Course in (Quantum) Machine Learning

Adrián Pérez-Salinas
$\langle aQa \rangle^L$: Applied Quantum Algorithms, Leiden University, The Netherlands

January 15, 2025

Contents

Chapter 1

Introduction

The discipline of machine learning comprises a large variety of algorithms and techniques with the objective of making a machine automatically understand the hidden relationships of a training data. Machine learning has witnessed in recent years an exponential increment in its capabilities and in the influence it has on society. Some of the most celebrated examples nowadays are generative models, exemplified by Chat-GPT or Dall-E, although there have been many other examples applied to hard problems, such as protein folding [15].

Machine learning has enabled us to make use of the gigantic power of data to address problems that were considered intractable beforehand.

1.1 What is machine learning

We can begin by describing what is machine learning from a technical perspective. The central elements of machine learning will be covered in the subsequent paragraphs.

Machine learning model A machine learning model is a program f with an input x and an output $f(x)$, performing an operation to be determined by data. The models f are usually specified by an architecture and a set of tunable parameters θ. This means that an architecture is capable of realizing a large family of models $\mathcal{F} = \{f_\theta\}_\theta$, and we will need to adjust the parameters θ to find the optimal function.

Machine learning models can follow a large variety of architectures, to be specified later. Some architectures are more suited to some problems than other. For example, convolutional neural networks are highly convenient to deal with problems in which data is given in the form of images. Another examples are transformers, which have been extensively used in large language models. Multilayer fully connected neural networks have also been used with wide purposes. When training the parameters θ, the learning model finds an optimal solution *within* the architecture, but cannot outperform the architecture itself.

Data In machine learning models, data is given by the pairs (x, y), where x will be the data input into the model, and y will be output from the model. In certain cases, this data is truncated to only x. This data will be generically referred to as $\mathcal{D} = \mathcal{X} \times \mathcal{Y}$. The structure of the data *in principle* contains all information required to solve a particular problem. The goal of a machine learning algorithm is to infer from data the implicit function $f : \mathcal{X} \to \mathcal{Y}$, with the aim that we can predict the effect of this function on previously unseen data.

Data can be divided in several types of data, for instance training set, test set or validation set. Before delving into the differences of these kinds of data it is important to highlight some particularities of the framework. The data points $\mathcal{D}$ conform a distribution of points (x, y) which is in many cases continuous. In other words, one cannot access to all data available in $\mathcal{D}$, but only sample from it. This gives rise to the concept of training/test data.

Training data is the discrete subset of data $\mathcal{T} = \{(x, y)\}, (x, y) \sim \mathcal{D}$ used to train a model. The goal of the model, in broad terms, is to design a function f such that $f(x) \approx y(x)$ for all, or the majority, of data pairs (x, y).

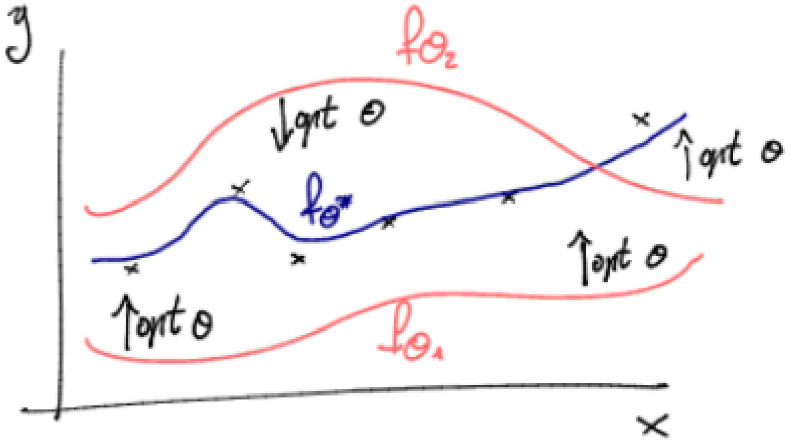


Figure 1.1: Optimization process of a machine learning algorithm. The model is capable of representing many functions (in red), which are optimized to fit the existing training data (in blue).

Test data is another discrete subset $\mathcal{C}$ of the same form as $\mathcal{T}$, that is not used to train the machine learning model. However, the success of the model depends on the performance over $\mathcal{C}$. Would the model perform accurately on $\mathcal{C}$, the model is considered successful. In this scenario, the model will have *learnt* the data, as opposed to *memorizing* it. Hence, the model is capable of generalizing the acquired knowledge.

There exists also the *validation set* $\mathcal{V}$, as a third dataset serving as a touchstone of machine learning model. The $\mathcal{V}$ is used to adjust the outcomes of a machine learning model after training. This step is not always necessary or used, but it is very common.

Training A fundamental step of all machine learning algorithms is the optimization of the parameters, or training the model. To perform training, one needs a) a training dataset and b) a training algorithm. The goal of the training is to minimize some cost function that quantifies how accurately is the algorithm predicting the outcomes of a data set.

The first ingredient is the loss function $\mathcal{L}(\theta)$, which can be computed with respect to different datasets, namely

$$\mathcal{L}_{\mathcal{D}}(\theta), \qquad \mathcal{L}_{\mathcal{T}}(\theta), \qquad \mathcal{L}_{\mathcal{C}}(\theta) \tag{1.1}$$

It is possible to compute $\mathcal{L}_{\mathcal{T}}(\theta), \mathcal{L}_{\mathcal{C}}(\theta)$, but $\mathcal{L}_{\mathcal{D}}(\theta)$ is not computable, since the dataset $\mathcal{D}$ is not accessible. In a nutshell, the only available information over $\mathcal{D}$ is obtained by sampling. Hence, only Monte-Carlo-like approximations can be computed.

A common example for the loss function is given by the mean squared error

$$\mathcal{L}_{\mathcal{T}}(\theta) = \frac{1}{|\mathcal{T}|} \sum_{x \in \mathcal{T}} (y(x) - f_\theta(x))^2, \tag{1.2}$$

although one can use of many other distance functions, with different performances for specific problems. The training process minimizes $\mathcal{L}_{\mathcal{T}}(\theta)$ with the hope to minimize $\mathcal{L}_{\mathcal{D}}(\theta)$ as well.

The second relevant ingredient is the optimization algorithm. Optimizing multidimensional functions is, in general, a very hard problem. It can be proven that it belongs to the class of NP-hard problems, hence finding the optimal solution is not a feasible scenario. However, some machine learning algorithms have specific optimization algorithms that enhance their applicability. This is not enough to find the optimal solution, but it provides *good* solutions, that is, instances that *approximately* solve the problem. The optimization problem is formulated as finding

$$\theta^* = \operatorname{argmin} \mathcal{L}_{\mathcal{T}}(\theta). \tag{1.3}$$

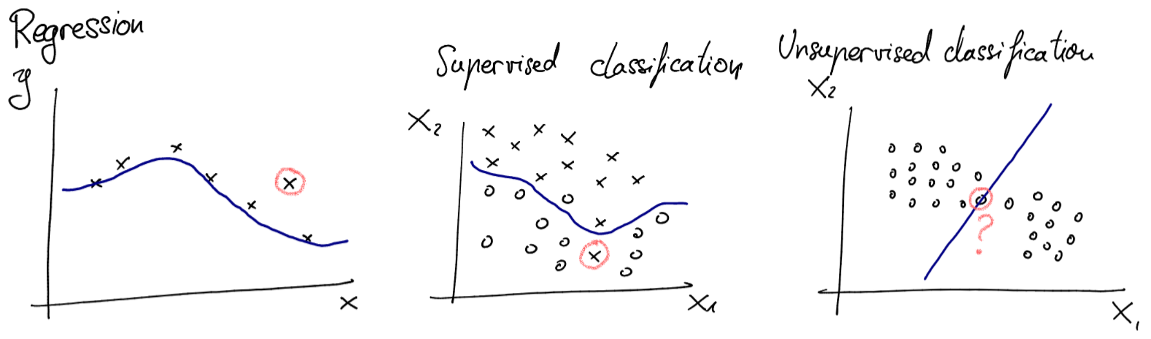


Figure 1.2: Different tasks to be addressed with machine learning. In regression, the task is to fit an unknown function known only through data. In supervised classification, the algorithm intends to separate between labeled data, to learn the feature that performs the classification. In unsupervised classification, the goal is to clusterize data, without any prior reference.

There exist a plethora of algorithms to solve optimization problem, and in general no optimization algorithm is capable of solving all of them, even approximately. Among the available options, one can distinguish simplicial algorithms (Nelder-Mead [21]), evolutionary algorithms (CMA-ES [11]) or gradient-based approach to first or higher orders [17, 5, 22].

In particular, neural networks admit the use of *backpropagation* as an optimization algorithm. This enables using neural network as a learning method, and lies at the core of machine learning as a product for industrial purposes. Backpropagation will be covered in depth within the scope of this course.

A third relevant ingredient is the generalization. As previously mentioned, the optimization of a machine learning algorithm can only learn the training data. The hope is that this learning is enough to learn also the test or validation data. An interpretation is that the model can actually *learn* the internal properties and structure of the data, instead of just *memorizing* the training data, and not being capable of inferring any information from previously unseen data. The difference between these two quantities is usually called the generalization error,

$$\mathcal{G}(\theta) = \left| \mathcal{L}_{\mathcal{D}}(\theta) - \mathcal{L}_{\mathcal{T}}(\theta) \right| , \tag{1.4}$$

usually studied through generalization bounds [29].

This topic will not be covered in these lectures, hence it is relevant to give some information about it. Generalization bounds capture the complexity of the model, that is, how many choices the training discards. As a consequence, it is possible to know how many training data is needed to narrow the outcomes of the learning algorithm. Intuitively, if the model is too sophisticated, it can learn any kind of data, but also there exist many possible outcomes that correspond to the same data. The training is not capable of knowing which of the outcomes is optimal for unseen data. Hence, the model generalizes poorly. In this case, the model is said to *overfit* the training data.

1.2 Types of machine learning

There exist several different types of machine learning algorithm, depending on the input and output of the problem at hand, and the objective of the learning procedure. In this section, we will give a brief description of the most relevant ones, with the goal of providing a comprehensive but not exhaustive overview of the field.

The different types for machine learning are here organized in an ascending order of lack of knowledge. The less is known about the data, the more unstable and difficult to train the machine learning algorithms become.

1.2.1 Regression

For solving a regreesion problem in machine learning, consider a function of the form

$$y : \mathbb{R}^m \to \mathbb{R}^n, \tag{1.5}$$

usually with $m > n$, but not requiredly. The goal is to find a hypothesis function $f_\theta(\cdot)$ such that, for all x in the domain of f,

$$y(x) \approx f_\theta(x). \tag{1.6}$$

The function $f_\theta(x)$ is known as the hypothesis function. A machine learning algorithm is capable of implementing a family of functions

$$\mathcal{H} = \{h_\theta\}_\theta, \tag{1.7}$$

where each θ is the label corresponding to each hypothesis function. The functional form of the hypothesis function usually depends explicitly on the parameters. An example of a hypothesis family is

$$\mathcal{H}_{\sin} = \{\sin(x + \theta)\}_\theta \,. \tag{1.8}$$

A machine learning algorithm capable of representing this hypothesis family will be capable of learning sinoidal functions, but will struggle with data that does not correspond to this particular functional form.

One could agree that machine learning for regression is nothing but statistical regression with a extra steps. Consider for instance the hypothesis family given by

$$\mathcal{H}_{\mathrm{poly},N} = \left\{ \sum_{n=0}^{N} a_n x^n \right\}, \tag{1.9}$$

for $x \in \mathbb{R}$. For this hypothesis family, it is possible to use N data points to find the optimal configuration of $\{a_n\}$ to fit the data. Alternatively, one can use $M \gg N$ data points and statistical regression procedures to obtain the configuration minimizing the errors. This model would be capable of representing any polynomial up to degree d. Since all analytical functions admit a representation in polynomials by virtue of Taylor's expansion, polynomial regression suffices in principle to approximate any function (within the limits of Taylor's theorem). However, many terms would be needed to approximate functions with slowly vanishing terms.

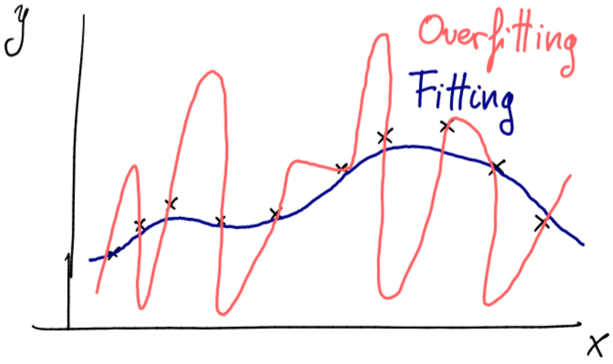


Figure 1.3: Overfitting in regression models for machine learning. The overfitted function can reach zero training error on the training data. However, its behavior between these points is unpredictable, yielding to large errors in test data. On the other hand, a fitted function may learn training data only approximately, with the advantage of yielding good approximations for test data. There is no *a priori* manner to know if a model is overfitting, one can only assess that property through testing.

While the previous paragraph is always available, it clashes with the spirit of machine learning. Regression techniques as the one explained before requires an *a priori* knowledge of the problem to be solved, sometimes dubbed as an ansatz. Machine learning, and in particular deep learning (to be discussed in Section 2.1), generates large hypothesis families with no immediate constructions of their functional forms to automatically look for an optimal solution.

Regression tasks allow for an immediate interpretation of *overfitting*, a common name for generalization error, see Chapter 1. Machine learning algorithms look for an optimal configuration of its internal degrees of freedom with respect to a training data, and it *hopes* for good performance also in test data. The relevant figure of merit is the accuracy with respect to the unseen data, to ensure *learning* of the internal structure of the data, rather than *memorizing*. Overfitting can be usually prevented by limiting the change rate of the hypothesis functions, usually through regularization cost functions.

