathcal{O}(0.01)$.

Softmax: Strictly speaking, the softmax function is not a unary function. While it is applied to each node individually, it depends on all nodes in the layer since it normalizes its output such that all values sum to one. This allows for a probability interpretation of all output values. Hence, softmax activation is usually applied to the output layer of NNs that are tasked with classifying input, i.e. assigning an input to one (or more) classes based on its features. In general, such problems are called classification problems. If there are only two classes, the problem is referred to as **binary classification** and if there are more than two classes, the problem is called **multi-class classification** problem. In multi-class classification, one distinguishes **one-vs-all classification problems** (also known as single membership problems), where each input belongs to exactly one output, and **mixed membership problems**, where an input can belong to more than one class. For binary classification, one can also just use a logistic sigmoid (where we label the two classes by 0 and 1) or a tanh function (where we label the two classes by ± 1). As with the other activation functions, the derivative is very simple; it can again be expressed in terms of the original function evaluation,

$$\partial_{x_\mu} a(x_\nu) = a(x_\mu)(\delta_{\mu,\nu} - a(x_\nu)) , \tag{12}$$

which speeds up training.

Besides these activation functions, all kinds of other activation functions are conceivable, such as **binary** functions that e.g. add or multiply the signal of two nodes. These play a big role in **equation learning**.

There is no clear-cut way of telling which activation function to use. Nowadays, (leaky) ReLU activation is a very common choice if one just needs to introduce non-linearity. In fact, its popularity has made ReLU NNs a subject of many recent studies. Mathematically, a NN with ReLU activation functions can be studied using tropical geometry [42], a subfield

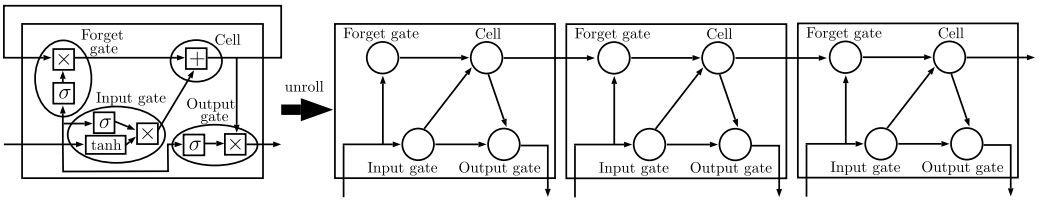


Fig. 5. Recurrent neural network and unfolded feed-forward neural network.

of algebraic geometry in which (tropical) addition of two variables is given by the maximum of the variables and (tropical) multiplication of two variables is given by the addition of the variables.

Usually, a neural network (or an ML algorithm in general) depends on the parameters θ that are optimized during training, as well as on other parameters (e.g. regularization parameters, the number of layers and nodes per layer, . . .). The latter are often fixed at design time and not changed during training. These parameters are referred to as **hyperparameters** in order to distinguish them from the parameters θ that are actually changed during training.

2.3. Recurrent layers and long short-term memory layers

In contrast to feed-forward neural networks, recurrent neural networks have parts of their output fed back as input. In this way, the NN retains information across several inputs. A simple way of picturing recurrent neural networks is by unfolding: the recurrent part of the network is repeated indefinitely, combining its previous state with new input, thus creating an arbitrarily deep feed-forward neural network, see Fig. 5. Several operations of combining the output of the previous iteration with the input of the next iteration are conceivable. In principle, RNNs can retain dependence on previous states for an arbitrarily long time. However, due to finite numerical precision in practical applications, evaluation of derivatives during training can vanish or explode, which leads to problems in retaining information.

A common recurrent neural network layer is the **Long Short-Term Memory** (LSTM) layer. There are several versions of LSTM layers, but a common implementation features four main building blocks:

- the **cell** c retains previous information,
- the **input gate** controls how much new information is stored in the cell,
- the **forget gate** controls how much of the stored information is retained in each iteration, and
- the **output gate** controls how much of the stored information is used in the output computation.

All gates have two inputs: the "data signal" D which contains the actual information and the "control signal" C which controls how open the gate is, i.e. how much of the signal passes through the gate.[2] The control mechanism is usually implemented via a logistic sigmoid function. The control input is mapped to (0, 1), which is then multiplied with the data signal, cf. Fig. 5. In this way, the control signal determines the relative fraction of the input signal that propagates through the gate.

Let us assume that we have an LSTM with input size m and output size n. Then, for each gate, the control signal C at the tth input step consists of the cell state c_{t-1} of the previous step as well as the new input x_t. Each of these is multiplied with their own weight matrices $\widetilde{W}$ and W of size $n \times n$ and $n \times m$, respectively, and then a bias of size n is added, and then they are activated with a logistic sigmoid. For the forget gate, the data signal is the stored information, i.e. the output of the cell from the previous iteration. If the control signal is negative, the sigma-activation function maps it close to zero and most of the information is forgotten, while an activation value close to one retains most information. For the input gate, the data signal is the new input of the LSTM layer, which is multiplied by weights, shifted by biases and then activated (usually with a tanh). The control gate determines how much of the tanh-activated LSTM input is allowed to pass through to the cell, which adds the signal to its current state. It keeps this value for the next iteration and forwards it as data signal to the output gate. The latter than multiplies again this data signal with the result from the logistic sigmoid of the control signal. The result is the output of the LSTM. To summarize, the LSTM performs the following operations

$$\begin{aligned} C_t^{\text{forget}} &= \sigma(W^{\text{forget}} \cdot x_t + \widetilde{W}^{\text{forget}} \cdot c_{t-1} + b^{\text{forget}}), \\ C_t^{\text{in}} &= \sigma(W^{\text{in}} \cdot x_t + \widetilde{W}^{\text{in}} \cdot c_{t-1} + b^{\text{in}}), \\ C_t^{\text{out}} &= \sigma(W^{\text{out}} \cdot x_t + \widetilde{W}^{\text{out}} \cdot c_{t-1} + b^{\text{out}}), \\ D_t &= \tanh(W^{\text{data}} \cdot x_t + \widetilde{W}^{\text{data}} \cdot c_{t-1} + b^{\text{data}}). \end{aligned} \tag{13}$$

[2] This bears some resemblance to transistor gates in hardware: the control signal would be the base-emitter current and the data signal the emitter–collector current.

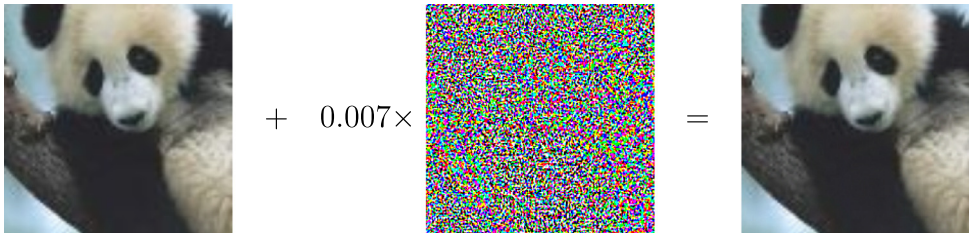


Fig. 6. Illustration of how a CNN can be tricked with adversarial attacks.
Source: Pictures taken from [44].

Altogether, each building block of an LSTM with input size m and output size n will have an $n \times m$ weight matrix for the new input, an $n \times n$ matrix for the cell state, and a bias vector of length n, i.e. there are $4(n^2 + nm + n)$ parameters in an LSTM.

LSTM layers are often used for continuous streams of inputs and outputs, which occur for example in natural language processing: in order to process a sentence or word, it is important to retain information of the previous words or letters encountered in order to suggest the next word or letter, but eventually the information is not needed anymore. However, LSTMs can also be used to **average** over the input. In this case, the LSTM reads in the input vector element-wise. All intermediate outputs are discarded and only the last one, which has "seen" the entire input vector, is kept. This can be useful, if the input is e.g. permutation symmetric. If the next input vector has some semantic relation to the first, the cell state from the previous input vector is kept, otherwise it is erased each time after having read the input.

2.4. Convolutional/deconvolutional layers and pooling layers

Convolutional neural networks (CNNs) feature most prominently in image processing, often with the aim to recognize and identify objects within pictures. Images usually have a lot of pixels and would thus require huge (fully connected) feed-forward neural networks with many parameters. Convolutional layers circumvent this problem as we shall see below. We will only describe common CNN layers in this section; the architecture of CNNs (i.e. how these layers are combined into a NN) is treated in detail in Section 4.4.

In convolutional layers, a D-dimensional block, the so-called **filter** or **kernel** (we will be using both terms interchangeably) scans the image from the top left to the bottom right corner in discrete steps, collects information of all data within the filter, and outputs a number. This is again similar to the biological process of image recognition, where the visual cortex responds to specific regions of the visual input. Since the same (small) filter is used to parse the entire image, one needs far fewer parameters as compared to a fully connected feed-forward NN. However, the number of parameters is still exponential in the input dimension D. It is also customary to use many filters (order 10 to 100) simultaneously. The opposite operation of convolution is called deconvolution or transposed convolution.

A variation of convolutional layers is **pooling layers**. These also have a kernel sliding over the image. However, the kernel has no trainable parameters (weights), but performs a fixed mathematical operation such as taking the average or the maximum value of all values in the region to which the kernel is currently being applied.

While image recognition is probably one of the driving factors for NNs in science and industry applications, image recognition NNs are prone to be susceptible to adversarial attacks. Often it is enough to change one or a few pixels [43], or to add adversarial noise to an image [44] to trick the NN into reassigning the image from the correct class to any other class. From the point of view of a human being, this is particularly surprising, since the image does not necessarily change visibly by such attacks. This is illustrated in Fig. 6 (taken from [44]). The authors take a picture of a panda for which the network is 57.7% sure that it is a panda and add some adversarial noise (the amount of noise added is so small that it is not visually discernible for a human). This causes the CNN to be 99.3 percent confident that the picture is now a gibbon. While this might seem worrisome, humans can also be tricked by optical illusions, which might not trick a NN. So how big a problem these types of attacks are is up for debate, but it illustrates that, despite the intended similarity between pattern recognition in CNNs and the visual cortex, NNs and human recognize images differently.

Convolution layers

For a convolution layer with N_f filters of dimension $(d_1, d_2, \ldots, d_D)$, the number of parameters $N_{\text{param}}^{\text{CNN}}$ is

$$N_{\text{param}}^{\text{CNN}} = N_f(d_1 \cdot d_2 \cdot \ldots \cdot d_D)\,. \tag{14}$$

Some NN libraries only offer implementations for two- or three-dimensional convolutional neural networks, which is natural for application in image processing: two-dimensional convolution layers are used to process black/white images, where each pixel is characterized by a 0 (black) or 1 (white) at its (x, y) coordinates. In order to store color information, one usually uses the RGB method, which is a list of the fraction of red, green, and blue that make up the pixel in additive color mixing at coordinate (x, y), see Fig. 7.

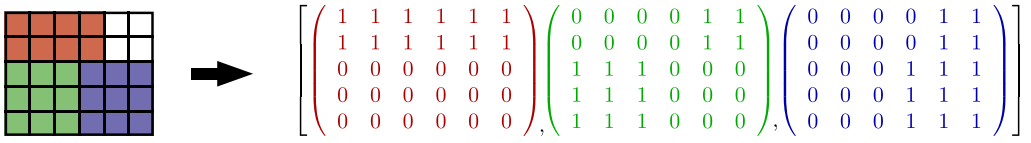


Fig. 7. Example for the encoding of an RGB image with 5 × 6 pixels by three matrices that encode the red, green, and blue channels.

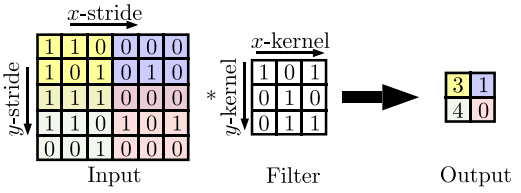


Fig. 8. Overview of a CNN reading a two-dimensional 5 × 6 matrix with x-kernel size 3, y-kernel size 3, x-stride 3, y-stride 2, and no zero-padding.

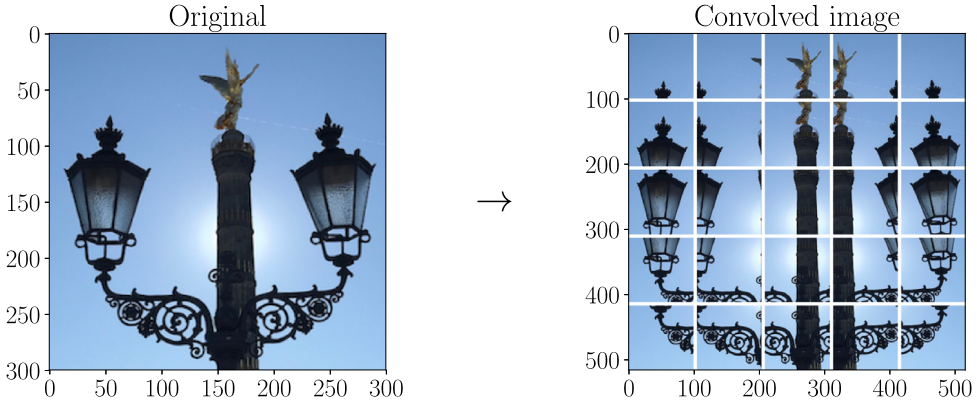


Fig. 9. Convolution operation on an image with 300 × 300 pixels with a kernel of size 100 and a stride of 50. Note how this means that each 100 × 100 block overlaps with (i.e. repeats contents from) the adjacent blocks. E.g. the first row of block 1 contains pixels 0 to 100 from the original's first row, the second block contains pixels 50 to 150, and so on.

For string applications, for example for processing toric diagrams of Calabi–Yau threefolds, (at least) four spatial dimensions are necessary and hence higher-dimensional CNN layers are needed.

Let us explain the actual convolution operation in more detail, see also Fig. 8 for an illustration using dummy data and Fig. 9 for a convolution operation on an actual image. The steps involved are as follows:

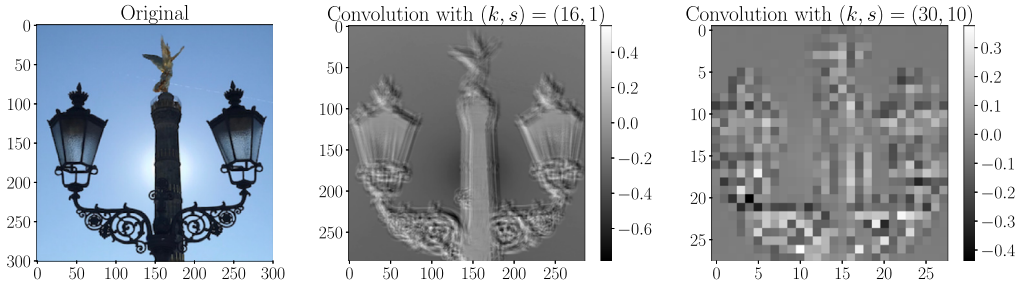
1. First, we create a set of kernels or filters. Their dimensions are called the **kernel sizes**, or sometimes simply **width**, **height**, **depth**,[3] …
2. Next, we define how the kernel moves across the picture. The number of pixels the kernel moves in a given direction is called its **stride**. If e.g. the x-stride is smaller than the filter width, the parts of the picture that are scanned by the filter will overlap.
3. Around the borders of the image, it can happen that the filter does not fully fit onto the picture. In this case, either the last pixels that cannot fit are discarded, or the picture is padded with zeros, i.e. the filter reads only zeros beyond the picture boundary. Depending on the implementation, padding might occur only at one side of the picture or symmetrically all around.
4. Finally the filter multiplies the numerical value of each pixel of the picture with the corresponding entry (or weight) in the filter, i.e. element-wise matrix multiplication, and then sums over all resulting values, thus outputting a single, real number. For example, the first entry in the result of Fig. 8 is obtained via

$$1\cdot 1+1\cdot 0+0\cdot 1+1\cdot 0+0\cdot 1+1\cdot 0+1\cdot 0+1\cdot 1+1\cdot 1=3\,. \tag{15}$$

Following this algorithm, the convolution operation is a map

$$\text{CNN:}\quad \mathbb{R}^{d_1\times d_2\times\cdots\times d_D}\to\left(\mathbb{R}^{o_1\times o_2\times\cdots\times o_D}\right)^{N_f}\,. \tag{16}$$

[3] Here, depth refers to the z-kernel size and should not to be confused with the depth of a NN.



(a) Original. (b) $(k, s) = (16, 1)$. (c) $(k, s) = (30, 10)$.

Fig. 10. Illustration how a convolutional layer processes a picture for different kernel and stride sizes.

For an input of total dimension D and sizes $(d_1, d_2, \ldots, d_D)$, a kernel with kernel sizes $(k_1, k_2, \ldots, k_D)$, strides $(s_1, s_2, \ldots, s_D)$, and $(p_1, p_2, \ldots, p_D)$ padded zeros,[4] the output dimensions $(o_1, o_2, \ldots, o_D)$ are

$$o_i = \frac{d_i - k_i + p_i}{s_i} + 1\,. \tag{17}$$

For the example of Fig. 8, we obtain

$$o_1 = \frac{6 - 3 + 0}{3} + 1 = 2 \qquad o_2 = \frac{5 - 3 + 0}{2} + 1 = 2\,. \tag{18}$$

In image processing, one usually chooses $k_D = d_D$ and $s_D = 1, p_D = 0$, i.e. the filter fills out the entire color space and convolves all color information.

Let us see how a convolutional layer sees a picture (cf. Fig. 10). We show two different convolutions. Both use one filter and no padding. In the first example, we choose a filter size of (16, 16, 3) and a stride of (1, 1), and for the second example we use a filter size of (30, 30, 3) and a stride of (10, 10). The original image has 300 × 300 pixels, with three color channels (RGB). Since the z-kernel size is 3, we convolve all color channels. Using Eq. (17), we end up with an array of size 285 × 285 × 1 and 28 × 28 × 1, respectively. In order to visualize the original image in Fig. 10a, we just plotted the RGB values, which are in the intervals [0, 1] × [0, 1] × [0, 1]. After the convolution, we only have one channel rather than three (we only use one filter). Moreover, the result of a convolution operation can be negative or larger than one. We thus visualize the output with a grayscale color map that assigns different shades of gray (from black to white) to each value. As we can see, the first filter in Fig. 10b produces a somewhat blurry image. The blurriness is due to the fact that every pixel is read and used multiple times (up to 16 times) by the filter. For larger kernels and pixels, the shapes become much coarser and a lot of the fine details are lost, cf. Fig. 10c. Note that after applying a ReLU activation function, those shades of gray that are darker than the shade corresponding to 0 would all be set to black. It has been found [45] that trained CNNs with ReLU activation perform edge detection. For nice illustrations of this see [45].

Pooling layers

Often, convolution layers are followed by **pooling layers**. Pooling layers further reduce the size of the input, which leads to fewer parameters in the successive layers. This helps with reducing training time and overfitting, cf. Section 3. Usually **max pooling** is used, but other pooling operations such as **average pooling** or $\boldsymbol{L_p}$**-norm pooling** can also be found. Pooling layers work similar to convolutional layers. However, instead of performing element-wise multiplication and subsequent addition, a max pooling layer just takes the maximum of all elements in the filter. The output dimensions are as in (17). However, since the filter has no parameters, one only uses one filter, $N_f = 1$. Since max pooling can be thought of as down-sampling, kernel and stride sizes should not be chosen too large in order to not lose too much information. Common parameter choices are $(k_i = 2, s_i = 2)$ or $(k_i = 3, s_i = 2)$.

We illustrate the result of pooling with the same image used before, see Fig. 11. We use kernel sizes and strides of (2, 2) and (6, 2), respectively. This time, we perform max pooling for each channel separately, such that we end up with arrays of size 150 × 150 × 3 and 73 × 73 × 3, respectively. Since we just take the maximum value over all pixels in the kernel for each RGB channel separately, the result is again an RGB image. For the small kernel sizes, the image has half the size with not too much loss in image quality. In contrast, the max pooling operation with kernel size (6, 6) is much coarser. Note especially how we lost essentially all details of the black structure in the bottom of the image and how e.g. the white condensation trail got enhanced. This happens because the lighter a color, the larger the pixel value in RGB encoding.

[4] Depending on the implementation, $p_1 = 1$ might add a zero on the left and right, or just on the right. In the former case, we need to replace p_i with $2p_i$ in (17).

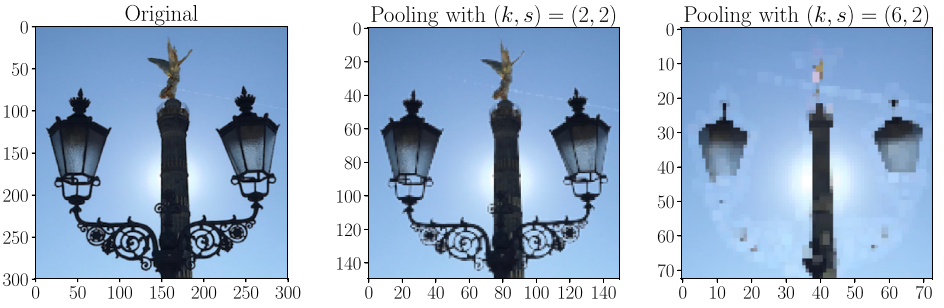


Fig. 11. Illustration how a pooling layer processes a picture for different kernel and stride sizes.

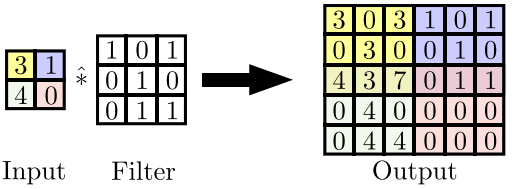


Fig. 12. Overview of a deconvolutional layer reading a two-dimensional 2×2 matrix with a filter of x-kernel size 3, y-kernel size 3, x-stride 3, y-stride 2, and no zero-padding.

Deconvolutional layers

The opposite operation[5] of a convolutional layer is called **deconvolution layer** or **transposed convolution layer**. The deconvolution is performed as follows:

- Each weight of the filter is multiplied with the first number from the input data and the result is stored as intermediate output of dimension $(k_1, k_2, \ldots, k_D)$.
- Then, each weight of the filter is multiplied with the next input number, generating another intermediate output of dimension $(k_1, k_2, \ldots, k_D)$.
- The result from the second operation is offset by $(s_1, s_2, \ldots, s_D)$ and the numbers are added element-wise.
- This process is repeated until the last input data has been read, multiplied with the filter weights and the result added to the intermediate output.

An example is given in Fig. 12. In this example, the x-stride equals the x-kernel size, so different elements from the intermediate results are not added in the x-direction. In contrast, the y-stride is two, so the intermediate deconvolution results overlap in the third output row. Denoting the input values by $d_{i,j}$, the output values by $r_{i,j}$, and the filter values by $w_{i,j}$, we find for example:

$$\begin{aligned} r_{3,1} &= d_{1,1} \cdot w_{3,1} + d_{2,1} \cdot w_{1,1} = 3 \cdot 0 + 4 \cdot 1 = 4\,, \\ r_{3,2} &= d_{1,1} \cdot w_{3,2} + d_{2,1} \cdot w_{1,2} = 3 \cdot 1 + 4 \cdot 0 = 3\,, \\ r_{3,3} &= d_{1,1} \cdot w_{3,3} + d_{2,1} \cdot w_{1,3} = 3 \cdot 1 + 4 \cdot 1 = 7\,, \ldots \end{aligned} \tag{19}$$

Note that in this section, we have first convolved some input in Fig. 8 and then deconvolved the result in Fig. 12. Since convolution is not bijective, there is no unique inverse map, and the input before convolution differs from the result of the deconvolution of the convolved input. Nevertheless, the operations are set up such that the overall dimensions match.

Unpooling layers

Sometimes unpooling is used as well. In contrast to deconvolution, which is a stand-alone operation, unpooling can only be performed if pooling had been applied beforehand. During pooling, the position of the maxima (for max pooling) are recorded in each pooling region [45] using so-called **switches**. In unpooling, each pooling region is padded with zeros and the value of the input to the unpooling filter is inserted at its original position as recorded by the switches.

[5] Of course, the convolution operation cannot be inverted.

2.5. Summary

Neural networks can be understood as concatenation of linear maps with non-linear activation functions. The resulting map depends on the parameters of the linear maps plus possibly other parameters that enter the activation functions. There are several unary, parameter-free activation functions that are commonly used. They all have the advantage that their derivative can be easily computed in terms of the function value itself, which speeds up training.

In feed-forward NNs, information is passed on from one layer to the next. For some applications, where the previous inputs are important, recurrent NN layers can be used. For images, one usually applies convolutional neural networks.

3. Training and evaluating neural networks

Training a neural network describes the process of updating the weights, biases (and potentially other) parameters of a neural network, which we jointly denote by θ. Training requires defining a goal which shall be reached and then modifying the parameters θ until the result comes reasonably close. For example, in the context of **supervised machine learning**, the neural network is given a set of N input vectors x, each of dimension n_0, and their corresponding n_s-dimensional target output vectors y. During training, the network adapts the parameters such that its output $\hat{y} = \ell^{(l_s)} = h_\theta(x)$ comes close to the ground truth values y. This adaptation is usually done using **gradient descent**, see Section 3.3. The **loss** or **cost** function L of the NN is given by the deviation of the ground truth y from the result $\hat{y}$ obtained by the NN. Depending on the application of the NN, there are different metrics to measure this deviation, which we discuss in Section 3.5. Initialization of the NN parameters is discussed in Section 3.4.

3.1. Basics of training NN

Training NNs requires reducing the loss function, which requires optimizing a non-linear function in a very high-dimensional parameter space. Note that updating a single weight $w_{\mu\nu}^{(i+1)}$ will change the value at one node $\ell_\mu^{(i+1)}$ in the next layer. Due to the fact that layers are usually fully connected in NNs, this will change the value at all subsequent nodes $\ell^{(k)}$ with $k > i+1$ and hence also influence all subsequent weights.

The task of optimizing a multi-dimensional parameter space is notoriously difficult due to the exponential growth of the problem. It is only thanks to advances in mathematical optimization that it is nowadays feasible to train NNs with thousands or millions of parameters. Of course, physicists are very familiar with the problem of finding a minimum in a complicated, high-dimensional energy landscape from e.g. the scalar potential in supersymmetric theories or sigma models.

Training, test, and validation data sets

During training in supervised ML, the NN is provided with input–output pairs. We discuss how to choose these pairs in more detail in Section 3.2. Eventually, one wants to apply the NN to data for which the output is not known (otherwise we could just use a database of all the data we have). In order to be able to evaluate how well the NN performs on unseen data, it is important to withhold some of the pairs during training and use them to monitor the training progress as well as to benchmark the final performance of the NN. To this end, the data is split into a **training set** and a **test set**. The former is used during training, the latter during evaluation. How the train:test split is performed depends on the amount of data available, but a common split is in the ballpark of

$$\text{train} \ : \ \text{test} = 90 : 10 \,. \tag{20}$$

Sometimes, a **validation set** is used on top of the train and test sets, with a common split then being

$$\text{train} \ : \ \text{validate} \ : \ \text{test} = 80 : 10 : 10 \,. \tag{21}$$

The idea behind the introduction of a validation set on top of the test set is that one might want another set which the NN does not see during training in order to find the best hyperparameters for the network and to decide when to stop training. The NN is trained with the training set, which contains the bulk of the data, designed with the validation set, and evaluated with the test set. The NN's performance should be comparable to the training and validation set, otherwise parameters or hyperparameters of the NN have been fine-tuned (overfitted) to the train/validation set combination.

Once training and evaluation are completed and the NN passes the evaluation threshold to be used in an environment with new data, the NN can be trained with all available data. As a general rule of thumb, the more data is used during training time, the better the NN will eventually perform. However, since one lacks an independent measure of performance during this final training step, one has to be careful to not overtrain or undertrain the network.

Please cite this article as: F. Ruehle, Data science applications to string theory, Physics Reports (2019), https://doi.org/10.1016/j.physrep.2019.09.005.

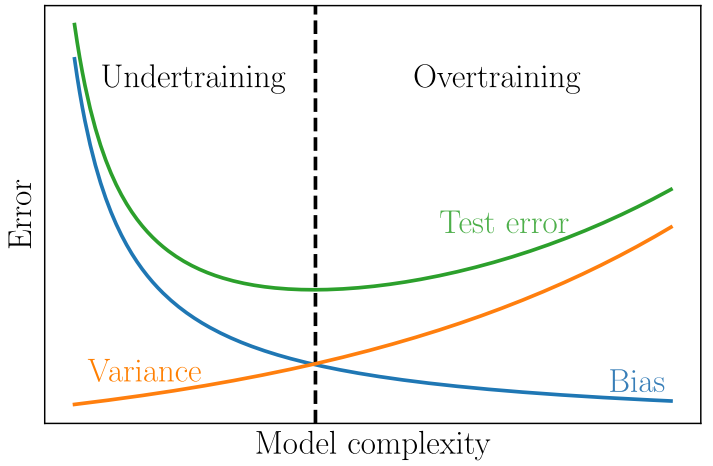


Fig. 13. The total (squared) error on unseen test data is given by the sum of the variance and the bias.

The bias–variance problem

The problem of (over- or under-)fitting a set of training data for future predictions is known as the **bias–variance problem** in supervised machine learning. Since we are trying to make a future prediction based on previous input, it is useful to cast the problem into the language of statistics. In statistics, the **bias** of an estimator is the difference between the expected value $\mathbb{E}[\cdot]$ of that estimator and the actual value, while the **variance** is the expectation of the squared deviation from the mean. Denoting the function we are learning to approximate by $f(x)$ and its approximation by $\hat{f}(x)$, we have

$$\text{Bias}[\hat{f}(x)] = \mathbb{E}[\hat{f}(x)] - f(x)\,, \qquad \text{Var}[\hat{f}(x)] = \mathbb{E}[\hat{f}(x)^2] - \mathbb{E}[\hat{f}(x)]^2\,, \tag{22}$$

or, in physics' Bra-Ket-notation,

$$\text{Bias}[\hat{f}(x)] = \langle\hat{f}(x)\rangle - f(x)\,, \qquad \text{Var}[\hat{f}(x)] = \langle\hat{f}(x)^2\rangle - \langle\hat{f}(x)\rangle^2\,. \tag{23}$$

With this, we can write the squared expected error for an unseen data point x as

$$\langle(f(x) - \hat{f}(x))^2\rangle = \left(\text{Bias}[\hat{f}(x)]\right)^2 + \text{Var}[\hat{f}(x)]\,. \tag{24}$$

Thus, to reduce the error we can reduce the bias or the variance. Overfitted models will have a low bias (they will predict the training data very well), but a high variance (they will oscillate a lot to interpolate between the training data points). Underfitted models will not be complex enough to model the function underlying the input data, so their bias will be large, but they also do not fluctuate a lot to fit the data, so their variance will be small. This is illustrated in Fig. 13. An example for overtraining is given in Fig. 17 of Section 3.6, where we discuss overtraining in more detail.

One way to reduce variance is to train in parallel several models and have them "vote" on the final outcome, i.e. one averages over the output of different models. This is known as **bagging**. A way to reduce the bias is to train several models consecutively, where the next in the sequence is trained on improving the worst results of its predecessor. This is known as **boosting**. Since both methods use several predictors, they are known as **ensemble methods**.

The training process

Often, data sets addressed with NN techniques are huge and might not fit completely into the memory. Also, some training methods update the weights after all training data has been seen, while others perform intermediate updates that use only parts of the training data. In more detail:

- A training **epoch** comprises processing the entire training set once. The number of training epochs depends on the problem at hand, but a single epoch is usually not enough. Often, the training set is shuffled after each epoch. However, in some cases it can be beneficial to order the training set from "easy" to "hard", such that the NN first learns the general features of the data set and then the more specialized cases. This requires of course a metric for "hardness" and a good knowledge of the underlying data.
- A **mini-batch** or sometimes just **batch** contains a subset of the training set. While splitting the training set into mini-batches hinders parallelizing the training, the split usually provides better performance and the fact that less memory is used can increase the throughput, see e.g. [46]. Often, one uses rather small batches (with batch size N_B of 2 to 128). In most applications, the mini-batches are created anew from the (reshuffled) training set after each epoch.

- An **iteration** is the act of processing one mini-batch. Hence, denoting the cardinality of the entire training set by N_T, the number of iterations in an epoch is N_T/N_B.

It is a priori hard to tell how many training epochs will be necessary. Instead of specifying this beforehand, training progress is usually monitored and stopped after a certain accuracy has been reached, after the NN does not improve any further, or after a maximum amount of time has elapsed. We will discuss how to monitor the training process and when to stop training in Section 3.6, and how to evaluate the performance of the NN in Section 3.8.

Summary

In training, the NN adapts its parameters to reproduce some given ground truth values y from its inputs x. The input–output pairs are iterated several times in epochs, which are broken down into smaller batches. The approximation error can be expressed in terms of the bias and the variance; just minimizing the former leads to unstable NNs whose predictions fluctuate a lot, while minimizing the latter leads to NNs that do not capture the complexity of the data set.

3.2. Generating the training set

The training set is very important to obtain good results in supervised machine learning. As a general rule of thumb, the more data is available, the better the NN will perform after training. This is why NNs are mainly used in big data applications. Nevertheless, it might be necessary to use more training data than actually available, in which case we need to use techniques from **data augmentation**. A second important question is which features are included in the training set. Sometimes, features can be derived from given features or additional information can be added to the input to provide more data, a process known as **feature engineering**. In other cases, it might be beneficial to perform **feature reduction** instead and select only the independent or most relevant features, which can facilitate and speed up training.

Data augmentation

There are different augmentation techniques, depending on the underlying data. Augmenting the training set before training is called **offline augmentation** while augmenting the training set during training (usually performed for each mini-batch) is called **online augmentation**. The former has the advantage that augmented results are stored and hence need not be generated several times. However, it inflates the number of data and there might simply not be enough storage capacity if the training set was already large before augmentation.

The simplest way to increase the amount of data is to use **cloning**, which means that (some) input–output pairs are just duplicated and included several times in the training set. Of course, cloning the entire data set is equivalent to running more training epochs. So, cloning can be useful if the data set is very unbalanced. If we want to perform a binary classification task and one class occurs much more frequent than the other, the NN might learn to ignore the less frequent one and to always predict the more frequent class as its outcome. By cloning elements from the less frequently occurring class, we can obtain a more balanced data set. This approach is called **oversampling**. A complementary approach is **undersampling**, where elements from the larger class are randomly removed to create a more balanced data set.

An oversampling technique that is more sophisticated than simply cloning elements is the so-called **Synthetic Minority Oversampling Technique** (SMOTE) [47]. The idea is to create new, synthetic examples for the minority class. This is done by linearly extrapolating feature vectors of two instances of the minority class. In more detail:

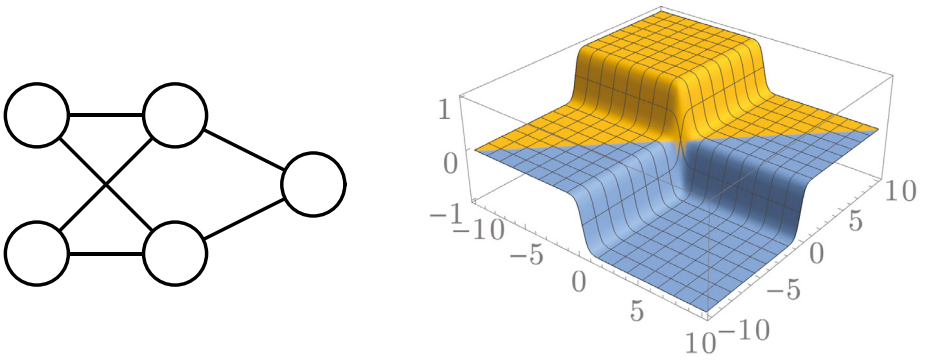
1. Find the k nearest neighbors x_j, $j = 1, \ldots, k$ of a given input x_i that belongs to the minority class that shall be upsampled. An algorithm for finding the k nearest neighbors (and performing classification based on the neighbors) is discussed in Section 9.1.
2. Calculate the distance vector $d = x_j - x_i$ between the point x_i and one of its nearest neighbors.
3. Create a new input with a new, synthetic feature vector $x_{\text{new}} = x_i + \lambda d$, $\lambda \in [0, 1]$.

This synthetic input, if it existed, would be expected to belong to the same (minority) class as its neighbors. Hence the extra input is expected to not throw off the NN but instead teach it more about the minority class, which should help it find good heuristics to distinguish features that lead to different classes. A similar technique can also be applied to regression input data.

In the particular case in which input and output are integers, as is often the case in string theory, one can augment the data by adding some noise (e.g. Gaussian, uniform) to the integer input features. The idea is that this noise, if small enough, should not change the integer output of the NN. Moreover, addition of noise can help prevent overtraining and lead to better results, cf. Section 3.6.

Another technique for oversampling is to use symmetries of the input data to generate more, equivalent data. If the input data is an image, one can perform operations like scaling, horizontal/vertical flipping, cropping, or rotating the image. If the input is e.g. a permutation-invariant list of data, one can generate permutations of this list. Often, physical systems come with a wealth of symmetries that can be applied to augment the data. This is useful since physicists are often interested in classifying models up to symmetry, so the network can and will most likely be confronted with many equivalent data, and it should learn the different representations.

However, the wealth of symmetries can also be a curse, since it requires the NN to learn the symmetry in order to make correct predictions. This requires in general much more complex NNs. Let us illustrate this using the example of



(a) NN Architecture. (b) NN output.

Fig. 14. Illustration of a simpler NN architecture (as compared to the one presented in Fig. 2), which works correctly if the permutation symmetry is taken into account (orange region).

Section 2.1. The problem is symmetric under the exchange of the two input parameters. If we could have assumed in this example that the permutation symmetry is accounted for outside the NN computation and the NN is simply only presented with inputs such that $x_1 \leq x_2$, we would have had to consider only one half-plane. In that case, a simpler NN suffices: all we need to do is to form the linear combination

$$\hat{y} = \sigma(wx_2) - \sigma(wx_1) \tag{25}$$

with some large enough $w > 1$ that turns the sigma function into an approximate step function: if x_2 is negative, so is x_1. The sigma function will map both close to zero and the difference $\hat{y}$ in Eq. (25) will be essentially zero. Similarly, if x_1 is positive, so is x_2, the sigma function will map both close to one and their difference is again close to zero. Lastly, if x_1 is negative and x_2 is positive, $\sigma(wx_2)$ is close to one and $\sigma(wx_1)$ is close to zero, so that the result is close to one, as desired. Hence, we need a simple NN with one hidden layer and logistic sigmoid activation. The weight matrices $w^{(i)}$ are simply

$$w^{(1)} = w\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}, \qquad w^{(2)} = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}, \tag{26}$$

and the biases are zero. We depict the NN architecture and its output in Fig. 14. The orange region is the region where $x_1 \leq x_2$ and the NN produces the desired output, while the blue region has $x_1 > x_2$, for which the NN obviously fails to produce the correct output. This is why we need the more complicated setup given in Fig. 2 if we do not take the permutation symmetry into account. While the increase in complexity of the required NN seems not too bad, it can become very substantial, e.g. when using rotation-symmetric images in many dimensions.

The problem can be circumvented by engineering features that are invariant under a given symmetry. This can be done by defining **invariants** or **normal forms** of physical systems, which are constructed such that the redundancies are removed. A very simple example of an invariant under base change is to characterize a square matrix by its eigenvalues (or Jordan blocks), since often the choice of coordinate system is irrelevant. A more advanced example for a normal form is the normal form that takes care of toric morphisms of reflexive polytopes in toric geometry [48,49]. In this sense, normal forms are an orthogonal approach to data augmentation; instead of training the network with data that falls into the same class under a symmetry operation, we provide input that is invariant under a given symmetry to begin with and hence we cannot (and need not) augment the data anymore.

In some cases, it might also make sense to provide the normal form together with other (possibly augmented) data. This can be necessary if the normal form does not remove all symmetries. For example, even after computing the Jordan blocks or eigenvalues, a permutation symmetry of the eigenvalues might remain, depending on the application.

Feature engineering and feature reduction

Feature engineering is the process of creating features for the NN inputs. How this is done depends of course on the data under investigation. The idea is to facilitate learning for the NN by providing as many (independent) features as possible, or to apply some transformation to the features beforehand which is likely to be relevant for the solution, such that the NN does not have to learn this combination.

In order to generate (new) features, one needs to understand the properties of the underlying data. In the case of matrices, one might for example provide the eigenvalues, the rank, the determinant, etc. These features are of course not independent, since e.g. the determinant is the product of the eigenvalues. Nevertheless, providing both might be useful. If the NN does not include a multiplication layer (and standard NNs with the usual unary activation functions do not), the NN cannot directly compute the determinant by itself (but it can learn to approximate it). In contrast to the determinant,

the sum of eigenvalues can be easily computed by the NN by setting all weights that connect all nodes of layer $\ell^{(i)}$ to a node μ in layer $\ell^{(i+1)}$ to one,

$$w_{\mu\nu}^{(i+1)} = 1 \qquad \forall \nu = 1, \ldots, n_i \,. \tag{27}$$

In order to perform feature engineering or feature reduction, the best scenario is to use some a priori knowledge of which feature combination will enter the function f one is trying to approximate with the NN, engineer these features by hand, and feed these to the NN input layer. In other words, if the function f is a functional

$$f(X) = f[g_1(X), \ldots, g_r(X)] \,, \tag{28}$$

where each $g_i(X)$ is a certain function or transformation of the original features[6] X_μ, one can compute the $g_i(X)$ beforehand and then approximate f using $g_i(X)$ rather than X. In case $r < n_0$, we will use fewer (transformed) features than the ones present in the original feature vector X. These cases correspond to feature reduction.

However, finding the right g_i requires knowledge of the final function, which is often not given. As a milder form, if one suspects the functional dependence of the output on (a set of) the features, one can perform **boosting**. This means that a guess $\hat{y}^{\text{guess}}$ for the final output is computed and this output is used as one input feature for the neural network (together with other features), i.e. the NN is boosted with some approximate computation. The NNs task is then to learn the corrections to this guessed function, i.e. to find $\hat{y}$ such that $y = \hat{y} + \hat{y}^{\text{guess}}$. We will discuss several feature reduction techniques, e.g. autoencoders (Section 4.6), principal component analysis (Section 7.3) or decision trees with and without boosting (Section 9.2) in later sections.

Here, we only want to introduce a very simple way of identifying independent features via their **correlation matrix**. The correlation matrix is a symmetric $F \times F$ matrix, where F is the total number of features. Its entries (μ, ν) correspond to the correlations between feature X_μ and feature X_ν of feature vector X. This two-point function only serves to identify correlation between two features. The correlation between features (X_μ, X_ν) is defined as

$$\text{corr}(X_\mu, X_\nu) = \frac{\langle (X_\mu - \langle X_\mu \rangle)(X_\nu - \langle X_\nu \rangle) \rangle}{\sigma_{X_\mu} \sigma_{X_\nu}} \,, \tag{29}$$

where σ_X is the standard deviation and $\langle \cdot \rangle$ is the expected value. Since the training set is a finite set of finite-dimensional vectors X, each feature will only take a discrete number of values a finite number of times, such that the expected value or mean simply becomes

$$\langle X \rangle = \sum_{q=1}^{N_X} p_q x_q \,, \tag{30}$$

where x_q are the values that X can take, and p_q is the probability with which X takes this value in the underlying training set. The correlation is valued in the interval $[-1, 1]$, with 0 indicating completely uncorrelated features, 1 indicating complete correlation, and -1 indicating complete anti-correlation. In particular, the diagonal entries of the correlation matrix are 1.

Summary

It is important to create well-balanced training data sets. Balancing can be achieved by over- or undersampling. The former generates new data from existing training samples, while the latter disregards some of the input data. A second important question is which features to use for training. Too few features might not allow the NN to reconstruct the output, while too many or irrelevant features might lead to inefficient training. Methods like autoencoders, principal component analysis, decision trees, and correlation matrices can be used to identify the most important features.

3.3. Gradient descent

The most commonly employed method in training NNs is **gradient descent** (GD). As the name suggests, a minimum for the loss function is found by following the direction of steepest descent, i.e. along $\nabla_\theta J$, where J is the **total loss function**. However, also other techniques can be used to solve the optimization problem of finding the best combination of parameter values of the NN, such as genetic algorithms (see Section 5).

Gradient descent will stop once a critical point is reached, since then the gradient vanishes. Due to the complicated landscape of the loss function, there are potentially many saddle points. This makes it very hard to find minima or maxima [50] (even local ones) using any technique, not just gradient descent. It is furthermore very hard to verify that a generic extremum is a minimum, see for example [19] for a discussion of this in the physics literature.

Why the verification that an extremum is a minimum is hard can be understood as follows: if the Hessian is just positive definite, the critical point is a minimum and the problem is actually not hard. It only becomes hard (in the

[6] We will denote the feature (input) vector with a capital X in this paragraph, since we want to denote the values that each feature of the training set can take by x_μ; this is also the standard notation in statistics for a random variable.

sense of computational complexity, see also Section 10.2 for more details) if there are saddle points. If the Hessian has both positive and negative eigenvalues, the critical point is called a **strict saddle point**. Since the gradient vanishes, the minimization algorithm can get stuck there and not continue its descent towards a minimum. However, the chance of actually getting stuck is negligible [51]. The worst case is when the Hessian is positive semi-definite, i.e. all eigenvalues are non-negative. In that case, the critical point could be a minimum or a saddle point, but we do not know which one it is. We thus need more information, e.g. from higher derivatives, to decide the nature of the critical point. This is what makes the problem hard in theory. In practice, this might not be as bad as it seems, since at least for polynomials, extrema with positive semi-definite Hessians are a measure zero set [52].

3.3.1. Backpropagation

Training of neural networks proceeds in two steps known as the **forward pass** and the **backward pass** or **backpropagation**. See [53] for a very early application of backpropagation to NN training, [54,55] for successive work, and [56] for more references. We use the notation of Section 2.2 in the following discussion. Also note that, before we can even begin the training, we need to initialize the neural network. We will assume that the NN has been initialized and discuss details of initialization in the next Section 3.4.

In the forward pass, the input x is first fed into the input layer $\ell^{(0)}$ of the NN. Then the data is propagated forward by matrix multiplication, adding the bias, and applying the activation functions a_i until all values $\ell^{(i)}$, $i = 0, 1, \ldots, s$ have been computed and stored. In supervised learning, we have input–output pairs (x, y), so we can compare how the output y differs from $\hat{y} = \ell^{(s)}$ for a given input $x = \ell^{(0)}$. This difference J is computed with regard to a cost function L plus a **regularizer** or **weight decay** term r,

$$J(\theta, x, y) = L(\theta, x, y) + \lambda r(\theta)\,. \tag{31}$$

We discuss different cost functions in Section 3.5. The parameter λ is called the **weight decay parameter**. The choice for the value of λ depends again on the problem at hand, but typically $\lambda \sim \mathcal{O}(10^{-3})$. There are also various choices for the regularizer or weight decay functions $r(\theta)$, as long as they have the property that they lead to a weight decay, i.e. favor small weights. As the name weight decay function indicates, these functions only regularize the weights but not the other parameters (such as the biases) of the NN. The advantage of including weight decay is that small weights tend to prevent overfitting, see Section 3.6 for more details.

In one iteration of gradient descent, the parameters of the NN get updated. Denoting the parameters of the ith layer collectively by $\theta^{(i)}$, the update is

$$\theta_a^{(i)} \to \theta_a^{(i)} - \alpha \frac{\partial J(\theta, x, y)}{\partial \theta_a^{(i)}}\,. \tag{32}$$

The extent to which the NN follows the direction of the (regularized) steepest gradient is called the **learning rate** α. Of course, α is an important parameter in training. If α is chosen too small, the NN takes a very long time to converge. If α is chosen too large, one might overshoot the minimum and descend (or even ascend) into the wrong direction. Some training methods also adapt the learning rate over training time.

In the update process, the deviation $\delta^{(s)}$ between the desired output y and the actual output $\ell^{(s)} = h_\theta(x)$ is computed and then backpropagated through the network to recursively obtain the other deviations $\delta^{(i)}$ and corresponding updates to $\theta^{(i)}$ for the other layers. In more detail, we use the successive mappings (5) and the chain rule, to get

$$\frac{\partial J}{\partial \theta_a^{(i)}} = \frac{\partial L(\theta, x, y)}{\partial z^{(i+1)}} \frac{\partial z^{(i+1)}}{\partial \theta_a^{(i)}} + \lambda \frac{\partial r(\theta)}{\partial \theta_a^{(i)}}\,. \tag{33}$$

Remember that z is the vector obtained from applying the linear map between layers but before applying the activation function, cf. (3). The second term and the second factor of the first term are easily computed. The first factor of the first term, which is essentially the derivatives of the activation function, gives the deviations $\delta^{(i)}$ of the other hidden layers. These are used to backpropagate $\delta^{(s)}$ through all intermediate layers down to $\delta^{(0)}$, as explained in Eq. (37). As mentioned in Section 2.2, for many popular activation functions, the derivative is either very simple (e.g. for identity, leaky ReLU) or proportional to the function itself (e.g. for logistic sigmoid, tanh). Since the activation function has been evaluated and the value stored in the forward pass, the result can be reused in the backward pass. Having obtained all deviations $\delta^{(i)}$, we can finally compute the gradient $\nabla_\theta J$ and update the parameters.

To illustrate the procedure, let us focus for concreteness on the simple but frequently encountered case where the NN parameters are just weights and biases, $\theta_a^{(i)} = \{w_{\mu\nu}^{(i)}, b_\mu^{(i)}\}$. Furthermore, we use the mean square error function (64) for the loss L and regulate only the weights but not the biases,

$$\begin{aligned} L(w, b, x, y) &= \frac{1}{n_s} \sum_{\mu=1}^{n_s} (y_\mu - \ell_\mu^{(s)})^2\,, \\ r(w) &= \frac{1}{2} \sum_{i=0}^{s-1} \sum_{\mu_{i+1}=1}^{n_{i+1}} \sum_{\mu_i=1}^{n_i} \left(w_{\mu_{i+1}\mu_i}^{(i)}\right)^2\,. \end{aligned} \tag{34}$$

Denoting the unary activation function of the ith layer by $a^{(i)}(\cdot)$, we have

$$\ell_\mu^{(i+1)} = a^{(i+1)}(z_\mu^{(i+1)}) \quad \text{with} \quad z_\mu^{(i+1)} = \sum_{\nu=1}^{n_i} w_{\mu\nu}^{(i+1)} \ell_\nu^{(i)} + b_\mu^{(i+1)}\,, \tag{35}$$

and hence

$$\frac{\partial z_\mu^{(i+1)}}{\partial w_{\mu\nu}^{(i+1)}} = \ell_\nu^{(i)}\,, \qquad \frac{\partial z_\mu^{(i+1)}}{\partial b_\mu^{(i+1)}} = 1\,. \tag{36}$$

Next, we equate the deviations $\delta^{(i)}$ with the derivative $\partial_z L$ in (33). For the deviations $\delta^{(s)}$ at the output layer we find

$$\delta_\nu^{(s)} = \frac{\partial L(\theta, x, y)}{\partial z_\nu^{(s)}} = -\frac{2}{n_s}\left(y_\nu - a^{(s)}(z_\nu^{(s)})\right)\; a'^{(s)}(z_\nu^{(s)})\,. \tag{37}$$

In order to obtain the other $\delta^{(i)}$, we proceed recursively. For $\delta^{(s-1)}$, we obtain

$$\begin{aligned}\delta_\nu^{(s-1)} &= \frac{\partial L(\theta, x, y)}{\partial z_\nu^{(s-1)}} = \sum_{\mu=1}^{n_s} \frac{\partial L(\theta, x, y)}{\partial z_\mu^{(s)}} \frac{\partial z_\mu^{(s)}}{\partial z_\nu^{(s-1)}} \overset{(37)}{=} \sum_{\mu=1}^{n_s} \delta_\mu^{(s)} \frac{\partial z_\mu^{(s)}}{\partial z_\nu^{(s-1)}} \\ &\overset{(35)}{=} \left(\sum_{\mu=1}^{n_s} w_{\mu\nu}^{(s)} \delta_\mu^{(s)}\right)\; a'^{(s-1)}(z_\nu^{(s-1)})\,,\end{aligned} \tag{38}$$

where we used the fact that each $z_\mu^{(s)}$ depends on all $z^{(s-1)}$ for the second equality. Note that the weight matrix is transposed due to the fact that we are propagating backwards. The computation for the other $\delta^{(i)}$ is analogous, with the result

$$\delta_\nu^{(i)} = \left(\sum_{\mu=1}^{n_{i+1}} w_{\mu\nu}^{(i+1)} \delta_\mu^{(i+1)}\right)\; a'^{(i)}(z_\nu^{(i)})\,. \tag{39}$$

With this, we can substitute (36), (37) and (39) into (33) and find

$$\frac{\partial L}{\partial w_{\mu\nu}^{(i)}} = \delta_\mu^{(i+1)} \ell_\nu^{(i)}\,, \qquad \frac{\partial L}{\partial b_\mu^{(i)}} = \delta_\mu^{(i+1)}\,. \tag{40}$$

Finally, we arrive at the weight and bias updates (32),

$$w_{\mu\nu}^{(i)} \to w_{\mu\nu}^{(i)} - \alpha\left[\delta_\mu^{(i+1)} \ell_\nu^{(i)} + \lambda w_{\mu\nu}^{(i)}\right]\,, \qquad b_\mu^{(i)} \to b_\mu^{(i)} - \alpha \delta_\mu^{(i+1)}\,. \tag{41}$$

This concludes one iteration of gradient descent via backpropagation.

There are several versions of gradient descent algorithms, which differ by how much of the training set is used for the update. We discuss the most common ones and refer to [57] for an overview.

3.3.2. Stochastic/batch/mini-batch gradient descent

In the version discussed above, we perform a weight update after processing each input–output pair (x, y). This is called **stochastic gradient descent** (SGD). The opposite approach would be to feed all input–output pairs to the NN before updating the weights, which is called **batch gradient descent** (BGD). An intermediate approach is to perform weight updates after each mini-batch,[7] which is called **mini-batch gradient descent** (MBGD). Note that for $N_B = 1$, MBGD becomes SGD, and for $N_B = N_T$ it becomes BGD. In MBGD or BGD, we still feed input–output pairs one at a time to the NN and collect the change in the NN parameters similar to (41). However, all parameter updates are collected and averaged prior to actually performing the update. So, if we have N_B input–output pairs (x^a, y^a), we first compute the summed losses

$$\Delta w^{(i)} = \frac{1}{N_B} \sum_{a=1}^{N_B} \nabla_{w^{(i)}} L(w, b, x^a, y^a)\,, \qquad \Delta b^{(i)} = \frac{1}{N_B} \sum_{a=1}^{N_B} \nabla_{b^{(i)}} L(w, b, x^a, y^a) \tag{42}$$

analogously to (41). Once the gradients have been evaluated for all (x^a, y^a) pairs in the (mini-)batch, the parameters are updated with the averaged change,

$$w^{(i)} \to w^{(i)} - \alpha\left[\Delta w^{(i)} + \lambda w^{(i)}\right]\,, \qquad b^{(i)} \to b^{(i)} - \alpha \Delta b^{(i)}\,. \tag{43}$$

The stochasticity of SGD is thus a consequence of descending not along the (on average) best direction but along the direction that improves a single, randomly chosen input–output pair. This erratic descent pattern can help with overcoming saddle points in the gradient descent [58]; more complicated saddle points require more intricate techniques [51].

[7] Sometimes mini-batches are just called batches, so one has to be careful which procedure is meant by batch gradient descent.

Intuitively, saddle points and local minima are avoided by SGD since a local minimum or a saddle point that appears in the landscape of the BGD will not be present if we just look at a single random sample.[8] While there is the danger that the pair we looked at was an outlier and thus the gradient does not take us in the best direction, these effects average out over several SGD updates.

On the other hand, due to this more erratic descent, SGD fluctuates more and one might be worried that it fluctuates around an actual (local or global) minimum of the full batch. This can be countered by using a **learning rate schedule** which decreases the learning rate over time or once the change between epochs falls below a certain threshold.

Another advantage of SGD over BGD or MBGD is that SGD can be faster if already a subset of the training data approximates the averaged gradient descent direction reasonably well or if not all training data can fit into memory.

A disadvantage is that SGD is inherently sequential and does not parallelize in general. In contrast, in BGD, the changes $\Delta\theta$ in (42) can be computed in parallel for different input–output pairs, and then summed and updated. In MBGD, one can at least parallelize over the elements in a mini-batch. Given the massive parallel computing powers of modern CPU or GPU clusters, parallelizing can make a huge difference in training time.

If the NN was eventually settling in the global minimum of the loss function, it would of course not matter which training method is used. However, as studies of the Hessian of the loss functions show, NNs almost always end up in very flat local minima or saddle points (i.e. many eigenvalues of the Hessian are close to zero) [60]. This is not too surprising, since the landscape of the loss function is very complicated and has many local extrema. Interestingly, the minima found or selected by the different gradient descent algorithms do have different properties. For example, it is observed in [61] that SGD or MBGD leads to better generalization properties of the network. Even if the NN had comparable errors on the training set, NNs trained with SGD or MBGD performed better on the test set as compared to models trained with BGD. This is called the **generalization gap** of NNs. Moreover, NNs trained with BGD are much more prone to adversarial attacks as compared to training with SGD [60].

As explained above, in order to increase parallelization, it is preferable to have MBGD with larger mini-batches. Unfortunately, the generalization gap starts to occur before the mini-batch size reaches a size to make use of contemporary parallel computing resources [62]. A possibility to mitigate this is to dynamically adjust the batch size during training [63].

3.3.3. Momentum gradient descent

The idea of momentum gradient descent (MGD) is to add a momentum term to the GD. The reason is that the random descents of e.g. SGD perform poorly if there is a very steep direction in the BGD around which the SGD fluctuates. The momentum makes the GD depend on the direction of the previous descent. Labeling each parameter update by a time step t, we replace the updates (42) by

$$\begin{aligned} \Delta\theta_t &= \beta\, \Delta\theta_{t-1} + \sum_{a=1}^{N_B} \nabla_\theta\, J(\theta_t, x, y)\,, \\ \theta_{t+1} &= \theta_t - \alpha\, [\Delta\theta_t + \lambda \nabla_\theta\, r(\theta_t)]\ . \end{aligned} \tag{44}$$

The parameter β controls the momentum of the update with $\beta = 0$ corresponding to ordinary GD. Momentum helps to traverse local extrema (especially minima and saddle points) and smoothes the zigzag path of SGD. Typical values for β are $\beta \sim 0.9$.

A slight modification to momentum GD is the **Nesterov accelerated GD** [64] algorithm, where the gradient is evaluated at the position where the gradient will take us in the next time step rather than where we currently are, i.e. the first line of (44) is replaced by

$$\Delta\theta_t = \beta\, \Delta\theta_{t-1} + \sum_{a=1}^{N_B} \nabla_\theta\, J(\theta_t - \beta\, \Delta\theta_{t-1}, x, y)\,. \tag{45}$$

3.3.4. Adaptive gradient descent

As mentioned above, it can be tricky to find the correct learning rate or gradient step size α. While learning schedules can ameliorate the problem, they are also fixed before training and hence do not take into account features of the training data. Moreover, gradients in the first layers of a deep NN can be very small, especially if the (initial) weights are very small. In this respect, momentum GD is better since the updates differ for different parameters depending on the momentum in their respective directions. Of course, all adaptive methods can be applied to SGD, MBGD, or BGD (but usually they are used together with SGD or MBGD with small mini-batch sizes).

Another approach is to use **Adaptive Subgradient Descent** (AdaGrad) [65], where the learning rate for each parameter depends inversely on the gradient in its direction (rather than the momentum). The idea is to provide a larger learning rate for parameters with a small gradient and a small learning rate for parameters with a large gradient, thus equalizing learning rates over parameter space,

$$\theta_{t+1,a} = \theta_{t,a} - \frac{\alpha}{\sqrt{G_{t,a} + \varepsilon}} \left[\Delta\theta_{t,a} + \lambda \nabla_{\theta_{t,a}}\, r(\theta_t)\right]\ . \tag{46}$$

[8] However, there are also studies illustrating that for large NN, SGD does not outperform GD [59].

Here, the subscript a labels the parameters θ of the network,

$$G_{t,a} = \sum_{\tau=0}^{t} \left(\nabla_{\theta_{\tau,a}} J(\theta_{\tau,a}, x, y)\right)^2 \tag{47}$$

is the sum of squares of the previous gradients in this direction up to the current time step, and $\varepsilon \sim \mathcal{O}(10^{-8})$ is a regularizer to prevent division by zero. Dampening the denominator with a square root has been found to lead to better results than just taking the accumulated gradients $G_{t,a}$. Note that $G_{t,a}$ keeps accumulating over time and hence the learning rate eventually tends to zero. Other adaptive methods, such as **RMSProp** or **AdaDelta** [66], get around this learning rate decay by letting the earlier contributions to $G_{t,a}$ in (47) decay exponentially. For example, in RMSProp, we let

$$G_t = \gamma G_{t-1} + (1-\gamma)\left(\nabla_{\theta_t} J(\theta_t, x, y)\right)^2 . \tag{48}$$

As for MGD, typical choices for γ are of the order $\gamma \sim 0.5$ to 0.9.

