UNIVERSIDAD
DE GRANADA

Escuela Técnica Superior de Ingenierías Informática y Telecomunicación
Facultad de Ciencias

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO DE FIN DE GRADO

Mathematics of Supervised Machine Learning applied to Quantum Computing: addressing the entanglement detection problem

Presentado por:
Ana Martínez Sabiote

Curso académico 2023-2024

Mathematics of Supervised Machine Learning applied to Quantum Computing: addressing the entanglement detection problem

Ana Martínez Sabiote

Ana Martínez Sabiote *Mathematics of Supervised Machine Learning applied to Quantum Computing: addressing the entanglement detection problem*.
Trabajo de fin de Grado. Curso académico 2023-2024.

Responsable de tutorización

Carlos Cano Gutiérrez
Departamento de Ciencias de Computación e I.A.

Antonio Lasanta Becerra
Departamento de Álgebra

Doble Grado en Ingeniería Informática y Matemáticas

Escuela Técnica Superior de Ingenierías Informática y Telecomunicación
Facultad de Ciencias

Universidad de Granada

Declaración de originalidad

D./Dña. Ana Martínez Sabiote

Declaro explícitamente que el trabajo presentado como Trabajo de Fin de Grado (TFG), correspondiente al curso académico 2023-2024, es original, entendido esto en el sentido de que no he utilizado para la elaboración del trabajo fuentes sin citarlas debidamente.

En Granada a 3 de diciembre de 2023

Fdo: Ana Martínez Sabiote

Índice general

Agradecimientos

Llegué a este camino decidida a llenar las páginas inertes, las caras desconocidas y las calles escondidas, en vida. Aún caminando no del todo consciente del movimiento, poco a poco he acabado escribiendo este trabajo de matemáticas e informática que me ha hecho reconciliarme con el camino, en el que descubrí a alguien: era yo.

Gracias a quienes ya estábais desde siempre y para siempre, por ser refugio cuando el mundo ha sido más hostil, estar siempre conmigo a pesar de la distancia y hacer todo posible. Estas palabras se quedan cortas para expresar cuánto agradezco teneros a mi lado.

Gracias a mi compañero de todo, Lorenzo, por ser hogar, creer en mí y apoyarme desde el principio de un largo final. Este logro es también tuyo.

Gracias a quienes he encontrado a lo largo de estos años, dentro de la Universidad y fuera de ella, profesores y amigos con los que andamos el viaje mejor acompañados.

Y gracias a quién me ha visto poquito y ahora no me ve, pero siempre predecía que llegaría aquí.

Summary

This text revolves around supervised machine learning and quantum computing, aiming to study its convergence into quantum machine learning from a theoretical and practical perspective. For the latter, we propose a study of the entanglement detection problem through a quantum machine learning lens. The quantum separability problem addresses detection of quantum entanglement, a fundamental feature of quantum mechanics, and can be formulated as a binary classification problem.

The goals of this thesis are:

1. To provide a solid mathematical understanding of supervised machine learning, quantum computing and their intersection: supervised quantum machine learning.

2. Approaching the quantum separability problem with quantum machine learning methods, including the implementation of the methods and the corresponding experimentation.

The first goal is addressed in chapters 1, 2, 3. The first chapter attempts to build the mathematical fundamentals of supervised machine learning following the three component schema: the data, the model and the learning procedure. It focuses on two models: support vector machines and neural networks. In the second chapter, an exploration into quantum computing begins by introducing the foundational postulates of quantum mechanics, continues with the state and prove of the no-cloning theorem, one of the most important theorems in quantum computing, and finishes with the concept of entanglement (which is at the core of the quantum separability problem) and an introduction to open quantum systems. In the third chapter, the focus shifts towards the realm of quantum machine learning and the study of quantum support vector machines and quantum neural networks, where we delve into their mathematical formulation. These prepare the ground to explore the adaptation and implementation of these quantum models in the following chapter where we apply them to the entanglement detection problem.

The second goal is addressed in chapter 4. The implementation of the proposed methods is carried out in Python using Google Colab as the primary computing environment and the PennyLane quantum computing library, which enabled the integration of quantum computing functionalities. We performed an extensive experimentation to assess the performance of quantum support vector machines and quantum neural networks on the quantum entanglement detection problem with 3×3 bipartite system states.

Finally, the fifth chapter discusses the prospect of the impact quantum computers can wield in the domain of machine learning. Through a comprehensive analysis, it aims to identify the decisive aspects on quantum advantage and future perspectives.

Code and data availability: `https://github.com/anamarsabi/tfg`

Keyworkds: supervised machine learning, quantum computing, quantum machine learning, quantum support vector machines, quantum neural networks, entanglement detection, PennyLane.

Introduction

Machine learning was born at the intersection of statistics, mathematics and computer science. It is considered as the “art” of making computers learn from data. On the other hand, quantum computing is the study of information processing tasks that can be accomplished using a quantum mechanical system. Automatically learning from data and the quantum realm are two fields that generate excitement and are widely researched and also discussed by the general population. Therefore, it is natural to explore the idea of joining both branches: quantum machine learning, a term that has been around for barely 10 years but that has established itself as an active sub-discipline of quantum computing research.

This thesis embarks on a journey through the convergence of machine learning and quantum computing, aiming to unravel the intricacies, explore their synergies and understand the implications they hold. Machine learning is viewed as a toolbox of techniques to learn from data and make reliable predictions. It is a field driven by an empirical approach. Our motivation is to shift the focus from practical to theoretical, focusing on its mathematical fundamentals to understand the working principles that drives this field to success. Quantum computing is built upon the framework of quantum mechanics. Departing with some basics of algebra and functional analysis, we will study the theory of quantum computing to then take a step forward to pass from theory to practice and merge both fields into quantum machine learning. We aim to establish a solid mathematical framework that supports Quantum Support Vector Machines and Quantum Neural Networks, to then apply this techniques to approach the quantum separability problem through a machine learning lens. This problem addresses detection and classification of quantum entanglement, a fundamental feature of quantum mechanics.

There are two main goals for this work:

1. To acquire a solid understanding of supervised machine learning and quantum computing to be able to dive into the field of quantum machine learning from both a mathematical and practical perspective.

2. To approach a machine learning problem with quantum machine learning methods, its implementation and experimentation.

In order to accomplish these goals, this thesis is organized to achieve a series of objectives throughout each chapter:

- **Chapter 1: Mathematical fundamentals of Machine Learning**. Machine learning models are usually conceived as a “black box” that is fed with data and learns to generate reliable outputs. This is one of its most significant criticisms, having led to huge efforts towards Explainable AI (XAI) [AAES+23]. This motivates the next objectives:

    1. To understand what happens in the “black box” of supervised machine learning techniques.

2. To develop the mathematical theory that supervised machine learning is built upon following the three component schema: the data, the model and the learning procedure.
3. To focus on the supervised machine learning methods of support vector machines and neural networks.

The main sources of bibliography for this chapter has been [YSAML12, Jun22, Nie15, SP21]

- **Chapter 2: Introduction to Quantum Computing**. It encompasses our first approach to quantum computing and quantum mechanics, in which we focus on:

4. To research on the arise of quantum computing.
5. To define and understand the theory behind quantum computing from a mathematical perspective, formulating the postulates of quantum mechanics.

We remark the main books [NC10, Sch16] accessed for the writing of this chapter.

- **Chapter 3: Quantum Machine Learning**. We will introduce quantum machine learning, the field of machine learning techniques that involve the use of quantum computing at some phase of the process.

6. To define quantum machine learning approaches and determine which branch we will focus on this thesis.
7. To study the quantum version of support vector machines and neural networks with a strong focus on their mathematical framework.

- **Chapter 4 Case study: entanglement detection**. Its focus is shifted towards computer science: we will face a binary classification problem with quantum machine learning methods. The starting point has been the papers [CDMAK23, USBM23], in which the quantum separability problem (determining if a quantum state is entangled or not) is approached with classical machine learning methods. The objectives set for this chapter are:

8. To get acquainted with PennyLane quantum computing framework.
9. To understand the quantum separability problem. [GT09]
10. To implement and experiment with quantum support vector machines and quantum neural networks applied to our chosen problem.
11. To eventually compare the outcomes from classical machine learning with those obtained with our quantum models.

- **Chapter 5: Feasibility of Quantum Machine Learning problems**. We will assess the practical implications of Quantum Machine Learning, considering its potential and challenges. Our aims for this chapter are:

12. To explore the advantages of quantum machine learning against classical machine learning.
13. To overview future perspectives.

The core of this study on quantum machine learning of chapters 3 and 5 relies on [SP21, CGCDM23].

1. Mathematical fundamentals of Machine Learning

1.1. Introduction

Machine learning is a field closely related to both computer science and statistics. It initially evolved as a sub-field of research within the broader domain of artificial intelligence. In an informal way, we can say machine learning consists of computer algorithms that generalise knowledge from patters found in data to make predictions in unknown situations, in other words: "learning". This is a skill that becomes natural to humans in many aspects of life throughout history. We can think from some tribes or farmers understanding of weather, all the way to doctors; all of them have acquired a level of expertise on specific fields as a result of many iterations of observing meaningful indicators and their connection to forthcoming occurrences. Additionally, this knowledge passes from generation to generation whether verbally or documented. However, the amount of data a single human can process will always be limited since the process of learning is constrained by time, economic factors and complexity of the domains. There is where the potential of machine learning lies. This field is attractive and revolutionary because it provides us the ability to learn from extremely large data sets whose patterns tend to be very complex. Humans not only cannot handle the big dimensions of the dataset but also cannot really understand the mechanisms of the system to be able to model it with equations. Instead, machine learning assumes a general mathematical model which is then tuned to the specific problem using the provided data. The resulting model produces reliable predictions although it usually does not show the relationships present in the data that induce that behaviour or output. This is why machine learning techniques are usually conceived as a "black box" that is fed with data and learns to generate reliable outputs. This is one of the strengths of machine learning: its tools can be applied to a wide range of tasks and different domains. However, one of the most significant criticisms is precisely the opacity of the techniques, since it is hard to follow the decision making process of a model that is not formulated carefully to capture the essence of a mechanism as in physics or statistics. For example, data on demographic, medical and social context are quite risky to model since we have limited understanding of the input/output process. Image classification also presents this challenge, where the objective is to assign labels to images based solely on their digital representation. Understanding what makes an image a dog or a cat proves to be a complex procedure. Nevertheless, in the context of classification, rather than understanding the intricacies of the labeling process, the goal is to replicate it.

In this introduction we have broadly mentioned that "the model learns from data". Next, we will dive deeper into this statement that comprises the most important components of a machine learning problem: the model, the data and the learning procedure. We will try to understand what happens in the "black box" (Figure 1.1) with a mathematical study of machine learning techniques.

Input X
Black box
Output Y

Figure 1.1.: Machine learning blackbox paradigm whose goal is to reproduce input/output pairs based on prior observations.

1.2. Supervised learning

First, we will present the two fundamental learning problems that are faced in the context of machine learning: supervised and unsupervised. The problem of supervised learning can be described in the following formulation:

Definition 1.1 (Supervised learning problem)**.** Let $\mathcal{X}$ be an input domain, $\mathcal{Y}$ be an output domain and $\mathcal{D} = \{(x_1, y_1), ..., (x_M, y_M)\} \subset \mathcal{X} \times \mathcal{Y}$ be a dataset that contains inputs x_i and target outputs y_i sampled from a probability distribution $p(x, y)$. Given a new unclassified input $x \in \mathcal{X}$, guess or predict the corresponding output $y \in \mathcal{Y}$

In the context of unsupervised learning discussed here, the data samples lack labels, and the objective is to draw similar samples.

Definition 1.2 (Unsupervised learning problem)**.** Let $\mathcal{X}$ be an input domain and $\mathcal{D} = \{x_1, ..., x_M\} \subset \mathcal{X}$ be a dataset of input samples drawn from a probability distribution $p(x)$. The problem consists on drawing a new sample from p.

There are other tasks in unsupervised learning such as clustering and dimensionality reduction.

This thesis will be constrained to the case of supervised learning tasks. The standard procedure in supervised learning can be summarized in the following steps:

1. Choose a model family $\{f\}$, that is, a set of trial functions that map inputs from $\mathcal{X}$ to outputs from $\mathcal{Y}$. They usually depend on a set of parameters θ. Each set θ defines a particular model in the family $\{f_\theta\}$. Next we will proceed to pick the best model to reproduce the data.

2. This task is formally known as risk minimisation. This process uses a loss function $L(f(x), y)$ which we can understand as a measure of the discrepancy between the model's prediction $f(x)$ for an input $x \in \mathcal{X}$ and its label $y \in \mathcal{Y}$.

3. Once we can compute the loss function, we need to choose an optimization algorithm to minimize the average loss over all the data depending on the parameters. Popular choices are gradient descent and stochastic gradient descent.

The difficulty in machine learning lies in the task of selecting the most suitable model without having access to all the data, but instead only having a finite set of examples. Consequently, the essence of learning lies in choosing a model that generalises well from these limited examples to the entire data domain.

As we can see, the model, the data and the learning procedure (which corresponds to the last two steps) are the main components of supervised learning.

1.2.1. Data

Data is the starting point of machine learning and its fuel. The dataset of a supervised learning problem $\mathcal{D} = \{(x_1, y_1), ..., (x_M, y_M)\} \subset \mathcal{X} \times \mathcal{Y}$ is sampled from a joint distribution $p(x, y)$. The set of data samples is taken independently and identically distributed, shortened iid. This is a strong assumption that means that any sample does not depend on the value of the others and has been drawn from exactly the same distribution. This does not really correspond with real life situations but facilitates the learning theory built upon it. Learning in non-iid scenarios remains an open research question.

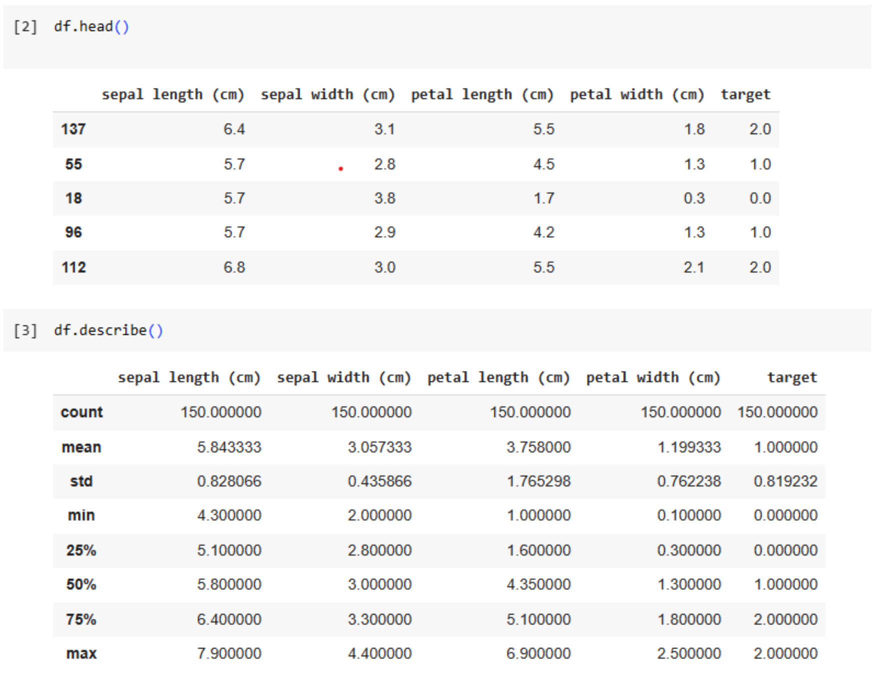
We are abstractly talking about data, although what it really is? A **dataset** is a collection of **data samples**. Each data sample is a description of an event, an object or the item in question. This description is a collection of **attributes** or features and their values, which can be qualitative or quantitative, are called **attribute values**. Figure 1.2 depicts an example of a dataset.

Let $\mathcal{D} = \{(x_1, y_1), ..., (x_M, y_M)\} \subset \mathcal{X} \times \mathcal{Y}$ be a dataset containing M samples: $x_i \in \mathcal{X}$ and its corresponding label $y_i \in \mathcal{Y}$. Each data sample $x_i = (x_{i1}, ..., x_{iK}) \in \mathcal{X}$ is a vector of attributes in the K-dimensional input space $\mathcal{X}$ and x_{ij} is the value of the jth attribute of the sample x_i. Regarding the output space, it determines an important distinction in the type of supervised learning problem to be solved. If $\mathcal{Y}$ is a set of discrete class labels $\{l_1, ...l_R\}$, we are dealing with a **classification** problem. In the particular case of $\mathcal{Y} = \{l_1, l_2\}$ consits on two labels, then it is called a **binary classification** problem. If the output space is continuous, such as the space of real numbers $\mathbb{R}$ or an interval therein, then it is a **regression** problem, that is, the prediction is continuous-valued.

It is desirable to have a data input space $\mathcal{X}$ in $\mathbb{R}^K$, but sometimes raw data is not from a numerical domain and it is needed to first find a suitable quantitative representation of the attributes. There are different techniques depending on the type of data, such as text or images. We cannot go into greater depth because it is a field beyond the scope of this thesis.

There is vital step for any machine learning application: **data pre-processing**. This practice impacts the output of the machine learning algorithm and thus, improves its predictive power. Real world data is usually incomplete, dirty and inconsistent, hence data preprocessing techniques are needed to improve the accuracy and efficiency of the machine learning model used. Dirty data include missing data and wrong noisy data . The presence of a high proportion of the data is dirty, implies that the outcome of applying a machine learning algorithm will likely result in an unreliable model. So first, the data must be cleaned to remove or repair dirty data. Then, a series of data transformations are performed because the data collected in a dataset may not be useful enough, that is, the original attributes may have a meaning in its domain but are not convenient to obtain accurate predictions. There are a series of available transformations to change the original attributes or to generate new attributes with better properties. We can point out data normalization, polynomial transformations, one-hot-enconding (to transform categorical data), dimensionality reduction techniques, among others. For a more in-depth exploration of this topic, we would recommend consulting the book [GLH15], which provides comprehensive coverage and valuable insights on the subject of data pre-processing.

Finally, we have to mention the routinary division of the dataset into three subsets called: training, validation and test. The process of "learning" of the ML algorithm is also well-known as training. As we have mentioned before, learning is equivalent to generalising well. The training set is used to minimise the cost function (the average of the loss values coming from all data samples). The validation set is used to estimate the performance after



```
[2]  df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
137	6.4	3.1	5.5	1.8	2.0
55	5.7	2.8	4.5	1.3	1.0
18	5.7	3.8	1.7	0.3	0.0
96	5.7	2.9	4.2	1.3	1.0
112	6.8	3.0	5.5	2.1	2.0

```
[3]  df.describe()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333	1.000000
std	0.828066	0.435866	1.765298	0.762238	0.819232
min	4.300000	2.000000	1.000000	0.100000	0.000000
25%	5.100000	2.800000	1.600000	0.300000	0.000000
50%	5.800000	3.000000	4.350000	1.300000	1.000000
75%	6.400000	3.300000	5.100000	1.800000	2.000000
max	7.900000	4.400000	6.900000	2.500000	2.000000

Figure 1.2.: Example of a dataset. It corresponds to the scikit-learn Iris Dataset.
This data sets consists of 3 different types of irises' (Setosa, Versicolour, and Virginica) thus, define a three label classification problem.
There are four attributes: petal and sepal length and width.

training in order to adapt hyperparameters of the algorithm. The model is implicitly fitted to the validation set because we discard those models that did not perform well on this data. Once the model is fully specified, the test set is used for the final evaluation of the model's generalization ability to provide an unbiased evaluation of the final model. The separation of data into these three partitions is really important for obtaining a good model: one that can generalize well to unseen instances, avoiding overfitting.

1.2.2. Model

The second main component of machine learning is a model, or precisely, a model family. The goal is to obtain a model that fits the data and produces reliable outputs. In mathematics, a model is a map from an input space $\mathcal{X}$ to an output space $\mathcal{Y}$, in other words, a function that reproduces the pattern in the data.

$$\mathcal{X} \longrightarrow \mathcal{Y}$$

In the context of computation, this mapping is achieved through an algorithm. At a higher level of abstraction, a model sets the hypothesis or rules that leads from input to output. Therefore, models in machine learning are functions, algorithms or rules that establish a connection between input data and output, depending on the point of view we fix. We will continue with the mathematical formulation:

Definition 1.3 (Deterministic model)**.** Let $\mathcal{X}$ be an input domain and $\mathcal{Y}$ an output domain for a supervised machine learning problem. We define a deterministic model as a function

$$f : \mathcal{X} \longrightarrow \mathcal{Y}, \quad f(x) = y, \quad x \in \mathcal{X}, y \in \mathcal{Y}$$

Usually, this model depends on a set of parameters θ and hence defines a model family $\{f_\theta\}$

Depending on the output domain $\mathcal{Y}$ we can distinguish between deterministic models for regression and classification tasks.

Theoretically, machine learning methods could use any possible mapping function

$$f : \mathcal{X} \longrightarrow \mathcal{Y}$$

to make predictions of a label $y \in \mathcal{Y}$ by computing $\hat{y} = f(x)$. The collection of all possible maps from the input space to the output domain is denoted by $\mathcal{Y}^{\mathcal{X}}$. To obtain the best model that fits our data, we have to search over this set $\mathcal{Y}^{\mathcal{X}}$. However, it is usually far too extensive for practical machine learning methods to fully explore it. As an example, if we consider a regression problem with dataset in which every sample is only characterized by an real-valued attribute and a real-valued label, then $\mathcal{Y}^{\mathcal{X}}$ would be the collection of all real functions of real variable. Instead, the problem would get simplified if we search for the best model in a subset of the real functions of real variable such as linear functions. In that case we are tackling a regression problem with a linear regression algorithm. In general, practical machine learning methods search and evaluate only a small subset of all possible maps $\mathcal{Y}^{\mathcal{X}}$. This subset of computationally feasible maps is what we referred as model family (denoted before as $\{f\}$, also called hypothesis space $\mathcal{H}$. The selection of a model family for a machine learning problem is a design choice which involves a tradeoff between computational complexity and the statistical properties of the resulting machine learning method. This choice takes into account the computational resources at the disposal of a machine learning method. Different computational infrastructures tend to align with different hypothesis spaces. For instance, deep learning methods deployed in a cloud computing environment typically use much larger hypothesis spaces derived from deep neural networks. While deep learning methods would be an overkill for a small embedded system, which would perform much better with a linear hypothesis space.

In a general sense, the design choice regarding the model family of a machine learning method has two main demands to be balanced:

- The model family as to be sufficiently large so that it contains at least one accurate predictor map $\hat{f} \in \{f\}$. If the model family is too small, it may not encompass a predictive mapping necessary to capture the (potentially complex and nonlinear) relationship between inputs and outputs.

- It has to be sufficiently small such that its processing fits the available computational resources including memory, bandwidth and processing time. At the same time, we must be able to search efficiently to find a good model. Constraining the size of the hypothesis space is also related to the prevention of overfitting since in a large model family we might, by chance, stumble upon a function that almost perfectly predicts the labels of the training set while showing poor generalization power for other data.

1.2.3. Learning procedure

A machine learning method has to select the best model from the model family of all computationally feasible predictor maps. This raises the question: which model is the best model out all the possible ones? We need a measure of quality of a model, that is, how close are the model predictions to the target labels. There is where the concept of **loss function** appears.

Definition 1.4 (Loss function)**.** A loss function is a map

$$\begin{aligned} L : \mathcal{X} \times \mathcal{Y} \times \mathcal{H} &\longrightarrow \mathbb{R}^+ \\ ((x,y),f) &\longmapsto L(f(x),y) \end{aligned}$$

which assigns a pair consisting of a input $x \in \mathcal{X}$, its label $y \in \mathcal{Y}$ and a model f to the non-negative real value $L(f(x),y)$ named loss value.

The loss value $L(f(x),y)$ measures the difference between the true label y and the prediction by the model $f(x)$. Now that we have formulated this concept, we can rephrase the basic procedure of machine learning to find a model out of all the model family that minimizes the loss $L(f(x),y)$ for all data inputs.

The loss function is another design choice for the machine learning technique. This decision should take into account the computational complexity for finding a minimum loss (benefiting differentiable and convex functions), robustness to outliers, statistical aspects and interpretability. There exist numerous loss functions, and perhaps the simplest one for classification tasks relies on a model's accuracy, which measures the proportion of correctly classified examples:

$$\text{accuracy} = \frac{\text{number of samples classified correctly}}{\text{total number of samples}}$$

The corresponding loss would be the error:

$$\text{error} = 1 - \text{accuracy}$$

Most machine learning methods use a continuous-valued loss function, since differentiability is desirable to compute gradients. One of the most popular loss functions is the norm in l_2 or squared Euclidean distance which is known as **mean-squared-error loss**

$$L(f(x), y) = (f(x) - y)^2$$

The norm in l_1 also provides us another loss function:

$$L(f(x), y) = |f(x) - y|$$

Other options are the *hinge loss*

$$L(f(x), y) = \max(0, 1 - yf(x))$$

and the *logistic loss*

$$L(f(x), y) = \log(1 + e^{yf(x)})$$

The loss function is the starting point of the **risk minimisation** process that consists on minimising the expected or average loss (called risk) of a model over the data distribution.

Definition 1.5 (Risk)**.** Let the inputs $x \in \mathcal{X}$ and outputs $y \in \mathcal{Y}$ to a supervised learning problem be sampled from a distribution $p(x, y)$. The **expected loss** or **risk** of a model f with loss L is defined as

$$\mathcal{R}_f = \mathbb{E}[L_f] = \int_{\mathcal{X}\times\mathcal{Y}} L(f(x), y)p(x, y)dx\,dy$$

Note that the integral runs over all possible data pairs and thus, the expectation is taken over the data distribution.

The risk provides an indication of the model's performance across the entire data domain, with a lower risk indicating better performance. The goal to solve a supervised learning problem will be to minimise the risk, although the integral over all possible data pairs is impossible to compute in practice, so we need to estimate the risk over the M data samples we are provided in a dataset. This estimation is called **empirical risk**.

Definition 1.6 (Empirical risk)**.** Consider the dataset $\mathcal{D} = \{(x_1, y_1), ..., (x_M, y_M)\} \subset \mathcal{X} \times \mathcal{Y}$. The empirical risk estimates the risk over the dataset as the average loss:

$$\hat{\mathcal{R}}_f = \hat{\mathbb{E}}[L_f] = \frac{1}{M}\sum_{i=1}^{M} L(f(x_i), y_i)$$

This estimation is warranted because of the law of large numbers, departing from the assumption that we consider the dataset to be independent and identically distributed variables from the distribution $p(x, y)$. Therefore, the sample average converges to the expected value if the number of samples M of the dataset $\mathcal{D}$ is sufficiently large.