1.2.2 Supervised classification

Supervised classification is a paradigmatic example of machine learning. In many machine learning talks, the subject is introduced as an algorithm capable of telling whether a given picture is a picture of a cat or a dog. In more formal terms, the goal is to receive data that is labelled as belonging to one class out of two (or many). Then, for new data, the algorithm must distinguish the classes among them.

Supervised classification can be seen as a particular case of regression, restricting the data function to the form

$$y : \mathbb{R}^n \to \mathbb{Z}. \tag{1.10}$$

One can map the data functions from regression (y_r) to classification (y_c) as follows. Consider the image of y_r being $\mathbb{R}$. Then, an integer rounding of the output would be enough to perform classification. Alternatively, if the image is $[0, 1]^C \in \mathbb{R}^C$, the classification can be performed by considering y_r as a step function, doable by rounding functions.

The paradigmatic example of supervised classification are support vector machines (SVM) (see Section 2.2). In this method, data points are classified with respect to a hyperplane dividing the data space in two pieces. To this end, the concept of *kernel* arises. A kernel is a function ϕ embedding data x into a feature space where the classification is performed. In a nutshell, it is possible to perform classification by just computing inner products between data points (in the feature space), rather than embedding this data into the space. In general, choosing the right kernel ϕ is crucial to the performance of the algorith. This acquires special relevance in quantum machine learning, where the first quantum advantages were found for quantum SVMs for well chosen embedding kernels [19].

In the references to binary classifications for SVMs, we will only focus on binary classification, as it is usually done in machine learning references. The reason is that it is possible to to combine binary classifiers to create a classifier for larger number of classes C. A possible approach is to create a binary classifier for distinguishing $C = 0$ and $C \neq 0$. Then, a second classifier is applied to the data labeled as $C \neq 0$ to classify this data as $C = 1$ or $C \neq 1$. This procedure can be repeated until the end of classes. Another approach is to utilize several classifiers to distinguish $C = i$ or $C \neq i$, for all $i = \{0, 1, \ldots C - 1\}$[1], and crosscheck the outputs.

1.2.3 Unsupervised classification

Unsupervised classification, also known as clustering, is the learning algorithm of dividing data in several classes, based on the internal structure of the data. In this case, there is no right answer, since the data does not provide a *true labelling*. In unsupervised learning, the performance can only be assessed with respect to the input of data, and not with respect to the output. Hence, unsupervised classification crucially depends on the quality of the data, not only on the capabilities of the learning procedure.

Consider the example here given in Figure 1.4. Visually, it is obvious that data can be divided in two classes. If the two *moons* are located one distant away from the other, a SVM together with euclidan distance should suffice to provide an accurate discrimination. In contrast, if the *moons* are located close to each other, it would be necessary to introduce a appropriate data embedding to ensure linear classification.

Unsupervised learning will not be exhaustively covered in these notes. However, good references can be found in [24].

1.2.4 Generative modelling

Generative modelling is a subfield of machine learning consisting in obtaining a model from which it is possible to sample data that resembles some other training data.

Consider a model $\mathcal{A}$ that is capable of outputting random variables of the form

$$x \sim \mathcal{D}_{\mathcal{A}}, \tag{1.11}$$

where $\mathcal{D}_{\mathcal{A}}$ is a probability distribution specified by the machine learning algorithm, either by its architecture or by its parameters.

[1] In the notes, I will start the indexing in 0, to keep consistency with the code in `Python`.

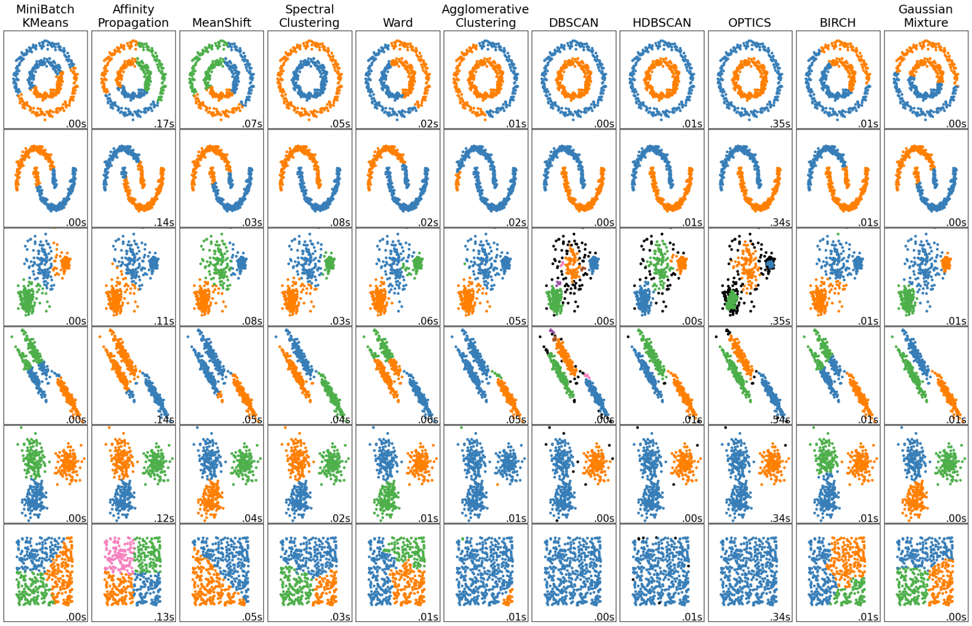


Figure 1.4: Example of data and proposed solutions by different algorithms for a supervised learning problem [24]. The data does not provide any information on how go distinguish between classes or even how many classes one should consider. Thus, this information has to be input into the algorithm by the user, yielding different results.

The random variable x is intended to be indistinguishable (or at least close enough) to some other training data, that is

$$x_t \sim \mathcal{D}_T \tag{1.12}$$

$$x_t \sim \mathcal{D}_\mathcal{A} \tag{1.13}$$

$$\mathcal{D}_T \approx \mathcal{D}_\mathcal{A}. \tag{1.14}$$

Hence the task is to approximate $\mathcal{D}_\mathcal{A}$ as much as possible to $\mathcal{D}_T$.

There exist several manners to sample random variables from a specified probability distribution. A simple way is to generate a function

$$f : \mathbb{R}^m \to \mathbb{R}^n, \tag{1.15}$$

and then apply f to a random variable sampled from a simple probability distribution. For example, consider $z \in \mathcal{N}(0,1)$, with $\mathcal{N}$ being a Gaussian distribution. Then, we define

$$x \equiv f(z) \sim \mathcal{D}_\mathcal{A}, \tag{1.16}$$

and the generative modeling problem can be understood as a re-interpretation of a regression problem, with a modified cost function.

Alternatively, one can define a model that generates samples instead of transforming a random variable. Consider for instance a binary decision tree. Each new branch can take two possibilities, namely 0 or 1, at random. At the end of the tree, the model returns the followed path, which is completely specified by a bitstring of 0's and 1's. Each bitstring will be sampled with different probability, depending on the model itself. In a side note, these decision-based algorithms can only return integer numbers, which can approximate real numbers to arbitrary accuracy.

A strong difficulty of generative modelling is that it is not possible to have access to the probability distributions from which data is sampled, either for training (as discussed before) or for generated data (since the possibilities are usually very large). Hence, one cannot rely on exhaustive descriptions of the data, but only on approximate descriptions that *hopefully* will suffice to solve the problem at hand. The same argument holds to compare target and obtained probability distributions. Measuring the distance between these distributions is in general difficult, hence imaginative methods have been developed, such as adversarial training or approximated distances, to be discussed in Section 3.2.

In spite of all the formal and theoretical problems that arise in generative models, this branch of machine learning / artificial intelligence has exhibited a strong development in recent years, being ChatGPT or Dall-E the most celebrated examples. Since these algorithms are capable of *mimicking* existing probability distributions, they excel at outputting samples of material that is well-known, for example tax forms, cover letters, or human faces. However, these models struggle when generating data which is significantly different from established sources. This is a consequence of data being impossible to sample exhaustively.

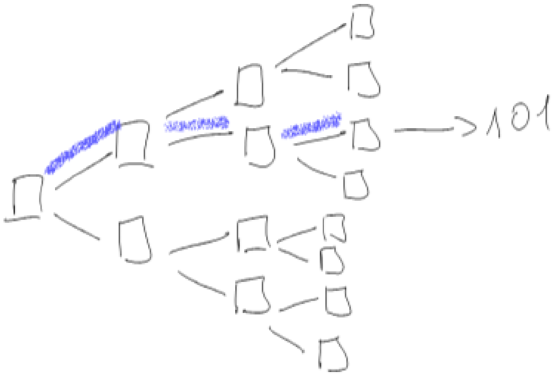


Figure 1.5: Example of a sampling model based on decision diagrams.

1.2.5 Reinforcement learning

In reinforcement learning, a machine learning algorithm is trained to interact with some environment and obtain some reward from it. At each step, the machine or *agent* can perform one *action* on the environment. The internal states of the agent and the environment may change. The action will return some *reward* on the agent, awarded by the *interpreter*, which can be constructed from data. The goal of the agent is to perform the sequence of actions that maximizes the obtained reward from an environment.

As an example, consider playing chess. The machine is initially equipped with the basic moves of the pieces. The agent is one player, and the environment is the opponent. The interpreter is composed by large data samples on previous matches played by humans. The machine moves one piece (takes one action) and the environment responds with another move. The position of the board can be compared to a stored database to estimate the probability of each player to win. Over time, the machine learns to play chess intelligently, by just imitating those moves that maximize the winning probability.

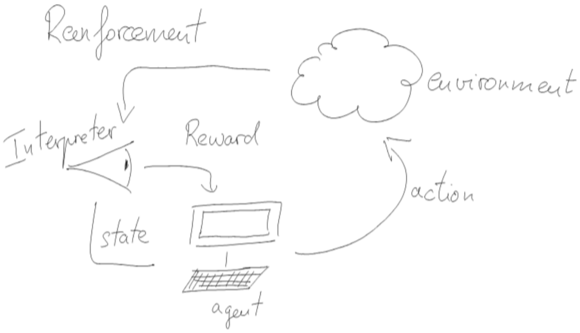


Figure 1.6: Scheme for reinforcement learning

Chapter 2

Regression and supervised classification

In this chapter we will cover the basic of regression and supervised classification, both theoretical notions (not so much) and numerical examples. We will create the models from scratch, to give ideas of what we are doing and why. We will cover neural networks (NN) and also support vector machines (SVM). These models are at the very core of machine learning.

2.1 Neural networks

Neural Networks (NN) are arguably the most relevant models in machine learning, and have been widely implemented to solve large varieties of problems. NNs are mathematical constructions with three main properties, which make them ideal to address machine learning tasks.

- *Expressivity:* NNs are models with large representation capabilities, that is, they can represent many different functions to accommodate arbitrary data. This implies that the models of the same form, possibly with different sizes, can be used to solve many different and complicated problems.
- *Trainability:* Apart form the expressive capabilities of NNs, its inner structure allows for an efficient training algorithm based on gradient optimization. Computing the gradients of the output functions of the NN with respect to their internal degrees of freedom is *efficient*, that is, it does not require any overhead with respect to computing the function itself. Hence, it is possible to find the optimal configuration of the NN with relatively small effort.
- *Linear algebra:* NN are based on matrix multiplication and application of easy functions repeatedly. This property allowed for the development of specific hardware, in particular GPUs, that are specifically suited to evaluate NNs, by accelerating matrix multiplication. The specific hardware, together with dedicated software and computational methods allowed for large-scale implementations of NN that have already revolutionized information processing.

2.1.1 What are neural networks

The mathematical structure of NNs is very simple. It is based on a feedforward mechanism that transports data from an input layer to an output data.

Let us consider first the single-layer neural network. The input data $x \in \mathbb{R}^m$ is input to the model through an input layer of m neurons. Then, this data is transferred to a middle (hidden) layer composed by h neurons. The input to any of these neurons is given by a linear combination of the input data as $w \cdot x + x_0$. This neuron then applies an *activation function*, $\sigma(\cdot) : \mathbb{R} \to \mathbb{R}$. Hence, the output of the hidden layer is given by

$$f_h(x) = \sigma_h(W_h x + b), \tag{2.1}$$

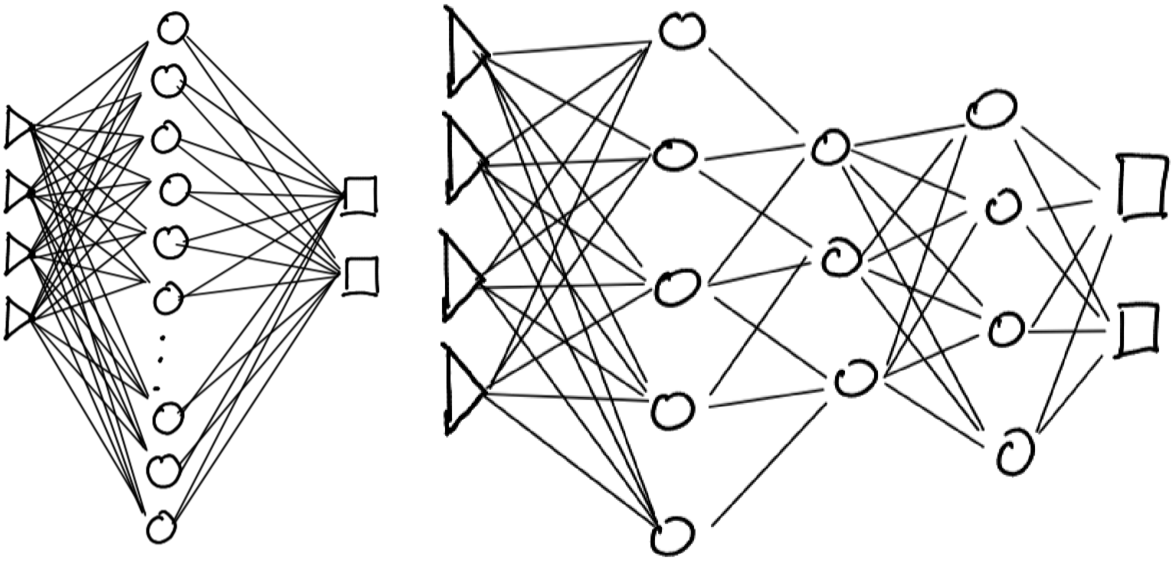


Figure 2.1: Single-layer and multilayer neural networks. Both models are universal, i. e., the output function (square) can be any function of the input (triangles). However, in practice, deep neural networks provide better processing layers than single layer.

with $W_h \in \mathbb{R}^{h \times m}, b \in \mathbb{R}^h$. The matrix of *weights* W_h and the *biases* b, together with the activation function, define the hidden layer. In a neural network with several layer, each weights matrix and bias, and each activation function are independently chosen.