For AdaDelta, the learning rate α is not taken as a hyperparameter at all but is also approximated by a root mean square error,

$$\alpha_{t+1} = \sqrt{E(\Delta\theta^2)_t + \varepsilon}\,, \tag{49}$$

of the averaged, decaying parameter updates $\Delta\theta_t$. The parameter ε prevents getting a learning rate of zero. The functions $E(\Delta\theta^2)_t$ are defined recursively via

$$E(\Delta\theta^2)_t = \gamma E(\Delta\theta^2)_{t-1} + (1-\gamma)\,\Delta\theta_t^2\,, \tag{50}$$

Note that we only use $E(\Delta\theta^2)_t$ instead of $E(\Delta\theta^2)_{t+1}$ to compute α_{t+1} in (49). The reason is that the latter would require using $\Delta\theta_{t+1}$, which is the quantity currently being computed and hence not yet at our disposal. The parameter updates for $\Delta\theta_{t+1}$ in AdaDelta are now given by substituting (48) and (49) into (46),

$$\theta_{t+1,a} = \theta_{t,a} - \frac{\sqrt{E(\Delta\theta^2)_t + \varepsilon}}{\sqrt{G_{t,a} + \varepsilon}}\left[\Delta\theta_{t,a} + \lambda \nabla_{\theta_{t,a}} r(\theta_t)\right] . \tag{51}$$

Finally, **Adaptive Moment Estimation** (ADAM) [67] combines RMSProp and MGD. It uses an exponentially decaying average of past gradients (like MGD) *and* of past gradients squared (like AdaDelta or RMSProp),

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\left(\nabla_{\theta_t} J(\theta_t, x, y)\right) , \\ v_t &= \beta_2 v_{t-1} \ \ + (1-\beta_2)\left(\nabla_{\theta_t} J(\theta_t, x, y)\right)^2 . \end{aligned} \tag{52}$$

The m_t and v_t are estimates for the first moment (the mean) and the second moment (the variance), which explains the name ADAM. The two hyperparameters β_1 and β_2 control the exponential decay of the two moments. Typical values are $\beta_1 \sim 0.9$ and $\beta_2 \sim 0.999$. Since β_1, β_2 are close to one and the initial moments are initialized with zero, $m_0 = v_0 = 0$, m_t and v_t are biased towards zero, especially in the beginning. To counter this, the authors of [67] introduce the bias-corrected moments

$$\hat{m}_t = \frac{m_t}{1-(\beta_1)^t}\,, \qquad \hat{v}_t = \frac{v_t}{1-(\beta_2)^t}\,. \tag{53}$$

Since typically $\beta_{1,2} \lesssim 1$, the denominator is close to zero for small t and the bias-corrected moments are large. As t grows, $(\beta_{1,2})^t \to 0$ and the denominator approaches 1. The parameter update is

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \varepsilon}\,\hat{m}_t\,, \tag{54}$$

with $\varepsilon \sim \mathcal{O}(10^{-8})$.

Summary

In gradient descent, a minimum in parameter space is found by following the direction of steepest descent. The parameters of the network are updated backwards, starting from the difference between the final output and the training output. Several techniques exist to prevent too small or too large gradients during training, to choose a good learning rate, and to overcome saddle points during training.

3.4. Parameter initialization

In order to perform the first forward/backward pass we need to initialize the weights. One option is to initialize them just with zeros. However, setting all weights equal means that all gradients will be the same, which will lead to identical parameter updates and hence never allow for different weight values. It is thus better to break the symmetry and initialize the weights with different values. Note that, if the NN parameters are both weights and biases, it is enough to break the symmetry on the weights, since this will lead to symmetry breaking for the biases during backpropagation as well.

Depending on the activation function, there is also the problem of **vanishing gradients** or **exploding gradients**. Note that for (leaky) ReLU, the gradients are constants and the problem does not occur. For other activation functions, such as tanh or logistic sigmoid, weights with $|w| \sim \mathcal{O}(5)$ will lead to very small gradients, especially at the early layers since the signal diminishes during backpropagation. This diminishing effect can be seen from Fig. 4: the derivative $\sigma'(x)$ is in the interval $[0, 0.25]$ and $\tanh'(x)$ is in $[0, 1]$; Moreover, for $x = 5$ the gradients are already $\mathcal{O}(10^{-3})$ and $\mathcal{O}(10^{-4})$, respectively. Hence, since the parameter updates at layer i depend multiplicatively on the gradients at layer $i+1, i+2, \ldots, s$ (where s is the final layer), the update values get smaller and smaller.

On the other hand, large weights can also lead to large error fluctuations and hence to large gradients, which is problematic for convergence. We have introduced weight regularization in Section 3.3 to counter this. These problems do not only arise in feed-forward NNs but also RNNs, which can be unrolled to (infinitely) deep feed-forward NNs.

A simple choice for initializing weights is to draw random numbers from a **normal** (i.e. Gaussian) **distribution** with zero mean and unit variance,

$$\theta \sim \mathcal{N}(0, 1)\,, \tag{55}$$

due to a lack of a priori knowledge of the actual distribution underlying the parameter values. Alternatively, one can draw from a **uniform distribution**,

$$\theta^{(i+1)} \sim \mathcal{U}\left(-\frac{1}{\sqrt{n_i}}, \frac{1}{\sqrt{n_i}}\right)\,. \tag{56}$$

Note that the interval is chosen with respect to the number of neurons in the previous layer.

A variation of this is **He initialization** [68]. It is often used for (leaky) ReLU functions and uses a different variance or boundary for the normal and uniform distributions, respectively,

$$\theta_i \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_i}}\right)\,, \qquad \theta_i \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_i}}, \sqrt{\frac{6}{n_i}}\right)\,. \tag{57}$$

The numerical factors are chosen such that

$$\frac{1}{2} n_i \mathrm{var}(\theta_i) = 1\,, \tag{58}$$

where $\mathrm{var}(\theta_i)$ is the variance of the parameters θ_i. Eq. (58) is a necessary condition to avoid exponential growth or decay of the input signals.

A very popular parameter initialization to combat the vanishing or exploding gradient problem for tanh activation is **Xavier initialization** [69]. The idea is to draw from a normal distribution whose variance is chosen such that the weights are neither too big nor too small. If we assume independence of the weights and the input values $\ell^{(i)}$ of the ith layer and that they have zero mean, we find for the variances

$$\mathrm{var}(\ell^{(i+1)}) = n_i\, \mathrm{var}(w^{(i)})\, \mathrm{var}(\ell^{(i)})\,. \tag{59}$$

So if we want the variance of $\ell^{(i+1)}$ to be the same as that of $\ell^{(i)}$, we need $\mathrm{var}(w^{(i)}) = 1/n_i$. However, if we backpropagate, we find the condition $\mathrm{var}(w^{(i)}) = 1/n_{i+1}$. Usually, both conditions cannot be met simultaneously, since NNs have a different number of nodes in different layers. For this reason, the authors of [69] propose a variance of $\mathrm{var}(w^{(i)}) = 2/(n_i + n_{i+1})$ as a compromise, and draw the weights at initialization from

$$\theta^{(i+1)} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{(n_i + n_{i+1})}}\right)\,. \tag{60}$$

Other variations include drawing from a uniform rather than normal distribution,

$$\theta^{(i+1)} \sim \mathcal{U}\left(-\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}, \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}\right)\,. \tag{61}$$

A technique to make the NN more robust against initialization and learning rate choices is to use **batch normalization** [70]. This means that after each mini-batch the input is rescaled to have zero mean and unit variance. The changes are made to the entire network by backpropagation. It also acts as a regularizer and can ameliorate overtraining in a way similar to weight regularization or clipping.

Another way of achieving normalization, i.e. zero mean and unit variance, is to use **scaled exponential linear units** (SELUs) [71], which are defined as

$$\mathrm{SELU}(x) = \begin{cases} \lambda x & \text{if } x > 0 \\ \lambda\alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}\,, \tag{62}$$

with typically $\alpha, \lambda > 1$. Note that this is continuous at $x = 0$, see Fig. 15. Furthermore, despite its resemblance to a combination of leaky ReLU and logistic sigmoid or tanh activation, the SELU unit cannot be reproduced using these

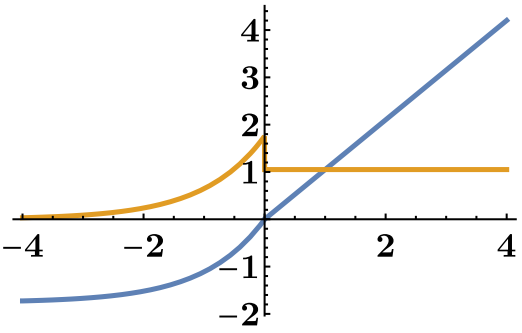


Fig. 15. Scaled exponential linear unit activation function (blue) and its derivative (orange) for $\alpha = 1.6733\,,\ \lambda = 1.0507$.

layers, i.e. a NN consisting of these layers cannot identically reproduce a SELU. Unlike the logistic sigmoid function, the SELU function can produce both positive and negative outputs, which is necessary to obtain zero mean. In order to change the variance such that it is unity, the activation function needs to have regions that have almost zero slope as well as regions with slopes larger than one (hence $\lambda > 1$). The hyperparameters α and λ need to be adjusted in order to ensure the self-normalizing property. If the weights are initialized with zero mean and unit variance, then

$$\alpha \sim 1.6733\,, \qquad \lambda \sim 1.0507\,. \tag{63}$$

For other cases see [71].

Yet another way to fight exploding gradients is **gradient clipping**. As the name suggests, one defines a maximum value for the gradient, and if the computed gradient is larger, it is set to this maximum value.

Summary

The NN parameters need to be randomly initialized for gradient descent to work. Carefully engineering the variance of the parameters can help to prevent too small/large gradients during training.

3.5. Loss functions

The choice of loss function depends on the area of application for the neural network. There is a wealth of loss functions, so we will only focus on the most common choices. See e.g. [72] for an overview and discussion of more loss functions, especially for classification NNs. We begin with the loss functions that are typically used in regression (Mean errors, L_p-norm errors, Huber loss) and then discuss the loss functions that are most commonly applied in classification tasks (Cross entropy, Kullback–Leibler, Hinge loss). We will follow the notational convention introduced in Section 2 and denote the input to the NN by x, the output of the NN by $\hat{y} = h_\theta(x)$, and the target output by y.

Mean errors

Among the mean error loss functions the **mean square error** (MSE) loss functions is one of the most commonly used. Mean errors are often used when the NN outputs are actual numerical values (rather than interpreting the output as probabilities), so for example in regression. The MSE loss is computed as

$$L^{\text{MSE}}(y, \hat{y}) = \frac{1}{n_s}\sum_{i=1}^{n_s}(y_i - \hat{y}_i)^2\,. \tag{64}$$

Sometimes, the **root mean square error** is used instead, which is just the square root of the MSE above. Of course, also other powers are conceivable. Let us at this point illustrate a problem of using MSE in conjunction with the logistic sigmoid function which we already hinted at in Section 2.2. Let us consider the simplest case, i.e. a single layer with a single node with logistic sigmoid activation and no bias. In this case $\theta = w$ and we find for the gradient of the MSE loss function L^{MSE} with respect to θ:

$$\nabla_\theta\, L^{\text{MSE}}(x, y) = \frac{\partial L^{\text{MSE}}}{\partial w} = \frac{\partial[(y - \sigma(wx))^2]}{\partial w} = -2x(y - \sigma(wx))\sigma'(wx)\,. \tag{65}$$

As can be seen from $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ or from Fig. 4b, the derivative term is very small if the absolute value of wx is very large. Thus, even if the discrepancy $(y - \sigma(wx))$ is very large, the loss can still be very small if the absolute value of wx is very large. Other activation functions do not suffer from this problem when used with the MSE loss function.

An alternative to the MSE is the **mean absolute error** (MAE), which is computed by summing the absolute value of the difference between $\hat{y}_i$ and y_i,

$$L^{\text{MAE}}(y,\hat{y}) = \frac{1}{n_s}\sum_{i=1}^{n_s}|y_i - \hat{y}_i|\,. \tag{66}$$

The difference between MAE and MSE is that MAE is more robust to outliers, since the difference is linear rather than squared. Thus, for MSE the NN will focus more strongly on fitting well an outlier while accepting being off for many other values. However, the gradient for MSE can be computed more easily than for MAE. Also, the MSE loss function is more stable in the sense that slightly changing one of the values $(y_i - \hat{y}_i)$ changes MSE less than MAE.

Another possibility to prevent over-emphasizing outliers is to take the **mean square logarithmic error** (MSLE),

$$L^{\text{MSLE}}(y,\hat{y}) = \frac{1}{n_s}\sum_{i=1}^{n_s}(\log(1+y_i) - \log(1+\hat{y}_i))^2\,. \tag{67}$$

For small values, MSE and MSLE are comparable as can be easily seen from Taylor expansion. The disadvantage is that MSLE is not defined for values of $\hat{y}$ or y smaller than -1. For large discrepancies, the error grows very slowly. This can be useful if the actual values y are very large, which will mean that all errors will be rather large generically (without fine-tuning, subtracting two order N numbers will result in an order N number).

Finally, there is the **mean absolute percentage error** (MAPE), which is defined as

$$L^{\text{MAPE}}(y,\hat{y}) = \frac{1}{n_s}\sum_{i=1}^{n_s}\left|\frac{(y_i - \hat{y}_i)}{y_i}\right|\cdot 100\,. \tag{68}$$

An obvious disadvantage is that this is not defined if one of the ground truth values y_i is zero. Furthermore, since a discrepancy can be too low by at most 100%, but too high by an arbitrary amount of percent, there is a disparity between these two cases and, as a consequence, models that systematically underestimate are given preference during training.

L_p-norm errors

Instead of taking the mean, we can also use the distance between the two vectors $\hat{y}$ and y in any L_p-vector norm,

$$L^{p\text{-NE}}(y,\hat{y}) = \|y-\hat{y}\|_p = \left(\sum_{i=1}^{n_s}(y_i - \hat{y}_i)^p\right)^{\frac{1}{p}}\,. \tag{69}$$

For $p = 1$, this is taken to be the absolute value norm, for $p = 2$ we get the Euclidean distance, and for $p = \infty$, this norm just gives the maximum vector component. Again, there are variations where the pth power of these norms are used to avoid roots. Usually one takes the cases $p = 1$ or $p = 2$, and the advantages or disadvantages are the same as in the averaged case, i.e. for MAE and MSE discussed above.

Huber loss

As is the case for the other loss functions discussed so far, the Huber loss function (HUB) is also mainly used for regression. It addresses the problem of quadratic sensitivity to outliers present for MSE or L_2-norm error by restricting the region where the error is squared to an interval Δ:

$$L^{\text{HUB}}(y,\hat{y}) = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & \text{if } |y-\hat{y}| < \Delta \\ \Delta|y-\hat{y}| - \frac{\Delta}{2} & \text{else}\,. \end{cases} \tag{70}$$

Cross entropy loss

Cross entropy loss (CEL) functions are used when the NN output is interpreted as probabilities, for example in a classification task. Before we talk about cross-entropy, let us quickly review entropy in an information theory context. Given a probability distribution P, the (Shannon) entropy is defined as

$$S(P) = \mathbb{E}_{x\sim P}[-\log_2 P(x)]\,, \tag{71}$$

where $x \sim P$ denotes that x is drawn from the probability distribution P. As a simple example consider the discrete case, where we are given a set P of cardinality N. Let there be furthermore $K < N$ different elements, which partitions P into K classes of elements, c_a, $a = 1,\ldots,K$, each of which has n_a elements. We then define $p_a = n_a/N$ and the entropy becomes

$$S(P) = -\sum_{a=1}^{K} p_a \log_2(p_a)\,. \tag{72}$$

The entropy is a measure for how many bits are needed on average to describe the distribution of values x in X. The **cross entropy** is comparing two distributions P and Q that are defined with the same support by averaging with respect to P but counting the bits with respect to Q

$$S(P,Q) = \mathbb{E}_{x\sim P}[-\log_2 Q(x)]\,. \tag{73}$$

Note that $S(P,P)=S(P)$. In the discrete case, Eq. (73) simply becomes

$$S(P,Q)=-\sum_{a=1}^{K} p_a \log_2(q_a)\,, \tag{74}$$

where p_a and q_a are the relative frequencies of the classes c_a in P and Q, respectively. The reason why probability theory enters in NNs is that we can see the NN as guessing which distribution or function is actually underlying the discrete, finite data set it is presented with during training.

In binary classification, where $y\in\{0,1\}$ labels the two classes, the **binary cross entropy loss** (BCEL) is then simply

$$L^{\text{BCEL}}(y,\hat{y})=-[y\log_2(\hat{y})+(1-y)\log_2(1-\hat{y})]\,. \tag{75}$$

The output $\hat{y}\in[0,1]$ of the NN is interpreted as the probability of some input belonging to class zero or class one, where $\hat{y}=1/2$ signals complete uncertainty. Note that, since y is either zero or one, only one of the two terms in (75) contributes. Due to the logarithm, the loss is very large if the network predicts the wrong class with a high certainty. If, for example $y=1$ but the network predicts a value $\hat{y}$ close to zero, then the error becomes the logarithm of a very small number (the second term in the sum is identically zero) and hence blows up. Similarly, the second term blows up if $y=0$ but the network predicts a value close to one. In contrast, if the NN assigns the correct class with absolute certainty, the loss is zero, as it should be.

Let us look at the generalization to a multi-class one-vs-all classification problem. We denote the set of different classes by C and the cardinality of C by K. The K individual classes are denoted by $c_a\in C$, $a=1,\dots,K$. We furthermore define $y_{c_a,x}$ as a binary indicator for which class the current input x belongs to (i.e. the K-dimensional zero vector with a single one at the ath position). Then the **negative logarithmic likelihood loss** (NLLL) or multi-class cross entropy loss is given by

$$L^{\text{NLLL}}(x,y)=-\sum_{a=1}^{K} y_{c_a,x}\log\hat{y}_{c_a,x}\,, \tag{76}$$

where $\hat{y}_{c_a,x}$ is the probability with which the NN assigns x to class c_a (usually the output of a softmax layer, i.e. $\sum_a \hat{y}_{c_a}=1$). If the NN assigns x to the correct class with a very low certainty, the corresponding logarithm blows up. Note, however, that since always only one term in the sum contributes, it is irrelevant how the probabilities over the other $K-1$ classes are distributed: If the NN predicts the membership probability of the correct class with low probability, it necessarily either predicted membership for a single, false class with a high probability or for multiple false classes with intermediate probabilities.

Remember that in backpropagation, we need the derivative of the loss function, cf. Eq. (33). The derivative of NLLL for a softmax activation function is just linear due to the log in NLLL in (76) and the exponential in softmax in (8) (note that only the term with y_{c,x_i} contributes),

$$\partial_{x_i} L^{\text{NLLL}}(x,y)=\hat{y}_i-1\,. \tag{77}$$

Kullback–Leibler loss

Kullback–Leibler (KL) is a measure for how much two probability distributions differ [73]. This plays an important role in GANs, cf. Section 4.7. It is defined as the difference between the entropy (71) and cross entropy (73) of two distributions P and Q,

$$\text{KL}(P\parallel Q)=S(P,Q)-S(P)=\mathbb{E}_{x\sim P}[\log_2 P(x)-\log_2 Q(x)] \tag{78}$$

Remember that $S(P,P)=S(P)$, so the KL divergence is zero if the two distributions are the same. Since the logarithm essentially encodes the number of bits, the KL divergence is a measure for how many bits of information are lost.

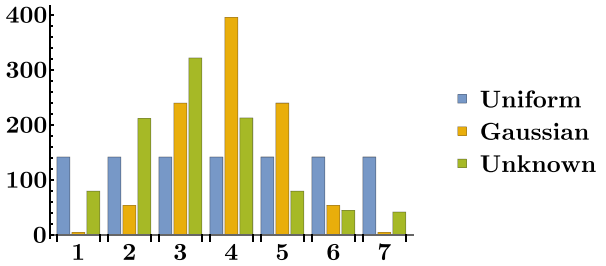
In the discrete case relevant for machine learning, the KL loss becomes

$$L^{\text{KL}}(y,\hat{y})=\sum_{a=1}^{K} y_{c_a,x}(\log y_{c_a,x}-\log\hat{y}_{c_a,x})=\frac{1}{n_s}\sum_{\mu=1}^{n_s} y_\mu(\log y_\mu-\log\hat{y}_\mu)\,. \tag{79}$$

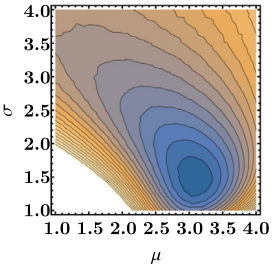
In the second equality, we have used the assumption that the last NN layer is a softmax layer with $n_s=K$ nodes and output value $\hat{y}_\mu$. The training ground truth is simply the indicator vector $y_{c_a,x}$. If $\hat{y}=y$, the entropy equals the cross entropy, the two distributions are the same, and the KL loss is zero. On the other hand, a value close to one indicates that the two distributions are very different. Note for later that the KL divergence is asymmetric, $\text{KL}(P\parallel Q)\neq\text{KL}(Q\parallel P)$.

In order to exemplify the underlying concepts, let us look at a simple example. Let us assume we have 7 classes $c_a=\{1,2,\dots,7\}$. We now draw 987 times from C following some unknown probability distribution R. The resulting histogram is given in Fig. 16a, together with the histogram that would result from drawing with a uniform distribution $P\sim\mathcal{U}(1,7)$ or a Gaussian distribution $Q\sim\mathcal{N}(4,1)$. The relative frequencies of elements p,q,r in P,Q,R are

	1	2	3	4	5	6	7
p	1/7	1/7	1/7	1/7	1/7	1/7	1/7
q	4/987	53/987	239/987	395/987	239/987	53/987	4/987
r	88/987	221/987	322/987	220/987	70/987	45/987	41/987



(a) Histogram for uniform (blue), Gaussian (orange) and our test distribution (green).



(b) KL loss between Gaussian and test distribution.

Fig. 16. Example to illustrate the KL loss between a Gaussian or uniform distribution and our example distribution. We minimize the loss between the Gaussian and the test distribution by adjusting the Gaussian parameters (μ, σ).

Let us now compute two KL losses, assuming our data was drawn from a uniform or Gaussian distribution, respectively. The KL losses are

$$\begin{aligned} \mathrm{KL}(P \parallel R) &= \sum_{a=1}^{7} p_{c_a}(\log p_{c_a} - \log r_{c_a}) \\ &= \frac{1}{7}(\log_2 \frac{1}{7} - \log_2 \frac{88}{987}) + \frac{1}{7}(\log_2 \frac{1}{7} - \log_2 \frac{221}{987}) + \cdots \approx 0.39\,, \end{aligned} \tag{80}$$

$$\begin{aligned} \mathrm{KL}(Q \parallel R) &= \sum_{a=1}^{7} q_{c_a}(\log q_{c_a} - \log r_{c_a}) \\ &= \frac{4}{987}(\log_2 \frac{4}{987} - \log_2 \frac{88}{987}) + \frac{53}{987}(\log_2 \frac{53}{987} - \log_2 \frac{221}{987}) + \cdots \\ &\approx 0.59\,. \end{aligned} \tag{81}$$

This means we lose slightly more information (bits) by approximating the unknown example distribution by a Gaussian with mean 4 and unit variance as compared to just a uniform distribution.

However, the parameters in our Gaussian, i.e. the mean μ and the standard deviation σ, have not yet been optimized for the test distribution but just chosen as $(\mu, \sigma) = (4, 1)$. In order to get a feeling for what would happen in an optimization, we minimize the KL loss with respect to these parameters, cf. Fig. 16b. We find that the best Gaussian fit has approximately $(\mu, \sigma) = (3, 1.5)$. From the histogram it is obvious that a Gaussian centered around 3 with a larger variance is a better fit. The KL loss for these parameters is 0.06, which improves the previous result by an order of magnitude and is a better fit than the flat distribution.

Hinge loss

Hinge Loss (HIL) is often used in support vector machines. For a binary classification task with ground truth labels $y_i = \pm 1$ and not necessarily normalized output $\hat{y}$, the loss is defined as

$$L^{\mathrm{HIL}}(y, \hat{y}) = \max(0, 1 - y\hat{y}) =: [1 - y\hat{y}]_+\,. \tag{82}$$

where the bracket $[\cdot]_+$ is defined as the maximum of 0 and $\cdot$. If the network is rather certain that the input belongs to a certain class, it assigns a large positive or negative value $\hat{y}$. If the prediction is correct, $\hat{y}$ and y will have the same sign and the loss is zero. If the prediction is incorrect or the NN is not certain and assigns an intermediate value $\hat{y} \in [-1, 1]$, the loss is linear in the network's assignment. Since the function's derivative is discontinuous at the origin, a variant called **square hinge loss** (SHIL) can be used, which is simply the square of HIL

$$L^{\mathrm{SHIL}}(y, \hat{y}) = \max(0, 1 - y\hat{y})^2\,. \tag{83}$$

This makes the derivative continuous.

Summary

There are different loss functions depending on whether the NN is used in regression or classification. Regression losses are based on the distance between the ground truth values y and the NN predictions $\hat{y}$. For classification, the losses usually try to minimize the entropy difference between the class distribution of the training set and the one used by the NN to predict the classes. Since derivatives of the losses appear in the gradient descent, all loss functions should have derivatives that are fast to compute and reasonably well-behaved.

3.6. Overfitting and underfitting

Overfitting or overtraining is a process that can occur both in statistics and in regression. In statistics, overfitting describes a situation where, in order to reduce the error, the NN learns to model every tiny feature in the training set (which might just be outliers or noise), rather than to capture the global structure. Another effect of overfitting in statistics is that the neural network learns the distribution of the training set, which might differ from the distribution of the actual data set on which the NN is to be used. For this reason, it is important to design the training set such that its data resembles the actual distribution of data as closely as possible and to stop training before overfitting.

In regression or interpolation, overfitting manifests as follows: Given a set of N data points, it is possible to find a degree $N-1$ polynomial which passes through these data points. However, the higher the degree of the polynomial, the more it oscillates in between the data points, which is often a poor fit to the actual data. To ameliorate this, one can use e.g. cubic splines, which interpolate between two data points using a cubic polynomial (a line would be enough to interpolate between two points; a cubic is used to make the total interpolation smooth at the connections of the splines).

The opposite of overfitting is underfitting. An underfitted model has not learned to use essential features of the input set. A reason for underfitting might be that the NN is simply not complex or powerful enough, i.e. it does not have enough parameters, to fully fit the data. Another reason might be that the NN has not been trained long enough and the NN has settled in a (rather bad) local minimum or has not yet converged to any local minimum at all, e.g. because it is stuck at a saddle point. A third reason might be that the training set is not large or diverse enough to contain enough information about the actual data distribution. Remedies are to include more or larger hidden layers, to increase training time or use other optimizers, and to enlarge or choose a better suited training set, respectively.

A very common technique to identify overfitting is to split the training data into a train and test (and validation) set as explained in Section 3.1. During training, we evaluate the training process of the incompletely trained NN on this test set. It is not unusual that the training set accuracy is a bit higher than the test set accuracy. If the test set accuracy does not improve further while the training set accuracy does, or if the training set accuracy is significantly higher than the test set accuracy, it is likely that the NN is being overtrained, or that the network's design and hyperparameter choice have been fine-tuned to the training set.

There are also techniques to prevent the NN from overfitting data in the first place. They are based on the idea that overfitting is avoided if the NN cannot rely too much on any given part of its layered evaluation process. Hence the network cannot develop different parts that are tailored to deal with specific parts of the training set. By preventing the NN from relying too much on any given weight (or combination of weights), the NN is trained to generalize better and hence is less likely to overfit the data.

Example for over- and underfitting

To illustrate over- and underfitting, we use a simple toy problem. We set up a NN with 2 hidden layers with 15 nodes and tanh activation. We train the NN on a Gaussian with some noise, overlaid with a small amplitude, high-frequency sinusoidal modulation,

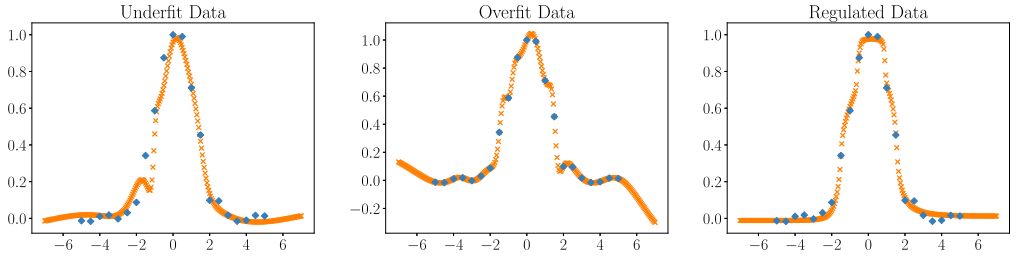
$$f(x) = 0.02\sin(3x) + e^{-\frac{(x+\epsilon)^2}{2}}, \qquad \epsilon \in \mathcal{U}(-0.3, 0.3)\,. \tag{84}$$

For our training set, we generate 21 equally spaced data points from the interval $x \in [-5, 5]$ and compute the corresponding y-values from (84). We initialize the weights with Xavier initialization and use batch gradient descent with an ADAM optimizer with mean square loss. The results are shown in Fig. 17. In the underfitted model 17a, we train for 200 epochs. The overall structure is well approximated, but the finer structure of the sinusoidal modulation in the training data set are not reproduced. In 17b, the model is trained for 10^5 epochs, which is severely overtraining it. It fits all training data points extremely well but oscillates in between. The third plot 17c has also been trained for 10^5 epochs, but uses dropout layers for regularization, as we will discuss next.

Dropout layers

A very common technique to prevent overfitting is the use of **dropout layers** [74]. These layers can be thought of as filters that block data propagation of some randomly chosen neurons of the layer that comes immediately before the dropout layer to the layer that comes immediately after the dropout layer. The dropout layer thus effectively disables or temporarily removes ("drops") some nodes from the layer before the dropout layer. This means their signal is ignored in the forward pass and no parameter updates are performed in the backward pass. The size of the fraction of nodes dropped is a hyperparameter that needs to be specified when designing the NN. Usual values are 10 to 40 percent.

Randomly removing neurons means that other neurons have to take over and the NN is forced to learn different representations of the same data. In a sense, by dropping part of the network, we can think of the full NN to consist of several NNs that all compute the result and vote in the end on a common result, see Fig. 18. Usually, a dropout layer is inserted towards the end behind a fully connected layer, but often several other dropout layers are introduced at different points of the network. There is no clear-cut rule on the number and position of dropout layers. Some insert dropouts after each layer, while others only include them right after the input and/or the output layer of the NN. If dropout is used, the NN needs to be large and hence complex enough to actually learn different representations of the data. In addition, a



(a) Underfit data. (b) Overfit model. (c) Regularization model.

Fig. 17. We draw data points (blue) from a Gaussian distribution with random fluctuations and sinusoidal modulations. The underfit model 17a captures the overall shape but does not see the modulations. The overfit model 17b captures the modulations, but oscillates in between training data points. The dropout regulated model 17c finds a compromise between the two.

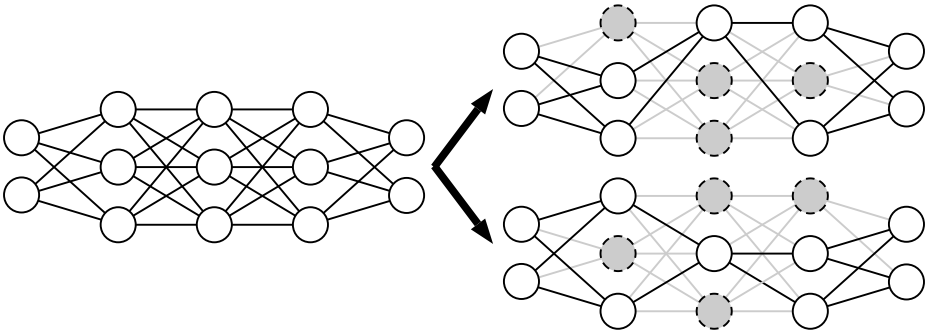


Fig. 18. Two different random dropouts for an example NN with a dropout layer after each hidden layer. Dropping nodes (dashed lines) and their subsequent connections (light gray) in each layer leads to different, less complex subnetworks. If either one learns to reproduce the training output from the input, the full NN can be interpreted as a collection of subnetworks that compute the output and vote on the final result.

larger learning rate and momentum should be chosen. Of course, once training is complete, the dropout layers should be deactivated in order to use all nodes of the NN.

We illustrate the effect of dropout regularization in Fig. 17c, where we add a dropout layer with a dropout rate of 10 percent after each hidden layer for the example NN that fits the modulated Gaussian. As we can see, even though we also train for 10^5 episodes, no overfitting occurs.

Weight regularization

Another technique that is often used independently or in conjunction with dropout is weight regularization or weight clipping. Weight regularization has been introduced in Section 3.3. It introduces a term proportional to the weights in the cost function of the NN, such that the NN favors small weights. Many functions can be chosen as a regularizer, as long as they have the property that they lead to a weight decay, i.e. favor small weights.

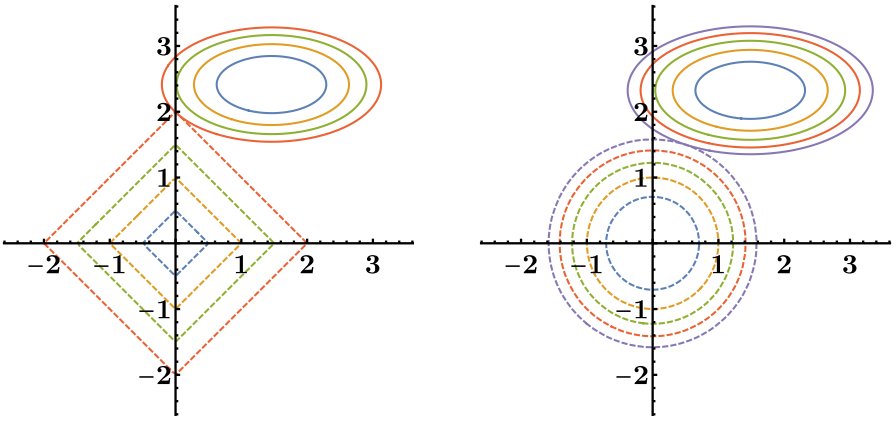
There are several ways to intuitively understand why small weights reduce the risk of overfitting. First, note that overfitting arises from too complex NNs. Small weights simplify the NN since they almost linearize the activation function (as one can truncate the Taylor expansion of the activation function after the first term). By removing the non-linearity, the NN performs three linear maps (matrix multiplication, linear activation, matrix multiplication), which is equivalent to just a single linear map, thus effectively removing the entire layer and simplifying the NN. Also, if the weights are small, the activation is weak for typical activation functions, which effectively removes the node from the NN computation, similar to dropout.

Common choices for weight regularizers are L_p-norm regularizers,

$$r(\theta) = r(w) = (\|w\|_p)^p = \sum_{i=0}^{s-1} \sum_{\mu_{i+1}=1}^{n_{i+1}} \sum_{\mu_i=1}^{n_i} |w^{(i)}_{\mu_{i+1}\mu_i}|^p \,. \tag{85}$$

with $p = 1, 2$. NNs with L_1-norm regularizers tend to have sparse weight matrices (i.e. matrices with many zero weights), while NNs with L_2-norm regularizers do not tend to allow for 0 weights, but due to the square, punish large weights much stronger than small weights. Since L_1-norm regularization simply ignores some nodes (by having zero weights), they show **feature selection** properties, i.e. they only use those features (or nodes in later layers) which are most important.

A regularization term leads to small weights, unless the cost incurred by increasing the weights is counterbalanced by a significant improvement of the NN, i.e. by a decrease of its loss. To see this note that $\lambda r(\theta)$ is added to the cost function



(a) L_1-norm regularization. (b) L_2-norm regularization.

Fig. 19. Contours of constant cost (solid lines) and constant regularization (dashed lines). The minimum of the loss is where the contours meet.

J in (31), with λ small and $r(\theta)$ a non-negative function. Hence the network can decrease the costs by simply choosing smaller θ, unless a decrease of the loss function L makes up for the increase of the regularizer $\lambda r(\theta)$.

In order to understand the difference between L_1-norm and L_2-norm regularization [75] let us look at a simple example with two weights, MSE loss, and identity activation

$$J = \frac{1}{2}([y_1 - (w_1 x_1 + b_1)]^2 + [y_2 - (w_2 x_2 + b_2)]^2) + |w_1|^p + |w_2|^p \tag{86}$$

The cost and regularizer function potentially compete with each other: The cost function might want larger w to get a better fit, but the regularizer wants smaller w (irrespective of the fit quality). We plot for some example values of (x, y, b) contours of constant cost and regularizer value in the w_1-w_2-plane in Fig. 19. The smallest overall cost is where the two contours meet. Since equidistant points in the L_1-norm lie on a diamond while they are on a sphere for the L_2-norm, chances are that the contour from the L_1-norm regularizer first intersects the contour from the cost function along one of the axes for L_1-norm regularization, while they intersect somewhere in the plane for L_2-norm regularization. This is why L_1-norm regularization leads to sparse matrices and consequently to feature selection.

Weight clipping is a simpler version of weight regularization. Instead of preventing weights from becoming too large by including them in the loss function, weight clipping simply sets weights whose values exceed a given maximum or minimum value (often chosen to be ± 2 or ± 3) to this boundary value.

Implicit regularization techniques

It has been found that **early stopping**, which means that training is ended before convergence is reached, has implicit regularization effects and leads to better generalization [76].

Another implicit regularization effect in deep NNs comes simply from the fact that gradient-based optimizers are used. This effect is currently not well understood and is an active field of research [77]. Studies with (deep) linear NNs (i.e. NNs with identity activation functions) suggest that this might come from the (in general non-unique) solution of matrix factorization picked by gradient descent [78,79]: as argued in Section 2, such NNs are equivalent to just a single matrix multiplication. However, in deep NNs, this single matrix multiplication is given by factorizing the matrix into the different weight matrices of the different layers, and each of these matrix entries are updated consecutively using gradient descent. See also Section 4.2 for a discussion of NN architectures and their (generalization) properties.

In [80], the authors also study generalization properties of NNs. However, they focus on the number of parameters (which are determined by the number of nodes and layers). They find that overparametrized NNs lead to better generalization, i.e. they are less sensitive to perturbations in the input data. This is surprising from the point of view that it could have been expected that simple models should be preferred to complex models. The study is carried out for a variety of models with different hyperparameters (different architectures, optimizers, etc.). In [81], a proposed explanation is that thanks to overparametrization, there is a high probability that a good solution to the optimization problem is in close vicinity to the (randomly initialized, overparametrized) NN. Moreover, the likelihood of encountering saddle points during optimization is small. Lastly, information tends to be distributed evenly among the neurons, i.e. the network does not rely too strongly on any particular neuron, which leads again to an implicit regularization.

Other regularization techniques

Further techniques that have a regularizing effect and can thus help prevent overfitting and improve training in general are **batch normalization** [70], **weight normalization** [82], and **layer normalization** [83]. As explained in Section 3.4,

batch normalization normalizes the input to the activation function for each batch to have zero mean $\langle\cdot\rangle$ and unit variance σ. Since shifting to zero mean cancels any bias (which is also a constant shift), the bias b can be ignored and set to the zero vector. After normalizing, batch normalization adds a new offset or bias β, which acts as a bias to the activation function. We use the notation of (3) with $b \equiv 0$, i.e. $z^{(i)}_{\mu,k}$ is the value of the μth node (prior to applying the activation function) of the ith layer for the kth mini-batch input. Batch normalization can be written as

$$\ell^{(i)}_{\mu,k} = a^{(i)}\left(\gamma_\mu \frac{z^{(i)}_{\mu,k} - \langle z^{(i)}_\mu\rangle}{\sigma(z^{(i)}_{\mu,k})} + \beta_\mu\right), \tag{87}$$

where β_μ and γ_μ are parameters to fill out most of the range of the activation function and

$$\langle z^{(i)}_\mu\rangle = \frac{1}{N_B}\sum_{k=1}^{N_B} z^{(i)}_{\mu,k}, \qquad \sigma^2(z^{(i)}_{\mu,k}) = \frac{1}{N_B}\sum_{k=1}^{N_B}(z^{(i)}_{\mu,k} - \langle z^{(i)}_\mu\rangle)^2. \tag{88}$$

are the mean and the variance.

In **weight normalization**, only the weights are normalized, i.e.

$$\ell^{(i)}_\mu = a^{(i)}\left(\gamma_\mu \frac{z^{(i)}_\mu}{\|w^{(i)}\|_2} + \beta_\mu\right). \tag{89}$$

In **layer normalization**, the mean and variance are normalized with respect to a single input over all nodes in the layer,

$$\ell^{(i)}_{\mu,k} = a^{(i)}\left(\gamma_\mu \frac{z^{(i)}_{\mu,k} - \langle z^{(i)}_k\rangle}{\sigma(z^{(i)}_{\mu,k})} + \beta_\mu\right), \tag{90}$$

where

$$\langle z^{(i)}_k\rangle = \frac{1}{n_i}\sum_{\mu=1}^{n_i} z^{(i)}_{\mu,k}, \qquad \sigma^2(z^{(i)}_{\mu,k}) = \frac{1}{n_i}\sum_{\mu=1}^{n_i}(z^{(i)}_{\mu,k} - \langle z^{(i)}_k\rangle)^2. \tag{91}$$

This looks very similar to batch normalization. In a sense, the averaging of layer normalization (91) is the transpose of the averaging in batch normalization (88): The former averages a single input k over all nodes $\mu = 1, \ldots, n_i$, while the latter averages for each node μ over all inputs $k = 1, \ldots, N_B$. Note that batch, weight, and layer normalization make the NN invariant under scaling of all weights, hence there is no point in using them in conjunction with weight regularization [84].

Finally, there is the option of adding random **noise** to the input. The idea is that the NN cannot rely on any given weight since it would also amplify the random noise. Instead, it has to learn the essential features of the data and hence rely on global properties that generalize rather than specializing on a specific part.

Summary

Overtraining occurs when the NN relies too much on specific weight combinations and is trained for too long such that it starts fine-tuning parameters and fitting noise rather than data in the training set. Overtraining results in large oscillations and bad generalization properties of the NN. Undertraining is the opposite effect where the NN is not trained long enough or is not complex enough.

Overtraining can be addressed with a variety of techniques, all of which are based on forcing the NN to not rely too heavily on any single neuron or combination of neurons. Undertraining can be prevented by using more complex NNs, increasing training time, or improving the training set.

3.7. Default choices for loss functions and optimizers

Most of the material discussed here is automated and provided in NN software packages. While understanding how the methods work and what the parameters do can significantly improve the training results, all the different choices might seem overwhelming at first. Certainly, there is not a canonical best choice and different methods have different merits, but a reasonable starting point is to perform a train:test split of 90:10, use random weight initialization, stochastic gradient descent with an ADAM optimizer, and a mean square error (for regression) or cross entropy (for classification) loss function with a dropout layer before the output layer and an L_2 weight regularizer.

3.8. Performance evaluation

The simplest way of analyzing the performance of a NN has already been discussed: split your training data into a train and a test set. Train with the training set only and evaluate how well the NN performs by applying it to the test set.

A variation of this is ***k*-fold cross validation**. This means that the training data is split into k sets of equal size. The NN is then trained k times. Each time, a different set is used as a test set and the remaining $k-1$ as training sets. The performance is evaluated for each of the k runs and then averaged. Of course, this means that training time is increased by a factor of k as compared to the simple train:test split.

Performance metrics

Next, we discuss metrics to measure how well a NN is performing. Depending on whether the NN is used for regression or classification, different metrics will be useful. For regression, one usually simply uses the **mean square error** or variations such as the **absolute mean error** or **root mean square error** to estimate the performance. For classification, there are many more metrics, which we discuss now.

Classification accuracy: The simplest is to just give the percentage of correctly classified input data. In case the training set is unbalanced, i.e. one class occurs much more frequently than the others, it is important to keep the **null accuracy** of the data set in mind. This is the accuracy that can be achieved by simply always guessing the most frequent class, independent of the input. However, when training classifiers it is usually better to have a balanced training set anyway. If the original set is not balanced, data augmentation can be used to produce a training set that is (more) balanced.

Confusion matrix: The confusion matrix V contains more information about the classification performance than just the classification accuracy. For a classification problem with K classes c_i, $i = 1, \ldots, K$, it is a $K \times K$ matrix whose rows indicate the actual classes and whose columns indicate the predicted classes. The matrix entries v_{ij} then indicate for each actual class c_i the class c_j which the NN assigned to the input. We give an example for a binary confusion matrix in Eq. (94). Given the training set $\mathcal{T} = \{(x_a, y_a)\}$, we define the sets

$$\mathcal{A}_i = \{x_a \in \mathcal{T} \mid y_a = c_i\}\,, \qquad \mathcal{P}_j(\mathcal{S}) = \{h_\theta(x_a) = c_j \mid x_a \in \mathcal{S}\}\,, \tag{92}$$

where $\mathcal{S}$ is any subset of $\mathcal{T}$. Then, the entries v_{ij} of the confusion matrix V are given by

$$v_{ij} = |\mathcal{P}_j(\mathcal{A}_i)|\,. \tag{93}$$

Thus, a perfect classifier produces a diagonal confusion matrix. Often, one also lists the total number of elements in each row and in each column. The **classification accuracy** is then given by the sum of all diagonal elements divided by the total number of elements in the training set. Similarly, the **classification error** or **misclassification rate** is given by the sum over all off-diagonal matrix elements divided by the total number of elements in the training set.

Receiver operating characteristic: The receiver operating characteristic (ROC) is defined for binary classification problems, but it can be extended to multi-class classification. In binary classification, we have two classes 0 and 1 (or false/true, negative/positive). The entries of the 2×2 confusion matrix are named accordingly:

- v_{00} are the true negatives (TN), since the actual class and the predicted class are negative,
- v_{01} are the false positives (FP) or Type I errors, since the actual class is negative, but the input has erroneously been classified as positive,
- v_{10} are the false negatives (FN) or Type II errors, since the actual class is positive, but the input has erroneously been classified as negative,
- v_{11} are the true positives (TP), since the actual class and the predicted class are positive.

Thus, the binary confusion matrix looks like

		predicted		
		neg	pos	
actual	neg	TN $= v_{00}$	FP $= v_{01}$	$v_{00} + v_{01}$
	pos	FN $= v_{10}$	TP $= v_{11}$	$v_{10} + v_{11}$
		$v_{00} + v_{10}$	$v_{01} + v_{11}$	

(94)

One furthermore defines the following quantities

- PPV $= v_{11}/\sum_i v_{i1}$, i.e. true positives over all predicted positives, is called the **positive predictive value** or **precision**,
- TPR $= v_{11}/\sum_i v_{1i}$, i.e. true positives over all actual positives, is called the **sensitivity** or **true positive rate** or **recall**,
- TNR $= v_{00}/\sum_i v_{0i}$, i.e. true negatives over all actual negatives, is called the **specificity** or **true negative rate** or **selectivity**,
- FPR $= v_{01}/\sum_i v_{0i}$, i.e. false positives over all actual negatives, is called the **false positive rate** or **fallout**,
- FNR $= v_{10}/\sum_i v_{1i}$, i.e. false negatives over all actual positives, is called the **false negative rate** or **miss rate**.

In binary classification, the NN usually outputs a real number in the interval [0, 1], which is interpreted as the NN's certainty that the input belongs to class 0 or 1. The boundary between the two classes is called the **threshold** or **decision boundary**. Usually this is at 0.5, indicating that the NN is completely unsure which class the input belongs to. In some applications, it might be important to minimize e.g. the false negatives, and then other thresholds can be chosen. Of course, this will change the other quantities as well.

We usually want to maximize both the sensitivity and the specificity (i.e. the correct predictions) by choosing an appropriate threshold. However, increasing one necessarily decreases the other. The ROC curve plots the false positive rate (which equals 1 − specificity) on the x-axis and the true positive rate on the y-axis while varying the threshold

between 0 and 1 in discrete steps. Note that for a threshold of one, both the true and false positive rates are zero (since everything is assigned to class 0, i.e. $v_{i1} = 0$), while for a threshold of 0 both the true and false positives rates are one (since everything is assigned to class 1, i.e. $v_{i0} = 0$).

For a point on the diagonal (the **line of no-discrimination**), the true and false positive rates are equal and the classifier is guessing randomly. For curves above the diagonal, the true positives are more frequent than the false negatives and the network is performing above average. Similarly, a ROC curve below the diagonal indicates a sub-par performance. The optimal point, also called the **perfect classification point**, is at (0, 1), i.e. no false positives and 100 percent of true positives. This means that there are also no false negatives.

While the perfect classification point is usually not reached, the steepness of the ROC curve is a good measure for how well a binary classifier is performing. The steeper a curve, the better the true positive predictions. Another common measure of quality is the **area under the ROC curve** (AUC). A perfect classifier would always have a true positive rate of 1, hence the maximum AUC is one. A classifier that performs randomly would have a ROC curve along the diagonal and hence an AUC of 0.5. Thus, the larger the AUC the better the classifier performs.

A further performance measure is the $\boldsymbol{F_1}$ **score**. It is the harmonic mean of the precision and the true positive rate,

$$F_1 = 2\,\frac{\mathrm{TPR}\cdot\mathrm{PPV}}{\mathrm{TPR}+\mathrm{PPV}}\,. \tag{95}$$

There are several possibilities to extend the ROC and AUC measure to **one-vs-all** multi-class classification problems. One-vs-all means that each input belongs to exactly one class. In this case, one can either draw one ROC curve for each class or perform averaging. In the former case, we can binarize the problem by computing K binary confusion matrices, each of which list the TN, TP, FN, FP with respect to a class c_i, $i = 1,\ldots,K$. In case of averaging, one can perform **micro-averaging** or **macro-averaging**. In micro-averaging, the ROC curve is obtained from computing the FPR and TPR by averaging over all K confusion matrices, while for macro-averaging, the FPR and TPR are computed for each confusion matrix and averaged subsequently,

$$\begin{aligned}
\mathrm{FPR}_{\mathrm{micro}} &= \frac{\sum_{k=1}^{K} v_{01,k}}{\sum_{k=1}^{K}\sum_{i=0}^{1} v_{0i,k}}\,, \qquad \mathrm{FPR}_{\mathrm{macro}} = \frac{\sum_{k=1}^{K}\mathrm{FPR}_k}{K}\,,\\
\mathrm{TPR}_{\mathrm{micro}} &= \frac{\sum_{k=1}^{K} v_{11,k}}{\sum_{k=1}^{K}\sum_{i=0}^{1} v_{1i,k}}\,, \qquad \mathrm{TPR}_{\mathrm{macro}} = \frac{\sum_{k=1}^{K}\mathrm{TPR}_k}{K}\,.
\end{aligned} \tag{96}$$

In cases where the training set is rather imbalanced, these quantities can differ significantly, with the micro-averaging being the more accurate measure. To illustrate this for the FPR, assume an extreme case of a 3-class one-vs-all classification problem with the following confusion matrices

$$V_{c_1} = \begin{pmatrix} 0 & 10 \\ * & * \end{pmatrix}\,, \qquad V_{c_2} = \begin{pmatrix} 1 & 0 \\ * & * \end{pmatrix}\,, \qquad V_{c_3} = \begin{pmatrix} 1 & 1 \\ * & * \end{pmatrix}\,. \tag{97}$$

The entries $*$ are not relevant in the computation of the FPR. The averaged rates are then

$$\begin{aligned}
\mathrm{FPR}_{\mathrm{micro}} &= \frac{10+0+1}{0+10+1+0+1+1} \approx 0.85\,,\\
\mathrm{FPR}_{\mathrm{macro}} &= \frac{1+0+0.5}{3} = 0.5\,.
\end{aligned} \tag{98}$$

The considerable difference stems from the unbalanced training set in which class c_1 occurs much more often than the others.

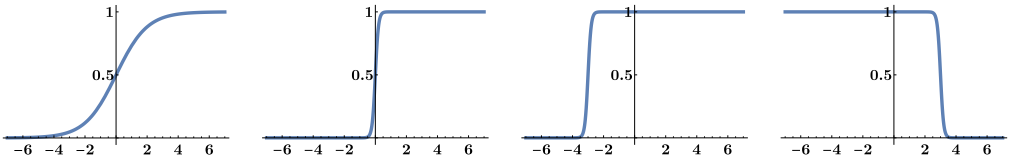
Summary

There are different metrics to analyze the performance of a NN. For regression NNs, one usually computes the distance between the points in the training set and the output of the NN in some metric. For classification tasks, the metric depends on the aim of the classification, e.g. whether one wants to optimize for identifying true positives. ROC, AUC, and the F1 score can be used to identify good thresholds for the decision boundary.

4. Common neural networks

4.1. Generalities: Universal approximation theorem

Before we introduce common neural network architectures, we want to give an intuition for why NNs work at all. After all, they are just consecutive applications of matrix multiplications and some simple function. As a matter of fact, the **universal approximation theorem** [85] proves that a NN can in principle approximate any (bounded continuous) function by using a single hidden layer, but an arbitrary number of nodes. If a NN can indeed learn any function, it can use this function in regression or as a decision boundary to separate feature space into different classes.



(a) $w=1,\ b=0$ (b) $w=10,\ b=0$ (c) $w=10,\ b=30$ (d) $w=-10,\ b=30$

Fig. 20. An illustration of how weights and biases control the shape and position of the logistic sigmoid function.

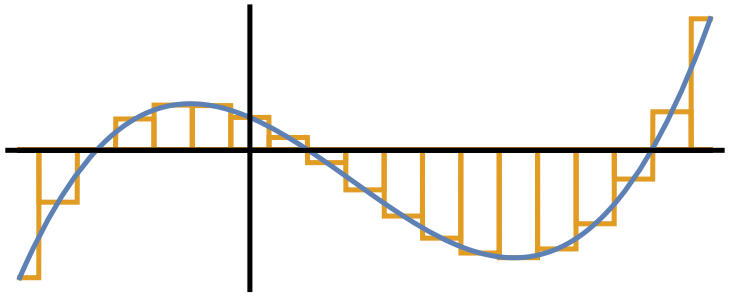


Fig. 21. Approximation of an arbitrary function by a series of step functions. The accuracy of the approximation is increased by including more steps.

Physicists are well acquainted with the idea of representing an arbitrary function using a superposition of functions, as e.g. done in Fourier analysis. The proof of the universal approximation theorem assumes the function is bounded and it uses a superposition of logistic sigmoid functions. To illustrate why this is possible, it is instructive to look at the roles the weight and bias play for a logistic sigmoid activation function [86]. As explained in Section 2.2, a logistic sigmoid layer computes

$$\sigma(wx+b)=\frac{1}{1+e^{-(wx+b)}}\,. \tag{99}$$

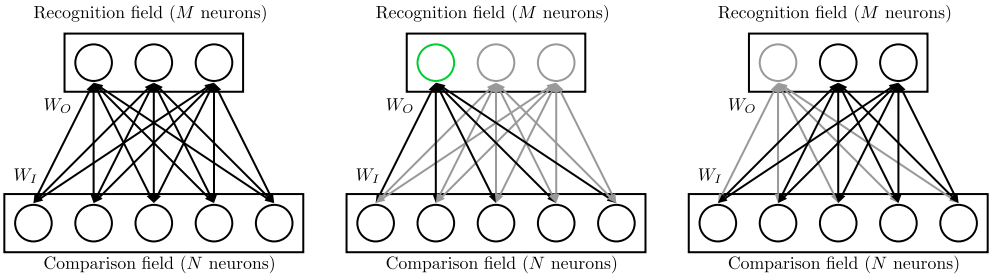
The bias b controls the position and w the shape of the logistic sigmoid function. For positive w, a positive b moves the logistic sigmoid to the left and a negative b moves it to the right. The sign of w controls whether the logistic sigmoid is increasing (positive w) or decreasing (negative w). Lastly, the magnitude of w sets the steepness of the logistic sigmoid. The larger the magnitude, the steeper the logistic sigmoid becomes until it approaches a step function for $|w|\to\infty$. We have plotted $\sigma(wx+b)$ for different weights and biases in Fig. 20.

Intuitively, by subtracting two logistic sigmoid activation functions with very positive or negative weights and whose step is offset by different biases, one can create a function that is zero everywhere except for a small interval where it takes any desired value. The position of this interval can be moved to an arbitrary position by adjusting the biases of the second matrix multiplication (that connects the hidden layer to the output layer), and the height of the step can be adjusted by the weights of the second matrix multiplication. In this way, one can approximate a function by piecewise constant step functions, cf. Fig. 21. Note that this is in general not the most efficient way of approximating a function and also probably not what the NN does; it is just meant to illustrate why the universal approximation theorem holds. Also, just like Fourier decomposition, very simple functions that could be easily expressed in some other function basis, might take a huge number of terms to approximate. For a similar reason, a NN with a single layer and a huge number of nodes is in practice often not a good architecture. Nevertheless, there are interesting theoretical results in the case of infinitely wide NNs as discussed in the next section.

4.2. Classification and regression NN

We have already seen several examples for NNs that can be used as classifiers or regressors for given input features (for images, convolutional neural networks are better suited, see Section 4.4). Simple classification and regression NNs consist of a collection of fully connected layers with some activation function. Regressor NNs usually end with an identity activation (if the co-domain is $\mathbb{R}^T$) or a ReLU (if the co-domain is $\mathbb{R}^T_{\geq 0}$), while classifier NNs usually end with logistic sigmoid or tanh (for binary classifiers), or with a softmax layer for multi-class one-vs-all classification tasks. Often, dropout and batch normalization layers are included in between the fully connected layers. See e.g. Fig. 3 for an example architecture.

Concerning the network topology or architecture, there is no clear cut best design principle. However, recent studies have shown some interesting observations and theoretical results. As explained in Section 3.6, deep, overparametrized



(a) Architecture of an ART NN.

(b) The strongest activated neuron (green) is used, all others are deactivated.

(c) If the match is below the vigilance, repeat for the other neurons.

Fig. 22. An overview of the categorization process of an ART NN.

NNs tend to generalize better. In [87], it was noted that deep NNs trained with SGD lead to better generalization in classification tasks than those trained with adaptive methods such as AdaGrad, ADAM, or RMSProp, even if they perform worse on the training set.

In [41], the authors study the behavior of randomly initialized wide and deep NNs. In this case, the input of the NN is mapped to an output obtained from multiplication and addition of random parameters. In the infinite width limit, the central limit theorem applies, which means that the function computed by the NN is a function drawn from a Gaussian process. By definition, in a Gaussian process any finite collection of random variables has a Gaussian distribution. In the case of infinitely wide NNs with one hidden layer this was described in [88].

In [89], the authors show that the apparently complicated learning process of a NN simplifies considerably for wide NN. In the infinite width limit, the learning process can be approximated by the first term of a Taylor expansion around the initial parameters of the network, i.e. the network behaves like a linear function in its parameters (but of course not in its input).

4.3. Adaptive resonance theory

Adaptive resonance theory (ART) can be used to categorize or cluster input into different, a priori unknown, classes. The implementation described here is an example for unsupervised machine learning, but versions for supervised [90] and reinforcement learning [91] exist as well. While it has been applied in engineering and the medical sector, among others, I am not aware of an application in high energy physics.

As in many instances we have previously seen, ART is again copying principles from nature and how the brain learns and processes information [92]. The idea is that the new input is compared to already known information. If it fits (to an extent determined by a model hyperparameter) into one of the categories that have already been learned, the new input is added to the memory of the corresponding category. If it differs too much from any of the previously seen inputs and does not fit into any known category, a new category is created. The weights of the NN serve as a memory for the information that has previously been seen. If some input resembles previously seen input (i.e. resonates with information already stored in the network) closely enough, the memory (weights) of the corresponding category are updated (adapted) to also remember the features of the new input that is believed to belong to the same category. This process of adapting resonant nodes is responsible for the name ART.

ARTs are realized as bidirectional NNs with two layers: the first is called the comparison field and consists of N neurons and the second is called the recognition field and consists of M neurons, cf. Fig. 22a. There are directed connections from all nodes of the comparison field to all nodes of the recognition field and a second set of directed connections in the other direction, which are given in terms of an $M \times N$ matrix w_I and an $N \times M$ matrix w_O, respectively (note that there are no bias vectors). w_I and w_O correspond to the memory of the machine. The size N of the comparison field matches the number of features encoded in the input vector and the size M matches the maximum number of categories or clusters that the machine can learn to distinguish. Since we have only two layers, we will call the input $\ell^{(0)} = x$, the value at the first layer $\hat{y} = \ell^{(1)}$ with $\hat{y}_\nu = \sum_\mu (w_I)_{\nu\mu} x_\mu$ and the reconstructed input $\hat{x}_\mu = \sum_\nu (w_O)_{\mu\nu} \hat{y}_\nu$.

In the first step, we apply w_I to the input vector. The neuron in the recognition field that receives the strongest signal determines the category into which the input most likely belongs. In order to check how well it fits with previous memories of objects in the same category, we apply the matrix w_O to the value that has been obtained in the previous step while setting the value of all other neurons to zero, cf. Fig. 22b. If the νth node of the recognition field was activated, this step essentially compares how close the weights $(w_O)_{\mu\nu}$ connected with this node match the components x_μ of the input vector $x = \ell^{(0)}$. If the result matches the input within a given tolerance, called vigilance ρ, the input is believed to fit into the previously identified category and the weights are updated proportional to a learning rate α to remember this input. If the mismatch is too large, the entire process is repeated, with the neuron in the recognition field that was previously

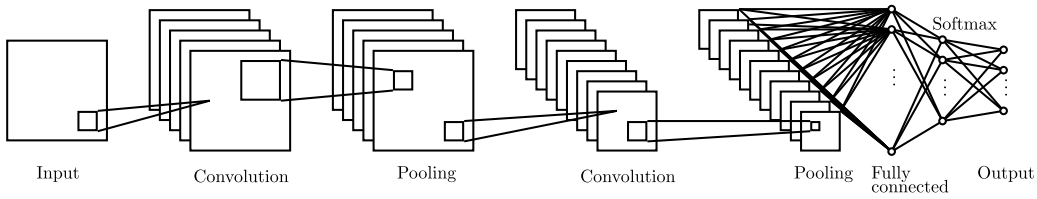


Fig. 23. Typical implementation of a CNN as pioneered in [93].

activated most strongly being deactivated, cf. Fig. 22c. This will lead to a strongest activation of another neuron in the recognition field, whose value is then again multiplied with w_0, compared with the input and the vigilance parameter, and so on.

If none of the neurons produces a match, the input is assigned to a previously unassigned neuron in the recognition field. If all neurons in this layer have been assigned and no match is produced, the capacity of the ART network is exhausted and the input could not be learned and clustered. In that case, one needs to either leave room for more categories M or change the vigilance parameter to fit the input into fewer distinct categories.