In practice, we select a model out of the model family minimising the empirical risk on the given dataset, that is, finding the model f^* such that

$$f^* = \min_{f \in \mathcal{H}} \hat{\mathcal{R}}_f$$

In parametrised model families, model selection translates to optimising the parameters. Therefore, the empirical risk minimisation can be reformulated as finding the parameter θ^* such that

$$\theta^* = \min_{\theta} \hat{\mathcal{R}}_{f_\theta}$$

Machine learning should solve the empirical risk minimisation problem without overfitting: there is an optimal solution of the empirical risk over the training data that performs poorly on new data. There are different strategies to prevent the model from overfitting: **regularisation techniques**. The most common one is to add a regularisation term $g(f_\theta)$ to the empirical risk and minimise the cost function

$$C(\theta) = \hat{\mathcal{R}}_{f_\theta} + g(f_\theta)$$

These regularisation terms penalize the model and impose additional constraints, for example the l_1 regulariser

$$g_{l_1}(\theta) = \sum_i |\theta_i|$$

benefits parameters with a small absolute value, while the l_2 regularisation term

$$g_{l_2}(\theta) = \sum_i \theta_i^2$$

adds preference to parameters with a small l_2 norm.

Once we have formulated the cost function, the learning procedure consists on an optimisation problem. While there is an extensive theory and algorithms for convex optimisation problems, machine learning tasks usually fall into non-convex optimisation problems for which there is significantly less information available. The popular choice to tackle this sort of non-convex optimisation are iterative searches that progressively calculate better candidates for the parameters of a model. In situations where there are multiple parameters and optimization occurs within high-dimensional spaces, we need to understand the landscape of the cost function. The main source of information in such cases is gradients: a vector of partial derivatives of a function with respect to its inputs. The gradient provides the direction of the most significant incline in the cost landscape.

Gradient descent is an iterative algorithm for finding the minimum of a differentiable function. The general idea of this algorithm is to update the parameters θ of a cost function $C(\theta)$ iteratively towards the direction of steepest descent in order to minimize a cost function. We can think of this situation in a real world scenario: suppose you are lost in the mountains blindfolded, you can only feel the slope of the ground below your feet. Your goal is to reach to the lowest point going downhill in the direction of the steepest slope.

This is mathematically expressed as

$$\theta_{t+1} = \theta_t - \eta \nabla C(\theta_t)$$

where η is a parameter called learning rate that determines the step size at each iteration and t is an integer that corresponds to the current iteration. Figure 1.3 illustrates this algorithm

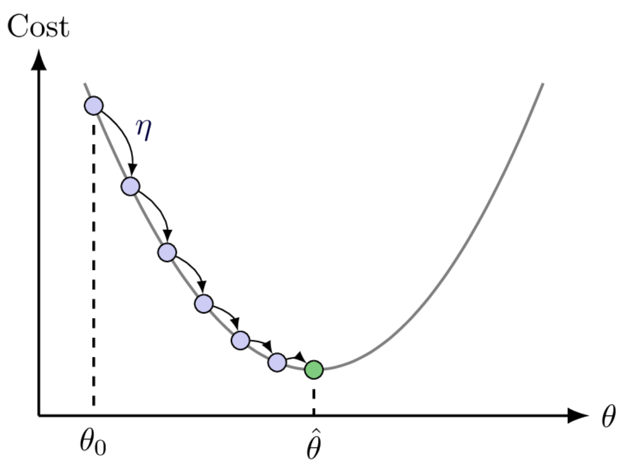


Figure 1.3.: Gradient descent

for a really simple scenario.

If the learning rate is too small, the algorithm will have to perform many iterations to converge. Whereas if the learning rate is too high, you might jump across the valley and end up on the other side, perhaps even at a higher point than in the previous step and eventually the algorithm may diverge. This situations are depicted in Figure 1.4.

The formula above of the gradient descent algorithm performs the purpose of the process: the gradient $\nabla C(\theta_t)$ always points towards the ascending direction in the landscape of the cost function C, thus $-\nabla C(\theta_t)$ points to the opposite direction, that is, the descending direction. The initial set of parameters θ_0 is randomly initialized. Then, it is gradually improved, at each step attempting to decrease the cost function.

In real case scenarios, cost functions may differ very much from the previous illustrations. There may be holes, plateaus, ridges and all sorts of irregular terrains, making convergence to the minimum very difficult.

There are several variations of the gradient descent algorithm such as batch gradient descent, stochastic gradient descent and mini-batch gradient descent. [Gér22]

1.2.4. Evaluation metrics

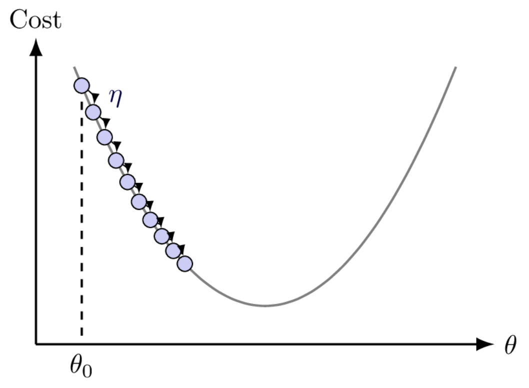
After training and finding a model that performs sufficiently well, it is time to evaluate the final model on the test set. First, we need to apply the model to the test set to obtain the predictions. Then, having the predictions and the labels we can apply different metrics to evaluate the performance of the model.

There are many performance measures available. In the context of classification, we will consider the following ones:

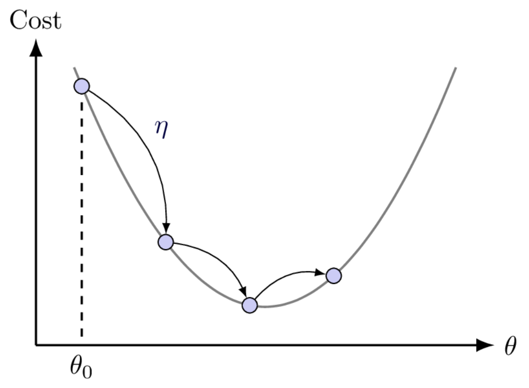
- **Accuracy.** This metric represents the ratio of correct predictions to the total number of samples.

$$\text{accuracy} = \frac{\text{number of samples classified correctly}}{\text{total number of samples}}$$

It gives us a general idea of the classification power of our model, although we have to



(a) Gradient descent algorithm taking a η learning rate too small



(b) Too big η learning rate for gradient descent

Figure 1.4.: Effect of different learning rates for gradient descent algorithm

	PP	PN
P	TP	FN
N	FP	TN

Table 1.1.: Confusion matrix of a binary classification problem.

take into account that it is not a good metric in the case of a dataset with unbalanced classes since it is "blind" to different classes. To understand it, let's consider a dataset with 100 samples, 97 of them are labeled 0 and 3 correspond to label 1. If our model predicts label 0 for all samples, the accuracy would be of 97% which does not represent the actual lack of classification power of the model.

- **Confusion matrix.** A confusion matrix is a table used to evaluate the performance of an algorithm. Each row corresponds to a label and each column represents a predicted label. Therefore, if n is the number of labels of the dataset, the confusion matrix is a $n \times n$ table that shows the correct and incorrect predictions made by the model compared with the actual labels.

  In the case of a binary classification problem we have the following confusion matrix, considering P is the number of positive labels (1) and N is the number of negative labels (0).

  In figure Table 1.1, TP stands for true positive, FP - false positive, TN - true negative and FN - false negative. These values are used to compute other metrics.

- **Precision.** It is the ability of a model not to label as positive a sample that is negative.

$$\text{Precision} = \frac{TP}{TP + FP}$$

  It is useful for unbalanced datesets, since the precision is inversely proportional to the number of false positives.

- **Recall.** It is the ability of the classifier to find all the positive samples.

$$\text{Recall} = \frac{TP}{TP + FN}$$

- F_1 **score.** It can be interpreted as the harmonic mean of the precision and recall.

$$F_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{2TP}{2TP + FP + FN}$$

  This metrics takes values in the interval $[0, 1]$. The higher F_1, the better is the performance of the model.

- **ROC curve.** It stands for Receiver operating characteristic curve. It is a graph that shows the performance of a binary classifier as its discrimination threshold (the threshold for deciding whether a prediction is labeled positive or negative) is varied. It represents the tradeoff between the TPR = true positive rate (fraction of true positives out of all the positives) and the FPR = false positive rate (fraction of false positives out of all the negatives). A perfect model would have high value of true positive rate and low value of false positive rate.

- **AUC.** It computes the area under the ROC curve. This metric summarizes the curve information in one number. AUC has a range of [0,1]. The greater the value, the better is the performance of our model.

Up to this point, we have observed that the objective in machine learning is to choose a model that reduces a specific cost function which depends on data samples of a particular task, with the ultimate goal of minimizing the cost when applied to unseen data. If the model is parametrised, the selection process involves optimizing or "training" these parameters minimise the cost.

There are many methods in machine learning, that is, model function families that can be used for prediction and a training strategy for how to use the data to construct a specific model that generalises from it. In the following sections, we will introduce two classical machine learning methods: support vector machines and neural networks, that will become important in the context of quantum machine learning in the following chapters.

1.3. Support vector machines

Support vector machine (SVM) is a supervised learning algorithm for binary classification. In particular, SVM are one of the most well-known kernel methods. Kernel methods work out machine learning problems based on the concept of measuring similarity between data points. This similarity measure is expressed by a **kernel**.

1.3.1. Kernel methods

Definition 1.7 (Kernel). Given a data domain $\mathcal{X}$, a kernel is a positive semi-definite bivariate function $\kappa : \mathcal{X} \times \mathcal{X} \longrightarrow \mathbb{R}$.

Mercer's theorem provides the following useful characterization of kernels.

Theorem 1.1. *The function $\kappa : \mathcal{X} \times \mathcal{X} \longrightarrow \mathbb{R}$ is a Mercer kernel function (also known as kernel function) where:*

1. *it is symmetric: $\kappa(x, y) = \kappa(y, x)$, $\forall x, y \in \mathcal{X}$,*
2. *its kernel matrix, that is, the kernel applied on a set of data points of the input space $\mathcal{D} = \{x_1, ..., x_M\}$ forms a $M \times M$ Gram matrix: $\mathcal{K}(i, j) = \kappa(x_i, x_j), \quad \forall i, j \in \{1, ..., n\}$ is positive semi-definite.*

A great part of the theory of kernel methods relies on **"the kernel trick"**: the fact that kernel functions can be represented as inner products of data that has been mapped from its original space $\mathbb{R}^n$ into a higher dimensional space $\mathbb{R}^N$ ($n < N$), in which machine learning algorithms can solve problems by the use of inner product. This higher dimensional space receives the name of feature space $\mathcal{F}$. The potential of this idea is that in a suitable feature space we would obtain linearly separable data by a hyperplane. The function that takes original data inputs into the feature space is called **feature map** $\phi : \mathcal{X} \longrightarrow \mathcal{F}$.

Thus, we just need to compute:

$$\kappa(x, y) = \langle \phi(x), \phi(y) \rangle_{\mathcal{F}}$$

In turn, a model is expressed in terms of a kernel function. We obtain different models by replacing the kernel functions.

Support vector machine provides a decision boundary (a hyperplane in the feature space) separating the considered classes such that its distance to each class is as large as possible. This idea derives from their initial introduction as maximum-margin classifiers, which advocates that during training, the goal is to maximize the separation between the decision boundary of a classifier and the training data points.

An introduction to maximum-margin classifiers will help us to understand better the working principle of support vector machines.

Let's consider a binary classification problem with an input domain $\mathcal{X} = \mathbb{R}^n$ and output domain $\mathcal{Y} = \{-1, 1\}$.

We assume that our data is linearly separable by a hyperplane, which means that there is a normal vector $\vec{w} \in \mathbb{R}^n$ and some constants $b \in \mathbb{R}$ which define a hyperplane $\vec{w} \cdot \vec{x} + b = 0$ that separates the training data so that points of the two different classes are on either side of the hyperplane. We will denote this hyperplane as H. We can determine to which side of the hyperplane our data $x \in \mathcal{X}$ is in terms of the sign of $\vec{w} \cdot \vec{x} + b$.

We take a linear model as this hyperplane and simplify the vectorial notation

$$f_w(x) = w^T x + b$$

Linear separation can be expressed by the restriction $f_w(x_i) > 0$ for data points labeled $y_i = 1$, and $f_w(x_i) < 0$ for inputs of class $y_i = -1$. According to this, linear separation can be summarised in the following condition

$$f_w(x_i) y_i > 0 \quad \forall i = 1, ..., M$$

There are infinite number of separating hyperplanes, but we are looking for the one that would maximize the distance to the training datapoints and therefore, obtain a classifier with a low risk because we expect unseen data to follow a similar distribution to the one that we have seen in the training data, that is, new samples of one class will be close to training samples of that class. Support vector machine looks for the separating hyperplane that maximises the distance between the closest training points and itself. This distance is known as *margin*.

Let's consider p_0 the closest data point to H and consider the hyperplane parallel to H that contains p_0. This hyperplane and its reflection define the margin of H and can be represented by the equations $\vec{w} \cdot \vec{x} + b = \pm C$ for some constant C. We can divide the expression by C to obtain the margin hyperplanes:

$$\hat{\vec{w}} \vec{x} + \hat{b} = \pm 1$$

where $\hat{\vec{w}} = \vec{w}/C$ and $\hat{b} = b/C$.

The separating hyperplane H can be represented by

$$\hat{\vec{w}} \vec{x} + \hat{b} = 0$$

We will denote the margin hyperplanes as H_1 and H_{-1} accordingly.

We are working under the assumption that there are no points inside the margin. Studying the sign of $\hat{w}\vec{x_0} + \hat{b}$ for $x_0 \in \mathbb{R}^n$ we can determine to which region it belongs:

- If $\hat{w}\vec{x_0} + \hat{b} = 0$, then x_0 is in the hyperplane H.
- If $0 < \hat{w}\vec{x_0} + \hat{b} < 1$, then x_0 is in the margin between the hyperplanes H and H_1.
- If $-1 < \hat{w}\vec{x_0} + \hat{b} < 0$, then x_0 is in the margin between the hyperplanes H and H_{-1}.
- If $\hat{w}\vec{x} + \hat{b} = \pm 1$, then x_0 is in one the hyperplanes that defines the margin, H_1 or H_{-1} respectively.

The condition of linear separation with margin can be expressed with the constraint:

$$f_w(x_i)y_i > 1 \quad \forall i = 1, ..., M$$

Let's remember that our goal is to maximize the margin while preserving the condition of linear separation with margin that we have derived above. Now we will deduce what is the expression of the margin, which is the distance between the hyperplanes H_1 and H_{-1}.

First we will see what is the distance between H and H_1: the length of the only vector in the direction normal to H_0 that connects $p \in H_0$ to a point in H_1. Since $\hat{w}$ is normal to H_0, the desired vector needs to be $\alpha\hat{w}$ for some scalar α. Therefore, $p + \alpha\hat{w} \in H_1$ and it must fulfill its equation:

$$\hat{w}(p + \alpha\hat{w}) + b = 1$$
$$\hat{w}p + b + \alpha||\hat{w}||^2 = 1$$

Since $p \in H_0$, we have that $\hat{w}p + b = 0$. We can obtain the scalar α from the expression above

$$\alpha||\hat{w}||^2 = 1 \Longleftrightarrow \alpha = \frac{1}{||\hat{w}||^2}$$

The length of $\alpha\hat{w}$ would be

$$||\alpha\hat{w}|| = |\alpha| \cdot ||\hat{w}|| = \frac{1}{||\hat{w}||^2}||\hat{w}|| = \frac{1}{||\hat{w}||}$$

Therefore, the distance between H_1 and H_{-1} is by symmetry the double of the distance between H and H_1, that is $\frac{2}{||\hat{w}||}$.

The problem of finding a hyperplane that separating the data maximizes the margin can be posed as the following optimization problem

$$\begin{aligned} \text{Maximize:} \quad & \frac{2}{||\hat{w}||} \\ \text{subject to:} \quad & f_w(x_i)y_i > 1 \quad \forall i = 1, ..., M \end{aligned}$$

The above optimisation problem of finding the maximum margin becomes equivalent to the next minimization problem

$$\begin{aligned} \text{Minimize:} \quad & ||\hat{w}|| \\ \text{subject to:} \quad & f_w(x_i)y_i > 1 \quad \forall i = 1, ..., M \end{aligned}$$

Alternatively, it is possible to minimise $\frac{1}{2}||\hat{w}||^2$ under the same constraint, leading to the same solution but easier to solve. Now we are going to briefly present the dual SVM problem which is equivalent to

$$\begin{aligned} \text{Minimize:} \quad & \frac{1}{2}||\hat{w}||^2 \\ \text{subject to:} \quad & f_w(x_i)y_i > 1 \quad \forall i = 1, ..., M \end{aligned}$$

This is a quadratic optimization problem with $n+1$ variables ($\hat{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$) and M constraints. It is computationally difficult to solve when n is large. The dual problem will also be a quadratic optimization problem but with M variables and $M+1$ constraints. The solution gives us the same optimal hyperplane but the dual problem is often easier and it also allows us to exploit the kernel trick that we mentioned before.

The first task is to construct the Lagrangian. The optimization variables are b and w. There are M constraints, so we introduce M penalty terms and a Lagrange multiplier α_m for each of them. The Lagrangian is:

$$\begin{aligned} \mathcal{L}(b, w, \alpha) &= \frac{1}{2}w^2 + \sum_{m=1}^{M} \alpha_m(1 - y_m(w^T x_m + b)) \\ &= \frac{1}{2}w^2 - \sum_{m=1}^{M} \alpha_m y_m w^T x_m - b\sum_{m=1}^{M} \alpha_m y_m + \sum_{m=1}^{M} \alpha_m \end{aligned}$$

First, we have to minimize $\mathcal{L}$ with respect to b and w. We obtain its partial derivatives:

- The partial derivative with respect to b is the coefficient of b because it appears linearly in the Lagrangian.
$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_{m=1}^{M} \alpha_m y_m$$
- Using some linear algebra identities, we can obtain that the gradient with respect to w is
$$\frac{\partial \mathcal{L}}{\partial w} = w - \sum_{m=1}^{M} \alpha_m y_m x_m$$

We can derive the dual form by noting that the gradient of $\mathcal{L}$ with respect to w is zero if

$$w = \sum_{m=1}^{M} \alpha_m y_m x_m$$

On the other hand, the partial derivative with respect to b is zero if

$$\sum_{m=1}^{M} \alpha_m y_m = 0$$

Therefore,

$$\mathcal{L}(b, w, \alpha) = \frac{1}{2}w^2 - \sum_{m=1}^{M} \alpha_m y_m w^T x_m + \sum_{m=1}^{M} \alpha_m$$

We can substitute $w = \sum_{m=1}^{M} \alpha_m y_m x_m$ in the first two terms of the sum. After groping and operating we would obtain the final expression of the Lagrangian that only depends on α

$$\mathcal{L} = \sum_{m=1}^{M} \alpha_m - \frac{1}{2} \sum_{m,n=1}^{M} y_m y_n \alpha_m \alpha_n x_m^T x_n$$

This is know as the dual Lagrangian function. It describes a convex optimisation problem with M variables, where M is the number of training samples.

Once obtained this formulation, the optimisation problem is translated into an equivalent minimization task:

$$\begin{aligned} \text{Minimize:} \quad & \sum_{m=1}^{M} \alpha_m - \frac{1}{2} \sum_{m,n=1}^{M} y_m y_n \alpha_m \alpha_n x_m^T x_n \\ \text{subject to:} \quad & \sum_{m=1}^{M} \alpha_m y_m = 0 \\ & \alpha_m \geq 0 \quad m = 1, ..., M \end{aligned}$$

Once the optimal Lagrangian multipliers α^* are found, we are just missing to compute the optimal hyperplane. The optimal weights w_* are obtained from the condition

$$w_* = \sum_{m=1}^{M} \alpha_m^* y_m x_m$$

We can compute predictions with the linear model

$$f_{w_*}(x) = w_*^T x + b$$

or in the dual space

$$f(x) = \sum_{m=1}^{M} \alpha_m^* y_m x_m^T x + b$$

It is worth noting that the dual Lagrangian optimisation expression and the model in the dual space above rely only on the inner product $\langle x_i, x_j \rangle$ between data samples x_i, x_j. We can replace this inner product with an inner product in a feature space $\mathcal{F}$ and there is where kernels appear: $\kappa(x_i, x_j) = \langle \Phi(x_i), \Phi(x_j) \rangle_{\mathcal{F}}$

This way, the linear model is turned into a linear model in the feature space:

$$f_w(x) = \langle w, \Phi(x) \rangle_{\mathcal{F}} + b$$

with $w \in \mathcal{F}$.

The initial condition of linear separability of the data is translated to a feature space of higher dimension. Nonlinear data can be linearly separable in a feature space where the maximum-margin classifier becomes a kernel method that can candle non separable data. This way, SVM can find nonlinear decision boundaries in input space.

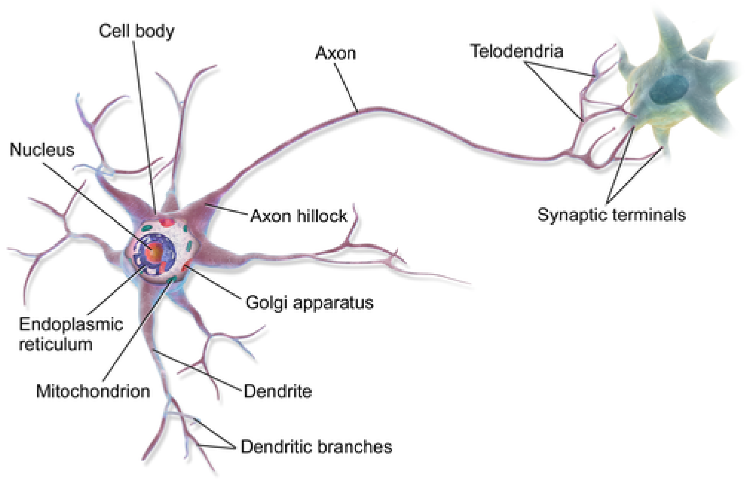


Figure 1.5.: Biological neuron. Image by BruceBlaus, CC BY 3.0, via Wikimedia Commons

1.4. Neural networks

Artificial neural networks are a fundamental component of artificial intelligence, that has revolutionized the field of machine learning and continue to drive innovation in various domains. These computational models were born taking the brain's architecture as inspiration. The human nervous system contains cells, which receive the name of neurons. They are composed of a cell body called "soma" containing the nucleus and most of the cell's complex components, and many branching extensions called "dendrites" plus one long extension called the "axon". At the tip of the ramifications of the axon there are tiny structures called "synaptic terminals", or simply "synapses", which are connected to the dendrites of other neurons. Figure 1.5 is an illustration of a neuron and its parts.

Biological neurons receive signals, short electrical impulses from other neurons through the synapses. A neuron fires its own signals once the incoming signal surpasses a certain threshold value. This is know as the **integrate and fire** principle, one of the simplest models of a neuron's electrical properties and probably the most commonly used in the field of neuroscience. The strengths of synaptic connections often change in response to external stimuli. At first glance, it looks like individual neurons behave in a rather simple way, yet they are organized in a vast network comprising billions of neurons, with each neuron usually connected to thousands of other neurons. The cortex of different species has been observed to display a laminar or layered structure. Each individual layer within the cortex showcases a unique distribution of different types of neurons and their respective connections. The structure of biological neural networks remains a topic of ongoing investigation and research.

This biological mechanism is simulated in artificial neural networks. The artificial neurons are the computational units that are connected with each other through weights, which model the strengths of synaptic connections present in the biological case. Every input received by a neuron is scaled by a weight factor, influencing the function computed by that specific unit. An artificial neural network processes the input values by propagating their computed values from the input neurons to the output neuron(s) while leveraging the weights as intermediary parameters, in order to learn. Similar to how biological organisms require external stimuli for learning, artificial neural networks rely on training data containing examples of input-output

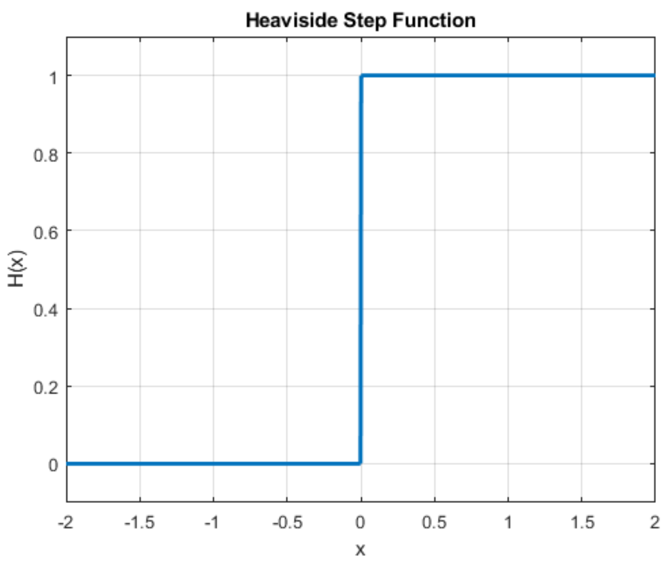


Figure 1.6.: Plot of the Heaviside step function

pairs, which serve as the external stimulus for the learning process.

Throughout this section, we will use the term neural networks to refer to artificial neural networks, leaving behind the biological ones.

1.4.1. Perceptron

The perceptron is the simplest neural network, proposed in 1957 by Frank Rosenblatt [Ros58]. It has a single input layer and an output node. It computes a weighted sum of its inputs $\{x_1, ..., x_n\}$: $z = w_1x_1 + ...w_nx_n = w^Tx$, where $x = (x_1, ..., x_n)$ is the input vector and $w = (w_1, ..., w_n)$ is the weight vector. Then, applies a non linear function φ called the activation function. Figure 1.7 displays the graphical representation of a perceptron.

In the original proposal of the perceptron, it was formulated with the Heaviside function, also known as step function, which is applied to that sum. Finally, it outputs the result. The model function of the perceptron is given by

$$f(x, w) = H(w^Tx)$$

where H is the Heaviside function (plotted in Figure 1.6)

$$H(x) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

The perceptron can be used for linear binary classification. It computes a linear combination of the inputs and if the result exceeds a threshold, it outputs the positive class, otherwise it outputs the negative class, just like it does a linear support vector machine.