For the output functions, the procedure is repeated, yielding

$$f(x) = \sigma_o(W_o \sigma(W_h x + b_h) + b_o). \tag{2.2}$$

This single-layer neural network suffices to provide an extremely rich hypothesis family, in fact *universal*, that is capable of representing any function. This is supported by the universal approximation theorem, see Section 2.1.2.

From the single-layer NN, it is immediate to construct a more sophisticated model known as deep neural network. Consider the same structure as explained in the single-layer neural network. The deep neural network is simply given by

$$f(x) = \sigma_L(b_L + W_L \sigma(b_{L-1} + W_{L-1}\sigma(b_{L-2} + W_{L-2}\sigma_{L-2}(\cdots \sigma(b_1 + W_1 \sigma(b_0 + W_0 x)) \cdots)))). \tag{2.3}$$

The definition of the NN creates an easy to evaluate and represent function with properties that are hard to characterize from an analytical point of view. Hence, the use of computational tools is a crucial element to study neural networks.

2.1.2 Universality of neural networks

Universal approximation theorem (UAT) [8] – Consider a NN where $\sigma_o(\cdot)$ is the identity function, and $n = 1$. Consider the activation function $\sigma_h(\cdot)$ to be any continuous sigmoidal function, that is

$$\lim_{x \to -\infty} \sigma(x) = 0 \tag{2.4}$$

$$\lim_{x \to \infty} \sigma(x) = 1. \tag{2.5}$$

Then, for every continuous function $y : [0,1]^m \to \mathbb{R}$, there exists a value of h such that, for every $\epsilon > 0$,

$$\max_{x \in [0,1]^m} |y(x) - f(x)| \leq \epsilon. \tag{2.6}$$

This is known as *uniform approximation*, implying that for any point x the approximation can be made arbitrarily small.

It is important to mention that this version of the theorem applies with some restrictions, specially in the choice of activation functions. However, it is possible to extend this result to more general situations, for example by using bounded, non-constant and continuous functions. In particular, we choose $\sigma(\cdot) = \cos(\cdot)$ to discuss the differences between NN and other representation theorems.

Consider the Fourier theorem, stating that any periodic square-integrable continuous function y, that is satisfying

$$\|y(x)\|_2 \equiv \int_{x\in[0,1]^m} |y(x)|^2 dx < \infty, \tag{2.7}$$

can be approximated by a function of the form[1]

$$f_N(x) = \sum_{\|n\|_1 \leq N} c_n \cos(n \cdot x) + s_n \sin(n \cdot x), \tag{2.8}$$

with

$$c_n = \frac{1}{(2\pi)^m} \int_{[0,1]^m} \cos(n \cdot x) y(x) \tag{2.9}$$

$$s_n = \frac{1}{(2\pi)^m} \int_{[0,1]^m} \sin(n \cdot x) y(x) \tag{2.10}$$

to an error

$$\|f_N(x) - y(x)\|_2 = \left| \|y(x)\|_2 - \sum_{\|n\|_1 \leq N} |c_n|^2 + |s_n|^2 \right|. \tag{2.11}$$

The intuition behind Fourier's theorem is as follows. The functions sin and cos constitute a basis in the space of square-integrable functions, including but not limited to continuous functions. This space is infinite-dimensional and equipped with the inner product

$$\langle f, g \rangle = \frac{1}{(2\pi)^m} \int_{x\in[0,1]^m} f(x) g^*(x) dx. \tag{2.12}$$

In the language of linear spaces, Fourier's theorem simply decomposes the function of interest in a basis of the space of functions, and creates a finite-dimensional representation of y. Since this function is square-integrable, the approximation obtains better accuracies as N increases.

The UAT, in its sin / cos form, can be rewritten as

$$f_h(x) = \sum_{i=1}^{h} c_i \cos(w_i \cdot x) + s_i \sin(w_i \cdot x). \tag{2.13}$$

The elements that construct this approximation have tunable weights. Each of these elements, for $w_i \in \mathbb{R}^m$ has a non-zero overlap with the basis elements $\sin(n \cdot x), \cos(n \cdot x)$. The immediate consequences, proven by UAT, is that NNs densely cover the space of continuous functions, in the supremum norm, due to the freedom in the weights. Additionally, note that the number of degrees of freedom scales as $h \times m$, while in the Fourier theorem, this number scales as N^m.

Extensions to UAT were proven for more general versions of single-layer NNs and also for deep NNs [14]. This collection of results theoretically supports the use of NNs for machine learning, but they do not justify *in practice* their widespread implementation for solving many different kind of problems. The usage is usually motivated by continuous good performance on the considered models on almost any task.

[1] Note that here $n \in \mathbb{Z}^m$, although the vectorial nature is not specifically mentioned.

2.1.3 Training of neural networks: backpropagation

We address now backpropagation, the most celebrated method to train NNs. The optimization process usually relies on gradient-based methods. For those methods to be used, it is required to estimate or compute the gradients of the neural networks, with respect to the internal degrees of freedom, namely weights and biases.

The framework of interest is as follows. Consider a loss function

$$\mathcal{L}_{\mathcal{T}}(W, b) = \frac{1}{|\mathcal{T}|} \sum_{x \in \mathcal{T}} D(y(x) - f_{\text{NN}}(x)), \tag{2.14}$$

for $D(\cdot, \cdot)$ being a distance, for example the mean squared error. We are interested in computing

$$\frac{\partial \mathcal{L}_x}{\partial W_l} = \frac{\partial D(y(x), f_{\text{NN}}(x))}{\partial f_{\text{NN}}(x)} \, \frac{\partial f_{\text{NN}}(x)}{\partial W_l} \tag{2.15}$$

$$\frac{\partial \mathcal{L}_x}{\partial b_l} = \frac{\partial D(y(x), f_{\text{NN}}(x))}{\partial f_{\text{NN}}(x)} \, \frac{\partial f_{\text{NN}}(x)}{\partial b_l} \tag{2.16}$$

$$\tag{2.17}$$

where the subindex x states the loss function evaluated in one data point. Aside from a small weight given by the distance function, the most relevant part is given by the derivatives of the NN with respect to the internal parameters. For the last layer, we can immediately compute the derivatives as δ_L. This quantity can be used to *backpropagate* the effect of training data on the derivatives with respect to the biases as

$$\frac{\partial f_{\text{NN}}(x)}{\partial b_L} = \sigma' \odot \frac{\partial f_{\text{NN}}(x)}{\partial \sigma_L} \frac{\partial D(y(x), f_{\text{NN}}(x)}{\partial y(x)} \tag{2.18}$$

$$\frac{\partial f_{\text{NN}}(x)}{\partial b_{l-1}} = \sigma'_{l-1} \odot (W_l)^T \frac{\partial f_{\text{NN}}(x)}{\partial b_l}, \tag{2.19}$$

with $\odot$ being a elementwise multiplication. The corresponding derivatives with respect to weights is just given by

$$\frac{\partial f_{\text{NN}}(x)}{\partial W_l} = \frac{\partial f_{\text{NN}}(x)}{\partial b_l} \left(\sigma_{l-1}\right)^T, \tag{2.20}$$

with $\sigma_l(x)$ being the output of the l-th layer of the NN.

The most important implication of backpropagation is that it is possible to compute the gradients with no extra cost as compared to evaluating the NN itself, up to matrix multiplication.

Stochastic Gradient Descent

Backpropagation is usually combined with stochastic gradient descent (SGD). This is nothing but computing average gradients over data, when the data is splitted in *batches*, and these batches are chosen at random. SGD favors training due to statistical fluctuations of the data, preventing the algorithm from being trapped in local minima of the optimization landscape.

2.1.4 Types of neural networks

There exist many different models on neural networks in the literature, that have been proposed and developed with the goal of efficiently addressing some particular problems. In this chapter we give an overview of them, with no aim to be exhaustive.

Activation functions

In practice, NNs can work with almost any kind of activation function that one can imagine. However, some examples have been more extensively used than other.

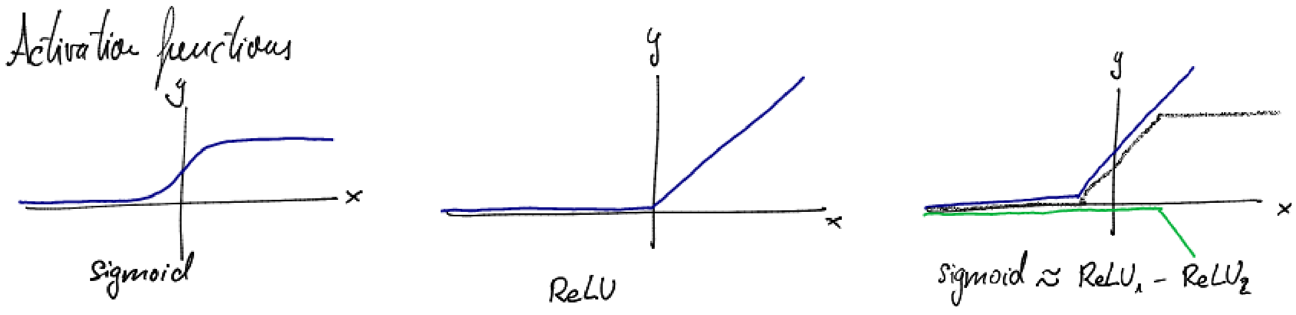


Figure 2.2: Recurrent neural network

Sigmoid – The sigmoid function is one of the most used activation functions in NNs. It is defined as

$$\sigma(x) = (1 + e^{-x})^{-1}. \tag{2.21}$$

The name *sigmoid* can provoke confusions with the notion of sigmoidal functions mentioned in **??**. In the theorem, a sigmoid function is just a function satisfying the conditions there specified. The choice here given fulfills those requirements, but it is a specific example.

This function has the property that

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)), \tag{2.22}$$

hence the back propagation algorithm can benefit from this choice to further reduce the computational cost of computing the derivatives.

ReLU – The ReLU function is defined as

$$\mathrm{ReLU}(x) = \max(0, x). \tag{2.23}$$

Interestingly, this function does not fulfill the requirements of sigmoidal functions specified in **??**. However, this can be solved by creating the function

$$g(x) = \frac{1}{|b - a|} \left(\mathrm{ReLU}(x - a) - \mathrm{ReLU}(x - b) \right), \tag{2.24}$$

with does fulfill the sigmoid conditions.

There is a more relevant reason to use ReLU functions, derived from the vanishing gradients problems. Recalling back propagation, it is noticeable that the size of the derivatives scales as $(\sigma')^L$, for fixed values of the weights. Hence, if $\sigma'| \leq 1$, the derivatives on the first layers are suppressed as compared to the last layers. In particular, the sigmoid function previously defined has derivatives close to zero in most of its domain, with the exception of the region around $x = 0$. By applying the ReLU function, these derivatives are constanly 1, avoiding the vanishing gradients problems.

Exotic neural networks

The NN previously explained is known as feedforward fully connected NN, and has all the relevant properties that make NNs unique. However, there exist many more examples of NNs.

Recurrent NNs – These networks propagate data not only forward, but also within the same layer, see Figure 2.2. Recurrent NNs are particularly useful for sequential data processing, in which the data of time step t depends on the output of step $t - 1$.

In the recurrent NN, the neurons have some internal memory, equivalent to the weights and biases in the feedforward NN, that is updated at every time step based on the input and its previous state. The memory keeps information from previous state and incorporate knowledge. There exists a variant of recurrent NNs named long-short term memory (LSTM) that solves the problem of vanishing gradients in recurrent NNs.

Convolutional NNs – Convolutional layers apply geometrical kernels, that is, a given fraction of the input is convolved with a given kernel, and the output is stored in the next layer, as

$$\mathrm{ConvNN}_{\mathrm{w}}(\mathrm{x})_i = \sum_j w_j x_{i+j}. \tag{2.25}$$

These NNs are particularly useful to process data in the form of images, since they are capable of extracting geometrical features of the input data. One can freely choose how the convolution is applied to optimize the performance of the model for the problem of interest.

Autoencoder – An autoencoder is a network that effectively reduces the dimensionality and complexity of the input data. Any kind of netowrk or layer here explained can be used to this purpose. The autoencoder is composed of two elements, namely the encoder and the decoder. The encoder received input x and outputs a succinct description of it. The decoder receives the succinct description and reverts the compression to obtain and output x'. If $x \approx x'$, then the decoder efficiently learns a dimensionality reduction of the input x.

Autoencoders can be used mainly in two tasks. First, one can consider large data to be compressed into a more simple representation. Since the simple representation is assumed to have the same information as raw data, it can be used as an input to another network to solve a learning problem. The cost of the network is reduced in this way. Second, one can generate data by inputing data in the compressed format to the decoder, creating a generative model.