Let us describe the process in more detail: First, one chooses a learning rate α and the vigilance parameter ρ such that

$$\alpha > 1\,, \qquad 0 \leq \rho < 1\,. \tag{100}$$

The larger the learning rate, the stronger old information will be overwritten in the learning process. The larger the vigilance, the finer the classification will be, i.e. the more similar the network's input needs to be in order to be considered to belong to the same category. The weights of the network will be initialized as

$$(w_I)_{\nu\mu} = 1\,, \qquad 0 < (w_0)_{\mu\nu} < \frac{\alpha}{\alpha + N - 1}\,. \tag{101}$$

We also assume that the input is normalized, $x_\mu \to x_\mu/\|x\|_2$. In order to find the strongest activated node, we perform the "forward pass", $\hat{y}_\nu = (w_I)_{\nu\mu}x_\mu$ and select the node with the largest $\hat{y}$. Next we perform the "backward pass" (note that, in contrast to NN training discussed in Section 3, a second, independent set of weights w_0 is used), $\hat{x}_\mu = (w_0)_{\mu\nu}\hat{y}_\nu$. Then, we compare how closely the resulting $\hat{x}$ resembles the original input x and compare it to the vigilance ρ. One possibility is to accept the match if

$$\|\hat{x}\|_2/\|x\|_2 < \rho \tag{102}$$

and reject it otherwise. If it is rejected, the next strongest activated node $\hat{y}_\nu$ is selected, $\hat{x}$ is computed and compared to x and so on, until one of the $\hat{y}$ produces an $\hat{x}$ that matches within the tolerance set by the vigilance. If this is not the case for any of the nodes in $\ell^{(1)}$, the input is assigned to a new, previously unused node in $\ell^{(1)}$. The weights for the selected node ν in $\ell^{(1)}_\nu$ are updated as follows:

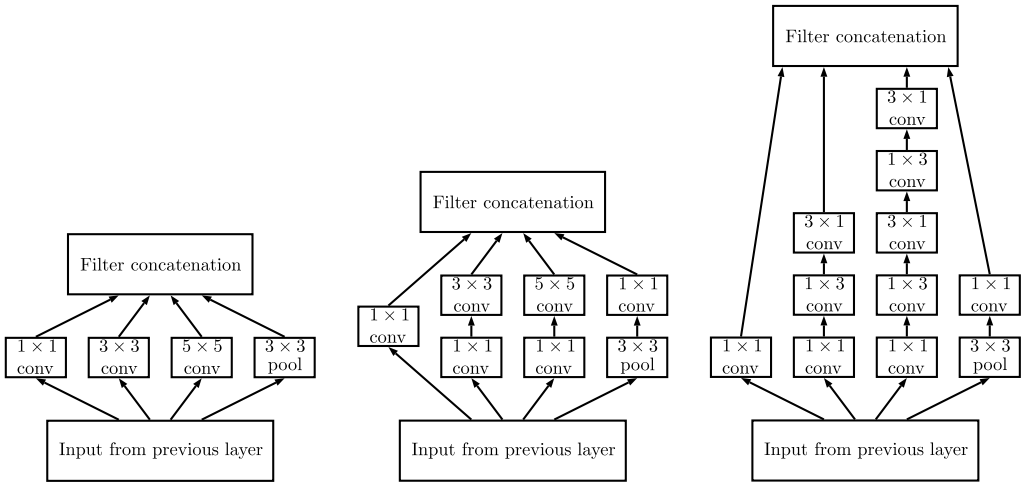
$$(w_I)_{\nu\mu} \to \frac{\alpha\hat{x}_\mu}{\alpha - 1 + \|\hat{x}\|_2}\,, \qquad (w_0)_{\mu\nu} \to \hat{x}_\mu\,. \tag{103}$$

Thus, the memory weights w_I are set to the (normalized) reconstructed input $\hat{x}$ of node $\ell^{(1)}_j$ weighted by the learning rate α, while the memory weights w_0 are simply set to the reconstructed input $\hat{x}$. Note that $\hat{x}$ itself is obtained from multiplication with w_0; in this way the update retains information from previous inputs.

4.4. CNNs, inception networks, deep residual networks

Convolutional Neural Networks (CNNs) are typically used in supervised learning as a classifier for images. We have already discussed the layers of CNNs used in image processing in Section 2.4. Typically, the input is fed through (a series of) convolutions with leaky ReLU followed by max pooling and sometimes a fully connected layer, with dropout layers in between. At the end of the network, there are usually one or two fully connected layers, with the last layer being a softmax layer which produces a vector of probabilities for a picture belonging to a certain class. An early and important implementation of this architecture [93] is known as LeNet after Yann Lecun, see Fig. 23.

Let us discuss the hyperparameter choices for the convolutional and pooling layers in CNNs. As explained in Section 2, the idea of filters is to detect certain defining properties within a picture. Thus, the filter size of the convolutions should be chosen to be as large as the features that are to be discovered. However, these might vary considerably in size. To account for this, **inception modules** perform convolutions with several filter sizes as well as max pooling in parallel and concatenate the result, rather than just using filters of the same size [94]. The full NN is built out of several consecutive inception modules. The authors of [94] propose a filter size of $1\times 1\times d_{\max}$, $3\times 3\times d_{\max}$, $5\times 5\times d_{\max}$, and one $3\times 3\times d_{\max}$ max pooling, see Fig. 24a. Here, the third dimension of the filter extends over the entire depth $d_{\max}$ of the input, which



(a) Basic inception module.

(b) Inception with reduction by 1×1 convolution.

(c) Replacing 5×5 convolutions by two 3×3 convolutions and 3×3 by 1×3 and 3×1.

Fig. 24. Overview of different inception module implementations.

is common in CNNs. For an RGB image, this would be three initially, but after performing convolutions, the depth will be the number of filters. A 1×1 convolution thus flattens the depth in the third dimension to one.

Since convolutions are rather expensive operations, the authors propose to apply an additional 1×1 convolution before performing the 3×3 and 5×5 convolutions, as well as after performing the max pooling, see Fig. 24b. This way, the computational costs of the inception layers are reduced, which makes it feasible to use several of them consecutively (in the paper, the authors use 9, leading to a NN of depth 27). This architecture became known as **GoogLeNet** (a combination of the employer of some of the authors and the LeNet architecture that was used by the authors as a team name in an image classification competition where they entered with this NN architecture), or as **Inception v1** (the name being derived from constructing deep architectures and using a network within a network [95]).

This is a rather deep network so it is prone to suffer from the vanishing gradient problem in the middle layers, as discussed in Section 3.4. The authors combat this by reading out the hidden states after two inception modules in the middle of the network, computing an auxiliary loss based on these, and optimizing the network to minimize the actual loss plus this auxiliary loss (note that this hidden output or auxiliary loss is ignored after training).

As the name Inception v1 suggests, there were further refinements to this architecture. **Inception v2** and **Inception v3** were introduced in [96]. The main changes include performance optimizations. It is for example cheaper to perform two 3×3 convolutions as compared to one 5×5 convolution. Furthermore, $n \times n$ convolutions can be factorized into $1 \times n$ and $n \times 1$ convolutions, which is also faster, cf. Fig. 24c. In v3, the authors include in addition (factorized) 7×7 convolutions, batch normalization, label smoothing (this is a regularizer in the loss function to prevent overfitting, see Section 3.3.1), as well as the use of an RMSProp optimizer (see Section 3.3.4).

There were further improvements to the inception modules [97]. In **Inception v4**, the initial part of the network, i.e. the layers that are between the input and the first inception module, were modified, the inception modules slightly changed, and reduction blocks were added. Another extension, called **Inception-ResNet** and also introduced in [97], combined inception with **deep residual NNs** [98]. In the latter, a convolution is performed on some input and the result is added to the original input. This requires the dimensions of the original input to match the dimension of the result of the convolution operation, which can be achieved for example by using $1 \times 1 \times d_{\text{max}}$ filters to flatten the depth. The name ResNet indicates that the original input is kept as a residual through the convolution (or inception) operation. Deep ResNets were found to train better and be more accurate than conventional networks. Inception networks are very popular, and many complete implementations (sometimes even including pretrained NNs) exist for various NN libraries.

In order to get an idea of the performance improvement, we compare the error rates for the different networks in Fig. 25. We show the results of the winners of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) over the years. The challenge contains different contests and has somewhat modified the images over the years (for a comprehensive study of different architectures see [99]). We look at the top-1 and top-5 errors for a single network classification task. Top-1 means that the CNN has to predict the correct class label (out of 1000) for the image in order to be correct. For top-5 errors, the network outputs the five most likely class labels. If the true label is in this top 5, the network prediction is counted as being correct. The error values for the inception networks (winner 2016) and ResNet

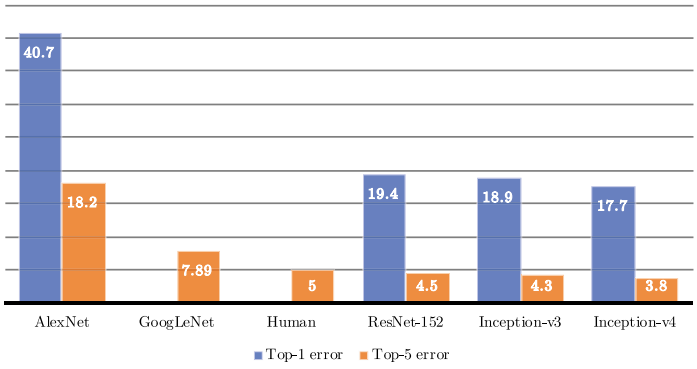


Fig. 25. Comparison of the classification errors of different CNN architectures for the ILSVRC classification challenge.

(winner 2015) are taken from [97]. For humans [100] and GoogLeNet [94] (winner 2014), only the top-5 errors were given. Human errors arise because either some images have been misclassified in the database or because humans disagree on what the image is primarily showing. I am not aware of results for LeNet on this data set, which is why results for AlexNet [101] (winner 2012), which is an improved version of LeNet, are given.

4.5. Boltzmann machines and deep belief networks

Restricted Boltzmann Machines (RBMs) [102] can be used to generate data, much like the more modern generative methods such as generative adversarial networks (see Section 4.7) or variational autoencoders (see Section 4.8). However, they have also been applied in other areas of unsupervised and supervised learning such as feature extraction [103], classification [104], or to initialize deep feed-forward NNs. Moreover, their setup allows for theoretical investigations using tools familiar from Physics, as we shall see below.

In order to explain the name, we first need to briefly introduce **Boltzmann machines** (BM); however, we will mostly be dealing with restricted BM in the following. A BM is a set of nodes $\{\ell\}$ together with undirected connections w between all nodes and biases b for each node. In the standard version, each node can take two values, either on or off, i.e. $\ell_\mu \in \{0, 1\}$. Such a BM is equivalent to the Ising model whose nearest neighbor interactions are given by w, the external magnetic field (times the magnetic moment) is given by b, and the spins pointing up or down are encoded in $\ell_\mu \in \{0, 1\}$. The energy or Hamiltonian for the BM is

$$E(\ell) = \sum_{\mu<\nu} w_{\mu\nu}\ell_\mu\ell_\nu + \sum_\nu b_\nu\ell_\nu \,. \tag{104}$$

In this case the weight matrix is symmetric and has zeros along the diagonal. The probability distribution (i.e. the configuration probability) of a given configuration ℓ is then

$$P(\ell) = \frac{1}{Z}e^{-E(\ell)}\,, \qquad Z = \sum_\ell e^{-E(\ell)}\,, \tag{105}$$

where the normalization factor Z, whose inclusion ensures that the probability is in $[0, 1]$, is the partition function of the system. In particular, the probability that the μth node is on is

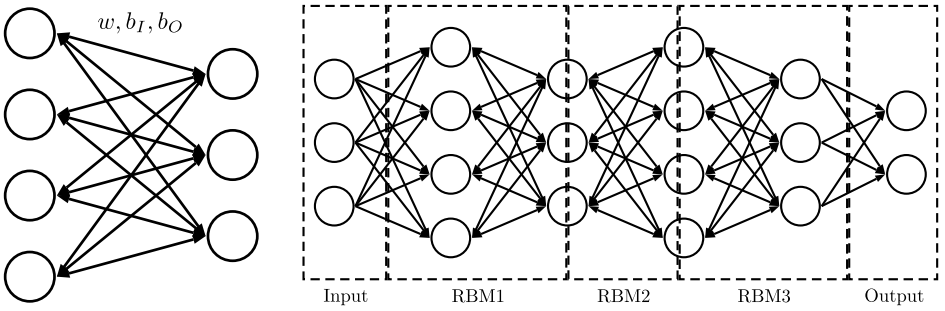
$$P(\ell_\mu = 1) = \frac{1}{1 + e^{-\Delta E_\mu}}\,, \tag{106}$$

i.e. a logistic sigmoid function. The energy difference ΔE_μ for flipping the μth spin (or turning the μth node on vs. off) is

$$\Delta E_\mu = E(\ell_\mu = 0) - E(\ell_\mu = 1) = \sum_{\nu>\mu} w_{\mu\nu}\ell_\nu + \sum_{\nu<\mu} w_{\nu\mu}\ell_\nu + b_\mu \tag{107}$$

For **restricted Boltzmann machines**, the set of nodes are divided into two sets $\{\ell^{(0)}\}$ and $\{\ell^{(1)}\}$. While each element of the first set still shares a weighted connection with each element of the second set and vice versa (i.e. the connection is still undirected), elements from the same set are not connected. In this way, we obtain a two-layered, bidirectional NN. In this case, the energy defined in (104) reads

$$E(\ell^{(0)}, \ell^{(1)}) = -\sum_{\mu,\nu} \ell^{(1)}_\mu w_{\mu\nu}\ell^{(0)}_\nu - \sum_\nu b^{(0)}_\nu\ell^{(0)}_\nu - \sum_\mu b^{(1)}_\mu\ell^{(1)}_\mu \,. \tag{108}$$



(a) A Restricted Boltzmann Machine.

(b) A Deep Belief Network.

Fig. 26. An example for a restricted Boltzmann machine (RBMs) and several stacked RBMs to form a deep belief network.

Assuming that the hidden states are actually hidden (i.e. not measurable or observable), we need to marginalize (i.e. sum) them in order to get the probability of a visible state $\ell^{(0)}$,

$$P(\ell^{(0)}) = \sum_{\ell^{(1)}\in\{0,1\}} \frac{1}{Z} e^{-E(\ell^{(0)},\ell^{(1)})} . \tag{109}$$

Since there is no inter-connection between the nodes of each layer, the hidden unit activations are mutually independent, as are the visible layer activations. In a 2-layer RBM with n_0 input and n_1 hidden nodes, the conditional probability of observing a configuration of visible units $\ell^{(0)}$, given a configuration of hidden units $\ell^{(1)}$, is then

$$P(\ell^{(0)}|\ell^{(1)}) = \prod_{\nu=1}^{n_0} P(\ell_\nu^{(0)}|\ell^{(1)}) . \tag{110}$$

Likewise, the probability for some hidden configuration $\ell^{(1)}$ given a visible configuration $\ell^{(0)}$ is

$$P(\ell^{(1)}|\ell^{(0)}) = \prod_{\mu=1}^{n_1} P(\ell_\mu^{(1)}|\ell^{(0)}) . \tag{111}$$

The individual activation probability for the two cases is simply

$$\begin{aligned} P(\ell_\nu^{(0)} = 1|\ell^{(1)}) &= \sigma\left(\sum_{\mu=1}^{n_1} \ell_\mu^{(1)} w_{\mu\nu} + b_\nu^{(0)}\right) , \\ P(\ell_\mu^{(1)} = 1|\ell^{(0)}) &= \sigma\left(\sum_{\nu=1}^{n_0} \ell_\nu^{(0)} w_{\mu\nu} + b_\mu^{(0)}\right) , \end{aligned} \tag{112}$$

where $\sigma(\cdot)$ is the logistic sigmoid, cf. (106).

Note that, unlike the ART network discussed in Section 4.3, the connections of an RBM use the same weights w, but different biases b_I and b_O, see Fig. 26a. The forward pass of the RBM (or any NN, for that matter) can be interpreted as using the input $x = \ell^{(0)}$ at the first layer to make predictions for the node activation $\ell^{(1)} = a^{(1)}(w \cdot x + b_I)$ of the second layer, given a set of weights w, i.e. $p(\ell^{(1)}|x; w)$. Similarly, the backward pass of the RBM can be interpreted as estimating the probability of a reconstructed input $\hat{x} = w^T \cdot \ell^{(1)} + b_O$ given some activation $\ell^{(1)}$ under the same weights w, $p(x|\ell^{(1)}; w)$. Together, this gives a joint probability distribution $p(x, \ell^{(1)})$ of inputs x and activations $\ell^{(1)}$. Often, the input is binary data, i.e. $\ell^{(0)} \in \{0, 1\}^{\otimes n_0}$ and the activation function is the logistic sigmoid function. As in other generative methods, the difference of the actual distribution and the distribution of the RBM can be computed using the KL divergence (79). The weights are updated to minimize the KL loss. In this way, the reconstructed input $\hat{x}$ computed from the activation $\ell^{(1)}$ will come from the same distribution as the actual input x.

In the case of binary data and logistic sigmoid activation function, the RBM can be sampled iteratively as follows. Start with some input vector $x^0 = \ell^{(0)}$. At the kth iteration, compute

$$\begin{aligned} z^{k+1} &= \begin{cases} 1 & \text{with probability } \sigma(w \cdot x^k + b_I) \\ 0 & \text{with probability } 1 - \sigma(w \cdot x^k + b_I) \end{cases} , \\ x^{k+1} &= \begin{cases} 1 & \text{with probability } \sigma(w^T \cdot z^{k+1} + b_O) \\ 0 & \text{with probability } 1 - \sigma(w^T \cdot z^{k+1} + b_O) . \end{cases} \end{aligned} \tag{113}$$

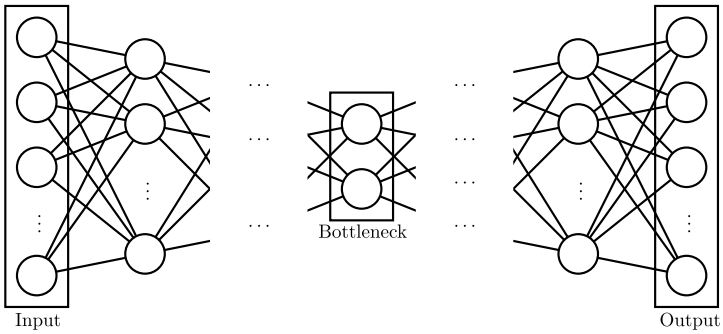


Fig. 27. Example architecture for an autoencoder.

This is sometimes called **Gibbs sampling**. In training RBMs, we would have to iterate x^k until convergence for each input. Since this is prohibitively expensive, one uses ***k*-Contrastive Divergence** (CD-k), which means that the sample is not obtained upon convergence, but is taken after k iterations.

In practice, $k = 1$ already works very well, i.e. two forward–backward-passes (strictly speaking, the second backward pass is not even used in the weight update). So, we start with x^0, compute z^1 from the forward pass, x^1 in the backward pass, and the corresponding z^2 in the second forward pass. For the weight update, one compares the association (activations) z^1 caused by the actual input x^0 with the association z^2 caused by the reconstructed input x^1. In more detail, one defines for each connecting edge $e_{\mu\nu}$ between node $\ell_\nu^{(0)}$ and $\ell_\mu^{(1)}$ the quantities

$$\text{pos}(e_{\mu\nu}) = x_\mu^0 z_\nu^1\,, \qquad \text{neg}(e_{\mu\nu}) = x_\mu^1 z_\nu^2\,. \tag{114}$$

Note that $\text{pos}(e_{\mu\nu}) = 1$ if both x_μ^0 and z_ν^1 are one. This thus measures the association between nodes which we want the network to learn. Likewise, $\text{neg}(e_{\mu\nu})$ measures the association between nodes of the network based on its own reconstructed input. The weight update is given by the difference between the two associations,

$$w_{ij} \to w_{\mu\nu} + \alpha(\text{pos}(e_{\mu\nu}) - \text{neg}(e_{\mu\nu}))\,, \tag{115}$$

where α is the learning rate.

Deep Belief Networks (DBNs) [103] or deep Boltzmann machines are networks obtained from stacking RBMs, see Fig. 26b. Using the procedure outlined above, DBNs can be pretrained layer-wise in an unsupervised fashion (aka their weights can be initialized in a clever way). In **greedy layer-wise unsupervised pretraining**, one does not use the actual output for training, but just the input. This is fed into the first layer and iterated back and forth until the value $\hat{\ell}^{(0)} = (w^{(1)})^T \ell^{(1)} + b_0$ reconstructed from the activation $\ell^{(1)} = w^{(1)} \cdot \ell^{(0)} + b_l$ by backpropagation approximates the original input $\ell^{(0)}$. Once this is achieved, the same procedure is repeated for the weights $w^{(2)}$ of the next layer $\ell^{(2)}$ with $\ell^{(1)}$ as input, and so on. The idea is that the weights after pretraining already approximate very well the distribution of the input, which provides a promising starting point for minimizing the loss with respect to the actual output in supervised training. After this pretraining phase, there can be a fine-tuning phase in supervised learning applications, where the DBN is treated like a feed-forward NN and is trained with input–output pairs.

4.6. Autoencoders

As the name suggests, autoencoders learn to encode input. They are given a feature vector as input and are trained to reproduce the exact same vector as output, i.e. to learn the identity function. However, the NN is designed such that the information has to propagate through a deep NN whose intermediate layer(s) contain fewer nodes than the input and output layer, cf. Fig. 27. Often, the NN architecture of autoencoders is symmetric, with the middle layer being the smallest one, i.e. the **bottleneck** layer which the information has to pass through. Since the presence of the bottleneck means that the NN has to learn to reproduce the input with far fewer features available, it is trained to efficiently encode the information content of the feature space. Ultimately, one is not interested in the output (since this is just the input, i.e. known data), but in the (compressed) data at the bottleneck layer. This hidden state of the NN is read out and processed/used in further applications.

The types of activation functions used in autoencoders depend on the features at hand. If the input is an image, the first part of the NN before the bottleneck can be chosen to be a typical CNN architecture, i.e. CNN layers followed by leaky ReLU and max pooling, while the second part can consist of the inverse operations, i.e. deconvolutions. In other cases, the activation functions might be chosen to be the same throughout the autoencoder (e.g. logistic sigmoid or tanh), but the number of nodes decreases on the left-hand side, thus funneling the input towards the bottleneck, and then increases again after the bottleneck, thus fanning out the compressed data.

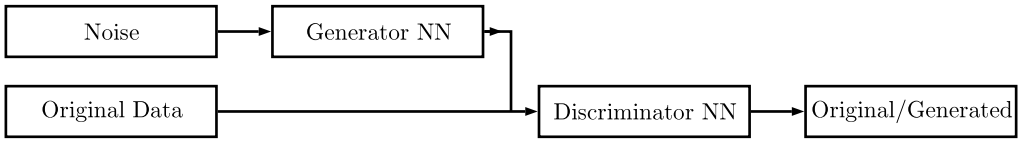


Fig. 28. Overview of a GAN. The generator tries to generate data that cannot be discerned from original data by the discriminator network.

The NN architecture, especially the size of the bottleneck, is usually found by trial and error. The ability of the NN to reconstruct the input will decrease with decreasing bottleneck size. The threshold at which one decides that too much information is lost in the bottleneck and one cannot further reduce its size, depends on the application and how well the encoding in the bottleneck needs to be. The data from the bottleneck layer can e.g. be used as input to other NNs, to identify clustering of models, or it can be analyzed further using for example decision trees, persistent homology, or other data analysis tools.

4.7. Generative adversarial networks

Generative adversarial networks (GANs), see [105] for the original paper and [106] for a nice overview, can be used to generate data that is similar (in an appropriate sense) to a set of input data. The idea of GANs is to have two NNs compete against each other, see Fig. 28. The **generator network** generates data from random noise and the **discriminator network** is trained to tell generated from original data. In this way, the generator is trained with the aim to generate data that cannot be discerned from real data by the discriminator network, while the discriminator network is trained to recognize data that is different from the original input. This causes the generated data to resemble more and more the real data. For example, when using GANs for image generation, the generator is often implemented using deconvolution layers and the discriminator using convolution layers.

The game that is played between the generator and the discriminator is a zero-sum non-cooperative game. If the discriminator wins (identifies data from the generator as generated), the generator loses and vice versa. Both networks try to minimize their losses individually, or equivalently, try to minimize the actions that the opponent tries to maximize. This is why such games are called **minmax** games. They are an example for an **actor–critic** model, in which the actor (generator) changes its behavior based on the critic (discriminator) and vice versa. We will discuss actor-critics in more detail in Section 8 in the context of reinforcement learning. In [107], it is explained that GANs can be thought of as actor-critics in an environment where the actor cannot affect the reward. The minmax game ends in a **Nash equilibrium**, in which no NN can improve without changing the parameters of the others (which it cannot). However, also loss functions other than minmax have been discussed for GANs, see [108,109].

Let us describe GANs in more detail. We define z to be the noisy input sampled from some distribution $p_z(z)$ (often normal or uniform distributions are used). The generator then maps z to generated data $G(z)$, while the discriminator D maps data x (real and generated) to $[0, 1]$ indicating the probability of the data being **original** (values close to 1) or **generated** (values close to 0). The loss function for the discriminator of a minmax-GAN (MM-GAN) [105] is thus

$$L_D^{\text{MM-GAN}} = -(\mathbb{E}_{x\sim p_{\text{data}}(x)}[\log(D(x))] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))])\,. \tag{116}$$

The overall minus sign simply turns the value function into a loss function. The first term in the sum is the log probability of the discriminator D predicting that real data is indeed real, while the second term is the log probability of D predicting that the generator's data $G(z)$ is indeed generated. So minimizing this loss means maximizing the accuracy of the discriminator predictions. Similarly, the loss function of the generator is

$$L_G^{\text{MM-GAN}} = \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]\,. \tag{117}$$

With this loss function, the generator NN is trained to generate data that has a low probability of being identified as generated (note the different sign as compared to the second term of (116)). In a variant known as non-saturating GANs (NS-GANs), the generator uses a different loss function,

$$L_G^{\text{NS-GAN}} = -\mathbb{E}_{z\sim p_z(z)}[\log(D(G(z)))]\,, \tag{118}$$

which trains the generator NN to generate data that has a high probability of being judged to be real data. NS-GANs often outperform MM-GANs. At the optimum, the generator has learned to produce data that is indistinguishable from real data, $p_{\text{data}} = p_{\text{generated}}$ and hence $D(x) = 0.5$. How much the generated and the actual data distributions differ can be quantified with different measures such as the Kullback–Leibler divergence [73], the Jensen–Shannon divergence [110], the Wasserstein distance, etc. There is a vast literature on different possible GAN loss functions and the corresponding differences in probability distributions. Each has certain advantages or disadvantages, see [108] for a discussion.

Unfortunately, it is quite challenging to construct and train the generator and discriminator in a way that ensures a fair competition between the networks. The problem is that if the discriminator becomes too powerful, the gradient vanishes for the generator and the generator cannot learn how to improve. If the discriminator is too week, two things can happen:

either, it is essentially randomly guessing and thus not providing any information for the generator how to improve, or the generator learns a weakness of the discriminator. The latter can lead to adversarial attacks as described in Section 4.4, where the generator exploits a weakness of the discriminator.

The problem of a vanishing gradient for a too powerful discriminator can be ameliorated by using Wasserstein GANs [111,112]. This can be understood as follows: often, the output dimension of the generator is much larger than the random noise input. This makes the high-dimensional output a (complicated) function of comparatively little input data. Likewise, in real-world examples, the real input data lies also on a manifold of very high codimension in the output space. This means in practice that the probability distributions of real and fake data have disjoint support (i.e. they do not intersect), which makes it (in theory) possible to device a discriminator that can distinguish both with extremely high accuracy. The problem manifests in the loss since the KL divergence of probability distributions with disjoint support is infinity, while the JS divergence jumps discontinuously once two formerly disjoint distributions start to overlap. In contrast, the Wasserstein distance is well-defined and continuous, thus providing sensible gradients for the weight updates.

Another problem that can occur for GANs is **mode collapse**, where the generator only generates data of a certain class (the one where it can most easily trick the discriminator) and hence the generator output becomes rather independent from the random noise input. This can happen if the discriminator is too weak or too powerful. Furthermore, it can happen that, during training, the parameters of the generator and discriminator (which depend on one another) oscillate but never converge to an acceptable solution.

Identifying mode collapse or failure of convergence is not always easy. In the case where the GAN outputs images, visual similarity of the output can be used to identify mode collapse. In more general cases, it can help to look at the loss of the generator and discriminator. In the case of mode collapse, the generator will keep producing the same images. If the generator identified a weakness or adversarial attack against the discriminator, the discriminator loss will stay high while the generator loss will decrease continuously. In cases where the discriminator can adapt to the adversarial attack, but the generator still suffers from mode collapse, an oscillatory pattern in the generator loss is expected: the generator keeps generating the same type of data, which is initially successful and its loss is low. Over time, the discriminator learns to distinguish this data from real data, which means that the generator loss increases until the generator switches to generating a new set of (almost identical) data that tricks the discriminator. As a consequence, the discriminator loss jumps up and the generator loss jumps down, and the process is repeated. In the case of failure of convergence, it is likely that the discriminator is too powerful, which is reflected in its loss being almost zero.

To tackle these problems, GANs are often closely monitored during training. Furthermore, tuning of hyperparameters and NN architecture can ensure a balanced competition between the NNs. Sometimes the discriminator is pretrained; if both networks start from scratch, the discriminator first needs to learn to classify the data before it can provide a serious competitor to the generator. Other methods include dynamically adjusting the learning rate of both networks during training.

4.8. Variational autoencoders

Like GANs, variational autoencoders can be used to generate data, see [113,114] for the original references and [115] for a review. In fact, the second part of an autoencoder, i.e. the part behind the bottleneck layer, generates data from an input (the bottleneck), just like the generator NN of a GAN. However, in contrast to the GAN generator, which generates the data from random noise, the autoencoder generates data from the bottleneck data. The idea is that bottleneck data, i.e. the input to the generative part of the autoencoder, encodes properties of the data it was trained with, rather than just random noise. So, input with similar features (e.g. input belonging to the same class) should be mapped to similar data at the bottleneck layer by the autoencoder, and hence be decoded to similar data. Thus, the type of output can be controlled more systematically by changing the input, in contrast to GANs. Moreover, the fact that the bottleneck data actually encodes properties of the input data should lead to more realistic output as compared to data generated from random noise.

At first, this seems of little use since, in order to use an autoencoder to generate data of a specific kind, we need to know the encoding at the bottleneck. This problem is solved by training the autoencoder to encode the bottleneck data in a unit Gaussian distribution (i.e. with zero mean and unit variance). This constraint distinguishes variational autoencoders (VAE) from regular autoencoders. Then, we can simply draw from a Gaussian distribution in order to produce data with the desired features.

In practice, this means that the VAE's loss consists of two parts: one part is the same as the loss used in regular autoencoders (so for example the mean square error (64) between the output and the input of the autoencoder), while the other part is the difference between the distribution of the data at the bottleneck layer and a unit Gaussian. As discussed in Section 3.5, the difference between distributions can be measured by the KL loss (79), so we can set the loss schematically to

$$L^{\text{VAE}} = L^{\text{MSE}}(x, \hat{y}) + L^{\text{KL}}(\text{bottleneck},\ \mathcal{N}(0,1))\,. \tag{119}$$

In order to apply backpropagation to (119), we need to use what is called the reparametrization trick [113,115]. The problem is that the act of drawing from $\mathcal{N}(0,1)$ is random and discrete and thus has no gradient. In the reparametrization

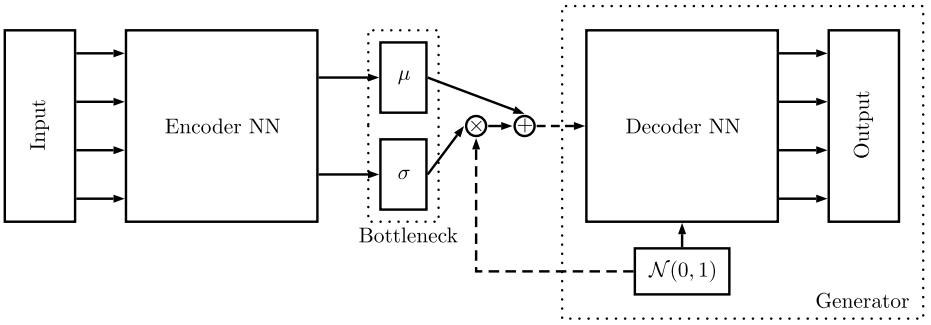


Fig. 29. Schematic architecture for a variational autoencoder. During training, random draws from $\mathcal{N}(0, 1)$ are provided as input to the bottleneck as a reference distribution to apply the KL divergence to. After training, when used as a generator, the dashed arrows are discarded and $\mathcal{N}(0, 1)$ is the only input to the decoder/generator.

trick, the value drawn from $\mathcal{N}(0, 1)$ is treated as an additional input. The loss can then be backpropagated such that the MSE between the output and the input data as well as the KL divergence between the bottleneck layer and $\mathcal{N}(0, 1)$ is minimized. By repeating this in batches for many input data and many random draws from $\mathcal{N}(0, 1)$, the VAE learns to encode the input data following a unit Gaussian distribution.

In more detail, we set up the encoder part of the VAE such that it produces a pair of vectors, one encoding the mean μ and one encoding the standard deviation σ. The decoder part is set up such that it expects a vector of random numbers (drawn from a unit Gaussian) and outputs data in the format of the input, see Fig. 29. During training, we now proceed as follows: first, the encoder computes (μ, σ) from the input. Then, we draw a random element from $\mathcal{N}(0, 1)$, multiply this with σ and add μ. This is then fed to the decoder. After training, in the generation phase, the encoder is discarded and the decoder is simply fed with a randomly drawn vector from $\mathcal{N}(0, 1)$.

5. Genetic algorithms

Genetic algorithms (GAs) [116] are a class of search or optimization algorithms designed to find solutions that maximize a function called the **fitness function**. They have been applied a few times in the past to address problems in theoretical physics and string theory [3,117–123]. Often, GAs are not considered machine learning algorithms since they do not learn anything; in contrast to neural networks, GAs do not have any parameters[9] that are adapted or trained. Since they are optimization algorithms, they can, however, be used for example to train NNs instead of using gradient descent and backpropagation. Other applications for GAs to NNs include optimizing the NN architecture, the NN training set, etc. See [3] for an example application of GAs to NNs in string theory.

Genetic algorithms are based on Darwin's principle of survival of the fittest. The basic idea is that, given an initial population (guess for a solution), the fitness of the individuals of the population is evaluated with respect to the fitness function we are trying to maximize. The fittest individuals get to reproduce (i.e. the properties of their offspring are derived from their own properties), followed by a mutation stage where properties are randomly changed. Then the whole procedure is repeated for the new generation. In order to prevent exponential growth of the population, individuals are removed from the population, either based on their fitness or on their age.

To this day, there is some controversy [124] as to why GAs actually work. An often quoted explanation [116,125,126] called the "fundamental theorem of genetic algorithms" proposes that GAs evolve exponentially many schemas (schemas define a set of individuals who have a certain subset of properties) with above-average fitness in successive generations. However, the evolution of schemas happens in situations where GAs perform very well as well as in situations where they perform poorly [124], thus casting doubt on schemas actually explaining why GAs work so well. For this reason, we will not go into details of schemas and content ourselves with the observation that GAs do work well, irrespective of what the explanation for why they work well will eventually be.

5.1. Basic vocabulary

Let us introduce next some vocabulary used in the GA literature. Since GAs are based on evolution in biology, the terminology is largely borrowed from this field. For a textbook introduction to GAs see for example [116].

An **individual** or **creature** or **phenotype** e is an ordered set of properties that approximate the solution to the problem we are trying to solve,

$$e = \{a_1, a_2, \ldots, a_n\} . \tag{120}$$

[9] They do have hyperparameters, i.e. constants that are set at compile time. In NNs, these hyperparameters correspond e.g. to the learning rate.

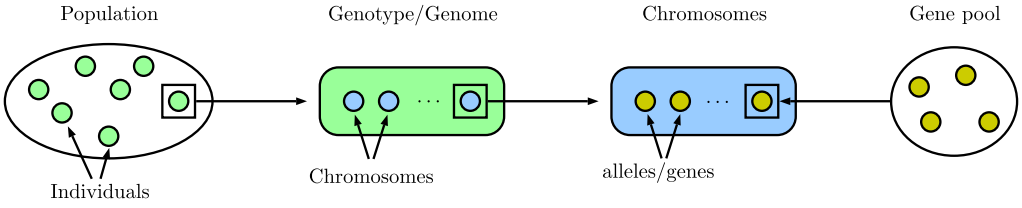


Fig. 30. Overview of the terminology used in GAs.

For example, if we wish to maximize a single real scalar function in one variable, an individual would just be a real number that consists of a collection of digits. Due to finite precision when representing a number on a computer, the real number will be characterized by a property that defines the sign, one that defines the position of the decimal point, and finitely many digits. The set P of all individuals is called a **population**, $P = \{e_i\}$. In each evolutionary step t, a new **generation** G_{t+1} is created from the last generation G_t, thus changing the population P.

The defining collection of properties of an individual, i.e. the sequence of a_i in (120), is called the **chromosome** or **genotype** or **genome**. The building blocks of the chromosome, i.e. each individual a_i, is called an **allele**. In biology, alleles are alternative forms of a gene that reside at the same position in a chromosome. The position of the allele in the chromosome, i.e. the index i in (120), is called the **locus** of the allele.

While many of the words introduced here are often used interchangeably it can be beneficial to discriminate between them, depending on the (nested) complexity of the individuals. For example, the authors of [122] distinguish between genotype and chromosomes: if the function to maximize depends on two variables, $f = f(x, y)$, the variables x and y are called chromosomes and the collection (x, y) is called the genotype. In [3], the function to maximize is a function of semantically different "variables". Hence, there the genome consists of various functional units (genes or alleles) of different complexity. The set of available genes, i.e. the domain of the a_i, is called the **gene pool** γ. We summarize the structure in Fig. 30.

The function f to be maximized is called the **fitness function**. Since GAs are based on survival of the fittest, the fittest individuals get to reproduce by passing on some of their genes, while the unfittest die out when going from one generation to the next. There are several possibilities for reproduction and survival selection, which we discuss below.

Once the new generation has been produced from the genes of the old ones, some alleles (or genes or chromosomes) are **mutated**, i.e. randomly changed and substituted by another allele. This replacement is similar to gene splicing in that the old gene is removed from the genome and replaced with a different gene from the gene pool. In the simplest case, in which all genes or alleles are of the same type, this does not change the length of the genome, while in other cases it does.

5.2. The fitness function

In optimization applications we usually want to maximize or minimize a given objective function g. Of course, the two problems are equivalent by just flipping the overall sign. Since we are talking about fitness, it appears natural to formulate the problem as a maximization problem of the objective function and set the objective function g equal to the fitness function f. However, some GAs work under the assumption that the fitness function is positive, while the objective function is in general not.

Let us from now on assume our task is to maximize the objective function (if we want to minimize, just replace $g \to -g$). A positive fitness for the entire current population can be ensured by shifting the objective function g up by a constant c and using this as a fitness function, $f = g + c$. In applications where the minimum of g is known (and negative), we can just choose $c = |g_{\min}|$. In other cases, we can choose the shift $c \geq |\min_i f(e_i)|$, i.e. larger than or equal to the absolute value of the fitness of the least fit individual e_i. Note that this means that the fitness function can change every generation. Alternatively, one can initialize c with some number and shift by this number until a more negative fitness requires a larger shift.

In search applications, one usually has a set of criteria which the solution has to satisfy simultaneously. In such cases, it is useful to write the criteria as functions $f_a = 0$. Bounds on certain parameters of the problem can for example be realized by step functions that assign 0 to the allowed interval and 1 else. One then minimizes the sum of the (absolute value) squares of the individual criteria f_a,

$$f = \sum_a c_a |f_a|^2 \,. \tag{121}$$

The function f will be minimized iff all $f_a = 0$, which means a solution has been found. The coefficients c_a can be used to weight the priority with which the individual functions f_a are taken into account in the overall minimization. Lastly, we want to point out that the fitness of individuals might not be given explicitly but rather depend on how well the other individuals perform. If one wanted e.g. to evolve good chess players, the fittest individual will be the one that can defeat most others from its population.

5.3. Selection for reproduction

After the fitness function has been defined, an initial population is formed at random. The fitness of each individual is evaluated and the fittest ones get to reproduce. There are many conceivable options for how fitness influences the reproduction stage. Depending on how strongly the fitness influences the reproduction probability, the evolutionary pressure on the individuals is higher or lower. If the fitness influences reproduction too weakly, the system is converging very slowly; if the pressure is too high, the system will converge prematurely to a local optimum. We will discuss different selection mechanisms in the following, see e.g. [127,128].

A common choice for breeding is **fitness proportionate selection** or **roulette wheel selection**. The name stems from the fact that selecting an individual for breeding with this algorithm is like betting on a ball to end in a certain sector of a roulette wheel, where the size of the sector is proportional to the fitness of the individual. In more detail, let us denote the total number of individuals in any generation by N and the fitness of the ith individual by F_i. Let us furthermore assume, as explained above, that the fitness is positive and define the sequence

$$S_j = \sum_{i=1}^{j} F_i \,. \tag{122}$$

Note that the distance $d_j = S_j - S_{j-1}$ between two successive elements just equals the fitness of individual e_j (this corresponds to segmenting the roulette wheel). We now generate a random number $R \in [0, S_N]$ (this corresponds to the ball landing in a random segment of the roulette wheel). Since $F_i > 0$, there exists a unique j such that $S_{j-1} \leq R < S_j$; this individual e_j is selected for reproduction.

A slight variation of this algorithm is known as **stochastic universal sampling**. Instead of spinning the wheel K times (drawing K times a single random number R), one can draw a single random number R and then select individuals e_j by producing K evenly spaced intervals

$$s_k \equiv (R + k/S_N) \bmod S_N \,, \qquad k = 0, 1, \ldots, K-1 \tag{123}$$

and finding the corresponding K individuals e_j with $S_{j-1} \leq s_k < S_j$. This increases the chance of the fittest individual being selected at least once, but also of less fit individuals to be selected. It can produce better results for rather unbalanced roulette wheels.

Still, the problem with the type of selection explained above is that the reproduction probability is given directly by the fitness. First, this is problematic if one individual develops that is much fitter than all others. This often happens at early stages, where one specific individual randomly happens to be much better than the average. This individual will then be selected for reproduction so frequently that it passes on its genes to almost all individuals of the next generation. This is the analog of inbreeding and will cause later generations to degenerate. If, on the other hand, most individuals have converged towards the solution of the optimization problem, as is usually the case in the later generations, all individuals have essentially the same fitness. Basing reproduction on fitness then means that those individuals that are closer to the actual solution do not have an advantage over the others, thus hindering further conversion. This is the analog of removing the evolutionary pressure from a population.

In order to circumvent this problem, one can use **fitness rescaling**. In linear rescaling, one defines the rescaled fitness f' as $f' = af + b$. While there are many choices for the slope a and the offset b, one usually wants to ensure that the average fitness before and after scaling stays the same, such that an individual with average fitness produces exactly one offspring. To fix the slope and intercept of the line f', one can fix a second point by e.g. demanding that the fittest individual generates k times more offspring than the average, $f'_{\max} = kf_{\max}$. In practice, one often chooses $k \in [1.2, 2]$. However, if there are a few individuals that are much less fit than the average, the rescaled fitness can become negative. To prevent this, one can simply cut off the negative values and set the rescaled fitness to zero.

Another possibility for selection is to use **tournament selection**. In k-way tournament selection, k individuals are chosen at random to compete against each other. The ones that win most often are deemed fittest and get to reproduce. This has the advantage that it is easy to implement and fast. Furthermore, it works with negative fitness values. Since the k participants of the tournament are chosen randomly, the selection of participants in the tournament is not proportional to the fitness of the individual and hence does not suffer from too small or too large evolutionary pressure, as can happen for the roulette wheel algorithms. Moreover, tournament selection can be used in cases without a globally defined fitness function, for example when evolving game players whose fitness is measured with respect to performance against other individuals that play the same game.

There are again several variations of tournament selection. First, the parameter k controls how high the evolutionary pressure is. The larger the k, the less likely are less fit individuals to win a tournament round. The case $k = 2$ is known as binary tournament selection. Second, there is the choice of whether or not the same individual can compete in multiple tournaments or whether the winner does not get to compete again. In the former case, the evolutionary pressure is again higher. Lastly, the tournament can be played probabilistically or non-probabilistically. In the latter case, the fittest individual wins for sure, while in the former case it wins with some probability p, which can be taken to be a function of the fitness of the individual. Again, the former case leads to a higher evolutionary pressure.

Another alternative selection algorithm, which also does not require positive fitness and is not directly proportional to the fitness value is **rank selection**. One assigns a rank r_j to each of the N individuals e_j according to their fitness F_j, with the fittest individual being ranked first. A new fitness F' is then computed as a normalized linear function of the rank,

$$F'_j = -\frac{r_j}{N} + \frac{N+1}{N} \,. \tag{124}$$

This satisfies $F'_j \in [1/N, 1]$, such that the ratio between the best and the worst individual is N. This fitness can then be used in a roulette wheel selection algorithm, where the fittest individual is selected N times more frequently for reproduction than the least fit one. Of course, we could have chosen any other linear function of the rank,

$$F'_j = -a\frac{r_j}{N} + b \,, \qquad a, b \geq 0 \,, \tag{125}$$

such that the ratio between the best and worst individual becomes

$$\frac{F'_1}{F'_N} = \frac{1}{N}\,\frac{Nb-a}{b-a} \,. \tag{126}$$

From (126) we see that the slope a controls the evolutionary pressure. In the limit $a \to 0$, the rank of the individual (and hence its fitness) becomes irrelevant and hence there is no evolutionary pressure. In the limit $a \to \infty$, we recover the situation of (124) where the fittest individual is selected N times more often than the worst.

A slight variation of this is **truncated rank selection**, where the individuals are ranked according to their fitness and the top p percent (with typical values of $p \sim 30\%$) are selected for mating. The mating process itself is completely random, i.e. the individuals are drawn from a flat prior rather than weighting their mating probability with the numerical value of the fitness function.

Sometimes **exponential rank selection** is used instead of linear rank selection. The procedure is the same: first rank individuals according to their fitness, compute a new fitness F' based on the rank, and then use roulette wheel selection for this new fitness. The only difference is that instead of making F' a linear function of the rank r as in (125), one uses an exponential,

$$F'_j = \frac{1-e^{r-N}}{C} \,, \tag{127}$$

where C is some normalization constant.

All these methods can be combined with **cloning** or **elitist selection**, which means that the k fittest individuals are carried over to the next generation (alternatively, individuals can be allowed to mate with themselves) independent of the survival selection algorithm. While this ensures that the currently best solution is always carried over to the next generation (sometimes, the clones are not even mutated), it reduces of course the variance, especially in small sized populations. If no mutation is performed, the algorithm can become stuck in a local optimum.

Which of the selection methods works best depends on the problem in question. In [129], the authors compare the different selection mechanisms discussed above. They find that, at least for their benchmark problem (the Traveling Salesman Problem), tournament selection works best.

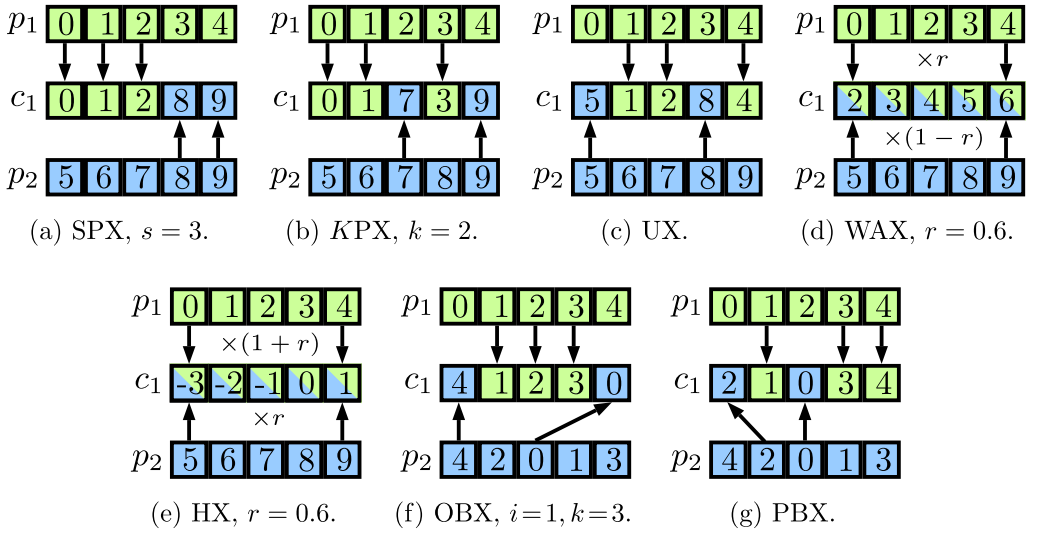
5.4. Selection for survival and number of offspring

In the second step of the selection procedure it needs to be decided how many offspring are produced and how many individuals survive and get carried over to the next generation. Usually, the total number N of individuals is kept constant or at least below a certain threshold.

In the simplest case, individuals only live for a single generation. Thus, without cloning, each pair of individuals selected for mating produces two offspring, and there are $N/2$ mating steps. Usually N is simply chosen even, but if N is odd, then the fittest (or a random) individual can be carried over to the next generation. In other cases, more children can be produced, but only the N fittest ones are carried over.

The above is an extreme case of **age-based** survival. In this survival algorithm, each individual gets to live for a fixed number g of generations before it dies and is replaced by offspring, irrespective of its fitness. The case above corresponds to $g = 1$. In the beginning, individuals are initialized with random ages. In this case, either the total population size varies but the number of offspring is fixed, or the number of offspring is adapted such that those individuals that die are replaced. Another possibility is to keep both population size and number of offspring fixed by assigning age 0 to the first N/g individuals, age 1 to the next N/g and age $g-1$ to the last N/g, and then allowing for N/g offspring (this requires $N/g \in \mathbb{Z}$).

In **fitness-based survival**, the k fittest individuals replace the k least fit individuals in the next generation. The least fit individuals to die can be selected using the algorithms outlined above, so for example via roulette wheel, tournament, or rank selection. One distinguishes two cases: In the first case, the offspring compete against each other and the k fittest offspring individuals replace the k least fit parent individuals (in this case the number of offspring is larger than k). In the second case, the parents and the offspring compete, and the N fittest individuals are selected from the pool of parents and children. In the first approach, the best individual can be lost, while the second approach will always retain the best individual, as in elitist selection. As explained above, elitist selection runs the risk of converging to a local optimum, so the first approach is more widely used. Common values for k are $N/7$ to $N/3$.



(a) SPX, $s = 3$. (b) KPX, $k = 2$. (c) UX. (d) WAX, $r = 0.6$.

(e) HX, $r = 0.6$. (f) OBX, $i = 1, k = 3$. (g) PBX.

Fig. 31. Overview of reproduction mechanisms: single point crossover (SPX), k-point crossover (kPX), uniform crossover (UX), whole arithmetic recombination (WAX), heuristic crossover (HX), order-based crossover (OBX), and position-based crossover (PBX).

5.5. Reproduction

After the parents have been selected, their offspring is produced either by cloning or by crossover. If crossover is used, the DNA of two or more parents is combined to form one or more children. There are several ways of combining the genetic information. Let us consider individuals e_i with a genome consisting of N alleles $a_j^{(e_i)}$, $j = 1, \ldots, N$. Let us denote parent individuals by p_i and their children by c_i. Often, one chooses two parents and two children. Let us discuss the most common crossover techniques; for a comparison, see e.g. [130,131]:

- **Single point crossover (SPX):** Choose a random locus $s \in [1, N)$ of the chromosome of p_1 and p_2. Form two children; child c_1 inherits alleles 1 to s from p_1 and alleles $s+1$ to N from p_2. Similarly, child c_2 inherits alleles 1 to s from p_2 and alleles $s+1$ to N from p_1, see Fig. 31a for $N = 5$ and $s = 2$.
- ***K*-point crossover** (***K*PX**): This is similar to single point crossover, except that the genes are spliced at K loci s_k, $k = 1, \ldots, K$ and the allele sequences are inserted in an alternating fashion from p_1 and p_2. SPX is KPX with $K = 1$ and $s = s_1$. In Fig. 31b, we illustrate the process for $N = 5$, $K = 3$ and $s_k = (2, 3, 4)$.
- **Uniform crossover (UX):** For the first child, choose randomly for each allele whether it is taken from p_1 or p_2. The second child will get the kth allele from p_1 if c_1 got it from p_2 and vice versa. The process is illustrated in Fig. 31c for a random sequence $(p_2, p_1, p_1, p_2, p_1)$.
- **Whole arithmetic recombination (WAX):** Choose a number $r \in [0, 1]$ and compute the alleles of the children by forming linear combinations, such that for child c_1 we have $a_j^{(c_1)} = r a_j^{(p_1)} + (1-r) a_j^{(p_2)}$ and similarly for the alleles of child c_2, $a_j^{(c_2)} = r a_j^{(p_2)} + (1-r) a_j^{(p_1)}$, see Fig. 31d for an illustration for c_1 with $r = 0.6$. Note that this requires either rounding or continuous alleles. For $r = 0.5$ this produces the arithmetic mean of the two parent alleles, while for $r = 0$ or $r = 1$, it corresponds to cloning.
- **Heuristic crossover (HX):** Choose a number $r \in [0, 1]$ and produce the alleles $a_j^{(c_1)} = a_j^{(p_1)} + r(a_j^{(p_1)} - a_j^{(p_2)})$ and $a_j^{(c_2)} = a_j^{(p_2)} + r(a_j^{(p_2)} - a_j^{(p_1)})$. Note that this requires again rounding or continuous alleles. It might nevertheless happen that some allele computed this way is not within the valid range (not in the gene pool). In this case, the allele is either clipped or the process is repeated for a different random value r. We illustrate this in Fig. 31e with $r = 0.6$

There are many other crossover operations tailored to specific applications. A wide variety of crossover algorithms have been developed for cases in which alleles are mutually distinct integers that encode elements that are drawn from a fixed set of possible elements. Problems of this type occur e.g. in knapsack or traveling salesman, onto which many optimization problems can be mapped, see e.g. [131]. Since in that case the genotypes are simply all permutations of all ordered (sub)sets of the N numbers encoding the elements of the set, these are sometimes called **permutation encodings**. In particular, there are

- **Order-based crossover (OBX):** A sequence of alleles a_i to a_{i+k} of length k is taken from one parent and the rest is filled up with alleles from the second parent such that each element occurs only once. The process is illustrated in Fig. 31f for $i = 1$ and $k = 3$.

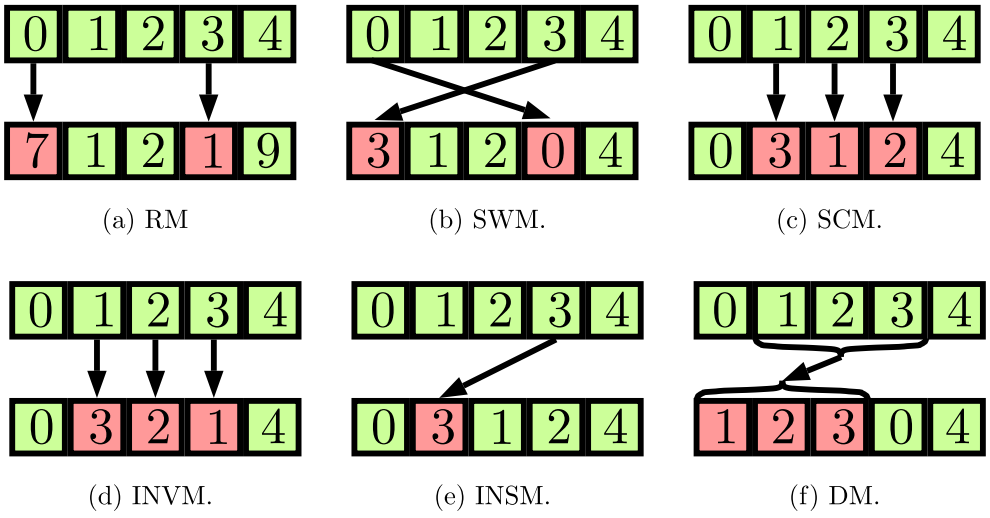


Fig. 32. Overview of mutation mechanisms: random mutation (RM), swap mutation (SWM), scramble mutation (SCM), inversion mutation (INVM), insertion mutation (INSM), displacement mutation (DM).

- **Position-based crossover (PBX):** Randomly select loci of alleles of p_1 and insert the alleles at the same position for the child. Fill up the missing alleles from p_2 in the order in which they appear, skipping loci that are already occupied by alleles from p_1. This is illustrated in Fig. 31g for randomly selected loci $s_i = (2, 4, 5)$.

Even more specialized crossover algorithms exist such as **CX crossover**, **subtour exchange crossover**, etc. For many applications, the simplest single point crossover works very well.

5.6. Mutation

As with crossover, there are many ways in which alleles can be altered. The most common ones are

- **Random Mutation (RM):** A set of alleles is chosen randomly and is replaced with other, also randomly chosen, alleles from the gene pool. This is illustrated in Fig. 32a for randomly chosen alleles (a_1, a_4).
- **Swap Mutation (SWM):** Two alleles are selected and their loci are interchanged. This is often used in permutation encodings, since it ensures that the alleles stay mutually distinct. Fig. 32b shows the process for swapping $a_1 \leftrightarrow a_4$.
- **Scramble Mutation (SCM):** Take all alleles in a random interval $[s_1, s_2]$ and mutate by randomly permuting ("scrambling") the alleles in this interval. As for swap mutation, this is popular for permutation encodings. The process illustrated in Fig. 32c has $[s_1, s_2] = [2, 4]$.
- **Inversion Mutation (INVM):** Take all alleles in a random interval $[s_1, s_2]$ and mutate by reversing the order of elements in the substring. Note that this is a special case of scramble mutation. In Fig. 32d we illustrate the process with $[s_1, s_2] = [2, 4]$.
- **Insertion Mutation (INSM):** Choose a random allele and a random locus s. Remove the allele from its locus and insert it at locus s, shifting all subsequent alleles to the right. In Fig. 32e, we have chosen allele a_4 and locus $s = 2$.
- **Displacement Mutation (DM):** Take all alleles in a random interval $[s_1, s_2]$ and mutate by moving the entire interval to a new locus s_3. The process is illustrated in Fig. 32f for $[s_1, s_2] = [1, 3]$ and $s_3 = 1$. Note that DM includes INSM as a special case where $s_1 = s_2$ and $s_3 = s$.

5.7. Mutation versus crossover

Note that both crossover and mutation alter the genome of the next generation. It is therefore interesting to ask which altering process is more efficient and how they depend on the number of generations considered, the population size, and the selection mechanisms for mating and survival. Of course, these are many hyperparameters, all of which also depend on the problem the GA is trying to optimize.

In a study that addresses these questions [132], it was observed that crossover is more successful than mutation. However, the difference is small and the selection mechanism has much more influence on the quality of the final result. It was furthermore found that choosing a large population with fewer generations produces similar results to choosing a small population with a larger number of generations. Note that the former favors crossover over mutation, since there are many individuals with only few chances of randomly evolving a good quality. In contrast, the latter favors mutation over crossover, since the genome of the few individuals in the population of each generation only realizes a small fraction

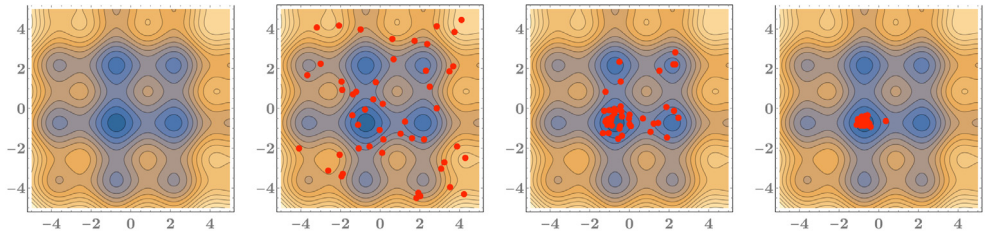


Fig. 33. The function we want to minimize using GAs is shown in Fig. 33a. Figs. 33b to 33d show the populations after the first, second, and third iterations. The algorithm converges extremely fast for this simple problem.

of all possible genomes, but there is a lot of time to probe the larger space of possible genomes via mutation. Also note that the schema explanation for why genetic algorithms work would favor crossover over mutation, since the chance of passing on a schema intact is higher in this case. Consequently, this would mean that large populations with fewer generations perform better, which is not observed in [132].

5.8. Example

Let us apply all these techniques to a simple example, i.e. the minimization of the function

$$f(x, y) = x^2 + y^2 + 6\sin(2x) + 6\sin(2y)\,, \tag{128}$$

cf. Fig. 33a. This function has many local minima; its global minimum is at $x = y \approx -0.7245$.

The chromosomes of each individual in the population consist of two alleles, the x- and y-coordinates of f. We furthermore set the population size to 50 and take $-f(x, y)$ to be the fitness function. Since this can be negative, we need to use a reproduction and survival algorithm that can deal with negative fitness values. We choose binary tournament selection for reproduction and 6-fold tournament selection (among the children only) for survival, and each player in the tournament is allowed to play multiple times. The fittest children replace their parents in the next generation. Since the parents did not compete in the survival algorithm, they die for sure, so there is no elitist selection. For reproduction, WAX crossover is used, where r is drawn from a uniform distribution for each child. In this case, two parents have $4 \times 50 = 200$ children. After survival selection, mutation is carried out by drawing a random value from a Gaussian distribution $\mathcal{N}(0, 0.5)$ and adding this value to x or y with a mutation probability of 0.1.

The resulting generations are plotted in Figs. 33b to 33d. Fig. 33b shows the first, randomly initialized population. In Fig. 33c, the next generation has been bred and mutated. As can be seen, the individuals are already concentrated at the local minima of the function. In the next generation, all individuals have settled into the global minimum.

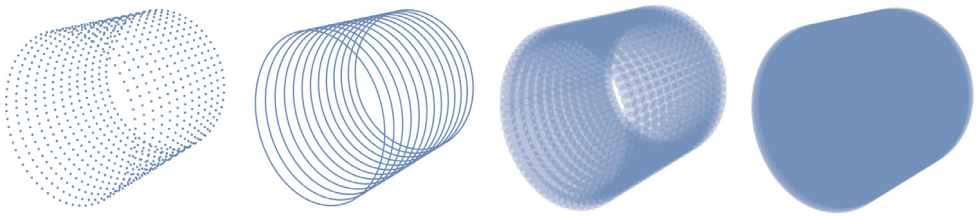
5.9. Summary

Genetic algorithms are parameter-free search and optimization algorithms. They iteratively approximate a solution by mimicking evolution: The best approximations are kept over several generations, altered in each step by combining properties from two parent solutions that work well and/or by randomly changing some of their properties. Small populations with many generations profit more from mutation, while large populations with a small number of generations profit more from crossover.