At the time the perceptron was proposed by Rosenblatt, the parameter optimization was performed in a heuristic way with hardware circuits (*Mark I perceptron*) and it was not presented in terms of a formal notion of optimization in machine learning. The training was

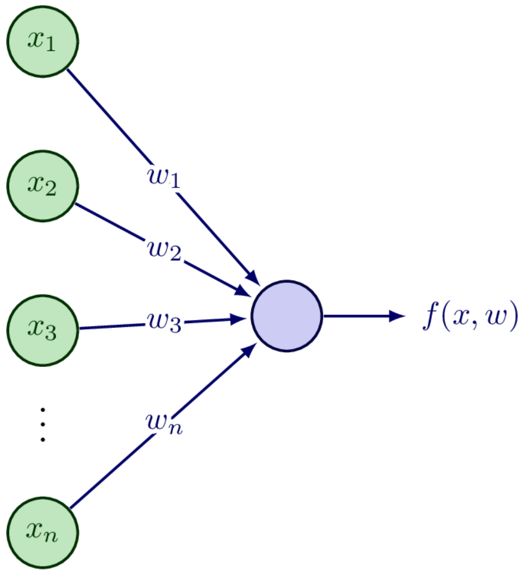


Figure 1.7.: Illustration of the perceptron model. The inputs are displayed as nodes of graph called *neurons*. Each input unit is linked to the output unit through a weighted connection.

inspired by *Hebb's rule*. Donald Hebb suggested in 1949 [Heb05] that when a biological neuron often triggers another neuron, the connection between them two grows stronger. The Hebbian learning is derived from this and states that the connection weight between two neurons is increased whenever they have the same output. Perceptrons are trained taking into account the error made by the network, reinforcing connections that help reduce the error. The training of the perceptron model is done by iterating through the training data and updating the weights according to the following rule

$$w_{i,j}^{\text{next step}} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

- $w_{i,j}$ is the connection weight between the i-th input neuron and the j-th output neuron.
- η is the learning rate.
- x_i is the i-th input value of the current training sample.
- y_j is the label of the j-th output neuron for the current training sample.
- $\hat{y}_j$ is the output of the j-th output neuron for the current training sample.

The perceptron model has several weak points. It is only valid for linearly separable datasets, which excludes learning a simple XOR function. Additionally, the Heaviside activation function is not differentiable.

1.4.2. Feed-forward neural networks

Neural networks are composed of one *input layer*, one or more layers of perceptrons called *hidden layers*, and one *output layer*. In other words, a neural network are stacked perceptrons, hence they are also referred to as multilayer perceptron. The most important neural network structure is the feed-forward neural network, which has an acyclic layering structure of perceptrons so that outputs of one perceptron are the input of another in the following layer. In the rest of the section we will simply say neural networks when referring to feed-forward neural networks.

The function that expresses neural networks as a deterministic model can be expressed as

$$f_{W_1,W_2,...,W_L}(x) = \varphi_L(w_L\varphi_{L-1}(W_{L-1}\cdots\varphi_2(W_2\varphi_1(W_1x))\cdots))$$

where $x \in \mathcal{X}$ is an input, W_i, $i = 1, ..., L$ is a real valued matrix of weights of the i-th layer and φ_i is the activation function in vectorised form, that is, the variable is a vector and the function acts element-wise returning a vector of the same size.

For many years, researchers encountered challenges to find a way to train neural networks, until the **backpropagation algorithm** was introduced. In essence, it is simply gradient descent using an efficient technique for computing the gradients automatically in two passes through the network (one forward and one backwards). The backpropagation algorithm is able to compute the gradient of the cost function (the model's generalisation error) with respect to every single model parameter. Once it has these gradients, it performs regular gradient descent. The whole process is repeated until convergence to the optimal solution is obtained. Later, we will come back to this idea in detail.

Previously, we presented the activation function of the perceptron: the Heaviside function. This introduces two problems when working with neural networks:

1. The Heaviside function is not differentiable at $x = 0$ and has 0 derivative elsewhere, so there is no gradient to work with since gradient descent cannot progress on a flat surface.

2. The goal of a neural network is to learn the values of the weights so that the output from the network correctly classifies the data, therefore, it is desirable to have stability, that is a small change in the weight to cause only a small corresponding change in the output of the network. This way, we can modify the weights little by little towards the behaviour that approximates best the data. The problem is that the output of the Heaviside function is 0 or 1 and thus, a small change in the weights of any single perceptron in the network can cause the output of that perceptron to completely flip from 0 to 1, or viceversa, affecting the behaviour of the rest of the neural network to completely change in some unexpected way.

There are other activation functions that allow us to properly work with neural networks and use the powerful tool of backpropagation and gradient descent. We will consider neural networks as stacked neurons, not necessarily perceptrons, thus, a concatenation of linear models and activation functions.

It is worth pointing out the sigmoid neuron. It is another type of artificial neuron, similar to perceptrons, but smooth and stable, we will see why next. First, it computes a weighted sum of its inputs $\{x_1, ..., x_n\}$: $z = w_1x_1 + ...w_nx_n = w^Tx$, where $x = (x_1, ..., x_n)$ is the input vector and $w = (w_1, ..., w_n)$ is the weight vector. Then, it applies the sigmoid function

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The sigmoid function is smooth, which makes that small changes in the weight cause only a

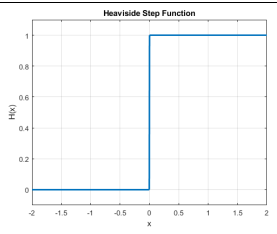
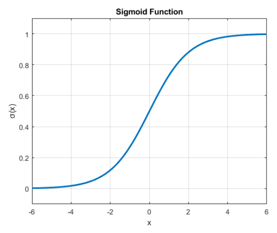
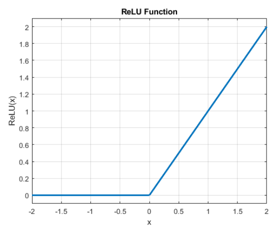
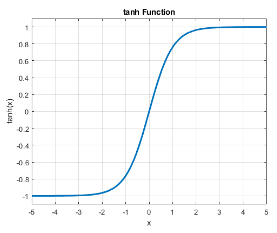
Function	Formula	Plot
Heaviside	$H(x) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$	
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	
ReLU	$\sigma(x) = \max\{0, x\}$	
Hyperbolic tangent	$\sigma(x) = \tanh(x)$	

Table 1.2.: Activation functions

small change in the output. It is also differentiable and has a well-defined nonzero derivative everywhere, allowing gradient descent to make progress at every step.

There are other options of activation functions as shown in Table 1.2.

The graphical representation of neural networks (such as Figure 1.8) really helps describing and visualizing its architecture because it intuitively presents information "flowing" from the inputs on the far left, along links and through hidden nodes, ultimately to output $f(x)$ on the far right. In order to avoid confusion with the notation, we will make an effort to detail it now that will reward us in the next sections.

Neural networks connect multiple neurons in layers, labeled by $l = 0, ..., L$ so that the units of each layer are connected to the units of the following layer. In the example of Figure 1.8, $L = 3$: the input layer $l = 0$, the output layer $l = L$ which determines the value of the function and the layers in between $1 \leq l < L$ are hidden layers. The first layer is formed by input units $x_1, ..., x_N$, the following $L - 1$ layers contain the hidden neurons $h_1^{(l)}, ..., h_{J_l}^{(l)}$ where $l = 1, ..., L$ and J_l is the number of neurons of the l-th hidden layer. The last layer consists on the output unit(s). It is common to add a bias term in the weighted sum of the inputs of a neuron, resulting in an extra unit in each layer with fixed value 1, so that the bias parameter

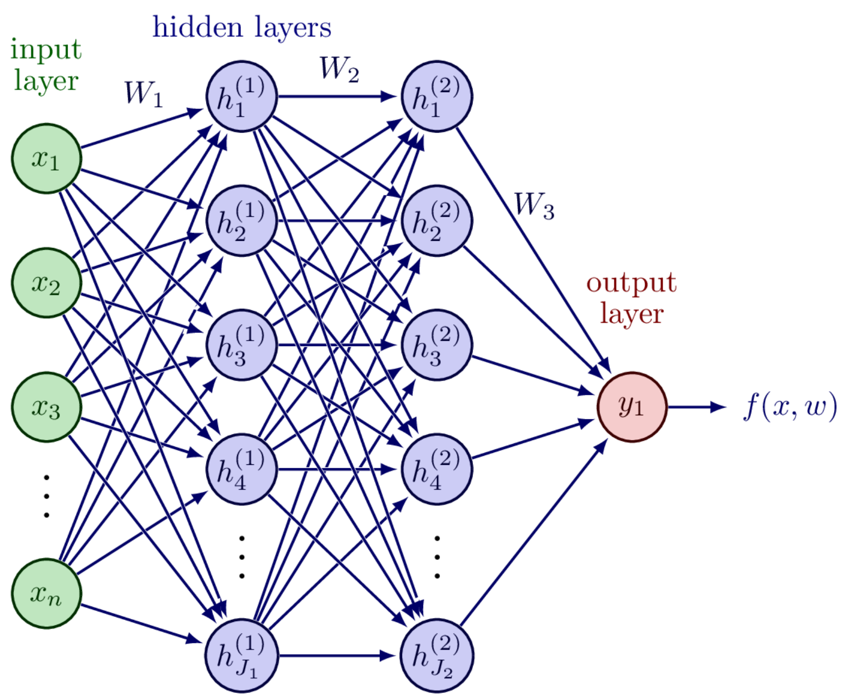


Figure 1.8.: Illustration of a neural network. Adapted from Izaak Neutelings, CC BY-SA 4.0

b is taken into account in the computations.

Every arrow represents a weight or connection strength from a neuron in a layer to a neuron in the next layer. The weights are represented by W_l, $l = 1, ..., L$, a real valued matrix of weights of the l-th layer. The weight into neuron j in layer l from node i in the previous layer $l-1$ is $w_l^{(i \to j)}$. Each neuron is updated by an activation function depending on all neurons feeding into it, and the update protocol prescribes that each layer is updated after the previous one, thus working layer by layer. When working this way, vector and matrix notation are useful: the output of neurons $1, ..., J_l$ in layer l is the vector $X_l \in \{1\} \times \mathbb{R}^{J_l}$ and the matrix of weights W_l of dimension $(J_{l-1}+1) \times J_l$ (taking into account the bias node).

The dimension of each layer $d = [J_1, J_2, ..., J_L]$ and the number of layers L determines the architecture of the neural network, which means the neural network model family whose parameters will be optimized to choose the specific configuration of weights that best generalises the data.

1.4.3. Forward propagation

The function that expresses the output of a neural network is

$$f_{W_1,W_2,...,W_L}(x) = \varphi_L(w_L \varphi_{L-1}(W_{L-1} \cdots \varphi_2(W_2 \varphi_1(W_1 x)) \cdots))$$

where $x \in \mathcal{X}$, W_i, $i = 1, ..., L$ is a real valued matrix of weights of the i-th layer and φ_i is the activation function in vectorised form. This is computed by the **forward propagation algorithm**. Taking into account the bias as node 0, we can observe that the inputs and outputs of a layer are related by

$$X_l = \begin{bmatrix} 1 \\ \sigma(W_l X_{l-1}) \end{bmatrix}$$

where $\sigma(W_l X_{l-1})$ is a vector whose components are the result of applying the vectorised form of the activation function to the weighted sum (with weights W_l) of the outputs from the previous layer (X_{l-1})

This computation starts by initializing the input layer $x_1 = x \in \mathcal{X}$.

After forward propagation, the output vector x_l at every layer $l = 1, ..., L$ has been computed.

1.4.4. Backpropagation Algorithm

Gradient descent is the algorithm used for finding the minimum of a differentiable function. It iteratively updates the parameters θ of a cost function $C(\theta)$ towards the direction of steepest descent in order to minimize a cost function. In the case of neural networks, the cost function and the model function of the neural network depend on $W = \{W_1, , ..., W_L\}$. Considering the squared error loss and the dataset $\mathcal{D} = \{(x_1, y_1), ..., (x_M, y_M)\}$, we will work with the following cost function:

$$C(W) = \frac{1}{M} \sum_{i=1}^{M} (f(x_i, W) - y_i)^2$$

The gradient descent algorithm is our preferred choice for performing the optimisation of the parameters since, when working with nonlinear activation functions, we are facing a difficult optimisation problem that generally is nonlinear and non-convex. The weights are updated in gradient descent at every step $t = 1, 2, ...$ in the following way

$$W(t+1) = W(t) + \eta \nabla C(W(t))$$

To perform gradient descent we need the gradient and to compute the gradient of $C(W)$, we need its derivative with respect to each weight matrix

$$\frac{\partial C}{\partial W_l} = \frac{1}{M} \sum_{i=1}^{M} \frac{\partial e_n}{\partial W_l}$$

where $e_n = (f(x_n, W) - y_n)^2$

We could think of obtaining $\frac{\partial e}{\partial W_l}$ using the finite differences numerical method. Taking into account we need to compute this method for every weight and for every data point, it becomes computationally prohibitive. The alternative that enables us to compute these gradients is a dynamic programming algorithm known as backpropagation. It is based on several applications of the chain rule. Next, we will deduce the algorithm step by step and then, we will be able to join all the pieces together to write its pseudo-code.

We want to obtain the partial derivative $\frac{\partial e}{\partial W_l}$

First, observe that variations on the weights W_l influence the output X_L of the neural network. We can view schematically this dependency

$$W_l X_{l-1} \xrightarrow{\sigma} X_l \longrightarrow W_{l+1} X_l \xrightarrow{\sigma} X_{l+1} \longrightarrow \cdots \longrightarrow X_L = f(x, W)$$

From now on, we will call $W_l X_{l-1} = S_l$, that is, the input of the l-th layer. This vector

$S_l = [s_{(l),j}]_{j=1}^{J_l}$ corresponds to the inputs of each j node of the l-th layer.

Since variations in W_l impact the output $f(x, W)$, in turn, it affects the loss error e. Let's view this effect of W_l element-wise. A change in $w_l^{i\to j}$ only affects $s_{(l),j}$ the input of the j-th node of the l-th layer because

$$s_{(l),j} = \sum_{i=0}^{J_{L-1}} w_l^{i\to j} x_{(l-1),i}$$

Thus, by the chain rule

$$\frac{\partial e}{\partial w_l^{i\to j}} = \frac{\partial s_{(l),j}}{\partial w_l^{i\to j}} \frac{\partial e}{\partial s_{(l),j}} = x_{(l-1),i}\delta_{(l),j}$$

where the last equality follows from the linear relation between $s_{(l),j}$ and $w_l^{i\to j}$. We have defined

$$\delta_{(l),j} = \frac{\partial e}{\partial s_{(l),j}}$$

The vector

$$\delta_l = [\delta_{(l),j)}]_{j=1}^{J_l} = \frac{\partial e}{\partial S_l}$$

denotes the sensitivity and it quantifies how the error e changes with S_l.

We can obtain X_l for every $l = 1, ..., L$ using forward propagation. So to get the partial derivatives, we need to obtain the sensitivity vectors δ_l for every layer. It turns out that the sensitivity vectors can be retrieved by "running backwards" our neural network. We can explicitly get the sensitivity of the output layer δ_L because $e = (X_L - y)^2 = (\sigma(S_L) - y)^2$. Therefore,

$$\begin{aligned}\delta_L &= \frac{\partial e}{\partial S_L} \\ &= \frac{\partial}{\partial S_L}(X_L - y)^2 \\ &= 2(X_L - y)\frac{\partial X_L}{\partial S_L} \\ &= 2(X_L - y)\sigma'(S_L)\end{aligned}$$

is the starting point of the backward recursion.

We will see how to obtain $\delta_{(l),j}$ element-wise. Since e depends on S_l only through X_l, by the chain rule

$$\begin{aligned}\delta_{(l),j} &= \frac{\partial e}{\partial s_{(l),j}} \\ &= \frac{\partial e}{\partial x_{(l),j}} \frac{\partial x_{(l),j}}{\partial s_{(l),j}} \\ &= \frac{\partial e}{\partial x_{(l),j}} \sigma'(s_{(l),j})\end{aligned}$$

Next, we will obtain the partial derivative $\frac{\partial e}{\partial X_l}$. A variation in X_l only affects S_{l+1} and thus, e. A particular component of X_l can affect every component of S_{l+1}, therefore we need to apply the chain rule taking into account all these dependencies:

$$\begin{aligned}\frac{\partial e}{\partial x_{(l),j}} &= \sum_{k=1}^{J_{l+1}} \frac{\partial s_{(l+1),k}}{\partial x_{(l),j}} \frac{\partial e}{\partial s_{(l+1),k}} \\ &= \sum_{k=1}^{J_{l+1}} w_{l+1}^{j\to k} \delta_{(l+1),k}\end{aligned}$$

because

- $s_{(l),j} = \sum_{i=0}^{J_{L-1}} w_l^{i\to j} x_{(l-1),i}$, then its derivative with respect to $x_{(l-1),j}$ is

$$\frac{\partial s_{(l),k}}{\partial x_{(l-1),j}} = w_l^{j\to k}$$

- and by definition we have that

$$\delta_{(l+1),k} = \frac{\partial e}{\partial s_{(l+1),k}}$$

Joining it all together, we obtain the expression for the components of the sensitivity vector

$$\delta_{(l),j} = \sigma'(s_{(l),j}) \sum_{k=1}^{J_{l+1}} w_{l+1}^{j\to k} \delta_{(l+1),k}$$

This resulting equation to compute the sensitivity at the l-th layer is expressed in terms of the sensitivity in the next layer $l+1$, producing a backward recursion. Therefore, if we know the sensitivities at layer $l+1$, we can get δ_l.

The consequent vector expression of the sensitivity is

$$\delta_l = \sigma'(S_l) \odot [(W_{l+1})^T \delta_{l+1}]_1^{J_l}$$

where $\odot$ denotes the Hadamard product, the elementwise product of the two vectors: $u, v \in \mathbb{R}^n$, then $u \odot v$ is a vector such that $(u \odot v)_i = u_i v_i$, $i = 1, ..., n$.

In the vector equation of the sensitivity δ_l in terms of the sensitivity in the next layer δ_{l+1}, W_{l+1} is the transpose of the weight matrix of the $l+1$-th layer, W_{l+1}. When we apply this transpose weight matrix we can think intuitively as moving the sensitivity error backward through the network. Then, we can interpretate the Hadamard product of $\sigma'(S_l)$ as moving the sensitivity error backwards through the activation function in layer l.

```
Backpropagation algorithm

Input: a data point $(X, y)$

1. Run forward propagation on $X$ to compute and save $X_l$ and $S_l$ for $l = 1, ..., L$
2. Initialization
   $\delta_L \leftarrow 2(X_L - y)\sigma'(S_L)$
3. Backpropagation recursion
   for $l = L - 1$ to 1 do:
       $\delta_l \leftarrow \sigma'(S_l) \odot [(W_{l+1})^T \delta_{l+1}]_1^{J_l}$
```

Finally, as a recapitulation, we will remember our goal was to perform gradient descent to optimise the weights of the neural network, for which we needed the gradient of the cost function with respect to every weight parameter. Using the chain rule, the expression for the required partial derivatives is

$$\frac{\partial e}{\partial w_l^{i \to j}} = x_{(l-1),i}\delta_{(l),j}$$

We can compute it because we can obtain the term X_l, $l = 1, ..., L$ with the forward propagation recursion

$$X_l = \begin{bmatrix} 1 \\ \sigma(W_l X_{l-1}) \end{bmatrix}$$

with initial condition $X_0 = X \in \mathcal{X}$, the input vector.

On the other hand, we can calculate the sensitivities δ_l, $l = 1, ..., L$ with backpropagation using the recursion

$$\delta_l = \sigma'(S_l) \odot [(W_{l+1})^T \delta_{l+1}]_1^{J_l}$$

with initial condition $\delta_L = 2(X_L - y)\sigma'(S_L)$.

2. Introduction to Quantum Computing

In 1981 at MIT, the first Physics of Computation Conference was held, organized by MIT and IBM, which brought together 50 scientists for three days. It was the founding forum to discuss the possibility of taking signals from the natural world to address the problems of designing increasingly powerful and efficient forms of computation. Attendees debated the limitations and possibilities brought about by the shrinking of computer components (Moore's Law). Ultimately, it was the time and place to discuss the intersection of the physical sciences and computer sciences, which helped to connect the paths of both disciplines. Today, this conference holds more weight than they could have imagined at the time. Perhaps due to the significance of its sponsoring institutions and the fame of some attendees, such as Freeman Dyson, John Wheeler, and Richard Feynman, we can say that it was the birthplace of physics of computation and especially the now flourishing field of quantum computing.

The opening speech of the Physics of Computation Conference was delivered by Richard Feynman and was titled "Simulating physics with computers". As the title suggests, it explores the idea of how physics can be simulated using computers, which is related to the possibilities of computers and also the possibilities of physics. Throughout the speech, he poses a series of questions.

First, he addresses the question: What type of physics will be simulated with computers? Classical physics is often described using differential equations, but the physical world is quantum, so the problem he raises is simulating quantum physics.

> "And I'm not happy with all the analyses that go with just the classical theory, because nature isn't classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy."

The idea that Richard Feynman explores is whether there is an exact simulation in which the computer will do exactly the same as nature.

So, how can quantum mechanics be simulated? To answer this question, Richard Feynman concludes that the only possibilities are: to use a probabilistic computer or to do it with a new type of computer, a quantum computer built with elements of quantum mechanics that obey the laws of quantum mechanics. He asserts that a quantum system can be simulated with quantum computational elements. He is not referring to a Turing machine but to a machine of a different type. Feynman believes that it is true that with an appropriate class of quantum machines, one could simulate any quantum system, including the physical world.

In his speech, Feynman only outlines ideas about what this quantum computer would be like. He also raises the question of which types of quantum systems are equivalent and the possibility of finding a class that could simulate all quantum systems. This leads to the question of whether a universal quantum computer could be found. He suggests using linear operators in a two-state quantum system as the basis for simulating any quantum system.

In the final part of the conference, he argues whether a quantum system can be probabilistically simulated with a classical universal probabilistic computer, meaning a computer that would obtain the same probabilities as the quantum system. He concludes that this is definitely not the case and that it is impossible to represent the results of quantum mechanics with a classical universal device.

Feynman proposes this union between computing and physics because he claims that the discovery of computing has proven very useful in many branches of human reasoning. For example, attempting to create a computer that understands our language made us realize how poor our understanding of language and grammar theory was. Similarly, we have gained some insights about psychology and philosophical issues by approaching them from a computational perspective. Feynman's intention was that introducing computational approach would provide us with new ideas if they were necessary.

2.1. Review on Hilbert spaces

Quantum theory can me understood as an extension of classical probability theory to make predictions about the stochastic outcomes of measurements on microscopic objects (such as electrons, protons, atoms, etc.) that form quantum systems.

The rigorous formulation of this requires some mathematical tools and notions that we will review next.

Definition 2.1 (Inner product)**.** Let X be a vector space, that is,

$$\psi, \phi \in X \text{ and } a, b \in \mathbb{C} \text{ verify } a\psi + b\phi \in X$$

The scalar product or inner product in X is an application

$$\begin{aligned} \langle \cdot, \cdot \rangle : X \times X &\longrightarrow \mathbb{C} \\ (\psi, \phi) &\longmapsto \langle \psi, \phi \rangle \end{aligned}$$

That verifies for all $\phi, \psi, \phi_1, \phi_2 \in$ X y $a, b \in \mathbb{C}$

1. It is hermitian: $\langle \psi, \phi \rangle = \overline{\langle \psi, \phi \rangle}$
2. It is positive definite: $\langle \psi, \psi \rangle \geq 0$ y $\langle \psi, \psi \rangle = 0 \Leftrightarrow \psi = 0$
3. It is sesquilinear, that is
   - $\langle \psi, a\phi_1 + b\phi_2 \rangle = a \langle \psi, \phi_1 \rangle + b \langle \psi, \phi_2 \rangle$ (Linearity of the second entry)
   - $\langle a\psi_1 + b\psi_2, \phi \rangle = \overline{a} \langle \psi_1, \phi \rangle + \overline{b} \langle \psi_2, \phi \rangle$ (Antilinearity of the first entry)

Definition 2.2 (Pre-Hilbert space)**.** If $\langle \cdot, \cdot \rangle$ is a inner product in a vector space $\mathbb{H}$, we will say that $\mathbb{H}$ is a pre-Hilbert space.

This inner product induces a norm in $\mathbb{H}$,

$$\begin{aligned} || \cdot || : \mathbb{H} &\longrightarrow \mathbb{R} \\ \psi &\longmapsto \sqrt{\langle \psi, \psi \rangle} \end{aligned}$$

Definition 2.3. A normed space M is complete if every Cauchy sequence in M is convergent. That is, if $\{x_n\}_{n \in \mathbb{N}}$ is a sequence in M such $\forall \epsilon > 0 \ \exists n_0 \in \mathbb{N}$ that $n, m \geq n_0$, $||x_n - x_m|| < \epsilon$, then $\{x_n\}$ converges in M. This spaces receive the name of **Banach spaces**.

Definition 2.4 (Hilbert space)**.** A Hilbert space $\mathbb{H}$ is a pre-Hilbert space that is complete respect to the norm induced by the inner product.

Equivalently, a Hilbert space is a Banach space with the norm induced by the inner product.

Definition 2.5 (Separable space)**.** A metric space M is separable if there exists a subset $N \subset M$ that is countable and dense.

Now we will quickly revise the basic concepts of linearly independence, span and basis.

Definition 2.6. Let X be a pre-Hilbert space and J an index set.

A set of vectors $\{v_i \,|\, i \in J\} \subset X$ is **linearly independent** if the coefficients $\alpha_i \in \mathbb{C}$, $i \in J$ such that

$$\sum_{i \in J} \alpha_i v_i = 0$$

only holds if $a_i = 0$ for all $i \in J$.

A set of vectors $\{v_i \,|\, i \in J\} \subset X$ **spans** X if for every $\omega \in X$, there are coefficients $\alpha_i \in \mathbb{C}$, $i \in J$ such that

$$\omega = \sum_{I \in J} \alpha_i v_i$$

Therefore, we denote it as

$$X = \text{span}\,\{v_i \,|\, i \in J\}$$

A linearly independent set $\{v_i \,|\, i \in J\}$ that spans X is called a **basis** of X.