Hopfield network – This network is a form of recurrent NN inspired by physics, in particular by spin glass systems. The Hopfield NN can be interpreted as a network in which neurons interact with each other, rather than acting on the input and feedforward it to the next layer. The Hopfield network provides memory, that is, its internal configuration evolves towards a ground state energy that stores patterns on training.

The Hopfield network is designed by

$$y_i = \mathrm{sign}\left(\sum_j w_{ij} x_j - b_i\right), \tag{2.26}$$

and they are designed to store information. If x is a corrupted state or a partial description of it, the Hopfield network will recover the memorized state. Convergence is not guaranteed.

Hopfield won the Nobel prize in Physics in 2024.

Boltzmann machine – The Boltzmann machine is inspired by Physics. Consider an Ising model as

$$H_{w,b}(x) = -\sum_i b_i x_i - \sum_{i,j} w_{i,j} x_i x_j, \tag{2.27}$$

with x being a boolean function. The weights and biases allows us to encode a probability distribution over the boolean functions given by

$$P(x) \propto \exp(-H_{w,b}(x)). \tag{2.28}$$

Boltzmann machines distinguish between visible and hidden units. This implies that one can use more than only weights and biases to model the output of the model. Additionally, these hidden units can be used as seed for randomness in generative models.

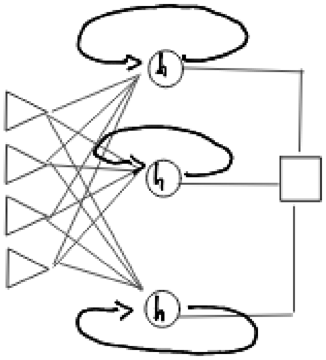
An interesting subclass of Boltzmann machines are restricted Boltzmann machines, which only allow for interactions between hidden and visible units, not inside the two components. These models have more efficient training algorithms.

Layers

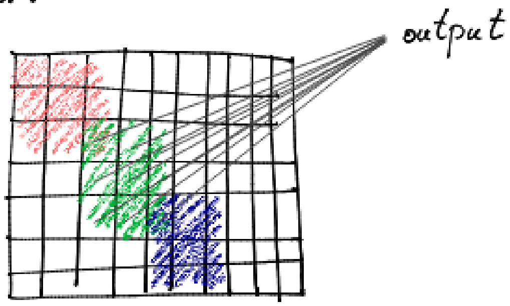
The previously mentioned networks and activation functions can, in principle, be combined at will to generate customizable neural networks tailored made for specific purposes. Those can also be combined with generic layers with particular effects.

The rational behind these layers is to remove sophistication of data that contributes redundantly to the learning process. This way, overfitting is prevented, and the training process is made less costly and easier to converge. Some examples are as follows.

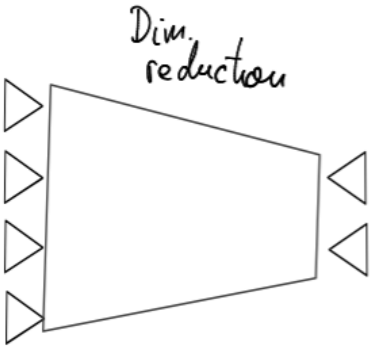
Recurrent NNs



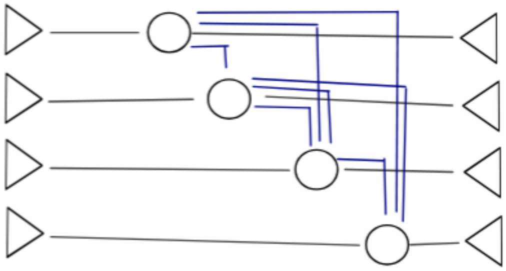
Convolutional



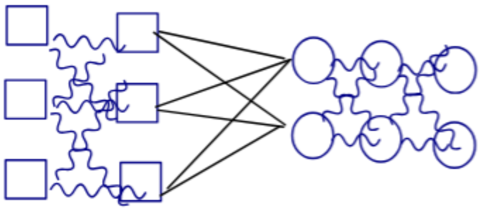
Autoencoder



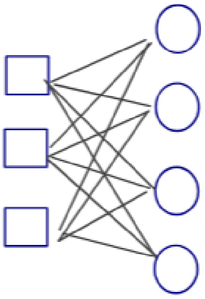
Hopfield.



Boltzmann



Restricted Boltzmann



DropOut – The DropOut layer randomly transforms features of the input to zero, or some other fixed or random value. The DropOut layers usually permit any kind of configuration.

Average – As denoted by the name, an average layer simply averages over partial descriptions of the inputs.

Pool – A pooling layer samples from the input, for instance by randomly selecting features or retrieving the maximal value.

2.1.5 Creating our own neural network

For this section, I refer the reader to the Jupyter Notebook available here.

2.1.6 Final remarks

In this section we have reviewed the basic of NNs, as well as some practical considerations for using them as machine learning algorithms.

The functioning principles of NNs can be understood from analytical considerations, such as the universality theorems and the backpropagation algorithm. Componentwise, it is also possible to acquire certain intuition on how each of them works. However, when all the elements are merged together (see Chapter 3), all intuition is lost, and it is really difficult to keep track of the internal processes of data within the network. This question gave rise to the problem of interpretability of machine learning, meaning *what are the processes that a NN carries out to extract knowledge from data.* Hence, numerical investigation has been a relevant tool to benchmark the performance of NNs in practice.

2.2 Support vector machines

Support Vector Machines (SVM) are a widely spread machine learning model with applications mainly on binary classification tasks. SVM are, in a nutshell, the process of finding the optimal frontier that linearly separates some data in two classes. The performance of any SVM depends on the properties of the data, or in available transformations that performs non-linear maps on the existing data.

There is a strong approach difference between SVMs and NNs. The functioning principle of NNs is to build a large pool of models that are efficient to optimize through gradient descent methods. This set of model is such that many possible solutions are available, and the difficulty is to find which one, among all possibilities, is optimal or close to optimal with respect to the training dataset. That is, NNs are models that do not require any a priori knowledge of the task to be solved, and the learning is conducted in an automatic fashion.

In contrast, SVMs do not properly *learn* the data, but rather they find the optimal separation by finding the solution to a quadratic optimization problem. Hence, it is required that information from the dataset is available, to construct a SVM with the possibilities to perform the required classification.

2.2.1 Binary classification of data

We address now the problem of using a SVM to binary classify data. Consider data in the form $\mathcal{D} = \{(x_i, y_i)\}_i$, with $x \in \mathbb{R}^m$, and $y \in \{-1, 1\}$. The goal of the SVM is to find a hyperplane that finds the optimal separation of data among classes. Such hyperplane is defined by a weights vector $w \in \mathbb{R}^m$, and bias . The classification is then given depending on whether the data point x is located at one side or the other with respect to the hyperplane, that is

$$y(x) = \text{sign}\left(w \cdot x - b\right). \tag{2.29}$$

Among the SVMs, we can distinguish two different cases, mainly hard- and soft-margin classifiers.

For hard-margin classifiers, the data must be linearly separable, and by assumption there exists a hyperplane that correctly classifies all data. This implies that there exist values for w, b such that

$$\forall x : \; y(x)(w \cdot x - b) \geq 1. \tag{2.30}$$

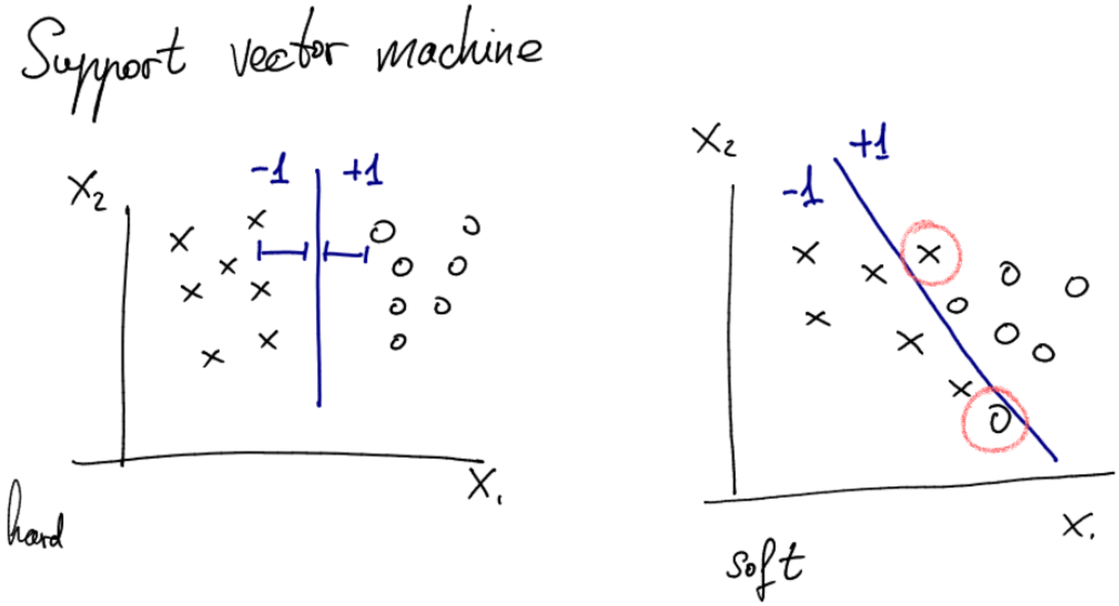


Figure 2.3: Support Vector Machines. The algorithm is capable of finding the optimal linear separation between labelled classes within the training dataset.

In the case of soft-margin classifiers, the goal is to minimize the loss function

$$C_{\mathcal{T}}(w,b) = -\frac{1}{\mathcal{T}} \sum_{x\in\mathcal{T}} \max\left(0, y(x)\left(w \cdot x - b\right)\right). \tag{2.31}$$

It is common practice to add a regularization term on $\|w\|$, yielding

$$w, b = \mathrm{argmin}_{w,b}\, \lambda \|w\|_2^2 + C_{\mathcal{T}}(w,b). \tag{2.32}$$

The regularization term is added to the optimization to enhance the generalization capabilities of the model. By limiting the sizes of w, it is only possible to find compromises between all data points to find the optimal classification. This optimization is referred to as the *primal* optimization of SVM.

It is interesting to find the solution of the optimization problem finding the *dual* optimization of the same problem. This duality is a consequence of Lagrangian optimization. Solving problems of the form

$$\text{minimize} \quad f(w) \tag{2.33}$$

$$\text{subject to} \quad g_i(w) \leq 0 \tag{2.34}$$

is equivalent, via Lagrangian multipliers, to solving the problem

$$\text{maximize} \quad f(w) + \sum_i c_i g_i(w) \tag{2.35}$$

$$\text{subject to} \quad \nabla f(w) + \sum_i c_i \nabla g_i(w) = 0. \tag{2.36}$$

In our case, we can identify

$$f(w) = \|w\|^2 \tag{2.37}$$

$$g_i(w) = y_i(w \cdot x_i - b) - 1. \tag{2.38}$$

The dual optimization is then

$$\min_{w,b} \max_{c\geq 0} L(w,b;c) \equiv \min_{w,b} \max_{c\geq 0} \lambda \|w\|^2 - \sum_i c_i y_i (w \cdot x_i + b) - 1. \tag{2.39}$$

Taking derivatives with respect to the internal degrees of freedom, we can obtain the constraints

$$\partial_w L(w, b; c) = 0 \to w = \frac{1}{2\lambda} \sum_i c_i y_i x_i \tag{2.40}$$

$$\partial_b L(w, b; c) = 0 \to \sum_i c_i y_i = 0, \tag{2.41}$$

which can be introduced into the original problem to find

$$c = \text{argmax}_c \max_{\sum_i c_i y_i = 0, c_i \geq 0} \sum_i c_i - \frac{1}{\lambda} \sum_{i,j} c_i c_j y_i y_j x_i \cdot x_j. \tag{2.42}$$

Notice that this new formulation allows to rewrite the optimization problem in terms of the internal products between data points, that is $x_i \cdot x_j$. The hyperplane parameters w, b can be recovered from c, if needed. However, the representer theorem allows to predict the class for each new point as

$$y(x_{\text{test}}) = \sum_{x_i \in \mathcal{T}} y_i y_{\text{test}} x_i \cdot x_{\text{test}} c_i, \tag{2.43}$$

recovering the weights-bias formulation.

Optimally, the soft-margin SVM will behave as a hard-margin classifier, provided the quality of the data. Would that not be possible, then the SVM finds the hyperplane that maximally separates the data.

2.2.2 Kernel trick

SVMs are optimal classifiers to find linear separations between classes. However, these methods are not capable of performing any non-linear separations. The kernel trick suffices to incorporate this capability into the SVM framework.

Consider the following classification problem. For $x \in \mathbb{R}^m$, the class is defined as

$$y(x) = \text{sign}(\|x\|_2 - R) = \text{sign}\left(\sum_i x_i^2 - R\right). \tag{2.44}$$

There is no value of (w, b) that can perform the correct classification in a SVM framework. However, by changing the relevant space $x \mapsto x^2$, the classification becomes trivial, by finding $w = (1, 1, \ldots, 1), b = R$.

This transformation is known as the Kernel trick, in which non-linear transformations are applied to the raw data to enhance classification. The map $\varphi(x)$ induces a transformation from the data space to a different feature space. The kernel is defined as

$$k(x_i, x_j) = \varphi(x_i) \cdot \varphi(x_j). \tag{2.45}$$

If this kernel is input to the dual optimization problem, it is immediate to observe that the kernel trick allows to perform linear classification in the space induced by the kernel, which does not need to be linear.