6. Persistent homology

Persistent homology is a data analysis tool that finds wide-spread applications, ranging from biology through chemistry to engineering and physics. In contrast to the methods discussed so far, persistent homology is not used to optimize data, but to analyze structure in data. For some recent applications to cosmology and string theory see [133] and [134,135], respectively. For the original work see [136] and for a textbook on the subject of persistence theory in data analysis see [137]. A nice recent introduction and overview of available tools can be found in [37].

The idea of persistent homology is to assign topological properties to discrete data. Since many string theorists are familiar with homology computations (usually for complex manifolds) as well as the notion of polytopes, simplices, etc., the mathematical aspects are not too hard to understand. In order to apply these concepts to discrete data points (sometimes also called **finite metric spaces** or **point clouds** in the literature), the data points are embedded in an F-dimensional feature space $\mathbb{R}^F$ and then replaced by F-dimensional balls of radius $\epsilon \in \mathbb{R}_{\geq 0}$.



(a) Points. (b) Circles. (c) Cylinder surface. (d) Ball.

Fig. 34. The structure of data depends on the scale at which the data is considered.

The problem with this approach is that this is not unique but depends on the choice of ϵ; as ϵ is increased, the balls begin to overlap and form new cycles. This is saying nothing else than that the structure of data depends on how finely-grained one looks at it. Look for example at Fig. 34: Depending on the scale, one might say the figure depicts a collection of points, a collection of circles, the surface of a cylinder, or a filled cylinder (i.e. topologically a ball).

With this picture, it is also rather easy to understand intuitively the changes in homology (thought of as the number of closed cycles that are not the boundary of another cycle). For very small ϵ, we just have a collection of disconnected components. As ϵ is increased, the components start to overlap and form non-trivial cycles, reducing the number of disconnected components in the process. Increasing ϵ further can fill in the cycles and lead to surfaces, and so on. Moreover, as higher-dimensional cycles start to emerge, they can be the boundary of lower-dimensional cycles, thus removing them from the homology.

The idea of persistent homology is to compute the homology of the resulting space as a function of ϵ. Cycles that persist over a larger range of ϵ values are taken to represent true features of the underlying data set. The nice thing about this is that the result is topological and hence does not depend too much on the chosen embedding into feature space (in the sense that homology is homotopy invariant).

6.1. Mathematical definition of persistent homology

In order to define persistent homology, one first defines a ***k*-simplex** as a k-dimensional polytope given by the convex hull of its $k+1$ vertices. A **face** of a simplex is the convex hull of a non-empty subset of the $k+1$ points that define the k-simplex; faces are simplices themselves. One furthermore uses the term **vertex** for a 0-dimensional face (point), **edge** for a 1-dimensional face (line) and **facet** for a codimension one face of the simplex.

A **simplicial complex** $\mathcal{K}$ is then defined as a set of simplices such that every face of a simplex of $\mathcal{K}$ is in $\mathcal{K}$ and the intersection of any two simplices $s_1, s_2 \in \mathcal{K}$ is either empty or a face of both s_1 and s_2. With this notation we can define the standard boundary operator ∂_p that maps a p-simplex to its boundary $(p-1)$ simplices, i.e. a collection of its $(p-1)$-dimensional faces. A ***k*-chain** is a subset of k-simplices of $\mathcal{K}$. One can furthermore define addition of k-chains with coefficients in some field of characteristic n. The set of all k-chains endowed with this addition forms a group which is denoted by C_k. Often one considers the case of characteristic $n=2$. By the Universal Coefficient Theorem, which states that the integer homology can be decomposed into a direct sum of a free part (i.e. a number of copies of $\mathbb{Z}$) and a torsion part, the result depends on the characteristic in the presence of torsion cycles. A cycle C is a q-torsion cycle if C is a non-trivial homology element, but qC is trivial in homology, i.e. it is the boundary of some higher-dimensional cycle. For practical purposes (i.e. if the q-torsion of the cycles is not too large), one can then compute the homology for several characteristics n and compare the results to infer q.

With the above definitions, we obtain a sequence of groups

$$0 \xrightarrow{\partial_{d+1}} C_d \to \ldots \to C_p \xrightarrow{\partial_p} C_{p-1} \xrightarrow{\partial_{p-1}} C_{p-2} \to \ldots \xrightarrow{\partial_1} C_0 \xrightarrow{0} 0\,, \tag{129}$$

where d is the dimension of the highest-dimensional simplex in $\mathcal{K}$. The pth homology is now defined as the homology of this sequence, i.e. the failure of being exact at position p,

$$H_p(\mathcal{K}) = \ker(\partial_p)/\mathrm{im}(\partial_{p+1})\,. \tag{130}$$

Elements of $\ker(\partial_p)$ are called p-**cycles** and elements of $\mathrm{im}(\partial_{p+1})$ are called p-**boundaries**. The dimension h_p of $H_p(\mathcal{K})$ is called the pth **Betti number** of $\mathcal{K}$.

Equipped with these notions we can now define **persistent homology**. Consider a filtration, i.e. a finite sequence of nested subcomplexes of $\mathcal{K}$,

$$K_1 \subset K_2 \subset \ldots \subset K_d = \mathcal{K}\,, \tag{131}$$

and compute the homology of each subcomplex. Due to the filtration (the filtration parameter is the size ϵ of the balls), we have inclusion maps $K_i \hookrightarrow K_j$, $i \leq j$ that induce maps $f_{i,j} : H_p(K_i) \to H_p(K_j)$ of the homology groups. The **p^{th} persistent homology** of $\mathcal{K}$ is now defined as the pair

$$H_p^{\text{PH}} := \left(\{H_p(K_i) \mid 1 \leq i \leq d\}\,,\ \{f_{i,j} \mid 1 \leq i \leq j \leq d\}\right). \tag{132}$$

Note that this contains more information than just the homology groups (or their Betti numbers) at each filtration step.

It can be shown that the Betti numbers cannot increase in the filtration. Intuitively, as we explained above, p-cycles can disappear because they are merged to $(p+1)$-cycles or because they become the boundary of newly formed $(p+1)$-cycles. They can also appear from lower-dimensional q-cycles (with $q < p$), but the theorem states that always at least as many "old" p-cycles disappear as "new" cycles appear.

In all this, we have just assumed that we have a collection of p-cycles which are determined by ϵ. There are different (but related) ways of assigning p-cycles to points in a metric space. One possibility is to use the **Vietoris–Rips complex** (VR complex), which says that $(p+1)$ points form a p-simplex if the largest maximal distance between all pairs of points is at most ϵ. This is closely related to the **Čech complex**, which assigns a simplex to every finite subset of balls with radius ϵ whose common intersection is non-empty. The VR complex at radius ϵ is roughly the same as the Čech complex at radius $\epsilon/2$; however, the latter depends on the metric space the complex is embedded in while the former is independent of the embedding.

It should be noted that both complexes require exponentially many ($\mathcal{O}(2^N)$ for N points) comparisons of distances, which becomes prohibitively expensive for a large number of points. To improve on this, one can use **α-complexes**. These are constructed from the (dual) Voronoi diagram (cf. Section 7.2) or the Delaunay triangulation of the point set. The Delaunay triangulation is constructed by triangulating the simplex in the following way: Triangulate the simplex by adding edges between points such that one can find a circle such that the two points of an edge or the three points of a triangle lie on the circle and no points are in the interior of the circle. One then assigns to the edges and triangles the smallest radius α of such a circle. The α-complex is the simplicial complex of all edges and triangles whose associated Delaunay radius is $\leq 1/\alpha$. This improves the complexity of the problem from $\mathcal{O}(2^N)$ to $\mathcal{O}(N^{\lceil F/2 \rceil})$ (where $\lceil\cdot\rceil$ is the ceiling function) for N points in F-dimensional feature space. For a large number of points in a feature space of fixed dimension, this is exponentially better behaved since N is in the base rather than the exponent.

Yet a different way to improve the computational complexity are **Witness complexes**. The idea is to construct the simplicial complex not from all points but from a much smaller "representative" subset L of points called **landmark points**. The complexity is $\mathcal{O}(2^{N_L})$ where N_L is the number of points in L. The set L can be constructed by either randomly sampling points from the point cloud or by using an optimization strategy for selecting landmark points. In order to construct the complex, one says that a point w is a (weak) witness for a simplex given by the points $\langle p_0, p_1, \ldots, p_k\rangle$ if its $k+1$ nearest neighbors in L are precisely $\{p_0, p_1, \ldots, p_k\}$. This can be made dependent on the filtration parameter ϵ by demanding that the former condition is met up to some tolerance ϵ.

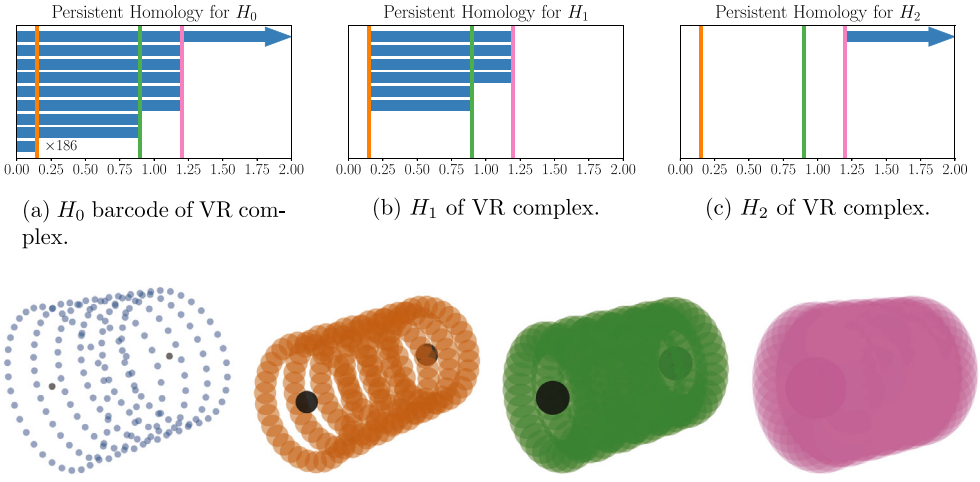
The different implementations available for computing persistent homology will use different complexes and hence the results can differ somewhat. Some implementations allow to choose the complex to use. Often, the Witness complex is fast and gives good results.

6.2. Barcodes and persistence diagrams

After detailing the mathematical definition of persistent homology let us now explain nice ways to visualize it. The first way of visualization is called a **barcode**. One creates a separate barcode for each homology group H_p. For any given, fixed p, one computes H_p^{PH}. Next, one creates a diagram with the number of filtration steps (i.e. the discrete choices for ϵ) on the x-axis. At each value of ϵ, one draws a point for each element of H_p^{PH}. Note that, since the Betti numbers of a fixed homology class are not increasing, there will be fewer and fewer points the larger the filtration parameter is. If $f_{i,i+1}$ maps an element from $H_p(K_i)$ to $H_p(K_{i+1})$, connect the corresponding points. If $f_{i,i+1}$ maps an element from $H_p(K_i)$ to 0, also connect the two points but delete the end point. Defined this way, the barcode one obtains depends on a choice of basis for the homology groups and will look quite chaotic. However, the **fundamental theorem of persistent homology** [138] ensures that for persistent homology over a field there exists a basis such that the barcodes are just a collection of half-open, straight lines.[10] By increasing the radius ϵ over time, we can read off the barcode from left to right as specifying when a cycle is born (from joining lower-dimensional cycles) and when it dies (by becoming a higher-dimensional cycle or by becoming a boundary). In order to interpret the features of our data, we look for cycles which live for a long time. Note that, depending on the choice for the upper limit of the filtration parameter, some cycles never die. This is indicated by putting an arrow to the right on the line corresponding to the cycle that has not died at the end of the filtration.

An alternative way of visualization are **persistence diagrams**. In these diagrams, one plots discrete "time" steps (i.e. the filtration parameter) on both the x- and y-axes. Then, we plot a point (i, j) for each cycle that is born at filtration step i and dies at filtration step j. That is, the x-axis denotes the time of birth and the y-axis the time of death. Note that $i < j$ (a cycle has to be born before it can die), i.e. all points will lie above the diagonal $y = x$. Since multiple cycles born at the

[10] Usually, libraries that compute persistent homology use this basis, cf. e.g. the output of Javaplex in Fig. 35.



(a) H_0 barcode of VR complex.

(b) H_1 of VR complex.

(c) H_2 of VR complex.

(d) Different topologies as a function of the filtration parameter.

Fig. 35. The structure of data depends on the scale at which the data is considered.

same filtration step i could also die at the same filtration step j, it is customary to either draw dots whose radius is given by the number of points the dot represents, or to put several dots next to each other. Cycles that are born at filtration step i but never die are indicated with a square (instead of a dot) with coordinates $(i, j_{\max})$, where $j_{\max}$ is the total number of filtration steps.

6.3. Summary and example

Persistent homology assigns a homology to a point cloud. It is based on blowing up points into larger and larger balls. Intersecting balls form p-cycles, for which the homology is computed. Cycles of the point cloud that persist for a longer time (i.e. for many different radii) are interpreted as features of the data. However, interpreting (especially higher) homologies of point clouds can be challenging.

Let us use the example of a point cloud and its corresponding barcodes given in Fig. 35. We use Javaplex to compute the persistent homology of the associated Vietoris–Rips complex. For small filtration parameter $\epsilon \in [0, 0.15)$ (up to the orange line) the data consists of 194 isolated data points: 6 rings with 32 points shown in blue, and 2 points at both "ends" of the cylinder shown in black for better visibility. In the corresponding barcode diagram 35a of H_0, we do not draw all 194 bars of H_0, since 186 look exactly the same. Up to the filtration parameter value indicated by the orange line, we have $b^\bullet = (194, 0, 0)$, one for each point.

For larger ϵ, the individual balls begin to overlap and form 6 one-cycles (circles), removing $6 \times 31 = 186$ individual points from b^0 and keeping 6 representative ones. The two points at the cylinder caps still stay isolated points throughout the interval $[0.15, 0.9)$ (i.e. up to the green line). Hence in this interval, our point cloud has the topology of two isolated points and six circles, so $b^\bullet = (2 + 6, 6, 0)$.

As we increase ϵ up to the purple line, i.e. in the interval $[0.9, 1.2)$, the two spheres at the cap merge with the two outer circles to form two disks, ending their lives as individual elements of H_0, while also filling in two circles to full disks and thus removing them from H_1. Thus, only the inner 4 circles remain as elements of H_1. We hence get the Betti number of two disks and four circles, $b^\bullet = (2 + 4, 4, 0)$.

Past this point, for $\epsilon > 1.2$, the four rings and two disks merge to form topologically a sphere. Out of the six elements of H_0, only one independent representative remains, and the last four elements of H_1 die to give birth to the sphere, i.e. an element in H_2. Hence from this point onward, we get the sphere Betti numbers $(1, 0, 1)$. Note that if we had computed even larger filtration parameter values, all points would have overlapped eventually, leading to just a single element of H_0. Since every barcode eventually ends this way, we did not include it here.

7. Unsupervised learning—clustering and anomaly detection

Clustering and anomaly detection are classical areas for unsupervised machine learning. In clustering, the algorithm is clustering similar data without the user telling the machine what the measure for similarity should be. The supervised counter-part of clustering is classification, see Section 9.

In anomaly detection, the machine observes typical data and identifies outliers, i.e. data that is "out of the ordinary". This can be used for example to identify suspicious activity in computer networks: Common network traffic is monitored and unusual events are reported since they might point towards attacks on the network. Of course, one does not know

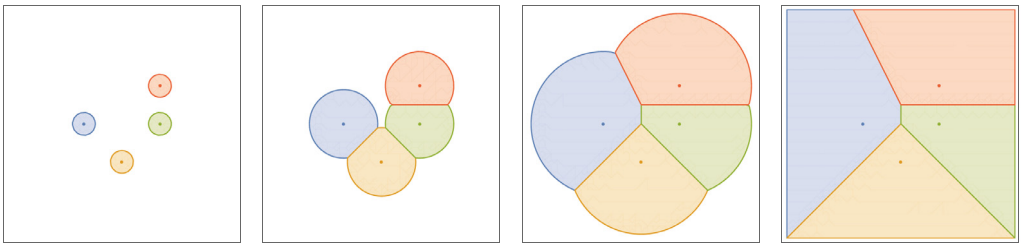


Fig. 36. Example for a Voronoi diagram in $\mathbb{R}^2$, Euclidean distance and four seed points. Points in the colored region have the property that they are closest to the seed point of the corresponding color.

in advance what the attack will look like, but one knows what usual activity looks like such that unusual events can be identified nevertheless. This has also important applications in physics, e.g. when searching for new or rare events at particle accelerators. It could potentially also be applied to discover e.g. "special" vacua in the string landscape. Clustering and anomaly detection are related, since one can cluster "usual" data and the anomalies will then lie outside the clusters.

We have already discussed a few algorithms that can be used for clustering and/or anomaly detection: Adaptive resonance theory in Section 4.3, autoencoders in Section 4.6 (in the sense that outliers will be reconstructed less well than typical data) and to some extent also persistent homology in Section 6 (in the sense that clusters in feature space will form disconnected components that only join for large radii, which is visible as persistent elements of b^0 in the barcode). In the following we will introduce a few more methods. Unless stated otherwise, we consider in this section a set of N data points in an F-dimensional feature space $\mathbb{R}^F$.

7.1. The curse of dimensionality

Most clustering algorithms discussed in this section cluster data based on relative proximity. This means they are prone to suffer from an effect called the "curse of dimensionality" [139], which causes the algorithms to behave badly if the dimension F of the feature space is large. The reason is that for large F, all points are "far away from each other". To explain what we mean by that, consider how the median distance r_m scales with F. Considering a unit ball with uniformly distributed data points, half the points will, by definition, be at a distance larger than r_m, i.e. lie in a shell between r_m and 1. We find for the probability of N random points lying in this shell

$$\frac{1}{2} = \left(\frac{1^F - r_m^F}{1^F}\right)^N . \tag{133}$$

We can also read this as an equation for the median distance r_m,

$$r_m = \left(1 - \left(\frac{1}{2}\right)^{1/N}\right)^{1/F} , \qquad \lim_{F\to\infty} r_m = 1 . \tag{134}$$

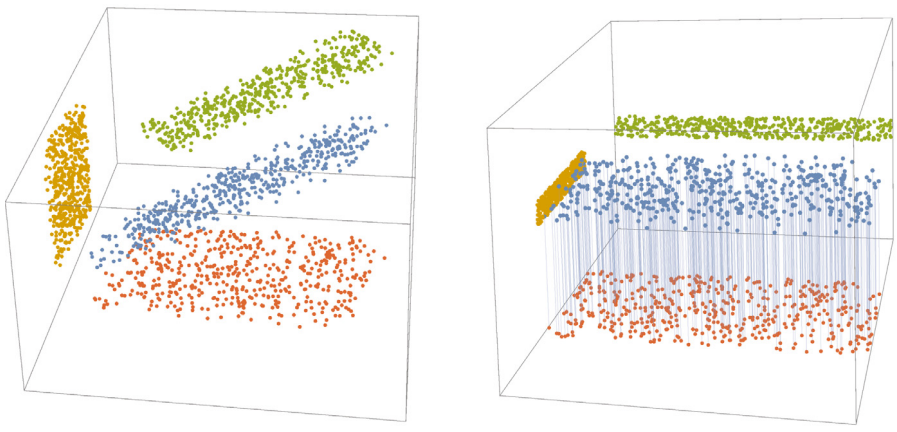
Thus for large F, the N points will lie just beneath the sphere bounding the unit ball and thus be far apart. This problem persists unless N grows exponentially compared to F. There is not much that can be done against the curse of dimensionality, except to reduce the feature space using e.g. autoencoders (cf. Section 4.6) or principal component analysis (as discussed in the next Section 7.3). Luckily, in many physics applications, the number N of points is large but the number of features needed to describe each point is manageable.

7.2. Voronoi diagrams

Voronoi diagrams often show up in clustering algorithms that assign points to clusters based on their distance to the cluster center. A D-dimensional Voronoi diagram is constructed as follows: For a given set $P = \{p_1, \ldots, p_N\}$ of seed points, partition the D-dimensional space into cells around each p_i such that all points in the cell associated with p_i are closer (or at most equidistant) to p_i than to any other p_j, $j \neq i$ in some metric (often the Euclidean metric). We illustrate the process of how a Voronoi diagram is formed for $D = 2$ and 4 points with Euclidean distance in Fig. 36. The Delaunay triangulation mentioned in Section 6 is the dual graph of the Voronoi diagram.

7.3. Principal component analysis

Principal component analysis (PCA) identifies linear combinations of features that carry the most information and discards the rest. As a result, data will be embedded in a lower-dimensional feature space, which can help ameliorate the curse of dimensionality discussed above. This reduction is only possible if data "clusters" along higher codimension subloci



(a) Original data set. (b) Rotated data set.

Fig. 37. PCA for the original and rotated data set. As can be seen from the projections onto the three coordinate planes, the principal component aligned with the vertical (z) axis of the rotated data set carries the least information, so by projecting down to the 2D x–y plane the points are still somewhat distinct.

in feature space. If, for example, the data points contain information or features $x_1, \ldots, x_k$ and the remaining features $x_{k+1}, \ldots, x_F$ are correlated with the first k features x_μ, the dimension of the feature space can be reduced without (too much) loss of information by forming linear combinations of the features $x_1, \ldots, x_k$ and discarding the derived features $x_{k+1}, \ldots, x_F$.

This is essentially the same as diagonalizing the $F \times F$ covariance matrix of the features via an orthogonal transformation. Imagine fitting an ellipsoid to (the covariance of) our data and identifying the main principal axes of the ellipsoid. If the ellipsoid is very elongated, the coordinates of the points along the elongated axis will essentially be enough to distinguish different points, while the coordinates along the short axis give much less information in terms of discriminating data. We thus identify the $k < F$ largest eigenvalues and their corresponding eigenvectors, rotate our feature space accordingly, and throw away the other $F - k$ dimensions. The result depends of course on the threshold for the eigenvalues above which the corresponding direction is kept.

In more detail, let us assume we have N data points with F features, given by an $N \times F$ matrix A. We first need to center our data by subtracting the mean in each direction of feature space. Depending on whether the variance of the features is important or not, we can also normalize to unit variance by dividing by the standard deviation of that feature.

Next, we compute the $F \times F$ covariance matrix $C = A^T \cdot A$ and diagonalize[11] it to find the eigensystem, $D = OCO^T$, ordered by eigenvalues λ_μ in descending order. Then we rotate the original data to align with the eigendirections of the covariance matrix, $\tilde{A} = A \cdot O$. In Fig. 37, we present the idea behind PCA in a toy example with $N = 500$ and $F = 3$. In this example, the height (distribution in z-direction) of all rotated data points is more or less similar, so we can project along this axis without losing much information.

There are different approaches to decide how many dimensions in the rotated feature space are kept. The simplest approach is to just choose a fixed number k and to keep only the first k features. This approach is viable if there is for some reason an upper bound on the dimension of feature space. However, with this approach, it can happen that one either discards features that are important to discriminate data, or one keeps more features than actually necessary.

In order to guarantee inclusion of the most important features needed to discriminate the data, it is useful to introduce the notion of **maximum variance explained** (MVE). First, note that the total variance is just given by the sum of the eigenvalues λ_μ of the covariance matrix C. The MVE of the μth feature is then defined as

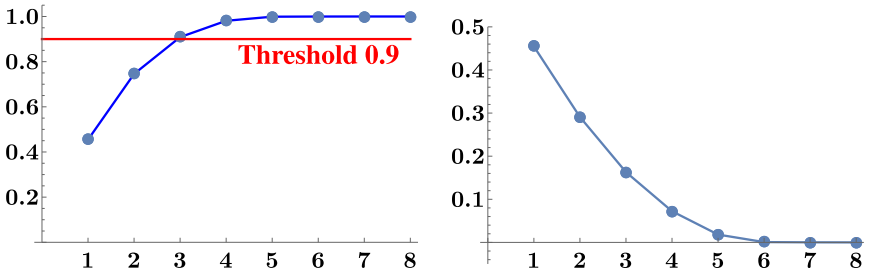
$$e_\mu = \lambda_\mu / \sum_\mu \lambda_\mu \,, \tag{135}$$

i.e. as the fraction of the total variance encoded in (or explained by) the μth direction in (rotated) feature space. With this, there are two possibilities for choosing the number of features to include:

- Choose a threshold for the percentage of the MVE (usually around 80–90 percent) and keep adding features until their combined MVE crosses this threshold. This will guarantee that a minimum variance is kept, but might require adding many directions with small but similar explained variance. The number k of features to retain can be read off easily from a plot with points $(k, \sum_{\mu=1}^{k} e_\mu)$ with $k = 1, \ldots, F$. In the example of Fig. 38a, we created a toy point set[12]

[11] Since the covariance matrix is real and symmetric, it can be diagonalized with an orthogonal transformation.

[12] We do not use the point set from Fig. 37 since there $d = 3$, which makes the problem too simple to illustrate the VAE method and the scree plot method.



(a) Number of feature dimensions vs percentage of MVE

(b) Scree plot.

Fig. 38. Plots of MVE percentage and scree plot to find the number of dimensions to keep in a toy example with $N = 5000$, $d = 8$.

with $N = 5000$ points in a $d = 8$-dimensional feature space. The points were drawn from a uniform distribution

$$[\mathcal{U}(-b_\mu, b_\mu)]^{\otimes 8} \quad \text{with} \quad b_\mu = (0.1,\ 0.5,\ 1,\ 5,\ 10,\ 15,\ 20,\ 25)\,, \tag{136}$$

and the threshold was chosen at 0.9. With this threshold, we need to include the first three eigendirections.

- Make a **scree plot** of the maximum variances explained vs. the number of features, i.e. plot the points (μ, e_μ) with $\mu = 1, \ldots, F$. Identify a (sharp) drop between the explained variances of two successive (rotated) features, and include all features up to this drop. This sharp kink is sometimes called the "elbow". In this way, one will choose an efficient encoding, but might fall short of including enough features to reach a certain percentage of the total variance explained. In the example of Fig. 38b, we would identify the elbow at $\mu = 4$, so we would include the first 4 eigendirections.

7.4. K-means clustering

K-means is a rather simple clustering algorithm that assigns points x_i from a set of N points to K clusters c_a, $a = 1, \ldots, K$, based on their mean distance from the cluster centers m_a. The number of clusters K has to be set by hand. For each cluster, the algorithm first picks a random center and assigns each data point to the cluster whose center is closest to them (usually in squared Euclidean distance). Once each point has been assigned, the new cluster centers are computed based on this assignment and the process is repeated until no cluster center gets updated anymore.

In more detail, the clusters at step t are given by

$$C_a^{(t)} = \{x_r \mid \|x_r - m_a^{(t)}\|_2^2 \leq \|x_r - m_b^{(t)}\|_2^2\,, \quad \forall\, b \in [1, K]\}\,. \tag{137}$$

This introduces boundaries between the data that lead to a Voronoi diagram, cf. Section 7.2. Once each data point has been assigned to exactly one cluster (if points lie exactly on the boundary, which is a measure zero set, they are only assigned to one cluster using some tie-break mechanism), the new center of each cluster is computed by taking the mean of all positions of all data points in each cluster,

$$m_a^{(t+1)} = \frac{1}{|C_a^{(t)}|} \sum_{x \in C_a^{(t)}} x\,, \tag{138}$$

where $|C_a^{(t)}|$ denotes the number of points in the cluster $C_a^{(t)}$. Using these new cluster centers $m_a^{(t+1)}$, the cluster to which each data point belongs is updated based on the point's distance from the new center position, see Fig. 39.

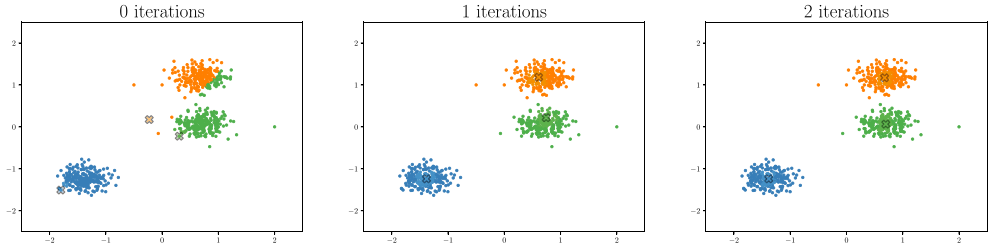
The algorithm presented above (sometimes referred to as Lloyd's algorithm [140,141]) is just one possibility of finding the final clusters, albeit a very effective one. In principle, one could also perform gradient descent to find the centers of each cluster. The loss function would be

$$L^{K\text{-Means}} = \sum_{i=1}^{N} \frac{1}{2} \min_a (x_i - m_a)^2\,, \tag{139}$$

and the update for the parameters (i.e. the cluster centers) are

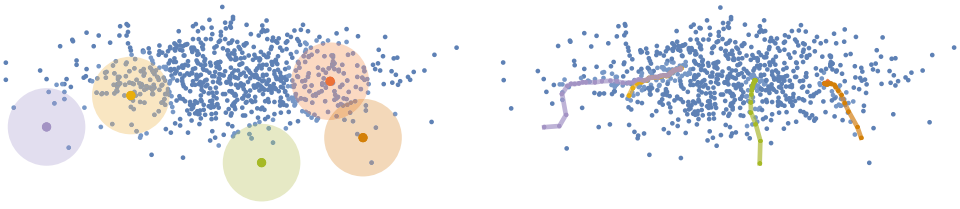
$$\Delta m_a = -\alpha \nabla_m L^{K\text{-Means}} = \begin{cases} \alpha \sum_i (x_i - m_a) & \text{for } i \text{ s.t. } x_i \text{ is closest to } m_a \\ 0 & \text{else} \end{cases} \tag{140}$$

However, this algorithm is in general less efficient than Lloyd's algorithm described above. The latter essentially minimizes using Newton's method, which is a second order approximation (meaning it performs updates based on the first two derivatives), while gradient descent only uses the first derivative [142].



(a) 0 iterations. (b) 1 iteration. (c) 2 iterations.

Fig. 39. We show how the cluster centers (marked by an $\times$) move in K-means clustering for several iterations. Note how they are shifted towards denser regions.



(a) Randomly initialized windows. (b) Trajectory during the algorithm.

Fig. 40. We show the initial window position and their trajectories for the mean shift clustering algorithm. We see how in each update the windows move closer to dense subregions.

While the algorithm is rather fast and simple, it has the problem that one needs to know in advance how many clusters the data contains, which might not be the case in many applications. One way of determining a good value for the number of clusters is to run the algorithm with a different number of clusters K and compute in each case the sum of squared distances of each point from its data center. Then plot K against this sum of squared distances and use the value for K for which there is an "elbow" in the plot. Note that the sum of squared distances is zero if there are as many cluster centers as data points. After the elbow, including more cluster centers partitions existing clusters into sub-clusters while only marginally improving the proximity of the points in the original cluster to the cluster center.

Another disadvantage is that, since the initial centers are assigned randomly, different runs of the algorithms on the same data set might lead to different results.

There are improvements for the initialization of the centers, e.g. ***K*-means++**, which chooses the first center at random, computes the distance of all points from this center, and assigns the next center randomly, but with probability proportional to the distance square from the first center. This will lead to more evenly distributed centers at the beginning. The extra time spent in choosing the centers is usually made up for by a faster convergence and lower error.

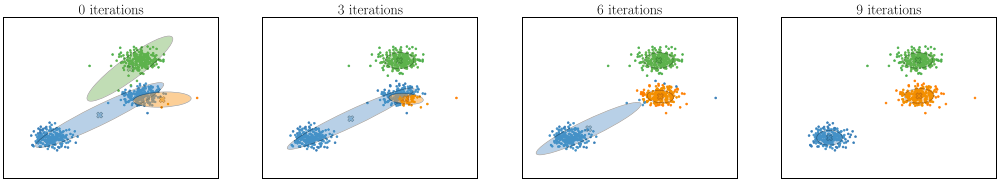
Another variation is ***K*-means mini-batch clustering**. Here, instead of using all N points for computing the cluster center, a mini-batch of size $b \ll N$ is selected randomly from all points. If one uses gradient descent for finding the new clusters, this is actually like mini-batch gradient descent in neural networks, with the limiting case of stochastic gradient descent for $b = 1$ and batch gradient descent for $b = N$.

Yet another variation is ***K*-median clustering**. The algorithm is the same, except that it uses the median instead of the mean to compute cluster centers. This makes the algorithm more robust to outliers, but also slower since computing the median requires sorting the data set. It corresponds to using the L_1-norm rather than L_2-norm in Eq. (139).

7.5. Mean shift clustering

In mean shift clustering, one finds cluster centers by sliding a circular window over the data points in feature space such that the window maximizes the number of points inside. This is done for many windows to identify all cluster centers. Then, points are assigned to the cluster with the nearest center.

In more detail, initialize $M > K$ windows (where K is the number of expected clusters) with radius r and center m_a, $a = 1, \ldots, M$. For each window, compute the mean of all points in that window and set the window center to this mean. This will cause the window to slide into more dense directions. This is repeated until the mean is not changed anymore for any of the windows. At that point, centers with overlapping windows are removed, only keeping the window with the most points. Finally, the distance of all points from the cluster center is computed and points are assigned to the closest cluster. For an illustration see Fig. 40.



(a) 0 iterations. (b) 3 iterations. (c) 6 iterations. (d) 9 iterations.

Fig. 41. We show the means and 1-σ contours for the Gaussian expectation–maximization clustering algorithm over various iterations. Note the probabilistic assignment to clusters, i.e. some points are assigned to a cluster whose center is not closest to them in Euclidean distance.

The advantage of mean shift clustering is that one need not know or fix the number K beforehand. Instead, on can choose M rather large, since windows that find the same cluster are removed afterwards anyway. Like all clustering mechanisms based on distance, mean shift clustering suffers from the curse of dimensionality. Also, the window size r is still a hyperparameter that needs to be chosen. Since mean shift clustering requires many nearest neighbor comparisons, it performs slowly for large data sets.

7.6. Gaussian expectation–maximization clustering

Gaussian expectation–maximization (EM) clustering is similar to the K-means algorithm, but instead of using spheres (or other equidistant surfaces corresponding to the norm that is chosen as a measure for distance), the points in each cluster are assumed to be distributed according to an F-dimensional Gaussian distribution, which allows for ellipsoids rather than spheres as bounding regions (see Fig. 41). The principal axes of the ellipsoid are determined by the standard deviation. In the limit of vanishing covariance, the ellipsoids become spheres and Gaussian EM becomes equivalent to K-means. Another way of stating this is to say that the algorithm finds the **maximum likelihood hypothesis**, i.e. the probability that all data points in feature space are actually drawn from K normal distributions with means m_a and variances σ_a, $a = 1, \ldots, K$. Since points are assigned to clusters with a probability determined by the Gaussian, points can belong to several clusters simultaneously, i.e. Gaussian EM allows for mixed membership.

As a first step, we choose the number K of clusters and randomly initialize the K Gaussians of dimension F. Now we compute for each point p_i the probability that it is drawn from the ath Gaussian distribution $\mathcal{N}_a$,

$$P(x_i \sim \mathcal{N}_a) = \frac{1}{\sqrt{2\pi\sigma_a^2}} \exp\left[\frac{-(x_i - m_a)^2}{2\sigma_a^2}\right]. \tag{141}$$

Using this, we assign the probability with which x_i belongs to the cluster c_a via

$$P(x_i \in c_a) = \frac{\omega_a\, P(x_i \sim \mathcal{N}_a)}{\sum_{b=1}^{K} \omega_b\, P(x_i \sim \mathcal{N}_b)}, \tag{142}$$

where the ω_a are new parameters encoding the probability $P(x_i \sim \mathcal{N}_a)$ of picking the distribution $\mathcal{N}_a$ as the one from which the data point is drawn. The denominator is the probability of observing x_i in cluster c_b weighted by the cluster probability ω_b. Since we are summing over all clusters, this is the probability of encountering x_i in our data set, if it is modeled by K Gaussians.

The algorithm iterates the following two steps until convergence:

- E-**step:** In the expectation step, each point is assigned probabilistically to the cluster it is expected to belong to, given the current parameters $(m_a, \sigma_a, \omega_a)$.
- M-**step:** In the maximization step, the parameters $(m_a, \sigma_a, \omega_a)$ are adjusted to find a new maximum likelihood based on the data assignment of the E-step.

Note that this can also be modeled as a minimization procedure of the KL-divergence (79) for the K Gaussians.

7.7. BIRCH

Balanced Iterative Reducing and Clustering using Hierarchies, or BIRCH for short [143], is another unsupervised clustering algorithm. It was designed for especially large data sets that cannot fit into main memory all at once. It is furthermore designed to be robust to "noise" or anomalies, i.e. data points that do not fit the general pattern and should not be clustered at all. Since the algorithm does not access all data when clustering, it is a local clustering algorithm.

As the acronym suggests, it uses a (height-balanced) tree structure to organize the data. So let us first introduce some terminology related to trees (in computer science). A tree is a collection of connected nodes. The nodes are organized in layers or levels. The top level has a single node called the **root node**. The $n^{(1)}$ nodes of the next level are all connected to

the root node at the top level. Let us consider a set of nodes $k_i^{(l)}$ at level l that are all connected to a unique node $k_j^{(l-1)}$ at level $(l-1)$. Then, the nodes $k_i^{(l)}$ are called the child nodes of $k_j^{(l-1)}$, and $k_j^{(l-1)}$ is called the parent node of each $k_i^{(l)}$. Nodes that do not have child nodes are called leaf nodes.

Rather than looking at each point individually, BIRCH introduces sub-clusters. Each sub-cluster represents a set of data points. By using sub-clusters, one can perform clustering even though one cannot fit the entire data set into the memory. Each sub-cluster consists of further, more finely grained sub-clusters, and so on. The algorithm keeps track of this hierarchical inclusion of sub-clusters via a tree: Assume that some cluster $c_i^{(l)}$ at level l can be further decomposed into three finer sub-clusters. Then, $c_i^{(l)}$ has three children $c_j^{(l+1)}$, $j = 1, 2, 3$. With this hierarchical structure, one can find the sub-clusters to which any given data point belongs in a time that is given by $B \times D$ where B is the **branching ratio** (i.e. the maximum number of children of a node) and D is the depth of the tree. Note that for a balanced tree, D grows only logarithmically in the number of points N, $D \sim \log_B N$.

The sub-clusters of our data are characterized by so-called cluster features (CF). These are the number of points in the cluster, the position of the cluster's "center of mass" in feature space, and the cluster's spatial extension around its center of mass. More precisely, the latter two are the mean and the variance of the data points in the clusters and are represented by F-dimensional vectors of linear sums (LS) and square sums (SS) of the coordinates. Hence, a cluster feature is a triple $\mathrm{CF}^{(a)} = (N^{(a)}, \mathrm{LS}^{(a)}, \mathrm{SS}^{(a)})$, $a = 1, \ldots, K$, with K being the number of clusters and

$$\mathrm{LS}^{(a)} = \sum_{i=1}^{N^{(a)}} x_i \,, \qquad \mathrm{SS}^{(a)} = \sum_{i=1}^{N^{(a)}} (x_i)^2 \,. \tag{143}$$

Merging two sub-clusters then simply corresponds to summing the cluster features.

The CF-tree is organized by assigning to each node a set of sub-clusters (i.e. a set of cluster features). The shape of the CF-tree is determined by the branching ratio B, the **threshold** T, which is the maximum radius a sub-cluster in a leaf node can have, and the maximum number L of entries that can be in the same leaf node. The radius R^a of a sub-cluster $\mathrm{CF}^{(a)}$ is defined as

$$R^{(a)} = \sqrt{\sum_{i=1}^{N^{(a)}} \frac{(x_i - c)^2}{N^{(a)}}} = \sqrt{\frac{N^{(a)}(C^{(a)})^2 - 2C^{(a)} \cdot \mathrm{LS}^{(a)} + \mathrm{SS}^{(a)}}{N^{(a)}}} \,, \tag{144}$$

where the F-dimensional vector C^a is the cluster's centroid,

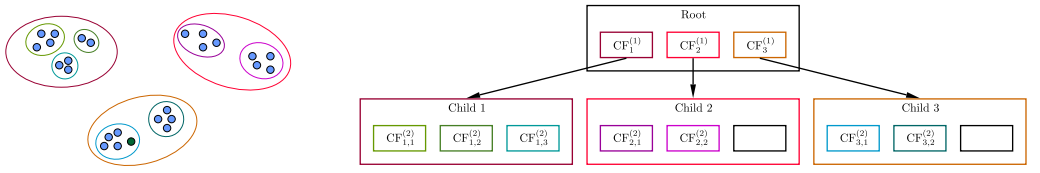
$$C^{(a)} = \frac{\mathrm{LS}^{(a)}}{N^{(a)}} \,. \tag{145}$$

The CF-tree is built sequentially one data point at a time. For each data point, it is decided whether it should be assigned to an existing cluster or whether a new cluster should be created. After the assignment, the tree is re-organized by merging clusters. In more detail, the following steps are performed:

1. Compare the position of a data point x to each CF in the root node. The data point x is now handed down the tree until the "best" sub-cluster for this data point is identified. Here, "best" means the sub-cluster that is closest to x. This is done by first identifying the CF in the root node which is closest to x, then checking all children of this root CF, then checking their children, and so on until the leaf nodes are reached.
2. Depending on the hyperparameters (B, T, L), several things might happen after this update.
    a. If the radius of the new leaf cluster does not exceed T, update the root node and all child nodes to contain this new data point and continue with the next data point.
    b. If the radius does exceed T, create a new leaf cluster, assign the data point to this new cluster, update all parent nodes and continue with the next data point.
    c. If, in the previous step, the leaf node has already L clusters, split the entire leaf node into two leaf nodes. Put the two most distant sub-clusters into separate leaf nodes and assign all other sub-clusters to the leaf node they are closest to. Then update all parent nodes and continue with the next data point.
    d. If, in the previous step, creating a new leaf node exceeds the branching ratio B, split the parent node, and so on. Note that this splitting is controlled by T. A larger T will allow for more points in each cluster, thus reducing the tree width.
3. Once the CF-tree is built by iterating 1. and 2., one can (optionally) modify the resulting tree by removing outliers and merging sub-clusters into larger clusters. Then, one uses a standard clustering algorithm (such as K-means) on the sub-clusters. Since each sub-cluster represents many data points, there are far fewer sub-clusters than original data, such that this becomes feasible.

Note that the result of BIRCH depends on the ordering of the input; different orderings can lead to different trees.

We give a simple example in Fig. 42. We have three clusters. The first consists of three sub-clusters, while the other two consist of two sub-clusters. In this case, the depth is 2 and the branching ratio is $B = 3$. Let us assume the point



(a) Cluster. (b) BIRCH Tree.

Fig. 42. Example for a cluster with sub-clusters and its corresponding BIRCH tree.

that is currently being read is the dark green point. At this stage, the algorithm will go through all cluster features in the root node to identify the sub-cluster this point belongs to. After identifying the third (orange) cluster as the correct one, it will pass the data point down to the third child node and look at its cluster features. It will decide that the light blue sub-cluster is the correct one. As drawn here, the radius does not exceed the threshold and the point is added to the sub-cluster. After that, the cluster features $CF_{3,1}^{(2)}$ and $CF_3^{(1)}$ are updated to account for the new point in the light blue sub-cluster of the orange cluster and the algorithm reads in the next point.

7.8. DBSCAN

Density-Based Spatial Clustering of Applications with Noise, or DBSCAN [144] performs clustering based on how dense points are, i.e. based on how many neighbors they have within a distance r. Usually, one uses again the Euclidean norm to measure distances. Cluster boundaries are formed by points with less than $n_{\min}$ neighbors and points within each cluster boundary are assigned to the same cluster.

In more detail, one defines **core points** as points that have at least $n_{\min}$ neighbors. A point q is called **directly reachable** from p if it is within distance r from a core point p. A point q' is called **reachable** from p if there exists a path from p to q' via core points $p_1, p_2, \ldots, p_n$ such that each p_{i+1} is directly reachable from p_i. Note that q' itself might not be a core point. In that case, it cannot contribute to reachability of other points and is hence at the boundary of a cluster. Points that are not reachable from any other point are classified as outliers or noise or anomalous points.

Note that reachability is non-symmetric: non-core points can be reachable, but no other point can be reached from them. In order to perform clustering, one thus introduces the notion of **density-connectedness**, which is symmetric. Two points p and q are density-connected, if there exists a core point c such that both p and q are reachable from c. A cluster then comprises all points that are mutually density-connected.

The advantage of DBSCAN is that it does not necessitate specifying the number K of clusters beforehand. It can cluster non-linearly separable data and is robust to noise or outliers. Like any other distance-based clustering algorithm, it suffers, however, from the curse of dimensionality for large feature space dimensions F. Furthermore, if the density of clusters fluctuates heavily across feature space, choosing the same value for the minimum number of neighbors $n_{\min}$ and distances r might not be feasible.

If $n_{\min}$ is chosen small, clusters tend to get bigger. Since the density scales with the dimension, common choices are $n_{\min} \sim F$ (meaningful results can then only be obtained for $n_{\min} > 1$). The choice of r is related to the choice of $n_{\min}$. If r is chosen small, there will be many smaller clusters, while a large r will lead to fewer clusters.

In general, the authors of [144] recommend choosing r as small as reasonably possible. In order to identify a good value of r, one can use a k-distance plot and identify an elbow. The k-distance $d_k(p)$ is defined for each point p in the data set to be the distance to its kth nearest neighbor. The authors find that the qualitative result does not change much when varying k and they use $k = 4$. The k-distance plot now sorts the points by their k-distance in descending order and plots the resulting list. If, for any given point p with k-distance $d_k(p)$, one chooses the minimum number of points $n_{\min} = k$ that need to be within a distance $r = k - d_k(p)$ to define a core point, all points with a larger k-distance correspond to noise and all points with smaller k-distance correspond to core points. Thus, the elbow identifies a good value for r that ignores noise but still finds enough clusters.

7.9. Comparison of clustering algorithms

The algorithms described above are all standard and have been implemented in various libraries. We will illustrate their behavior using the Scikit learn library for Python. This library implements all of the clustering algorithms discussed above (and more), and they even include code to generate and compare data sets. We plot the result of the clustering algorithms for $N = 2000$ data points in an $F = 2$ dimensional feature space in Fig. 43. We set $K = 3$ for the first two rows and $K = 2$ for the last two rows for the algorithms that need specification of the number of clusters.

First, by comparing the timing of the different algorithms in the bottom right of each plot, we see that K-Means mini-batch clustering (with batch size 100 in this example) is faster by a factor of 2 than K-Means with essentially equal results. Mean shift is for $N = 2000$ already much slower than the other algorithms, which could be countered by using fewer windows.

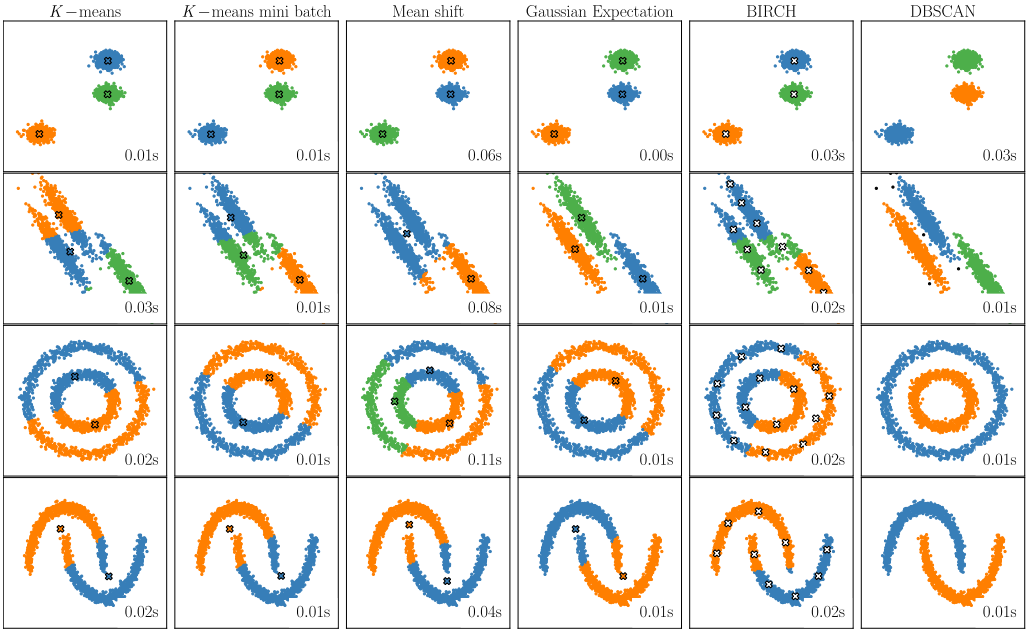


Fig. 43. Comparison of the clustering mechanisms discussed in this section for various cluster shapes. Cluster centers are marked with an ×.

Let us compare the different data sets (i.e. the rows) next. As we can see in the first row, when clusters are separated all algorithms identify the same clusters.

In the second row, the shapes of the clusters of the first row have been changed from spherical to ellipsoidal. One can nicely see the Voronoi cells emerging around the cluster centers for the first three algorithms. However, the results are not the ones a human would come up with. As expected, Gaussian Mixture can deal nicely with elongated clusters. Also DBSCAN, which looks for connected components, identifies the three clusters even if they are elongated. One can also see the outliers detected by DBSCAN in black. BIRCH uses several subclusters, which, at the final clustering step, lead to similar (poor) results as the first three algorithms.

For the third row, we see how all algorithms except DBSCAN produce similar results. DBSCAN finds the two connected rings, while the other algorithms produce Voronoi diagrams (see Section 7.2). In this case, there is no advantage of using ellipsoidal bounding regions from Gaussian mixture. The same is true for the fourth row.

In these simple examples, DBSCAN identifies clusters in a way that seems "correct" to a human. However, the way in which data can be clustered is of course not unique and it might be that the clusters should not be assigned based on the fact that they are almost connected (as done by DBSCAN in the last row), but based on where most data points lie (as done by the other algorithms in the fourth row). Moreover, the chance that two or more clusters align in a plane (as is the case in the third row) or that they carry non-trivial linking number in a high-dimensional feature space is small. In any case, in order to identify such structure in data, it is better to apply topological data analysis (see Section 6) rather than clustering.

8. Reinforcement learning

Reinforcement learning (RL) is a big and active field of computer science. For a standard textbook on the subject see e.g. [145]. RL has recently come to fame when used in Go to beat the world-champion [146,147], but it also finds application in other board and video games, protein folding, robotics, and many more. For an application to string theory see [24].

RL can be used when there is a notion of a good outcome, but it is a priori not clear how to achieve this outcome. It lends itself to applications in searching for good (series of) actions that produce the desired outcome.

The idea is based on behavioral psychology: The algorithm, called **agent** in the RL literature, interacts with its **environment** by taking **actions**. Based on these actions, the agent **steps** through the environment and arrives at a new **state**. Depending on the quality of the state, the agent is **rewarded** or **punished** for taking the action it took from the state it was in. The agent does that over and over, with the goal to maximize its long-term rewards. Usually, punishments are modeled as negative rewards.

RL ranges between supervised and unsupervised learning. It differs from unsupervised learning, since there is a clear notion of a goal that has to be achieved, and the algorithm is optimized to achieve this goal. It differs from supervised

learning in that there is no direct training data set; since we do not know the best strategy for taking steps and earning rewards, we cannot tell the algorithm a priori how to behave. Sub-optimal decisions are not actively discouraged, instead one tries to find a balance between **exploration** of the environment (i.e. getting to know different states and how actions in these states influence future rewards) and exploitation (i.e. select actions based on previous knowledge, rather than trying new actions for the given state). Finding a balance is not always easy and is known as the **exploration vs. exploitation problem**. Usually the algorithms are set up such that the rate of exploration goes down and the rate of exploitation goes up over time as more knowledge about the environment is acquired.

Mathematically, RL can be modeled as a **Markov decision process** (MDP). Given a state s and action a, the agent takes a step to a new state s' and receives a reward for this action. The Markov property is satisfied, since the probability of the agent moving into state s' depends on s and a, but is independent of the previous states and actions that took the agent into s. In this sense, MDPs are extensions of Markov chains by adding actions and rewards. Since the agent performs a series of steps, RL has a notion of time. In search algorithms, subsets of actions might commute or even centralize all actions, while others do not. For subsets that do commute, the order of actions does not matter, and RL can take the action at any point in time, while for others, an action might only become feasible once other actions have been taken. Still, the Markov property implies that if a state can be reached via different paths (i.e. via a different series of actions), it does not matter how the agent arrived at the current state.

Solving MDPs is the subject of **dynamic programming**. RL algorithms differ from the classical solution algorithms in that they use heuristics (often via deep neural networks) rather than solving the MDP exactly. For this reason, RL is sometimes referred to as approximate dynamic programming. The fact that RL uses approximation makes it applicable to large MDPs that cannot be tackled otherwise.

8.1. Ingredients of RL

We have introduced the basic ingredients for RL above. For subsequent discussions, it is beneficial to formalize these definitions.

- **Environment:** The environment $\mathcal{E}$ describes the states and the actions that the agent can take to move from one state to the next.
- **States:** We denote the set of states in the environment $\mathcal{E}$ underlying the MDP we are trying to solve by $\mathcal{S}$. At any **time** step t, the agent is in exactly one state $s_t \in \mathcal{S}$. We sometimes also use the notation $s, s', s'', \ldots$ to denote different states and follow the convention that states with more primes occur later in t. Usually, states are represented as vectors in $\mathbb{R}^{d_S}$. We denote the number of states by $|\mathcal{S}|$; typically $|\mathcal{S}|$ is very large or possibly infinite; if it was not, one could just try all states via brute force.
- **Episode:** Often, the environment contains terminal states. Once the agent reaches one of these terminal states, its task ends and the algorithm stops. This could be a win, loss, or draw in a board game, for example.
- **Actions:** The set of all possible actions in the environment $\mathcal{E}$ is denoted by $\mathcal{A}$. The set of actions possible for a state $s \in \mathcal{S}$ is denoted by $\mathcal{A}(s) \subset \mathcal{A}$. Their cardinalities are denoted by $|\mathcal{A}|$ and $|\mathcal{A}(s)|$, respectively. Often, the possible actions are just enumerated, such that $\mathcal{A}$ is a (finite) interval in $\mathbb{N}$. Typically, the number of actions is much smaller than the number of states, but there are also applications with continuous action spaces.
- **Policy:** A policy π describes how the agent selects an action $a \in \mathcal{A}(s)$ given its current state $s \in \mathcal{S}$. It can thus be understood as a map
$$\pi : \mathcal{S} \to \mathcal{A}\,. \tag{146}$$
If the policy is **deterministic**, it picks out a unique action a given s, $\pi(s) = a$. A **stochastic** policy $\pi(a|s)$ picks, given its state s, the action a according to a certain probability distribution. The policy fully describes the behavior of the agent.
- **Reward:** The reward $r_t \in \mathcal{R}$ is the feedback given to an agent for following his policy π and picking a certain action a_t at time step t given its state s_t. Rewards are typically modeled as real numbers, $\mathcal{R} \subseteq \mathbb{R}$, such that the reward function R is a map
$$R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}\,. \tag{147}$$
- **Return:** The return G_t is the accumulated future reward from the current time step t onward,
$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\,. \tag{148}$$
The factor $\gamma \in [0, 1]$ is known as the **discount factor**. It devalues future rewards in the return and thus incentivizes the agent to take actions that have a high reward in the near future rather than in the distant future. For episodal tasks, the sum truncates after a finite number of steps, i.e. when the agent reaches a terminal state. Note that G depends on π. If π is deterministic, G is fixed since all future actions and thus all future rewards are fixed, while for stochastic π there are many possible returns.

- **State value function:** Since the return depends on π, one defines the state value function (or just value function for short) as the expected return of a state s given the current state s_t,

$$v(s) = \mathbb{E}(G_t \mid s = s_t)\,. \tag{149}$$

  Note that this is in general different from the reward or return, since it captures the expected value of a state s.
- **Action value function:** Similar to the state value function, one defines the action value function

$$q(s, a) = \mathbb{E}(G_t \mid s = s_t, a = a_t) \tag{150}$$

  as the expected return for choosing an action a_t in state s_t at time step t.
- **State transition probability:** The probability for moving from a state s to a state s' with a given action a is denoted by $p(s'|s, a)$. For our purposes, s' is usually fixed given s and a, but in some cases the next state s' is drawn from a probability distribution to model environmental randomness.

The Markov decision problem is characterized by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma)$.

8.2. Example: States and actions for solving a maze

Let us give an example to illustrate these quantities. Assume we are trying to solve a maze, which is an $N \times M$ grid with a single entrance, a single exit, and walls that can occupy one or more grid positions. Furthermore, assume we can move one grid position at a time, either north, east, south or west. This data defines the environment $\mathcal{E}$.

The set of states $\mathcal{S}$ are all accessible grid positions, $\mathcal{S} \subset \mathbb{Z}^{N\times M}$. The unique terminal state is the maze exit, and the episode ends once the agent reaches the exit. The actions are $\{N, E, S, W\} \sim \{0, 1, 2, 3\}$ and they take the agent from a state (i.e. grid position) $g_{i,j}$ to $g_{i-1,j}, g_{i,j+1}, g_{i+1,j}, g_{i,j-1}$, respectively. For a given grid position, some actions might be illegal; for example, if the maze has a wall immediately to the left, the agent cannot execute action W.

For the policy, we want the agent to find the exit, while still exploring the maze. It should not wander off into completely wrong directions or keep running into dead ends, but it should try different paths. The optimal policy depends on the chosen reward function, but probably one would want the agent to solve the maze in as few steps as possible. If the goal was to just solve the maze, the reward could be 1 for reaching the exit and 0 for all other actions. However, this means that the agent has no incentive to not constantly run into walls and dead ends until it proceeds to the exit, so to encourage it to solve the maze as quickly as possible, we can punish it for every step it takes by assigning a reward of -1 to each action.

The return is then simply the negative of the total number of steps taken until the exit is reached. States that are on the (shortest) path from the entrance to the exit will have a higher value than states in dead ends. Likewise, actions that take you from the given position closer to the exit have a larger action value than actions that lead into dead ends.

8.3. The Markov decision problem

To solve the MDP $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, \gamma)$, we need to find the optimal policy π_* **and** the best prediction for the state values $v_\pi(s)$ and action values $q_\pi(s, a)$ for a given policy π. These two problems are called the **control problem** and the **prediction problem**, respectively. Note that they are not independent. The optimization problem we need to tackle involves the following steps:

- Find the optimal state value function $v_*(s)$, which is the maximum value function over all policies π,

$$v_*(s) = \max_\pi\ v_\pi(s)\,. \tag{151}$$

- Find the optimal action value function $q_*(s, a)$, which is the maximum action value function over all policies π,

$$q_*(s, a) = \max_\pi\ q_\pi(s, a)\,. \tag{152}$$

- Find the optimal policy π_*,

$$\pi_* \geq \pi \quad \forall \pi \tag{153}$$

  where the partial ordering "$\geq$" for policies satisfies

$$v_\pi(s) \geq v_{\pi'}(s) \quad \Rightarrow \quad \pi \geq \pi'\,. \tag{154}$$

The idea to solve this is to iteratively improve the policy, use this to improve the value function, use this improved value function again to improve the policy, and so on until convergence. These updates do require knowledge of the state transition probability $p(s'|s, a)$ and are therefore known as **model-based** algorithms. This means that the algorithm keeps track of $p(s'|s, a)$ for each pair (s, a). This is done in **tabular models** by simply storing each transition probability. Note that this table grows as $|\mathcal{S}|^2|\mathcal{A}|$, which makes it completely unfeasible for large state or action spaces.

In contrast, **model-free** algorithms approximate the state transition probability $p(s'|s, a)$ rather than storing all of them in a table. This has the advantage that one need not store a large table. Furthermore, the approximation will return an

estimate for $p(s'|s,a)$ based on previously acquired knowledge, whereas tabular methods act completely randomly if there is no entry in the table for the current state–action-pair (s,a).

There are different common choices for policies. The simplest one is the **greedy policy**, which always selects the best action as the next action. Such a policy will be purely exploiting and not exploring. In order to get some exploration, a simple modification is an **ϵ-greedy policy** with $\epsilon < 1$ which suggests the best action in $(1-\epsilon)$ cases and a random action (usually drawn from a uniform distribution) in ϵ cases. Often, one wants to explore more at early times and exploit knowledge more towards the end of training. This can be done by making ϵ time-dependent and decrease over time. Typical choices for epsilon are 0.1 up to 0.5 initially. We will present an example for an $\epsilon = \epsilon(t)$ that decreases over time in Section 8.5. An improvement to sampling actions randomly from a flat prior is to sample them according to their log probabilities. In practice, this is done using the Gumbel-max trick [148,149].

8.4. Solving the MDP with temporal difference learning

In temporal difference (TD) learning, the return in (148) is estimated using the state value, and this estimate is used to update the value function,

$$\begin{aligned} v(s) \quad &\to \quad v(s) + \alpha(G_t - v(s)) \\ &\approx v(s) + \alpha(r(s') + \gamma v(s') - v(s))\,, \end{aligned} \tag{155}$$

where α is the learning rate, s' is the next state, and $r(s')$ is the reward associated with s'. Note that the estimate for the entire return, which involves an infinite sum in (148), is based on just the reward and the value of the next state s'. Typical values for the learning rate are 10^{-3} to 10^{-1}; we use a value of 10^{-1} in the example in Section 8.5. Time-dependent learning rates are also conceivable. The idea is to start with a low learning rate, since the actions are more or less random anyway in the beginning, and then increase the learning rate over time as good actions become more apparent. However, if the learning rate is increased too early, this can lead to constantly overwriting previously learned information and result in oscillatory behavior. A technique to counter this is to keep track of how often each entry has been updated and decrease the learning rate with the number of updates. Note that this makes the learning rate dependent on the state, $\alpha = \alpha(s)$.

Similarly, the action value function is updated via

$$\begin{aligned} q(s,a) \quad &\to \quad q(s,a) + \alpha(G_t - q(s,a)) \\ &\approx q(s,a) + \alpha(r(s') + \gamma q(s',a') - q(s,a))\,, \end{aligned} \tag{156}$$

where we always follow the same policy (typically ϵ-greedy) to select the next action a'. This means we are constantly updating (and thus hopefully improving) the policy we are using to select actions. Such methods are called **on-policy learning**. Since the updates depend on the tuple (s,a,r,s',a'), the algorithm is known as **SARSA**.