An **orthonormal basis** $\{e_i \,|\, i \in J\}$ is a basis whose vectors are unitary ($||e_i|| = 1 \,\forall i \in J$) and verify

$$\langle e_i | e_j \rangle = \delta_{i,j} = \begin{cases} 0 & \text{if} \quad i \neq j \\ 1 & \text{if} \quad i = j \end{cases}$$

Theorem 2.1. *Every separable Hilbert space has an orthonormal basis.*

Proof. Let $\mathbb{H}$ be a Hilbert space and $\{u_n\}$ be a countable dense subset of $\mathbb{H}$. We consider the nondecreasing sequence of linear spaces $\{E_i\} = \textbf{span}\,\{u_1, u_2, ..., u_i\}$ such that $\bigcup_{i=1}^{\infty} E_i$ is dense in $\mathbb{H}$.

Let's pick any unit vector u_1 in E_1. If $E_2 \neq E_1$, there is some vector $u_2 \in E_2$ such that $\{u_1, u_2\}$ is an orthonormal basis of E_2. By repeating this iterative construction, we obtain an orthonormal basis of $\mathbb{H}$. [Bre11] □

Now we will get introduced to the notion of linear functional.

Definition 2.7 (Linear operator)**.** Let X and Y be to vector spaces. A linear operator from X to Y is an application $T : X \longrightarrow Y$ that verifies

$$T(\lambda_1 x_1 + \lambda_2 x_2) = \lambda_1 T(x_1) + \lambda_2 T(x_2) \quad \forall \lambda_1, \lambda_2 \in \mathbb{K},\, x_1, x_2 \in X$$

If $Y = \mathbb{K}$ the linear operator T receives the name of **linear functional**.

Proposition 2.1 (Continuous operator)**.** *Let X and Y be normed vector spaces and $T : X \longrightarrow Y$ a linear operator. Then T is continuous if and only if $\exists M \in \mathbb{R}_0^+$ such that*

$$||T(x)|| \leq M||x|| \quad \forall x \in X$$

Definition 2.8 (Operator space)**.** Given X and Y normed vector spaces over $\mathbb{K}$, we denote $\mathcal{L}(X,Y)$ the set of all linear operators from X to Y. Provided with the operations:

- $\forall T, S \in \mathcal{L}(X,Y),\ (T+S)(x) = T(x) + S(x) \quad \forall x \in X$
- $\forall \lambda \in \mathbb{K}, \forall T \in \mathcal{L}(X,Y),\ (\lambda T)(x) = \lambda T(x) \quad \forall x \in X$

The set $\mathcal{L}(X,Y)$ has structure of vector space.

Definition 2.9 (Dual space)**.** Let X be a normed vector space over $\mathbb{K}$. The operator space from X to $\mathbb{K}$, $\mathcal{L}(X,\mathbb{K})$ of all linear and continuous functionals in X is called dual space and it is denoted as X^*. It is provided with the norm

$$||f||_{X^*} = \min\left\{M \in \mathbb{R}_0^+ : |f(x)| \leq M||x||_X \quad \forall x \in X\right\}$$

In a Hilbert space, we can define linear functionals T_ψ for every $\psi \in \mathbb{H}$ with the help of the scalar product:

$$\begin{aligned} T_\psi : & \mathbb{H} \longrightarrow \mathbb{C} \\ & \varphi \longmapsto \langle \psi, \varphi \rangle \end{aligned}$$

By virtue of the Theorem of Riesz, Theorem A.3, for each functional $T \in \mathbb{H}^*$, there exists a unique $\psi \in X$ such that $T(x) = \langle x, \psi \rangle \quad \forall x \in X$.

This means that there is a bijection between $\mathbb{H}$ and its dual space $\mathbb{H}^*$: every linear and continuous functional from $\mathbb{H}$ to $\mathbb{C}$ is uniquely represented as a scalar product with a suitable vector in $\mathbb{H}$.

Therefore, $\mathbb{H}$ and $\mathbb{H}^*$ are isomorphic, in particular, if $\mathbb{H}$ is a separable Hilbert space, the dual space $\mathbb{H}^*$ is a vector space with the same dimension as $\mathbb{H}$.

This fact suggests the **Bra-ket notation** introduced by Paul Dirac, extensively used in Quantum Mechanics.

- Ket-vectors are elements of $\mathbb{H}$. A vector $\varphi \in \mathbb{H}$ in the Hilbert space is denoted in Dirac's notation as the ket $|\varphi\rangle \in \mathbb{H}$.
- Bra-vectors are elements of the dual space $\mathbb{H}^*$. A functional $T_\phi \in \mathbb{H}^*$ is denoted in Dirac's notation as the bra $\langle\phi| \in \mathbb{H}^*$.
- The action of a functional $\langle\phi| \in \mathbb{H}^*$ on a vector $|\varphi\rangle \in \mathbb{H}$ is denoted as the braket $\langle\phi|\varphi\rangle = \langle\phi, \varphi\rangle \in \mathbb{C}$. Note the scalar product.

In other words, every ket $|\varphi\rangle$ has a corresponding $\langle\varphi|$ that is unique. The scalar product of two vectors (kets) $|\phi\rangle$ and $|\varphi\rangle$ is given by the braket $\langle\phi|\varphi\rangle = \langle\phi, \varphi\rangle^*$

A basis of the Hilbert space $\mathbb{H}$ can be denoted according to Dirac's notation as $\{|\phi_i\rangle \mid i \in J\}$ and we can express any ket $|\varphi\rangle \in \mathbb{H}$ as

$$|\varphi\rangle = \sum_{i \in J} |\phi_i\rangle\, \alpha_i$$

where α_i are the components of $|\varphi\rangle$ in the basis $\{|\phi_i\rangle \mid i \in J\}$.

Now, let $\{|e_i\rangle \mid i \in J\}$ be an orthonormal basis of $\mathbb{H}$. The coefficients of any vector can be obtained from the inner product, since $|\varphi\rangle = \sum_{i\in J} |e_i\rangle\, \alpha_i$, then

$$\langle e_k|\varphi\rangle = \sum_{i \in J} \langle e_k|e_i\rangle\, \alpha_i = \sum_{i \in J} \delta_{ki}\alpha_i = \alpha_k$$

As a result,

$$|\varphi\rangle = \sum_{i\in J} |e_i\rangle \langle e_i|\varphi\rangle$$

Regarding the inner product of two vectors, we can use its representation in an orthonormal basis as

$$|\varphi\rangle = \sum_{i\in J} |e_i\rangle \varphi_i = \sum_{i\in J} |e_i\rangle \langle e_i|\varphi\rangle$$
$$|\phi\rangle = \sum_{i\in J} |e_i\rangle \phi_i = \sum_{i\in J} |e_i\rangle \langle e_i|\phi\rangle$$

$$\langle\phi|\varphi\rangle = \sum_{i\in J} \langle\varphi|e_i\rangle \langle e_i|\phi\rangle = \sum_{i\in J} \langle e_i|\varphi\rangle^* \langle e_i|\phi\rangle = \sum_{i\in J} \varphi_i^*\phi_i = \langle\varphi,\phi\rangle^*$$

Indeed, the isomorphism between $\mathbb{H}$ and $\mathbb{H}^*$ is given by the adjoint relation:

$$\begin{aligned} \mathbb{H} &\longrightarrow \mathbb{H}^* \\ |\varphi\rangle &\longmapsto \langle\varphi| = |\varphi\rangle^\dagger = \sum_{i\in J} \varphi_i^* \langle e_i| \end{aligned}$$

Thus, there is an adjoint basis in $\mathbb{H}^*$

$$\{|e_i\rangle\} \longmapsto \{\langle e_i|\}$$

Definition 2.10 (Adjoint operator)**.** Let X and Y be normed spaces. Consider the linear operator T from X to Y. The adjoint operator of T, noted as $T^\dagger$ is the operator $T^\dagger : Y^* \longrightarrow X^*$ defined as $\forall g \in Y^*$, $T^\dagger(g)$ is the functional given by $(T^\dagger g)(x) = g(T(x)) \quad \forall x \in X$. When X and Y are finite spaces of dimension n, then $T \in \mathcal{M}_{n\times n}$. Thus, its adjoint $T^\dagger$ is given by $(T^\dagger)_{ij} = T_{ji}^*$.

When T is a linear operator in a Hilbert space $\mathbb{H}$, we can apply the Riesz theorem, Theorem A.3, on both sides of the definition $(T^\dagger g)(x) = g(T(x))$ to obtain that the adjoint operator realices the following equivalent identity: $\langle T^\dagger(u), v\rangle = \langle u, T(v)\rangle$, $\forall v \in \mathbb{H}$.

Definition 2.11 (Self-adjoint operator)**.** A linear operator T in a Hilbert space $\mathbb{H}$ is a self adjoint operator, also known as Hermitian operator, if its adjoint is itself: $T^\dagger = T$. This is equivalent to say $\langle T(x), y\rangle = \langle x, T(y)\rangle \quad \forall x, y \in \mathbb{H}$.

Definition 2.12 (Unitary operator)**.** An operator T is unitary if $T^\dagger T = TT^\dagger = \mathbb{1}$

Definition 2.13 (Outer product)**.** Let X and Y be vector spaces. The outer product of $v \in X$ and $w \in Y$, using Dirac's notation is

$$|w\rangle \langle v|$$

is the only linear operator such that

$$(|w\rangle \langle v|) |x\rangle = \langle v, x\rangle |w\rangle \quad \forall |x\rangle \in X$$

.

Lastly, we will remember the concepts of eigenvalues and eigenvectors.

Definition 2.14 (Eigenvalue and eigenvector)**.** Given X a vector space over $\mathbb{K}$, $T \in \mathcal{L}(X)$ and $v \in X$. If there exist a $\lambda \in \mathbb{K}$ such that

$$Tv = \lambda v$$

Then, λ is an eigenvalue of T and v is a eigenvector of T.

Next, we will present two definitions that allow us to quantify the characteristics of a system defined within a Hilbert space.

Definition 2.15 (Observable)**.** In physics, an observable, such as position or momentum, are physical quantities governed by a real-valued function in the case of classical mechanics. In the case of quantum mechanics, an observable is given by an operator that can be measured.

Definition 2.16 (Expected value)**.** Given an operator T, the expected value $\langle T \rangle$ is defined as the probabilistic value expected to be obtained when a measurement of the observable associated with T is performed. If the system is in a state ψ, the expected value of T is given by $\langle T \rangle_\psi = \langle \psi | T | \psi \rangle$.

Finally, we will point out a particular type of observable:

Definition 2.17 (Hamiltonian)**.** In quantum mechanics, the operator corresponding to the total energy (kinetic and potential) of the system is called the Hamiltonian.

2.2. Postulates of Quantum Mechanics

Now, we will formulate the framework of quantum mechanics for finite-dimensional systems with the guidance of a number of postulates which specify the mathematical objects used to describe certain physical entities. We will follow [NC10].

2.2.1. State space

Postulate 1

Associated to any isolated physical system is a complex vector space with inner product (that is, a Hilbert space) known as the **state space** of the system. The system is completely described by its **state vector**, which is a unit vector in the system's state space.

A quantum system with a state vector $|\psi\rangle$ is called a pure state. However, usually systems cannot be described by an unique pure state, instead they are in a statistical ensemble of different pure states, each of them with its own probability. In this case, the state of a system is given by a operator called **density operator** or **density matrix**, defined by

$$\rho = \sum_{i=1}^{n} p_i \, |\psi_i\rangle \, \langle \psi_i|$$

where all $p_i \geq 0$ and $\sum_i p_i = 1$.

Additionally, a density matrix has the following properties:

1. ρ is self-adjoint
$$\rho^\dagger = \rho$$
2. ρ has trace 1
$$\mathrm{Tr}(\rho) = 1$$
3. ρ is positive
$$\rho \geq 0$$

This postulate does not provide the state space for a given system. Although, for the purpose of computation, the state spaces to consider should be discrete and finite-dimensional Hilbert spaces. These are isomorphic to the $\mathbb{C}^K$, and a quantum state therefore has a representation as a complex-valued vector.

The simplest quantum mechanical system is the one qubit system.

Definition 2.18 (Qubit)**.** A state vector of the state space $\mathbb{C}^2$ is called a qubit.

Now we pick up again Dirac's notation and we assume $|0\rangle$ and $|1\rangle$ form an orthonormal basis in a 2-dimensional Hilbert space. Thus, any state vector in the state space can be written as a linear combination of the basis vectors:

$$|\varphi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

where $\alpha, \beta \in \mathbb{C}$ are called amplitudes. A state vector is by definition unitary. This condition translates into $\langle\varphi|\varphi\rangle = 1$, which is equivalent to $|a|^2 + |b|^2 = 1$. This is known as normalization condition.

A qubit is the minimal information unit in quantum computing. It is the analogous to the classical bit in classical computation. Intuitively, the states $|0\rangle$ and $|1\rangle$ are equivalent to the values 0 and 1 which a bit may take. The difference relies on the fact that $|0\rangle$ and $|1\rangle$ are not the only states of a qubit, one can prepare a qubit in any arbitrary superposition of these two states: $\alpha\,|0\rangle + \beta\,|1\rangle$.

As we know, classical computation is built based on bits. Similarly, quantum computing is constructed upon the qubit.

We conclude with some usual terminology. The orthonormal basis of the state space receives the name of computational basis. In the case of the state space of a 1-qubit system, the constituents of the computational basis are the column vectors

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \qquad |1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$$

and hence,

$$|\varphi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \alpha\begin{bmatrix}1\\0\end{bmatrix} + \beta\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}\alpha\\\beta\end{bmatrix}$$

Additionally, taking into account a general orthonormal basis $|e_i\rangle$ of $\mathbb{H}$, with $\langle e_i|e_j\rangle = \delta_{ij}$, we realise that any $|\varphi\rangle \in \mathbb{H}$

$$|\varphi\rangle = \sum_i \langle e_i|\varphi\rangle\,|e_i\rangle$$

This previous equation also implies the **completeness relation**:

$$\sum_i |e_i\rangle \langle e_i| = \mathbb{1}$$

which holds for any orthonormal basis $|e_i\rangle$ of a finite $\mathbb{H}$.

We can generalise this to the case of n-qubits, thus we say that any linear combination $\sum_i \alpha_i |\phi_i\rangle$ is a superposition of the states $|\phi_i\rangle$ with amplitude α_i for the state $|\phi_i\rangle$.

In particular, we will see that the state space will be $\mathbb{C}^{2^n}$ for an n-qubit quantum computation system.

In conclusion, this first postulate sets the framework in which quantum mechanics takes place: Hilbert spaces. In particular, for quantum computation where we work with a finite number of qubits, we will consider finite Hilbert spaces. Linear algebra and functional analysis provide us the tools we need to operate in them. It is worth pointing out that the definition of a qubit is derived from quantum physics and mathematics, describing the qubit as a mathematical object independent of its physical implementation. This enables the construction of quantum computing regardless of its physical implementation.

2.2.2. Observables

Postulate 2

An observable, that is, a physically measurable quantity of a quantum system is represented by a Hermitian or self-adjoint operator on a Hilbert space $\mathbb{H}$.

Let $\mathcal{M}$ be a self-adjoint operator in $\mathbb{H}$ that acts on ket $|\varphi\rangle$ characterising the system. By virtue of the spectral theorem, Theorem A.1, such self-adjoint operators have a diagonal matrix representation in terms of the orthonormal basis of eigenvectors $\{|\mu_j\rangle\}$, with the eigenvalues in the diagonal. We can formulate this like

$$\mathcal{M} = \sum_k \lambda_k |\lambda_k\rangle \langle \lambda_k|$$

where λ_k are the eigenvalues and $|\lambda_k\rangle$ their corresponding eigenvectors.

The only possible values that the observable can take are its eigenvalues, which are real because the observable is a self-adjoint operator. Therefore, the values of physical observables are always real numbers.

2.2.3. Time Evolution

Postulate 3

The evolution of a closed quantum system is described by a unitary transformation. That is, the state $|\varphi\rangle$ of the system at time t_1 is related to the state $|\varphi'\rangle$ of the system at time t_2 by a unitary operator U which depends only on the times t_1 and t_2.

$$|\varphi'\rangle = U(t_1, t_2) |\varphi\rangle$$

This postulate encloses how a quantum state $|\varphi\rangle$ evolves through time but it does not tell you which unitary operators describe this evolution. In the context of quantum computing, any unitary operator can be realized in realistic systems and we will define them. While we may not know the specific unitary transformation for an arbitrary system, we are able to create systems that follow specific desired transformations. These principles form the foundation of quantum gates and quantum circuits

Now we will have a look at an example of an unitary operator on a single qubit which is important in quantum computation. We will detail at the end of this chapter a collection with the most important quantum gates.

Example 2.1 (NOT gate)**.** Let's consider the operator given by the matrix

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

known as the Pauli X matrix σ_x or NOT gate. It is evidently a unitary operator, since $X^\dagger X = \mathbb{1}$.

We can observe that

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

which is precisely what Postulate 3 states, the evolution of our state vector following the unitary operator X: $X|0\rangle = |1\rangle$.

It is worth pointing out two aspects of this postulate.

- It requires that the physical system system is closed, that is, it is not interacting in any way with the exterior. This is quite a strict assumption, since in reality, all systems interact at least somewhat with others and the only real closed system is the universe as a whole. However, some systems can be described to a good approximation as being closed.
- It refers to the description of evolution of a closed quantum system at two different times t_1 and t_2. It is natural to consider the evolution of a quantum system in continuous time. For this, we will consider the Schrödinger equation, which provides a continuous time description of a closed system. We can redefine the Postulate 3 in these terms.

Postulate 3′

The time evolution of the state of a closed quantum system is described by the Schrödinger equation,

$$i\hbar \frac{d|\phi\rangle}{dt} = H|\phi\rangle$$

where $\hbar$ is a physical constant known as Plank's constant and H is a fixed Hermitian operator known as the Hamiltonian of the closed system.

The Hamiltonian of the system is fixed and it represents the energy of the system. The value $\hbar$ is physical constant that we can absorb into the Hamiltonian to simplify the equation. At the end of the day, a Hamiltonian is an observable and we can apply the results from Postulate 2. As we will see later, circuits in quantum computers are built up from elementary gates that act as unitary operators on the states. In order to implement such gates, one then tries to create Hamilton operators that generate a time evolution implementing the desired gate.

Since the Hamiltonian is a self-adjoint operator, by the Spectral decomposition theorem, Theorem A.1

$$H = \sum_j \lambda_j \left|\lambda_j\right\rangle \left\langle\lambda_j\right|$$

with eigenvalues λ_j and corresponding eigenvectors $|\lambda_j\rangle$. Conventionally, $|\lambda_j\rangle$ are referred to as *energy eigenstates* and the eigenvalues λ_j as the energy of the state $|\lambda_j\rangle$.

Now we will consider the example for a single qubit that has Hamiltonian

$$H = \hbar\omega X, \quad \omega > 0,\, X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

to check that there is a one to one correspondence between the continuous time-varying postulate $3'$ using the Hamiltonian and the more stationary discrete-time version using the unitary operator (Postulate 3).

The energy eigenstates of this Hamiltonian are the ones of X:
$(|0\rangle + |1\rangle)/\sqrt{2}$ and $(|0\rangle - |1\rangle)/\sqrt{2}$ with corresponding energies (eigenvalues): $\hbar\omega$ and $-\hbar\omega$.

The solution to Schrödinger's equation is

$$|\phi(t_2)\rangle = \exp\left[\frac{-iH(t_2 - t_1)}{\hbar}\right] |\phi(t_1)\rangle$$

Now, we define the operator

$$U(t_1, t_1) = \exp\left[\frac{-iH(t_2 - t_1)}{\hbar}\right] |\phi(t_1)\rangle$$

which is an unitary operator. In fact, any unitary operator U can be expressed as $U = \exp(iK)$ for some Hermitian operator K.

Therefore, we obtain that the state $|\phi(t_2)\rangle$ is related with the state $|\phi(t_1)\rangle$ through the unitary operator $U(t_1, t_2)$:

$$|\phi(t_2)\rangle = U(t_1, t_2) \, |\phi(t_1)\rangle$$

Like this, we have obtained the equivalence of postulates 3 and $3'$.

In quantum computing, the circuit model usually consists of "applying" a unitary operator to a quantum system. A priori we may think that this act contradicts what we have just stated about the evolution of a closed quantum system because we are interacting with it and thus, the system is not closed. It turns out to be possible to write down a time-varying Hamiltonian for a quantum system that is not closed but overall the system evolves according to Schrödinger's equation with a time-varying Hamiltonian to some good approximation. The main exception to this are measurements, which we will dive into next.

2.2.4. Measurements

A closed quantum system evolves according to unitary evolutions by virtue of Postulate 3. But the moment there is an observation of the system, this interaction makes the system no longer closed and thus, not necessarily subject to unitary evolution. To explain what happens in this context we will introduce the following postulate, which describes the effects of measurement operations on quantum systems.

Postulate 4

Quantum measurements are described by a collection $\{M_m\}$ of *measurement operators*. These are operators acting on the state space of the system being measured. The index m refers to the measurement outcomes that may occur in the experiment. If the state of the quantum system is $|\phi\rangle$ immediately before the measurement then the probability that result m occurs is given by

$$p(m) = \langle\phi|M_m^\dagger M_m|\phi\rangle$$

and the state of the system after the measurement is

$$\frac{M_m|\phi\rangle}{\sqrt{\langle\phi|M_m^\dagger M_m|\phi\rangle}}$$

The measurement operators satisfy the **completeness equation**:

$$\sum_m M_m^\dagger M_m = \mathbb{1}$$

The completeness equation expresses the fact that probabilities of the possible different outcomes sum one.

$$\sum_m p(m) = \sum_m \langle\phi|M_m^\dagger M_m|\phi\rangle = \langle\phi|\sum_m M_m^\dagger M_m|\phi\rangle = \langle\phi|\phi\rangle = 1$$

Reciprocally, the fact of every state verifying this equation implies the completeness equation.

In the context of quantum computing, an important measurement is the measurement in the computational basis. Now we will review it for the case of a single qubit.

When measuring a qubit there are two possible outcomes: $|0\rangle$ and $|1\rangle$ defined by two measurement operators $M_0 = |0\rangle\langle 0|$ and $M_1 = |1\rangle\langle 1|$ respectively. We can observe that:

- These measurement operators are Hermitian:

$$M_i^\dagger = (|i\rangle\langle i|)^\dagger = (\langle i|)^\dagger(|i\rangle)^\dagger = |i\rangle\langle i| = M_i$$

- They verify $M_i^2 = |i\rangle\langle i|i\rangle\langle i| = |i\rangle\langle i| = M_i$
- The computational basis is orthonormal and thus, fulfills the completeness relation:

$$\sum_i |i\rangle\langle i| = 1$$

  Then, the completeness equation also holds:

$$\sum_i M_i^\dagger M_i = \sum_i M_i^2 = \sum_i M_i = \sum_i |i\rangle\langle i| = 1$$

Now, let's measure. Suppose we have the state $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$. The probability of obtaining measurement outcome 0 is

$$p(0) = \langle\phi|M_0^\dagger M_0|\phi\rangle = \langle\phi|M_0|\phi\rangle = |\alpha|^2$$

Analogously, the probability of obtaining the measurement outcome 1 is $p(1) = |\beta|^2$.

The completeness equation $|\alpha|^2 + |\beta|^2 = 1$ is fulfilled thanks to the state vector normalization condition.

After measurement, depending on the measurement outcome 0 or 1, the state would change respectively to:

$$\frac{M_0 |\phi\rangle}{|\alpha|} = \frac{\alpha}{|\alpha|} |0\rangle$$
$$\frac{M_1 |\phi\rangle}{|\beta|} = \frac{\beta}{|\beta|} |1\rangle$$

Multipliers like $\frac{\alpha}{|\alpha|}$ which have modulus one, can be ignored so the two post-measurement states are effectively $|0\rangle$ and $|1\rangle$. Thus, in this example, if we measured a 0, the post-measurement vector will be $|0\rangle$ and viceversa. Any measurements performed after the first one will yield exactly the same result. This behaviour is called *qubit collapsing*: the measurement changes state of a qubit, collapsing it from its superposition of $|0\rangle$ and $|1\rangle$ to the specific state consistent with the measurement result.

There is a special class of measurements worth pointing out known as **projective measurements**. Many applications of quantum computation work primarily with projective measurements.

Definition 2.19 (Projective measurement)**.** A projective measurement is described by an observable, $\mathcal{M}$, a Hermitian operator on the state space of the system being observed. The observable has a spectral decomposition

$$\mathcal{M} = \sum_m m P_m$$

where P_m is the projector onto the eigenspace of $\mathcal{M}$ with the eigenvalue m. The possible outcomes of the measurement correspond to the eigenvalues, m, of the observable. Upon measuring the state $|\phi\rangle$, the probability of getting result m is given by

$$p(m) = \langle\phi|P_m|\phi\rangle$$

Given that outcome m occured, the state of the quantum system immediately after the measurement is

$$\frac{P_m |\phi\rangle}{\sqrt{p(m)}}$$

$$\phi \equiv \text{State before measurement} \overset{\text{measurement}}{\longrightarrow} \frac{P_m |\phi\rangle}{\sqrt{p(m)}} \equiv \text{state after measurement}$$

Projective measurements can be understood as a special case of Postulate 4. In addition, we can quickly realise that projective measurements are orthogonal ($\mathcal{M}_m \mathcal{M}_{m'} = \delta_{mm'} \mathcal{M}_m$) and satisfy the completeness equation ($\sum_m \mathcal{M}_m^\dagger \mathcal{M}_m = \mathbb{1}$). Consequently, Postulate 4 reduces to a projective measurement.

These kind of measurements have nice properties and simplify the calculation of average values of the measurements. By definition, the average value of a measurement is the expected value, which for a discrete variable is a weighted average of the values of the variable, with weights given by the probabilities of each value:

$$\begin{aligned}\mathbf{E}(\mathcal{M}) &= \sum_m mp(m) \\ &= \sum_m m \langle\phi|P_m|\phi\rangle \\ &= \langle\phi| \left(\sum_m mP_m\right) |\phi\rangle \\ &= \langle\phi|\mathcal{M}|\phi\rangle\end{aligned}$$

A last remark regarding projective measurements is that usually, rather than giving an observable $\mathcal{M}$ to describe a projective measurement, we can provide a complete set of orthogonal projectors P_m satisfying the relations $\sum_m P_m = \mathbb{1}$ and $P_m P_{m'} = \delta_{mm'} P_m$. The corresponding observable implicitely defined by its projectors is $\mathcal{M} = \sum_m mP_m$. The notion of "measuring in a basis $\{|m\rangle\}$" is also commonly used. Considering $\{|m\rangle\}$ that form an orthonormal basis, this measurement simply performs the projective measurement with projectors $P_m = |m\rangle\langle m|$ and thus, the observable is $\mathcal{M} = \sum_m m\,|m\rangle\langle m|$.