There exist several kernels that are commonly used to perform classification tasks, such as

Gaussian kernel – This kernel is defined as

$$k(x_i, x_j) = \exp\left(-\gamma \|x_i - x_j\|^2\right), \tag{2.46}$$

with γ being a tunable hyperparameters. This kernel performs well in classification tasks where data corresponding to the same classes concentrate in a relatively small region. The kernel rapidly decays for distances above γ^{-1}.

Polynomial kernel – The kernel is

$$k(x_i, x_j) = (x_i \cdot x_j)^d, \tag{2.47}$$

for a given (possibly tunable) given d. This kernel suffices to create boundaries that are sensitive to high-order polynomial.

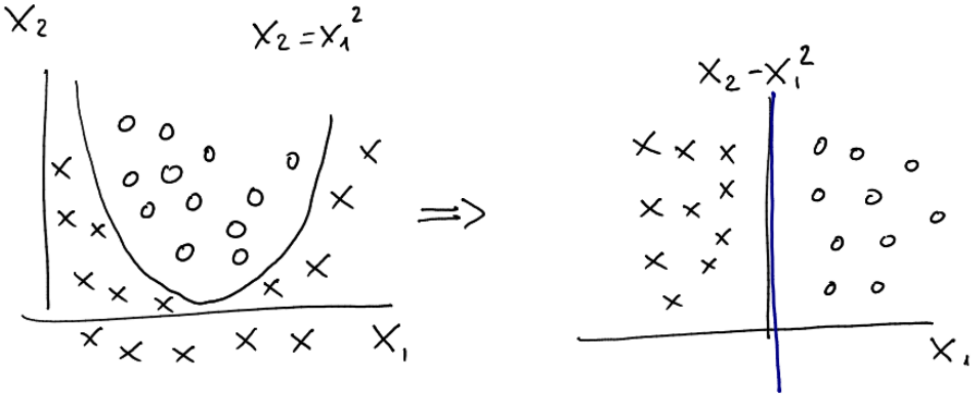


Figure 2.4: Description of the kernel trick for the classification of a non-linearly-separable dataset.

Sigmoidal kernel – The this kernel is defined as

$$k(x_i, x_j) = \sigma(a x_i \cdot x_j + b), \tag{2.48}$$

and also rapidly decays at distances a^{-1} from the bias b.

DLP kernel – This kernel is useful in quantum machine learning, to demonstrate the existence of quantum advantages. It is defined as follows. Let p be a prime number, and n the number of digits needed to represent it. For the cyclic group $\mathbb{Z}_p^*$ and for all $k = \{1, 2, \ldots, n-1\}$, we create the quantum state

$$C_{x,k} = \frac{1}{\sqrt{2^k}} \sum_{i \in \{0,1\}^k} \left| x \cdot g^i \right\rangle . \tag{2.49}$$

The kernel is defined as

$$K(x_i, x_j) = |\left\langle C_{x_i,k} \middle| C_{x_j,k} \right\rangle|^2 \tag{2.50}$$

2.2.3 Creating our own support vector machine

We will use the code in the associated jupyter notebook to create and train our own support vector machine from scratch.

2.3 Useful metrics for errors

Machine learning algorithms are based on heuristics, hence one needs normalized metrics to compare the performance from one model to another. There exist several established quantities. Some of the most common ones are the following. As a side note, these metrics are defined for binary classifications. However, one can have a collection of metrics by giving the same quantities to all classes to assess the performance of a multi-class classification.

Accuracy – The accuracy measures the proportion of correct predictions with respect to the size of the entire dataset, that is

$$\text{Accuracy} = \frac{\text{True positives}}{\text{Size dataset}}. \tag{2.51}$$

For multiclassifications, the accuracy is measured with respect to each of the classes.

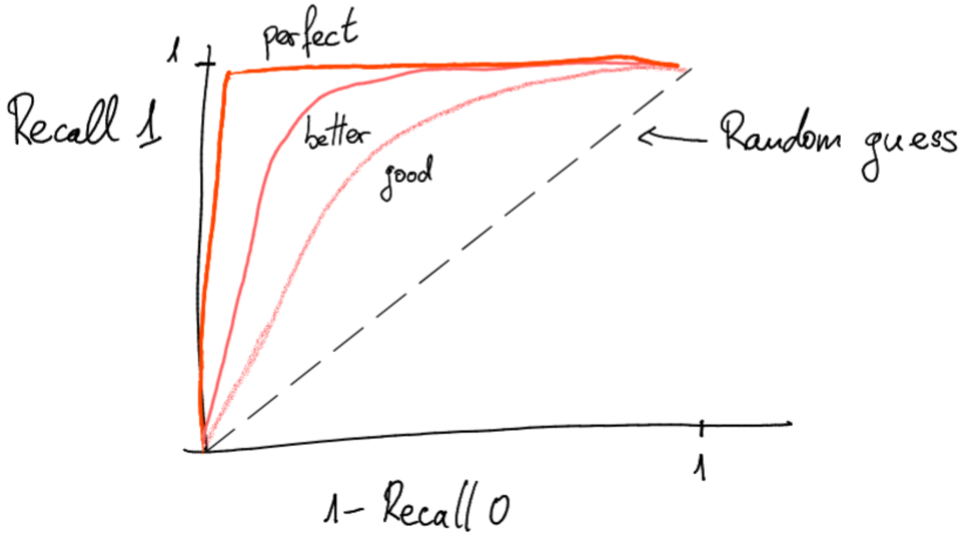


Figure 2.5: ROC curve for a binary classifier. Ideally, the classifier will be as far as possible from the random guess, and maximize the recall in classes 0 and 1 at the same time.

Precision – This metric measures the proportion of true predictions with respect to the number of predictions, that is

$$\text{Precision} = \frac{\text{True positives}}{(\text{True} + \text{False})\,\text{positives}}. \tag{2.52}$$

Recall – The recall measures the proportion of true preductions with respect to the number of positives, that is

$$\text{Recall} = \frac{\text{True positives}}{\text{True positives} + \text{False negatives}}. \tag{2.53}$$

F1 score – The F1 score balances precision and recall, and it is defined as

$$\text{F1} - \text{Score} = 2\frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.54}$$

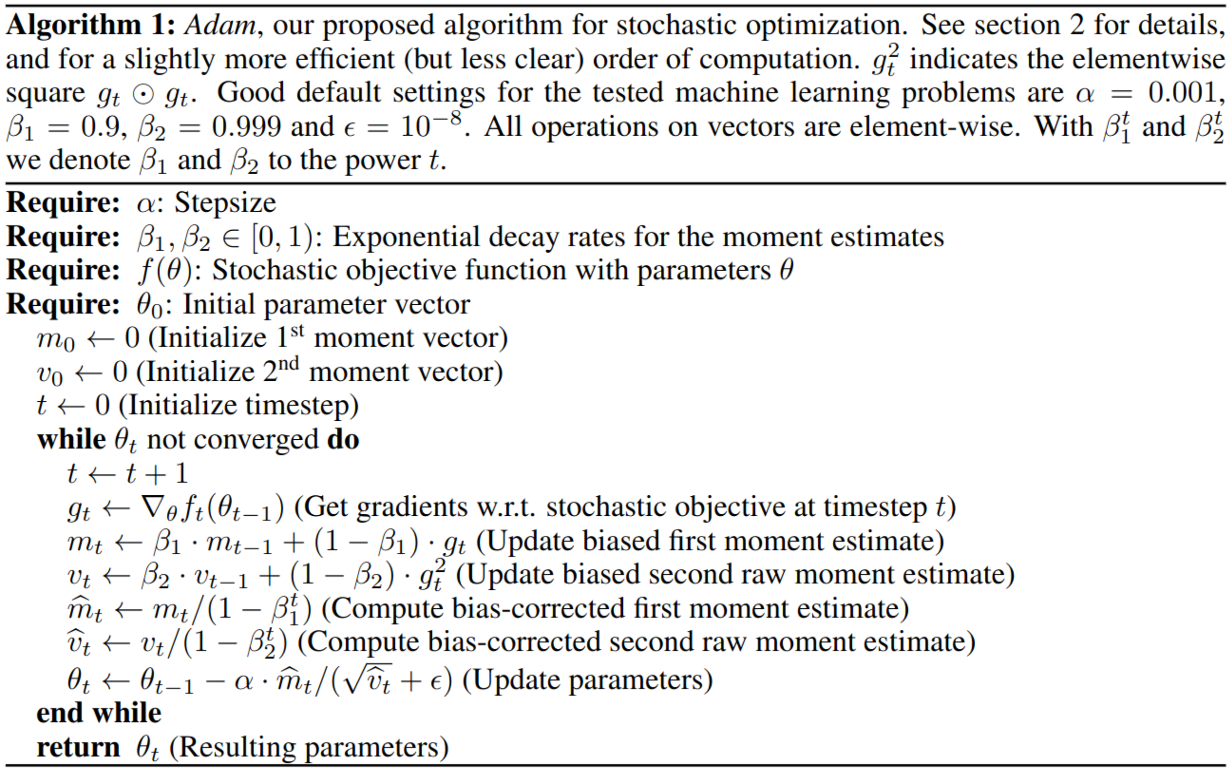
ROC curve – The ROC (Receiver Operating Characteristic) is a curve that compares the ratio of the sensitivity of the model, that is the recall, against the presence of false alarms. To obtain a ROC curve we need to vary the discrimination threshold between the two classes, and consider the Recall for the classes 1 and 0 as $\text{Recall}_{0/1}$. The ROC curve is given as in Figure 2.5. The usual relevant number is the area under the curve, a number normalized between 0 and 1.

2.3.1 Common gradient-descent optimizers – ADAM

The ADAM optimizer [17] is one of the most common gradient descent optimizers, widely used in machine learning and neural networks. In words of the authors

> The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients.

The ADAM optimizer is based on averages of gradients, given to the first and second order, by m_t and v_t respectively in the code. During the first timesteps, the algorithm provides large variabilty in these averages,



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

while as the iteration number grows they converge to a stationary point. This is achieved by controlling the hyperparameters α, β_1, β_2, which make the influence of each new step decay in each new iterations.

Chapter 3

Introduction to TensorFlow

As we have seen in previous chapters, it is possible to create NNs, SVMs and also other models for machine learning. However, the current extensive usage of machine learning is only possible due to the existence of dedicated hardware and software packages with specific optimization. Most of these packages are highly developed packages with pre-defined computations and automatic pipelines that allow researchers and data scientists to create machine learning models with small effort and high efficiency, in a plug-and-play strategy. All within a matter of minutes. In this chapter, we will review one of these tools: TensorFlow.

TensorFlow is a machine learning pipeline developed by Google. It machinery is very easy to understand and scalable. It consists of *layers*, that can be combined to build mainly NNs with a flexible architecture. The only requirement is to make the relative sizes of consecutive layers match, to effectively pass the information forward to the NN. After the architecture is designed, all the backpropagation accesory elements needed to conduct the learning of some unknown data is pre-implemented in the software, and it is possible to perform the entire learning process with a few lines of code.

See for instance the code below.

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(128, activation='relu'),
  tf.keras.layers.Dropout(0.2),
  tf.keras.layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)

model.evaluate(x_test,  y_test, verbose=2)
```

This code performs the following steps

1. Load the TensorFlow package

2. Load the dataset of interest. This step is here for demonstration purposes, since the MNIST dataset is a very common dataset to test machine learning models. The MNIST dataset is composed by squared images (28 x 28 in the format of TensorFlow) of handwritten digits. The goal of the learning process is to recognize the number written in each of the images.

3. Preprocess the data.

4. Generate a model composed by the following layers

   (a) A *flatten* layer, to transform a square image into a 1-dimensional array

   (b) A dense fully connected layer, as in Section 2.1

   (c) A dropout layer with probability 0.2 to drop (transform to zero)

   (d) A second fully connecter layer with output 10, which is the number of classes required.

5. Compiling the model, to enhance performance for computation. This step has the information of which optimizer to use, which loss to use for training, and how to benchmark the accuracy for the trained model.

6. Fit, or train, the model with respect a training data.

7. Evaluate the model in the training data.

The official guide of TensorFlow confirms that this model trains within a few seconds, and achieves accuracies of classification of nearly $\sim 98\%$.

The performance of these models depend mainly on how the compilation process works. At this step, TensorFlow is capable of very effectively dealing with hardware acceleration, for instance using GPUs or parallelism. There exist exhaustive resources to optimize code to be used in TensorFlow (or similar) packages, that can boost the data analysis capabilities of almost any kind of data.

TensorFlow is highly optimized to perform the tasks it was designed for. However, the only flexibility one can use is the one that has been foreseen. Creating customized models is possible, but it requires deeper knowledge of the internal machinery to ensure compatibility of all the required elements.

For the next sessions, it is recommendable to refer to the Notebooks.

3.1 Neural networks with TensorFlow

TensorFlow does not add anything fundamentally new on NNs to be added to the content given in the previous chapter. However, TensorFlow is highly optimized to run large NNs on high-dimensional data and perform classification tasks in a matter of seconds. For instance, consider the MNIST dataset, that is handwritten numbers as in Figure 3.1. A code of the form

```
import tensorflow as tf
((X_train, y_train), (X_test, y_test)) = mnist.load_data()