A variation of that is **Q-learning**, which is an **off-policy** algorithm, meaning that the agent follows one policy, but optimizes another. The only difference is that in the action value function update, we use the value of the best possible action that could have been performed in s', i.e.

$$q(s',a) = \text{argmax}_a\, q(s',a)\,, \tag{157}$$

rather than the one corresponding to the action a' that has actually been taken,

$$\begin{aligned} q(s,a) \quad &\to \quad q(s,a) + \alpha(G_t - q(s,a)) \\ &\approx q(s,a) + \alpha(r(s') + \text{argmax}_a[q(s',a)] - q(s,a))\,. \end{aligned} \tag{158}$$

The advantage of this procedure is that one can prove convergence, in contrast to SARSA, given enough exploration time. Nevertheless, SARSA has been demonstrated to work very well in concrete applications. As discussed above, instead of computing and storing the state and action values, they are approximated by neural networks if the state and action spaces are large. In the case of Q-learning, the approximation algorithm is then called **deep Q-learning**.

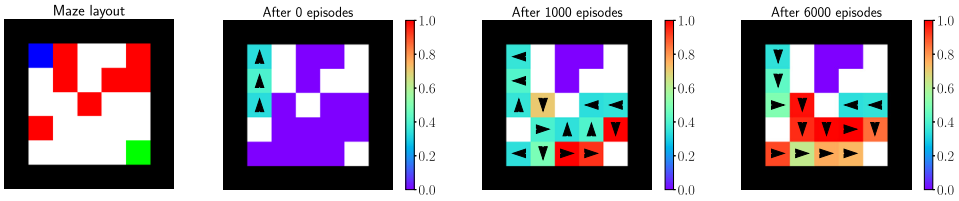
Instead of using just the next state to approximate the return in (155) and (156), we can use the reward of the next N states and approximate the rest of the sum by $v(s_{t+N})$,

$$G^N(t) = \gamma^N v(s_{t+N}) + \sum_{k=0}^{N-1} \gamma^k r_{t+k+1}\,. \tag{159}$$

For $N=1$ this reduces to the case discussed above. Taking $N=T$, where T is the number of steps until the episode ends, we do not perform any approximation at all. On the other hand, this requires playing an episode until the end before we can even perform the first state-value and action-value function updates, which is unfeasible for long episodes. Playing or "unrolling" each episode until a terminal state is reached is known as the **Monte Carlo** approach to solving the MDP.

Another approach, known as the **λ-return** is to estimate $G(t)$ based on a weighted linear combination of $G^N(t)$ with weight parameter λ,

$$G_\lambda(t) = (1-\lambda)\sum_{k=0}^{\infty} \lambda^k G^k(t)\,. \tag{160}$$



(a) Maze layout. (b) 1 episode. (c) 1001 episodes. (d) 6001 episodes.

Fig. 44. Demonstration of the SARSA algorithm to solve a maze. We show in 44a the layout of the maze where the start position is blue, the exit is green, walls are black and pitfalls are red. The subsequent figures illustrate $q(s, a)$ after the specified amount of episodes. The arrow indicates the best action, while the color indicates the certainty for this action being the best. White squares in Figs. 44b to 44d are pitfalls.

Again, playing an episode until the end and updating (155) and (156) based on $G_\lambda(t)$ reduces to Monte-Carlo methods of solving the MDP.

8.5. Example: Solving a maze with tabular SARSA

Before we continue with deep learning in RL in the next section, where the policy and state value functions are approximated by NNs, let us present an example for the SARSA algorithm that learns to solve a maze similar to the one introduced in Section 8.1. In this maze, we make state transitions non-probabilistic and instead of adding walls in the interior we add pitfalls. If the agent falls into a pit, it receives a negative reward of -50, the episode ends and the agent has to start over. If the agent finds the exit, it receives a positive reward of 10^5 and the episode ends as well. Moreover, we end an episode after 10,000 steps and reset the agent to its starting position. Other than that, each step in the maze is penalized by -1 and performing an illegal move, i.e. running into a wall that bounds the maze, is penalized by -2. Note that we add walls as well as pitfalls to demonstrate that the agent learns to avoid illegal moves as well as moves that are legal but incur a large punishment. We furthermore set the learning rate to $\alpha = 0.1$, the discount factor to $\gamma = 0.9$ and use an ϵ-greedy policy with

$$\epsilon = \epsilon(t) = \frac{0.4}{1 + 10^{-4} t} \,. \tag{161}$$

Note that by choosing a large initial ϵ and reducing it each step, we encourage exploration at early times and exploitation at late times.

We randomly generate a maze on a 5×5 grid and include 5 pitfalls, one entry (i.e. a unique start position) and one exit (i.e. a unique end position). We illustrate the updates to the action-value function in Fig. 44 in the following way: we plot in each cell of the maze an arrow indicating the best next action if one followed a (greedy) policy. In case of two or more actions having the same best value, we pick a random one. The color of the cell indicates the percentage with which the agent recommends this action over the other three.

Fig. 44a shows the layout of the maze. The blue square at position $(1, 1)$ is the start position of the agent. The black squares are the outer walls of the maze and cannot be traversed. The red squares indicate the pitfalls, and the green square at position $(5, 5)$ is the maze exit.

In the zeroth episode, the agent chose (randomly) the actions S, W, S, S, i.e. it walked down, into a wall, then down twice and fell into a pit. The initial action "S" leads to a punishment of $r_0 = -1$, such that with $\alpha = 0.1$ we have after this step

$$q(s = (1, 1), a = S) = -0.1 \,. \tag{162}$$

Thus the agent would prefer a different action next time; little does it know the severe punishments of the other actions yet. In any case, the agent transitions to the new state $s_1 = (2, 1)$. In the next step, the agent performs action "W", which is an illegal move and gets penalized with $r_1 = -2$, thus

$$q(s = (1, 1), a = S) = -0.1 \,, \quad q(s = (2, 1), a = W) = -0.2 \,. \tag{163}$$

Since the action is illegal, the agent stays in the same state, $s_2 = s_1 = (2, 1)$. Since the agent still has better alternatives, it does not try W again but instead selects randomly "S", which incurs a punishment of $r_2 = -1$, so that

$$\begin{aligned} &q(s = (1, 1), a = S) = -0.1 \,, \quad q(s = (2, 1), a = W) = -0.2 \,, \\ &q(s = (2, 1), a = S) = -0.1 \,, \end{aligned} \tag{164}$$

and leads to $s_3 = (3, 1)$. Note that next time the agent enters $(2, 1)$ it will try walking N or E, since it received punishments for going W or S. In s_3, the agent chooses action "S" again, leading into a pit with punishment $r_3 = -50$ and ending the

episode. At this point

$$\begin{aligned} q(s=(1,1),a=S)&=-0.1\,, \quad q(s=(2,1),a=W)=-0.2\,,\\ q(s=(2,1),a=S)&=-0.1\,, \quad q(s=(3,1),a=S) \;=-5\,. \end{aligned} \tag{165}$$

We illustrate this in Fig. 44b. Note that for fields (1, 1) and (3, 1), the agent actually only knows that it does not want to go down the next time it enters this field, but the other three directions have not been explored and are equally likely. We just randomly assign up arrows as the best action. The color indicates that the certainty for this action being the best one is in fact only 0.33. Similarly, on the field (2, 1), the agent will choose action N or E next time. We randomly pick N, but the color indicates a certainty of 0.5 (with going east next time also having 50 percent).

After 1000 episodes (see Fig. 44c), the agent has started to randomly find the exit, both by entering from the left and from the top. Hence, the actions for the fields above and left of the exit indicate with almost 100 percent certainty the correct move. It is also interesting to see that the agent is getting fairly certain that the correct move from position (3, 2) is going down; up and right would take it into a pit, and left would take it back where it was coming from. Finally, in Fig. 44d, we see $q(s, a)$ after 6000 episodes. The information of the large reward from the exit at position (5, 5) has back-propagated to earlier fields. We can see the gradient flow with arrows pointing towards the exit and the best action becoming more pronounced the closer the agent gets to the exit field. We also want to point out the sealed off region in the top middle part. Since the agent can never enter there, the SARSA table for $q(s, a)$ stays empty during all episodes. Furthermore, the fields at (3, 4) and (3, 5) are not very well explored as indicated by their light blue color. This happens because the agent has learned that moving down in the fourth row is beneficial, hence these fields are not entered often.

Let us discuss what would change if we used a NN instead of a tabular method. Each time the agent enters a new field, it has to randomly guess an action. Instead, a neural network will learn heuristics and recommend the next move based on previous experience. For the layout of our maze, the heuristic it might learn in the beginning is that moving down is good, since moving into the wall on the left or top, or even into the pits on the right, leads to a higher penalty than just moving down. However, just moving down is not a good strategy either, so the agent will learn eventually that it has to move right on field (3, 1). After that, a good heuristic will be moving right and/or down. Of course, while this works for the particular maze layout, this might not be promising in others.

A problem when exploring the maze is that the agent has to learn several pieces of information that are obvious to us as an outsider:

- The state transitions are non-probabilistic, i.e. the same action always leads to the same next state.
- Maze space is flat, i.e. moving left and down leads to the same state as moving down and left.
- There is only one exit, and the agent has no way of knowing where it is until it randomly found it.

Especially the third point indicates how we could improve the agent's behavior if we had a metric to measure how far away from a good solution the agent is. A simple metric would be to reward actions based on the distance from the exit, such that actions that lead the agent closer to the exit yield a higher reward (or a smaller punishment). Note that, while this is not the case in the randomly generated maze at hand, the shortest path to the exit might be obstructed by a pit. In that case, the agent would have to take a detour, temporarily inflicting a larger punishment, but eventually a larger return. This is why it is important to set up the agent with the goal to maximize the long-term return, not just the short-term reward.

8.6. RL and deep learning

As explained above, for large systems it is not feasible to store transition probabilities. Likewise, storing the state-value function $v(s)$ would require a vector of length $|\mathcal{S}|$ and storing the action-value function $q(s, a)$ a matrix of size $|\mathcal{S}| \cdot |\mathcal{A}|$. In addition, one would need many iterations to solve for optimal policies and state value functions. Instead, we will replace the policy and value function iterations by neural networks with parameters θ_v and θ_q,

$$v_\pi(s) \sim \hat{v}(s, \theta_v)\,, \qquad q_\pi(s, a) = \hat{q}(s, a, \theta_q)\,. \tag{166}$$

In this way, instead of storing $|\mathcal{S}|$ and $|\mathcal{S}| \cdot |\mathcal{A}|$ numbers for the state and action value functions, we only need $|\theta_v|$ and $|\theta_q|$ numbers, respectively. Furthermore, the NNs will perform regression to predict state and action values for previously unknown states s or state–action pairs (s, a). This regression will be based on heuristics learned by exploring the environment, while tabular methods act completely randomly when encountering a previously unknown state.

Value function approximation

As explained in Section 8.3, in order to solve the MDP we need to iterate the value function and policy approximation until convergence. Let us first discuss value function iteration. Assuming we knew the exact state and action value functions, we could define the approximation error resulting from using neural networks in (166) as

$$\begin{aligned} J_v(\theta_v) &= \mathbb{E}_\pi[(v_\pi(s) - \hat{v}(s, \theta_v))^2]\,,\\ J_q(\theta_q) &= \mathbb{E}_\pi[(q_\pi(s, a) - \hat{q}(s, a, \theta_q))^2]\,, \end{aligned} \tag{167}$$

i.e. as the mean squared error between the exact functions $v_\pi(s)$ (or $q_\pi(s,a)$) and their NN approximations (denoted by a hat). The parameter updates would then simply be computed via gradient descent (see Section 3.3),

$$\begin{aligned}\Delta\theta_v &= -\frac{1}{2}\alpha\nabla_{\theta_v}J_v(\theta_v) = \alpha\,(v_\pi(s)-\hat{v}(s,\theta_v))\,\nabla_{\theta_v}\hat{v}(s,\theta_v)\,,\\ \Delta\theta_q &= -\frac{1}{2}\alpha\nabla_{\theta_q}J_q(\theta_q) = \alpha\,(q_\pi(s)-\hat{q}(s,a,\theta_q))\,\nabla_{\theta_q}\hat{q}(s,a,\theta_q)\,,\end{aligned} \tag{168}$$

until the approximation is good enough.

Now, we do of course not have the exact value functions $v_\pi(s)$ and $q_\pi(s,a)$ at our disposal; indeed that is why we approximate them. Thus, we replace them by **estimates** or **targets** T_v and T_q, such that the updates become

$$\begin{aligned}\Delta\theta_v &= \alpha\,(T_v-\hat{v}(s,\theta_v))\,\nabla_{\theta_v}\hat{v}(s,\theta_v)\,,\\ \Delta\theta_q &= \alpha\,(T_q-\hat{q}(s,a,\theta_q))\,\nabla_{\theta_q}\hat{q}(s,\theta_q)\,.\end{aligned} \tag{169}$$

As introduced in the previous section, a good estimate is to use the N-step approximation (159) or the λ-return (160) for T. This has the same limiting cases discussed previously: If each episode is played to the end, $N = N_{\text{end}}$, the N-step approximation becomes a Monte-Carlo method and if $N = 1$, we get the one-step approximation

$$\begin{aligned}T_v &= G_v^1(t) = r_{t+1} + \gamma\;\hat{v}(s_{t+1},\theta_v)\,,\\ T_q &= G_q^1(t) = r_{t+1} + \gamma\;\hat{q}(s_{t+1},a_{t+1},\theta_q)\,.\end{aligned} \tag{170}$$

Policy approximation

Having discussed value function approximation, let us now turn to the second part of solving the MDP, which is policy iteration. Once the value functions have converged to their optimum, the optimal policy is to just act greedily and choose the best action every time. Instead of doing this, we can also approximate the policy directly, rather than inferring it from the value functions. Similarly to the value function approximations, we will use a NN with parameters θ_π and optimize an objective function J with gradient ascent. For episodal tasks, i.e. tasks that end, one typically chooses the expected return for a given state s,

$$J_\pi(\theta_\pi) = v_{\pi_\theta}(s) = \mathbb{E}_{\pi_\theta}[G^1]\,, \tag{171}$$

and maximizes this return via gradient ascent,

$$\Delta\theta_\pi = \alpha\nabla_{\theta_\pi}J_\pi(\theta_\pi)\,. \tag{172}$$

By the **policy gradient theorem** [145], we have

$$\nabla_{\theta_\pi}J_\pi(\theta_\pi) = \mathbb{E}_{\pi_\theta}[\nabla_{\theta_\pi}\log\pi_\theta(s,a)\;q_{\pi_\theta}(s,a)]\,, \tag{173}$$

where $\nabla_{\theta_\pi}\log\pi_\theta(s,a)$ is called the **score function**. Again, we do not actually know $q_{\pi_\theta}(s,a)$, so we need to approximate it, using e.g. Monte Carlo, the N-step return approximation, the λ-return approximation, or another NN approximation $\hat{q}_{\theta_q}(s,a)$. The latter is known as **actor–critic** method.

Actor–critic algorithms

In actor–critic models, the critic is judging the action value and updates the action value function. The actor acts accordingly by updating the policy approximation. We denote the action-value and policy approximations by $\hat{q}(s,a,\theta_q)$ and $\hat{\pi}(s,a,\theta_\pi)$, respectively. For the method to work efficiently, we need the two optimizations of $\hat{q}$ and $\hat{\pi}$ to be (approximately) compatible,

$$\nabla_{\theta_q}\hat{q}(s,a) = \nabla_{\theta_\pi}\hat{\pi}(s,a,\theta_\pi)\,. \tag{174}$$

In such cases, the **compatible function approximation theorem** ensures that the approximation of q by $\hat{q}$ does not spoil the results of the policy gradient theorem and one can use $\hat{q}$ in (173).

In fact, it has also been proposed to not only use the action value function in the policy approximation (173), but the state value function as well. Note that $v = v_{\pi_\theta}(s)$ does not depend on the action and hence including it in (173) does not change the expectation values of the policy gradient theorem. The difference between the action value and the state value for a given policy π_θ is called the **advantage**,

$$A_{\pi_\theta}(s,a) = q_{\pi_\theta}(s,a) - v_{\pi_\theta}(s)\,. \tag{175}$$

Using the advantage as a so-called **baseline** in (173) instead of $q_{\pi_\theta}(s,a)$, we thus get

$$\nabla_{\theta_\pi}J_\pi(\theta_\pi) = \mathbb{E}_{\pi_\theta}[\nabla_{\theta_\pi}\log\pi_\theta(s,a)\,A_{\pi_\theta}(s,a)]\,. \tag{176}$$

Agents that optimize (176) are called **advantage actor-critics**.

8.7. Summary

RL algorithms solve a Markov decision problem. The functions that enter in the solution are the policy, the state value, and the action value, which are inter-dependent. Since this dependence makes the problem very hard to solve in general, a solution is found via recursive iteration until a fixed point is reached. Tabular methods iterate exact policies and value functions, while deep RL methods use neural networks to approximate these functions.

As an example, in advantage actor–critic RL, we can use two neural networks, the policy network and the state value network. The input to the policy network is a vector that uniquely specifies the state of the environment the agent is in. Depending on the nature of the state (a vector of properties, an image, natural language, ...), the subsequent layers consist of fully connected layers, CNNs, RNNs, ... The final layer is a softmax layer with $|\mathcal{A}|$ nodes, assigning probabilities to each possible action. The state value network also has as its input the same vector specifying the state in the environment, but the output is a single number, i.e. the approximated state value. The advantage can be computed by approximating the action value function by $G^N(t)$ and taking the difference with the approximated state value $\hat{v}_{\theta_v}(s)$.

9. Supervised learning — classification and regression

Both classification and regression are typical areas for supervised machine learning. In single membership (also called 1-vs.-all) classification, the machine is trained with pairs (x_i, y_j), where x_i, $i = 1, \ldots, N$ is some input and y_i is a class c_k, $k = 1, \ldots, K$, which the input belongs to, $y_i = C(x_i) = c_k$. Usually, y_i is either encoded as an integer numbering the K classes, $y_i \in \{1, 2 \ldots, K\}$, or as a K-dimensional zero vector with a single 1 at position $\mu = k$. In two-class classification, it is also common to use $y \in \{0, 1\}$ or $y \in \{-1, 1\}$.

The classification problem then consists of assigning previously unseen input to any of the classes c_k, $k = 1, \ldots, K$. Cases in which a single input can belong to several classes are also possible and referred to as multi-class classification problem. For these, the vector $(y_i)_\mu$ has zeros everywhere except at the positions that correspond to the classes that x_i belongs to. In regression problems, the machine is trained with pairs (x_i, y_i), where x_i is again an input and y_i the corresponding output. Often $x_i \in \mathbb{R}^F$ and $y_i \in \mathbb{R}^T$.

We have already discussed how neural networks can be used for both classification and regression tasks. In the following, we will introduce other common algorithms that can be used. Note that, since we are dealing with supervised learning, most of the methods are ensemble methods, thus addressing the bias–variance problem introduced in Section 3.1.

9.1. k-nearest neighbors

The k-nearest neighbor (kNN) algorithm can be used in classification or regression. It is probably one of the simplest algorithms for either task. In either case, the value for the input in question is assigned based on the value of the k nearest neighbors. Note that this distance-based method means it suffers, similarly to the clustering algorithms of Section 7, from the curse of dimensionality (see Section 7.1). This is why kNN is often applied after Principal component analysis (PCA, see Section 7.3). Note furthermore that the algorithm is very local, i.e. it only depends on the close vicinity of the point in question and does not take into account global structures or properties of the data set. Due to this, it is also sensitive to noise.

In typical applications k is small, of order 1 to 10, and the Euclidean distance is used. If one uses this distance measure, one should design features such that input that is similar is indeed close in feature space with respect to this metric (see neighborhood component analysis below on how to do this in an automated fashion). In statistical applications, it is also sensible to use the **Mahalanobis distance** [150], which is defined as

$$d_{\mathrm{M}}(x_1, x_2)_{\mathrm{MD}} = \sqrt{(x_1 - x_2)^T \cdot C^{-1} \cdot (x_1 - x_2)}\,, \tag{177}$$

where C is the covariance matrix, cf. Section 7.3. Note that if C is the identity matrix, this reduces to the Euclidean distance. For cases where the input is a string over a finite alphabet Σ (e.g. in text processing, the alphabet is the actual alphabet, i.e. $\Sigma = [A, B, \ldots, Z]$, or in cases where one works over finite fields of characteristic p, the alphabet is $\Sigma \sim \mathbb{Z}_p$), a common choice for distance is the **Hamming distance**. This is defined as the minimum number of positions at which the two input strings differ. This assumes that all input strings are padded to equal length. Finding a good metric is the subject of **metric learning** (see large margin nearest neighbor metric learning below).

In classification, the input is assigned to the class which the majority of the nearest neighbors are in. In case of a tie, some tie-breaking mechanism can be used, like including one more nearest neighbor or assigning one of the tied classes randomly. If used in regression, the output value $\hat{y}$ is the average of the values y_j of the k nearest neighbors x_i of the input x in feature space. Denoting the index set of the k nearest neighbors of y by N, the average could just be the arithmetic mean,

$$\hat{y} = \frac{1}{k} \sum_{j \in N} y_j\,. \tag{178}$$

Instead of this, some other averaging method could be used, such as average weighted by the inverse distance,

$$\hat{y} = \sum_{j\in N} \frac{d(x,x_j)}{d_{\text{tot}}} y_i \,, \tag{179}$$

where $d_{\text{tot}} = \sum_{j\in N} d(x,x_j)$ is the total distance of all kNN. This reduces to the arithmetic mean for k equidistant points. The quality of the kNN algorithm can be evaluated using the methods introduced in Section 3.8.

There are several improvements to the algorithm, especially on how to choose a good metric. The idea behind this **metric learning** is to look at the labeled data and find a metric for feature space in which points with the same label are close to each other. Two popular algorithms are **neighborhood component analysis** (NCA) [151] and **large margin nearest neighbor** (LMNN) [152]. The former finds a linear transformation on feature space, keeping the Euclidean distance as a metric, while the latter finds a suitable matrix which can serve as a metric. In this sense, NCA finds the vielbein while LMNN finds the metric.

Neighborhood component analysis

In NCA, one wants to find a transformation matrix A such that classification with Euclidean distance on an input $A\cdot x$ is maximized. Since this is an optimization problem, it is desirable to use iterative methods to approximate the maximum, like gradient descent. The problem is that the notion of nearest neighbor changes discontinuously for continuous changes in A, rendering gradient-based methods inapplicable. To remedy this, one can perform optimization based on the softmax function applied to the Euclidean distance between data points, $\|A\cdot x_1 - A\cdot x_2\|_2^2$ rather than to $A\cdot x$ directly. We define

$$p_{ij} = \begin{cases} \frac{\exp\left[-\|A\cdot x_i - A\cdot x_j\|_2^2\right]}{\sum_{r=1}^N \exp\left[-\|A\cdot x_i - A\cdot x_r\|_2^2\right]} & \text{if } j\neq i \\ 0 & \text{if } j = i\,. \end{cases} \tag{180}$$

Note that the sum runs over all N input–output pairs. The probability with which a point x_i is classified correctly is then

$$p_i = \sum_{\substack{j \\ C(x_j)=C(x_i)}} p_{ij}\,, \tag{181}$$

where $C(x_i)$ is the class which x_i belongs to. From this, we define the objective function f as the expected number of correctly classified points,

$$f(A) = \sum_i p_i = \sum_{i=1}^N \sum_{\substack{j \\ C(x_j)=C(x_i)}} p_{ij}\,, \tag{182}$$

and maximize it using e.g. gradient ascent. With the expression for the gradient of the softmax function, we end up with

$$\nabla_A f = -2A \sum_{i=1}^N \sum_{\substack{j \\ C(x_j)=C(x_i)}} p_{ij} \left((x_i - x_j)^2 - \sum_{r=1}^N p_{ir}(x_i - x_r)^2 \right) . \tag{183}$$

Maximizing f is equivalent to minimizing the L_1-norm between the predicted and the actual class distribution. Alternatively, one could minimize the Kullback–Leibler divergence (see Eq. (79)),

$$g(A) = \sum_{i=1}^N \log(p_i)\,. \tag{184}$$

Large margin nearest neighbor

In large margin nearest neighbors one does not transform feature space but finds instead a positive semi-definite matrix G that induces a metric,

$$d(x_i,x_j) = (x_i - x_j)^T \cdot G \cdot (x_i - x_j)\,. \tag{185}$$

For a given point x_i, one defines the **target neighbors** to be k distinct points x_j, $j = 1,\ldots,k$ that have the same class as x_i. We denote this set of points by T_i. One furthermore defines **impostors** to be points which are nearest neighbors of x_i but belong to a different class, $C(x_i)\neq C(x_j)$. We thus want to find a metric in which target neighbors are close and impostors are far away. Since there is no natural scale in the problem, we define that far away means at least one unit further away than the target neighbors. Finding a metric in which target neighbors are close can be achieved by minimizing a loss given by the average distance between points and their target neighbors,

$$L^{\text{Target neighbors}} = \sum_{i,j\in T_i} d(x_i,x_j)\,. \tag{186}$$

The loss for the impostors can most conveniently be defined using a version of Hinge loss (82) to clip penalties for points that are further away than one unit, i.e.

$$L^{\text{Impostor}} = \sum_{i,j\in T_i} \sum_{\substack{r \\ C(x_r)=C(x_i)}} \max\left[0, (d(x_i,x_j)+1) - (d(x_i,x_r))\right] . \tag{187}$$

We now need to minimize the combined loss $L^{\text{Target neigbors}} + L^{\text{Impostor}}$, using e.g. again gradient descent.

Summary of nearest neighbor algorithms

Nearest neighbor algorithms assign values to points based on the values of the surrounding points. Their result depends on the metric in feature space, which determines which point are neighbors. If no natural metric is known to the user, metric learners can be used to find good metrics, either by transforming feature space itself and keeping the standard Euclidean metric (NCA), or by keeping feature space and learning an appropriate metric instead of the Euclidean one (LMNN).

9.2. Decision trees and random forests

Decision trees are also supervised algorithms that can be used in classification and regression. As the name suggests, they perform the respective task by building a tree-like structure during training. The nodes of the tree perform boolean tests on the input features and then split subsequently into according branches.

Decision trees work for inputs where the features are categorical (i.e. the μth feature is e.g. the name of a country) or numerical (i.e. the μth feature is in $\mathbb{R}$). If the features are numerical, they have a natural ordering, and the boolean test is of the form $x_i^\mu \leq \kappa$ for each input x_i and feature space direction μ. If the features are categorical, they do not have a natural ordering, and the boolean tests are of the form $x_i^\mu \stackrel{?}{=} f$ where f is any of the possible classes that the μth feature can be in (in the example above, this would split the tree according to whether or not the country feature of input x_i is equal to country f).

If each node performs exactly one split, the node will have exactly two children, resulting in a binary tree. If a node performs n boolean comparisons, the node will have $n+1$ children, resulting in an $(n+1)$-ary tree. In general, different features x_μ can have different domains, such that some nodes can have more direct children than others. The tree is built recursively by continuing to perform comparisons for each component of the feature vector. The leaves then contain the value that is to be assigned to an input that satisfies all criteria of all parents.

One of the big advantages of decision trees is that they can be easily represented graphically and interpreted by a human, which is why they are sometimes referred to as "white box" models. Since the most relevant features are on the top of the tree (i.e. close to the root nodes), while features that do not lead to a distinction of input at all will not be part of any node of the tree, decision trees can also serve as feature reduction models. As is always the case with trees, the cost of using the tree grows proportional to the depth of the tree, i.e. logarithmically in the number of nodes for balanced trees.

A disadvantage is that the trees can become very long or very unbalanced (meaning that some decision branches are much deeper than others). Methods that lead to a more balanced tree are called **tree pruning**. The idea is to remove the least significant leaves and sub-branches to create a more balanced tree.

Another disadvantage is that decision trees are unstable, meaning that a small change in the input variables can result in a completely different tree.

Also, while accessing tree nodes scales logarithmically with the number of nodes, training a decision tree (i.e. finding boolean splitting criteria at each node that minimize the expected number of tests needed to classify some unknown input) is very hard (NP-complete) [153,154]. For this reason, heuristics and approximations are used in training which optimize decisions locally at each node. This can lead to solutions that are non-optimal globally. Also, decision trees often suffer from overfitting, especially if the number of features is large. Several improvements to the original decision tree implementations have been proposed to mitigate these problems. Often, the performance can be improved by first using feature reduction algorithms such as PCA (see Section 7.3) before applying decision trees.

CART

An early but very popular decision tree algorithm is the **Classification and Regression Tree** (CART) [155]. It is a binary decision tree, i.e. it performs exactly one boolean check at each node. For both classification and regression tasks, the decision trees are trained using **recursive binary splitting**. This splitting leads to a partitioning of feature space into regions with linear boundaries. Thus, each node is associated with a region in feature space that is bounded by lines that indicate the (in)equalities of all boolean criteria of all parent nodes.

Finding split points: In order to decide the split, we have to decide two questions:

1. Which feature do we use for splitting?
2. Given a feature, which value of the feature should we use for splitting?

In the simplest approach, one iterates over all features, and tries all possible split values for each feature.[13] One then computes the loss of splitting this feature at this point, and chooses the split that minimizes the loss after splitting. The loss depends on whether one is facing a regression or a classification problem, as discussed next. Note that this search strategy becomes very inefficient if there are either a lot of features, or if a feature can take many different values. This is why it is a good idea to perform feature reduction first.

Let us denote the number of different values of the μth feature by N^μ. For numerical features, there will be $N^\mu - 1$ splits. If N^μ is large, one can just try to perform T equally spaced splits,

$$\min_i(x_i^\mu) + \frac{k}{T}(\max_i(x_i^\mu) - \min_i(x_i^\mu)), \quad k = 1, 2, \ldots, T-1\,. \tag{188}$$

If the features are categorical, there are $2^{N^\mu - 1} - 1$ different ways of splitting the set into two inequivalent subset. If N^μ is large, this becomes prohibitively expensive. If at least the output y_i is numerical, one can order the classes of the μth feature by their average output value and then perform a numerical split as above based on these values.

Cost functions: Let us discuss the cost functions next. For classification one typically uses the **Gini index** or **Gini impurity**. This measures for each node how "impure" the data is, i.e. it measures for the subset of input data that ends up on this node (meaning the input data that satisfies all boolean criteria in the path from the root node to the current node) whether they actually do belong to the same class. In more detail, let us look at a node a. The boolean decision boundaries introduced by its ancestor nodes will define a region in feature space. Let this region contain $|N_a|$ elements x_i, $i \in N_a$, and compute the fraction of elements that belong to class c_k, $k = 1, \ldots, K$ via

$$p_{a,k} = \frac{1}{|N_a|}\sum_{i\in N_a}\delta_{C(x_i),y_k}\,. \tag{189}$$

The Gini impurity at this node is now given as the sum over the fractions of elements that belong to a class c_k times the fraction of elements that belong to any other class,

$$G_a^K = \sum_{k=1}^{K} p_{a,k}(1 - p_{a,k})\,, \tag{190}$$

As a simple example, consider the case of just two classes, and let p be the fraction of elements in class c_1, so that $(1-p)$ is the fraction of elements in class c_2. A perfectly pure node would have all its elements assigned to just one class and hence its Gini impurity is zero. A completely impure or mixed node would have half of the input assigned to its class and the other half to the other class, hence its impurity would be 0.5. This makes the Gini impurity a good function for the loss to minimize in the classification task.

In regression mode, one usually uses the mean square error between the prediction and the labels. Thus, at node a with $|N_a|$ elements in its region, one defines the mean

$$y_a = \frac{1}{|N_a|}\sum_{i\in N_a} y_i\,, \tag{191}$$

and uses the deviation of the elements from this mean as a loss,

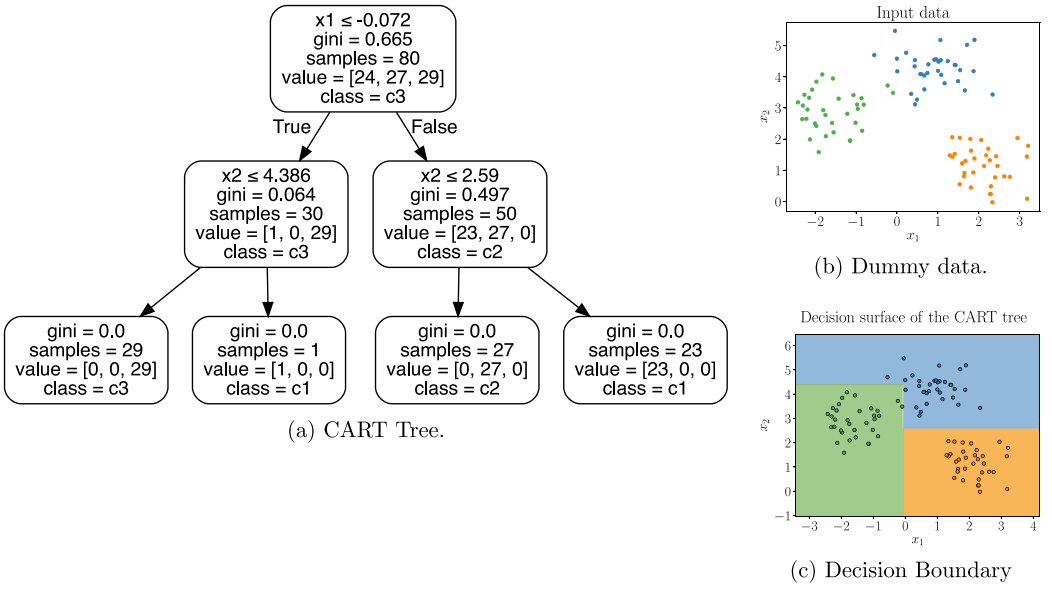
$$L^{\text{MSE}} = \frac{1}{|N_a|}\sum_{i\in N_a}(y_i - y_a)^2\,. \tag{192}$$

Of course, one could also use different norms for the error function.

Stop criteria for splitting: There can be several criteria for when to stop splitting. A naive but potentially not very useful criterion is to stop splitting once a further split does not positively impact the loss anymore. For classification tasks this will be the case once all inputs with the same class end up at the same leaf node. For regression, this is the case once all inputs with the same value end up at the same leaf node. Of course, there is the trivial solution of putting each input into its own leaf node, which is why one usually uses other constraints. One possibility is to constrain the maximum depth of the tree. Typically, this depth is chosen small, especially if one wants to interpret the data. Another possibility is to constrain the minimum number of points in each region. In either case, chances are that one ends up with leaf nodes which do not all have the same value. In classification, the class of a leaf node is then chosen to be the class of most elements in its associated region. In regression, the value of a leaf node is the mean of the values of all elements in its associated region.

We present an example for a CART decision tree in Fig. 45.

[13] If the values of a feature are continuous, one tries splitting at each value encountered in the training set.



(a) CART Tree.

(b) Dummy data.

(c) Decision Boundary

Fig. 45. Example for a CART tree and its corresponding decision boundaries.

ID3, C4.5 and C5.0

ID3 is short for **Iterative Dichotomizer 3** [156], and C4.5 [157] and C5.0 are successors of ID3 with various improvements. Let us just discuss ID3, which is used in classification.

Instead of trying all features in each step, ID3 splits on each feature only (at most) once. Moreover, instead of splitting according to the Gini impurity, ID3 chooses its splits so as to maximize information, or equivalently to minimize entropy. Using the same notation as in (189), one defines the entropy in the usual way,

$$S(N_a) = -\sum_{k=1}^{K} p_{a,k} \log_2 p_{a,k} \,, \tag{193}$$

see (71). Note that, similarly to the Gini impurity, a pure node (where all elements belong to the same class c_l) has $p_{a,l} = 1$ and hence zero entropy.

The algorithm now proceeds as follows. At the root node, where N_a contains the indices of all input data, it computes the entropy for each of the F features in the input vectors $x_i \in \mathbb{R}^F$ and splits depending on the feature with the least entropy, which is the feature that encodes the most information. Depending on the implementation, the split could be binary as in CART or n-ary for this feature. In the next step, for each of the children of the root node, one computes again the entropy, using the other $F - 1$ features. This procedure is iterated until either

- All points in the region of a node belong to the same class. Then this node is turned into a leaf associated with this class and not split further.
- All features have been used. In this case, the node is turned into a leaf associated with the class shared by most points in its associated region.
- There are no elements in the region associated with that node. In this case, a leaf node is created whose class is the class of the majority of the points in the parent node.

Boosted decision trees

The idea of boosted decision trees is to take the initial result of a DT and successively improve it (we briefly mentioned boosting in Section 3.2 as a means to reduce the variance). Here, "improving" means with respect to a given loss function. The best direction of improvement is given by the gradient of a loss function. The loss function should be differentiable and a measure for the quality of the results. Typically, for classification problems one uses cross entropy loss (of the softmax of the classes, see (76)). For regression one uses MSE or MAE loss (see (64) or (66), respectively). A popular implementation is **extreme gradient boosting** (XGBoost) [158].

In more detail, we start with a decision tree DT_1 which produces outputs $\hat{y}_i^{(1)} = \mathrm{DT}(x_i)$. Now we compute the loss $L(y_i, \hat{y}_i)$ and from it the gradient

$$r = -\nabla_{\hat{y}} L(y_i, \hat{y}_i) \,. \tag{194}$$

This quantity r is called the **pseudo-residual**. Next, we fit a regressor (usually another decision tree) to this loss function, i.e. we train another decision tree DT_2 with pairs (x_i, r_i). The idea is that if we can predict the loss based on the x_i, we can correct our previous prediction $\hat{y}_i$ by $\mathrm{DT}_2(x_i)$,

$$\hat{y} \to \hat{y} + \alpha\ \mathrm{DT}_2(x_i)\,, \tag{195}$$

with learning rate α. Of course we need not stop there. We can again compute the loss and train another regressor to fit the new loss, and so on. This procedure can be repeated until the maximum number of iterations is reached or the pseudo-residuals become smaller than a certain threshold.

Adaboost

A different version of boosted decision trees is the **Adaptive Boosting** (AdaBoost) algorithm [159]. The idea is to successively train several decision trees DT_t, just like in the previous case. In the end, we will determine the output value for some input x by a weighted vote of the several decision trees, so we introduce a weight $w^{(t)}$ for each tree. Adaboost is often used for binary classification problems, so let us focus on this case. This is not too restrictive, since a classification problem with K classes can be reformulated as K binary classification problems, each classifying whether or not an input X belongs to class c_k, $k = 1, \ldots, K$. Let us label the two classes for a given input x_i by $y_i = \pm 1$. Each tree would then give a prediction $\hat{y}_i \in [-1, 1]$, where the sign determines the class to which x_i belongs and the absolute value indicates how certain the tree is of this classification.

Instead of training regressors on the residual of losses as for XGBoost, we train each tree with the original input–output values, weighting elements that have been previously misidentified stronger for the next tree. So we introduce another set of weights $\theta_i^{(t)}$ for each input–output pair (x_i, y_i). Hence, for this algorithm, we need decision trees that can deal with weighted losses, i.e. pay more attention to getting some input–output pairs correct than others.

For the first tree, we just use a usual decision tree, i.e. we weigh each input–output pair with the same weight $\theta_i^{(1)} = 1/N$. Now, we run the DT, compute its predictions $\hat{y}_i^{(1)} = \mathrm{DT}_1(x_i)$, and from them the weighted error (or loss) for this first tree. For binary classification, the weighted error is the weighted number of misidentified points,

$$E^{(t)} = \frac{\sum_{i=1}^{N} \theta_i^{(t)} (1 - \delta_{y_i, \hat{y}_i^{(t)}})}{\sum_{i=1}^{N} \theta_i^{(t)}}\,. \tag{196}$$

Based on this error E, we can compute the weight $w^{(t)}$ this tree will have in the final vote. If the tree has a low error, its weight should be amplified and if it has a high error, its weight should be diminished. One thus sets

$$w^{(t)} = \frac{1}{2} \log\left(\frac{1 - E^{(t)}}{E^{(t)}}\right)\,. \tag{197}$$

Note that trees with weighted accuracy of less than 50 percent will actually get a negative vote contribution, i.e. one takes the opposite of what they say. Since they classified less than 50 percent correct (again, subject to the weights) in a binary classification task, this seems fair.

For the next tree, we now update the weights $\theta_i^{(t)}$ such that the next tree pays more attention to the points that have been misclassified by the previous tree(s),

$$\theta_i^{(t+1)} = \frac{1}{M} \theta_i^{(t)} e^{-E^{(t)} y_i \hat{y}_i^{(t)}}\,, \tag{198}$$

where M is a normalization constant for the weights. Note that the product $\hat{y}_i^{(t)} y_i$ is positive if $\hat{y}_i^{(t)}$ predicts the correct class $y_i = \pm 1$. However, we also have a factor $E^{(t)}$, which can have both signs. So if the prediction for the class is correct but $E^{(t)}$ is negative, the sign will change. But that is what we want, since a negative $E^{(t)}$ means that the final vote of this tree will be inverted, since it got less than 50 percent correct. So altogether, since there is an overall minus sign in the exponent, the weights for the next tree of correctly predicted labels will be exponentially damped, while the weights of incorrectly classified labels will be exponentially enhanced. This means that the next tree focuses strongly on predicting previously misclassified points correctly.

The above is repeated for a total of T trees, and the final prediction $\hat{y}$ for some input x is then

$$\hat{y} = \sum_{t=1}^{T} w^{(t)}\ \mathrm{DT}_t(x)\,. \tag{199}$$

Random forests and bootstrap aggregated decision trees

Random forests are also based on several trees voting on the outcome (like the Entmoot in Lord of the Rings, but faster). However, instead of boosting, i.e. sequentially improving the prediction of the last tree, random forests use a concept called **bagging** or **bootstrap aggregation**, in which T trees are trained in parallel on different random subsets of the training set. They thus address the bias–variance problem by reducing the variance by averaging over several weak learners. Note that this bears some similarity with dropout layers in NNs, which effectively also lead to different sub-NNs voting on the final result (see Section 3.6). **Rotation forests** are also random forests, but each tree in the forest applies PCA to the subset of its input values prior to running the decision algorithm.

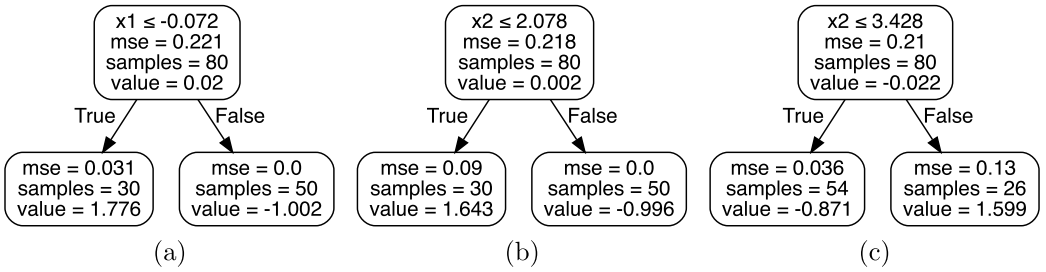


Fig. 46. Three gradient-boosted trees to classify the dummy data of Fig. 45b.

Summary and comparison of different decision trees

Decision trees assign values based on subsequent boolean comparisons, which makes their outcome easy to interpret. Their results can be improved by using ensemble methods, i.e. by training several trees. Boosted trees successively improve the result of the previous tree, while forests take the average of several trees that operate independently. Thus, if a model suffers from overfitting, use random forests. If its results are poor even on the training set, try boosting.

Let us finally use a toy example to illustrate the different tree algorithms. As for the clustering algorithms in 7.9, we generate simple dummy data to keep the trees simple. In fact, the data is simple enough for all models to achieve 100 percent accuracy. We use data belonging to three classes c_k, $k = 1, 2, 3$ in a two-dimensional feature space with features $x_1, x_2 \in \mathbb{R}$, cf. Fig. 45b. There are 80 data points in total; 24 in class c_1 (blue), 27 in class c_2 (orange), and 29 in class c_3 (green).

We first discuss the results of the CART tree. The root node gets associated with class c_3, since this class has most points. This leads to a Gini impurity of 0.665: Indexing the root note by 0, we first compute $p_{0,k}$ following (189),

$$p_{0,1} = \frac{24}{80}\,, \qquad p_{0,2} = \frac{27}{80}\,, \qquad p_{0,3} = \frac{29}{80}\,. \tag{200}$$

Then, the Gini impurity (190) is

$$G_0^3 = \frac{24}{80}\left(1 - \frac{24}{80}\right) + \frac{27}{80}\left(1 - \frac{27}{80}\right) + \frac{29}{80}\left(1 - \frac{29}{80}\right) \approx 0.665. \tag{201}$$

The algorithm decides to split on x_1 by introducing a boundary $x_1 \leq -0.072$, splitting c_3 off from c_1 and c_2. Looking at the decision boundary diagram in Fig. 45c, we find that the region to the left of vertical line now contains all green dots plus one blue dot at around $(-0.5, 4.5)$. So in the left branch, the DT introduces another boundary at $x_2 = 4.368$, separating the 29 green dots from this one blue dot (the horizontal line between the green and the blue region in Fig. 45c). This corresponds to the two left leaf nodes. Both of these have Gini impurity 0, indicating that they are pure nodes, i.e. all points that end up in this node correspond to only one class. Going back to the root and concentrating on the right branch of the tree (which is also the right part in the diagram of Fig. 45c), we can separate the orange and the blue class with one more line around $x_2 = 2.5$. Indeed, the DT splits on feature x_2 at $x_2 = 2.59$, separating the 23 blue dots (remember that one is to the left of the line $x_1 = -0.072$) from the 27 orange dots. Again, the leaf nodes are pure and the Gini impurity is zero.

The result for the ID3 tree looks exactly the same (except for the fact that it splits on entropy and not on Gini-impurity). So we content ourselves with computing the entropy at the root note just for the sake of illustration. Using (200) in (193), we find

$$S(N_0) = \frac{24}{80}\log_2\frac{27}{80} + \frac{27}{80}\log_2\frac{24}{80} + \frac{29}{80}\log_2\frac{29}{80} \approx 1.581. \tag{202}$$

Likewise we find for the child nodes of the root node $S(N_1) \approx 0.211$, $S(N_2) \approx 0.995$ and zero for all leaf nodes, since these are pure nodes.

Let us next look at the gradient-boosted tree in Fig. 46. We have a classification problem with three classes, which can, as explained above, be converted into 3 binary classification problems. The first tree made it its task to decide membership in the green class c_3. It introduces the same decision boundary $x_1 = -0.072$ as the CART tree. As can be seen from Fig. 45c, this provides a perfect classifier for points not in c_3, and gets most points in c_3 right, up to the one blue dot that also falls into this class. Getting non-membership right means that the MSE of the right leaf is zero. Since membership of c_3 is almost correct on the left leaf, the MSE is also rather small for the left leaf. The second tree made it its task to decide membership of the orange class c_2. It uses the decision boundary $x_2 = 2.078$. Everything above that line is not in c_2, which is why the MSE of the right leaf is again zero. Note that the CART tree introduced a boundary at $x_2 = 2.59$, which was right in the middle between the blue and the orange class. The reason why the gradient-boosted classifier introduces the boundary as low as possible (meaning at the minimum x_2 that still separates off all points that are not in the orange class) is to include as few green points as possible. Since there are 27 orange points, and this leaf has 30 samples, we

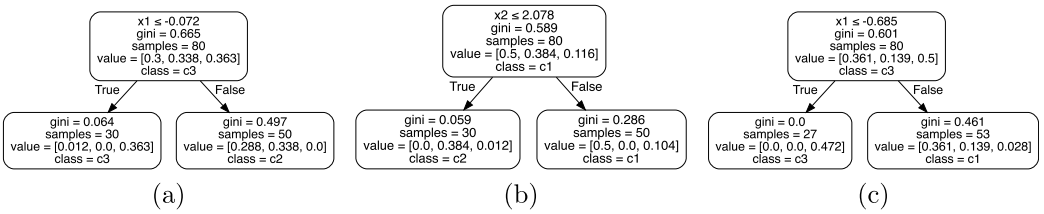
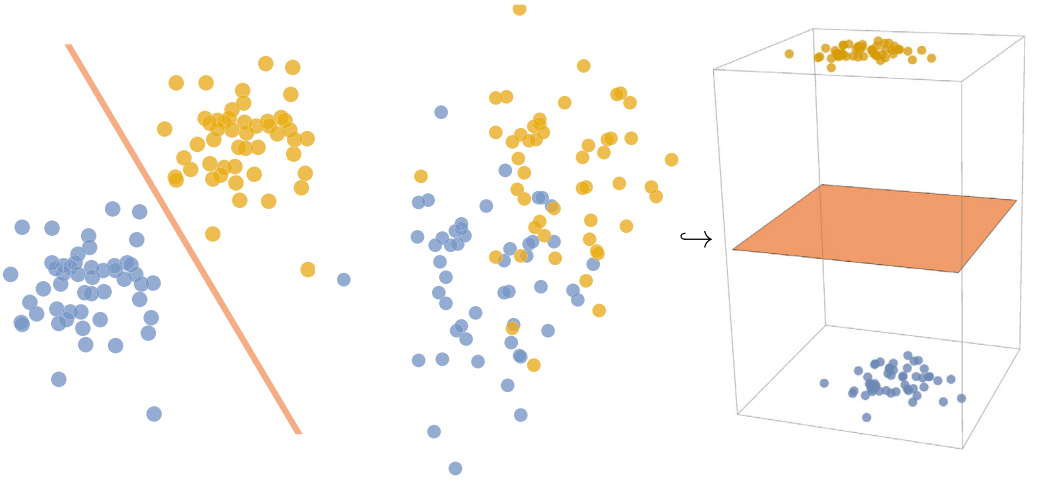


Fig. 47. Three ADABoost trees to classify the dummy data of Fig. 45b.



(a) Linear separable in 2D. (b) Not linear separable in 2D but in 3D.

Fig. 48. Basic idea behind SVMs.

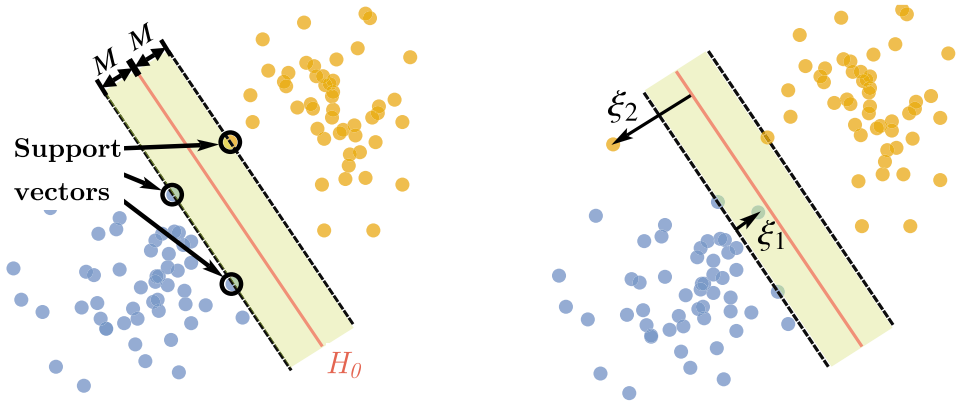
see that the region below this decision boundary includes 3 green points. The third tree introduces another horizontal boundary, this time at $x_2 = 3.428$. This boundary mostly separates the blue class from the others, but not exactly. It was found such that the sum of all three decision trees gives the right answer for the 1 misclassified blue point (in the top left) of the first tree and the 3 misclassified green points (in the bottom left) of the second tree. Note that the boundary is roughly in between the 4 misclassified points.

We present the trees resulting from AdaBoost in Fig. 47. The first tree in the boost sequence does precisely the same thing as the root node in the CART tree, i.e. it introduces a decision boundary at $x_1 = -0.072$, splitting off (most of) the green class 3 from the others. This by itself would be a poor classifier, but since there are most elements in c_3, that is the best you can do with a single split on a single feature. The second tree improves on that by splitting the original data set horizontally at $x_2 = 2.078$, which separates the blue and orange classes, but splits the green one in the middle (however, the first tree is already good at identifying points in the green class). Lastly, the third tree splits again vertically at $x_1 = -0.685$ to account for the blue dot that is misclassified by the first tree. Note that even though most leaf nodes have non-zero Gini-impurity, the combined result still leads to 100 percent accuracy on the example data set.

9.3. Support vector machines

Support vector machines (SVMs) [160] can be used in regression or classification. Like decision trees, they introduce a linear decision boundary of codimension one in feature space. However, instead of including more and more boundaries like trees, SVMs always just use a single hyperplane. In order to be able to deal with data in classification tasks that are not linearly separable (i.e. that cannot be separated in distinct classes with a hyperplane), or with data that is not linearly dependent for regression, they map data points into a higher-dimensional feature space such that data becomes linearly separable. Note that the map need not be linear itself, which amounts to warping the data (or the space). This is called the **kernel trick** in the literature [161,162], see Fig. 48.

The name "support vector machine" is derived from how the hyperplane is found. For classification, the hyperplane should be introduced such that it has the largest possible distance between the data points of two different classes that come closest. This is to reduce the generalization error; in the gap between the classes we do not know whether future data points will belong to one class or the other.



(a) Separating hyperplanes and margins. (b) Slack variables.

Fig. 49. Illustration of quantities entering the SVM algorithm for finding the hyperplane decision boundary.

The closest data points of two different classes are the **support vectors** for the hyperplane, i.e. they determine the hyperplane's position. The normal vector specifying the plane can be found via gradient descent (see Section 3.3) with Hinge loss (see Eq. (82)) and L_2-norm regularization (see Section 3.3.1), or equivalently by solving a constrained quadratic programming problem.

Let us explain this in more detail. We start with SVMs as binary classifiers. As mentioned before, cases with K classes can be reduced to K binary classifications. We first explain how to find the linear decision boundary for cases that are linearly separable (called **hard margin**) and then how to find the best decision boundary for non-linearly separable cases (called **soft margin**). After that we explain the kernel trick and finally how to apply SVMs in regression.

SVMs as binary classifiers

We fix a hyperplane H_0 in $\mathbb{R}^F$ by specifying its (not necessarily normalized) normal vector w and distance b,

$$H_0: \quad w \cdot x + b = 0\,. \tag{203}$$

For the binary classification problem, we assume, as for the decision trees, that the two classes are labeled by $y_i = \pm 1$. Since the equation is homogeneous, there is no natural scale in the problem and we can define two parallel hyperplanes

$$H_1: \quad w \cdot x + b = 1\,, \qquad H_2: \quad w \cdot x + b = -1\,. \tag{204}$$

These two hyperplanes are a distance of $2M = 2/\|w\|_2$ apart, where M is known as the **margin**. If the data is linearly separable, these planes can be found such that there are no points between the two hyperplanes, or put differently, no points are within a distance of M of the original hyperplane H_0 in (203). This means we can use the original hyperplane H_0 as a classifier for any point x via

$$\hat{f}(x) = \text{sign}(w \cdot x + b)\,. \tag{205}$$

Note that $w \cdot x + b$ computes the signed distance of x from H_0. We illustrate the procedure in two dimensions in Fig. 49a.

In order to allow for as much wiggle room as possible for future data, we want to maximize the size of the gap $M = 1/\|w\|_2$ (or equivalently minimize $\|w\|_2$) subject to the constraint that the hyperplanes do separate the data,

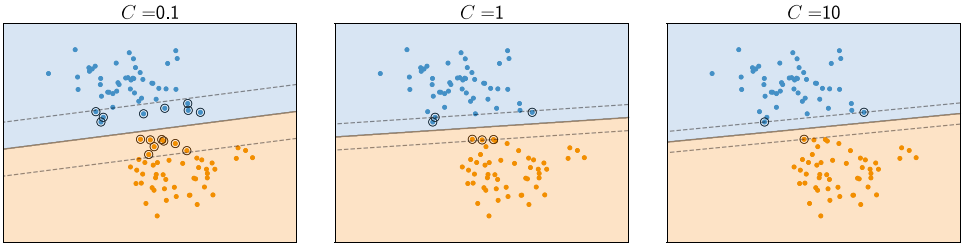
$$\min_{w,b} \|w\| \qquad \text{such that} \quad y_i(w \cdot x_i + b) \geq 1 \quad \forall i = 1, \ldots, N\,, \tag{206}$$

which is a classical optimization problem that is quadratic in w and has linear inequality constraints. Note that we can treat both classes simultaneously with the same inequality sign, since the sign of y_i (which is ± 1) and the sign of bracket (which compute the signed distance) cancel.

Before we go into details about how to solve this, let us discuss the case where the data is not linearly separable. In this case there is some arbitrariness in what we call the best separation. We define the **slack vector** $\xi = (\xi_1, \ldots, \xi_N)$ (see Fig. 49b) and replace the constraint in the optimization problem (206) by

$$\min_{w,b} \|w\| \quad \text{such that} \begin{cases} y_i(w \cdot x_i + b) \geq 1 - \xi_i & \forall i = 1, \ldots, N \\ \xi_i \geq 0\,, \quad \sum_{i=1}^N \xi_i \leq V \end{cases} \tag{207}$$

for some constant V. Note that the first inequality implies that any point with $\xi_i > 1$ will be misclassified. Since the constant V in the second inequality bounds the sum of ξ, and since each point x_i that cannot be classified correctly



(a) $C = 0.1$. (b) $C = 1$. (c) $C = 10$.

Fig. 50. Illustration of the margins (broken lines) and decision boundary (solid line) as a function of C. Support vectors are circled.

necessitates a $\xi_i > 1$, the parameter bounds the total number of misclassified points that are allowed. Of course, we do not know how many points we need to misclassify, so we include V in the objective function to minimize,

$$\min_{w,b,\xi} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\xi_i \quad \text{such that} \begin{cases} y_i(w\cdot x_i + b) \geq 1 - \xi_i \\ \xi_i \geq 0 \end{cases} \quad \forall i = 1,\ldots,N \tag{208}$$

where the **cost** C replaces V of (207). The separable case corresponds to $C \to \infty$ which enforces $\xi_i = 0$. Note that the optimization will fix the ξ_i for us, but we can control indirectly with C how the decision boundary will be placed. Smaller C will lead to a larger M, which means more points will fall in between the parallel hyperplanes. More points in between the margins potentially lead to a larger misclassification on the training set, but wider margins might lead to better generalization, i.e. better results on unseen data. Large C and small margins mean that the classifier will work well on the training set, but might overfit the data; another incarnation of the bias–variance problem explained in Section 3.1. We show results for different values of C in Fig. 50. The classifier function is in all cases given by (205), with w, b determined by (207).

By solving the constraint in (207) for ξ_i and dividing the objective function by C, we can reformulate the quadratic optimization problem (207) as

$$\min_{w,b} \frac{1}{2C}\|w\|^2 + \sum_{i=1}^{N}[1 - y_i(w\cdot x_i + b)]_+ \tag{209}$$

where $[\cdot]_+$ is the Hinge loss (82). Interpreting the second term as a loss, we can interpret the first term as a regularizer, in this case we are regularizing with the L_2-norm of the weights (i.e. the entries of the normal vector) w. By thinking about the optimization problem of finding SVM decision boundaries in this way, one can ask whether one could not use other loss functions, and indeed one can. If one uses the MSE loss instead, one gets a quadratic penalty for points that are within the margin, which means that these strongly influence the result. Another possible choice is the Huber loss (70).

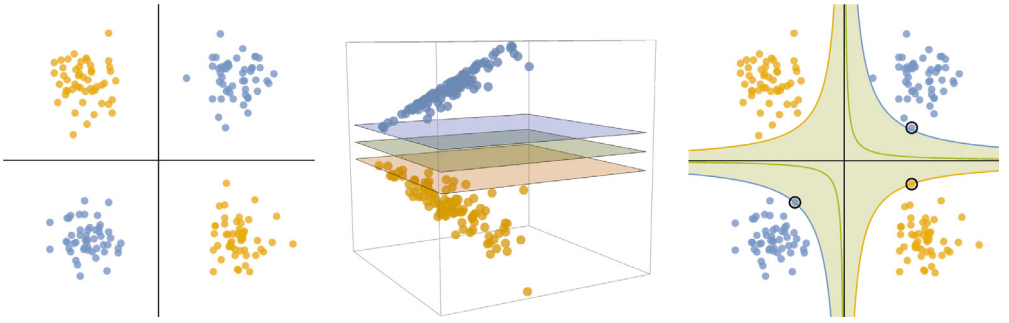
Kernel trick

The idea of the kernel trick is to replace feature space, which is the vector space $\mathbb{R}^F$, by the vector space of polynomials (or power series) whose variables are inner products of feature space vectors. In the case of polynomials (power series) one can choose a finite-dimensional (infinite-dimensional) basis. In other words, the kernel function $k(x_i, x_j)$ is a function of $x_i \cdot x_j$ only, $k(x_i, x_j) = k(x_i \cdot x_j)$. The point is that the optimization problem described above is fixed in terms of the inner products of all pairs x_i and x_j, as we shall see below. One furthermore defines the transform functions $\varphi(x_i)$ via $\varphi(x_i)\cdot\varphi(x_j) = k(x_i, x_j)$ such that the following diagram commutes

$$\begin{array}{ccc} \mathbb{R}^F \times \mathbb{R}^{\mathbb{F}} & \xrightarrow{\langle\cdot,\cdot\rangle} & \mathbb{R} \\ \Big\downarrow{\scriptstyle \varphi(\cdot)\times\varphi(\cdot)} & & \Big\downarrow \\ \mathbb{R}[[x]] \times \mathbb{R}[[x]] & \xrightarrow{\langle\cdot,\cdot\rangle} & \mathbb{R}[[x]]\,. \end{array} \tag{210}$$

Here $\langle\cdot,\cdot\rangle$ describes the inner product and $\mathbb{R}[[x]]$ is the ring of power series over $\mathbb{R}$ with variable x. Popular kernel functions are

- **Homogeneous polynomials of degree d**: $k(x_i, x_j) = (x_i \cdot x_j)^d$
- **Inhomogeneous polynomials of degree d**: $k(x_i, x_j) = (1 + x_i \cdot x_j)^d$
- **Gaussian radial basis**: $k(x_i, x_j) = e^{-\gamma\|x_i - x_j\|_2^2}$ with $\gamma = 1/(2\sigma^2) > 0$



(a) Original data. (b) 3D embedding. (c) Projected decision boundaries.

Fig. 51. Illustration of the kernel trick using the homogeneous degree 2 polynomial kernel (211).