To conclude this section, the most important aspect to grasp from this postulate is the big difference between classic and quantum measurements. The measured state changes after the measurement. This means that the mere act of observing the system interferes with it. This intriguing aspect is one of the fundamentals of quantum mechanics.

2.2.5. Composite systems

So far, we have worked with quantum systems of one qubit. But the real power of quantum computing is exploited when we are operating with composite quantum systems made up of two (or more) distinct qubits. We introduce the last postulate, which allows us to comprehend the state space associated with a physical system comprised of smaller subsystems. This will enable us to study the particular case of quantum computing systems made up of multiple qubits.

First we need to introduce the concept of **tensor product**. We will work with finite Hilbert spaces because such are the ones we work with in the quantum computing world. In this thesis we won't be concerned with the general infinite-dimensional case. We will follow the definition from [Sch16]

Definition 2.20 (Tensor product)**.** Let $\mathbb{H}_A$ and $\mathbb{H}_B$ finite Hilbert spaces of dimension A and B respectively, $|\varphi\rangle \in \mathbb{H}_A$ and $|\phi\rangle \in \mathbb{H}_B$ vectors in these and define

$$\begin{aligned}|\varphi\rangle \otimes |\phi\rangle : \mathbb{H}_A \times \mathbb{H}_B &\longrightarrow \mathbb{C} \\ (\xi, \nu) &\longmapsto \langle\xi|\varphi\rangle_{\mathbb{H}_A} \langle\nu|\phi\rangle_{\mathbb{H}_B}\end{aligned}$$

This map is anti-linear in ξ and ν and continuous. We define the set of all such maps and denote it by

$$\mathbb{H}_A \otimes \mathbb{H}_B := \{\Phi : \mathbb{H}_A \times \mathbb{H}_B \longrightarrow \mathbb{C} \mid \text{anti-linear and continuous}\}$$

This is a vector space over $\mathbb{C}$ since

- The null-map is the null-vector

- Given $\Phi \in \mathbb{H}_A \otimes \mathbb{H}_B$, $-\Phi$ is the additive-inverse vector for Φ
- For $\Phi_1, \Phi_2 \in \mathbb{H}_A \otimes \mathbb{H}_B$ and $a, b \in \mathbb{C}$, the map defined by

$$(a\Phi_1 + b\Phi_2)\,(\xi, \nu) := a\Phi_1(\xi, \nu) + b\Phi_2(\xi, \nu) \in \mathbb{H}_A \otimes \mathbb{H}_B$$

According to the previous definition of tensor product, $|\varphi\rangle \otimes |\phi\rangle$ is a vector in the vector space of the anti-linear and continuous maps $\mathbb{H}_A \otimes \mathbb{H}_B$ from $\mathbb{H}_A \times \mathbb{H}_B$ to $\mathbb{C}$.

Beyond the abstract construction, in practice, the tensor product of two column vectors $|a\rangle \in \mathbb{H}_A$ of dimension $A \times 1$ and $|b\rangle \in \mathbb{H}_B$ of dimension $B \times 1$ is a column vector of dimension $A \cdot B \times 1$ denoted $|a\rangle \otimes |b\rangle$, $|a\rangle\,|b\rangle$ or $|ab\rangle$.

$$|a\rangle = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_A \end{bmatrix} \in \mathbb{H}_A, \quad |b\rangle = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_B \end{bmatrix} \in \mathbb{H}_B$$

$$|a\rangle \otimes |b\rangle = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_A \end{bmatrix} \otimes \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_B \end{bmatrix} = \begin{bmatrix} a_1 \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_B \end{bmatrix} \\ a_2 \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_B \end{bmatrix} \\ \vdots \\ a_A \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_B \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_1 b_1 \\ a_1 b_2 \\ \vdots \\ a_1 b_B \\ a_2 b_1 \\ a_2 b_2 \\ \vdots \\ a_2 b_B \\ \vdots \\ a_A b_1 \\ a_A b_2 \\ \vdots \\ a_A b_B \end{bmatrix}$$

It is worth pointing out that, by definition, the tensor product satisfies the following properties:

- Given $\alpha \in \mathbb{C}$ and $|\varphi\rangle \in \mathbb{H}_A$ and $|\phi\rangle \in \mathbb{H}_B$:

$$(\alpha\,|\varphi\rangle) \otimes |\phi\rangle = |\varphi\rangle \otimes (\alpha\,|\phi\rangle) = \alpha(|\varphi\rangle \otimes |\phi\rangle)$$

- Given $\alpha, \beta \in \mathbb{C}$ and $|\varphi\rangle \in \mathbb{H}_A$ and $|\phi\rangle \in \mathbb{H}_B$:

$$\alpha(|\varphi\rangle \otimes |\phi\rangle) + \beta(|\varphi\rangle \otimes |\phi\rangle) = (\alpha + \beta)\,|\varphi\rangle \otimes |\phi\rangle$$

- For any $|\varphi_1\rangle, |\varphi_2\rangle \in \mathbb{H}_A$ and $|\phi\rangle \in \mathbb{H}_B$:

$$(|\varphi_1\rangle + |\varphi_2\rangle) \otimes |\phi\rangle = |\varphi_1\rangle \otimes |\phi\rangle + |\varphi_2\rangle \otimes |\phi\rangle$$

- For any $|\varphi\rangle \in \mathbb{H}_A$ and $|\phi_1\rangle, |\phi_2\rangle \in \mathbb{H}_B$:

$$|\varphi\rangle \otimes (|\phi_1\rangle + |\phi_2\rangle) = |\varphi\rangle \otimes |\phi_1\rangle + |\varphi\rangle \otimes |\phi_2\rangle$$

Our goal now is to provide the tensor product space $\mathbb{H}_A \otimes \mathbb{H}_B$ with an inner product.

First, for vectors of form $|\varphi_k\rangle \otimes |\phi_k\rangle \in \mathbb{H}_A \otimes \mathbb{H}_B$ with $|\varphi_k\rangle \in \mathbb{H}_A$ and $|\phi_k\rangle \in \mathbb{H}_B$ for $k = 1, 2$ we define the inner product:

$$\langle \varphi_1 \otimes \phi_1 | \varphi_2 \otimes \phi_2 \rangle := \langle \varphi_1 | \varphi_2 \rangle_{\mathbb{H}_A} \langle \phi_1 | \phi_2 \rangle_{\mathbb{H}_B}$$

To define the inner product for all $\Phi \in \mathbb{H}_A \otimes \mathbb{H}_B$ we consider $\{u_i\}_{i=1}^{A}$ and $\{v_j\}_{i=1}^{B}$ to be orthonormal basis of $\mathbb{H}_A$ and $\mathbb{H}_B$ respectively. The inner product of the spaces $\mathbb{H}_A$ and $\mathbb{H}_B$ can be used to extend an inner product in the tensor space $\mathbb{H}_A \otimes \mathbb{H}_B$ noting that the set $\{|u_i\rangle \otimes |v_j\rangle\}$ is an orthonormal basis because

- It is orthogonal

$$\langle u_i \otimes v_j | u_k \otimes v_l \rangle = \langle u_i | u_k \rangle \langle v_j | v_l \rangle = \delta_{ij}\delta_{jl} = \begin{cases} 1, \text{ if } i = j \text{ and } k = l \\ 0, \text{ otherwise} \end{cases}$$

- It spans the space: considering an arbitrary $\Phi \in \mathbb{H}_A \otimes \mathbb{H}_B$:

$$\begin{aligned}\Phi(\xi, \nu) &= \Phi\left(\sum_i \langle u_i|\xi\rangle |u_i\rangle, \sum_j \langle v_j|\nu\rangle |v_j\rangle\right) \\ &= \sum_{i,j} \Phi(|u_i\rangle, |v_j\rangle) \langle u_i|\xi\rangle \langle v_j|\nu\rangle \\ &= \sum_{i,j} \Phi_{ij}(|u_i\rangle \otimes |v_j\rangle)(\xi, \nu)\end{aligned}$$

where $\Phi_{ij} = \Phi(|u_i\rangle, |v_j\rangle) \in \mathbb{C}$ is a complex scalar. This proves that every vector $|\Phi\rangle \in \mathbb{H}_A \otimes \mathbb{H}_B$ can be written as a linear combination of the elements of the basis $\{|u_i\rangle \otimes |v_j\rangle\}$ in the form:

$$|\Phi\rangle = \sum_{i,j} \Phi_{ij}(|u_i\rangle \otimes |v_j\rangle)$$

Proposition 2.2. *Let $\mathbb{H}_A$ and $\mathbb{H}_B$ be Hilbert spaces with orthonormal basis $\{u_i\}_{i=1}^{A}$ and $\{v_j\}_{j=1}^{B}$ respectively. Then, $\{|u_i\rangle \otimes |v_j\rangle\}_{i,j}$ is an orthonormal basis of $\mathbb{H}_A \otimes \mathbb{H}_B$ and the tensor product space is a Hilbert space provided with the inner product given by*

$$\begin{aligned}\langle \Phi | \Psi \rangle &= \langle \sum_{i,j} \alpha_{ij}(|u_i\rangle \otimes |v_j\rangle \,|\, \sum_{k,l} \beta_{kl}(|u_k\rangle \otimes |v_l\rangle\rangle \\ &= \sum_{i,j,k,l} \alpha_{ij}^{*}\beta_{kl} \langle u_i \otimes v_j | u_j \otimes v_k \rangle \\ &= \sum_{i,j} \alpha_{ij}^{*}\beta_{ij}\end{aligned}$$

for all $\Phi, \Psi \in \mathbb{H}_A \otimes \mathbb{H}_B$

Proof. $\{|u_i\rangle \otimes |v_j\rangle\}_{i,j}$ is an orthonormal basis of $\mathbb{H}_A \otimes \mathbb{H}_B$ and has been proven above the proposition.

It is necessary to verify that $\langle\Phi|\Psi\rangle$ is positive-definite.

For any $|\Phi\rangle = \sum_{i,j} \alpha_{ij}(|u_i\rangle \otimes |v_j\rangle) \in \mathbb{H}_A \otimes \mathbb{H}_B$ we have

$$\langle\Phi|\Phi\rangle = \sum_{i,j} |\alpha_{ij}|^2 \geq 0$$

Therefore, $\langle\Phi|\Phi\rangle = 0$ if and only if $\alpha_{ij} = 0$ for all $i = 1, ..., A$, $j = 1, ..., B$, which happens if and only if $|\Phi\rangle = 0$ □

Now, we will see that the tensor product space is a Hilbert space. $\mathbb{H}_A \otimes \mathbb{H}_B$ is a complex vector space with inner product, which induces a norm:

$$||\Phi||^2 = \langle\Phi|\Phi\rangle = \sum_{i,j} \alpha_{ij}^* \alpha_{ij} = \sum_{i,j} |\alpha_{ij}|^2$$

for all $|\Phi\rangle = \sum_{i,j} \alpha_{ij}(|u_i\rangle \otimes |v_j\rangle) \in \mathbb{H}_A \otimes \mathbb{H}_B$.

For finite-dimensional spaces (which is in the context we are working), $\mathbb{H}_A \otimes \mathbb{H}_B$ is complete in this norm and thus, a Hilbert space.

Example 2.2. As an example, consider $\mathbb{H}_A = \mathbb{H}_B \cong \mathbb{C}^2$ with the canonical basis $\{|0\rangle, |1\rangle\} = \{(1\,0)^T, (0\,1)^T\}$.

Then, $\mathbb{H}_A \otimes \mathbb{H}_B \cong \mathbb{C}^4$ and we have the basis

$$|0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \quad |0\rangle \otimes |1\rangle = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \quad |1\rangle \otimes |0\rangle = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \quad |1\rangle \otimes |1\rangle = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

which corresponds to the canonical base of $\mathbb{C}^4$. This base is usually denoted as $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ or even $\{|0\rangle, |1\rangle, |2\rangle, |3\rangle\}$

The general expression of a state of such system would be

$$|\Phi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$$

where $\alpha_{00}, \alpha_{01}, \alpha_{10}, \alpha_{11}$ are complex amplitudes that verify $|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$.

If we measure in the basis we have just specified, both qubits of the generic state $|\Phi\rangle$ we will obtain 00 with probability $|\alpha_{00}|^2$, 01 with probability $|\alpha_{01}|^2$, 10 with probability $|\alpha_{10}|^2$ and 11 with probability $|\alpha_{11}|^2$. In all these cases, the state would collapse to the state corresponding to the outcome of the measurement, just like in one-qubit systems.

We can also measure one of the qubits. We can consider the projectors $P_0 = |00\rangle\langle 00| + |01\rangle\langle 01|$ and $P_1 = |01\rangle\langle 01| + |11\rangle\langle 11|$. If we measure the first qubit, the outcome would be 0 with probability

$$p_0 = \langle\Phi|P_0|\Phi\rangle = |\alpha_{00}|^2 + |\alpha_{01}|^2$$

This probability comes from he coefficients associated with the first qubit being 0. After performing the measurement, if the outcome is 0, the qubit would collapse to the state

$$\frac{P_0 |\Phi\rangle}{\sqrt{p_0}} = \frac{\alpha_{00} |00\rangle + \alpha_{01} |01\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}}$$

The denominator $\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}$ keeps the state normalized.

After measuring the first qubit, we can measure the second qubit. The outcome for the second qubit can be 0 or 1 with respective probabilities

$$p'_0 = \frac{|\alpha_{00}|^2}{|\alpha_{00}|^2 + |\alpha_{01}|^2}$$
$$p'_1 = \frac{|\alpha_{01}|^2}{|\alpha_{00}|^2 + |\alpha_{01}|^2}$$

The sum of probabilities of the different outcomes for the second qubit must sum up to 1 and it is worth noting that they do: $p'_0 + p'_1 = 1$

The situation is analogous in which the result of the measurement of the first qubit is 1 with probability

$$p_1 = \langle\Phi|P_1|\Phi\rangle = |\alpha_{10}|^2 + |\alpha_{11}|^2$$

The sum of probabilities of the different outcomes for the first qubit also add up to one thanks to the normalization condition of the vector $|\Phi\rangle$

$$p_0 + p_1 = |\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1$$

Finally, we can extend the concept of linear operator to the tensor product.

If A, B are operators acting on $\mathbb{H}_A$ and $\mathbb{H}_B$ respectively, we define the operator $A \otimes B$ acting on $\mathbb{H}_A \otimes \mathbb{H}_B$ by

$$(A \otimes B)(|\varphi\rangle \otimes |\phi\rangle) = (A |\varphi\rangle) \otimes (B |\phi\rangle) \quad \forall |\varphi\rangle \otimes |\phi\rangle \in \mathbb{H}_A \otimes \mathbb{H}_B$$

We can extend this to every linear operator C on $\mathbb{H}_A \otimes \mathbb{H}_B$ considering

$$C = \sum_{i,j} c_{ij} A_i \otimes B_j$$

with A_i and B_j operators on $\mathbb{H}_A$ and $\mathbb{H}_B$ respectively. By definition,

$$C(|\varphi\rangle \otimes |\phi\rangle) = \sum_{i,j} c_{ij} A_i \otimes B_j(|\varphi\rangle \otimes |\phi\rangle) = \sum_{i,j} c_{ij} A_i |\varphi\rangle \otimes B_j |\phi\rangle$$

Now that we are familiar enough with the tensor product, we will go through postulate 5 regarding composite systems.

Postulate 5

The state space of a composite physical system is the tensor product of the state spaces of the component physical systems. Moreover, if we have systems numbered 1 through n, and system number i is prepared in the state $|\phi_i\rangle$, then the joint state of the total system is $|\phi_1\rangle \otimes |\phi_2\rangle \otimes ... \otimes |\phi_n\rangle$

It will help us understand the composite systems to understand the intuition of the introduction of the tensor product in this context. We would expect that there exists *some canonical way* of describing composite systems in quantum mechanics, much like how the Cartesian product is employed in vector spaces. The idea comes from the superposition principle of quantum mechanics: if $|\phi\rangle$, $|\psi\rangle$ are two states of a quantum system, then any superposition $\alpha|\phi\rangle + \beta|\psi\rangle$ should also be a state of a quantum system, where $|\alpha|^2 + |\beta|^2 = 1$. For composite systems, an intuitive approach is that if $|\Phi\rangle$ is a state of a system A and $|\Psi\rangle$ is a state of a system B, then there should be some corresponding state in the composite system AB: $|\Phi\rangle|\Psi\rangle$ which naturally takes us to the tensor product.

Indeed, the tensor product operation can be repeated and a vector $|\varphi\rangle$ can be tensored with itself n times: $|\varphi\rangle \otimes \overset{n}{\cdots} \otimes |\varphi\rangle$ and it is denoted as $|\varphi\rangle^{\otimes n}$. In the same way, we can perform n times the tensor product of a Hilbert space: $\mathbb{H} \otimes \overset{n}{\cdots} \otimes \mathbb{H}$ represented by $\mathbb{H}^{\otimes n}$. Using the standard basis of the state space $\mathbb{H} = \mathbb{C}^2$ to build a basis of higher tensor powers $\mathbb{H}^{\otimes n}$ as in E.g. 2.2, it can be generalized and leads to the computational basis notion of multiple qubit systems.

Definition 2.21. Given $x \in \mathbb{N}, x < 2^n$, it can be expressed in binary in the form

$$x = \sum_{i=0}^{n-1} x_i 2^i$$

for $x_0, x_1, ..., x_{n-1} \in \{0, 1\}$ coefficients of the binary representation of x. For each such x, we define a vector $|x\rangle \in \mathbb{H}^{\otimes n}$ as

$$|x\rangle := |x_{n-1}...x_1x_0\rangle := |x_{n-1}\rangle \otimes \cdots \otimes |x_1\rangle \otimes |x_0\rangle \in \mathbb{H}^{\otimes n}$$

Note that,

$$\begin{aligned} \mathbb{H}^{\otimes n} &= \mathbb{H}_{n-1} \otimes \cdots \otimes \mathbb{H}_j \otimes \cdots \otimes \mathbb{H}_0 \\ &\ni |x_{n-1}\rangle \otimes \cdots \otimes |x_j\rangle \otimes \cdots \otimes |x_0\rangle \end{aligned}$$

Proposition 2.3. *The set of vectors* $\{|x\rangle \in \mathbb{H}^{\otimes n} \mid x \in \mathbb{N},\ x < 2^n\}$ *forms an orthonormal basis in* $\mathbb{H}^{\otimes n}$. *It receives the name of* ***computational basis****.*

Proof. We consider any $|u\rangle, |v\rangle \in \{|x\rangle \in \mathbb{H}^{\otimes n} \mid x \in \mathbb{N},\ x < 2^n\} = \mathcal{B}$, We will check if they are orthogonal:

$$\begin{aligned} \langle u|v\rangle &\overset{1}{=} \langle u_{n-1}...u_0|v_{n-1}...v_0\rangle \\ &\overset{2}{=} \prod_{i=0}^{n-1} \langle u_i|v_i\rangle = \begin{cases} 1 & \text{if} \quad u_i = v_i \quad \forall i \\ 0 & \text{if} \quad \text{otherwise} \end{cases} \end{aligned}$$

where we use:

1. The binary representation of the elements of $\mathcal{B}$:

$$u = \sum_{i=0}^{n-1} u_i 2^i$$

   Analogously for $|v\rangle$.

2. We generalise this definition of inner product

$$\langle \varphi_1 \otimes \phi_1 | \varphi_2 \otimes \phi_2 \rangle := \langle \varphi_1 | \varphi_2 \rangle_{\mathbb{H}_A} \langle \phi_1 \otimes \phi_2 \rangle_{\mathbb{H}_B}$$

   to the n-fold tensor product vectors.

Therefore, $\mathcal{B}$ forms a set of 2^n orthonormal vectors in $\mathbb{H}^{\otimes n}$. This orthonormal set is complete because the number of orthonormal vectors is equal to the dimension of the Hilbert space $\mathbb{H}^{\otimes n}$, and thus, $\mathcal{B}$ constitutes an orthonormal basis for this Hilbert space. □

Corolary 2.1. *The tensor product of* $\mathbb{C}^2$ *with itself n times:* $\mathbb{C}^2 \otimes \overset{n}{\cdots} \otimes \mathbb{C}^2$ *is isomorphic to* $\mathbb{C}^{2^n}$

Proof. We want to proof that $H = \mathbb{C}^2 \otimes \overset{n}{\cdots} \otimes \mathbb{C}^2$ is isomorphic to $\mathbb{C}^{2^n}$. We can observe that $dim(H) = dim(\mathbb{C}^2)^n = 2^n$ and thus, H is a finite normed space of dimension 2^n. The Hausdorff Theorem, Theorem A.2, assures us that there is an isomorphism between H and the canonical complex vector space of the same dimension: $\mathbb{C}^{2^n}$. □

2.3. Entanglement

Postulate's 5 introduction to composite systems allows us to define the phenomenon of **entanglement**.

In order to define when a quantum system is entangled, we will first define when it is not entangled.

A state $|\psi\rangle$ is a product state if it can be expressed as the tensor product of two other states: $|\psi_1\rangle$ and $|\psi_2\rangle$:

$$|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle$$

If $|\psi\rangle$ is not a product state, in other words, it is not possible to decompose it as the tensor product of other states, then we say that it is **entangled**.

Example 2.3.

- $|10\rangle$ is a product state since $|10\rangle = |1\rangle \otimes |0\rangle$ by definition.
- $\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$ is also a product state because we have factor $|0\rangle$ on the second qubit:

$$\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) = \left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right)|0\rangle$$

- $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is an entangled state. We will reason by contradiction and suppose that it is possible to write it as a product of two one-qubit states: $a|0\rangle + b|1\rangle$ and $c|0\rangle + d|1\rangle$

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = (a|0\rangle + b|1\rangle)(c|0\rangle + d|1\rangle) = ac|00\rangle + ad|01\rangle + bc|01\rangle + bd|11\rangle$$

Discussing the system, we realise that $ad = 0$ because there is no term $|01\rangle$, which implies that either $a = 0$, but then $ac = 0$ or $d = 0$, which implies $bd = 0$. In both cases, we arrive to a contradiction. Therefore, the state is entangled and it is known as Bell State.

Indeed, the computational basis consists of product states in the way we have defined it in the previous section. Although we can easily realise that, in particular, the four-dimensional space $\mathbb{H}^{\otimes 2}$ also admits other orthonormal basis. A remarkable one of them is the **Bell Basis**, formed by the vectors

$$|\Phi^{\pm}\rangle = \frac{1}{\sqrt{2}}(|00\rangle \pm |11\rangle)$$
$$|\Psi^{\pm}\rangle = \frac{1}{\sqrt{2}}(|01\rangle \pm |10\rangle)$$

The Bell basis does not consist of product states, but entangled states as we proved for $|\Phi^{+}\rangle$ in the example above.

2.4. No-cloning theorem

A really basic operation that comes to our mind when we think of classical computation is copying a register, or a variable, or a file. We would like to have a procedure to make exact replicas or *clones* of a state. However, another particular property of quantum systems is that cloning arbitrary quantum states is impossible.

If we intend to build a device to copy a quantum state, there are only two possible quantum operations to manipulate the composite system: unitary evolutions and measurements. If we perform an observation, the original state will irreversibly collapse into one of the observable's eigenstates, corrupting the information contained in the qubit(s). Clearly, this outcome is not what we want. Instead, we should use unitary operations.

Given $\mathbb{H}$ a state space, a random $|\psi\rangle \in \mathbb{H}$ the state to be copied and $|b\rangle \in \mathbb{H}$ where we would like to copy $|\psi\rangle$, in other words, $|b\rangle$ is "a blank piece of paper". Then, supposing we are able to control the Hamiltonian to make the state evolve this way, our copying operation would be:

$$\begin{aligned} U : \mathbb{H}^{\otimes 2} &\longrightarrow \mathbb{H}^{\otimes 2} \\ |\psi\rangle \otimes |b\rangle &\longmapsto U(|\psi\rangle \otimes |b\rangle) = |\psi\rangle \otimes |\psi\rangle \end{aligned}$$

where U is a unitary operator.

The result known as no-cloning theorem assures this is impossible. Next, it will be stated and proved.

Theorem 2.2 (No-cloning theorem). *There is no unitary operator $U : \mathbb{H}^{\otimes 2} \longrightarrow \mathbb{H}^{\otimes 2}$ and state $|b\rangle \in \mathbb{H}$ such that for any $|\psi\rangle$ the copying procedure is performed:*

$$U(|\psi\rangle \otimes |b\rangle) = |\psi\rangle \otimes |\psi\rangle$$

Proof. We will prove this theorem by contradiction, so let's suppose it exists such unitary operator U and such "blank state" $|b\rangle$. Now, we take two $|\psi\rangle, |\phi\rangle$ for which this cloning

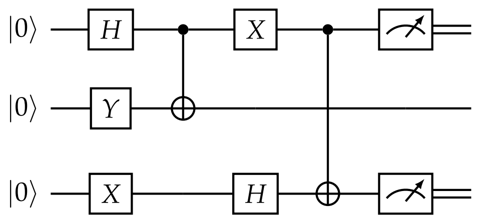


Figure 2.1.: Example of quantum circuit

procedure works. Then, we have:

$$U(|\psi\rangle \otimes |b\rangle) = |\psi\rangle \otimes |\psi\rangle$$
$$U(|\phi\rangle \otimes |b\rangle) = |\phi\rangle \otimes |\phi\rangle$$

We take the inner product of these two equations,

$$\langle U(|\psi\rangle \otimes |b\rangle) |\, U(|\phi\rangle \otimes |b\rangle)\rangle = \langle |\psi\rangle \otimes |\psi\rangle \mid |\phi\rangle \otimes |\phi\rangle\rangle$$

we proceed using that U is an unitary operator and the definition of inner product in the composite space Def. 2.2.5,

$$\langle (|\psi\rangle \otimes |b\rangle) | (|\phi\rangle \otimes |b\rangle)\rangle = \langle \psi|\phi\rangle \langle \psi|\phi\rangle$$
$$\langle \psi|\phi\rangle \langle b|b\rangle = \langle \psi|\phi\rangle^2$$

where we have used again the inner product in the composite space. Since $|b\rangle$ is a valid state, it is a normalized vector and thus, $\langle b|b\rangle = 1$. We finally obtain

$$\langle \psi|\phi\rangle = \langle \psi|\phi\rangle^2$$

This is only possible if $\langle \psi|\phi\rangle = 0$ or ± 1, that is, if $|\psi\rangle$ and $|\phi\rangle$ are either the same state or they are orthogonal. But $|\psi\rangle$ and $|\phi\rangle$ are arbitrary state, therefore, a single universal U cannot clone a general quantum state. □

2.5. Circuit model

There are several quantum computing models, such as quantum Turing machines and adiabatic quantum computing. However, the most popular model is the **quantum circuit model**. For the purposes of this thesis, we shall stay within the quantum circuit model of computation. It provides a robust and powerful framework for computation that exploits the properties of quantum mechanics.