# We create now the model through the application of several layers.

model = tf.keras.models.Sequential([
  tf.keras.layers.InputLayer(shape=(28, 28)) ,
  tf.keras.layers.Flatten(input_shape=(28, 28)), # Transformation of squared images to 1-dimensional
  tf.keras.layers.Dense(128, activation='relu'), # fully connected layer, with output 128
  tf.keras.layers.Dense(10)                      # Second layer, with output 10, and final layer
])

model.compile(
    optimizer=tf.keras.optimizers.Adam(0.001), # Optimizer, can be changed
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True), # Loss function, can be ch
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()], # Metrics, to check the performance
)

model.fit(
    X_train, y_train, epochs = 10
) # In each epoch, the training data is shuffled to improve learning


model.compile(
    optimizer=tf.keras.optimizers.Adam(0.001), # Optimizer, can be changed
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True), # Loss function, can be ch
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()], # Metrics, to check the performance
)

model.fit(
    X_train, y_train, epochs = 10
) # In each epoch, the training data is shuffled to improve learning

test_loss, test_acc = model.evaluate(X_test, y_test, verbose=2)

print('\nTest accuracy:', test_acc)

y_pred = np.argmax(model.predict(X_test), axis = 1)
```

is capable of returning a prediction accuracy of $97,34\%$ within a training time of 30 seconds in a laptop. The prediction can be seen in Figure 3.1.

3.1.1 Fashion MNIST dataset

We can use TensorFlow to classify more complex datasets, in particular one of images obtained from a catalogue of clothes. This dataset, the Fashion MNIST, as well as the digits MNIST, are common testbeds for machine learning algorithms. Performing a similar neural network from the one obtained in the previous example, we can see the results in Figure 3.2.

For this classification task, we have added a convolutional layer, which is designed to work well for image classification. In this case, the training is longer, of ca. 5 minutes in a laptop.

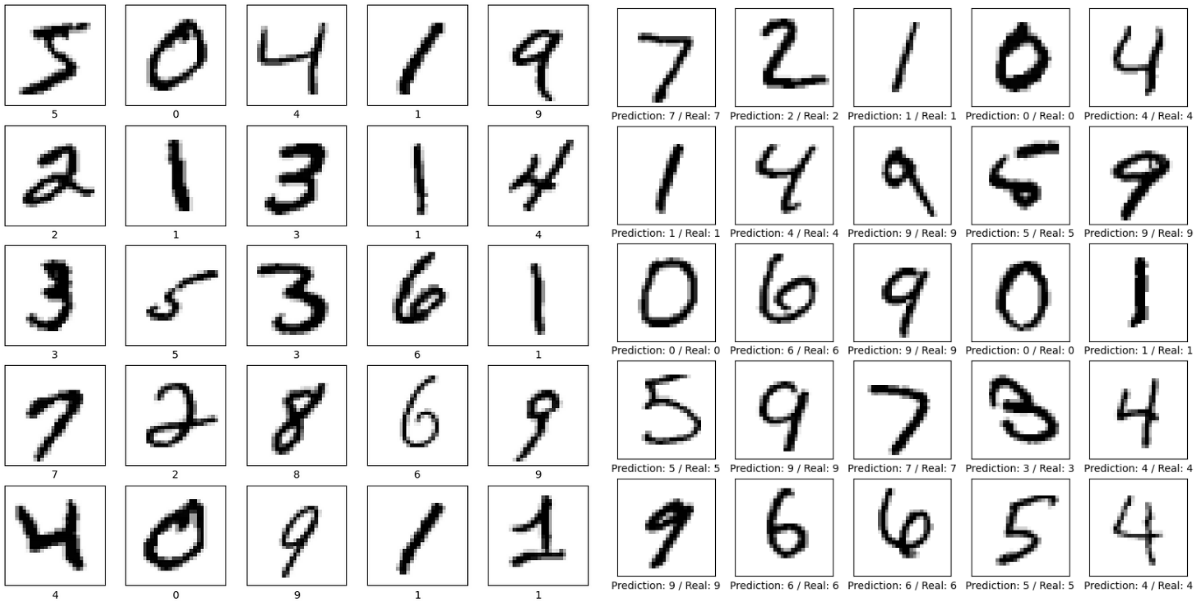


Figure 3.1: The MNIST dataset of handwritten digits (left) and its predicted classification using a simple `TensorFlow` neural network.

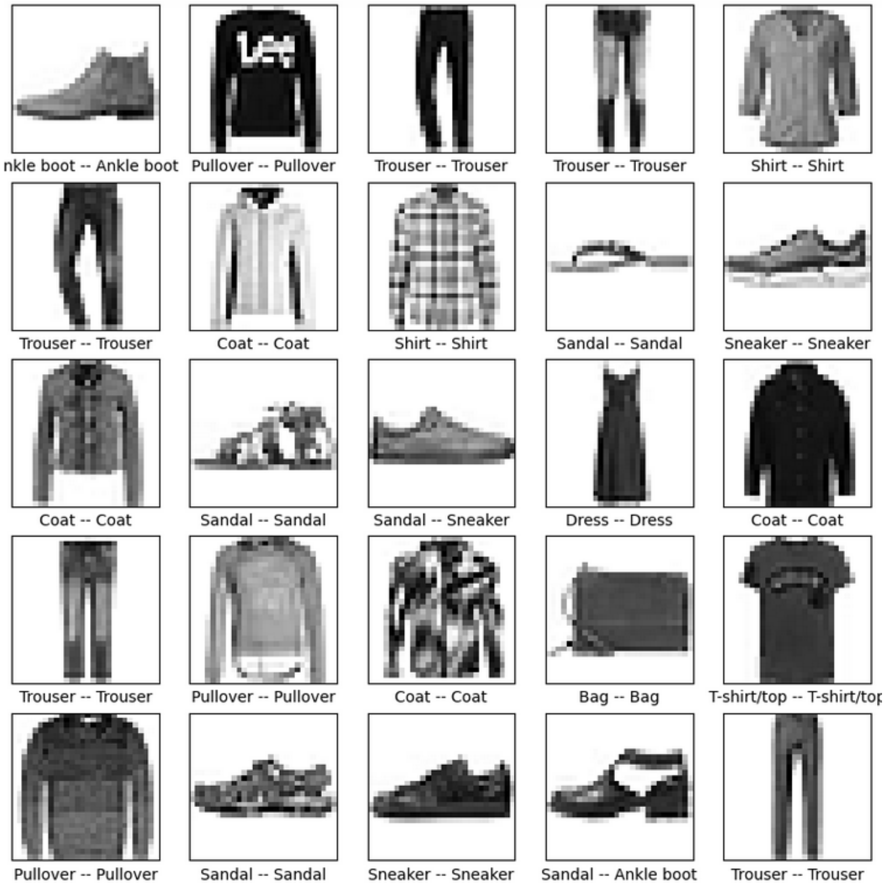


Figure 3.2: The prediction of a `TensorFlow` simple neural network on the Fashion MNIST dataset of clothes images.

```
import tensorflow as tf
((X_train, y_train), (X_test, y_test)) = fashion_mnist.load_data()

# We create now the model through the application of several layers.
# IN this example we use a convolutional neural networks, that we have mentioned it is ideal for ima

# Question: Do we need it?

model = tf.keras.models.Sequential([
  tf.keras.layers.InputLayer(shape=(28, 28, 1)),
  tf.keras.layers.Conv2D(10, 2, input_shape=(28, 28, 1)), # Convolutional layer of squared images
  tf.keras.layers.Flatten(input_shape=(10, 10)),
  tf.keras.layers.Dense(128, activation='relu'), # fully connected layer, with output 128
  tf.keras.layers.Dense(10)                      # Second layer, with output 10, and final layer
])

model.compile(
    optimizer=tf.keras.optimizers.Adam(0.001), # Optimizer, can be changed
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True), # Loss function, can be ch
    metrics=[tf.keras.metrics.SparseCategoricalAccuracy()], # Metrics, to check the performance
)

model.fit(
    X_train, y_train, epochs = 10
) # In each epoch, the training data is shuffled to improve learning
```

3.1.2 Classifying the Ising model

In the final example for neural networks, we will use TensorFlow to classify data from an Ising model, a common example in statistical Physics. The Ising model is defined by a Hamiltonian of the form

$$H = -\sum_{i,j} s_i s_j, \tag{3.1}$$

with s_i, s_j being spins up and down, organized over a square lattice. The goal of the Ising model is to obtain the state at which the model ended, as a function of the temperature. We can identify two main cases in the Ising model, mainly the ferromagnetic and antiferromagnetic phases, or ordered and disordered. In a nutshell, for low temperatures, the system will naturally fall into a low-energy state, that is $s_i = s_j$. On the other hand, for high temperatures, the thermal excitations allow for disordered phases. The phase transition happens at $T = 2.269K$. See **??** for a graphical example of both phases.

We can use tensorflow to classify data from both phases using neural networks and the states (or images) as data. A code of the following form suffices to achieve accuracies over 95%.

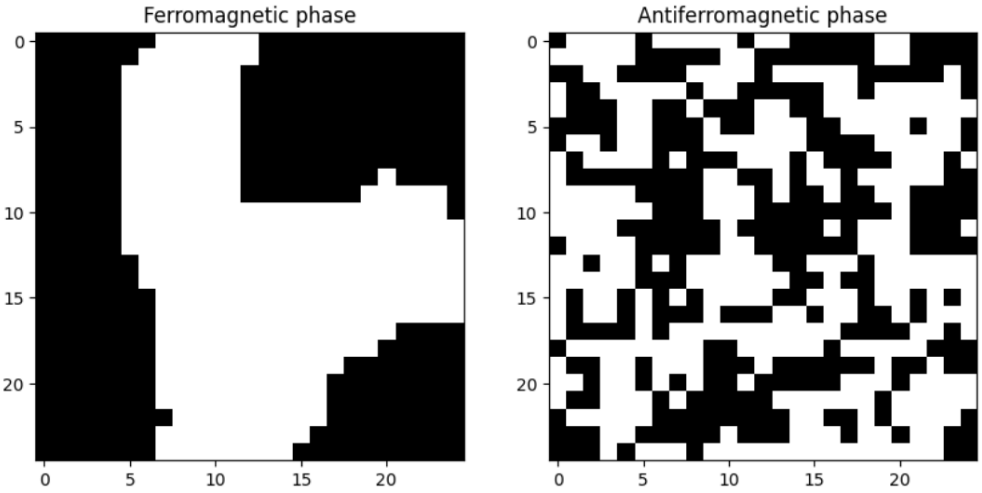


Figure 3.3: Two Ising phases: ferromagnetic or ordered (left) and antiferromagnetic or disordered (right).

```
import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.regularizers import l2

# Initialize model
model = Sequential()

# Add hidden layer with 32 units and rlu activation. Add output layer with sigmoid activation.
model.add(Dense(25, activation=’relu’, kernel_regularizer=l2(0.1), input_dim=25**2))
model.add(Dense(15, activation=’relu’, kernel_regularizer=l2(0.1)))
model.add(Dense(1, activation=’sigmoid’))