- **Tanh**: $k(x_i, x_j) = \tanh(\kappa_1\, x_i \cdot x_j + \kappa_2)$ for some $\kappa_1 > 0$, $\kappa_2 < 0$

Let us give a simple example. Take $(x_i^\mu, x_j^\nu) \in \mathbb{R}^2 \times \mathbb{R}^2$ and take the kernel function to be a homogeneous degree two polynomial. Then

$$\begin{aligned} k(x_i, x_j) = (x_i \cdot x_j)^2 &= \left(\sum_{\mu=1}^{2} x_{i,\mu} x_{j,\mu}\right)^2 \\ &= (x_{i,1}x_{j,1})^2 + 2x_{i,1}x_{j,1}x_{i,2}x_{j,2} + (x_{i,2}x_{j,2})^2 \,. \end{aligned} \tag{211}$$

From this we find $\varphi(\cdot)$ which, written as a vector in $\mathbb{R}[x_1, x_2]$ with basis $\{1, x_1, x_2, x_1^2, x_1x_2, x_2^2\}$, reads

$$\varphi(x_\mu) = (0, 0, 0, 1, \sqrt{2}, 1)^T \,, \tag{212}$$

such that indeed

$$k(x_i, x_j) = \varphi(x_1) \cdot \varphi(x_2) \,. \tag{213}$$

We illustrate the steps for non-linearly separable data in Fig. 51.

Let us now explain why the optimization problem (208) depends only on the inner product of feature vectors. To see this, we incorporate the constraints as Lagrange multipliers (α_i, λ_i) and write

$$\mathcal{L} = \frac{1}{2}\|w\|_2^2 + C\sum_{i=1}^{N}\xi_i - \sum_{i=1}^{N}\alpha_i[y_i(w \cdot x_i + b_i) - (1 - \xi_i)] - \sum_{i=1}^{N}\lambda_i\xi_i \,. \tag{214}$$

Next we dualize the Lagrangian by "solving the algebraic equations of motion",

$$\begin{aligned} \nabla_w \mathcal{L} &= w - \sum_{i=1}^{N}\alpha_i y_i x_i \stackrel{!}{=} 0 \,, \\ \nabla_b \mathcal{L} &= -\sum_{i=1}^{N}\alpha_i y_i \stackrel{!}{=} 0 \,, \\ \nabla_{\xi_i} \mathcal{L} &= C - \alpha_i - \lambda_i \stackrel{!}{=} 0 \qquad \forall i \,, \end{aligned} \tag{215}$$

and substitute this back into the original Lagrangian to obtain

$$\mathcal{L}_D = \sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{N}\alpha_i\alpha_j y_i y_j\, x_i \cdot x_j \,, \tag{216}$$

which, as promised, only depends on the scalar products $x_i \cdot x_j$ for all pairs i, j. If we combine this with the kernel trick, we need to optimize the objective function (216) in the higher-dimensional (polynomial) vector space. All we need to do is to replace $x_i \cdot x_j$ by the corresponding transform $\varphi(x_i) \cdot \varphi(x_j)$. The classifier (205) then simply becomes

$$\hat{f}(x) = \text{sign}(w \cdot \varphi(x) + b) = \text{sign}\left(\sum_{i=1}^{N}\alpha_i y_i \varphi(x) \cdot \varphi(x_i) + b\right) \,, \tag{217}$$

where we used (215) in the second step.

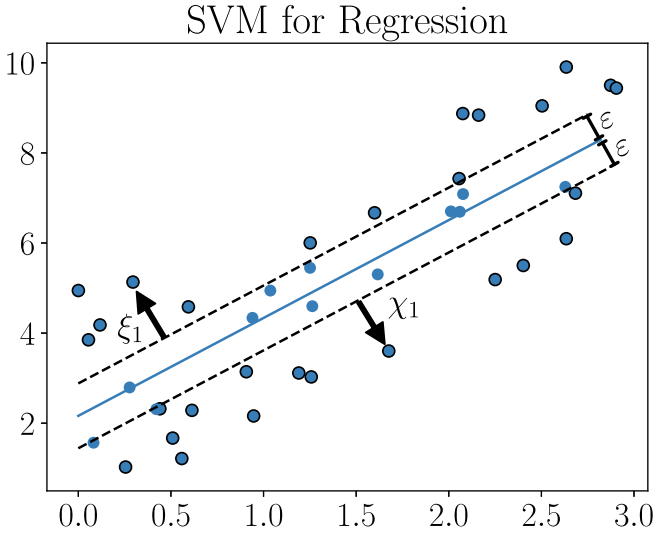


Fig. 52. SVM for regression. We show regression without kernel (linear regression) for $\varepsilon = 0.8$. Support vectors are circled in black.

Regression with SVMs

The idea for regression with SVMs is similar to finding a decision boundary for classification [163]. Without the kernel trick, one just wants to find the hyperplane that minimizes the summed distance to all points x_i. For given training pairs (x_i, y_i) with $x_i \in \mathbb{R}^F$, $y_i \in \mathbb{R}^T$, we need to solve the minimization problem

$$\min_{w,b,\xi,\zeta} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}(\xi_i + \zeta_i) \quad \text{such that } \forall i \begin{cases} y_i - (w \cdot x_i + b) \leq \varepsilon + \xi_i \\ y_i - (w \cdot x_i + b) \geq -\varepsilon - \zeta_i \\ \xi_i, \zeta_i \geq 0 \end{cases} \tag{218}$$

Note that this is an inverted version of (208), in the sense that points within the "margin" or tolerance interval $[-\varepsilon, \varepsilon]$ are treated as if they were lying directly on the hyperplane and are not penalized. We furthermore have two sets of slack parameters ξ, ζ that measure the distance from the bounding ε-planes to points outside the "margin", see Fig. 52. The solution to the minimization problem and the kernel trick (needed in non-linear regression) are analogous to the classification case.

10. String theory applications

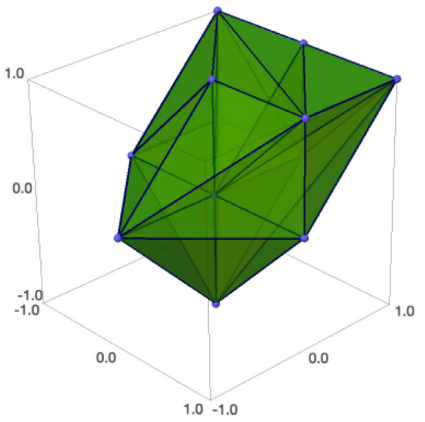
10.1. String theory data sets

In order to understand why string theory is amenable to tools from data science and machine learning, it is worthwhile to recap where the big data in string theory is actually coming from. At the level of M-Theory [164,165] (or F-Theory [10]) in 11 (or 12) dimensions the theory is essentially unique. In ten dimensions, there are five known consistent $\mathcal{N} = 1$ superstring theories (plus some without supersymmetry). All the non-uniqueness comes from compactification of the 10D (or 11D or 12D) theory to 4D on some compact six-dimensional (or seven-dimensional or eight-dimensional) manifold. The two main factors that lead to extraordinarily large numbers are the choices for the compactification geometries and the choices for fluxes.

The compactification geometry

While it is not necessary to have a space–time interpretation for the compactification manifold, a lot of work on string phenomenology of the recent years is based on models that do. Within these, Calabi–Yau (CY) manifolds are by far the most popular ones. The simplest construction is the so-called quintic, which is defined as the anti-canonical hypersurface in the (complex) projective ambient space $\mathbb{P}^4$. The name stems from the fact that the corresponding sections of the anti-canonical bundle are of degree 5. This is the only hypersurface in a projective ambient space leading to a CY, and there are three ways of generalizing the construction:

- Use complete intersections (i.e. several hypersurfaces) or just a collection of ideals to define the CY instead of just a single hypersurface,
- Use a more general ambient space,
- Use both more complicated ambient spaces and more complicated defining equations.



(a) 3D polytope.

$$\begin{array}{ll} v_1 = (-1,1,0), & v_2 = (0,-1,0) \\ v_3 = (1,0,0), & v_4 = (0,1,0) \\ v_5 = (-1,0,0), & v_6 = (0,0,-1) \\ v_7 = (0,0,1), & v_8 = (0,1,1) \\ v_9 = (-1,1,1), & v_{10} = (1,0,1) \\ v_{11} = (1,1,1) & \end{array}$$

(b) Vertices.

Fig. 53. Example for a triangulated 3D toric variety whose anti-canonical hypersurface defines a K3.

The first route led to a classification of all Complete Intersection Calabi–Yau (CICY) threefolds in a (product of) projective ambient spaces. There are 7890 such spaces, which is large enough to be non-trivial, but still a very small number by comparison. They were fully classified in [166]. Later this was extended to fourfolds in [167], of which there are almost a million. These spaces are uniquely specified by giving the number m and dimensions n_i, $i = 1, \ldots, m$ of the ambient space projective factors $\mathbb{P}^{n_i}$, and the multi-degrees q_i^a of the $a = 1, \ldots, K$ defining polynomials with respect to the $i = 1, \ldots, m$ projective ambient space scalings (in fact, the dimension of the ambient space $\mathbb{P}^{n_i}$ factors is fixed by the multi-degrees of the equations defining the CY),

$$n_i + 1 = \sum_{a=1}^{K} q_i^a \,. \tag{219}$$

This allows these compactification spaces to be encoded in the so-called integer-valued $m \times K$ configuration matrices.

Let us illustrate the description of CICYs via configuration matrices using simple examples. The simplest one is the quintic threefold X, which is given by a single polynomial of degree 5 in $\mathbb{P}^4$, hence its configuration matrix is

$$X \sim \left[\mathbb{P}^4 \mid 5\right] . \tag{220}$$

This can be shortened to just $X \sim [5]$, since this uniquely fixes the ambient space to be $\mathbb{P}^4$. A more complicated example is

$$X \sim \left[\begin{array}{c|cc} \mathbb{P}^1 & 1 & 1 \\ \mathbb{P}^2 & 3 & 0 \\ \mathbb{P}^2 & 0 & 3 \end{array}\right] , \quad \text{or simply} \quad X \sim \left[\begin{array}{cc} 1 & 1 \\ 3 & 0 \\ 0 & 3 \end{array}\right] . \tag{221}$$

The second route led to a classification of all complex four-dimensional reflexive polyhedra that encode toric ambient spaces (such that the CY is defined as a hypersurface in this ambient space). The classification is known as the Kreuzer–Skarke database [168] and there are almost half a billion of these CY constructions. Recently, all weight systems for CY fourfolds have been classified [169], of which there are 322 billion, with roughly half of them corresponding to reflexive polytopes. There are two equivalent ways of encoding the information that specifies the CY in the toric ambient space. As in the case of CICYs in projective ambient spaces, the multi-degree of the hypersurface equation is fixed in terms of the toric ambient space, so it is enough to specify the latter (as we shall discuss in detail below, we also need to specify the triangulation). For reflexive polytopes, one often just specifies the vertices v_i, $i = 1, \ldots, r$ of the polytope, which is a collection of integer-valued vectors v_i^a ($a = 1, 2, 3, 4$ for 4D reflexive polytopes that define CY threefolds) or five-vectors ($a = 1, 2, 3, 4, 5$ for 5D reflexive polytopes that define CY fourfolds). Equivalently, one can specify the GLSM charges (which can be chosen to be integers as well), which correspond to the kernel of the $4 \times r$ (or $5 \times r$) matrix of the defining vectors of the toric polytope. The former presentation is more common and allows for "drawing" the toric ambient space in 4 (or 5) dimensions.

We give a simple example of a three-dimensional polytope in Fig. 53. The polytope is reflexive and the anti-canonical hypersurface corresponds to an (elliptically fibered) CY twofold, i.e. to a K3. This simple polytope has already 36 (fine, regular, star) triangulations, and we show a random one of them. We also give the vertices that define the 3D reflexive polytope.

The third possibility would be to look at nef partitions or complete intersections in toric ambient spaces. To the best of my knowledge, there is no database for this type of construction, although nef partitions of reflexive polyhedra can be computed using SAGE or PALP.

With F-Theory becoming increasingly popular, a new way of constructing CY manifolds became popular. Since F-Theory necessitates an elliptic fibration (with or without a holomorphic section), one can just study the elliptic fibration itself and not worry about the base too much, or one can just look at the base over which one can fiber the F-Theory torus. The former data is encoded in toric tops, which were classified in [170], generalizing the original construction of [171]. There are 16 2D polyhedra that lead to tori, and on top of each of them one can add a hand full of tops (leading to toric threefolds with a gauge group as specified by the top), such that there are order 100 relevant tops (the number of tops is infinite, but order 100 can be completed to compact toric CY three- or fourfolds), whose geometry encodes the gauge group of the F-Theory compactification. The polytope in Fig. 53 corresponds to such a construction. It has one of the 16 2D polytopes at height 0 (this corresponds to the elliptic fiber), a single point at height -1 (this corresponds to a trivial top without any gauge algebra associated with it), and an SU(5) top at height one.

Note that the number of possible tops seems small, but in constructing the full CY, one can add a top over each base ray of the base of the elliptic fibration, leading to a vast number of possibilities; in fact, since most of the half a billion CY threefolds are elliptically fibered, we know that there are around half a billion top combinations for threefolds; the tops just give us a nice way of constructing a CY with the properties we want rather than finding the corresponding CY in the Kreuzer–Skarke database. The construction, which is based on the 16 2D polytopes giving rise to CY onefolds, can again be generalized by using nef partitions of 3D reflexive polytopes, for which there are 3134 [172]. The toric bases for F-Theory constructions have also been classified, at least for CY threefolds [173], for which there are 61,539.

For toric constructions based on ambient spaces of dimension 3 and higher, something fundamentally changes as compared to the 2D ambient space case: the triangulation of the reflexive polytope is not unique anymore. Note that this does not happen for the CICYs, whose Stanley–Reisner ideal is unique also in the three- and fourfold cases. The choice of triangulation fixes the curves and triple (or quadruple) intersections in the CY threefold (or fourfold) and thus impacts the resulting physics. Since the number of triangulations grows exponentially with the number of points of the toric polytope, this leads to a tremendous increase in the number of geometries. This impacts both the direct construction of threefolds or fourfolds from the Kreuzer–Skarke or Scholler–Skarke database, as well as the construction of 3D toric bases for fourfolds. Note that it does not directly impact the construction of the toric tops; even though these are also 3D toric varieties and hence have a wealth of possible triangulations, different choices of triangulations lead to the same physics since this corresponds to different choices of Weyl chambers of the Lie algebras encoded in the top. The triangulation dependence leads to an estimated $10^{3,000}$ distinct base geometries for F-Theory [174]. The authors of [15] present an explicit construction mechanism for $\mathcal{O}(10^{755})$ different toric F-Theory geometries, where the huge factor is again due to the different ways of triangulating the polytope. Just as a side remark, since the number of triangulations grows exponentially with the number of points, the one or two CYs with the largest number of points are completely dominating this count, such that all others of the half a billion 4D toric polyhedra can be completely neglected in obtaining these numbers.

Note that in order to specify the CY uniquely, we need to specify the triangulation, which means we need to give more information than just the vertices of the toric polyhedron. One possibility is to specify the polyhedron and all faces in all codimensions, or (equivalently) the Stanley–Reisner ideal. This raises the interesting question of which data is needed in order to uniquely specify the CY. According to Wall's theorem [175], we need to give the triple intersection numbers, the Hodge numbers, and the first Pontryagin class of the tangent bundle in order to specify the manifold up to homotopy equivalence.

It is a very interesting question whether or not the number of Calabi–Yau manifolds is finite. This question is difficult to answer, since we do not know all possible ways of constructing Calabi–Yau manifolds (on top of the methods explained above, there are non-linear sigma models, determinantal varieties, . . .). A conjecture known as Reid's fantasy [176] indeed postulates that there are (up to equivalences) only a finite number of CY threefolds (the conjecture is known to be true for CY one- and twofolds). It is also known that the number of elliptically fibered CY threefolds [177,178] and fourfolds [179] with a section is finite (up to birational equivalence). Furthermore, for the known constructions, it was observed that essentially all CY manifolds are elliptically fibered [180,181]. Thus, if essentially all CYs are elliptically fibered, and if the number of elliptically fibered CYs is finite, this seems to point towards the number of CYs being finite, but this reasoning is of course far from being a proof of finiteness.

The flux landscape

A second degree of freedom that opens up upon compactifying string theory is the possibility to add gauge flux along the internal, compactified dimensions. The first estimates of 10^{500} consistent flux vacua (per Calabi–Yau) of string theory were obtained in [12,14,182].

Of course, these have not been constructed explicitly. There are, however, some data sets of string models that have been constructed explicitly. We only list those that contain a somewhat large number of samples, i.e. at least $\mathcal{O}(10^4)$ MSSM-like models. The authors of [183] construct 179,520 MSSM-like Gepner models. In [184], 11,940 heterotic models were constructed based on heterotic toroidal orbifolds. In [185], 35,000 SU(5) GUT models based on $E_8 \times E_8$ heterotic line bundles were constructed. In [186], $\mathcal{O}(10^6)$ MSSM-like models were identified for the heterotic $E_8 \times E_8$ string, the SO(32) string, and the non-supersymmetric SO(16) $\times$ SO(16) string. A set of $\mathcal{O}(10^7)$ MSSM-like models based on the free fermionic construction has been obtained in [187].

Besides these explicit constructions, recent interest (also caused by the success of ML in string theory) has led to new estimates of the number of string vacua. The authors of [16] repeated the analysis of [14,182] for F-Theory and obtained an estimate of $10^{272,000}$ flux vacua. Recently, it was described in [188] how to obtain 10^{15} MSSM-like models by adding a three-dimensional toric base and flux to one of the 16 fiber polytopes mentioned before. For the heterotic models, there are estimates [189] of $\mathcal{O}(10^{23})$ MSSM models based on CICY geometries and $\mathcal{O}(10^{723})$ models based on the Kreuzer–Skarke database.

Again, one could ask whether the number of consistent flux vacua for a given Calabi–Yau is finite. While there is no proof, this has been demonstrated for concrete examples[14] in [18,191–193].

10.2. Computational complexity of string theory problems

As mentioned at the very beginning, many problems that need to be addressed in finding string vacua can be mapped onto problems that are either NP-hard or even undecidable. Early aspects of complexity and decidability in the string landscape have been discussed in [194,195] and [196], respectively. For a computer science introduction to complexity theory see [197–199].

In general, a problem can be thought of as an algorithm that, given some input of length N, computes an output B. If $B = \{\text{yes, no}\}$, the problem is called a decision problem. Often, problems that are not decision problems are optimization problems, i.e. they ask to find a maximum or minimum of some function of the input N. It is in many cases easier to make statements about decision problems. Note that optimization problems can often be reformulated (in a weaker version) as decision problems: If the optimization problem is to minimize a function f, the corresponding decision problem could be whether there exists some input x^* such that $f(x^*) \leq f_0$ for some fixed f_0. We will make the distinction clear using the example of the traveling salesman below.

Problems in the class P can be solved in polynomial time, i.e. the computation time grows at most like a polynomial in the parameter that specifies the input size of the problem, while NP problems can be verified in polynomial time, i.e. given a supposed solution, it can be verified in polynomial time that the proposed solution indeed solves the problem. Note that this does not mean that one needs to be able to verify in polynomial time that something is not a solution. For that reason, one defines co-NP problems, which are problems whose complements are in NP. Note furthermore that trivially $\text{P} \subseteq \text{NP}$, but the converse is believed to be false, i.e. $\text{P} \neq \text{NP}$, although this is not proven.

The reduction to other (known) problems is an important aspect of complexity theory. It might be hard to prove that a certain problem is in NP, but it might be easy to find a map from the current problem onto a different problem that is known to be in NP. If the map itself is a polynomial in the input size, and the problem can be mapped onto (or reduced to) an NP problem, then the original problem is NP. This leads to the notion of NP-hard and NP-complete problems. These are the hardest problems to solve in the following sense: A problem G is called NP-hard if there exists a polynomial time reduction map to G from **every** problem in NP. Since the map need not be surjective, NP-hard problems are not necessarily in NP, i.e. they can contain instances that are beyond NP. NP-hard problems that are in NP themselves are called NP complete. Alternatively, a problem is NP-hard (NP-complete) if there exists a polynomial map from another NP-hard (NP-complete) problem to G. It seems very hard to prove that every problem in NP can be reduced to G, which makes it surprising that there are any problems that are known to be NP-hard or NP-complete. Note that if someone was to find a polynomial time solution algorithm for any NP-complete problem G, then, by definition, any problem in NP could be mapped with a polynomial time algorithm onto G and then solved in polynomial time, which would imply P=NP. As mentioned before, it is commonly believed that no polynomial time algorithm exists for any NP-complete problem.

There are problems that are even harder than NP. In fact, one defines an entire polynomial hierarchy of complexity classes, but we will not go into more details here. Instead, we would lastly like to mention the complexity class of bounded-error quantum polynomial time (BQP), which contains decision problems that can be solved on a quantum computer in polynomial time with an error probability of less than 33 percent. The most prominent example of a problem that is believed to be in NP but can be solved in polynomial time on a quantum computer is factoring an integer via Shor's algorithm [200]. It is believed that $\text{BQP} \neq \text{NP}$, thus factoring integers is most likely not NP complete. On the other hand, that means that current NP-complete problems cannot be solved in polynomial time, even on a quantum computer. We present an overview of the nesting of complexity classes under the assumption that they are all mutually distinct in Fig. 54.

There is a detailed list of problems that are known to be NP-hard or NP-complete in [197]. Among the NP-hard or NP-complete problems that are relevant for string theory (and for many other applications) are:

[14] There is an infinite family of Type IIA flux vacua [190], but their number at any given energy scale is finite.

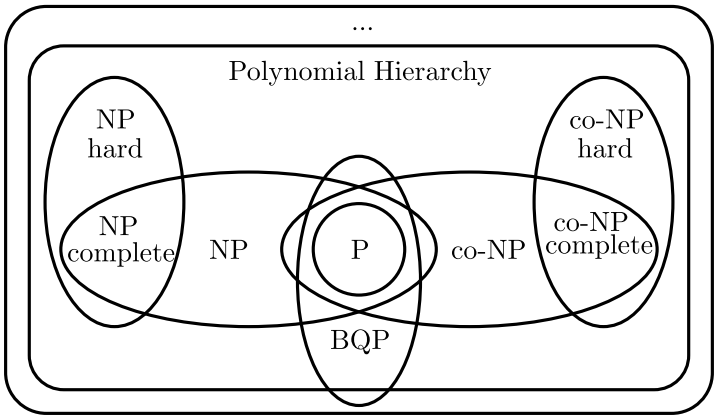


Fig. 54. Overview of the complexity hierarchy under the assumption that the complexity classes P, NP, BQP, ... are mutually distinct.

Traveling salesman	Given a list of cities and distances between them, find the shortest closed loop that visits all cities.
Knapsack	Given a set of items, each of which has a certain weight and value, find the most valuable combination of items that does not exceed a certain total weight.
Subset sum	Given a set of integers, is there a subset such that their sum is zero (or any other value).
Rural postman	Given a graph G, its set of edges E, non-negative integer edge lengths ℓ_i, and a subset of edges $E' \subset E$, is there a closed loop in G that includes each edge in E' with total length less than some constant B.

Let us exemplify the difference between an optimization problem and a decision problem using the traveling salesman problem (TSP). The way TSP is stated above, it is an optimization problem. The corresponding decision problem would be to ask whether there exists a route of total length less than some constant ℓ_*. Clearly, if we solve the optimization problem, we can solve the decision problem: we just need to find the shortest route and compare it to ℓ_*. However, the converse is not true. If we find a route with length less than ℓ_*, we do not know whether it is the shortest route. Note that this means that the decision version is proven to be NP-complete, but the optimization version is only known to be NP-hard. The difference comes precisely from the fact stated before: If we are handed the supposedly shortest path in the optimization version, we have no (known) way of checking that it actually is the shortest path in polynomial time, i.e. we cannot verify the yes instance. Conversely, we can also not verify the no-instance in polynomial time, since this would require to know the shortest path as well.

The traveling salesman problem and the rural postman problem (RPP) are problems in graph theory and have relevance for example in quiver gauge theories. Indeed, solving RPP corresponds to finding superpotential terms of mass dimension at most B that involve a certain set of fields [19]. Knapsack and subset sum are closely related problems that require finding optimal subsets. For example subset sum can be mapped onto the problem of finding (integer) fluxes such that the resulting superpotential leads to a small cosmological constant [18].

There are even decision problems that cannot be decided by any algorithm. We have introduced the most famous one, the halting problem, in Section 1. It asks to decide, given any algorithm x as input, whether x terminates. While the halting problem might not seem relevant to string theory, it was proven that solving Diophantine equations is in general undecidable [201]. This point has also been discussed in a string theory context in [196].

10.3. Computing line bundle cohomologies

While the index

$$\chi(X,V) = \sum_{i=0}^{3} (-1)^i h^i(X,V) \tag{222}$$

of a vector bundle V over a CY threefold X can be computed rather fast, knowing the individual dimensions of the cohomology groups is computationally much more expensive. The index can be computed by the Hirzebruch–Riemann–Roch index theorem. It is a polynomial that is at most of degree n for a CY n-fold X,

$$\chi(X,V) = \int_X \mathrm{ch}(V)\mathrm{Td}(X)\,, \tag{223}$$

Table 1
NN architectures of [3] for binary classification. FC are fully connected layers.

(a) NN stability classifier		(b) NN vector-like pair classifier	
Layer	Parameters	Layer	Parameters
Input	2 nodes	Input	4 nodes
FC	4 nodes	FC	300 nodes
ReLU	4 nodes	Sigmoid	300 nodes
FC	4 nodes	Fully connected	300 nodes
ReLU	4 nodes	Tanh	100 nodes
Output	1 node	Summation	–
Sigmoid	1 node	Output	1 node

where $\mathrm{ch}(V)$ is the Chern character of V, $\mathrm{Td}(X)$ is the Todd class of the tangent bundle of X, and the integral picks out the (n,n)-form part of the expression. For a line bundle ($V=L$) on a CY threefold X,

$$\chi(X,L)=\int_X \mathrm{ch}(L)\mathrm{Td}(L)=\int_X \frac{1}{6}c_1(L)^3+\frac{1}{12}c_1(L)c_2(X)\,, \tag{224}$$

i.e. the index is given by a cubic polynomial (where we used $c_1(X)=0$).

The individual cohomology group dimensions can be computed by Gröbner basis computations, which is notoriously costly (it scales with a double exponential), or by sequence chasing in spectral sequences or Koszul sequences (often, this requires explicit knowledge of the maps occurring in the sequences). Unfortunately, the individual cohomology dimensions are of great phenomenological importance. In order to check bundle stability (i.e. the SUSY D-term conditions), we need to know about subsheaves injecting into the vector bundle, which can be determined by studying sections (i.e. $h^0(X,V)$ and/or, by Serre duality, $h^3(X,V^*)$) of the bundle in question. The individual dimensions $h^1(X,V)$ and $h^2(X,V)$ count the number of generations and anti-generations and are therefore also very important to know (since e.g. the Higgs is vector-like, presence of a Higgs pair can only be decided from the individual cohomology dimensions, not from the index). Note that, if only one $h^i(X,V)$ is non-zero, the index is (up to a sign) equal to this cohomology dimension and can thus be computed fast. This makes it also interesting to know which bundles have more than one $h^i\neq 0$.

These types of questions were the subject of [3]. In order to address the question of bundle stability, a NN was used that acts as a binary classifier for stability of a line bundle (cf. Section 2.1) over the CICY threefold

$$X\sim\left[\begin{array}{c|c}\mathbb{P}^2 & 3\\ \mathbb{P}^2 & 3\end{array}\right]. \tag{225}$$

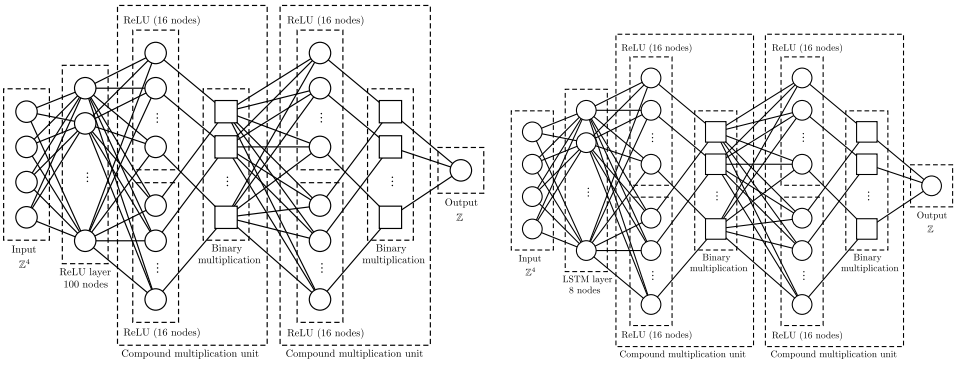
As an input, the first Chern classes of the line bundles over the $h^{1,1}(X)=2$ divisors of the CICY were used. Since the input features did not include any information on the CICY (such as its configuration matrix (225)), and the results depend crucially on the normal bundle of the variety, the NN cannot generalize to other CICYs. If one was after such a generalization, the configuration matrix would have to be included together with the data specifying the bundle. A simple feed-forward NN with two hidden layers with 4 nodes and ReLU activation, followed by either a logistic sigmoid or a softmax in the output layer was used for the binary classification problem. Note that for binary classification, either softmax or logistic sigmoid (or even tanh) work. The NN architecture is summarized in Table 1a.

The second question that was studied was whether a line bundle L on a CICY X with configuration matrix

$$X\sim\left[\begin{array}{c|c}\mathbb{P}^1 & 2\\ \mathbb{P}^1 & 2\\ \mathbb{P}^1 & 2\\ \mathbb{P}^1 & 2\end{array}\right] \tag{226}$$

leads to vector-like pairs, i.e. whether or not $h^1(X,L)\neq 0$ and $h^1(X,L)\neq 0$. This is a binary classification problem, for which we used the architecture in Table 1b. We trained the NN with 3000 line bundle models for 90 s using random initialization, L_2 weight regularization, an ADAM optimizer with learning rate 0.01, and a batch size of 128. The test set had 7000 elements. Note that there are much fewer cohomologies in the training set than in the test set. The reason was that we were interested in seeing whether the NN can learn the correct classification relatively little input. This is important since computing line bundle cohomologies is computationally very expensive, so we want the NN to be able to learn from a relatively small data set. It was found that the NN correctly identified those models that have vector-like pairs in more than 85 percent of the cases.

As a third application, line bundle cohomologies of a CICY in a product of projective ambient spaces are computed. Often, these CICYs come with permutation symmetries (such as permuting $\mathbb{P}^n$ ambient space factors of the same dimension) which can be inherited by the bundle. This led to the question of what the best NN architecture would be to take these symmetries into account. One possibility is to use the last state of an LSTM, which leads to an "averaging" over the input. Other possibilities are just using a wide and deep enough network. Since training time for the problem is short, I used genetic algorithms to find or "evolve" viable neural network architectures and hyperparameters. Since the result



(a) NN architecture evolved to compute h^0 and h^3.

(b) NN architecture evolved to compute h^1 and h^2.

Fig. 55. NN architectures to compute h^0 or h^3 (left), and h^1 or h^2 (right).

is a (at most) cubic polynomial, I also included parameter-free non-unary layers such as summation and multiplication layers that form the sum or product of two inputs. These layers were also used later in the regression equation learner in [202] (see Section 10.5.2). The addition and multiplication layers are compound layers which are set up as follows: first, the input layer is duplicated. Then, each of the copies is fed into a ReLU layer (note that these two ReLU layers are independently defined for each of the copies, i.e. at this level the NN is not full connected). Finally, the output of the first node of the first ReLU layer is added to or multiplied with the first output of the second ReLU layer. Likewise, the second output of the first ReLU layer is added to or multiplied with the second output of the second ReLU layer, and so on for all input data of the compound layer. For a diagram of the compound layer see Fig. 55.

In addition, the gene pool included standard fully connected layers with different activation functions (logistic sigmoid, tanh, ReLU, softmax) and LSTMs. The number of nodes per layer is a free hyperparameter (we allow for up to 400 nodes per layer) that can be changed by the GA. The GA was set up with 10 individuals per generation, a maximum of 10 generations and 1 min training time per individual in each generation. The fitness function is the percentage of correctly predicted cohomology dimensions of the test set, where we computed 2401 different cohomologies and performed a train:test split of 1:3. Again, there were much fewer elements in the training set than in the test set since computing cohomologies is very expensive.

For mating, we use roulette wheel selection with uniform crossover. We produce 20 children and carry the 10 fittest over to the next generation. We furthermore give the 2 fittest individuals a 50% chance to clone themselves instead of reproduction via crossover (i.e. there is a 50% chance for elitism). The mutation rate is 10%. When mutating, three things can happen: A new NN layer is added to the genome from the gene pool, a NN layer is removed from the genome, or a NN layer is replaced with another layer from the gene pool.

Due to a symmetry in the problem (Serre duality, paired with the fact that we train with both positive and negative line bundle charges), the cohomologies of a line bundle L over a CY X satisfy $h^0(X,L) = h^3(X,L^*)$ and $h^1(X,L) = h^2(X,L^*)$. Indeed, we find that the NNs evolved to compute the zeroth and the third cohomology group dimension have a similar architecture, and the NNs evolved to compute the first and the second cohomology dimensions have a similar architecture, cf. Figs. 55a and 55b, respectively. The accuracies (i.e. the fitness) obtained for h^0 and h^3 are 94% and for h^1 and h^2 are 71%.

It is interesting that the NNs, given in Fig. 55a, evolve two multiplication layers, which is precisely enough to produce a cubic polynomial. Moreover, interestingly the NNs for h^1 and h^2 evolve an LSTM, maybe to average over the input. We have trained the NN by only inputting the defining data of the line bundle L, but not the data defining the CY X. So the trained NN will only work for the specific CY it was trained on. It would be interesting to see whether a NN trained with the configuration matrix defining X on top of the line bundle data L can predict cohomology dimensions for a variety of CICYs and line bundles. While this was beyond the scope of the paper, it was verified that the same NN architecture leads to comparable accuracies when trained on another CICY with another set of line bundles.

The authors of [203] also use machine learning to compute line bundle cohomologies, in their case for toric varieties. In contrast to [3], they split the computation into different parts.

Interestingly, it has been observed in [203,204] that the expression for each $h^\bullet(X,L)$ is a polynomial. This can be motivated by noting that the index, which is the alternating sum of the cohomology group dimensions, is a polynomial, cf. Eqs. (222) and (224). However, the polynomial is different for different values of the bundle and CY Chern classes. This means that, for a fixed CY, one can divide the space $\mathbb{Z}^{h^{1,1}}$ of possible line bundles

$$L = c_1(D_1, \ldots, D_{h^{1,1}}) = \mathcal{O}_X(k_1, \ldots, k_{h^{1,1}}) \tag{227}$$

Table 2
NN architectures of [1] for binary classification. FC are fully connected layers.

(a) First NN architecture of [1]		(b) Second NN architecture of [1]	
Layer	Nodes	Layer	Nodes
Input	5 nodes	Input	180 nodes
FC	100 nodes	FC	1000 nodes
Sigmoid	100 nodes	Sigmoid	1000 nodes
Tanh	100 nodes	FC	100 nodes
FC	100 nodes	Tanh	100 nodes
Summation	100 nodes	Summation	100 nodes
Output	1 node	Output	1 node

into different regions. Within each region, one can then find a polynomial expression for $h^\bullet(X, L)$. In order to identify the different regions, the authors of [203] compute many cohomology dimensions and form discrete derivative (finite differences) between them. They then use clustering to identify the various regions, where they give the integers k_i characterizing the line bundle, as well as the discrete derivatives, as features. Once the different regions have been identified, they fit a (cubic) polynomial to each region. The fact that the computation of line bundle cohomologies can be performed in this two-step procedure (clustering + fitting), can at least be taken as a hint for why the NNs of [3] were able to compute line bundle cohomologies rather effectively.

To summarize, we have seen that evolving NNs and equation learners with GAs can be useful for computing line bundle cohomologies. NNs were trained to predict line bundle stability, the presence of vector-like pairs, and the individual line bundle cohomology group dimensions with good results. Interestingly, the GA evolved an equation learner NN which had the correct layers to compute the cohomologies. The process can also be split up into an unsupervised part which identifies regions in which the cohomology group dimensions are given by the same polynomial expression. This expression is then found by fitting polynomials to the data.

10.4. Deep learning in Kreuzer–Skarke and CICY data set

The set of CYs described by hypersurfaces in toric varieties or complete intersection Calabi–Yau three- and four-folds in products of projective ambient spaces are the largest data sets of Calabi–Yau manifolds. Note that the classification of these manifolds is complete in all cases, i.e. there do not exist any more CYs of the respective type. All papers we are reviewing here use NNs to classify or compute quantities in this data set that are already known and that can be computed using techniques in algebraic geometry, i.e. without any NN or ML. Hence, the papers reviewed in this section should be considered as proof-of-concept studies, which investigates applicability of ML to (topological) string theory data.

Binary classification of $h^{2,1} \overset{?}{>} 50$ for CICY threefolds

The examples of [1] are mainly binary classification problems for topological quantities of CYs. In the first example, a set of 7555 CYs X that can be written as hypersurfaces in $W\mathbb{P}^4$ is classified according to whether or not the dimension of their complex structure moduli space (counted by $h^{2,1}(X)$) is larger than 50: $h^{2,1}(X) > 50$ is labeled by $y = 1$ and $h^{2,1}(X) \leq 50$ is labeled by $y = 0$. The input features are just the weights of $W\mathbb{P}^4$, i.e. $x \in \mathbb{Z}^5$. The NN used for this classification task is somewhat unusual, cf. Table 2a: It has a hidden layer with 100 nodes and two successive activation functions (a logistic sigmoid and a tanh). The reason why one usually does not use two activation functions is because the logistic sigmoid maps the input to the interval [0, 1], and the tanh on that interval is almost linear. The author remarks in subsequent publications that the results obtained by using two activations were slightly better than with using just one. After that, the NN has two linear layers, one layer with 100 nodes and identity activation, and a summation layer (which just sums up all components, i.e. its weights are a 1×100 matrix whose values are all fixed to 1 and whose biases are fixed to zero). The operations of these layers could be collapsed to a single layer with 1 node (and a 1×100 weight matrix). Also, the NN does not end with a softmax or logistic sigmoid function as is done usually in classification to give a probability interpretation to the NN output.

The author does not specify other hyperparameters such as which weight initialization, which optimizer, or which training method was used. Since no train:test split is performed it is hard to evaluate the network's performance. The paper quotes an accuracy of 96.2% if trained and validated on the entire data set and an accuracy of 80% if trained on 3000 data points and evaluated on all 7555 (i.e. roughly half of the input–output pairs have been seen by the NN during training). The same network achieves 82% accuracy when used as a binary classifier that classifies the 7555 CYs into a class whose Euler number is divisible by three and a class of CYs whose Euler number is not.

Binary classification of divisibility by three of the index of spectral cover bundles

Next, the author of [1] uses the NN architecture of Table 2b for the data set of spectral cover vector bundles on CY threefolds that are elliptically fibered over a Hirzebruch base. The NN architecture used this time is more standard. It consists of 2 hidden layers, one with 1000 nodes and logistic sigmoid activation and one with 100 nodes and tanh

activation. The output layer is again chosen to be a summation layer, i.e. to not contain trainable parameters and just sum up the 100 outputs of the tanh-activated nodes of the previous layer.

Let us explain the feature space in a bit more detail: A Hirzebruch surface $\mathbb{F}_r$ is a $\mathbb{P}^1$ fibration over $\mathbb{P}^1$. It is common to denote the base $\mathbb{P}^1$ by S and the fiber $\mathbb{P}^1$ by E, such that

$$E.E = 0\,, \qquad E.S = 1\,, \qquad S.S = -r\,, \tag{228}$$

where a lower point denotes the intersection product in $\mathbb{F}_r$. The parameter r encodes the non-triviality of the fibration. The space can be written torically as a fan whose rays are given by

$$v_1 = \begin{pmatrix}1\\0\end{pmatrix}\,, \quad v_2 = \begin{pmatrix}0\\1\end{pmatrix}\,, \quad v_3 = \begin{pmatrix}0\\-1\end{pmatrix}\,, \quad v_4 = \begin{pmatrix}-1\\-r\end{pmatrix}\,. \tag{229}$$

For elliptically fibered CY threefolds over a base $B = \mathbb{F}_r$, we have $0 \leq r \leq 12$, since for higher r, the elliptic fibration becomes too singular to resolve crepantly by blowups in the fiber.

A spectral cover bundle is described in terms of a spectral sheet and a spectral sheaf. The cohomology class of the spectral sheet C_V can be written as

$$[C_V] = n\sigma + \eta\,, \tag{230}$$

where $\sigma \sim B$ is the section of the elliptic fibration and η is a divisor dual to an effective curve class $\check{\eta}$ in B, with

$$\check{\eta} \sim aS + bE\,, \qquad a, b \in \mathbb{Z}_{\geq 0}\,. \tag{231}$$

The choice of n in (230) encodes the structure group of the spectral cover bundle, which is SU(n). The spectral sheaf $\mathcal{N}_V$ is in this case a line bundle on C_V whose first Chern class is

$$c_1(\mathcal{N}_V) = n\left(\frac{1}{2}+\lambda\right)\sigma + \left(\frac{1}{2}-\lambda\right)\pi^*\eta + \left(\frac{1}{2}+n\lambda\right)\pi^*c_1(B) \tag{232}$$

where

$$\lambda = \begin{cases} m+\frac{1}{2} & \text{if } n \text{ is odd}\,,\\ m & \text{if } n \text{ is even}\,. \end{cases} \tag{233}$$

Thus, the feature space is given by $(r, n, a, b, \lambda) \in \mathbb{Z}^4 \times \frac{1}{2}\mathbb{Z}$. The NN in [1] is designed to output the structure group of the bundle (which is also an input feature), and a class indicator specifying whether or not $\chi(V)$ is divisible by 3. Having a vector bundle whose index is a multiple of 3 is necessary (but not sufficient) to obtain three families, since the GUT gauge group (which is given by the commutant of SU(n) with E_8) is broken by a freely acting Wilson line supported on a non-trivial $\pi_1(X)$, which reduces the index of the bundle by the order of the group $\pi_1(X)$. Using the same NN architecture as before (cf. Table 2b), the accuracies are 100% and 69% for training and testing with the full data set and for training with half and testing with the full data set, respectively.

Classification of $h^{1,1}$ for self-mirror Kreuzer–Skarke models

Next, the author of [1] studies three-dimensional toric varieties Y given by 3D polyhedra. There are 4319 such polyhedra; see Fig. 53 for one example. These can be taken to either define CY two-folds (i.e. K3 surfaces) X_2 if one considers the variety given by the anticanonical hypersurface inside the 3D toric ambient space Z, or non-compact CY four-folds X_4, if one adds the anti-canonical bundle over the toric space. The non-compact four-fold X_4 can in turn be thought of as a real cone over a compact, real seven-dimensional Sasaki–Einstein manifold Y.

In any case, the model is specified completely by the vertices of the polyhedron, which are in turn given by a set of three-dimensional vectors in $\mathbb{Z}^3$. Since the various polyhedra contain a different number of vertices (between 5 and 39), the input is padded with zeros. The vectors are then all concatenated to form an input feature vector $x \in \mathbb{Z}^{3\cdot 39}$. Again, the network specified in Table 2b is used in order to binary classify the input: For the K3, the classes are whether or not the rank of the Picard group of X_2 is larger than 10. For the non-compact fourfolds, the two classes are whether or not the volume of the associated Sasaki–Einstein manifold Y is larger than 200. The NN reaches accuracies of 59 and 57 percent, respectively. While this is not said explicitly, it seems that the networks have been trained and tested with the full data set of 4319 polyhedra. The author also considers a subset of 10,000 self-mirror CY threefolds from the Kreuzer–Skarke database with $h^{1,1} = h^{2,1} \in \{14, 15, 16, 17, 18\}$. The input are now 4D vectors, again padded with zeros, and the problem is phrased as a classification problem with 5 classes according to the values of $h^{1,1}$. The NN reaches accuracies of 61% when trained and tested on the full 10,000 CYs and 78% when trained on 4000 and tested on all 10,000. It is surprising that the accuracy of the NN that was trained with only a subset of data is higher than the one of the NN which has seen all input data.

Table 3
NN architectures of [205] for regression. FC is a fully connected layer and LSTM is a Long Short-Term Memory layer.

(a) Regressor NN of [205]		(b) Classifier NN of [205]	
Layer	Parameters	Layer	Parameters
Input	180 nodes	Input	180 nodes
FC	1000 nodes	LSTM	–
Summation	1	Tanh	–
Output	1	ReLU	–
Sigmoid	1	Dropout	20%
		Batch norm	–
		LSTM	–
		Tanh	–
		ReLU	–
		Dropout	20%
		FC	120 nodes
		Tanh	–
		ReLU	–
		Dropout	20%
		FC	120 nodes
		Tanh	–
		ReLU	–
		Dropout	20%
		Output	20 nodes
		Softmax	20 nodes

Regression for the number of points in a quiver gauge theory

Another example presented in [1] is predicting the number of points in a toric diagram associated with a quiver gauge theory. The data set consists of 375 quiver theories. Let us explain the feature space of the quivers: A quiver consists of a set of nodes and directed vertices (arrows) between the nodes. Each node corresponds to a U(n) gauge group whose rank n is specified in the quiver by associating the number n with the node. An arrow connecting node i with node j corresponds to a field transforming in the bi-fundamental representation $((\boldsymbol{n_i})_1, (\overline{\boldsymbol{n_j}})_{-1})$ of U(n_i)×U(n_j). In the simplest case studied here, all $n_i = 1$. We can then associate a GLSM to the quiver. The GLSM superpotential is given by the gauge-invariant terms that can be formed out of the fields corresponding to the arrows, and the GLSM charges are given by the U(1) gauge theories corresponding to the nodes of the quiver. Alternatively, one can represent a GLSM as a toric variety, whose vertices are given by the kernel of the GLSM charge matrix.

The NN specified in Table 2b is used to predict the number of points of the toric diagram constructed from the GLSM associated with the quiver as explained above. The network is trained with the GLSM charge matrix, which is a 33×36 matrix for the data set at hand. It reaches an accuracy of 99.5% when trained and validated on the full set of 375 quiver gauge theories.

Regression of $h^{1,1}$ for CICYs I

As a further example in [1], the NN in Table 2b is used to predict the dimension $h^{1,1}(X)$ of the Kähler moduli space of a CICY threefold. There are 7890 CICYs, and they are determined in terms of their $m \times K$ configuration matrix, as explained in Section 10.1. The value of $h^{1,1}(X)$ in this data set ranges between 1 and 19. In the entire set, the number m of projective ambient space factors does not exceed 12, and the number K of equations does not exceed 15. The input features are taken to be the entries of the configuration matrix, padded by zeros and flattened into a single input vector $x \in \mathbb{Z}^{12\times 15}$. Note that padding with zeros could be potentially dangerous in this case, since 0 is a value that each feature can actually take; for that reason, it might be preferable to pad with values that are not in the domain of the features, e.g. with -1 in this example. Again, no train:test split is performed, and the NN learns to predict $h^{1,1}(X)$ with 99.91% accuracy on the train set, and reaches an accuracy of 77% when trained with 5000 CICYs and evaluated on all 7890 CICYs (which contain the 5000 training CICYs). A similar study is repeated for the almost 1 million CICY fourfolds with comparable results.

Regression of $h^{1,1}$ for CICYs II

The authors of [205] revisit the prediction of $h^{1,1}$ for CICY threefolds via regression. They train on CICY configuration matrices with small Hodge numbers and try to predict $h^{1,1}$ for large Hodge numbers. They use NNs and SVMs as regressors and a NN for classification. Since there are only 19 different possibilities for $h^{1,1}(X)$, the regression problem can be phrased as a 1-vs-all classification problem with 19 classes. The SVM regressor is run with a Gaussian kernel with variance $\sigma = 2.74$, $C = 10$ and tolerance $\epsilon = 0.01$. The NN architecture for the regressor and classifier is summarized in Table 3.

The NN design used in [205] is again somewhat unusual. The output layer of the regressor NN given in Table 3a applies a logistic sigmoid, which maps the output to the interval [0, 1]. Since the Hodge numbers $h^{1,1}$ are integers in the range $1 \leq h^{1,1} \leq 19$, the authors divide the ground truth values by 19 and use the value of $y = h^{1,1}(X)/19$ that comes closest to the output $\hat{y}$ of the NN. The second NN involves two LSTM layers in sequence. While the authors do not explain how

Table 4
NN architectures of [206] for regression and classification. FC is a fully connected layer and Conv2D is a 2D convolutional layer. k gives the kernel size and f the number of filters of the Conv2D layer.

(a) Regressor NN of [206]		(b) Classifier NN of [206]	
Layer	Parameters	Layer	Parameters
Input	180 nodes	Input	180 nodes
FC	876 nodes	Conv2D	(f : 54, k : 3×3)
ReLU	876 nodes	ReLU	–
Dropout	20.72%	Dropout	50%
FC	461 nodes	Conv2D	(f : 56, k : 3×3)
ReLU	461 nodes	ReLU	–
Dropout	20.72%	Dropout	50%
FC	437 nodes	Conv2D	(f : 55, k : 3×3)
ReLU	437 nodes	ReLU	–
Dropout	20.72%	Dropout	50%
FC	929 nodes	Conv2D	(f : 43, k : 3×3)
ReLU	929 nodes	ReLU	–
Dropout	20.72%	Dropout	50%
FC	404 nodes	FC	169 nodes
ReLU	404 nodes	ReLU	–
Dropout	20.72%	Dropout	50%
Output	1 node	Output	20 nodes
ReLU	1 node	Sigmoid	20 nodes

they use the LSTMs, they most likely apply them to "average" over the configuration matrix. The authors apply a tanh and a ReLU after the LSTM (which already applies a tanh), so the input is activated three times. Since the first tanh of the LSTM maps the output into the interval $[-1, 1]$, the second tanh is essentially linear and the third activation (ReLU) cuts off the negative part, i.e. it maps the output linearly into $[0, 1]$. Again, the authors state that this architecture leads to better results than using a single activation function. The final layer is a softmax layer, as is usual for 1-vs-all classification problems.

The authors find that both the NN and the SVM perform poorly in the extrapolation task when trained with CICY configuration matrices of small $h^{1,1}$ manifolds, with accuracies of 5% and 30%, respectively. These accuracies are obtained when training on CICYs up to $h^{1,1} = 10$ and predicting on the remaining ones with $11 \leq h^{1,1} \leq 19$. When randomly including 10% of the manifolds with $h^{1,1} > 10$, the results improve somewhat. They also try feature engineering and include the squares or cubes of the configuration matrix entries. The idea is that the Euler number χ, which is linear in $h^{1,1}$, is cubic in the configuration matrix entries. However, this only slightly improves the performance. Surprisingly, the accuracy decreases drastically for the models that were trained with some configuration matrices of larger $h^{1,1}$. It drops from around 70% (achieved by the NN classifier, when trained predominantly on CICYs with $h^{1,1} \leq 2$) to 20% when predominantly trained on CICYs with $h^{1,1} \leq 10$. This is surprising, since the training set is better balanced and much larger (order 1000 for $h^{1,1} \leq 2$ and order 7000 for $h^{1,1} \leq 10$).

Regression of $h^{1,1}$ for CICYs III

In [206], the authors investigate further properties of CICYs using NNs and SVMs. Just like [3], they also use GAs to find the best hyperparameters. In their first study, they use SVMs and NNs to predict $h^{1,1}(X)$ for CICY threefolds X. This is done again with a classifier NN with 20 classes (the authors include a class for $h^{1,1} = 0$). The SVM regressor uses a Gaussian kernel with variance $\sigma = 2.74$, cost $C = 10$ and tolerance $\epsilon = 0.01$.

We summarize the NN architecture for the regression and classification NN in Table 4. For the genetic algorithm, they do not specify the selection mechanism, other than that they use 20% elitist selection. Note that the classification NN uses 20 logistic sigmoid instead of a softmax activation function in the last layer. The authors do not specify the stride of the CNN and do not detail how the different filters are combined in between the convolutional layers. They also do not specify the initialization method or the loss function used for classification. They find accuracies of 70%, 78% and 88% for the SVM, the NN regressor and the NN classifier, respectively, for train:test splits of roughly 90:10.

Binary classification by favorability of CICYs

Next, the authors of [206] study again a binary classification task, this time to predict whether a CICY is favorable. This is the case if $h^{1,1}$ equals the number of ambient space $\mathbb{P}^n$ factors, i.e. the number of rows in the configuration matrix. They use an SVM classifier with Gaussian kernel, $\sigma = 3$ and $C = 0$. The NN has one hidden layer with 985 nodes and ReLU activation, followed by a 46% dropout layer and an output layer with logistic sigmoid activation. They perform a train:test split of around 90:10 and achieve accuracies of around 90% for the validation set.

Binary classification by freely acting discrete symmetries of CICYs

Lastly, the authors of [206] study the binary classification of CICYs into classes that have a discrete symmetry and classes that do not. Since most CICYs do not have a (linearly realized), discrete, freely acting symmetry, the authors perform data augmentation via the Synthetic Minority Oversampling Technique (SMOTE) and by using symmetries (permutations of rows or columns of the configuration matrix). The performance is evaluated using a confusion matrix, the F-value and the AUC under the ROC curve. The authors find that neither the NN nor the SVM performs well on this problem, with the best F-value of only 0.25 for the SVM and 0.26 for the NN. They also find that SMOTE augmentation does not improve performance of the SVM. Data augmentation via permutation led to better (but still not good) results, with an F-value of 0.45 and 0.36 for the SVM and the NN, respectively.

Binary classification of elliptically fibered CICYs with small hodge numbers

Essentially the same network as in Table 2b is used in [207] (they use 5 instead of 100 nodes in the hidden layers) to binary classify CICY threefolds and fourfolds of small Hodge numbers according to whether or not they are elliptically fibered. Since essentially all CICYs are elliptically fibered, the authors perform data augmentation of the non-fibered cases to obtain a balanced training set. Up to the Hodge number they consider ($h^{1,1} \leq 4$), there are 643 CICYs, 53 of which are not elliptically fibered (in fact, these 53 are the only ones in the entire data set that are not elliptically fibered). Since CICYs are invariant under permuting rows or columns of their configuration matrix, it is easy to generate equivalent CICYs by just performing these permutations. The authors include permutations until they have roughly 5000 elliptically fibered and 5000 non-elliptically fibered CICYs. The configuration matrices for these small Hodge numbers are padded to 6×7, which is flattened into an input vector of length 42.

The authors do not specify the hyperparameters. They analyze the performance of the NN with different train:test splits. Already for a train:test split of 30:70, the accuracy is above 99%. The analysis is repeated for CICY fourfolds with $h^{1,1} \leq 5$ using the same NN. In this case, the input, i.e. the padded CICY configuration matrices, are 5×9, which is flattened to an input vector with 45 features. The data is again augmented to create a somewhat balanced data set with 70,000 elliptically fibered and 54,000 non-elliptically fibered CICY fourfolds. For a train:test split of 90:10, the NN reaches accuracies of around 85%. The authors also perform an experiment where they assign the input randomly to two classes and show that in this case the NN achieves accuracies of 50%, as expected.

10.5. Conjecture generation, intelligible AI and equation learning

10.5.1. Conjecture generation via logistic regression

The use of machine learning in string theory to generate and prove conjectures was pioneered in [4]. The authors use linear and logistic regression, as well as decision trees. They use the Scikit learn library implementation of these algorithms. This paper appeared shortly after [1–3] and is among the first papers on machine learning in string theory. The authors first consider the set of 4319 reflexive toric 3D polytopes, whose anti-canonical hypersurface defines a K3 surface, and investigate whether a machine learning algorithm can predict the number of fine regular star triangulations (FRSTs) of these polytopes. The number of FRSTs has been computed explicitly in [208] using different techniques for toric threefolds A with $h^{1,1}(A) \leq 22$, which allows the authors to cross-check their results. The largest $h^{1,1}(A)$ in this data set is $h^{1,1}(A) = 35$.

The authors proceed by first estimating the number of fine regular triangulations $N_{\text{FRT}}(F)$ of each facet F of the 3D reflexive polytope Δ. The number N_{FRST} of FRST triangulations can then be estimated to be

$$N_{\text{FRST}} \simeq \prod_{F \in \Delta} N_{\text{FRT}}(F)\,. \tag{234}$$

The feature vector used for training is a vector in $\mathbb{Z}^4$,

$$x^{(0)} = (n_p, n_i, n_b, n_v)\,, \tag{235}$$

where n_p the total number of points, n_i is the number of interior points, n_b is the number of boundary points, and n_v is the number of vertices of a facet. To test the extrapolation robustness, the authors train the regression tree with the results for N_{FRT} for toric varieties A with $4 \leq h^{1,1}(A) \leq 18$, compute the extrapolation to $19 \leq h^{1,1}(A) \leq 21$, and compare with the exact result.

Out of the tested methods, k-nearest neighbor regression and regression trees work best. Since the latter produce better results for larger $h^{1,1}(A)$, and the authors are interested in using the result to estimate the number of FRSTs for all toric varieties in the set (which have up to $h^{1,1}(A) = 35$), they proceed with regression trees. The authors evaluate the quality of the fit using the mean absolute percentage error (MAPE) for the predictions of the regression tree and find an average of 6.38%. For the extrapolated cases, the MAPE is between 10 and 17 percent. The predictions from the regression tree for values of $h^{1,1}(A)$ up to 27 also agree with the estimates for these cases obtained in [208] using different techniques.

As a second example, the authors consider non-Higgsable clusters in F-Theory. It is observed that an E_6 gauge group appears predominantly in the ensemble of models constructed in [15] if one blows up over the vertex $v_{\text{E}_6} = \{1, -1, -1\}$ (there is no guarantee that E_6 does appear over this vertex, but in a random sampling it has been found that if at all, E_6 appears over this vertex). This leads to a binary classification problem of whether a given model will have an E_6 gauge

group. The feature vector is the maximum number of blowups over each of the vertices of the toric polytope Δ. Since the training set is unbalanced (1:1000 of the models have an E_6 gauge group), the authors perform oversampling and use a train:test split of 80:20, which achieves accuracies around 99% using logistic regression, support vector machines, decision trees or k-NN. By further studying the predictions of the logistic regression, the authors observe that E_6 never occurs if there are up to 5 possible blowups over v_{E_6} and it occurs rarely if there are at most 4 blowups possible over v_{E_6}. The authors make these notions more precise, formulate a conjecture for when E_6 can occur and prove it subsequently. This is the first example for a conjecture obtained from machine learning in string theory that has been turned into a proven theorem.

10.5.2. Regression via equation learning

In [202], the authors apply equation learning in order to estimate the number of fine regular star triangulations (FRSTs) of the Kreuzer–Skarke data set. The total estimate of geometries is $10^{10,505}$. The authors train a NN to estimate the number of fine regular triangulations (FRTs) of all facets and then compute the number of FRSTs by multiplying all FRTs of all facets of a given polytope. The entire data set contains 7.5 billion facets (as counted by the vertices of the dual polytope). However, many of these are most likely equivalent up to toric morphisms. Hence the authors face the inverse problem discussed in Section 3.2: The data set contains a lot of symmetry augmented data and the authors want to reduce this data set. In order to do so, the authors use the so-called Kreuzer normal form, which is identical for polytopes that differ only by toric morphisms. In order to apply the algorithm to facets, one constructs a polytope for the facet by adding the origin, which is the unique interior point of the polytope whose facets are being investigated, since this polytope is reflexive. Using PALP to carry out the computations, the authors find that there are 45 million inequivalent facets, i.e. only 0.6% of the facets in the data set are inequivalent.

As a next step, the authors compute the number of FRTs for as many of the facets as possible. They are able to find all FRTs for 472,880 facets, i.e. for 1% of all facets. However, these facets appear in the vast majority (approximately 88%) of the toric polytopes at least once. As a first attempt, the authors use a regression tree. The features are the same as in the toric threefold case discussed in the previous section, cf. Eq. (235). They perform a train:test split of 60:40 and obtain a MAPE of 5%. In order to judge how well the extrapolation works, the test set contains the known quantities for larger $h^{1,1}$, and the authors find that the error increases to over 10% for the known extrapolated cases. In particular, the regression tree never predicts a value for the number of FRT that is larger than the largest number in the training set.

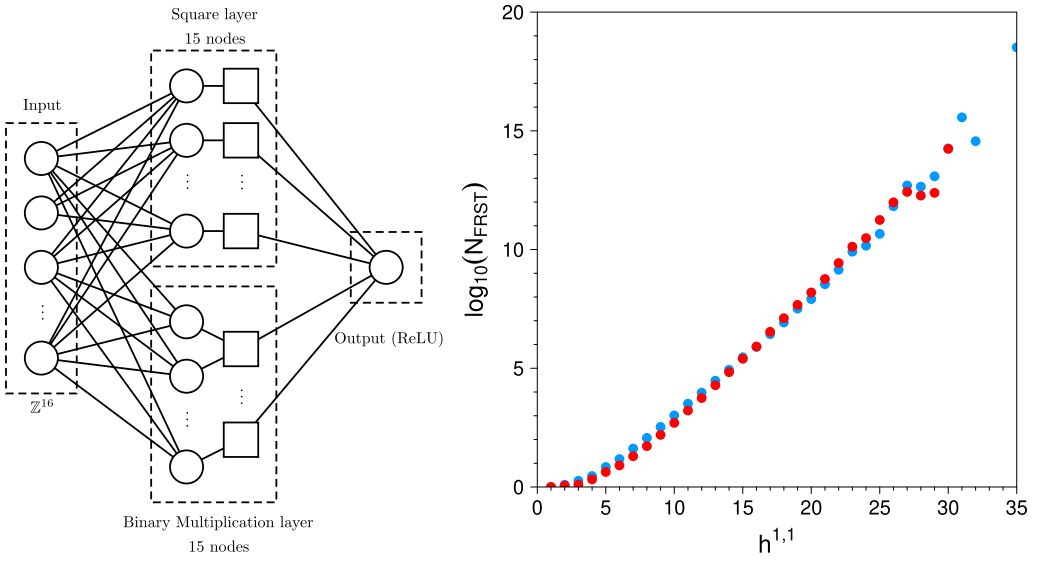
In order to improve on the situation, the authors turn to a NN. They use it as an equation learner as described in [209] and first applied in string theory in [3]. The idea is to include layers that perform an algebraic operation (such as multiplication of all features), which cannot be done directly by a NN. The authors suspect a polynomial behavior for the growth of FRTs. For the NN, more features on top of those specified in Eq. (235) are provided:

- The number of points in the 1- and 2-skeleton of the facets. The n-skeleton of a polytope consists of all points that are not interior to any $(n+1)$-dimensional face.
- The first $h^{1,1}$-value at which the facet appears in a dual polytope.
- The number of faces and edges.
- The number of possible flips of any FRT. A flip consists of removing an (internal) edge and adding another. Since edges correspond to curves, this can be thought of as blowing down a curve and blowing up another one.
- The number of 1-, 2-, and 3-simplices in a triangulation (note that these are invariant under flips).
- The number of 1- and 2-simplices in a triangulation (without removing possible redundancies).
- The number of 1- and 2-simplices which N 2- and 3-simplices have in common, respectively, for $2 \leq N \leq 5$.