Every computation has three elements: data, operations and output. In the quantum circuit model, these correspond to the following concepts: qubits, quantum gates and measurements. Quantum circuits have a diagram representation, we have an overview of it at Figure 2.1

The three horizontal lines are *wires* and they represent the qubits. They are all prepared to the initial state $|0\rangle$. The wires carry the information around the circuit and the *quantum gates* perform manipulations of the information. They are represented by boxes with the name or initial of the operation inside. In the figure, we start applying the Hadamard, X and Y gates, which we will introduce next. Controlled-NOT operations are also performed. Finally, a measurement operation is performed on the top and bottom qubits. Measurements are portrayed in the circuit using the gauge symbol. After performing a measurement, the wires are represented with double lines to indicate that the state of the qubit has collapsed to a classical value, and from then on we are working with classical bits, not quantum data.

Previously in this chapter we have defined the concept of qubit. We can develop the theory of quantum computation independently from its possible physical implementations thanks to its mathematical formulation. Now, we will introduce another useful way to think about qubits taking into account its geometric representation. We defined qubits as normalized vectors in $\mathbb{C}^2$, thus the state of a qubit is described with two complex numbers, each of them having two real components. This would lead to a four-dimensional real space representation. Instead, all possible states of a qubit can be represented in a sphere, using the polar representation of a complex number:

$$z \in \mathbb{C},\, z = |z|e^{i\alpha} \text{ with } \alpha \in [0, 2\pi]$$

The amplitudes of a quantum state $|\phi\rangle = a\,|0\rangle + b\,|1\rangle$ can be written in polar coordinates:

$$a = |a|e^{i\alpha_1} \quad b = |b|e^{i\alpha_2}$$

Thanks to the normalization condition, we know that $0 \leq |a|, |b| \leq 1$, therefore there must exist $\theta \in [0, \pi]$ such that $\cos(\theta/2) = |a|$ and $\sin(\theta/2) = |b|$. Then, a generic quantum state can be parametrised as

$$|\phi\rangle = \cos(\theta/2)e^{i\alpha_1}\,|0\rangle + \sin(\theta/2)e^{i\alpha_2}\,|1\rangle$$

Since we can multiply $|\phi\rangle$ by a complex number of modulus 1 without changing its state (known as global phase), we can multiply $|\phi\rangle$ by $e^{-i\alpha_1}$ and obtain:

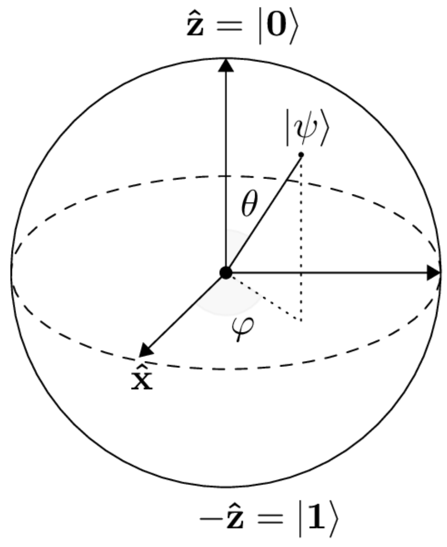
$$|\phi\rangle = \cos(\theta/2)\,|0\rangle + \sin(\theta/2)e^{i\varphi}\,|1\rangle$$

where $\varphi = \alpha_2 - \alpha_1 \in [0, 2\pi]$

Interpreting the numbers θ and φ as the polar and azimuthal angle, respectively, define a point in the unit three dimensional sphere called Bloch sphere. $|0\rangle$ is mapped to the North pole and $|1\rangle$ to the South pole. The Bloch sphere is illustrated in Figure 2.2. In general, states that are orthogonal with respect to the inner product are antipodal on the sphere. This visualization is useful for interpreting the state of a single qubit. Many operations on single qubits are neatly described within the Bloch sphere. However, this intuition is limited to one qubit because there is no simple generalization of the Bloch sphere known for multiple qubits.

2.5.1. Quantum gates

Postulates in subsection 2.2.4 and subsection 2.2.3 highlighted the evolution of quantum states as well as their measurement. In quantum computing, time evolution operators are performed by quantum logic gates and measurements are done with respect to the computational basis. Next we will look at quantum gates.



$\hat{\mathbf{z}} = |\mathbf{0}\rangle$
$|\psi\rangle$
θ
$\hat{\mathbf{y}}$
φ
$\hat{\mathbf{x}}$
$-\hat{\mathbf{z}} = |\mathbf{1}\rangle$

Figure 2.2.: The Bloch sphere representation of a qubit

One-qubit quantum gates are formally described by 2×2 unitary transformations.

NOT gate

The NOT gate, so-called X gate , is the quantum equivalent of the classical NOT gate. It is represented by the matrix

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

It acts as:

$$|0\rangle \longmapsto |1\rangle$$
$$|1\rangle \longmapsto |0\rangle$$

In other words, applied to a generic single-qubit state, the X gate swaps the amplitudes of the $|0\rangle$ and $|1\rangle$ components.

Indeed, this gate corresponds to the Pauli X matrix. Coming back to the Bloch sphere, the X gate acts like a rotation of π radians around the X axis of the Bloch sphere. We can generalize this behavior to obtain rotations of any angle around any axis of the Bloch sphere.

For the X axis we may define

$$R_X(\theta) = e^{-i\frac{\theta}{2}X} = \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$$

Analogously, for Y and Z axis:

$$R_Y(\theta) = e^{-i\frac{\theta}{2}Y} = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$$
$$R_Z(\theta) = e^{-i\frac{\theta}{2}Z} = \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix}$$

When $\theta = \pi$ the resulting matrices are known as **Pauli matrices**. Each of them define a quantum gate known as

$$Y = R_Y(\pi) = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$
$$Z = R_Z(\pi) = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Another known gate coming from rotations of the Bloch sphere is the S gate which results from a $\frac{\theta}{2}$ rotation around the Z axis, known as **phase gate**:

$$S = R_Z(\frac{\pi}{2}) = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$$

Hadamard gate

Another remarkable gate is the Hadamard gate, denoted by H (which is not to be confused with the Hamiltonian also denoted by H). It is represented by the matrix

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The Hadamard gate effect on the basis states $|0\rangle$ and $|1\rangle$ is

$$|0\rangle \longmapsto \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$
$$|1\rangle \longmapsto \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

Additionally, it is useful to keep in mind that

$$H = \frac{X+Z}{\sqrt{2}}$$

The role of the H gate is to create superposition of qubits.

For composite systems, we can guess that the tensor product will have a great importance again. The simplest way to build quantum gates on composite systems are tensor products of 2×2 unitary operators. In the case of two-qubit systems, the corresponding gates are 4×4 unitary operators constructed from the tensor product of two one-qubit quntum gates. Let U_1 and U_2 be one-qubit gates, the matrix of the gate $U_1 \otimes U_2$ is given by the tensor product of the matrices associated to U_1 and U_2:

$$U_1 \otimes U_2 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \otimes \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} & a_{12} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \\ a_{21} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} & a_{22} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \end{bmatrix}$$
$$= \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{bmatrix}$$

By taking tensor products of one-qubit gates, we can only obtain operations that act on each qubit individually, but there are many unitary matrices that cannot be written as the tensor product of other matrices. A remarkable one is the controlled-NOT gate.

CNOT gate
The CNOT gate, known as controlled NOT gate is given by the unitary matrix

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

It takes two input qubits, known as the control qubit and the target qubit. The state of the target qubit is changed, based on the value of the control qubit. This is the principle of controlled gates. In particular, the CNOT gate, performs the NOT operation when the control qubit is in state $|1\rangle$, otherwise, it remains unchanged.

Next we will show how this gate acts on the elements of the two-qubit computational basis

$$CNOT\,|00\rangle = |00\rangle\,, \quad CNOT\,|01\rangle = |01\rangle\,, \quad CNOT\,|10\rangle = |11\rangle\,, \quad CNOT\,|11\rangle = |10\rangle$$

More generally, given an arbitrary qubit unitary operation denoted by U. A controlled U operation is a two qubit transformation, again with a control and a target qubit. If the control qubit is set, then U is applied to the target qubit, otherwise the target qubit is left alone.

The previously presented quantum gates are summarised at Table 2.1.

Departing from this overview of quantum gates we have the basis to build more complex gates and circuits.

2.6. Open quantum systems

As we have briefly mentioned in previous sections, the most widely treated and studied quantum systems are those called closed systems, which are the ones that only interact with themselves, isolated from the rest of the universe. By definition, it is a system which does not interchange information with another system. All the previous description of quantum mechanics was built in the realm of closed quantum systems. However, beyond theoretical level, closed quantum systems are not that relevant because no system can be completely isolated from its environment in the real world. Indeed, for this to happen, it would be

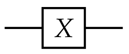
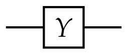
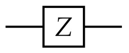
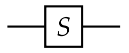
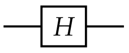
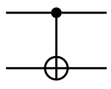
Gate	Circuit representation	Matrix
X-Pauli (NOT)		$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Y-Pauli		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Z-Pauli		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Phase		$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
Hadamard		$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
CNOT		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

Table 2.1.: Summary table of useful quantum logic gates and their representations

needed a sort of "infinite barrier" around the system in question that cannot be recreated in an experimental environment. Therefore, since no quantum system is completely isolated from its surroundings, it is important to develop a theoretical framework for treating these interactions in order to obtain an accurate understanding of quantum systems. We have decided to introduce the notion of open quantum systems, a quantum mechanical system that interacts with an external quantum system, which is known as the environment or bath. We will analyze its evolution considering that it is not isolated from the rest of the universe and the mutual actions between the system and the environment.

Definition 2.22 (Open quantum system)**.** An open quantum system is a quantum mechanical system S, which is associated to a Hilbert space $\mathbb{H}_S$, that is interacting with another quantum system E: the environment, which is associated to a Hilbert space $\mathbb{H}_E$. Let's assume that $\dim(\mathbb{H}_S) = n$ and $\dim(\mathbb{H}_E) = n$.
Therefore, S is a subsystem of the total system $(S + E)$, whose corresponding Hilbert space is the tensor product $\mathbb{H}_T = \mathbb{H}_S \otimes \mathbb{H}_E$, that is nm-dimensional.

Traditionally, the time evolution of quantum systems has been described through a unitary transformation that connects the states of the system at two points in time. However, this description becomes excessively restrictive when attempting to analyze the time evolution of an open quantum system. To understand how the open quantum system that we have just defined evolves in time we will denote H_s the open quantum system Hamiltonian, H_e the environment Hamiltonian and H_{se} the system-environment interaction Hamiltonian. Assuming that the entire open quantum system is a large closed system, its time evolution is governed by a unitary transformation generated by a global Hamiltonian given by

$$H = H_s \otimes I_e + I_s \otimes H_e + \alpha H_{se}$$

where I_s and I_e are the identity operators of the system and environment Hilbert spaces respectively, α is a coupling constant that depends on the interaction between the system and the environment.

Since the system $S + E$ evolves according to a unitary time evolution operator $U(t, t_0)$, the density operator of the composite system at an initial time t_0 is described by a density matrix $\rho(t_0)$. The density operator at a time t is given by $\rho(t) = U(t, t_0)\rho(t_0)U^\dagger(t, t_0)$. The quantum states of the subsystems S and E at time t are represented by their respective reduced density operators denoted by $\rho_S(t)$ and $\rho_E(t)$. These operators contain all the relevant statistical information for potential measurements performed on each subsystem. To obtain $\rho_S(t)$ and $\rho_E(t)$, it is needed to calculate the partial trace of $\rho(t)$ with respect to the degrees of freedom of E and S, respectively.

$$\begin{aligned}\rho_S(t) &= \mathrm{Tr}_E(\rho(t)) \\ \rho_E(t) &= \mathrm{Tr}_S(\rho(t))\end{aligned}$$

Definition 2.23 (Partial trace)**.** Let X, Y be finite vector spaces of dimension n and m respectively. The partial trace over Y is a linear operator defined as $\mathrm{Tr}_Y : \mathcal{L}(X \otimes Y) \longrightarrow \mathcal{X}$ such that

$$\mathrm{Tr}_Y(T \otimes U) = \mathrm{Tr}(U)T \quad \forall T \in \mathcal{L}(X), \forall U \in \mathcal{L}(Y)$$

Analogously, the partial trace over X is $\mathrm{Tr}_X : \mathcal{L}(X \otimes Y) \longrightarrow \mathcal{Y}$ such that

$$\mathrm{Tr}_X(T \otimes U) = \mathrm{Tr}(T)U \quad \forall T \in \mathcal{L}(X), \forall U \in \mathcal{L}(Y)$$

As the density operator of the environment is positive and normalized, it has a spectral decomposition in an orthonormal basis with non negative eigenvalues. Hence,

$$\rho_E(t) = \sum_v \lambda_v \left|v\right\rangle \left\langle v\right|$$

where λ_v are the eigenvalues and $\{|v\rangle\}$ are the corresponding orthonormal eigenvectors.

We can express the reduced density operator of S performing the partial trace in the orthonormal basis of the environment eigenstates:

$$\begin{aligned}\rho_s(t) &= \mathrm{Tr}_E\left[(U(t, t_0)\rho(t_0)U^\dagger(t, t_0)\right] \\ &= \sum_\mu \langle\mu|U(t, t_0)\rho(t_0)U^\dagger(t, t_0)|\mu\rangle\end{aligned}$$

We will assume, as is customary, that at the initial time the density operator of the closed system $S + E$ is separable, that is $\rho(t_0) = \rho_S(t_0) \otimes \rho_E(t_0)$ and that the reduced density operator $\rho_E(t_0)$ is equal to a certain fixed value $\rho_{E,0}$ regardless of the value of t_0. Then, the initial state of the total system at t_0 would be $\rho(t_0) = \rho_S(t_0) \otimes \rho_{E,0}$. It follows that

$$\begin{aligned}\rho_S(t) &= \mathrm{Tr}_E\left[U(t, t_0)\rho_S(t_0) \otimes \rho_{E,0}U^\dagger(t, t_0)\right] \\ &= \sum_{\mu v} K_{\mu v}(t, t_0)\rho_S(t_0)K^\dagger_{\mu v}(t, t_0)\end{aligned}$$

where $K_{\mu v}(t, t_0)$ are operators acting on the Hilbert space of S, known as Kraus operators.

They are given by

$$K_{\mu v}(t, t_0) = \sqrt{\lambda_v}\, \langle \mu | U(t, t_0) | v \rangle$$

The equation defining the system in terms of Kraus operators is called the Kraus Operator Sum representation.

Just as the Schrödinger equation describes how pure states evolve in time, the equation governing the temporal evolution of the density operator of a closed quantum system is the Liouville-von Neumann equation

$$\frac{d\rho(t)}{dt} = -\frac{i}{\hbar}\,[H(t), \rho(t)]$$

where $H(t)$ represents the Hamiltonian of the system, the square brackets denote the commutator of the operators enclosed within and $\hbar$ is Planck's constant.

By the Taylor expansion around $t = 0$ we have:

$$\rho(dt) = \rho(0) + \frac{d\rho(t)}{dt}|_0 + O(dt^2)$$

Using the Kraus Operator Sum representation,

$$\rho(dt) = \sum_{\alpha} K_\alpha(dt)\rho(0)K_\alpha^\dagger(dt)$$

where we have collected the μv indices into a single index α.

It can be shown ([Lid19]) that the reduced density operator of S fulfills the evolution equation

$$\frac{d\rho_S(t)}{dt}|_0 = -\frac{i}{\hbar}\,[H_S, \rho_S(0)] + \sum_{j=1}^{n^2-1} \gamma_j \left[L_j \rho_S(0) L_j^\dagger - \frac{1}{2}(L_j^\dagger L_j, \rho_S(0)) \right]$$

This result is valid as a short time expansion near $t = 0$. Making the assumption that the previous equation is valid for all times $t > 0$, which is essentially the Markovian limit, that states that there is no memory in the evolution, as manifested by the fact that the evolution resets every dt. This takes us to

$$\frac{d\rho_S(t)}{dt} = -\frac{i}{\hbar}\,[H_S(t), \rho_S(t)] + \sum_{j=1}^{n^2-1} \gamma_j(t) \left[L_j(t)\rho_S(t)L_j^\dagger(t) - \frac{1}{2}(L_j^\dagger(t)L_j(t), \rho_S(t)) \right] = \mathcal{L}\rho$$

where $H_S(t)$ is an operator playing the role of the Hamiltonian for S but which also contain information about the environment E, n is the dimension of the Hilbert space of S, $\{\gamma_j(t)\}_{j=1}^{n^2-1}$ is a set of nonnegative functions with dimensions of frequency, $\{L_j(t)\}_{j=1}^{n^2-1}$ is a set of dimensionless operators known as Lindblad operators or quantum jump operators, and the braces denote the anticonmmutator of the operators enclosed within. This equation is known as **Gorini-Kossakowski-Sudarshan-Lindblad** (GKSL) equation or simply as the **Lindblad master equation**. The first term on the right-hand of the equation represents the unitary evolution of the system, whereas the second term describes the dissipative aspect of the dynamics. In the typical case where the time evolution operator $U(t, t_0)$ is invariant under temporal translations, that is $U(t + \tau, t_0 + \tau) = U(t, t_0)$, $\forall t, t_0, \tau \in \mathbb{R}$, then $H(t)$, $L_j(t)$

and $\gamma_j(t)$ become constants, resulting in a time-independent Lindblad master equation.

The generator of the evolution, $\mathcal{L}$ is called the Lindblandian. The form of the dissipative part of the Lindblandian, is

$$\mathcal{L}_D[\cdot] = \sum_{j=1}^{n^2-1} \gamma_j \left[L_j \rho_S(0) L_j^\dagger - \frac{1}{2}(L_j^\dagger L_j, \rho_S(0)) \right], \quad \gamma_j > 0$$

We can now define the decoherence phenomena. Decoherence is what happens when $\mathcal{L}_D \neq 0$. This means that the evolution of the density matrix is governed not only by the Liouville-von Neumann component $\frac{-i}{\hbar}[H, \cdot]$ responsible for the unitary evolution, but also by the dissipator, which gives rise to non-unitary evolution.

The formal solution of the Lindblad equation $\frac{d\rho_S(t)}{dt} = \mathcal{L}\rho_S$ is

$$\rho_S(t) = e^{\mathcal{L}t}\rho_S(0) \equiv \Lambda(t)\rho_S(0)$$

For a certain t, the application $\Lambda(t)$ is called a dynamic application. If we consider the family of maps over time generated by the GKSL equation we obtain a one-parameter family $\{\Lambda(t) : t \geq 0\}$ of dynamic applications that satisfy the following properties:

1. Identity operator: $\Lambda(0) = \mathbb{1}$

2. Closed under multiplication: since we are assuming a Markovian behaviour, it follows the Markov property,

$$\Lambda(t_1)\Lambda(t_2) = \Lambda(t_1 + t_2), \quad t_1, t_2 \geq 0$$

   which is also the property of the semigroup.

3. Associative: $(\Lambda(t_1)\Lambda(t_2))\Lambda(t_3) = \Lambda(t_1)(\Lambda(t_2)\Lambda(t_3))$

Therefore, the family of one-parameter dynamic applications generated by the GKSL equation form a **quantum dynamical semigroup**. For providing the definition of this structure, first we need to introduce the following notions.

Let X be a Banach space and let $f \in X^*$. We denote $\varphi_f : X \longrightarrow \mathbb{R}$ the linear functional $\varphi_f(x) = \langle f, x \rangle$. As f runs through X^* we obtain a collection $\{\varphi\}_{f \in X^*}$ of maps from X into $\mathbb{R}$. We define a new topology on the set X:

Definition 2.24 (Weak topology)**.** The weak topology $\sigma(X, X^*)$ on X is the coarsest topology associated to the collection $\left\{\varphi_f\right\}_{f \in X^*}$

We are now going to define a topology on X^* called the **weak* topology** and denoted by $\sigma(X^*, X)$. For every $x \in X$ consider the linear functional $\varphi_x : X^* \longrightarrow \mathbb{R}$ defined by $f \longmapsto \varphi_x(f) = \langle f, x \rangle$. As x runs through X we obtain a collection $\{\varphi_x\}_{x \in X}$ of maps from X^* into $\mathbb{R}$.

Definition 2.25 (Weak* topology)**.** The weak* topology $\sigma(X^*, X)$ is the coarsest topology on X^* associated to the collection $\{\varphi_x\}_{x \in X}$

Next, we will characterise the notion of convergence in these topologies.

Definition 2.26.

- A sequence $\{x_n\}$ in X converges to x in the weak topology $\sigma(X, X^*)$,

$$x_n \rightharpoonup x$$

if and only if

$$\langle f, x_n \rangle \longrightarrow \langle f, x \rangle \ \forall f \in X^*$$

- A sequence $\{f_n\}$ in X^* converges to f in the weak* topology $\sigma(X^*, X)$,

$$f_n \overset{*}{\rightharpoonup} f$$

if and only if

$$\langle f_n, x \rangle \longrightarrow \langle f, x \rangle, \forall x \in X$$

It is worth pointing out that if $f_n \overset{*}{\rightharpoonup} f$ in $\sigma(X^*, X)$ and $x_n \rightharpoonup x$ in $\sigma(X, X^*)$, in general, one cannot conclude that $\langle f_n, x_n \rangle \longrightarrow \langle f, x \rangle$.

Now, we will briefly go through some concepts of the theory of Banach algebras, a large are in functional analysis. We will assume that the underlying field of scalars $\mathbb{K}$ is the field of complex numbers $\mathbb{C}$. An algebra over $\mathbb{C}$ is a vector space $\mathcal{A}$ over $\mathbb{C}$ that also has a multiplication defined on it that makes $\mathcal{A}$ into a ring such that if $\alpha \in \mathbb{C}$ and $a, b \in \mathcal{A}$, $\alpha(ab) = (\alpha a)b = a(\alpha b)$

Definition 2.27 (Banach algebra)**.** A Banach algebra is an algebra $\mathcal{A}$ over $\mathbb{C}$ that has a norm $|| \cdot ||$ relative to which $\mathcal{A}$ is a Banach space and such that for all $a, b \in \mathcal{A}$,

$$||ab|| \leq ||a|| \, ||b||$$

A C*-algebra is a particular type of Banach algebra that is intimately connected with the theory of operators on a Hilbert space.

If $\mathcal{A}$ is a Banach algebra, an **involution** is a map

$$\begin{aligned} \mathcal{A} &\longrightarrow \mathcal{A} \\ a &\longmapsto a^* \end{aligned}$$

such that the following properties hold for $a, b \in \mathcal{A}$, $\alpha \in \mathbb{C}$:

1. $(a^*)^* = a$
2. $(ab)^* = b^* a^*$
3. $(\alpha a + b)^* = \bar{\alpha} a^* + b^*$

Definition 2.28 (C^*-algebra)**.** A C^*-algebra is a Banach algebra $\mathcal{A}$ with an involution such that for every a in $\mathcal{A}$,

$$||a^* a|| = ||a||^2$$

Definition 2.29 (Von-Neumann algebra)**.** A Von-Neumann algebra $\mathcal{A}$ is a C^*-algebra of the bounded operators $\mathcal{B}(\mathbb{H})$ in a Hilbert space that is closed under the weak* topology and contains the identity $I \in \mathcal{B}(\mathbb{H})$

Lastly, we can define the concept of quantum dynamical semigroups.

Definition 2.30 (Dynamic semigroup)**.** Let $\mathcal{A}$ be a von-Neumann algebra. A dynamic semigroup is defined as a one-parameter family of applications $\Phi_t : \mathcal{A} \longrightarrow \mathcal{A}$ with the following properties:

1. Φ_t is positive for all $t \geq 0$
2. $\Phi_t(I) = I$
3. $\Phi_s \cdot \Phi_t = \Phi_{s+t}$
4. $\Phi_t(X) \overset{*}{\rightharpoonup} X$ when $t \longrightarrow 0$
5. Φ_t is weakly* continuous operator in $\mathcal{A}$

2.7. Closing remarks

The revolution in the world of physics in the 1920s led to the creation of quantum mechanics, an indispensable part of science ever since. Quantum computing emerges from this field, as proposed by Richard Feynman, in order to understand and simulate it.

Another major triumph of the 20th century was computer science. Within a century, there were proposals for the Turing machine, the construction of the first computers, and hardware development. In 1965, Gordon Moore declared that computational power would double at a constant cost every two years, meaning that the number of transistors in a microprocessor would double every two years. However, there is a limit to this due to quantum effects interfering with the operation of increasingly smaller electronic components. To overcome this barrier, one possible solution is to shift the computing paradigm: quantum computing.

In essence, quantum computing is the union of both branches, but after its invention and theoretical formulation: What are its possibilities? Quantum computing surpasses classical computing for some specialized tasks. The potential applications of quantum computing are still to be determined as it does not provide efficient solutions to all problems. However, this does not mean that quantum computing is an illusion or mere theory. Quantum algorithms have been developed, such as the famous Grover's Algorithm (for searching an unstructured search space) and Shor's Algorithm (for factorizing a number). The field of quantum information theory has been created, encompassing quantum computing, cryptography, and quantum communication. Furthermore, progress has been made in the theory of quantum error correction to protect data integrity.

On the other hand, companies are in a race to build quantum computers and have made many tools available to researchers and developers to program quantum algorithms and run them on their computers, such as IBM Quantum Composer and Quantum Lab, IBM's Qiskit, Google Quantum AI's Cirq, PennyLane, D-Wave's Leap, and many more.

Quantum computing will coexist with classical computing; it is not a paradigm that will replace the other, as both have their strengths. It is a burgeoning new field with challenges and questions to investigate, engaging physicists, engineers, and mathematicians in research groups and companies today.