# Compile model
model.compile(optimizer=’Adam’, loss=’binary_crossentropy’, metrics=[’accuracy’])

# Fit model
model.fit(X, y, epochs=40, verbose=0)

# Add training and test errors to lists
train_errors = model.evaluate(X,y,verbose=0)[1]
test_errors = model.evaluate(X_test,y_test,verbose=0)[1]
```

3.2 Generative modeling with TensorFlow

Generative modelling is more complicated than just classification. In this example we will use adversarial training to construct a generative algorithm.

Adversarial training is composed by two elements: the generator and the discriminator. The generator is the neural networks from which new samples will be taken. Alternatively, the discriminator is a trained model capable of distinguishing whether a given data is taken from the dataset or not. This yields two different cost functions,

$$\mathcal{L}_{\text{Gen}} = -\frac{1}{|\mathcal{T}_{\text{Gen}}|} \sum_{x \in \mathcal{T}_{\text{gen}}} D(x) \tag{3.2}$$

$$\mathcal{L}_{\text{Disc}} = \frac{1}{|\mathcal{T}_{\text{Gen}}|} \sum_{x \in \mathcal{T}_{\text{gen}}} D(x) - \frac{1}{|\mathcal{T}_{\text{Real}}|} \sum_{x \in \mathcal{T}_{\text{Real}}} D(x) \tag{3.3}$$

which compete. In our case, use the cross-entropy between two probability distributions, but any other can be in principle used. Since both the generator and the discriminator are neural networks, one can use the backpropagation trick to compute gradients and update the network, allowing for an efficient training process. From this point, it is possible to choose an optimizer that completes the work for us. The optimizer just applies gradient-descent-like algorithms.

[TO BE COMPLETED]

Chapter 4

Quantum Machine Learning

Quantum machine learning (QML) is the discipline that combines quantum computing with machine learning to leverage quantum processing capabilities.

There exist several relevant concepts that distinguish classical and quantum types of machine learning, and that also distinguish QML from pure quantum computing.

Quantum computing in a nutshell – Quantum computing is a new paradigm for computation stemming from translating bits, that is units of information with the states 0/1 into qubits, quantum units of information that can take any superposition

$$|q\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,. \tag{4.1}$$

A n-qubit quantum state, in full generality, is written as

$$|\psi\rangle = \sum_{i=0}^{2^n-1} \alpha_i\,|i\rangle\,, \tag{4.2}$$

under the constraint that $\langle\psi|\psi\rangle = 1$.

In classical computing, available classical operations are those functions transforming bitstrings into bitstrings. All logical operations can be performed by combining FANOUT (copies) and NAND operations, that is

$$\mathrm{NAND}(0,0) = 1 \tag{4.3}$$
$$\mathrm{NAND}(0,1) = 1 \tag{4.4}$$
$$\mathrm{NAND}(1,0) = 1 \tag{4.5}$$
$$\mathrm{NAND}(1,1) = 0. \tag{4.6}$$

In the case of quantum computing, the available logical operations are unitary gates $U \in \mathcal{S}U(2^n)$, allowing for a much larger variety of operations. For instance, a single-qubit quantum gate is any operation of the form

$$U(\theta,\phi,\lambda) = \begin{pmatrix} \cos(\theta) & -\sin(\theta)e^{i\phi} \\ \sin(\theta)e^{i\lambda} & \cos(\theta)e^{i\phi}e^{i\lambda} \end{pmatrix} \tag{4.7}$$

This change of paradigm allows quantum computers to perform operations that cannot be efficiently computed with classical resources.

Efficiency of computations – A key concept is complexity theory is that of efficiency of computations, that is, what is the computational cost of running a given algorithm, in terms of operations. The predominant argument is scaling, that is, what is the functional form that dominates the effort, with respect to the size of the problem n. For example, if an algorithm runs in time

$$T(n) = 56n^4 + 4n^2\log n + n\log n + n^{-2}, \tag{4.8}$$

we say that

$$T(n) \in \mathcal{O}(n^4) = \mathcal{O}(\text{poly}(n)). \tag{4.9}$$

By convention, an algorithm is called efficient if it runs in $\mathcal{O}(\text{poly}(n))$ time steps and needs a memory storage of the same size. On the other hand, if the cost exceeds polynomial scaling (even by a bit), the algorithm is considered as inefficient. In the end, the relevant

Notice that this convention might be unpractical in realistic settings. Consider for example two algorithms, with running times

$$T_1(n) \in \mathcal{O}(n^{10}) \tag{4.10}$$

$$T_2(n) \in \mathcal{O}(n^{\log n}). \tag{4.11}$$

The scaling of $T_2(n)$ is faster than that of $T_1(n)$, hence it is in the long run desirable to use $T_1(n)$. However, for moderate problem sizes, it is reasonable that a worst scaling is preferrable for practical reason. The same argument holds for prefactors, which are not captured by big-O scalings.

Simulability – Simulability is a concept that relates different computational paradigms. If two computing paradigms are universal, for example classical and quantum, then both can perform any arbitrary operation. However, it might be the case that one of the paradigm achieves this operation using less *basic* operations than the other. Depending on how much the difference is, then it will be possible to simulate one system with the other.

In particular, we mention that a quantum computation is simulable with a classical computer if $T_C/T_Q \in \mathcal{O}(\text{poly}(n))$, where n is the number of qubits. These considerations might also depend on the error of the simulation process, thus allowing not for exact computations, but approximate results.

Notice that in principle it should be easy to find quantum computations that are not simulable. Storing a quantum state in a classical computer requires $\mathcal{O}(2^n)$ memory, and any operation will demand similar times to be executed. Assume now that we want to compute a particular property of the quantum state, such as an expectation value or the overlap with respect to a given state. Since this is a limited information, there exist many cases in which partial descriptions of quantum states suffice to extract the information of interest. Examples of this kind are tensor networks [23] or stabilizer states [10].

Hence, it is in general not trivial to assess whether a quantum computation is classically simulable or not, or for which interesting problems there exist classical approaches. The class of computations that are quantumly easy but classically hard are called BQP (Bounded Quantum Polynomial) operations. The most celebrated example of a BQP problem is Shor's algorithm [27], which is capable of solving the problem of factoring an integer number $N = pq$, for p and q primes, and the discrete logarithm problem (DLP), that is, given x, finding y such that $x = g^y \mod N$.

Dequantization – Dequantization is the process of transforming a QML algorithm into a classical algorithm, i. e. to be ran in a *classical* computer, that performs the same or similar task. The classical version of the algorithm does not need to be similar or inspired by the quantum, it only needs to perform the same task.

Dequantization is conceptually related but different from simulability of quantum computing. For a quantum algorithm to be simulable, we need a classical computer capable of performing exactly the same operation. However, for a quantum algorithm to be dequantizable, we just need a classical algorithm that performs the same task, maybe through different methods. Examples of this concept will appear in Section 4.2.2.

The power of (quantum) data – This chapter will cover some examples and basic concdpts of quantum machine learning, together with examples of the problems in a computer. I will use `QIBO` for it.

4.1 Kernels in quantum machine learning

Kernels are most direct example to apply machine learning with quantum computers. For the kernels, consider a data-dependent unitary operation $V(x)$ capable of applying a map to an initial state, that is

$$|\psi(x)\rangle = V(x)\,|0\rangle\,. \tag{4.12}$$

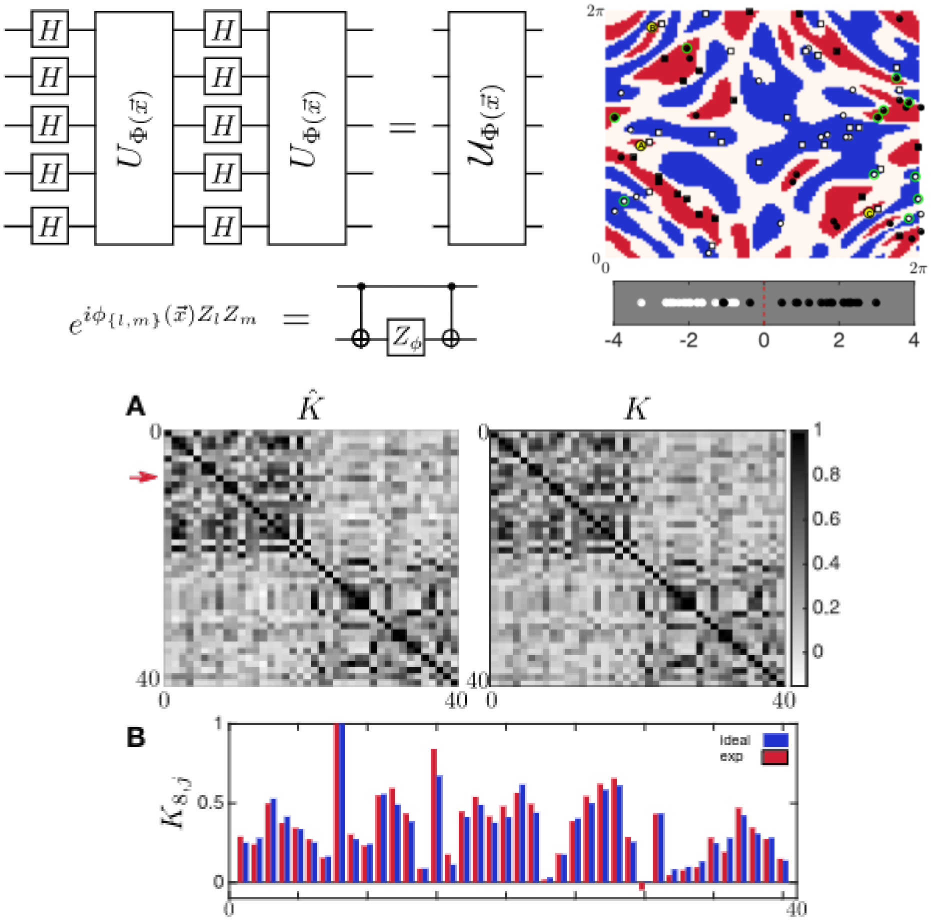


Figure 4.1: Classification problem using a quantum kernel, as seen in [12]. The dataset is generated from the quantum kernel, making it a perfect candidate for using quantum classifiers as the optimal classification tool.

Computing the kernel of this feature map is easy, by just computing

$$k(x_i, x_j) = |\langle\phi(x_i)|\psi(x_j)\rangle|^2 = |\langle 0| V^\dagger(x_i)V(x_j)|0\rangle|^2. \tag{4.13}$$

From this point on, applying QML is equivalent to using a SVM to solve a learning problem, with the modification that the new kernel cannot be (in principle) computed using classical resources.

For an illustrative example, consider this example in Figure 4.1 [12]. The depicted feature map transforms $x \in \mathbb{R}^2$ into a real number $y \in [-1, 1]$. The goal is to find the sign of y. By applying the kernel trick from SVM with this kernel, we can classify this data. Generating this data using classical resources is hard because the considered circuit has properties to avoid classical simulability[1]. Notice however that the role of the quantum computer is not to learn the classification, but to transport the data into a linear subspace where classification is doable with linear separations.

Kernels in QML may suffer from the *curse of dimensionality*. The Hilbert space in which quantum states are embedded is exponentially large. If two states $|\psi\rangle, |\phi\rangle$ are chosen at random, their relative overlap will be $\sim e^{-n}$ with high probability [28]. This means that any kernel matrix used with this kernel will be close to the identity, meaning all points have almost zero overlap with all other points, irrespective of their classes. That being the case, the QML algorithm is effectively capable of learning the training data, but performs randomly on unseen data.

[1] Upon scaling in qubits and specific circuit architectures

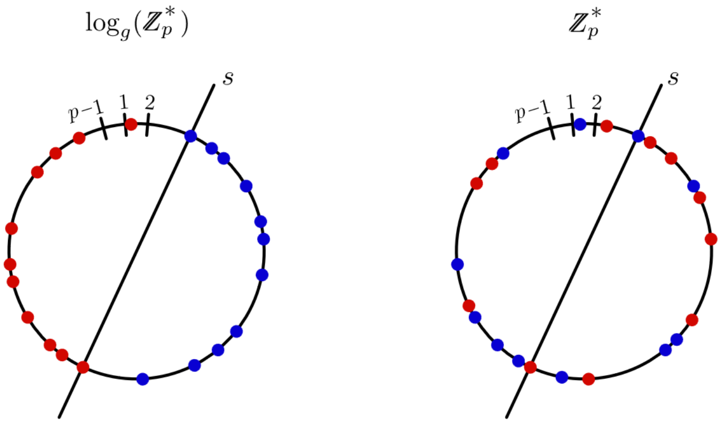


Figure 4.2: Representation of the DLP kernel, used to solve a classification task with provable quantum advantage.

4.1.1 Discrete Logarithm Problem and quantum advantage

The existence of BQP problems can be used to prove that there exist learning problems that cannot be solved with a classical computer [19].

Consider the following problem. Let $\mathbb{Z}_p^*$ be the data space, that is the cyclic group $Z_p^* = \{0, 1, 2, \ldots p-1\}$. Let $n = \lceil \log_2 p \rceil$. For any $s \in \mathbb{Z}_p^*$ define the mapping

$$f_s(x) = \begin{cases} 1 & \log_g x \in [s, s + (p-3)/2] \\ -1 & \text{else} \end{cases} \tag{4.14}$$

This mapping divides the data space into two classes of the same size.

Based on assumptions of the DLP problem, one can make use of the following feature map to use a SVM that is capable of solving the problem. The kernel is given by

$$|\psi_{y,k}\rangle = \frac{1}{\sqrt{2^k}} \sum_{i \in \{0,1\}^k} \left| y g^i \right\rangle, \tag{4.15}$$

for a k to be specified later. The kernel is given by

$$C(y_i, y_j) = |\left\langle \psi_{y_i,k} \middle| \psi_{y_j,k} \right\rangle|^2. \tag{4.16}$$

Assuming that $y = g^x$, the states previously given can be understood as interval states, that is, states that span all states in the interval $[x, x+1, \ldots, x+2^k-1]$. The kernel is thus equivalent to computing intersection of the corresponding intervals.

The DLP mapping admits an interpretation in terms of scrambling. The modular exponentiation transforms $y = g^x$, which is nothing but a hard-to-describe permutation. The inversion is hard to compute classically. By creating the previously defined kernels, we are covering sufficiently many cases of the DLP to ensure that, if two data points correspond to the same class, then the kernel will be sufficiently far from 0, hence a margin classification is possible.

The key element of this problem is that the kernel is efficient to apply only for quantum computers. For one particular element, computing the modular exponentiation is easy. The difficulty arises when exponentially many of these elements are considered simultaneously, as given in this case. Shor's algorithm [27] shows efficiency for this kernels. In particular, it is proven that if this problem is solvable with a classical machine, then there exists a classical algorithm to solve the DLP.

4.2 Variational quantum machine learning

Variational quantum computing witnessed a large growth in the last decade, from its inception [26]. Variational methods consist in applying parameter-dependent quantum circuits $U(\theta)$. These circuits can help with solving

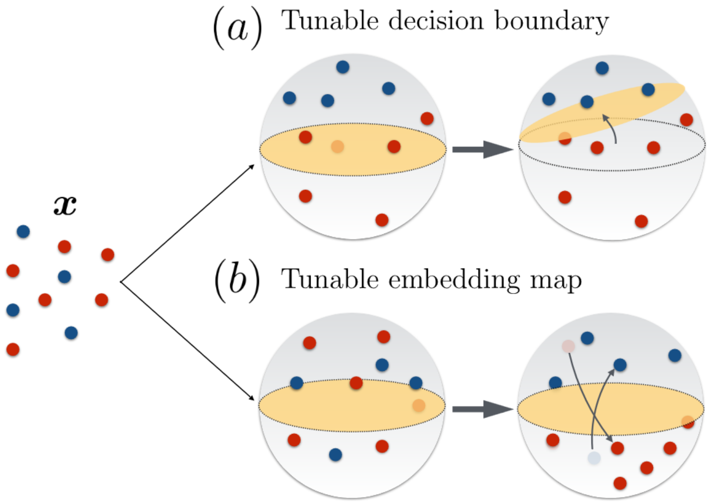


Figure 4.3:

current hardware limitations in state-of-the art prototypes, including but not limited to available number of qubits, tolerance to error or number of operations. For comprehensive reviews, see [**bharti2022noisy**, 6]