The authors construct a NN which has both unary and binary activation functions. The unary nodes compute the square of the input, while the binary layers are multiplication layers which multiply two input values and return the product as output. The authors use a NN with a single hidden layer consisting of 45 nodes. 15 of the nodes are unary layers and 30 are binary multiplication layers. The output layer is one-dimensional and has a ReLU activation function, since the NN is trained to output the log of the number of FRTs, which is non-negative. The NN architecture is summarized in Fig. 56a. In order to better illustrate the difference between the unary and binary activation functions, we represent the fully connected feed-forward layer with unary activation functions with circles as usual and binary activation functions (that square or multiply their input) with squares.

The NN is trained with an ADAM optimizer with hyperparameters $\beta_1 = 0.9$, $\beta_2 = 0.99$ and $\epsilon = 10^{-8}$. The authors furthermore use L_1 weight regularization with $\lambda = 0.001$ and a dropout layer with 10% dropout rate after the hidden layer. The NN is trained with polytope data up to $h^{1,1} = 11$ and tested for $12 \leq h^{1,1} \leq 14$, for which analytic results are still known. It is found that the MAPE stays constant across this range, so in particular for the extrapolated cases. Nevertheless, the amount of extrapolation needed in this data set is huge: analytic results are known up to $h^{1,1} = 14$, while the highest $h^{1,1}$ in the data set is 491.

In order to better cross-check the architecture, the authors thus return to the toric threefolds discussed above. For this data set, reliable numbers for FRTs up to 25 are available. Skipping the regularity check and just finding fine triangulations, the authors can even compute the number of fine triangulations up to $h^{1,1} = 30$. In this data set, regularity is a rather weak (but very expensive) condition to check (at most 2.6% of the fine triangulations of a facet are not regular). They find



(a) NN architecture used in [202]. (b) Extrapolation validation for toric threefolds.

Fig. 56. The NN architecture of the equation learner used in [202] and a comparison of the extrapolation performed by the NN (blue) against the known results for toric threefolds (red).

that training the NN in these cases with data for $4 \leq h^{1,1} \leq 11$ and extrapolating to $12 \leq h^{1,1} \leq 30$, the extrapolation of the NN agrees with the known result, cf. Fig. 56b.

The results of the paper seem to imply that the number of FRSTs of a toric polytope can be computed in terms of a quadratic polynomial in the input data described above. It would be interesting to understand this relation on an analytic basis.

10.5.3. Classifying gauge groups from F-theory bases

In [210], the authors compare different machine learning techniques to predict the type of non-Higgsable gauge groups in F-Theory on elliptically fibered Calabi–Yau fourfolds over a toric threefold base B_3 described in Weierstrass form

$$y^2 = x^3 + f(B_3)xz^4 + g(B_3)z^6\,, \tag{236}$$

where $[x : y : z]$ are the coordinates of $\mathbb{P}_{2,3,1}$. In fact, it is enough to only provide information about the threefold base B_3. The authors first restrict to bases without so-called $(4, 6)$-curves . These are curves (i.e. codimension two loci in the base) which force a vanishing order of 4 and 6 for f and g in (236). They train a decision tree for these bases and find that it predicts the gauge algebra with an accuracy of 99.5%. They then apply this trained decision tree to bases with $(4, 6)$-curves and still obtain large accuracies, around 85%–98% on the test set.

In more detail, the feature vectors used for the classification contains all triple intersections of a base divisor D (for which the presence of a non-Higgsable gauge group shall be tested) with all its "neighboring" divisors D_i, $i = 1, \ldots, h^{1,1}(D) + 2 =: p$. This feature vector is $5p$-dimensional. The first p elements are the triple-intersection numbers $D.D_i^2$, $i = 1, \ldots, p$. These fix the topology of D. The next p features are the triple intersections $D^2.D_i$ which fix the normal bundle N_D of D in B_3. The last $3p$ features are the remaining triple intersection numbers D_i^3 and $D_i^2 D_{i+1}$ and $D_i D_{i+1}^2$ for $i = 1, \ldots, p$ (and the convention that $p + 1 \equiv 1$). These encode the properties of the divisors D_i neighboring D. Note that these features are not necessarily all independent.

The non-Higgsable gauge groups G that appear in the data set used by the authors are

$$G = \{\varnothing\,, \mathrm{SU}(2),\ \mathrm{SU}(3),\ G_2,\ \mathrm{SO}(8),\ F_4,\ E_8\}\,. \tag{237}$$

This turns the problem into a classification problem with $|G| = 7$ classes. Since the data set is extremely unbalanced (70% of the training set has $G = \varnothing$ while 2.7×10^{-4} % has $G = \mathrm{SO}(8)$), the authors use over- and undersampling to create a balanced training set with $\sim 1.4 \times 10^6$ samples in total ($\sim 20,000$ per class). The authors use Scikit learn for their analysis (except for the feed-forward NN implementation, for which they use Keras) and find the following results (the authors specify in which cases they deviate from the standard Scikit learn parameters but do not state explicitly which value they used):

- **Decision trees:** Using an untrimmed tree with Scikit learn's standard parameters, an accuracy of 99.5% is reached within 22 s of training.

- **Random Forests:** Using 10 trees and Scikit learn's standard hyperparameters, random forests reach an accuracy of 99.6% after 53 s of training.
- **SVMs:** Using a Gaussian kernel and optimizing C and γ, SVMs reach an accuracy of 97.4% after 24 s of training.
- **NN:** The NN consists of 2 hidden layers with 10 nodes each, one dropout layer, and a softmax activation function in the output layer. They train for 5 epochs but do not provide further details on the activation function of the hidden layers, the initialization, optimizer, loss function, etc. The NN training takes 3.2 h and reaches accuracies of 97.0%. The authors remark that upon further hyperparameter tunings, better results can most likely be achieved, but they will focus on DT in the remainder anyway.
- **Logistic regression:** Optimizing C, L_1 and L_2, an accuracy of 77.6% is reached with 324 s of training.

The accuracy is impressively high for most of the techniques tested by the authors. Using decision trees, the authors study in more detail which features the ML algorithm uses in the classification process for bases with $h^{1,1} = 1, 2, 3$.

They find that in the case where $h^{1,1}(D) = 1$ (the unique possibility for this case is that the divisor inside B_3 has the topology of a $\mathbb{P}^2$), the most important feature is D^2D_1, which directly encodes the normal bundle: For $\mathbb{P}^2$ the normal bundle inside B_3 can be written as $N = aH$ where $a \in \mathbb{Z}$ and H is the hyperplane class of $\mathbb{P}^2$. The intersection D^2D_1 then is simply given by a. This is to be expected, since the normal and the canonical bundle of the base divisor D enter in f and g in (236) and thus determine their vanishing order and consequently the resulting gauge group in F-Theory. From the decision trees, the authors can infer conditions on the intersection numbers of D with D_i that lead to an E_8 singularity. In a similar way, conditions on the triple intersection numbers can be derived that are necessary for a particular gauge group to appear, see [210] for details.

The $h^{1,1}(D) = 2$ case corresponds to Hirzebruch surfaces $\mathbb{F}_n$. These can be thought of as $\mathbb{P}^1$ fibrations over $\mathbb{P}^1$ and their normal bundle is given by $N = aS + bE$ where $a, b \in \mathbb{Z}$ and S and E are the classes of the base and fiber $\mathbb{P}^1$, respectively. Interestingly, the authors find from the decision trees that the most important feature is the coefficient a in the normal bundle, while b and, surprisingly, n (i.e. the twisting parameter that encodes the non-triviality of the fibration of $\mathbb{P}^1$ over $\mathbb{P}^1$ leading to $\mathbb{F}_n$) are much less important.[15] Again, the authors derive necessary conditions from the decision tree for the appearance of non-Higgsable gauge groups. The authors use these conditions to construct previously unknown infinite chains of non-Higgsable SU(3) groups.

The cases with $h^{1,1}(D) = 3$ can be obtained by blowing up $\mathbb{F}_n$ at a point. The normal bundle of D inside B_3 can be expanded in terms of the three curves of this divisor that form a basis of $H^{1,1}(D)$. Again, the most important feature is the coefficient of one of the three curve classes that appear in the expansion of the normal bundle. The other two parameters are the second and sixth most important features.

The authors also discuss qualitatively results for larger $h^{1,1}(D)$. However, the decision trees get more and more complicated. Interestingly, the authors observe that for $h^{1,1} > 4$ the number of nodes N_{nodes} in the decision tree grows linearly with the number of training samples N_{samples},

$$N_{\text{nodes}} = 0.038 N_{\text{samples}} + 937.59\,. \tag{238}$$

In contrast, the depth of the tree is uncorrelated with the number of samples.

In summary, the authors achieve impressive results on the accuracy of predicting gauge groups based on toric data for 3D bases in F-theory. The authors find and apply new rules using intelligible AI (decision trees). They further demonstrate that the results obtained for bases without $(4, 6)$-curves can be transferred with only a small loss in accuracy to cases with $(4, 6)$-curves, even though these were not included in the training set. They also observe a linear relation between the number of training samples and the number of nodes in the decision tree.

10.6. Generating superpotentials with GANs

In [211], generative adversarial networks were used to generate superpotentials (i.e. holomorphic functions). The authors study the simple case of global $\mathcal{N} = 1$ supersymmetry whose superpotential is a quadratic polynomial in one chiral superfield $\Phi = (\phi, \psi)$,

$$W = \sum_{i=0}^{2} \alpha_i \Phi^i\,, \qquad \Rightarrow \qquad V = \frac{\partial W}{\partial \phi}\frac{\partial \overline{W}}{\partial \overline{\phi}}\,. \tag{239}$$

The resulting scalar potential V is consequently also a quadratic polynomial. The coefficients α_i are drawn from a uniform distribution. They compute the superpotential by varying the real and imaginary part of ϕ in the interval $[-2, 2]$. They discretize this box by using a grid of size 64×64. The potential is then normalized such that the maximum real and imaginary part is 1. They plot the scalar potential on this grid in a contour plot, i.e. they assign a color based on the value of the scalar potential at each grid position. Since the underlying superpotential is holomorphic, it has to satisfy the Cauchy–Riemann equations.

[15] Note that $\mathbb{F}_n$ has a $-n$ curve, which, in F-Theory compactifications to 6D, is the main feature that decides the non-Higgsable gauge group.

Table 5
GAN used for generating superpotentials in [211]. Conv2D and TConv2D are 2D convolutional layers and 2D transposed convolutional layers with f filters and kernel size k (and stride 1). FC are fully connected layers.

(a) Discriminator		(b) Generator	
Layer	Parameters	Layer	Parameters
Input	$64 \times 64 \times 2$ nodes	Input	100 nodes (noise)
Conv2D	$(f: 8,\ k: 2 \times 2)$	FC	65536 nodes
Leaky ReLU	0.2	Batch norm	65536 nodes
Dropout	40%	ReLU	65536 nodes
Conv2D	$(f: 16,\ k: 2 \times 2)$	Reshape	$16 \times 16 \times 256$ nodes
Leaky ReLU	0.2	Dropout	40%
Dropout	40%	TConf2D	$32 \times 32 \times 256$ nodes
Conv2D	$(f: 32,\ k: 2 \times 2)$	Conv2D	$(f: 128,\ k: 3 \times 3)$
Leaky ReLU	0.2	Batch norm	$32 \times 32 \times 128$ nodes
Dropout	40%	ReLU	$32 \times 32 \times 128$ nodes
Conv2D	$(f: 64,\ k: 2 \times 2)$	TConf2D	$64 \times 64 \times 128$ nodes
Leaky ReLU	0.2	Conv2D	$(f: 64,\ k: 3 \times 3)$
Dropout	40%	Batch norm	$64 \times 64 \times 64$ nodes
Flatten	262,144 nodes	ReLU	$32 \times 32 \times 128$ nodes
Output	1 node	Conv2D	$(f: 32,\ k: 2 \times 2)$
Sigmoid	1 node	Batch norm	$64 \times 64 \times 32$ nodes
		ReLU	$64 \times 64 \times 32$ nodes
		Conv2D	$(f: 2,\ k: 2 \times 2)$
		Tanh	$64 \times 64 \times 2$ nodes

The authors of [211] set up the GAN as detailed in Table 5 and train it with 10,000 randomly generated (holomorphic) superpotentials of degree two. They deliberately use small convolutions in the discriminator (kernel size 2 and stride 1): this allows the NN to check the Cauchy–Riemann equations (i.e. holomorphicity of its input) locally but makes it impossible for the discriminator to distinguish "real" and "fake" superpotentials based on global properties. The reason why the authors chose to do this is because they wanted to see whether the generator generates superpotentials that are holomorphic but not just quadratic polynomials like the ones the discriminator was trained with.

The GAN is trained on a GPU with mini-batch size of 256 with an RMSProp optimizer with a learning rate of 2×10^{-4} and a weight decay rate of 6×10^{-8}. The authors find that after 20,000 iterations the deviation from a holomorphic function at each grid point, as measured by the Cauchy–Riemann equations, becomes much smaller than the absolute value of the superpotential at that grid point (the authors average over 16 generated examples). This illustrates that the GAN learns to generate holomorphic functions. The authors also find that the GAN generates images of potentials that cannot be fit by a quadratic polynomial, even though the training sample was generated from quadratic polynomials. As mentioned above, the authors deliberately designed the discriminator such that it would not be powerful enough to check whether a generated image has the same global properties (i.e. one extremum) as a training image. The rationale behind this is that the authors just want any holomorphic function, not necessarily a quadratic one, which was indeed achieved with the GAN of Table 5.

10.7. Study distribution of string vacua

10.7.1. Topological data analysis of compactification geometries

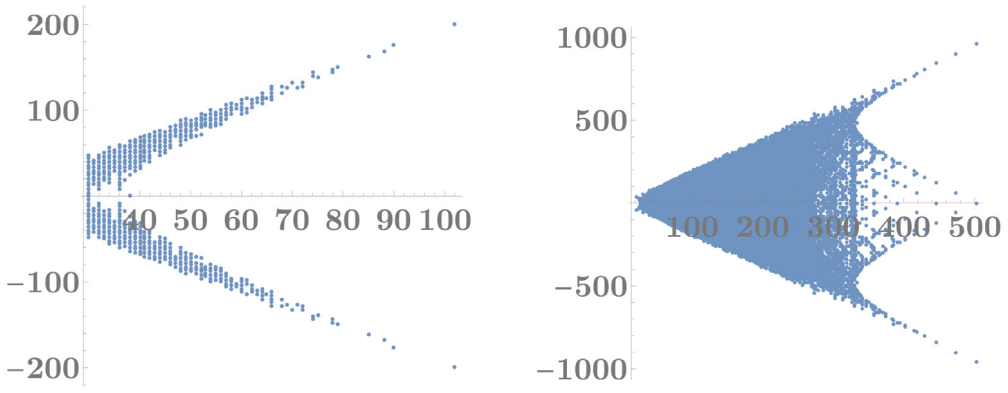
The first application of topological data analysis (TDA), more precisely persistent homology (PH), to string compactification spaces was carried out in [134]. The author studies the Kreuzer–Skarke (KS) and the complete intersection Calabi–Yau (CICY) data sets. A two-dimensional feature vector is used to encode the Hodge numbers via

$$x^{(0)} = (h^{1,1}(X) + h^{2,1}(X),\ \chi(X)), \tag{240}$$

where $\chi(X) = 2(h^{1,1}(X) - h^{2,1}(X))$ is the Euler number of the threefold X. In this way, the feature vector encodes the two independent Hodge numbers of the CY threefold X. The author augments the Hodge numbers with the ones of the mirror CYs. These are in general not in the CICY data set (a notable exception is the Schoen manifold, which is self-mirror), since the mirror of a CICY cannot be written as a CICY in a product of projective ambient spaces. The point clouds obtained by plotting the features (240) are given in Fig. 57.

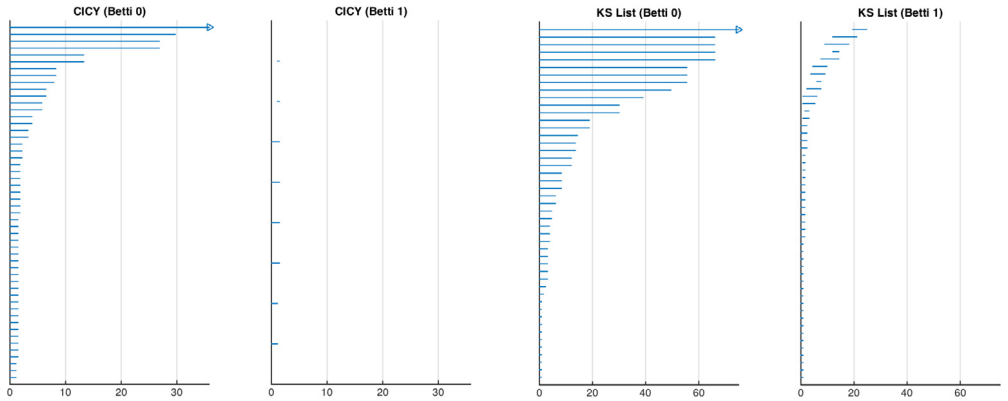
The PH is computed using a lazy witness complex with 200 landmark points. In order to speed up the computation, the homology is computed in a finite field of characteristic 2, i.e. in $F_2 \simeq \mathbb{Z}_2$. The resulting barcodes for the KS and the CICY data set were computed in [134] using JAVAPLEX and are given in Fig. 58.

Let us discuss some of the features of the barcode. Most elements of h^0 are short-lived, indicating a dense and somewhat uniform distribution of points in the point cloud. Bars in the barcode of b^0 that persist longer indicate the presence of isolated points or clusters, of which there are a few. The fact that some bars in the barcode come in pairs, i.e. they die at the same time, is due to mirror symmetry, which leads to a reflection-symmetric point cloud along $h^{1,1}(X) = h^{2,1}(X)$. The



(a) CICY point cloud. (b) KS CY point cloud.

Fig. 57. Point clouds for the features of (240) for the (mirror symmetry augmented) CICY and the KS data set.



(a) PH of CICYs. (b) PH of KS CYs.

Fig. 58. Persistent homology for CYs constructed as complete intersections in projective ambient spaces and as hypersurfaces in toric ambient spaces as obtained in [134].

fact that not all bars come in pairs indicates the presence of points or clusters close to or on this line. Most one-cycles are short-lived, indicating the absence of larger voids in the distribution. Comparing the CICY and the KS barcodes, it is found that in the CICY case the barcodes become trivial at a filtration parameter of $\epsilon \sim 30$, while this happens much later for KS CYs (at $\epsilon \sim 70$). This indicates that the KS set is topologically richer. Indeed, the maximum Hodge number of a CICY is 101 (which is $h^{2,1}$ of the quintic and is included as a possible value for $h^{1,1}$ by mirror symmetry), while the maximum Hodge number of a KS CY is 491. Furthermore, all 7890 CICYs in the CICY list have only 266 different Hodge pairs (529 when mirror symmetry is included), while KS CYs have 30,108 different pairs.

The author repeats the PH analysis for just the tip of the point cloud, i.e. for CYs with small Hodge numbers. It is observed that in the case $h^{1,1} + h^{2,1} < 40$, two one-cycles form at rather large filtration parameter (the fact that there are two is due to mirror symmetry). These two one-cycles also form if one zooms in even more, i.e. restricts to CYs with $h^{1,1} + h^{2,1} < 22$. The author argues that, in a sense, CYs with small Hodge numbers are special, since they have a special topological feature.

As a next example, the author considers Landau–Ginzburg models as classified in [212,213]. They are closely related to the compactification spaces discussed above, since the classification focuses on Landau Ginzburg orbifolds with 5 chiral fields, which correspond to Calabi–Yau manifolds given as hypersurfaces in a weighted projective ambient space. Since these are toric, they allow for a description in terms of a 2D gauged linear sigma model (GLSM). This sigma model has different phases (Higgs branches), one of which is the Landau–Ginzburg orbifold phase. The states in the Landau–Ginzburg model are given by the chiral ring R of the model, which is defined by

$$R = \frac{\mathbb{C}[Z_1, \ldots, Z_N]}{\langle \partial W \rangle} . \tag{241}$$

Here, Z_i are two-dimensional $\mathcal{N} = (2, 2)$ chiral superfields, $\mathbb{C}[Z_i]$ denotes the polynomial ring over $\mathbb{C}$, and $\langle \partial W \rangle$ is the Jacobian ideal of the GLSM superpotential. The classification of [212,213] contains more than 10,000 models with 3798

inequivalent spectra. Since $\mathcal{N} = (2, 2)$ models are considered, the Landau–Ginzburg models correspond to the standard embedding in the heterotic $E_8 \times E_8$ theory. Standard embedding means that the gauge connection of the vector bundle of the heterotic theory is equal to the spin connection of the CY (which is SU(3) for CY threefolds).

The Hodge numbers in the (geometric) CY phase can be determined from the chiral ring (241). The author computes again the PH and the resulting barcodes (over the field $\mathbb{Z}_2$) for these models and finds that they differ from the above barcodes, even though the construction and the defining data are very similar. The author goes on to study PH of flux vacua, which we discuss in the next section.

10.7.2. Topological data analysis of flux vacua

In addition to studying PH of string compactification spaces, the author of [134] also applies TDA to flux vacua in Type IIB and heterotic string theory. The heterotic flux vacua are MSSM-like line bundle models on complete intersection Calabi–Yau (CICY) manifolds constructed in [185]. The Type IIB examples considered are a rigid CY and a hypersurface in weighted projective space. These two examples are also considered in [135]. In addition, [135] also discusses the PH of a symmetric six-torus.

A heterotic string compactification on a smooth CY threefold is described in terms of geometry and gauge data. The geometry of the models constructed in [185] was chosen to be a CICY from the data set of [166]. Given a background geometry X, a vector bundle V has to be chosen such that the Bianchi identities for the three-form field H are satisfied, which implies

$$\mathrm{ch}_2(TX) - \mathrm{ch}_2(V) = [W]\,, \tag{242}$$

where $\mathrm{ch}_2(\cdot)$ denotes the second Chern character and $[W]$ is the cohomology class of an effective curve W (rather, W is the four-form Poincare dual to an effective curve).

In order to satisfy the SUSY equations of motion, Donaldson and Uhlenbeck and Yau showed that the vector bundle V needs to be polystable with vanishing slope, i.e. the bundle needs to be the direct sum of stable bundles V_i which have the same slope

$$\mu(V_i) = \frac{1}{\mathrm{rk}(V_i)} \int_X c_1(V_i) \wedge J \wedge J\,, \tag{243}$$

where $\mathrm{rk}(V_i)$ is the rank of the bundle V_i, J is the Kähler form on X, and $c_1(V)$ is the first Chern class of V. Stability means that every sheaf $\mathcal{F}$ that injects into V has a strictly smaller slope than V. This condition is very hard to check in practice. A huge simplification is obtained if one assumes that V is a sum of line bundles $\mathcal{L}_a$,

$$V = \bigoplus_{a=1}^{\mathrm{rk}(V)} \mathcal{L}_a\,. \tag{244}$$

In this case, each constituent bundle $\mathcal{L}_a$ does not have a subbundle (since $\mathcal{L}_a$ is already of rank one), so all that remains to be checked is the slope zero condition and the Bianchi identities.

A line bundle on X is defined in terms of its first Chern class, which is given in terms of $h^{1,1}(X)$ integers, see also Eq. (227). The bundles in [185] are $S(U(1)^5)$ bundles, i.e. they consist of 5 line bundles with one tracelessness condition. At this point, this only leads to SU(5) GUT models. In order to break the GUT group down to the gauge group $SU(3) \times SU(2) \times U(1)$ of the Standard Model, a freely acting discrete Wilson line is used. Since a CY has no non-trivial one-cycles, $h^{1,0}(X) = h^{0,1}(X) = 0$, this Wilson line needs to be supported by a non-trivial first fundamental group of X, $\pi_1(X) \neq \varnothing$. Since all CICYs have a trivial $\pi_1(X)$, one needs to consider free quotients of X by discrete symmetry groups.

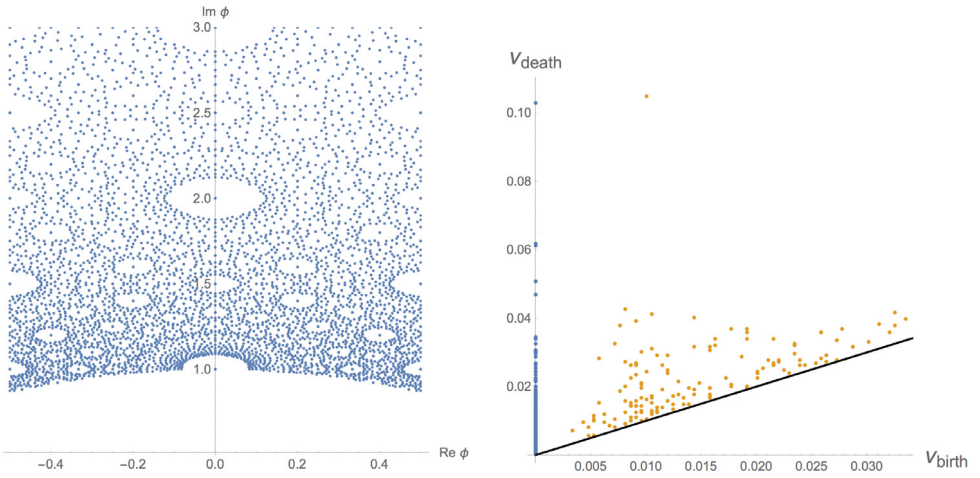
The author embeds the data describing these line bundle models into a 7-dimensional feature space. He writes that 5 of the features are $c_2(\mathcal{L}_i)$, $i = 1, \ldots, 5$. Since the second Chern class of a line bundle is zero, I assume what is meant is

$$c_{2,i}(V) = \int_{D_i} c_2(V)\,, \tag{245}$$

where D_i, $i = 1, \ldots, h^{1,1}$ is a divisor basis of $H_4(X)$. Since the author assigns 5 features to $c_{2,i}$, I assume that he focused on the line bundle models over CICYs with $h^{1,1}(X) = 5$ (roughly a third of the models of [185] are on such CYs). The feature vector is then

$$x^{(0)} = (h^{1,1}(X), h^{2,1}(X), c_{2,1}(V), c_{2,2}(V), c_{2,3}(V), c_{2,4}(V), c_{2,5}(V))\,. \tag{246}$$

The author computes the PH for the first four Betti numbers b_0 to b_3 and finds that the data set leads to non-trivial elements in all 4 homology classes, even though the higher cycles are very short-lived. Next, the author adds in two more features: the number of vector-like pairs in the fundamental representation **5** of SU(5) (i.e. before GUT breaking), and the number of Higgs pairs (i.e. the number of vector-like doublets after GUT breaking). Note that, depending on the action of the quotient, the latter can be smaller than the former. The author computes the barcodes for two classes of models: the first has three vector-like **5**-plets and the second has one vector-like **5**-plet; both lead to models with one Higgs pair after including the Wilson line. It is observed that the models with one **5**-plet have a richer PH as compared to the ones with three **5**-plets. Following the conjecture that phenomenologically interesting models have a more complex PH, this



(a) Φ values for a rigid CY. (b) Persistence diagram.

Fig. 59. Vacuum solutions for the axio-dilaton Φ in a rigid CY and the corresponding persistence diagram, taken from [135]. The long-lived one-cycles (orange) in the diagram correspond to the voids in the distributions of flux vacua.

opens up the possibility of selecting string vacua based on how rich their topology is. Clearly, it is desirable to study the PH of more vacua in order to check this conjecture.

Next, rigid CYs are studied. A rigid CY threefold X is a CY without complex structure deformations, i.e. $h^{1,2}(X) = h^{2,1}(X) = 0$. Hence the third Betti number is 2, accounting for the holomorphic $(3,0)$-form Ω (and its conjugate $(0,3)$-form). A symplectic basis of $H_3(X,\mathbb{Z})$ is then given by one A and one B cycle with

$$\int_A \Omega = 1\,, \qquad \int_B \Omega = i\,. \tag{247}$$

Writing $A = -h_1 - ih_2$ and $B = -f_1 - if_2$, the D3 brane charge induced by fluxes is the product

$$N_{\text{flux}} = f_1 h_2 - f_2 h_1\,, \tag{248}$$

and the induced superpotential for the axio-dilaton ϕ reads

$$W = A\phi + B\,. \tag{249}$$

The F-flat direction is

$$D_\phi W \stackrel{!}{=} 0 \quad \Rightarrow \quad \phi = -\overline{(\bar{B}, A)}^{\,T}\,. \tag{250}$$

The authors of [134,135] compute random vacua solutions and use the SL$(2,\mathbb{Z})$ symmetry of the axio-dilaton to map the solutions into the fundamental domain. The valid solutions for ϕ obtained in [135] this way are plotted in Fig. 59a.

Note that the structure of vacuum solutions has large voids with one data point in their center. On persistence diagrams, this structure will be reflected in long-lived one-cycles. Moreover, the death of the one-cycle corresponding to the void will be correlated with the death of a zero-cycle (when the point in the center fills in the void). The authors construct a lazy witness complex with 750 landmark points. Indeed, this behavior is observed in the persistence diagrams. In order to illustrate a further application, the authors of [134,135] focus on the imaginary part of Φ and observe that many 0-cycles die at filtration parameter 1. This is because tadpole cancellation and flux quantization dictate that the spacing of two vacua is in multiples of i at large Im(Φ). At smaller Im(Φ), the spacing is 0.5, and the authors also observe a larger number of cycle death at this value.

As a second example, the authors of [134,135] consider the CY threefold defined by a hypersurface in $W\mathbb{P}^4_{1,1,1,1,4}[8]$ with homogeneous coordinates $[x_0 : x_1 : \ldots : x_4]$,

$$\sum_{i=0}^{3} x_i^8 + 4x_4^2 - 8\psi x_0 x_1 x_2 x_3 x_4 = 0\,. \tag{251}$$

This CY allows for an orientifold action $x_4 \to -x_4$, $\psi \to -\psi$, combined with a worldsheet parity reversal. The anti-canonical hypersurface in $W\mathbb{P}^4_{1,1,1,1,4}[8]$ has furthermore a $\mathbb{Z}_8^2 \times \mathbb{Z}_2$ discrete symmetry, and Eq. (251) has the most general form compatible with this symmetry. This means that the discrete quotient reduces the original $h^{2,1} = 149$ down to $h^{2,1} = 1$, which is parametrized by ψ. The point $\psi = 1$ is a conifold point. Using Monte Carlo methods, it has been confirmed that vacua cluster around this conifold point.

For this CY, we need to consider two complex fields, the axio-dilaton ϕ and the complex structure deformation field ψ. Again, there are symmetries in the solution. The axio-dilaton ϕ has, as before, an SL(2, $\mathbb{Z}$) symmetry and the complex structure has a logarithmic monodromy around the conifold point. To account for this, the authors map ϕ into the fundamental domain and take $\arg(\psi) \in [-\pi, \pi)$.

There is a further difference as compared to the previous case. Now, the authors want to identify clusters rather than voids. While the former are more easily identifiable via long-lived 1-cycles, the latter just lead to many zero-cycles that die quickly, a behavior that is also found in uniform distributions. In order to account for this, the authors consider a multiparameter filtration and add the density as a second filtration parameter (in addition to the radius). The density is defined as the inverse distance to the kth nearest neighbor. The problem with multiparameter filtrations is that there is no straightforward way of visualizing the result. One possibility is to plot a persistence diagram for each path in the multiparameter filtration space. Instead of doing so, the authors compute persistence diagrams at two fixed parameters for the density threshold. Indeed, when adding a large threshold for the density (i.e. densely populated areas are included very late in the length persistence diagram, leading to a void where the dense cluster is), a long-lived one-cycle appears, which corresponds to the clustering of vacuum solutions around the conifold point.

The new example computed in [135] is the symmetric six-torus $T^6 = (T^2)^3$ with all three complex structure parameters set equal, $\tau_1 = \tau_2 = \tau_3 = \tau$. With this restriction, the complex structure vacuum manifold induced by fluxes is again complex two-dimensional and parametrized by the axio-dilaton ϕ and the complex structure τ. Symmetries are again taken into account by mapping solutions into the fundamental domains of the two corresponding SL(2, $\mathbb{Z}$) groups. The symmetric six torus allows for solutions which have vanishing superpotential at tree level. It is observed that these $W = 0$ solutions give rise to a different structure of solutions as compared to the generic case. In particular, the $W = 0$ solutions exhibit a richer persistent homology, i.e. higher-dimensional and longer-lived cycles appear. The authors speculate that this might be linked to the richer number-theoretic structure that emerges in flux space for $W = 0$ solutions.

In summary, the authors have shown that flux vacua have an interesting structure, reflected in a non-trivial PH. For the heterotic vacua, it was found that the point cloud describing the vacua has non-trivial k-cycles for all k that were checked (up to $k = 3$). For the rigid CYs, the complex structure flux vacua points can be plotted explicitly (since they only involve one complex scalar field) and an interesting structure can be observed. This structure is reproduced by a persistent homology computation. In the case of a hypersurface in weighted projective space, PH confirms clustering of solutions around the conifold point, which has also been observed in Monte Carlo studies. In the last example of a symmetric torus it is again observed that more special vacua (with vanishing superpotential) have a richer persistent homology than more general solutions.

10.7.3. Clustering of vacuum distributions from autoencoders

The authors of [214] also study vacua in string theory, albeit with a different approach (and in a different string theory). They study phenomenological properties of conformal field theories on heterotic orbifolds. The authors consider a $\mathbb{Z}_6$ toroidal orbifold. This is constructed by starting from a $T^6 = (T^2)^3$ with coordinates z_i, $i = 1, 2, 3$ and modding out the $\mathbb{Z}_6$ action

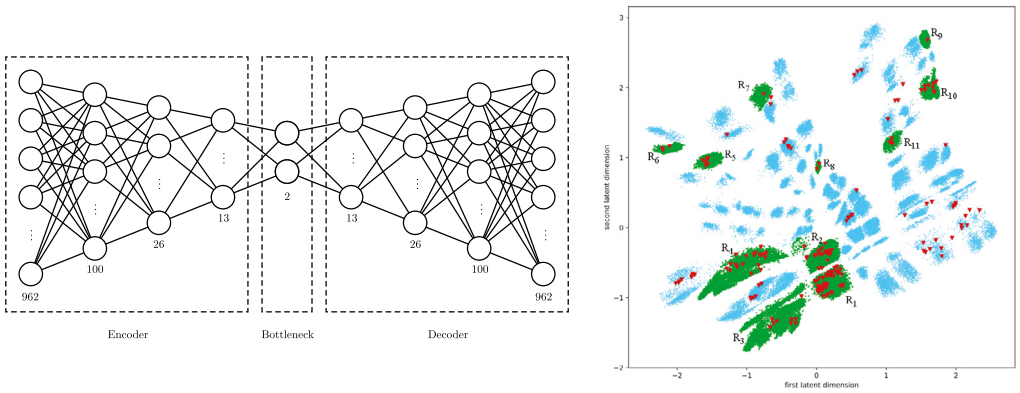
$$(z_1, z_2, z_3) \to (e^{2\pi i/6} z_1, e^{2\pi i/3} z_2, e^{2\pi i/2} z_3)\,. \tag{252}$$

Modular invariance of the one loop string partition function of the heterotic string requires this action be combined with an action on the $E_8 \times E_8$ root lattice of the heterotic string. This action is encoded in a 16-dimensional vector that specifies (fractional) lattice shifts in the 16 Cartan directions of $E_8 \times E_8$. One can furthermore include discrete Wilson lines on the orbifold, which also act as (fractional) shifts in the Cartan directions. On the six-torus, there are 6 possible Wilson lines (one for each one-cycle); however, only three of these are compatible with the orbifold action. This means that after fixing the geometry, all properties of the model are encoded in four 16-dimensional vectors in $\frac{1}{6}\mathbb{Z}^{4\times 16}$ describing the shift and the Wilson lines. Using a computer program [215], the authors construct $\mathcal{O}(700, 000)$ inequivalent models, i.e. $\mathcal{O}(700, 000)$ consistent combinations of shift vectors and Wilson lines that lead to inequivalent physics. The authors define two models to be inequivalent if they are distinguished on the level of the 4D massless spectrum, i.e. if they have a different gauge group and/or massless matter content.

This parametrization has a huge redundancy: Shift vectors and Wilson lines that look inequivalent could differ by:

- integer combinations of root lattice vectors in $E_8 \times E_8$,
- Weyl group actions in $E_8 \times E_8$.

Note that the symmetry group of this problem is huge (since the Weyl group of E_8 is huge), with more than 10^{19} elements. In order to account for these symmetries in the feature vector, the authors do not use directly the four 16-dimensional vectors that characterize the model. Instead, they construct a 26-dimensional feature vector as follows: The orbifold geometry has 12 fixed points, and each fixed point comes with a unique combination of shift vectors and Wilson lines. This unique combination means that at each of the fixed points, a different set of $E_8 \times E_8$ root lattice vectors transforms covariantly under the shifts and Wilson lines. As a consequence, the local gauge group at each of the 12 fixed points differs. The overall (global) gauge group is given by the intersection of all 12 local gauge groups in $E_8 \times E_8$. The 26 features encode the rank of the local and global gauge groups as follows: the first 12 vectors are the rank of the surviving subgroup of the first E_8 at each of the 12 fixed points. Similarly, the second 12 features are the rank of the surviving gauge group from



(a) Autoencoder architecture. (b) Bottleneck layer values [214].

Fig. 60. Autoencoder used to encode features in a $\mathbb{Z}_6$ orbifold and the resulting bottleneck layer. Phenomenologically interesting models correspond to red triangles. Green clusters are clusters to which at least one phenomenologically interesting model from the training set was mapped.
Source: Fig. 60b is taken from [214].

the second E_8. Finally, the last two features are the rank of the global gauge group coming from the first and second E_8, respectively.

The authors perform a train:test split of 60:40. They observe that the performance of the autoencoder can be enhanced if, instead of the 26-dimensional feature vector, a one-hot encoding of this vector is used. Each feature can have 37 different values (i.e. there are 37 ways in which an E_8 can be broken into subgroups on this orbifold). In the one-hot encoding, each feature is replaced by a 37-dimensional vector which contains only zeros and a single one at the position corresponding to the particular value which the feature has. Doing this for each of the 26 features one obtains a very sparse vector (i.e. a vector with many zeros) in $\{0, 1\}^{26\times 37} = [0, 1]^{962}$. The architecture of the autoencoder is summarized in Fig. 60a.

For the implementation, the authors use TensorFlow. Each layer is activated with a SELU activation function and they use an MSE loss function. They perform several tests in order to find a NN architecture that works well. With a latent space dimension of 2, the MSE on the test set is 0.013, which corresponds to the successful reconstruction of 16 features on average.

The latent bottleneck dimension of the autoencoder is 2 which allows for easy visualization of the compressed features, cf. Fig. 60b. The red triangles are phenomenologically interesting models of a larger data set with $\mathcal{O}(6, 300, 000)$ models (see below). The 11 green clusters $R_1, \ldots, R_{11}$ are clusters to which at least one phenomenologically interesting model of the training set is mapped. From this result, it can be seen that phenomenologically interesting models tend to cluster. Since the authors have an application in mind where the landscape is searched based on identifying "fertile" clusters and checking the phenomenology of all models in this cluster, they need to know which feature gets mapped to which cluster. They study this via a decision tree which classifies an input into one of the 11 clusters R_1 to R_{11} or assigns it to the rest (R_0). They construct a balanced training set by weighting the phenomenologically interesting models more strongly. For the implementation of the decision tree they use Scikit learn. The performance of the decision tree is evaluated with a confusion matrix, which shows that the tree works very well.

In order to further verify their approach, the authors perform an extensive search and identify $\mathcal{O}(6, 300, 000)$ additional models. The authors furthermore study the phenomenology of all of these models and find that the entire data set contains 177 MSSM-like models; the original set with $\mathcal{O}(700, 000)$ models contained 18 MSSM-like models. It is found that the new MSSM-like models also tend to cluster, and many of them are mapped to the fertile clusters identified by the training set with $\mathcal{O}(400, 000)$ elements.

To summarize, the authors demonstrate that MSSM-like orbifold models are mapped to the same clusters by an autoencoder. Using a decision tree, the authors can identify models (i.e. features) that are mapped to the same clusters. In future applications, this can be used to construct a fraction of all possible models, train the autoencoder and the decision tree, and then study only promising models, i.e. models whose features are such that they are mapped to fertile clusters.

10.8. Finding string vacua with reinforcement learning

In [24], we applied reinforcement learning (RL) to search for viable string vacua. As reviewed in Section 10.2, this requires solving nested NP-hard and undecidable problems, which makes a direct brute-force approach unfeasible. This is why we want to use neural networks which can perform better by either using approximations and heuristics, or by identifying patterns in the input that allows to focus on special subclasses of the NP-hard or undecidable problems which are easier to solve.

Since we cannot train a NN with a set of example solutions (since none are known), we face a search problem which we address with RL. We focus on finding MSSM-like models in Type II compactifications. More precisely, we consider orientifolds of a toroidal orbifold $T^6/(\mathbb{Z}_2 \times \mathbb{Z}_2)$. The discussion can be either phrased in Type IIA or in Type IIB (using mirror symmetry), but we will use the Type IIA description. This model has the advantage that it has been studied extensively in the past in [18,216]. A brute force search was run in [216], which took almost a year and led to almost a billion models (none of which had the exact MSSM spectrum). The authors have published many statistics on the vacua with which we can compare our results.

First, we explain what the state and action spaces are. The states are given by specifying a set of D6 brane stacks in the geometry. We choose D6 branes which extend through 4D Minkowski space and wrap a one-cycle in each of the three tori.

The $\mathbb{Z}_2 \times \mathbb{Z}_2$ orbifold action combined with the $\mathbb{Z}_2$ orientifold action only allows for two discrete choices of the complex structure parameters of each torus, referred to as untilted and tilted torus, respectively. We first tried to use three untilted tori ($\tau_a = i$ for $a = 1, 2, 3$), since this gives the least constraining tadpole equations, but the agent never found an odd number of generations. This led us to conjecture that 3 generations with three untilted tori are impossible. This is indeed the case and can be proven rigorously. For this reason, we looked at the case where one of the tori is tilted.

A D6 brane stack in this background geometry is defined by 7 numbers: the number N of branes in the stack and the 6 winding numbers (n_a, m_a), $a = 1, 2, 3$, around the 6 one-cycles of the 3 tori,

$$\text{Brane stack} \sim (N, n_1, m_1, n_2, m_2, n_3, m_3)\,. \tag{253}$$

For a tilted torus, the winding number m can be half-integer in a convenient one-cycle basis, while the windings are integer for untilted tori. This means that the state space can be modeled by $N_{\text{stacks}} \times 7$ integers which specify the N_{stacks} D6 brane stacks in the model. In theory, this leads to an (countably) infinite state space. In practice, we truncate the state space by imposing an upper bound on the number of brane stacks, on the number of branes in each stack, and on the absolute value of each winding number. Note that, while it is known that the solution space is finite [18], the number of solutions is unknown. The number of states grows rapidly with the cutoff. For example, truncating by allowing at most 4 brane stacks with at most three branes per stack (which is generically needed to accommodate the standard model gauge group), and a maximum absolute value of 4 for the windings, we already find 10^{16} inequivalent states.

For the actions there are several possibilities. We want to traverse the space of possible brane stacks. One possibility is to just include "increment" actions a_{inc} and "decrement" actions a_{dec}. The former increase any of the $7n_{\text{max-stacks}}$ numbers that specify the state space by 1 unit, while decrement action decrease any of the numbers by one unit. We call the agent performing these actions the "flipping agent". This agent can perform illegal actions:

- It can decrease the number of branes in a stack below 0.
- It can increase the winding or number of branes above the truncation values we have chosen.
- It can generate non-coprime winding numbers. This is not inconsistent, but corresponds to an equivalent brane stack with coprime windings and more branes in the stack.
- It can produce equivalent brane stacks. This is also not inconsistent, but they should be treated as the same stack whose number of branes is given by the sum of branes of the individual stacks.

If the agent performs an illegal action, we punish it and do not change the state.

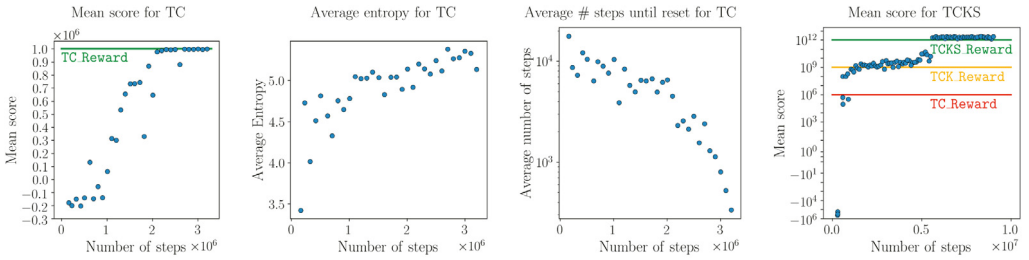
A second possibility is to generate a set of possible winding number combinations up to some maximum winding. A systematic way of constructing these branes is described in [18]. An action would then be to either increase or decrease the number of branes in each stack, or to exchange the set of six winding numbers of the branes in the stack by another set of six winding numbers from the pre-computed list. We call the agent performing these actions the "stacking agent". This agent can also perform illegal actions:

- It can decrease the number of branes in a stack below 0.
- It can assign same winding numbers to different brane stacks. This is in itself not inconsistent, but the stacks should be treated as the same stack whose number of branes is given by the sum of branes of the individual stacks.

If the agent performs an illegal action, we punish it and do not change the state.

While both the flipping and the stacking actions allow the agents to explore the state space, they lead to rather different exploration routes, i.e. the flipping and the stacking agent perceive the state space differently. In particular, states that are close for the flipping agent might be far away for the stacking agent and vice versa. It is a priori unclear which method is better suited for RL, and our study shows that there is not a big difference: the flipping agent seems to learn a bit slower, but eventually produces comparable results to the stacking agent.

Next, we need to specify the rewards for the agent. We need to choose brane stacks such that the mathematical string consistency conditions (tadpole cancellation, the K-Theory constraint, and the supersymmetry conditions) as well as the phenomenological conditions (MSSM gauge group, three families of MSSM particles, one pair of Higgs fields, no exotics) are met. The first two conditions can be written as a set of non-linear (quartic and cubic) Diophantine equations in the brane stacks, more precisely in the number of branes in the stack and the winding numbers around the three tori. Note that the SUSY constraints are actually not Diophantine, since the torus radii appear, which can take any real value. They can be formulated as Diophantine equations if one allows only rational radii (since $\mathbb{Q}$ is dense in $\mathbb{R}$ and we do not include



(a) Agent learning the TC constraint.

(b) Agent learning the TCKS constraint.

Fig. 61. We illustrate how the agent learns to satisfy string consistency conditions: For tadpole cancellation, we demonstrate that the agent learns to satisfy the constraints (first plot) and gets more efficient in finding solutions (second plot) while still exploring different states in the space as measured by the entropy (third plot). The fourth plot demonstrates the same behavior when all consistency conditions are imposed.
Source: The plots are taken from [24].

fluxes to stabilize these moduli, the moduli space will be restricted but still at codimension zero, such that we can find solutions in $\mathbb{Q}$ if we can find solutions in $\mathbb{R}$). Another technique is to use constrained quadratic programming to check whether solutions to the SUSY conditions exist. The gauge group and particle content can be computed from the number of branes in each stack and the winding numbers as well. Rewards and punishments are awarded based on how close the agent got to satisfying these constraints.

Since several constraints need to be taken into account, we are dealing with a multi-task RL problem. One approach to deal with this is to assign an order in which the constraints are checked. In doing so, computationally expensive checks (such as checking SUSY or the number of exotics), can be checked in the end once all other constraints are checked and met. This leads to an ordering in checking the constraints, and it is again unclear which ordering works best. Another approach is to train different agents on different tasks and then combine their knowledge. We have tried both approaches. We furthermore reset the agent after taking 10,000 steps or after solving all constraints.

Let us next discuss the results. In a first set of experiments, we verify that the agent can learn to solve the mathematical consistency conditions, which hints at an underlying structure in the solution space. When looking just at the coupled set of Diophantine equations that describe the tadpole cancellation (TC) condition, we find that the agent learns to solve these equations after around 2 million exploration steps, cf. Fig. 61a. Before that, the agent accumulates a negative return from punishments, either due to illegal actions, or due to its distance from an actual tadpole canceling configuration. Once it finds a state (i.e. a set of brane stacks) for which all tadpoles are canceled, it receives are large reward (of 10^6). In order to exclude that, once a tadpole canceling state has been found, the agent always keeps returning to this single state, we show in the third plot that the entropy of the states does not decrease, i.e. the agent finds different states, all of which satisfy the constraint. In the second plot, we show the average number of steps per episode. In this setup, an episode ends after 10,000 steps or once the TC condition is met. Initially, the agent does not find a TC state, and the average number of steps in an episode is 10^4. Over time, the agent gets more and more efficient in solving TC, such that in the end it only needs $\mathcal{O}(100)$ steps to arrive at a TC state. In Fig. 61b, we demonstrate how the agent learns sequentially to solve TC, the K-Theory constraint, and the SUSY conditions, for which it earns a reward of 10^6, 10^9 and 10^{12}, respectively. We also check that the entropy stays constant and that the average number of steps per episode decreases.

In a second set of experiments, we try different orderings in which the consistency conditions are checked. The brute-force approach of [216] first solves the SUSY constraints and then K-Theory and tadpole. For this ordering, a human-derived strategy has been found that uses so-called filler branes [217] to decouple the SUSY constraints from the tadpole constraints: by adding branes with a specific pattern of their winding numbers, the tadpole can be changed without modifying the SUSY constraint. In order to check whether an RL agent can learn this filler brane strategy, we impose this ordering, ask the agent to find consistent models, and record the type of branes the agent used. Indeed, we find that the agent uses mainly filler branes to satisfy the constraints.

However, checking the SUSY constraints first is not ideal since they are computationally very expensive to verify. For this reason, we studied the ordering of Fig. 61b, in which we check the tadpole first, then K-Theory and then SUSY. For this ordering, no human-derived strategy is known which partially decouples the equations, i.e. which solves SUSY without modifying the tadpole. Nevertheless, we observe that the agent learns to satisfy all constraints. Even more interestingly, it learns to do so twice as efficiently as compared to the other ordering, and $\mathcal{O}(200)$ times more efficient than a random walker.

Comparing the statistics of consistent models found by an RL agent with the ones obtained in [216], we find comparable results. This indicates that the agent explores the full solution space and does not focus on a particular corner of the landscape (other than finding consistent models more efficiently).

To summarize, we have shown that an RL agent can efficiently search the landscape of Type II orientifold string vacua. It learns human-derived strategies, but can also find new, previously unknown solution strategies. It outperforms a random walker by a factor of up to $\mathcal{O}(200)$.

Table 6
Constraints on the SUSY parameters imposed in [118].

θ	ϕ	$m_{3/2}$	$\tan\beta$	μ	m_t	M_{GUT}	M_{I}
30°–90°	0°–90°	50–1500 GeV	2–50	> 0	175 GeV	2×10^{16} GeV	5×10^{11} GeV

10.9. Finding minima with genetic algorithms

Searches of parameter spaces based on genetic algorithms (see Section 5) have been considered in [118,122,123], which we now review.

10.9.1. Searches in SUSY parameter spaces

Genetic algorithms (GAs) were first applied to SUSY models in [118] and more recently in [123]. In [118], the authors are interested in experimental signatures from SUSY breaking. They ask: given different SUSY breaking mechanisms at some high scale, which precision in the measurement at the low (electroweak) scale is needed in order to discriminate the different SUSY breaking mechanisms. In this way, they define a distance measure at the low scale at which the measurement is performed. Two models will be indistinguishable if their predictions overlap at the low scale, i.e. if their minimum distance is zero. On the other hand, if the regions do not overlap, there will be a non-zero distance, and this distance defines the experimental precision necessary to tell the two high-scale models apart.

Let us next explain why the authors use genetic algorithms rather than some other minimization (or maximization) algorithm. Every high-scale SUSY breaking mechanism comes with some set of parameters for the soft masses. In a full-fledged model of SUSY breaking, these will be fixed, but if one lacks a full explanation of how SUSY breaking is generated these can be taken as free parameters of an effective theory. However, the parameters are not completely arbitrary: using the renormalization group equations, a point in the high-scale parameter space is mapped onto a point in the low-scale space of observable parameters. If this point violates any of the currently known measurements (e.g. absence of radiative electroweak symmetry breaking), the point needs to be excluded at the high scale. So one is interested in the pre-image of the points in the low-scale space that are compatible with the measurements under the RGE flow. Unfortunately, the renormalization group is not actually a group; in particular, it does not have an inverse, so this is a difficult question. What one does instead is to sample points at the high scale, run them down, and see whether the point obtained at the low scale is consistent with all current measurements. Doing this many times, one can approximate a region in the high-scale parameter space that leads to consistent models under the RGE flow.

In addition to bounding the region in the high-scale parameter space using consistency conditions at the low scale, some upper bounds can be imposed directly on the high-scale parameters to avoid too large a fine-tuning of the Higgs potential parameters. This leads to a compact region in the high-scale parameter space. The constraints imposed by the authors in [118] are summarized in Table 6. The authors use string-inspired SUSY breaking scenarios known as the "GUT scenario", the "early unification scenario" and the "mirage scenario". In these scenarios, SUSY is broken by an F-term that is due to the dilaton S, the Kähler modulus T governing the volume of the CY, and a blowup mode B. Their relative contribution to the F-term is parametrized by two goldstino angles θ and ϕ. In addition, $m_{3/2}$ is the gravitino mass, $\tan\beta$ is the ratio of vevs of the two Higgs fields, μ is the Higgs μ parameter, m_t is the top mass, and M_{GUT} and M_{I} are the GUT scale (for the GUT scenario) and the intermediate scale (for the early unification and the mirage scenario), respectively.

The experimental bounds (as of 2004 when this paper was published) imposed by the authors at the low scale are

$$\begin{aligned} m_{\tilde{\chi}_1^0} > 45\,\text{GeV}\,, \quad m_{\tilde{\chi}_1^\pm} > 103\,\text{GeV}\,, \quad m_{h^0} > 113.5\,\text{GeV}\,, \\ -4.2 < \delta a_m \times 10^{10} < 41.3\,\text{GeV}\,. \end{aligned} \tag{254}$$

The fact that many a priori unpredictable regions in the high-scale parameter space have to be removed means that no meaningful derivative can be computed in these cases, which leads to the failure of other maximization algorithms such as gradient ascent. If one knew the boundary of the feasible high-scale parameter space well enough, this might be avoidable; however, computing enough points and running them down to the low scale is too computationally costly to be feasible. Hence the authors use GAs to solve the extremization problem.

Using the parameters in Eq. (254), the authors compute the masses M_i, $i = 1, \ldots, 25$, of the SUSY particles (neutralinos, gluinos, sneutrinos, squarks, sleptons, etc.) at the low scale for the different scenarios. Denoting the masses for two different scenarios A and B by M_i^A and M_i^B, they define the fitness function f as

$$f = \frac{M^A + M^B}{M^A - M^B} = \sqrt{\frac{\sum_i (M_i^A + M_i^B)^2}{\sum_i (M_i^A - M_i^B)^2}}\,. \tag{255}$$

Clearly, this is maximized (or its inverse is minimized) if $M^A = M^B$. If this happens for different scenarios, this means that the low-scale SUSY masses will be the same and the models cannot be distinguished.

Let us finally discuss the GA. The chromosome consists of eight genes, which are the input parameters at the high scale summarized in Table 6. For the selection process, the authors use truncated rank selection. The crossover mechanism is

whole arithmetic recombination on all genes (which the authors call intermediate crossover), and the mutation rate is 0.25 for each gene. No elitist selection is performed, the population size is set to 300, and the algorithm is ended if the fitness function does not improve over 20 generations. The authors find that the maximum fitness comparing two of the three models against each other is $\mathcal{O}(100) - \mathcal{O}(200)$. This means that the SUSY partner masses will differ on the level of $0.5 - 1\%$ for the three scenarios. Hence, if they can be measured within this range of precision, low energy observations will be able to distinguish between these three high energy breaking mechanisms.

In [123], the authors study compatibility of the pMSSM, which is a SUSY model for the SM with 19 free parameters, with experimental constraints such as $g-2$ of the muon or the Fermi-LAT excess in gamma rays from the galactic center. The procedure is similar to the one of [118]: The GUT scale parameters are constrained by low-scale physics as well as conditions imposed at the high scale (similar to Table 6). The authors use these parameters as input and compute several observables. The fitness function is taken to be the inverse of χ^2 of this fit. Hence maximizing fitness minimizes χ^2 and optimizes the fit. In this way, the parameters at the high scale that best reproduce the observables can be found.

The authors use the PIKAIA library for the GA implementation. They choose a population size of 100 and run over 300 generations. PIKAIA is set up to adjust the mutation rate dynamically (between 5×10^{-4} and 0.25 with an initial value of 5×10^{-3}) based on the fitness of the individuals. It uses 2- and 3-point crossover simultaneously. They use rank selection for the breeding selection, replace the entire population for survival selection, but implement elitism of 1, i.e. the fittest individual is carried over to the next generation. The authors show that GAs work very efficiently: they produce better fits in shorter time as compared to other algorithms such as Monte-Carlo based methods. The authors find that both $g-2$ and the Fermi-LAT observation cannot be explained within the pMSSM and that these have the largest influence on the χ^2 of the fit.

In summary, GAs are used to study low energy observables from high energy boundary conditions. Linking the two is computationally costly. GAs are found to perform better than other methods, and there are situations in which alternative methods such as gradient descent cannot be applied at all. With these techniques, the authors showed that different SUSY breaking scenarios can in principle be distinguished by measurements at low scales with a precision at the 1 percent level in [118]. In [123], it was observed that good fits can be found by minimizing χ^2 as an inverse fitness function, and that two experimental observations cannot be explained using just the pMSSM with 19 parameters.

10.9.2. Searches in free fermionic constructions

Genetic algorithms were also used to search for phenomenologically interesting vacua in free fermionic constructions of heterotic string theory [122]. Since statistics on the frequency of models in this setup are available, the authors can compare the performance of GAs against a random walker. They focus on three generation Pati–Salam models with a top Yukawa coupling and without exotics. The relative frequency of these models is $1 : 10^{10}$, but GAs find them several orders of magnitude faster, at a rate of $1 : 10^5$.

The free fermionic formulation of the heterotic string uses 20 left-moving and 44 right-moving real fermionic fields. The 44 right-moving fields pick up phases when parallel transported around non-contractible loops. Different phases will lead to different gauge groups and particle spectra. These phases are collected in the so-called "basis vector" v and they form an input to the free fermionic model (in the sense that they are not a priori fixed). In addition to these phases, one can choose different generalized GSO (GGSO) projections

$$c\begin{bmatrix} v_i \\ v_j \end{bmatrix} = \pm 1 \tag{256}$$

These phases form the second part of the input of free fermionic models. They act on the spectrum and modify it accordingly. The basis vectors and boundary conditions need to be chosen such that they are compatible with modular invariance of the one-loop string partition function. In practice, the authors start with a set of basis vectors that gives an SO(10) GUT group and modify it to break to Pati–Salam,

$$\mathrm{SO}(10) \longrightarrow \mathrm{SO}(6) \times \mathrm{SO}(4) \simeq \mathrm{SU}(4) \times \mathrm{SU}(2)_L \times \mathrm{SU}(2)_R\,. \tag{257}$$

With this choice of basis vectors, one has 51 possibilities for the GGSO phases, leading to $2^{51} \approx 10^{15}$ possibilities, which is a very large search space.

The authors impose the 5 constraints

1. Three generations
2. At least one Higgs pair to break the Pati–Salam group
3. At least one SM Higgs pair
4. No fractionally charged exotics
5. Existence of top Yukawa coupling

The authors assign to each of these five constraints a desired value μ_i and an actual value ν_i, $i = 1, \ldots, 5$. These are taken to be integers: For constraints that can simply be counted, such as the number of generations (i.e. for constraint 1), $\mu_1 = 3$ and ν_1 equals the actual number of generations. In the case of the top Yukawa coupling (condition 5), which is binary, $\mu_5 = 1$ and ν_5 is one or zero, depending on whether or not the top Yukawa coupling is present. Note that this is

similar to what was done for the rewards in the RL search of Type II orientifold models. The overall fitness function then is just the weighted squared difference between the target and the actual values,

$$f(\mu_i, \nu_i) = 10 - \sum_{i=1}^{5} \gamma^i (\mu_i - \nu_i)^2 \,, \tag{258}$$

where γ^i weighs the relative importance of the different constraints. The authors find that the performance of the GA is not very sensitive to the precise form of the fitness function f. The parameters have been chosen such that the maximum fitness is 10.

The authors perform different experiments and find that a population size of 50 works very well. For the mutation rate, they find a value of approximately 1%. They use roulette wheel selection with scaled fitness function and no elitist selection. The chromosome of the problem has 51 binary alleles, encoding the 51 choices for the GGSO phases. For the crossover procedure, they use 2-point crossover, which effectively swaps the middle part of the two parents.

To summarize, the authors find that GAs can improve the search for free fermionic models by several orders of magnitude as compared to a random walker. The result is not very sensitive to the precise form of the fitness function but rather sensitive to the mutation probability. Changing it from 1% to 2% makes the algorithm perform one order of magnitude worse. This trend roughly continues for changes from 2% to 3% and from 3% to 4%.

10.9.3. Searches for de Sitter and slow-roll in type IIB with non-geometric fluxes

In [120], GAs were used to find fluxes in Type IIB string theory that lead to stable vacua with a positive cosmological constant. In [121], the authors applied the same techniques to search for slow-roll inflation. The string compactification model is based on Type IIB string theory compactified on an orientifold of the $T^6/(\mathbb{Z}_2 \times \mathbb{Z}_2)$ orbifold, similar to the study in Section 10.8. However, the focus of these studies is entirely on cosmology, while the focus of the study of Section 10.8 was entirely on particle physics aspects.