3. Quantum Machine Learning

Quantum machine learning is the field that involves the application of machine learning techniques, with the distinctive aspect that quantum computing is integrated at some phase of the process. As seen in chapter 1 the basic ingredients of a machine learning setup consists on a computational model to tackle the problem, lots of data to "feed" the model and a algorithm of optimization of the configuration of the model so that our model learns from the training data. Therefore, quantum machine learning can vary from the use of a quantum computer in some part of the model, the use of data generated by some quantum process to the use of a quantum computer to process quantum-generated data to achieve improved insights.

In an attempt to further specify the broad world of quantum machine learning we are going to follow the categorization proposed by Schuld and Petruccione in [SP21]. There are four intersections on the possible ways to combine quantum computing and machine learning depending on the generation of the data (by a quantum system (Q) or a classical system (C)) and the information processing device (being a quantum (Q) or a classical (C) device).

- The CC intersection covers classical data being processed classically, which is the conventional approach to machine learning, but in this contexts, it concerns the classical machine learning techniques that are based on ideas from quantum computing. So there is no actual quantum computing involved in this approach, just some inspiration.
- The QC intersection considers classical machine learning algorithms that operate on quantum data. It can be understood whether as classical machine learning methods that are run on data generated by quantum processes or as the application of classical machine learning to quantum computing.
- The kind of quantum machine learning that is referred on this thesis and in the bibliography taken into consideration is the intersection CQ: classical data being processed by a quantum computer. These machine learning techniques are implemented within a quantum computer that is fed classical data. Quantum computing is involved in the model or in the training of the machine learning techniques.
- Ultimately, there is the QQ category that works on quantum data being processed by a quantum computer. As in the CQ intersection, quantum computing may be involved in the models or in the training processes, but in this case, the quantum data needs to come from measuring a quantum system (in a physical experiment or in a simulation of it run on a quantum computer) such as [PYF21] and [SACC21] proposals. QQ can be viewed as an extension of CQ and it is definitely a promising area but the required technologies and research on this intersection are still scarce.

In this thesis we will focus on CQ quantum machine learning, but within this category there is a broad range of possibilities. We will leave aside the proposals of quantum algorithms that could speed up classical machine learning because they rely on a full error-corrected

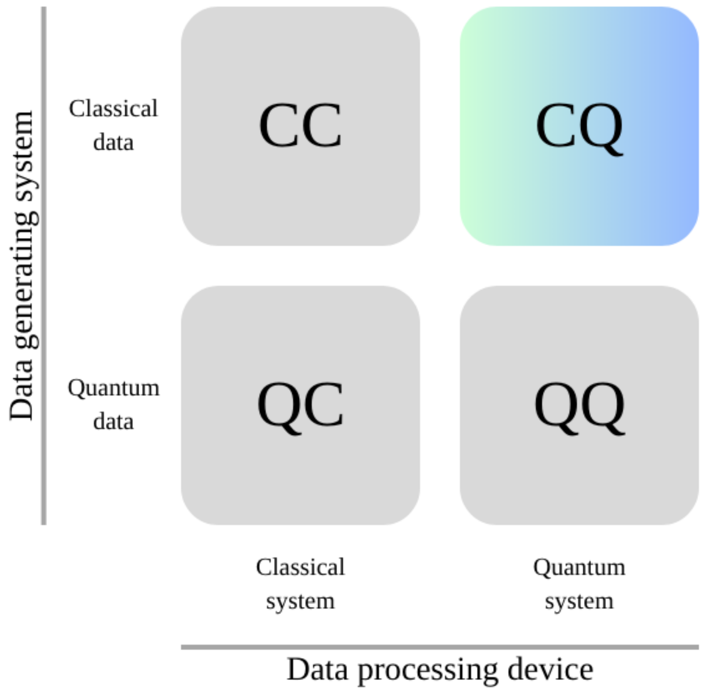


Figure 3.1.: The four approaches to combine quantum computing and machine learning, categorized according to the nature of the data and the system used

quantum computer that is currently beyond the limits of the state-of-the-art. Instead, we will dedicate to the study of fully quantum-oriented models that can be run on NISQ devices. [1] The goal is to use a quantum device as a machine learning model, not just as an alternative hardware to accelerate computation. This leads to new models and training algorithms derived from a quantum computational paradigm that require a the solid understanding of the intricacies of machine learning. Exploring the possibilities of NISQ computers and machine learning algorithms suitable to be executed on them will be a crucial field in the short and medium terms. It may bring us closer to the way for the first practical applications of quantum computing come to life.

3.1. Quantum support vector machines

Quantum support vector machines are a particular case of general support vector machines that rely on kernel methods in which we use quantum computers to map data into a space of quantum states. The kernel method, as seen in subsection 1.3.1, consists in mapping the data from its original space to a higher dimensional space known as feature space in which we wish our data could be separable by a hyperplane. In other words, by raising the dimensionality, the so called feature map can transform the problem into a linearly separable one. Quantum computing and kernel methods are based on a similar principle, a straightforward way to visualize it is presented in Figure 3.2. Both have mathematical frameworks in which information is mapped into and then processed in high-dimensional spaces to which we have only limited access. In kernel methods, the access to the feature space is facilitated though kernels (which are inner products of feature vectors). In quantum computing, access to the Hilbert space of quantum states is given by measurements, which can also be expressed through inner products of quantum states.

Therefore, we can view quantum support vector machine as ordinary support vector machines that rely on kernel methods where the feature space $\mathcal{F}$ is a certain space of quantum states. In order to train and use QSVM for classification, we will be able to operate as usual with classical SVM, except for the computation of the kernel function, which will require a quantum computer to

1. Take as input two vectors in the original space of data, $\mathcal{X}$.
2. Map each vector to a quantum state through a feature map, Φ.
3. Compute the inner product of the quantum states and return it.

Let's dive deeper into the mathematical framework that supports QSVM. Kernel theory in feature space for machine learning uses linear algebra and functional analysis as tools to pull data to a feature space for the sake of better discrimination of classes or simpler representation of data.

3.1.1. Feature map

Feature maps play an important role in machine learning, since they map any type of input data into a higher dimensional space with a well-defined metric.

Essentially feature maps are just circuits that are parametrized exclusively by the original (classical) data and thus prepare a quantum state that depends only on that data. Consider a

[1]NISQ stands for noise intermediate-scale quantum computers, that are at our disposal now.

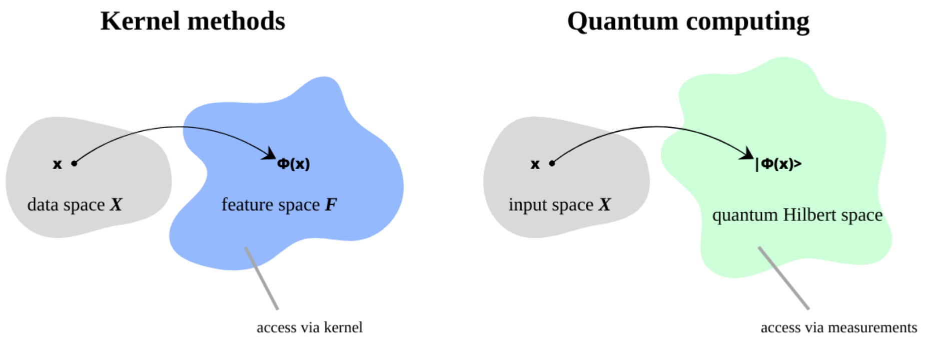


Figure 3.2.: Parallelism between the framework of kernel methods and the world of quantum computing. Illustration of own production inspired by [SP21].

data embedding circuit Φ that depends on some classical data $x \in \mathcal{X}$. For each input $x \in \mathcal{X}$, we will have a circuit $\Phi(x)$ such that the output of the feature map will be the quantum state $\varphi(x) = \Phi(x) |0\rangle$.

Definition 3.1 (Feature map)**.** Given a Hilbert space $\mathcal{F}$, called the feature space, an input set $\mathcal{X}$ and $x \in \mathcal{X}$ a sample from the input set. A feature map is a map $\Phi : \mathcal{X} \longrightarrow \mathcal{F}$ from inputs to vectors in the Hilbert space. The vectors $\Phi(x) \in \mathcal{F}$ are called feature vectors.

By definition of inner product, every feature map gives rise to a kernel.

Theorem 3.1. *Given a feature map $\Phi : \mathcal{X} \longrightarrow \mathcal{F}$. The inner product defined in the feature space of two feature vectors (obtained via the feature map) defines a kernel.*

$$\kappa(x, y) = \langle \Phi(x), \Phi(y) \rangle_{\mathcal{F}}$$

Proof. We shall see that the previously defined kernel is a semi-definite function by showing that its Gram matrix is positive definite. Let $c_i, c_j \in \mathbb{C}$ and $x_i \in \mathcal{X}$ for $i, j \subseteq \{1, ...M\}$ with $M \geq 2$

$$\sum_{i,j=1}^{M} c_i c_j \kappa(x_i, x_j) = \left\langle \sum_{i=1}^{M} c_i \Phi(x_i), \sum_{j=1}^{M} c_j \Phi(x_j) \right\rangle = || \sum_{i=1}^{M} c_i \Phi(x_i) ||^2 \geq 0$$

□

3.1.2. Reproducing kernel Hilbert spaces

Kernel theory gives rise to the reproducing kernel Hilbert space (RKHS), a rather abstract concept yet useful in order to understand the importance of kernels for machine learning, as their connection to linear models in feature space.

So far, we are dealing with two Hilbert spaces: the Hilbert space of the quantum system and the feature space $\mathcal{F}$ that contains the embedded data by the feature map. Now we will build another feature space for the quantum kernel, derived directly from the kernel. This feature space is a Hilbert space $\mathcal{R}$ of functions and due to its definition it is called the reproducing kernel Hilbert space.

Definition 3.2 (RKHS)**.** Let $\mathcal{X}$ be a non-empty input set and $\mathcal{R}$ a Hilbert space of functions $f : \mathcal{X} \longrightarrow \mathbb{K}$ that map inputs to real numbers. Let $\langle \cdot, \cdot \rangle$ be the inner product on $\mathcal{R}$.
$\mathcal{R}$ is a reproducing kernel Hilbert space if every point evaluation is a continuous functional $F : f \longrightarrow f(x) \quad \forall x \in \mathcal{X}$. This condition is equivalent to the condition that there exists a function $\kappa : \mathcal{X} \times \mathcal{X} \longrightarrow \mathbb{K}$ for which $\langle f, \kappa(x, \cdot) \rangle = f(x)$, with $\kappa(x, \cdot) \in \mathcal{R}, \forall f \in \mathcal{H}, x \in \mathcal{X}$.

The function κ is the unique reproducing kernel of $\mathcal{R}$.

To understand it better let's consider a kernel function of two variables $\kappa(x, y)$. For n points we fix one of the variables to have $\kappa(x_1, y)$, $\kappa(x_2, y)$,..., $\kappa(x_n, y)$. These are n functions of the variable y. RKHS is a function space which is the set of all possible linear combinations of these functions:

$$\mathcal{R} = \left\{ f(\cdot) = \sum_{i=1}^{n} \alpha\kappa(x_i, \cdot) \quad \forall y \in \mathcal{X} \right\}$$

As we have just seen, the functions that belong to the RKHS are linear combinations of its elementary kernel functions $\kappa(x, \cdot)$ where one variable is fixed in a possible data sample $x \in \mathcal{X}$. This kernel $\kappa(x, \cdot)$ assign a distance measure to every data point. Computing $\kappa(x, y)$ gives us the distance between the two points $x, y \in \mathcal{X}$. Therefore, the functions $f(y) = \sum_{i=1}^{n} \alpha\kappa(x_i, y) \in \mathcal{R}$ are linear combinations of data similarities.

For example, using one of the most widely used kernels, a Gaussian kernel

$$\kappa(x, y) = \exp \frac{||x - y||}{2\sigma^2}$$

the functions $f \in \mathcal{R}$ are linear combinations of Gaussians centred in each data point. The kernel regulates the "smoothness" of the functions in $\mathcal{R}$ by changing the variance of the Gaussian.

In order to calculate the inner product of two functions in RKHS $f, g \in \mathcal{R}$, since every function in RKHS can be written as a linear combination, $f = \sum_{i=1}^{n} \alpha_i\kappa(x_i, \cdot)$ and $g = \sum_{j=1}^{n} \beta_j\kappa(y_j, \cdot)$,

$$\begin{aligned}
\langle f, g \rangle &= \left\langle \sum_{i=1}^{n} \alpha_i\kappa(x_i, \cdot), \sum_{j=1}^{n} \beta_j\kappa(y_j, \cdot) \right\rangle \\
&\overset{(1)}{=} \left\langle \sum_{i=1}^{n} \alpha_i\kappa(x_i, \cdot), \sum_{j=1}^{n} \beta_j\kappa(\cdot, y_j) \right\rangle \\
&= \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i\beta_j\kappa(x_i, y_j)
\end{aligned}$$

where (1) uses that the kernel is symmetric. Indeed this is a well-defined inner product:

1. It is symmetric, because so it is κ

$$\langle g, f\rangle = \sum_{j=1}^{n}\sum_{i=1}^{n} \beta_j \alpha_i \kappa(y_j, x_i) = \langle f, g\rangle$$

2. It is bilinear

   Observe,

$$\langle f, g\rangle = \sum_{j=1}^{n} \beta_j \sum_{i=1}^{n} \alpha_i \kappa(x_i, y_j) = \sum_{j=1}^{n} \beta_j f(y_j)$$

   Then we have,

$$\begin{aligned}\langle f_1 + f_2, g\rangle &= \sum_{j=1}^{n} \beta_j (f_1(y_j) + f_2(y_j)) \\ &= \sum_{j=1}^{n} \beta_j f_1(y_j) + \sum_{j=1}^{n} \beta_j f_2(y_j) \\ &= \langle f_1, g\rangle + \langle f_2, g\rangle\end{aligned}$$

   Similarly, we can show that $\langle f, g_1 + g_2\rangle = \langle f, g_1\rangle + \langle f, g_2\rangle$

3. It is positive semi-definite, since the kernel κ is positive semi-definite. Next we will see that in fact, it is strictly positive definite. It is the last characteristic missing to prove it is an valid inner product.

The name RKHS consists on several parts, let's analyse its meaning:

- "Reproducing" because of the reproducing property of this space.

  We consider $g \in \mathcal{R}$ a kernel in RKHS space. In other words, we take g to have only one component $g(x) = \sum_{j=1}^{n} \beta_j \kappa(x_i, x) = \beta\kappa(x, x)$. We take $\beta = 1$ to have $g(x) = \kappa(x, x)$.

  We also consider $f \in \mathcal{R}$ as $f = \sum_{i=1}^{n} \alpha_i \kappa(x_i, y)$. According to Def. 3.1.2, the inner product of $f, g \in \mathcal{R}$ is:

$$\begin{aligned}\langle f(x), g(x)\rangle &= \langle f, \kappa_x(\cdot)\rangle \\ &= \left\langle \sum_{i=1}^{n} \alpha_i \kappa(x_i, x), \kappa(x, x) \right\rangle \\ &= \sum_{i=1}^{n} \alpha_i \kappa(x_i, x) = f(x)\end{aligned}$$

  This means that the function f is reproduced from the inner product of that function with one of the kernels of the space. In conclusion, the reproducing property is

$$\langle f, \kappa(x, \cdot)\rangle = f(x)$$

  with $\kappa(x, \cdot), f \in \mathcal{R}, x \in \mathcal{X}$.

  It is worth pointing out the special case

$$\langle \kappa(x, \cdot), \kappa(x, \cdot)\rangle = \kappa(x, x)$$

- "Kernel" because of the kernels associated to RKHS as priorly stated.
- "Hilbert space": because $\mathcal{R}$ is a complete prehilbertian space.
  We need to check that $\langle \cdot, \cdot \rangle$ is strictly positive definite:

$$|f(x)|^2 \overset{(1)}{=} |\langle \kappa(x,\cdot), f\rangle|^2 \overset{(2)}{\leq} \langle \kappa(x,\cdot), \kappa(x,\cdot)\rangle \cdot \langle f, f\rangle = \kappa(x,x)\langle f, f\rangle$$

  where we use
  1. The reproducing property
  2. The special case of the reproducing property item 3.1.2

  It follows that $\langle f, f\rangle = 0$ if $f = 0$ is the identically null function. Therefore $\langle \cdot, \cdot \rangle$ is strictly positive definite and we can claim that it is an inner product. Now, we consider the norm

$$||f|| = \sqrt{\langle f, f\rangle}$$

  $\mathcal{R}$ includes the limit points of sequences that converge in that norm. That makes it a complete space.

Since a feature map induces a kernel and a kernel gives rise to a reproducing kernel Hilbert space, we can construct a unique reproducing kernel Hilbert space for any given feature map.

Theorem 3.2. *Let $\Phi : \mathcal{X} \longrightarrow \mathcal{F}$ be a feature map over an input set $\mathcal{X}$, giving rise to a complex kernel $\kappa(x,y) = \langle \Phi(x), \Phi(y)\rangle_{\mathcal{F}}$. The corresponding reproducing kernel Hilbert space has the form*

$$\mathcal{R}_\kappa = \{f : \mathcal{X} \longrightarrow \mathbb{K} \,|\, f(x) = \langle w, \Phi(x)\rangle_{\mathcal{F}}, \quad \forall x \in \mathcal{X}, w \in \mathcal{F}\}$$

Definition 3.3 (Linear models). Let $\mathcal{X}$ be a data domain and $\Phi : \mathcal{X} \longrightarrow \mathcal{F}$ a feature map. A linear model in $\mathcal{F}$ is a function that maps every $x \in \mathcal{X}$ to the inner product of its feature map $\Phi(x)$ and a certain vector in the feature space $w \in \mathcal{F}$. Formally,

$$f(x) = \langle \Phi(x), w\rangle_{\mathcal{F}}$$

with $w \in \mathcal{F}$.

Theorem 3.3. *Deterministic quantum models are linear models in data-encoding feature space. Let $f(x) = \text{tr}\{\rho\mathcal{M}\}$ be a quantum model and $\Phi : \mathcal{X} \longrightarrow \mathcal{F}$ the feature map $\Phi(x) = \rho(x) \in \mathcal{F}$. The quantum model f is a linear model in $\mathcal{F}$.*

This was the missing point to understand the relevance of the reproducing kernel Hilbert space: the functions living in the RKHS are the quantum model functions, which we have just stated that are linear models. Thus, the RKHS is equivalent to the space of linear models derived from the reproducing kernel. To conclude, the kernel essentially defines the class of functions that the linear model can express and, as a result, learn.

As seen in Theorem 3.2, the functions $\langle w, \cdot\rangle$ in the RKHS $\mathcal{R}_\kappa$ associated with feature map Φ can be interpreted as linear models for which $w \in \mathcal{F}$ defines a hyperplane in feature space.

3.1.3. Quantum kernels

As anticipated in subsection 1.3.1, the relation between kernels and inner products has a great importance in machine learning since they are a means of computation in the feature space without having to deal with the feature vectors $\Phi(x)$. Therefore, the classifier can be inferred from a kernel function that encodes the scalar product between the new features.

Now, let the feature space be the Hilbert space of the quantum system. We want to find out the kernel associated to that system, which we will call the quantum kernel. The methodology of work would be to encode some input $x \in \mathcal{X}$ into a quantum state $|\Phi(x)\rangle \in \mathcal{F}$ described by a vector in the feature Hilbert space. This "input-encoding" procedure corresponds to what we have defined as **feature map** $\Phi : \mathcal{X} \longrightarrow \mathcal{F}$. There are different input encoding techniques in quantum machine learning, we discuss more on them in this section. According to Theorem 3.1 the feature map Φ induces a kernel κ:

$$\kappa(x,y) = \langle \Phi(x), \Phi(y) \rangle_{\mathcal{F}} \quad \forall x, y \in \mathcal{X}$$

As stated by Theorem 3.2 the kernel κ induces a RKHS $\mathcal{R}_\kappa$ (where κ is the reproducing kernel). The functions in $\mathcal{R}_\kappa$ are the inner products of the feature maps of the input data and a vector $|w\rangle \in \mathcal{F}$ which defines a linear model

$$f(x,w) = \langle w|\Phi(x)\rangle$$

Note that we use Dirac brackets $\langle \cdot|\cdot \rangle$ instead of the inner product $\langle \cdot, \cdot \rangle$ to denote the inner produts in a quantum Hilbert space.

It is worth pointing out that the idea of interpreting $x \longrightarrow |\Phi(x)\rangle$ is the starting point that allows us to make use of the entire framework of kernel theory.

Definition 3.4 (Quantum kernel)**.** Given Φ a feature map over an input domain $\mathcal{X}$. A quantum kernel is the Hilbert-Schmidt inner product between two feature vectors $\varphi(x), \varphi(y)$ where $x, y \in \mathcal{X}$:

$$\kappa(x,y) = \operatorname{tr}\{\varphi(y), \varphi(x)\} = |\langle \Phi(y)|\Phi(x)\rangle|^2$$

The quantum kernel previously defined is indeed a kernel because it is a positive definite function. We can see that it is a product of the complex kernel $\kappa(x,y) = \langle \Phi(y)|\Phi(x)\rangle$ and its complex conjugate $\kappa(x,y)^* = \langle \Phi(x)|\Phi(y)\rangle$. It is needed to check that the complex conjugate of a kernel is a kernel itself. Given $x_i \in \mathcal{X}, \quad i = 1, ..., n$ and $c_i \in \mathbf{C}$,

$$\begin{aligned}
\sum_{i,j=1}^{n} c_i c_j^* (\kappa(x_i, x_j))^* &= \sum_{i,j=1} c_i c_j^* \langle \Phi(x_i), \Phi(x_j)\rangle \\
&= (\sum_{i=1}^{n} c_i \langle \Phi(x_i)|) \cdot (\sum_{i=1}^{n} c_i^* |\Phi(x_i)\rangle) \\
&= || \sum_{i=1}^{n} c_i^* |\Phi(x_i)\rangle ||^2 \geq 0
\end{aligned}$$

So the complex conjugate of a kernel is also positive definite.

Lastly, the product of two kernels is known to be a kernel.

To bring to life the definition of quantum kernel, we need to dive into different examples of data encoding feature maps, its feature-embedding circuits and kernels they give rise to.

1. **Basis encoding.** The data encoding feature map of basis encoding maps a n bit binary string $x = (b_1, ...b_n)$ with $b_i \in \{0, 1\}, \quad i = 1, ..., n$ to a computational basis state in a n-qubit system, which in turn corresponds to a standard basis vector $|i_x\rangle$, with i_x being

the integer representation of the bitstring), in a 2^n-dimensional Hilbert feature space. This feature map is given by

$$\Phi : x \in \{0,1\}^n \longrightarrow |i_x\rangle$$

Basically this feature map maps each data input to a state from an orthonormal basis.

The associated quantum kernel is the Kronecker delta

$$\kappa(x,y) = \langle \Phi(x), \Phi(y) \rangle_{\mathcal{F}} = \langle i_x | i_y \rangle \overset{(1)}{=} \delta_{x,y}$$

in (1) we observe we are computing the inner product in a quantum Hilbert space (hence the Dirac notation) of orthonormal states by definition of the feature map.

This kernel is a binary similarity measure on the input space that is only nonzero for two identical inputs, thus really strict and hardly ever a good choice of data encoding for quantum machine learning tasks.

2. **Amplitude encoding.** The data encoding feature map of amplitude encoding maps each input vector into the amplitudes of a quantum state with respect to a fixed basis, the orthonormal computational basis. Let $\mathbf{x} = (x_0, ..., x_{N-1}) \in \mathbb{R}^N$ be a input vector of dimension $N = 2^n$. This encoding technique maps it to the amplitudes of a n-qubit state $|\psi_\mathbf{x}\rangle$.

$$\Phi : \mathbf{x} \in \mathbb{R}^N \longrightarrow |\psi_\mathbf{x}\rangle = \frac{1}{\sqrt{\sum_i x_i^2}} \sum_{i=0}^{N-1} x_i |i\rangle$$

where $|i\rangle$ denotes the i-th computational basis state. It is worth pointing out that we have included a normalization factor to make sure that the output is, indeed, a quantum state. This definition of amplitude encoding doesn't work for the null vector.

Amplitude encoding offers an important benefit in terms of spatial efficiency compared to basis encoding. Using basis encoding, in a n-qubit system we can only codify strings of n classical bits, whereas using amplitude encoding we can store vectors of dimension 2^n.

The associated quantum kernel is:

$$\kappa(\mathbf{x}, \mathrm{y}) = \langle \Phi(\mathbf{x}), \Phi(\mathrm{y}) \rangle = \langle \psi_\mathbf{x} | \psi_\mathrm{y} \rangle = \mathbf{x}^T \mathrm{y}$$

3. **Copies of quantum states.** Similarly done as in amplitude encoding, this data encoding technique maps each input vector $\mathbf{x} \in \mathbb{R}^N$ to d copies of an amplitude encoded quantum state.

$$\Phi : \mathbf{x} \in \mathbb{R}^N \longrightarrow |\psi_\mathbf{x}\rangle \otimes ... \otimes |\psi_\mathbf{x}\rangle$$

The associated quantum kernel is known as homogeneus polynomial kernel

$$\kappa(\mathbf{x}, \mathrm{y}) = \langle \Phi(\mathrm{x}), \Phi(\mathrm{y}) \rangle = \langle \psi_\mathbf{x} | \psi_\mathrm{y} \rangle \otimes ... \otimes \langle \psi_\mathbf{x} | \psi_\mathrm{y} \rangle = (\mathbf{x}^T \mathrm{y})^d$$

Next, we will go through one of the main achievements of classical kernel theory: the representer theorem. It states that the function f from the RKHS which minimises the cost function can always be expressed as a weighted sum of the kernel between x and the training data.

Theorem 3.4 (Representer theorem)**.** *Given $\mathcal{X}$ an input domain, $\mathcal{Y}$ an output domain. Let κ be $\kappa : \mathcal{X} \times \mathcal{Y} \longrightarrow \mathbb{R}$ a kernel, with a corresponding reproducing kernel Hilbert space $\mathcal{R}_\kappa$, $\mathcal{D}$ a training data set consisting of M data pairs $\mathcal{D} = \{(x_1, y_1), ..., (x_M, y_M) \in \mathcal{X} \times \mathcal{Y}\}$. Assume a cost function C that quantifies the quality of a model by comparing predicted outputs $f(x_i)$ with targets y_i and consider a strictly monotonic increasing regularisation function $g : [0, \infty) \longrightarrow \mathbb{R}$.*

Then any function $f_{opt} \in \mathcal{R}_\kappa$ that minimises the cost function C

$$f_{opt} \in \text{argmin}_{f \in \mathcal{R}_\kappa} \{\hat{R_L}(f) + g(||f||_{\mathcal{R}_\kappa})\}$$

admits a representation as

$$f_{opt}(x) = \sum_{i=1}^{M} \alpha_i \kappa(x, x_i)$$

where $\alpha_i \in \mathbb{R}, \forall 1 \leq i \leq M$.