The functioning principle of a variational quantum algorithm is as follows. An ansatz of the form $|\psi(\theta)\rangle = U(\theta)|0\rangle$ is used to generate a quantum state encoding the solution of a problem of interest. The problem itself is encoded in a Hamiltonian H. Solving the problem of interest becomes equivalent to finding the ground state of this Hamiltonian, and the optimal parameters are given by

$$\theta^* = \text{argmin}\left(\langle\psi(\theta)| H |\psi(\theta)\rangle\right). \tag{4.17}$$

In general, solving this problem is hard and intractable, belonging to the class QMA [16].

4.2.1 Variational kernel-based quantum machine learning

There exists an alternative interpretation of kernels for QML. The aforementioned feature map can be understood is only a manner to input data into a quantum circuit. Then, one can use an optimized measurement strategy to perform classification. This strategy, which has been extensively used in research, plays the role of solving the optimization problem in a kernel-tricked SVM.

Consider the kernel problem previously mentioned. The feature map can be followed by a tunable quantum operation, and then a fixed observable. In the Heisenberg picture, optimizing the quantum circuit is equivalent to finding the optimal measurement [13]. By the representer theorem, one can rewrite this problem in terms of kernels. This means that the feature map is crucial for the performance of the classification, since it upper bounds the possible distinction between classes even of the optimal measurement strategy.

4.2.2 Data re-uploading for quantum machine learning

One can consider the option of constructing a tunable kernel, so that the feature map is capable of effectively *learning* the data and adapt the transformation from the data to the feature space to the target function to be learnt. A method to achieve this goal is the so-called *re-uploading* scheme. This method is characterized by creating feature maps as

$$U(x, \theta) = \prod_{i=1} V(x) W(\theta_i). \tag{4.18}$$

It is possible to show that the function generated by

$$h_\theta(x) \equiv \langle 0| U^\dagger(x, \theta) H U(x, \theta) |0\rangle \tag{4.19}$$

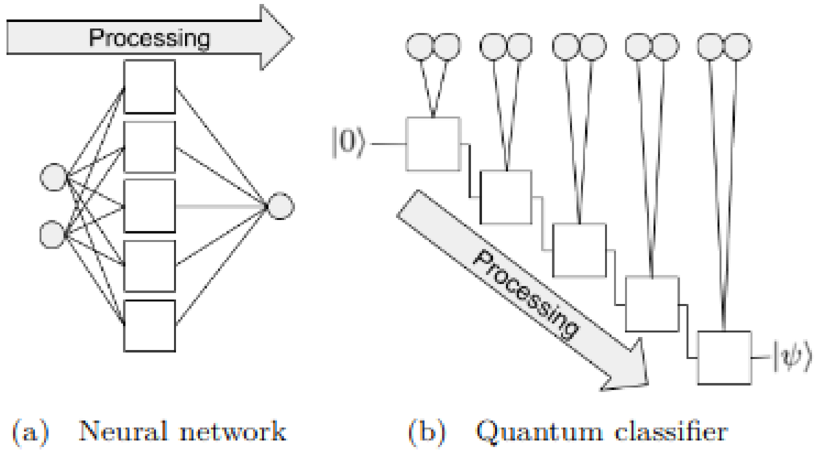


Figure 4.4: Scheme for a re-uploading strategy in a quantum computer, and comparison to a single-layer neural network. In both cases, data is uploaded several times into the pipeline.

satisfy universality theorems in a similar way as in Section 2.1.2, even for single qubit quantum circuits. This provides a *theoretical* capability of these parameter-dependent kernels to solve any kind of learning problems. The interpretation of re-uploading kernels is available in Figure 4.3. It has been recently shown that re-uploading schemes, used for quantum data, can also provide universality in the achieved functions [25], with $\log d$ qubits, being d the dimensionality of the data.

4.2.3 Variational algorithms lack biases

An important issue of variational quantum algorithms is that of *barren plateaus* (BP) [20, 18]. BPs are the phenomenon of exponential concentration in large-dimensional spaces. It has been repeatedly observed that variational ansatzes for general purpose exhibit poor performances on almost any dataset (for large number of qubits). The reason is that variational ansatzes evely explore the Hilbert space. The outcome states are similar to random states, which unavoidably follow exponential concentrations in any property to be computed.

In particular, BPs are defined in the literature for fixed-data (or data-less) problems as

$$\mathrm{Var}_{\theta\sim\Theta} \langle 0| U^\dagger(\theta, x) H U(\theta, x) |0\rangle , \tag{4.20}$$

but it is possible to find similar results by considering variances or averages over x, kernel considerations or concentrations [28, 3].

The take-home message for variational algorithms is to learn from the DLP spirit to solve a problem. We cannot expect QML to work as deep learning, that is, hoping a generic model to learn arbitrary data efficiently. In contrast, we need to rely on specific problems to be solved, and methods that exploit the specific structure of the problem to provide more efficient solutions [1]. Characterizing the gradual transformation from one extreme to the other of the kernel-variational spectrum is still an open question [9].

4.3 Generative modeling for quantum machine learning

Generative modeling for QML is relevant for its fundamental differences with respect to the previously tracted approaches. As seen in **??**, generative modeling relies on creating samples from a model, with the goal of resembling data taken from an unknown probability distribution. In quantum computing, sampling and computing expectation values are commonly known as strong and weak simulation of quantum computing, implying that sampling is harder (and includes) the computation of expectation values. As a reference, the first demonstrations of quantum advantages rely on sampling problems [2, 30], and there exist quantum circuits that permit classical simulation to compute expectation values, but not for sampling. [IQP]

For the reasons stated above, generative modeling must be considered for its possible implications in the findings of quantum advantages. In the following, we will cover the most relevant approaches.

4.3.1 Quantum-circuit Born machines

A Born is defined as a quantum circuit that creates a quantum state, from which bitstrings are sampled. The probability to sample a bitstring x is given by

$$p_\theta(x) = |\langle x| U(\theta) |0\rangle|^2. \tag{4.21}$$

Then, the samples x obtained from the Born machines are compared to the training data, for instance using a Wasserstein distance or a Kullblack-Leibler divergence, to find the configuration of parameters θ that minimizes these distances.

If designed variationally, Born machines encode states that resemble samples from the Haar-random distribution of quantum states. This hinders the training of Born machines, for the difficulty of scaping from those configurations, in analogy to the BP phenomenon.

4.3.2 Expectation value samplers

The expectation value samplers (EVS) are a quantum analogy to classicla models applying arbitrary transformations to random variables, drawn from a simple probability distribution. In these models, we can obtain a set of correlated random variables as

$$x_i = \langle 0| U^\dagger(z,\theta) \hat{O}_i U(z,\theta) |0\rangle, \tag{4.22}$$

for $z \sim \mathcal{N}(0,1)$, for instance. The observables $\hat{O}_i$ label the different features in the sampled random variable. EVS inherit the properties of the feature map used to transform z into an arbitrary random variable [4].

4.3.3 Quantum Boltzmann machines

The quantum Boltzmann machine is analogous to the classical one, but the bits have been replaced with qubits. The state of interest is given by the Hamiltonian defining the Boltzmann machine as

$$\rho = \mathrm{Tr}\left(e^{-H}\right)^{-1} e^{-H}. \tag{4.23}$$

The goal is to optimize H, possibly depending on parameters, such that several measurements of the form

$$x_i = \mathrm{Tr}\left(\rho \hat{O}\right) \tag{4.24}$$

can be used to solve problems of interest and learning from data.

Quantum Boltzmann machines are trainable due to sample complexity [7], thus they are a promising candidate to build trainable models for quantum machine learning.

Bibliography

[1] Scott Aaronson. *How Much Structure Is Needed for Huge Quantum Speedups?* Sept. 2022. DOI: 10.48550/arXiv.2209.06930. arXiv: 2209.06930 [quant-ph].

[2] Frank Arute et al. “Quantum Supremacy Using a Programmable Superconducting Processor”. In: *Nature* 574.7779 (Oct. 2019), pp. 505–510. ISSN: 1476-4687. DOI: 10.1038/s41586-019-1666-5.

[3] Alice Barthe and Adrián Pérez-Salinas. “Gradients and Frequency Profiles of Quantum Re-Uploading Models”. In: *Quantum* 8 (Nov. 2024), p. 1523. DOI: 10.22331/q-2024-11-14-1523.

[4] Alice Barthe et al. *Expressivity of Parameterized Quantum Circuits for Generative Modeling of Continuous Multivariate Distributions.* Feb. 2024. DOI: 10.48550/arXiv.2402.09848. arXiv: 2402.09848 [quant-ph].

[5] R. Byrd et al. “A Limited Memory Algorithm for Bound Constrained Optimization”. In: *SIAM J. Sci. Comput.* (1995). DOI: 10.1137/0916069.

[6] M. Cerezo et al. “Variational Quantum Algorithms”. In: *Nature Reviews Physics* 3.9 (Sept. 2021), pp. 625–644. ISSN: 2522-5820. DOI: 10.1038/s42254-021-00348-9.

[7] Luuk Coopmans and Marcello Benedetti. “On the Sample Complexity of Quantum Boltzmann Machine Learning”. In: *Communications Physics* 7.1 (Aug. 2024), pp. 1–9. ISSN: 2399-3650. DOI: 10.1038/s42005-024-01763-x.

[8] G. Cybenko. “Approximation by Superpositions of a Sigmoidal Function”. In: *Mathematics of Control, Signals and Systems* 2.4 (Dec. 1989), pp. 303–314. ISSN: 1435-568X. DOI: 10.1007/BF02551274.

[9] Elies Gil-Fuster et al. *On the Relation between Trainability and Dequantization of Variational Quantum Learning Models.* June 2024. DOI: 10.48550/arXiv.2406.07072. arXiv: 2406.07072 [quant-ph, stat].

[10] Daniel Gottesman. *The Heisenberg Representation of Quantum Computers.* July 1998. DOI: 10.48550/arXiv.quant-ph/9807006. arXiv: quant-ph/9807006.

[11] Nikolaus Hansen. *The CMA Evolution Strategy: A Comparing Review.* 2006.

[12] Vojtěch Havlíček et al. “Supervised Learning with Quantum-Enhanced Feature Spaces”. In: *Nature* 567.7747 (Mar. 2019), pp. 209–212. ISSN: 1476-4687. DOI: 10.1038/s41586-019-0980-2.

[13] Carl W. Helstrom. *Quantum Detection and Estimation Theory.* Mathematics in Science and Engineering v. 123. New York: Academic Press, 1976. ISBN: 978-0-12-340050-5.

[14] Kurt Hornik. “Approximation Capabilities of Multilayer Feedforward Networks”. In: *Neural Networks* 4.2 (Jan. 1991), pp. 251–257. ISSN: 0893-6080. DOI: 10.1016/0893-6080(91)90009-T.

[15] John Jumper et al. “Highly Accurate Protein Structure Prediction with AlphaFold”. In: *Nature* 596.7873 (Aug. 2021), pp. 583–589. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03819-2.

[16] Julia Kempe, Alexei Kitaev, and Oded Regev. *The Complexity of the Local Hamiltonian Problem.* Oct. 2005. DOI: 10.48550/arXiv.quant-ph/0406180. arXiv: quant-ph/0406180.

[17] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* Jan. 2017. DOI: 10.48550/arXiv.1412.6980. arXiv: 1412.6980 [cs].

[18] Martin Larocca et al. “Diagnosing Barren Plateaus with Tools from Quantum Optimal Control”. In: *Quantum* 6 (Sept. 2022), p. 824. ISSN: 2521-327X. DOI: 10.22331/q-2022-09-29-824. arXiv: 2105.14377 [quant-ph].

[19] Yunchao Liu, Srinivasan Arunachalam, and Kristan Temme. “A Rigorous and Robust Quantum Speed-up in Supervised Machine Learning”. In: *Nature Physics* 17.9 (Sept. 2021), pp. 1013–1017. ISSN: 1745-2481. DOI: 10.1038/s41567-021-01287-z.

[20] Jarrod R. McClean et al. “Barren Plateaus in Quantum Neural Network Training Landscapes”. In: *Nature Communications* 9.1 (Nov. 2018), p. 4812. ISSN: 2041-1723. DOI: 10.1038/s41467-018-07090-4.

[21] J. A. Nelder and R. Mead. “A Simplex Method for Function Minimization”. In: *The Computer Journal* 7.4 (Jan. 1965), pp. 308–313. ISSN: 0010-4620, 1460-2067. DOI: 10.1093/comjnl/7.4.308.

[22] “Fundamentals of Algorithms for Nonlinear Constrained Optimization”. In: *Numerical Optimization*. Ed. by Jorge Nocedal and Stephen J. Wright. Springer Series in Operations Research and Financial Engineering. New York, NY: Springer, 2006, pp. 421–447. ISBN: 978-0-387-40065-5. DOI: 10.1007/978-0-387-40065-5_15.

[23] Román Orús. “A Practical Introduction to Tensor Networks: Matrix Product States and Projected Entangled Pair States”. In: *Annals of Physics* 349 (Oct. 2014), pp. 117–158. ISSN: 0003-4916. DOI: 10.1016/j.aop.2014.06.013.

[24] Fabian Pedregosa et al. “Scikit-Learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12.85 (2011), pp. 2825–2830. ISSN: 1533-7928. URL: http://jmlr.org/papers/v12/pedregosa11a.html.

[25] Adrián Pérez-Salinas et al. *Universal Approximation of Continuous Functions with Minimal Quantum Circuits*. Nov. 2024. DOI: 10.48550/arXiv.2411.19152. arXiv: 2411.19152.

[26] Alberto Peruzzo et al. “A Variational Eigenvalue Solver on a Photonic Quantum Processor”. In: *Nature Communications* 5.1 (July 2014), p. 4213. ISSN: 2041-1723. DOI: 10.1038/ncomms5213.

[27] Peter W. Shor. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer”. In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1484–1509. ISSN: 0097-5397. DOI: 10.1137/S0097539795293172.

[28] Supanut Thanasilp et al. *Exponential Concentration in Quantum Kernel Methods*. Apr. 2024. DOI: 10.48550/arXiv.2208.11060. arXiv: 2208.11060.

[29] Michael M. Wolf. *Mathematical Foundations of Supervised Learning*. 2023.

[30] Han-Sen Zhong et al. “Quantum Computational Advantage Using Photons”. In: *Science* 370.6523 (Dec. 2020), pp. 1460–1463. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.abe8770.