The authors of [120,121] also twist the tori (not to be confused with the tilting of Section 10.8), which allows them to include more generic, non-geometric fluxes. There is an entire Physics Report on non-geometric backgrounds [218], so we will not review them here. In short, in addition to the standard F and H fluxes, this allows the authors to also include non-geometric Q fluxes at the cost of writing SL(2, $\mathbb{Z}$) covariant rather than SL(2, $\mathbb{Z}$) invariant actions. In total, there are 14 flux parameters that can be chosen. A random search would not be applicable, since the fraction of fluxes that lead to a stable de Sitter vacuum is expected to be tiny (or even completely absent according to the latest de Sitter swampland conjecture [219]).

The authors choose a population size of 16 and run the algorithm for 50,000 iterations. The alleles are the 14 flux quanta and the mutation probability is 2:3. If one of the flux quanta is chosen for mutation, a random number drawn from a normal distribution with zero mean and variance $\sigma = 0.01$ is added to the flux. As a selection process, the authors just take the 16 fittest individuals over to the next generation, which could lead to extreme elitism (carrying all individuals over). To somewhat balance that, they combine this with age-based survival and remove individuals after 2000 generations. If the algorithm is stuck for 500 iterations (i.e. all individuals have not changed for 500 generations), the standard deviation of the mutation rate is increased by 50%.

The authors do not use any crossover (cloning), only mutation. For the fitness function, the authors compute the energy scale and the minimum eigenvalue of the Hessian,

$$\tilde{\gamma} = \frac{|DW|}{3|W|^2} \,, \qquad \eta = \min \text{eigenval}(m^2_{IJ}) \,, \tag{259}$$

where D is the Kähler-covariant derivative. A stable de Sitter solution has $\tilde{\gamma} > 1$ and $\eta > 0$. The quantities $\tilde{\gamma}$ and η are computed for all parents and their cloned, mutated children. The fitness is then proportional to how close the individuals are getting to such a solution. With this procedure, the authors find 3 sets of fluxes that lead to stable de Sitter solutions.

In [121], the authors search for flux quanta (this time restricting to geometric fluxes) that allow for slow roll inflation, i.e. they search for an inflaton potential for which the slow roll parameters are small,

$$\epsilon = \frac{3}{2} \frac{\dot{\phi}^2}{V + \dot{\phi}^2} \ll 1 \,, \qquad \eta = -\frac{\ddot{\phi}}{H\dot{\phi}} \ll 1 \,, \tag{260}$$

where ϕ is the inflaton field, V is the inflaton potential, and H is the Hubble parameter, $H = \frac{\dot{a}}{a}$ for scale factor a. This can be cast in terms of flatness of the potential,

$$\epsilon_V = \frac{1}{2} \frac{K^{IJ} D_I V D_J V}{V^2} \,, \qquad |\eta_V| = \left| \min \text{eigenval} \left(\frac{K^{JK} D_I D_K V}{V} \right) \right| \,, \tag{261}$$

which, for the solutions found in [121] is of the order of 0.1. In (261), K_{IJ} is the standard Kähler potential on toroidal orbifolds,

$$K = -\log(-i(S + \overline{S})) - \sum_{\alpha=1}^{3} \log(-i(T_\alpha + \overline{T}_\alpha)) - \sum_{\alpha=1}^{3} \log(-i(U_\alpha + \overline{U}_\alpha)) \,, \tag{262}$$

for the dilaton S, the three Kähler moduli T_α and the three complex structure parameters U_α, expressed in the 14 real fields

$$S = \chi + ie^{-\phi}\,, \quad T_\alpha = \chi_\alpha^{(1)} + ie^{-\phi_\alpha^{(1)}}\,, \quad U_\alpha = \chi_\alpha^{(2)} + ie^{-\phi_\alpha^{(2)}}\,, \tag{263}$$

i.e. $\alpha = 1, 2, 3$ and $I, J = 1, \ldots, 14$. Finally, V is the standard scalar potential

$$V = e^K(-3|W|^2 + K^{\alpha\bar\beta} D_\alpha W D_{\bar\beta}\overline{W})\,, \tag{264}$$

where the Gukov–Vafa–Witten superpotential W is induced by the fluxes and can be written as a polynomial of degree three in the complex structure moduli U_α.

To summarize, genetic algorithms were used in [120,121] to find (respectively non-geometric and geometric) fluxes that lead to stable de Sitter vacua and to slow roll inflation. As in the free fermionic case, such solutions are very rare, which makes random searches or brute force approaches unfeasible.

10.10. Volume-minimizing Sasaki–Einstein

As briefly mentioned in Section 10.4, non-compact CY n-folds X_n can be described as the total space of an anti-canonical bundle over a toric $(n-1)$-fold base T_{n-1}. These can in turn be thought of as a real cone over a compact, real $2n-1$-dimensional Sasaki–Einstein manifold Y_{2n-1}. The authors of [2] consider the case $n = 3$. These correspond to 4D $\mathcal{N} = 1$ theories on the world volume of a stack of D3 branes probing a CY threefold. The corresponding underlying toric 2-fold is then characterized by toric diagrams in 2D.

The authors use NNs to learn a connection between the toric data defining the non-compact CY threefold X_3 and the minimum volume of the underlying Sasaki–Einstein five-fold Y_5. This relationship is of great interest, since by the AdS/CFT correspondence [220], this is related to a-charge maximization and thus to the central charge of the 4D $\mathcal{N} = 1$ superconformal field theory. The paper [2] is among the first to apply machine learning to string theory problems. To be more precise, the authors use a combination of convolutional and fully connected layers to predict the minimum volume.

The Kähler metric on X_3 is given by

$$ds^2(X_3) = dr^2 + r^2 ds^2(Y_5)\,, \tag{265}$$

with $Y_5 = X_3|_{r=1}$. With this, the volume of Y can be defined as

$$\mathrm{vol}(Y_5) = \int_{r\leq 1} \omega^3\,, \tag{266}$$

where the Kähler form ω is

$$\omega = \frac{1}{2} i\partial\overline{\partial} r^2\,. \tag{267}$$

The authors use supervised machine learning where the input is toric data and the output is the inverse of the minimum volume. The authors use two different types of features: the first part of the input is a three-dimensional feature vector

$$f = \{I, E, V\}\,, \tag{268}$$

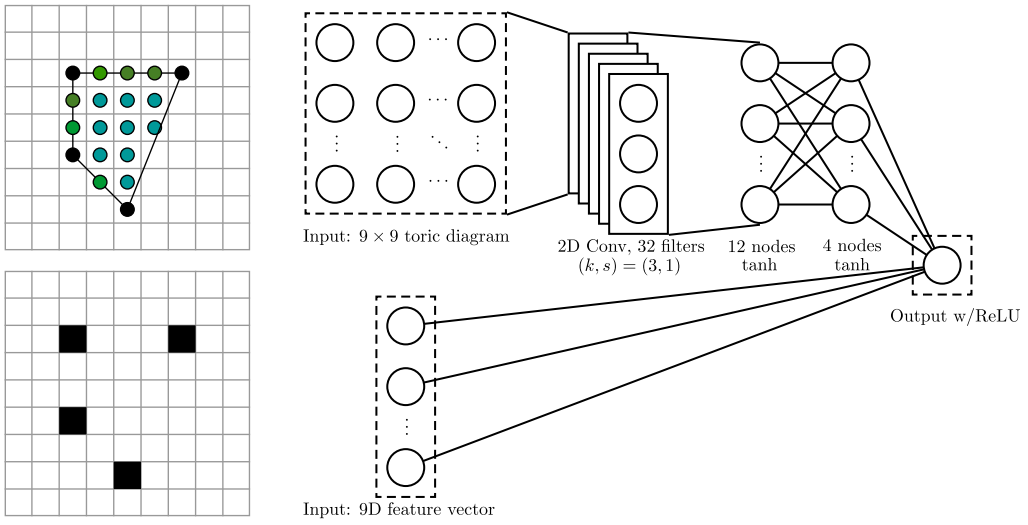
where I is the number of internal lattice points, E is the number of edges, and V is the number of vertices of the toric diagram. The second part of the input features is a "black and white image" of the toric diagram, i.e. a 9×9 square matrix with zeros everywhere except for the position of the vertices, which are assigned a one (see Fig. 62a for one example). The relevant set of toric diagrams contains 15,151 inequivalent lattice polytopes. The authors mod out $\mathrm{GL}(2, \mathbb{Z})$ rotations to ensure that equivalent toric diagrams are only included once and that they fit into the 9×9 square. Many polytopes will be smaller. Their toric diagram input is padded with zeros and positioned around the center of the "image". In the data set there are 4 special cases which are removed, leaving 15,147 lattice polytopes. The corresponding volume is computed and rounded to four digits, leaving 797 distinct volumes.

As a first step, the authors do not use the "image" of the toric diagram at all. Moreover, they just use a simple feed-forward NN without hidden layers and with identity activation at the output layer, i.e. they perform linear regression on the input features. Instead of just using the three input features (268), the authors perform feature engineering and add the monomials containing the product of 2 of the inputs $\{I, E, V\}$ as well, i.e. they use

$$x^{(0)} = \{I, E, V, IE, IV, EV, I^2, E^2, V^2\}\,. \tag{269}$$

With this, the NN is trained. The authors use Keras with a Theano backend. They perform a train:test split of 75:25, use the ADAM optimizer and train for 5000 epochs with batch size 1000. This simple NN achieves a correct prediction rate of 97.8%. It is observed that without adding the degree 2 monomials, i.e. performing linear regression on just the three input vectors leads to much worse results, while adding degree three monomials to the input does not further improve the accuracy.

Next, the authors combine this NN with a CNN. So in addition to the one-layer linear regression, they use a convolutional layer with two subsequent fully connected layers. The output of the two NNs is combined to form a single



(a) Example encoding. (b) NN architecture.

Fig. 62. NN architecture used to learn the Sasaki–Einstein volume and an example for the input toric diagram encoding (with $\{I, E, V\} = \{9, 8, 4\}$).

output (corresponding to the inverse volume), which is activated with a ReLU activation function (the volume is positive). The NN architecture is illustrated in Fig. 62b. The authors use one 2D convolutional layer with 32 filters of size 3×3 and identity activation. The two following fully connected layers have 12 and 4 nodes and a tanh activation function. The authors find that the result is not too sensitive to the NN architecture, but fewer hidden layers with tanh activation seem to work best. Using the same training parameters as above, this NN architecture has above 99% accuracy.

The results obtained in this paper are very interesting. While it is clear that the volume of the Sasaki–Einstein manifold (and thus the process of a-maximization) has to be encoded in the toric data, its direct relation with the features used as inputs is not known. The work hints at such a direct relation and enables to bypass the minimization computation with a very high accuracy.

10.11. Deep learning Ads/CFT and holography

In [221,222], the authors study the AdS/CFT correspondence with NNs and apply it to QCD. The authors use a NN to describe the bulk geometry, where the hidden layers discretize the radial direction of AdS. With this setup, the weights of the NN correspond to the metric components of the AdS bulk. The finite temperature CFT lives at the boundary of the AdS space, with a black hole (BH) horizon sitting at the other end. The authors study a scalar ϕ^4 theory on this curved AdS background.

They consider a scalar field in curved $(D+1)$-dimensional AdS,

$$S = \int dx^{D+1} \sqrt{-\det g} \left[-\frac{1}{2} g^{\mu\nu} \partial_\mu \phi \partial_\nu \phi - \frac{1}{2} m^2 \phi^2 - V(\phi) \right] , \tag{270}$$

with metric

$$ds^2 = -f(\eta) dt^2 + d\eta^2 + g(\eta)(dx_1^2 + \cdots + dx_{D-1}^2) . \tag{271}$$

For simplicity, the authors restrict ϕ to depend on the radial direction η only. The two boundaries are at

$$\begin{aligned} &\text{CFT boundary:} \quad f \approx g \approx e^{2\eta/L} && (\eta \to \infty) , \\ &\text{BH boundary:} \quad f \approx \eta^2 , \quad g \approx \text{const.} && (\eta \to 0) . \end{aligned} \tag{272}$$

Introducing $\pi := \partial_\eta \phi$, the equations of motion (EOM) for ϕ can be written as

$$\partial_\eta \pi + h(\eta)\pi - m^2 \phi - \frac{\delta V(\phi)}{\delta \phi} = 0 , \tag{273}$$

with $h(\eta) := \partial_\eta \log \sqrt{f(\eta) g(\eta)^{D-1}}$. By discretizing the η direction into N steps $\Delta\eta$, the EOM can be written as

$$\begin{aligned} \phi(\eta + \Delta\eta) &= \phi(\eta) + \Delta\eta \pi(\eta) , \\ \pi(\eta + \Delta\eta) &= \pi(\eta) - \Delta\eta \left(h(\eta)\pi(\eta) - m^2 \phi(\eta) - \frac{\delta V(\phi)}{\delta \phi} \right) . \end{aligned} \tag{274}$$

Table 7
Dictionary between AdS/CFT and a deep Boltzmann machine [223].

AdS/CFT		DBM
Discretized bulk coordinate z	$\leftrightarrow$	Hidden layers
QFT source term J	$\leftrightarrow$	Input $x^{(0)}$
Bulk field $\phi(x,z)$	$\leftrightarrow$	Hidden states $\ell^{(i)}$
QFT generating function $Z[J]$	$\leftrightarrow$	Probability distribution $P(\ell^{(0)})$
Bulk action	$\leftrightarrow$	Energy function $E(\ell^{(0)}, \ell^{(i)})$

The input data $(\phi(\infty), \pi(\infty))$ is fixed at the CFT boundary, and the output of the NN $(\phi(0), \pi(0))$ is at the BH horizon. The output data is fixed by demanding

$$0 \stackrel{!}{=} \left[\frac{2}{\eta}\pi - m^2\phi - \frac{\delta V(\phi)}{\delta\phi}\right]_{\eta=0}, \tag{275}$$

which is imposed on the final pair $(\phi(0), \pi(0))$.

The discretized process can be modeled with a NN with two input nodes $(\phi(\infty), \pi(\infty))$, N hidden layers $(\phi(\eta^i), \pi(\eta^i))$, and two output nodes $(\phi(0), \pi(0))$. The weights are then

$$w^{(i)} = \begin{pmatrix} 1 & \Delta\eta \\ \Delta\eta\, m^2 & 1 - \Delta\eta\, h(\eta^i) \end{pmatrix}, \tag{276}$$

the biases are trivial, and the activation functions f_1 and f_2 of the two nodes of each layer are given by

$$f_1(\ell_1^{(i)}) = \ell_1^{(i)}, \qquad f_2(\ell_2^{(i)}) = \ell_2^{(i)} + \Delta\eta \frac{\delta V(\ell_1^{(i)})}{\delta \ell_1^{(i)}}. \tag{277}$$

As a first test case, the authors study an AdS Schwarzschild BH in 3 dimensions. The physical parameters are chosen as $m^2 = -1$ and $V(\phi) = \frac{1}{4}\phi^4$. In this case, the metric is known analytically,

$$h(\eta) = 3\coth(3\eta). \tag{278}$$

The NN used to learn this metric has 10 hidden layers and the two boundaries are placed at $\eta_0 = 1$ and $\eta_{11} = 0.1$. The authors use the PyTorch library with mini-batch size 10, 100 epochs, L_1 norm error and weight regularization. They find that the majority of the bulk metric is learned extremely well, but that there is a 30% systematic error at the horizon.

In a next step, the authors apply the same method to experimental data. The goal is to learn $h(\eta)$, which is mapped to the magnetization curve of the material $Sm_{0.6}Sr_{0.4}MnO_3$ at a temperature of 155 K. The initial conditions for ϕ are set to a linear combination of the magnetization M and the external magnetic field H. It is found that the magnetization curve $h(\eta)$ nicely reproduces the measured magnetization curve.

Note that this work is quite different from usual NN applications in that the authors are actually not interested in the output of the NN but in its weights. These correspond to the bulk metric and thus to quantities that can be measured in an experiment. The same technique is also applied to AdS/QCD in [222]. In that case, the input was lattice QCD data of the chiral quark condensate $\langle q\bar{q}\rangle$ as a function of the quark mass m_q. The learned metric has a wall and a horizon. With the learned metric, the authors can compute the quark–antiquark potential and find it to have both a linear (Debye) and a screening part. It is remarkable that the metric has both features of a confining and a deconfining phase simultaneously.

10.12. Boltzmann machines

10.12.1. Boltzmann machines and AdS/CFT

In [223], the authors cast AdS/CFT [220] in the language of a deep Boltzmann machine. The relation between the bulk gravity and the boundary QFT is

$$Z_{\text{QFT}}[J] = \int_{\phi(x,z)|_{z=0}} D\phi e^{-S_{\text{gravity}}[\phi]}, \tag{279}$$

where $z = 0$ is the boundary of the AdS bulk, $\phi = \phi(x,z)$ is a bulk field, and J is a source term on the QFT boundary. By comparing the quantities of a deep Boltzmann machine (DBM) with the physical quantities of the AdS/CFT correspondence (279), one can establish a "dictionary" between the two, as summarized in Table 7.

With this dictionary, the authors discuss the simplest case of a free massive scalar in an AdS_{D+1} bulk geometry,

$$S = \int d^D x dz \frac{1}{2}\left[a(z)(\partial_z\phi)^2 + b(z)\sum_{I=1}^{D-1}(\partial_I\phi)^2 + d(z)(\partial_\tau\phi)^2 + c(z)m^2\phi^2\right], \tag{280}$$

where τ is the Euclideanized time direction and a, b, c, d are functions given in terms of the metric and satisfy

$$a \sim b \sim c \sim (L/z)^{D-1}, \qquad c \sim (L/z)^{D+1} \tag{281}$$

near the AdS boundary $z \sim 0$, where L is the AdS radius. The bulk direction is discretized,

$$z^k = k\Delta z\,, \tag{282}$$

and similarly for the other coordinates x^I and τ,

$$x_{I,i} = i\Delta x\,, \qquad x_D = \tau_l = l\Delta\tau\,. \tag{283}$$

This discretizes also the metric (and thus the functions a, b, c, d) and the field ϕ, which is now defined at lattice points in the discretized spacetime,

$$\phi(x_{i,l}, z_k) := \ell^{(k)}(x_{i,l}, z_k) := \ell^{(k)}_{i,l}\,. \tag{284}$$

Lastly, in the action (280), differentiation is turned into finite differences and integration is turned into summation,

$$\begin{aligned} S = \sum_{i,k,l} \Big[& a_k \frac{1}{2(\Delta z)^2}(\ell^{(k+1)}_{i,l} - \ell^{(k)}_{i,l})^2 + c_k \frac{m^2}{2}(\ell^{(k)}_{i,l}) \\ & + b_k \frac{1}{2(\Delta x)^2}(\ell^{(k+1)}_{i+1,l} - \ell^{(k+1)}_{i,l})(\ell^{(k)}_{i+1,l} - \ell^{(k)}_{i,l}) \\ & + d_k \frac{1}{2(\Delta \tau)^2}(\ell^{(k+1)}_{i,l+1} - \ell^{(k+1)}_{i,l})(\ell^{(k)}_{i,l+1} - \ell^{(k)}_{i,l}) \Big] \end{aligned} \tag{285}$$

This corresponds to a DBM with

$$S = E = \sum_k \left[\sum_{i,j} \sum_{l,m} w^{(k)}_{ij,lm} \ell^{(k)}_{i,l} \ell^{(k+1)}_{j,m} + \widetilde{w}^{(k)}_{ij,lm} \ell^{(k)}_{i,l} \ell^{(k)}_{j,m} \right], \tag{286}$$

where the weights are given by

$$\begin{aligned} w^{(k)}_{ij,lm} = & -\frac{a_k}{(\Delta z)^2}\delta^j_i \delta^m_l + \frac{b_k}{(\Delta x)^2}(2\delta^j_i\delta^m_l - \delta^j_{i+1}\delta^m_l - \delta^{j+1}_i\delta^m_l) \\ & + \frac{d_k}{(\Delta x)^2}(2\delta^j_i\delta^m_l - \delta^j_i\delta^m_{l+1} - \delta^j_l\delta^{m+1}_l)\,, \\ \widetilde{w}^{(k)}_{ij,lm} = & \left(\frac{a_k + a_{k+1}}{2(\Delta z)^2} + m^2\frac{c_k}{2}\right)\delta^j_i\delta^m_l\,. \end{aligned} \tag{287}$$

To summarize, the authors explained how to map AdS/CFT to a DMB. In this map, the first layer corresponds to the CFT boundary, the last layer to the black hole horizon, and the weights to the bulk metric. The hidden states encode discretized scalar bulk fields and the probability distribution is the generating functional of the QFT theory on the boundary.

10.12.2. Boltzmann machines and the Riemann theta function

In [224,225], the authors introduce Boltzmann machines whose activation functions are Riemann Theta functions. While the paper itself is not related to string theory, Riemann Theta functions do feature prominently in string partition functions, which is why we want to briefly describe them.

The authors use a non-restricted Boltzmann machine (BM), i.e. they allow for connections among nodes within the same layer. They consider a BM with N_v visible layer nodes and N_h hidden layer nodes. The connection between the nodes can then be described by the block-matrix

$$A = \begin{pmatrix} Q & W^T \\ W & T \end{pmatrix}, \tag{288}$$

where T is an $N_v \times N_v$ matrix modeling the connection of the nodes of the visible layer among each other, W is the usual $N_v \times N_h$ weight matrix connecting the visible and the hidden layers, and Q is an $N_h \times N_h$ matrix modeling the connection of the hidden layer nodes. The BM is designed to have continuous input parameters but discrete hidden parameters. The matrix W is restricted to be real and A to be positive definite. Combining the hidden and visible state vector into a single vector $x = (h \;\; v)^T$, the energy of the Boltzmann machine can be written as

$$E(v,h) = \frac{1}{2}x^T \cdot A \cdot x + B^T \cdot x\,, \tag{289}$$

where B is the bias vector $B = (B_h \;\; B_v)^T$. Since A is taken to be positive definite, the energy is positive for large enough x. The partition function of the BM is then

$$Z = \int_{-\infty}^{\infty} [dv] \sum_{[h]} e^{-E(v,h)}\,, \tag{290}$$

where $[dv]$ is the measure $dv_1 \dots dv_{N_v}$ and $[h] = \{h_1, \dots, h_{N_h}\}$. Now, the point is that in this setup, the free energy

$$F = -\log \sum_{[h]} e^{-E(v,h)} \tag{291}$$

can be written in terms of a (rescaled) Riemann Theta function,

$$\theta(z|\Omega) = \sum_{n\in\mathbb{Z}^g} e^{2\pi i(\frac{1}{2}n^T\cdot\Omega\cdot n + n^T\cdot z)}\,, \quad \tilde{\theta}(z|\Omega) = \theta\left(\frac{z}{2\pi i}\Big| -\frac{\Omega}{2\pi i}\right)\,, \tag{292}$$

as

$$F(v) = \frac{1}{2}v^T\cdot T\cdot v + B_v^T\cdot v - \log\tilde{\theta}(v^T\cdot W + B_h^T|Q)\,. \tag{293}$$

With this, the probability for the BM to be in a specific state is given by the Boltzmann distribution

$$P(v,h) = \frac{e^{-E(v,h)}}{Z}\,. \tag{294}$$

The probability distribution for the visible units is then

$$P(v) = \frac{e^{-F(v)}}{Z}\,, \tag{295}$$

which can be given analytically and in closed form using (293) and (290), and involves quotients of Riemann Theta functions. Similarly, a closed form expression for the conditional probabilities of the hidden unit values can be given,

$$P(h|v) = \frac{P(h,v)}{P(v)} = \frac{e^{-\frac{1}{2}h^T\cdot Q\cdot h - (v^T\cdot W + B_h^T)\cdot h}}{\tilde{\theta}(v^T\cdot W + B_h^T|Q)}\,. \tag{296}$$

With these explicit expressions, the BM can be trained directly, and one does not need to use contrastive divergence. However, since training with gradient descent requires evaluation of derivatives of θ-functions, which is rather costly, the authors train the BM with a genetic algorithm. The authors also demonstrate the use of the BM with θ function activation in NNs to approximate probability distributions, continuous time series, and for classification.

To summarize, the authors consider BM with continuous visible and discrete hidden states. Under mild assumptions concerning the connections of nodes within a layer, they can solve for the probability distribution of the BM analytically and express it in terms of ratios of Riemann Theta functions. They then discuss several applications in NNs with good results.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

I thank Stephan Stieberger and Physics Reports for inviting me to write this review. I also thank Andre Lukas for his support and encouragement to pursue this direction. I am also greatly indebted to Callum Brodie for proofreading and providing comments on the manuscript.

I am grateful for many inspiring conversations with Koji Hashimoto, Sven Krippendorf, Andre Lukas, Brent Nelson, Rak Seong, Gary Shiu and Greg Yang. I want to thank especially Jim Halverson for many interesting discussions and fruitful collaborations.

I also thank the EPSRC, UK network grant EP/N007158/1 for support during the time I started to work on machine learning, and the ICTP Trieste, Microsoft Research, the BCTP at Bonn University, and the Simons Center for Geometry and Physics, USA for financial support for schools and conferences we organized on machine learning. I am furthermore thankful for support and hospitality by the Banff International Research Station and the Casa Matematica Oaxaca, Universidad Nacional Autónoma de México, University of Pennsylvania, Northeastern University, and the Aspen Center for Physics, where part of this work was carried out.

References

[1] Y.-H. He, Deep-Learning the Landscape, 1706.02714.
[2] D. Krefl, R.-K. Seong, Machine learning of Calabi-Yau volumes, Phys. Rev. D 96 (6) (2017) 066014, 1706.03346.
[3] F. Ruehle, Evolving neural networks with genetic algorithms to study the String Landscape, J. High Energy Phys. 08 (2017) 038, 1706.07024.
[4] J. Carifio, J. Halverson, D. Krioukov, B.D. Nelson, Machine learning in the string landscape, J. High Energy Phys. 09 (2017) 157, 1707.00655.

[5] M. Green, J. Schwarz, E. Witten, Superstring Theory Vol. 1: Introduction, in: Cambridge Monographs On Mathematical Physics, Univ. Pr, Cambridge, Uk, 1987, p. 469.
[6] M.B. Green, J.H. Schwarz, E. Witten, Superstring Theory Vol. 2: Loop Amplitudes, Anomalies and Phenomenology, in: Cambridge Monographs On Mathematical Physics, Univ. Pr, Cambridge, Uk, 1987, p. 596.
[7] L.E. Ibanez, A.M. Uranga, String Theory and Particle Physics: An Introduction to String Phenomenology, Cambridge University Press, 2012.
[8] R. Blumenhagen, D. Lüst, S. Theisen, Basic Concepts of String Theory. Theoretical and Mathematical Physics, Springer Heidelberg, Germany, 2013.
[9] E. Witten, String theory dynamics in various dimensions, Nuclear Phys. B 443 (1995) 85–126, hep-th/9503124. [, 333(1995)].
[10] C. Vafa, Evidence for f theory, Nuclear Phys. B 469 (1996) 403–418, hep-th/9602022.
[11] P. Candelas, G.T. Horowitz, A. Strominger, E. Witten, Vacuum configurations for superstrings, Nuclear Phys. B 258 (1985) 46–74.
[12] M.R. Douglas, The statistics of string / M theory vacua, J. High Energy Phys. 05 (2003) 046, hep-th/0303194.
[13] C. Vafa, The String landscape and the swampland, hep-th/0509212.
[14] S. Ashok, M.R. Douglas, Counting flux vacua, J. High Energy Phys. 01 (2004) 060, hep-th/0307049.
[15] J. Halverson, C. Long, B. Sung, Algorithmic universality in F-theory compactifications, Phys. Rev. D 96 (12) (2017) 126006, 1706.02299.
[16] W. Taylor, Y.-N. Wang, The F-theory geometry with most flux vacua, J. High Energy Phys. 12 (2015) 164, 1511.03209.
[17] D. Coppersmith, S. Winograd, Matrix multiplication via arithmetic progressions, J. Symbolic Comput. 9 (3) (1990) 251–280, Computational algebraic complexity editorial.
[18] M.R. Douglas, W. Taylor, The landscape of intersecting brane models, J. High Energy Phys. 01 (2007) 031, hep-th/0606109.
[19] J. Halverson, F. Ruehle, Computational complexity of vacua and near-vacua in field and string theory, Phys. Rev. D 99 (4) (2019) 046015, 1809.08279.
[20] E.W. Mayr, H.J. Prömel, A. Steger (Eds.), Lectures on Proof Verification and Approximation Algorithms, in: Lecture Notes in Computer Science, vol. 1367, Springer, 1998.
[21] C.L. Siegel, Zur theorie der quadratischen formen, Nachr. Akad. Wiss. Göttingen Math.-Phys. Kl II (1972) 21–46.
[22] K. Manders, L. Adleman, Np-complete decision problems for quadratic polynomials, in: Proceedings of the Eighth Annual ACM Symposium on Theory of Computing STOC '76, ACM New York, NY, USA, 1976, pp. 23–29.
[23] A.K. Lenstra, H.W. Lenstra, L, Lovasz factoring polynomials with rational coefficients, Math. Ann. 261 (1982) 515–534.
[24] J. Halverson, B. Nelson, F. Ruehle, Branes with brains: Exploring string vacua with deep reinforcement learning, J. High Energy Phys. 06 (2019) 003, 1903.11616.
[25] W.R. Inc, Mathematica. Champaign, IL, 2019.
[26] Fabian Ruehle, Github page with material for this review, URL http://github.com/ruehlef/Physics-Reports.
[27] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer, Automatic differentiation in pytorch, in: NIPS Autodiff Workshop, 2017.
[28] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, Tensorflow: Large-scale machine learning on heterogeneous systems, 2015, Software available from tensorflow.org.
[29] F. Chollet, et al., Keras, 2015.
[30] S. Tokui, K. Oono, S. Hido, J. Clayton, Chainer: a next-generation open source framework for deep learning, in: Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS). 2015.
[31] ChainerRL..
[32] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym, 2016.
[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, E. Duchesnay, Scikit-learn: Machine learning in Python, J. Mach. Learn. Res. 12 (2011) 2825–2830.
[34] F.-M. De Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, C. Gagné, Deap: A python framework for evolutionary algorithms, in: Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation GECCO '12, ACM New York, NY, USA, 2012, pp. 85–92, Software availabe at http://deap.readthedocs.io/.
[35] P. Charbonneau, Release notes for PIKAIA 1.2, 2002.
[36] J. Williams, Pikaia, FORTRAN90.
[37] N. Otter, M.A. Porter, U. Tillmann, P. Grindrod, H.A. Harrington, A roadmap for the computation of persistent homology, EPJ Data Sci. 6 (2017) 17.
[38] A. Tausz, M. Vejdemo-Johansson, H. Adams, JavaPlex: A research software package for persistent (co)homology, in: H. Hong, C. Yap (Eds.), Proceedings of ICMS 2014, in: Lecture Notes in Computer Science, vol. 8592, 2014, pp. 129–136, Software available at http://appliedtopology.github.io/javaplex/.
[39] U. Bauer, Ripser: a lean c++ code for the computation of vietoris-rips persistence barcodes, 2017.
[40] The GUDHI Project, GUDHI User and Reference Manual, GUDHI Editorial Board, 2015.
[41] J. Lee, Y. Bahri, R. Novak, S.S. Schoenholz, J. Pennington, J. Sohl-Dickstein, Deep neural networks as Gaussian processes, 2017, arXiv e-prints, arXiv:1711.00165 1711.00165.
[42] L. Zhang, G. Naitzat, L.-H. Lim, Tropical geometry of deep neural networks, in: J. Dy, A. Krause (Eds.), Proceedings of the 35th International Conference on Machine Learning, in: Proceedings of Machine Learning Research, vol. 80, PMLR Stockholmsmässan, Stockholm Sweden, 2018, pp. 5824–5832, 1805.07091.
[43] J. Su, D. Vasconcellos Vargas, S. Kouichi, One pixel attack for fooling deep neural networks, 2017, arXiv e-prints, arXiv:1710.08864 1710.08864.
[44] I.J. Goodfellow, J. Shlens, C. Szegedy, Explaining and harnessing adversarial examples, arxiv e-prints, 2014, arXiv:1412.6572 14126572.
[45] M.D. Zeiler, R. Fergus, Visualizing and understanding convolutional networks, in: D. Fleet, T. Pajdla, B. Schiele, T. Tuytelaars (Eds.), Computer Vision – ECCV 2014, Springer International Publishing, Cham, 2014, pp. 818–833.
[46] D. Masters, C. Luschi, Revisiting small batch training for deep neural networks, 1804.07612.
[47] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, Smote: Synthetic minority over-sampling technique, J. Artif. Int. Res. 16 (2002) 321–357, 1106.1813.
[48] M. Kreuzer, H. Skarke, PALP: A package for analyzing lattice polytopes with applications to toric geometry, Comput. Phys. Comm. 157 (2004) 87–106, math/0204356.
[49] R. Grinis, A. Kasprzyk, Normal forms of convex lattice polytopes, arxiv e-prints, 2013, arXiv:1301.6641 1301.6641.
[50] K.G. Murty, S.N. Kabadi, Some np-complete problems in quadratic and nonlinear programming, Math. Program. 39 (1987) 117–129.
[51] A. Anandkumar, R. Ge, Efficient approaches for escaping higher order saddle points in non-convex optimization, 2016, arXiv e-prints arXiv:1602.05908 1602.05908.
[52] J. Nie, Optimality conditions and finite convergence of lasserre's hierarchy, 2012, arXiv e-prints arXiv:1206.0319, 1206.0319.

[53] D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning representations by back-propagating errors, in: J.A. Anderson, E. Rosenfeld (Eds.), Neurocomputing: Foundations of Research, MIT Press Cambridge, MA, USA, 1988, pp. 696–699, ch. Learning Representations by Back-propagating Errors.
[54] Y. LeCun, Une procédure d'apprentissage pour réseau à seuil asymétrique, in: Proceedings of Cognitiva, Vol. 85, Paris, 1985, pp. 599–604.
[55] D.B. Parker, Learning-Logic, Tech. Rep. TR-47, Center for Comp. Research in Economics and Management Sci. MIT, 1985.
[56] J. Schmidhuber, Deep learning in neural networks: An overview, Neural Netw. 61 (2015) 85–117, Published online 2014; based on TR arXiv:14047828 [cs.NE].
[57] S. Ruder, An overview of gradient descent optimization algorithms, 1609.04747.
[58] R. Ge, F. Huang, C. Jin, Y. Yuan, Escaping from saddle points — Online stochastic gradient for tensor decomposition, 2015, arXiv e-prints arXiv:1503.02101 1503.02101.
[59] L. Sagun, V. Ugur Guney, G. Ben Arous, Y. LeCun, Explorations on high dimensional landscapes, 2014, arXiv e-prints arXiv:1412.6615 1412.6615.
[60] Z. Yao, A. Gholami, Q. Lei, K. Keutzer, M.W. Mahoney, Hessian-based analysis of large batch training and robustness to adversaries, in: S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett (Eds.), Advances in Neural Information Processing Systems 31, Curran Associates, Inc., 2018, pp. 4949–4959, 1802.08241.
[61] N.S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, P.T.P. Tang, On large-batch training for deep learning: Generalization gap and sharp minima, in: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April, 24-26, 2017, Conference Track Proceedings. 2017. 1609.04836.
[62] N. Golmant, N. Vemuri, Z. Yao, V. Feinberg, A. Gholami, K. Rothauge, M.W. Mahoney, J. Gonzalez, On the computational inefficiency of large batch sizes for stochastic gradient descent, 1811.12941.
[63] S. McCandlish, J. Kaplan, D. Amodei, O.D. Team, An empirical model of large-batch training, 1812.06162.
[64] Y. Nesterov, A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$, Dokl. AN USSR 269 (1983) 543–547.
[65] J. Duchi, E. Hazan, Y. Singer, Adaptive subgradient methods for online learning and stochastic optimization, J. Mach. Learn. Res. 12 (2011) 2121–2159.
[66] M.D. Zeiler, ADADELTA: an adaptive learning rate method, 1212.5701.
[67] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, in: 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May, 7-9, 2015, Conference Track Proceedings. 2015. 1412.6980.
[68] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, in: Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV) ICCV '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 1026–1034.
[69] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: Y.W. Teh, M. Titterington (Eds.), Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, in: Proceedings of Machine Learning Research, vol. 9, PMLR Chia Laguna Resort, Sardinia, Italy, 2010, pp. 249–256.
[70] S. Ioffe, C. Szegedy, Batch normalization: Accelerating deep network training by reducing internal covariate shift, in: Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37 ICML'15, JMLR.org, 2015, pp. 448–456, 1502.03167.
[71] G. Klambauer, T. Unterthiner, A. Mayr, S. Hochreiter, Self-normalizing neural networks, in: Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 2017, Long Beach, CA, USA, pp. 971–980. 2017. 1706.02515.
[72] K. Janocha, W.M. Czarnecki, On loss functions for deep neural networks in classification, 2017, arXiv e-prints, arXiv:1702.05659 1702.05659.
[73] S. Kullback, R.A. Leibler, On information and sufficiency, Ann. Math. Stat. 22 (1951) 79–86.
[74] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R, Salakhutdinov dropout: A simple way to prevent neural networks from overfitting, J. Mach. Learn. Res. 15 (2014) 1929–1958.
[75] T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning, in: Springer Series in Statistics, Springer New York Inc., New York, NY, USA, 2001.
[76] Y. Yao, L. Rosasco, A. Caponnetto, On early stopping in gradient descent learning, Constr. Approx. 26 (2007) 289–315.
[77] M. Kubo, R. Banno, H. Manabe, M. Minoji, Implicit regularization in over-parameterized neural networks, 1903.01997.
[78] C. Zhang, S. Bengio, M. Hardt, B. Recht, O. Vinyals, Understanding deep learning requires rethinking generalization, in: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April (2017) 24-26, Conference Track Proceedings. 2016. 1611.03530.
[79] S. Gunasekar, B.E. Woodworth, S. Bhojanapalli, B. Neyshabur, N. Srebro, Implicit regularization in matrix factorization, in: I. Guyon, U.V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems, Vol. 30, Curran Associates, Inc., 2017, pp. 6151–6159.
[80] R. Novak, Y. Bahri, D.A. Abolafia, J. Pennington, J. Sohl-Dickstein, Sensitivity and generalization in neural networks: an empirical study, 2018, arXiv e-prints, arXiv:1802.08760, 1802.08760.
[81] Z. Allen-Zhu, Y. Li, Y. Liang, Learning and generalization in overparameterized neural networks, going beyond two layers, 1811.04918.
[82] T. Salimans, D.P. Kingma, Weight normalization: A simple reparameterization to accelerate training of deep neural networks, in: D.D. Lee, M. Sugiyama, U.V. Luxburg, I. Guyon, R. Garnett (Eds.), Advances in Neural Information Processing Systems, Vol. 29, Curran Associates, Inc., 2016, pp. 901–909, 1602.07868.
[83] J. Lei Ba, J.R. Kiros, G.E. Hinton, Layer normalization, 2016, arXiv e-prints, arXiv:1607.06450 1607.06450.
[84] T. van Laarhoven, L2 regularization versus batch and weight normalization, 1706.05350.
[85] G. Cybenko, Approximations by superpositions of sigmoidal functions, Math. Control Signals Systems 2 (1989) 303.
[86] M.A. Nielsen, Neural Networks and Deep Learning, Determination Press, 2015.
[87] A.C. Wilson, R. Roelofs, M. Stern, N. Srebro, B. Recht, The marginal value of adaptive gradient methods in machine learning, 2017, arXiv e-prints, arXiv:1705.08292 1705.08292.
[88] R.M. Neal, Bayesian Learning for Neural Networks, Springer-Verlag, Berlin, Heidelberg, 1996.
[89] J. Lee, L. Xiao, S.S. Schoenholz, Y. Bahri, J. Sohl-Dickstein, J. Pennington, Wide neural networks of any depth evolve as linear models under gradient descent, 2019, arXiv e-prints, arXiv:1902.06720 1902.06720.
[90] G.A. Carpenter, S. Grossberg, J.H. Reynolds, Artmap: Supervised real-time learning and classification of nonstationary data by a self-organizing neural network, Neural Netw. 4 (5) (1991) 565–588.
[91] A.-H. Tan, Self-organizing neural architecture for reinforcement learning, in: J. Wang, Z. Yi, J.M. Zurada, L. B.-Lu, H. Yin (Eds.), Advances in Neural Networks - ISNN 2006, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 470–475.
[92] S. Grossberg, Adaptive resonance theory: How a brain learns to consciously attend, learn, and recognize a changing world. neural, Neural Netw. : Off. J. Int. Neural Netw. Soc. 37 (2012).
[93] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, in: Proceedings of the IEEE, 1998, pp. 2278–2324.
[94] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, 2014, arXiv e-prints, arXiv:1409.4842 1409.4842.

[95] M. Lin, Q. Chen, S. Yan, Network in network, 2013, arXiv e-prints, arXiv:1312.4400 1312.4400.
[96] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, Z. Wojna, Rethinking the inception architecture for computer vision, 2015, arXiv e-prints, arXiv:1512.00567 1512.00567.
[97] C. Szegedy, S. Ioffe, V. Vanhoucke, A. Alemi, Inception-v4, inception-resnet and the impact of residual connections on learning, 2016, arXiv e-prints, arXiv:1602.07261 1602.07261.
[98] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, 2015, arXiv e-prints, arXiv:1512.03385 1512.03385.
[99] S. Bianco, R. Cadène, L. Celona, P. Napoletano, Benchmark analysis of representative deep neural network architectures, IEEE Access 6 (2018) 64270–64277, 1810.00736.
[100] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A.C. Berg, L. Fei-Fei, Imagenet large scale visual recognition challenge, 2014, arXiv e-prints, 1409.0575.
[101] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, Commun. ACM 60 (2017) 84–90.
[102] D.H. Ackley, G.E. Hinton, T.J. Sejnowski, A learning algorithm for boltzmann machines, Cogn. Sci. 9 (1) (1985) 147–169.
[103] G.E. Hinton, S. Osindero, Y.-W. Teh, A fast learning algorithm for deep belief nets, Neural Comput. 18 (2006) 1527–1554.
[104] H. Larochelle, Y. Bengio, Classification using discriminative restricted boltzmann machines, in: Proceedings of the 25th International Conference on Machine Learning ICML '08, ACM New York, NY, USA, 2008, pp. 536–543.
[105] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, Generative adversarial nets, in: Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, K.Q. Weinberger (Eds.), Advances in Neural Information Processing Systems, Vol. 27, Curran Associates, Inc., 2014, pp. 2672–2680.
[106] I. Goodfellow, NIPS 2016 tutorial: Generative adversarial networks, 2016, arXiv e-prints, arXiv:1701.00160 1701.00160.
[107] D. Pfau, O. Vinyals, Connecting generative adversarial networks and actor-critic methods, in: NIPS Workshop on Adversarial Training, 2016, 1610.01945.
[108] M. Lucic, K. Kurach, M. Michalski, S. Gelly, O. Bousquet, Are GANs created equal? a large-scale study, 2017, arXiv e-prints, 1711.10337.
[109] K. Kurach, M. Lucic, X. Zhai, M. Michalski, S. Gelly, The GAN landscape: Losses, architectures, regularization, and normalization, 2018, arXiv e-prints, 1807.04720.
[110] J. Lin, Divergence measures based on the shannon entropy, IEEE Trans. Inform. Theory 37 (1991) 145–151.
[111] M. Arjovsky, L. Bottou, Towards principled methods for training generative adversarial networks, 2017, arXiv e-prints, 1701.04862.
[112] M. Arjovsky, S. Chintala, L. Bottou, Wasserstein GAN, 2017, arXiv e-prints, 1701.07875.
[113] D.P. Kingma, M. Welling, Auto-encoding variational Bayes, 2013, arXiv e-prints, arXiv:1312.6114 1312.6114.
[114] D. Jimenez Rezende, S. Mohamed, D. Wierstra, Stochastic backpropagation and approximate inference in deep generative models, 2014, arXiv e-prints, arXiv:1401.4082 1401.4082.
[115] C. Doersch, Tutorial on variational autoencoders, 2016, arXiv e-prints, arXiv:1606.05908 1606.05908.
[116] D.E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, first ed., Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1989.
[117] A. Yamaguchi, H. Nakajima, Landau gauge fixing supported by genetic algorithm, Nuclear Phys. Proc. Suppl. 83 (2000) 840–842, hep-lat/9909064.
[118] B.C. Allanach, D. Grellscheid, F. Quevedo, Genetic algorithms and experimental discrimination of SUSY models, J. High Energy Phys. 07 (2004) 069, hep-ph/0406277.
[119] Y. Akrami, P. Scott, J. Edsjo, J. Conrad, L. Bergstrom, A profile likelihood analysis of the constrained MSSM with genetic algorithms, J. High Energy Phys. 04 (2010) 057, 0910.3950.
[120] J. Blåbäck, U. Danielsson, G. Dibitetto, Fully stable ds vacua from generalised fluxes, J. High Energy Phys. 08 (2013) 054, 1301.7073.
[121] J. Blåbäck, U. Danielsson, G. Dibitetto, Accelerated universes from type IIA compactifications, J. Cosmol. Astropart. Phys. 1403 (2014) 003, 1310.8300.
[122] S. Abel, J. Rizos, Genetic algorithms and the search for viable string vacua, J. High Energy Phys. 08 (2014) 010, 1404.7359.
[123] S. Abel, D.G. Cerdeo, S. Robles, The Power of Genetic Algorithms: what remains of the pMSSM?, 1805.03615.
[124] L. Altenberg, The schema theorem and price's theorem, in: L.D. Whitley, M.D. Vose (Eds.), Foundations of Genetic Algorithms, in: Foundations of Genetic Algorithms, vol. 3, Elsevier, 1995, pp. 23–49.
[125] J.H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence, MIT Press Cambridge, MA, USA, 1992.
[126] C.L. Bridges, Goldberg D.E., An analysis of reproduction and crossover in a binary-coded genetic algorithm, in: Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and their Application, L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1987, pp. 9–13.
[127] T. Blickle, L. Thiele, A comparison of selection schemes used in evolutionary algorithms, Evol. Comput. 4 (1996) 361–394.
[128] P. Charbonneau, B. Knapp, A user's guide to PIKAIA 1.0, 1995.
[129] N. Mohd Razali, J. Geraghty, Genetic algorithm performance with different selection strategies in solving tsp, in: Proceedings of the World Congress on Engineering, vol. 2. 2011.
[130] J. Mendes, A comparative study of crossover operators for genetic algorithms to solve the job shop scheduling problem, WSEAS Trans. Comput. 12 (2013) 164–173.
[131] L. Davis (Ed.), Handbook of Genetic Algorithms, Vol. 115, Van Nostrand Reinhold, New York, 1991.
[132] S. Luke, L. Spector, A revised comparison of crossover and mutation in genetic programming, in: Genetic Programming 1997: Proceedings of the Second Annual Conference, Morgan Kaufmann, 1998, pp. 240–248.
[133] A. Cole, G. Shiu, Persistent homology and non-gaussianity, J. Cosmol. Astropart. Phys. 1803 (03) (2018) 025, [1712.08159].
[134] M. Cirafici, Persistent homology and string vacua, J. High Energy Phys. 03 (2016) 045, [1512.01170].
[135] A. Cole, G. Shiu, Topological data analysis for the string landscape, J. High Energy Phys. 03 (2019) 054, [1812.06960].
[136] Edelsbrunner Letscher, Zomorodian, Topological persistence and simplification, Discrete Comput. Geom. 28 (2002) 511–533.
[137] S. Oudot, Persistence Theory: From Quiver Representations to Data Analysis, in: Mathematical Surveys and Monographs, American Mathematical Society, 2015.
[138] A. Zomorodian, G. Carlsson, Computing persistent homology, Discrete Comput. Geom. 33 (2005) 249–274.
[139] R. Bellman, Dynamic Programming, first ed., Princeton University Press, Princeton, NJ, USA, 1957.
[140] S. Lloyd, Least squares quantization in pcm, IEEE Trans. Inform. Theory 28 (1982) 129–137.
[141] Q. Du, V. Faber, M. Gunzburger, Centroidal voronoi tessellations: Applications and algorithms, SIAM Rev. 41 (4) (1999) 637–676, [http://doi.org/10.1137/S0036144599352836].
[142] L. Bottou, Y. Bengio, Convergence properties of the k-means algorithms, in: Proceedings of the 7th International Conference on Neural Information Processing Systems NIPS'94, MIT Press, Cambridge, MA, USA, 1994, pp. 585–592.
[143] T. Zhang, R. Ramakrishnan, M. Livny, Birch: An efficient data clustering method for very large databases, SIGMOD Rec. 25 (1996) 103–114.

[144] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, in: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA, AAAI Press, 1996, pp. 226–231.
[145] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press Cambridge, MA, 1998.
[146] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis, Mastering the game of Go with deep neural networks and tree search, Nature 529 (2016) 484–489.
[147] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, D. Hassabis, Mastering the game of go without human knowledge, Nature 550 (2017) 354.
[148] J.I. Yellott, The relationship between luce's choice axiom, thurstone's theory of comparative judgment, and the double exponential distribution, J. Math. Psych. 15 (2) (1977) 109–144.
[149] C.J. Maddison, D. Tarlow, T. Minka, A* sampling, in: Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, K.Q. Weinberger (Eds.), Advances in Neural Information Processing Systems 27, Curran Associates, Inc., 2014, pp. 3086–3094.
[150] P.C. Mahalanobis, On the generalized distance in statistics, Proc. Natl. Inst. Sci. (Calcutta) 2 (1936) 49–55.
[151] J. Goldberger, G.E. Hinton, S.T. Roweis, R.R. Salakhutdinov, Neighbourhood components analysis, in: L.K. Saul, Y. Weiss, L. Bottou (Eds.), Advances in Neural Information Processing Systems 17, MIT Press, 2005, pp. 513–520.
[152] K.Q. Weinberger, J. Blitzer, L.K. Saul, Distance metric learning for large margin nearest neighbor classification, in: Y. Weiss, B. Schölkopf, J.C. Platt (Eds.), Advances in Neural Information Processing Systems 18, MIT Press, 2006, pp. 1473–1480.
[153] L. Hyafil, R.L. Rivest, Constructing optimal binary decision trees is np-complete, Inform. Process. Lett. 5 (1) (1976) 15–17.
[154] S.K. Murthy, Automatic Construction of Decision Trees from Data: A Multidisciplinary Survey, in Data Mining and Knowledge Discovery, Kluwer academic publishers, Boston, 1998, pp. 1–49.
[155] L. Breiman, J.H. Friedman, R.A. Olshen, C.J. Stone, Classification and regression trees, in: The Wadsworth statistics/probability series, Wadsworth & Brooks/Cole Advanced Books & Software, Monterey, CA, 1984.
[156] J.R. Quinlan, Induction of decision trees, Mach. Learn. 1 (1986) 81–106.
[157] S.L. Salzberg, C4.5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc. 1993, Mach. Learn. 16 (1994) 235–240.
[158] T. Chen, C. Guestrin, Xgboost: A scalable tree boosting system, in: Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining KDD '16 Pp, ACM, New York, NY, USA, 2016, pp. 785–794.
[159] Y. Freund, R.E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, J. Comput. System Sci. 55 (1997) 119–139.
[160] C. Cortes, V. Vapnik, Support-vector networks, Mach. Learn. 20 (1995) 273–297.
[161] M.A. Aizerman, E.A. Braverman, L. Rozonoer, Theoretical foundations of the potential function method in pattern recognition learning, in: Automation and Remote Control No. 25 in Automation and Remote Control, 1964, pp. 821–837.
[162] B.E. Boser, I.M. Guyon, V.N. Vapnik, A training algorithm for optimal margin classifiers, in: Proceedings of the Fifth Annual Workshop on Computational Learning Theory COLT '92, ACM, New York, NY, USA, 1992, pp. 144–152.
[163] V.N. Vapnik, The Nature of Statistical Learning Theory, Springer-Verlag, Berlin, Heidelberg, 1995.
[164] J.H. Schwarz, The power of M theory, Phys. Lett. B 367 (1996) 97–103, hep-th/9510086 [, 390(1995)].
[165] P. Horava, E. Witten, Heterotic and type I string dynamics from eleven-dimensions, Nuclear Phys. B 460 (1996) 506–524, [hep-th/9510209] [, 397(1995)].
[166] P. Candelas, A.M. Dale, C.A. Lutken, R. Schimmrigk, Complete intersection calabi-yau manifolds, Nuclear Phys. B 298 (1988) 493.
[167] J. Gray, A.S. Haupt, A. Lukas, All complete intersection calabi-yau four-folds, J. High Energy Phys. 07 (2013) 070, [1303.1832].
[168] M. Kreuzer, H. Skarke, Complete classification of reflexive polyhedra in four-dimensions, Adv. Theor. Math. Phys. 4 (2002) 1209–1230, [hep-th/0002240].
[169] F. Schöller, H. Skarke, All Weight Systems for Calabi-Yau Fourfolds from Reflexive Polyhedra, [1808.02422].
[170] V. Bouchard, H. Skarke, Affine Kac–Moody algebras, CHL strings and the classification of tops, Adv. Theor. Math. Phys. 7 (2) (2003) 205–232, [hep-th/0303218].
[171] P. Candelas, A. Font, Duality between the webs of heterotic and type II vacua, Nuclear Phys. B 511 (1998) 295–325, [hep-th/9603170].
[172] V. Braun, T.W. Grimm, J. Keitel, Complete intersection fibers in F-theory, J. High Energy Phys. 03 (2015) 125, [1411.2615].
[173] D.R. Morrison, W. Taylor, Toric bases for 6D F-theory models, Fortschr. Phys. 60 (2012) 1187–1216, [1204.0283].
[174] W. Taylor, Y.-N. Wang, Scanning the skeleton of the 4D F-theory landscape, J. High Energy Phys. 01 (2018) 111, [1710.11235].
[175] C. Wall, Classification problems in differential topology. v. on certain 6-manifolds, Invent. Math. 1 (1966) 355–374.
[176] M. Reid, The moduli space of 3-folds with k = 0 may nevertheless be irreducible, Math. Ann. 278 (1987) 329–334.
[177] A. Grassi, On minimal models of elliptic threefolds, Math. Ann. 290 (2) (1991) 287–302.
[178] M. Gross, A finiteness theorem for elliptic calabi-yau threefolds, Duke Math. J. 74 (1994) 271–299.
[179] G. Di Cerbo, R. Svaldi, Birational boundedness of low dimensional elliptic Calabi-Yau varieties with a section, 2016, arXiv e-prints, arXiv: 1608.02997 [1608.02997].
[180] L.B. Anderson, X. Gao, J. Gray, S.-J. Lee, Fibrations in CICY threefolds, J. High Energy Phys. 10 (2017) 077, [1708.07907].
[181] Y.-C. Huang, W. Taylor, On the prevalence of elliptic and genus one fibrations among toric hypersurface Calabi-Yau threefolds, J. High Energy Phys. 03 (2019) 014, [1809.05160].
[182] F. Denef, M.R. Douglas, Distributions of flux vacua, J. High Energy Phys. 05 (2004) 072, [hep-th/0404116].
[183] T.P.T. Dijkstra, L.R. Huiszoon, A.N. Schellekens, Supersymmetric standard model spectra from RCFT orientifolds, Nuclear Phys. B 710 (2005) 3–57, [hep-th/0411129].
[184] H.P. Nilles, P.K.S. Vaudrevange, Geography of fields in extra dimensions: String theory lessons for particle physics, Modern Phys. Lett. A 30 (10) (2015) 1530008, [1403.1597].
[185] L.B. Anderson, A. Constantin, J. Gray, A. Lukas, E. Palti, A comprehensive scan for heterotic SU(5) GUT models, J. High Energy Phys. 01 (2014) 047, [1307.4787].
[186] S. Groot Nibbelink, O. Loukas, F. Ruehle, (MS)SM-like models on smooth Calabi-Yau manifolds from all three heterotic string theories, Fortschr. Phys. 63 (2015) 609–632, [1507.07559].
[187] A.E. Faraggi, J. Rizos, H. Sonmez, Classification of standard-like heterotic-string vacua, Nuclear Phys. B 927 (2018) 1–34, [1709.08229].
[188] M. Cvetič, J. Halverson, L. Lin, M. Liu, J. Tian, Quadrillion F-theory compactifications with the exact chiral spectrum of the standard model, Phys. Rev. Lett. 123 (10) (2019) 101601, [1903.00009].
[189] A. Constantin, Y.-H. He, A. Lukas, Counting string theory standard models, Phys. Lett. B 792 (2019) 258–262, [1810.00444].
[190] O. DeWolfe, A. Giryavets, S. Kachru, W. Taylor, Type IIA moduli stabilization, J. High Energy Phys. 07 (2005) 066, [hep-th/0505160].
[191] B.S. Acharya, M.R. Douglas, A Finite landscape? [hep-th/0606212].
[192] E.I. Buchbinder, A. Constantin, A. Lukas, The moduli space of heterotic line bundle models: a case study for the tetra-quadric, J. High Energy Phys. 03 (2014) 025, [1311.1941].

[193] S. Groot Nibbelink, O. Loukas, F. Ruehle, P.K.S. Vaudrevange, Infinite number of MSSMs from heterotic line bundles?, Phys. Rev. D 92 (4) (2015) 046002, [1506.00879].
[194] F. Denef, M.R. Douglas, Computational complexity of the landscape. i., Ann. Phys. 322 (2007) 1096–1142, [hep-th/0602072].
[195] F. Denef, M.R. Douglas, B. Greene, C. Zukowski, Computational complexity of the landscape II—Cosmological considerations, Ann. Physics 392 (2018) 93–127, [1706.06430].
[196] M. Cvetic, I. Garcia-Etxebarria, J. Halverson, On the computation of non-perturbative effective potentials in the string theory landscape: IIB/F-theory perspective, Fortschr. Phys. 59 (2011) 243–283, [1009.5386].
[197] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., New York, NY, USA, 1979.
[198] S. Vavasis, Nonlinear Optimization: Complexity Issues, in: International Series of Monographs on Computer Science Series, Oxford University Press, 1991.
[199] S. Aaronson, P=?np, Electronic Colloquium on Computational Complexity (ECCC) 24 (2017) 4.
[200] P.W. Shor, Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM J. Sci. Stat. Comput. 26 (1997) 1484, [quant-ph/9508027].
[201] J.V. Matijasevič, The Diophantineness of enumerable sets, Dokl. Akad. Nauk SSSR 191 (1970) 279–282.
[202] R. Altman, J. Carifio, J. Halverson, B.D. Nelson, Estimating Calabi-Yau hypersurface and triangulation counts with equation learners, J. High Energy Phys. 03 (2019) 186, [1811.06490].
[203] D. Klaewer, L. Schlechter, Machine learning line bundle cohomologies of hypersurfaces in toric varieties, Phys. Lett. B 789 (2019) 438–443, [1809.02547].
[204] A. Constantin, A. Lukas, Formulae for Line Bundle Cohomology on Calabi-Yau Threefolds, [1808.09992].
[205] K. Bull, Y.-H. He, V. Jejjala, C. Mishra, Getting CICY high, Phys. Lett. B 795 (2019) 700–706, [1903.03113].
[206] K. Bull, Y.-H. He, V. Jejjala, C. Mishra, Machine learning CICY threefolds, Phys. Lett. B 785 (2018) 65–72, [1806.03121].
[207] Y.-H. He, S.-J. Lee, Distinguishing Elliptic Fibrations with AI, [1904.08530].
[208] J. Halverson, J. Tian, Cost of seven-brane gauge symmetry in a quadrillion F-theory compactifications, Phys. Rev. D 95 (2) (2017) 026005, [1610.08864].
[209] G. Martius, C.H. Lampert, Extrapolation and learning equations, in: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings. 2017. [1610.02995].
[210] Y.-N. Wang, Z. Zhang, Learning non-higgsable gauge groups in 4D F-theory, J. High Energy Phys. 08 (2018) 009, [1804.07296].
[211] H. Erbin, S. Krippendorf, GANs for generating EFT models, [1809.02612].
[212] M. Kreuzer, H. Skarke, No mirror symmetry in Landau-Ginzburg spectra! Nuclear Phys. B 388 (1992) 113–130, [hep-th/9205004].
[213] M. Kreuzer, H. Skarke, All Abelian symmetries of Landau-Ginzburg potentials, Nuclear Phys. B 405 (1993) 305–325, [hep-th/9211047].
[214] A. Mütter, E. Parr, P.K.S. Vaudrevange, Deep learning in the heterotic orbifold landscape, Nuclear Phys. B 940 (2019) 113–129, [1811.05993].
[215] H.P. Nilles, S. Ramos-Sanchez, P.K.S. Vaudrevange, A. Wingerter, The orbifolder: A tool to study the low energy effective theory of heterotic orbifolds, Comput. Phys. Comm. 183 (2012) 1363–1380, [1110.5229].
[216] F. Gmeiner, R. Blumenhagen, G. Honecker, D. Lust, T. Weigand, One in a billion: MSSM-like D-brane statistics, J. High Energy Phys. 01 (2006) 004, [hep-th/0510170].
[217] M. Cvetic, T. Li, T. Liu, Supersymmetric patiSalam models from intersecting D6-branes: A road to the standard model, Nuclear Phys. B 698 (2004) 163–201, [hep-th/0403061].
[218] E. Plauschinn, Non-geometric backgrounds in string theory, Phys. Rep. 798 (2019) 1–122, [1811.11203].
[219] G. Obied, H. Ooguri, L. Spodyneiko, C. Vafa, De sitter space and the swampland, [1806.08362].
[220] J.M. Maldacena, The Large N limit of superconformal field theories and supergravity, Internat. J. Theoret. Phys. 38 (1999) 1113–1133, [hep-th/9711200]; Adv. Theor. Math. Phys. 2 (1998) 231.
[221] K. Hashimoto, S. Sugishita, A. Tanaka, A. Tomiya, Deep learning and the AdS/CFT correspondence, Phys. Rev. D 98 (4) (2018) 046019, [1802.08313].
[222] K. Hashimoto, S. Sugishita, A. Tanaka, A. Tomiya, Deep learning and holographic QCD, Phys. Rev. D 98 (10) (2018) 106014, [1809.10536].
[223] K. Hashimoto, AdS/CFT correspondence as a deep Boltzmann machine, Phys. Rev. D 99 (10) (2019) 106017, [1903.04951].
[224] D. Krefl, S. Carrazza, B. Haghighat, J. Kahlen, Riemann-Theta Boltzmann Machine, [1712.07581].
[225] S. Carrazza, D. Krefl, Sampling the Riemann-Theta Boltzmann Machine, [1804.07768].