Any model function $f \in \mathcal{R}_\kappa$ can be expressed,

$$f(x) = \sum_{i=1}^{\infty} \mu_i \kappa(x, x_i) \tag{$*$}$$

since f lives in the RKHS, a function space made up of linear combination of kernel functions.

Being f a model function that minimises the cost function C, by the representer theorem, can be written as

$$f_{opt}(x) = \sum_{i=1}^{M} \alpha_i \kappa(x, x_i) \tag{$**$}$$

The great step that the representer theorem allows us to take is passing from the model function in the feature space expressed by an infinite sum ($*$), to its expression formulated in terms of kernels over the finite M training inputs ($**$). Thus, even if we are trying to solve an optimization problem in an infinite dimensional space $\mathcal{R}_\kappa$ containing linear combinations of kernels centered on arbitrary x_i, then the solution lies in the span of the M kernels centered on the x_i. This constrains the difficulty of the optimisation since it is just needed to solve the convex optimisation problem (there is only one global minimum) of finding the parameters α_i, instead of explicitly optimising over an infinite dimensional RKHS. It follows from the fact that optimising over the RKHS of the quantum kernel is equivalent to optimising over the space of quantum models.

Proof. Consider the subspace spanned by $K_M = \{\kappa(x_i, \cdot)\}_{i=1}^{M}$. We will denote it as $\mathcal{Y} = \text{span}\{K_m\}$. Assume we project $f \in \mathcal{R}_\kappa$ onto $\mathcal{Y}$ with the orthogonal projection $P_\mathcal{Y} : \mathbb{H} \longrightarrow \mathcal{Y}$.

By the orthogonal projection theorem, Theorem A.4, the function f can be decomposed into $f_\parallel = P_\mathcal{Y}(f)$ and $f_\perp = P_{\mathcal{Y}^\perp}(f)$ and we have

$$||f||^2_{\mathcal{R}_\kappa} = ||f_\parallel||^2_{\mathcal{R}_\kappa} + ||f_\perp||^2_{\mathcal{R}_\kappa} \geq ||f_\parallel||^2_{\mathcal{R}_\kappa}$$

This implies that $||f||_{\mathcal{R}_\kappa}$ is minimized if $f \in \mathcal{Y}$.

Additionally, by the reproducing property of the RKHS $\mathcal{R}_\kappa$ we have for each $i = 1, ..., M$

$$\begin{aligned} f(x_i) = \langle f, \kappa(x_i, \cdot)\rangle_{\mathcal{R}_\kappa} &\overset{(1)}{=} \left\langle f_{\|}, \kappa(x_i, \cdot)\right\rangle_{\mathcal{R}_\kappa} + \langle f_\perp, \kappa(x_i, \cdot)\rangle_{\mathcal{R}_\kappa} \\ &\overset{(2)}{=} \left\langle f_{\|}, \kappa(x_i, \cdot)\right\rangle_{\mathcal{R}_\kappa} = f_{\|}(x_i) \end{aligned}$$

where we use

1. the orthogonal decomposition of f and the sesquilinear property of the inner product,
2. the orthogonal component has zero inner product with the basis of the subspace.

Therefore,

$$\hat{R_L}(f) = \frac{1}{M}\sum_{i=1}^{M} L(x_i, y_i, f(x_i)) = \frac{1}{M}\sum_{i=1}^{M} L(x_i, y_i, f_{\|}(x_i)) = \hat{R_L}(f_{\|}) \tag{3.1}$$

By virtue of Equation 3.1.3 and equation (3.1), we can say that

$$\min_{f\in\mathcal{R}_\kappa}\{\hat{R_L}(f) + g(||f||_{\mathcal{R}_\kappa})\} = \min_{f\in\mathcal{R}_\kappa}\left\{\hat{R_L}(f_{\|}) + g(||f_{\|}||_{\mathcal{R}_\kappa})\right\}$$

Which means this minimization depends only on the component of f lying in the subspace $\mathcal{Y}$, consequently we can present the function f_{opt} (solution of the optimization) to lie in the space as a linear combination of the basis vectors $\{\kappa(x_i, \cdot)\}_{i=1}^{M}$ as stated in ($**$).

$\square$

3.2. Quantum Neural Networks

The subject of this section are quantum neural networks (QNN), the family of quantum machine learning models that together with quantum support vector machines form the most popular part of QML.

Recovering the classification of quantum machine learning Figure 3.1, quantum neural networks are part of the CQ intersection. Nevertheless, unlike quantum support vector machines, which are a particular case of a classical machine learning method; quantum neural networks are not a particular case although they are inspired by the operating procedure of classical neural networks. Over the literature on this topic, there are different conceptions on what is referred as quantum neural networks. Here we will focus on quantum neural networks in the form of parameterized quantum circuits.

Analysing the workflow of classical neural networks, we can differenciate the following stages of the process:

1. **Data pre-processing.** It comprises the transformations performed on the input data (classical). The goal is to remain with more valuable information, codified in quantifiable attributes in the domain of the problem. The transformations forming this stage range from cleaning, instance selection, normalization, one-hot encoding, data transformation to feature extraction and feature selection.

2. **Data processing.** This stage in neural networks world refers to the flow of data through the layers of the neural network. The behaviour of this processing depends on some parameters, which are optimized during the training phase.

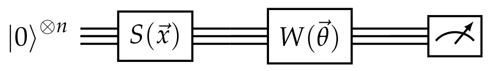


Figure 3.3.: General schema of a QNN.
A quantum neural network takes a classical input $\vec{x}$ and maps it to a quantum state $|\Phi(x)\rangle$ through a feature map Φ. This corresponds to the data pre-preprocessing stage and it is depicted in the above circuit as the block $S(\vec{x})$. The state $|\Phi(x)\rangle$ goes through a parametrised block that we have called "variational form" and it is represented by the block $W(\vec{\theta})$. It corresponds to the data processing stage. Finally, the output of the quantum neural network is the result of a measurement operation on the output of the variational form.

3. **Data output.** It consists on returning the output of the neural network, which is the output obtained through the final layer.

Following this strategy, we can define an analogous quantum neural network

1. **Data pre-processing.** As we are in CQ branch, we have classical data: numbers or arrays of numbers and we work with a quantum system, that works with quantum states. Thus, we need to embbed the classical data into the space of quantum states. We have already faced this task in subsection 3.1.1. Feature maps allow us to encode the classical input of a quantum neural network into a quantum state. Then, we can easily apply other well-known data pre-processing techniques such as normalization and scaling.

2. **Data processing.** Once working with quantum states that encode the classical input data according to a certain feature map, we have to process this "quantum input". We should discard replicating exactly the behaviour and structure of a classical neural network, given the difficulty to translate it to the current quantum hardware. Nevertheless, we can take as a valid idea from classical neural networks the application of some transformations that depend on some optimizable parameters.

   We can define this stage as the application of a circuit that depends on some optimizable parameters. This circuit receives the name of variational form or ansatz. We can understand it as a template of parametrised and fixed quantum gates, which define the architecture of the circuit, just like the layer structure defines the architecture of a classical neural network, thus it remembers us of the classical approach. The optimization of the parameters can be by the minimization of a cost function, which adapts the gates and hence, the circuit.

3. **Data output.** After the data processing stage we obtain a output quantum state, but we need to return classical output. As we are familiar with quantum computers, a measurement operation allows us to access the space of quantum states and retrieve the output. The choice of the measurement operation is up to our choice depending on the domain of the problem.

Figure 3.3 depicts the strategy just proposed for quantum neural networks as a quantum circuit. It receives the name variational circuit and for us, it will be a synonym of quantum neural network.

Since we are already familiar with feature maps, next we will be devoted to understanding variational circuits.

3.2.1. Variational circuits

Variational circuits have changed the research objective of quantum machine learning and what to pursue. Given the limitations of near-term quantum computing, instead of aiming at speedups for known models, it has risen a new model family whose usefulness was- and still is- unknown, as well as new research questions beyond the usual computational speedups questions.

Essentially, variational circuits are quantum circuits that depend on some parameters. Now, if we remember the definition from chapter 1, a machine learning deterministic model is a function $f_\theta : \mathcal{X} \longrightarrow \mathcal{Y}$ that maps input data to an output domain and depends on some parameters θ. We can intuit some analogy between variational circuits and deterministic machine learning models. In order to interpret a variational quantum circuit as a deterministic machine learning model, we apply a quantum circuit $U(x, \theta)$ to the initial state $|0\rangle$. This quantum circuit depends on both the input data x and the parameters θ. And then we consider the expectation of a measurement $\mathcal{M}$ as the output of the model.

Formally, we can define a deterministic quantum model as follows,

Definition 3.5 (Deterministic quantum model)**.** Let $\mathcal{X}$ be an input data domain, $U(x, \theta)$ a quantum circuit that depends on inputs $x \in \mathcal{X}$ and parameters $\theta \in \mathbb{R}^n$, $\mathcal{M}$ a self-adjoint operator representing a quantum observable. $|\Phi(x, \theta)\rangle$ denotes the state prepared by $U(x, \theta)$ to the initial state $|0\rangle$. The function $f_\theta : \mathcal{X} \longrightarrow \mathcal{Y}$ is defined as the expectation value of the observable $\mathcal{M}$ of the quantum system in the state $|\phi(x, \theta)\rangle$

$$f_\theta(x) = \langle \mathcal{M} \rangle_{x,\theta} = \langle \Phi(x, \theta) | \mathcal{M} | \Phi(x, \theta) \rangle$$

This function f_θ defines a deterministic variational quantum model.

The previously considered circuit $U(x, \theta)$ consists of a data embedding block $S(x)$ that corresponds to the data pre-processing stage, followed by a parametrised block $W(\theta)$ that we have previously named variational form. This interpretation of a variational circuit as a deterministic machine learning model is represented by *Figure* 3.3.

Now, let's dive deeper into how a variational form can be implemented. Theoretically, a variational form can have any internal structure, however in the context of quantum neural networks, variational forms follow a layered structure that reminds us of the spirit of classical neural networks. Next we will show this idea more accurately.

For variational form block $W(\theta)$ of N layers we would consider N parameters $\theta_1, ..., \theta_N$. Each layer i would be defined by a variational circuit W_i that depends on the parameter θ_i followed by a fixed unitary gate V_i independent of any parameters. The variational form is then formed by stacking these layers consecutively. Therefore, the variational form block is formulated as

$$W(\theta) = \prod_{i=1}^{N} W_i(\theta_i) V_i$$

Figure 3.4 represents the variational form that has been just described.

Analogously, the data embedding block $S(x)$ can also have a layered structure identical to the one proposed above considering $\{x_i\}_{i=1}^{M}$ inputs. Each layer consisting on S_i gate that depends on the i-th input followed by a fixed unitary gate T_i:

$$S(x) = \prod_{i=1}^{M} S_i(x_i) T_i$$

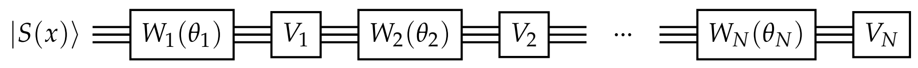


Figure 3.4.: A variational form of N layers

3.2.2. Measurements

Measurements are the final building block of every quantum neural network architecture. QNN take a classical input $x \in \mathcal{X}$ that is then fed through a feature map. The resulting quantum state is transformed by a parametrised block $W(\theta)$ that depends on some parameters θ and to conclude, a classical output is obtained through a measurement operation.

As we know, a measurement $\mathcal{M}$ is a Hermitian operator, that is, a self-adjoint operator. We are working in the quantum state space $\mathbb{H}$ which is a separable Hilbert space (view postulate 1 section 2.2. Therefore, we are in the hypothesis to apply the Spectral Theorem Theorem A.1 that grants us the existance of a orthonormal basis in $\mathbb{H}$ composed of eigenvectors of $\mathcal{M}$.

We will continue using the following notation

1. λ_k is the k-th eigenvalue of $\mathcal{M}$
2. $|\mu_k^j\rangle$ is the eigenvector associated to the k-th eigenvalue and j ranges from 1 to the geometric multiplicity of the corresponding eigenvalue.
3. $\left\{|\mu_k^j\rangle\right\}_{j,k}$ is the set of eigenvectors of $\mathcal{M}$.

The measurement operator $\mathcal{M}$ has a diagonal matrix representation in terms of the orthonormal basis of eigenvectors $\left\{|\mu_k^j\rangle\right\}_{j,k}$, with the eigenvalues in the diagonal. We can formulate this like

$$\mathcal{M} = \sum_{k,j} \lambda_k \, |\mu_k^j\rangle \langle \mu_k^j|$$

since the outer product of two eigenvectors (which are orthonormal) is:

$$|\mu_k\rangle \langle \mu_k| = \begin{bmatrix} 0 & & & & \\ & \ddots & & & \\ & & 1_{kk} & & \\ & & & \ddots & \\ & & & & 0 \end{bmatrix}$$

Given that $\left\{|\mu_k^j\rangle\right\}_{j,k}$ is a orthonormal basis of the quantum state space $\mathbb{H}$, any quantum state $|\varphi\rangle$ can be written as a linear combination of eigenvectors

$$|\varphi\rangle = \sum_{k,j} \langle \mu_k^j|\varphi\rangle \, |\mu_k^j\rangle \tag{3.2}$$

Combining all the above, we obtain that the expectation value of the measurement becomes

$$\mathcal{M} = \sum_{k,j} |\langle \mu_k^j|\varphi\rangle|^2 \lambda_k$$

which is a natural definition that agrees with the statistical expected value of the results obtained when we measure φ according to $\mathcal{M}$ since the term $|\langle\mu_k^j|\varphi\rangle|^2$ is the probability of measuring λ_k. This expression can be further simplified as follows:

$$\begin{aligned}\mathcal{M} &= \sum_{k,j} |\langle\mu_k^j|\varphi\rangle|^2\lambda_k = \sum_{k,j}\langle\mu_k^j|\varphi\rangle\,\langle\mu_k^j|\varphi\rangle\,\lambda_k \\ &\overset{(1)}{=} \sum_{k,j}\langle\mu_k^j|\varphi\rangle\,\langle\varphi|\mathcal{M}|\mu_k^j\rangle \overset{(2)}{=} \langle\varphi|\mathcal{M}\sum_{k,j}\langle\mu_k^j|\varphi\rangle\,|\mu_k^j\rangle \\ &= \langle\varphi|\mathcal{M}|\varphi\rangle\end{aligned}$$

where we have used

1. the fact that λ_k is a eigenvalue of $\mathcal{M}$: $\lambda_k\,|\mu_k^j\rangle = \mathcal{M}\,|\mu_k^j\rangle$,
2. and expressing $|\varphi\rangle$ as a linear combination of the orthonormal basis formed by eigenvectors (3.2).

Remembering our definition of quantum neural network Def. 3.5, we have just obtained the expression for the output of the model:

$$f_\theta(x) = \sum_{k,j} |\langle\mu_k^j|\Phi(x,\theta)\rangle|^2\lambda_k$$

where $|\Phi(x,\theta)\rangle$ denotes the state prepared by $U(x,\theta)$ to the initial state $|0\rangle$ and the term $|\langle\mu_k^j|\Phi(x,\theta)\rangle|^2$ is the probability of measuring λ_k. To ease the notation we will assume that the geometric multiplicity of each eigenvalue is 1 and denote the probability of measuring λ_k as $p(\lambda_k)$.

$$f_\theta(x) = \sum_k \lambda_k p(\lambda_k)$$

When we measure a single qubit in the computational basis, the coordinate matrix with respect to the computational basis of the Hermitian operator of the measurement could be

$$N = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

or

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Both of these operators represent the same observable but differ in the eigenvalues that they associate to the different possible outcomes. The first operator N associates the eigenvalues 0 to the state $|0\rangle$ and the eigenvalue 1 to the state $|1\rangle$. The second operator Z associates the eigenvalue 1 to the state $|0\rangle$ and -1 to the state $|1\rangle$. Indeed we can realise that the operator Z corresponds to the Pauli Z matrix, $Z = \sigma_Z$.

Let's consider the following example: we are working with a 2-qubit system and we take the measurement as $\mathcal{M} = \sigma_Z$, which is valid because it is self-adjoint.

The eigenvalues of σ_Z are $\lambda_+ = +1$ and $\lambda_- = -1$. The eigenvectors are μ_+ and μ_- respectively.

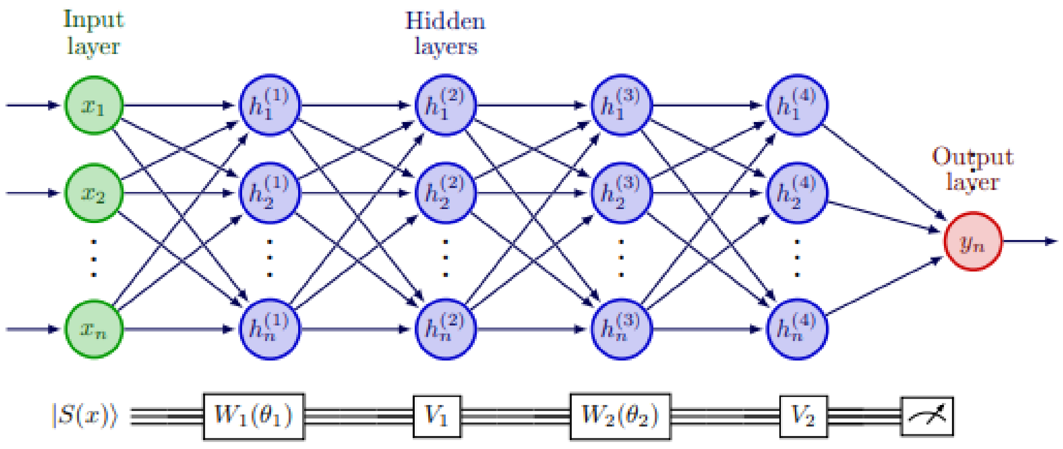


Figure 3.5.: Parallelism of a variational circuit with a classical neural network

$$\mu_+ = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle \qquad \mu_- = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle$$

When $|\Phi(x,\theta)\rangle$ denotes the state prepared by $U(x,\theta)$ to the initial state $|0\rangle$, the model reads

$$f_\theta(x) = 1\,|\,\langle 0|\Phi(x,\theta)\rangle\,|^2 + (-1)\,|\,\langle \Phi(x,\theta)\rangle\,|^2 = p(0) - p(1)$$

The expectation value $\langle \mathcal{M} \rangle$ when $\mathcal{M} = \sigma_Z$ of a single qubit measurement in the computational basis is a value in the range $[-1,1]$. In practice, the expectation is estimated by rerunning an algorithm s times to sample S bits in $\{-1,1\}$, where S is also known as the number of shots. The estimate is then computed as the average of the bits.

Let's stop a moment and recap what we have gone through this section so far. We have defined quantum neural networks as a deterministic quantum model that takes a classical input, map it to a quantum state through a feature map, that is then transformed by a variational form that depends on some θ parameters to then perform a measurement. In this field, some argue if quantum neural networks is a well-deserved name for variational circuits, because the essence of classical neural networks is their multi-layer perceptron structure. There are some ways to interpret variational circuits in which they are truly similar to neural networks. The data encoding step does not have a clear equivalent in neural networks, but we can consider each amplitude of $|\Phi(x,\theta)\rangle$ (denotes the state prepared by $S(x)$ to the initial state $|0\rangle$) as the value of an input neuron in the first layer of a neural network. Remembering the layered structure of the parametrised block $W(\theta)$, the gates used apply linear transformations on the feature vectors. Every such transformation can be viewed as a linear layer within a neural network. Some of this linear layers are fixed (V_i) and others are trainable ($W_i(\theta_i)$). Finally, the measurement corresponds to a non-linear layer with an absolute square activation that is followed by a final layer that linearly combines the absolute squares of the amplitudes with weights, which are the eigenvalues of the measurement operator.

After all, we have a model and we can construct it theoretically. However, there is a missing ingredient that we haven not given the proper attention. We have mentioned several times about the optimizable parameter θ, but up to know, we have not done anything about it. Moreover we are in the field of machine learning, so the step that we are missing is the training and that is where the optimization of θ parameter comes into the picture.

3.2.3. Training a quantum neural network

Training a quantum neural network consist on finding the parameters θ which minimise a data-dependent cost function. Consider a cost function $C(\theta)$ which relies on a model f_θ that depends on parameters θ. The partial derivative of the cost with respect to $\mu \in \theta$ is

$$\frac{\partial C}{\partial \mu} = \frac{\partial C}{\partial f_\theta} \frac{\partial f_\theta}{\partial \mu}$$

In the training of variational circuits, $\frac{\partial C}{\partial f_\theta}$ is a classical computation, so it can be tackled by classical techniques and libraries. The problem comes when computing the partial derivative of f_θ, which is the result of a quantum computation, with respect to $\mu \in \theta$ one of its variational parameters.

To unlock the potential of gradient-descent-based optimization strategies it is needed to have access to the gradients of quantum computations. By providing such partial derivatives, quantum computing can fit into hybrid machine learning pipelines without trouble and be trained.

First, we will go through a basic overview.

The model function of the quantum neural network is a scalar-valued function $f_\theta : \mathbb{R}^N \longrightarrow \mathbb{R}$. The gradient of the model is the vector of partial derivatives with respect to its parameters $\theta = \{\theta_1, ..., \theta_K\}$,

$$\nabla f_\theta = \begin{bmatrix} \frac{\partial f_\theta}{\partial \theta_1} \\ \vdots \\ \frac{\partial f_\theta}{\partial \theta_K} \end{bmatrix}$$

There are different approaches to evaluate the gradients of a numerical computation.

1. **Numerical differentiation.** We can always approximate the partial derivative numerically using the finite-differences method. Next we will apply it to a quantum model f_θ that depends on a parameter $\mu \in \theta$:

   $$\frac{\partial f_\theta}{\partial \mu} \approx \frac{f_\theta - f_{\theta+\delta\theta}}{||\delta\theta||}$$

   where $\delta\theta$ is the parameter set in which μ has been exchanged by $\mu + \delta\mu$, with $\delta\mu$ an infinitesimal shift.

   This is a linear approximation of the model between two different points. In a context where each function evaluation can only be estimated with an error, the finite-difference method is problematic. For small partial derivatives, we need more precision for each function evaluation, so we need more shots to estimate f_θ, in other words, more repetitions of the algorithm. In situations when the minimum has to be approximated closely, the optimisation landscape has many saddle points and there is a high variance of the measurements, numerical finite-differences methods are not really suitable.

2. **Automatic differentiation.** It is a programming paradigm in which the gradient is efficiently computed through the accumulation of intermediate derivatives, following the chain rule. A well known automatic differentiation technique is the backpropagation algorithm for the training of classical neural networks.

   Unfortunately, for f_θ it is not clear how this intermediate derivatives could be stored and reused inside of a quantum computation, since the intermediate quantum states cannot be measured without impacting the overall computation and backpropagated information cannot be shared because of the quantum no-cloning theorem.

3. **Parameter shift rule.** It comes from the idea of computing a partial derivative of a quantum computation using quantum computation, this means using the same circuit in the quantum neural network, yet shifting the values of the optimizable parameters. To compute gradients of quantum expectation values with respect to one of the variational parameters, the paramether shift rule follows this strategy: first we derive an equation for $\frac{\partial f_\theta}{\partial \mu}$, $\mu \in \theta$ as a linear combination of the same expectation but with the parameter "shifted". The parts of this equation can be evaluated on a quantum computer and consequently combined on a classical coprocessor. Like this, evaluating $\frac{\partial f_\theta}{\partial \mu}$ can often be done on a circuit architecture similar or identical to the one of f_θ and requires the evaluation of the model a few times, generally two, at different points in the parameter space.

   Definition 3.6 (Parameter shift rule)**.** Let $f_\mu = \langle \mathcal{M} \rangle_\mu$ be a quantum expectation value that depends on μ a classical parameter. A parameter shift rule is an identity of the form
   $$\frac{\partial f_\mu}{\partial \mu} = \sum_i a_i f_{\mu + s_i}$$
   where $a_i, s_i \in \mathbb{R}$

   A consideration is that the shifts s_i are not necessarily small infinitesimal values. With this computation, the gradient is not approximated, it is exact, however in practice, a quantum computer can only estimate an expectation value and therefore the parameter shift rule lets us compute an estimation of the analytic gradient. In contrast with the finite difference method through which we can compute an estimation of the approximate gradient.

   There would be no problem with chaining parameter shift rules to compute higher order derivatives such as the Hessian.

To conclude, on one hand we have found a valid method, the parameter shift method, to compute the partial derivatives of quantum computations, thus we can calculate $\frac{\partial f_\theta}{\partial \mu}$. On the other hand, $\frac{\partial C}{\partial f_\theta}$ is a classical computation, that can be tackled by classical techniques and libraries such as automatic differentiation. In consequence, we can compute the partial derivative of the cost with respect to $\mu \in \theta$,

$$\frac{\partial C}{\partial \mu} = \frac{\partial C}{\partial f_\theta} \frac{\partial f_\theta}{\partial \mu}$$

And we are in conditions to be able to apply gradient-descent-based optimization strategies to train quantum neural networks.

So, we have seen how to build a quantum neural network, its structure and operations and then we have figured out how to train it. Everything is proceeding smoothly up to this point. However, quantum neural networks also pose some challenges when it comes to training them that we haven't come accross yet. It is concerning the situations in which the training gradients vanish, and thus, the training can no longer progress. This situation is known as **Barren plateaus**.

A vanishing gradient means that with high probability the partial derivatives are close to zero in all their elements, Barren plateaus can be understood as areas in the landscape of the cost function in which the gradients become zero. In turn, the optimisation cannot proceed because the optimisation is slow and expensive. To resolve the small signal and avoid a random walk, high-precision measurements are needed, which are a problem when we are working with noisy near-term quantum computers.

There are different root causes for this phenomenon. They are observed when variational circuit architectures are highly expressive and Hilbert spaces large. To have an idea of what this means we can imagine we are moving over long distances in a very large space, the impact of an individual step towards the ultimate destination does not have a huge effect. Therefore, this derives into an important design principle of variational quantum models: to find a suitable, limited structure for the variational form which restricts training to the relevant subspace of the enormous Hilbert space.

The presence of barren plateaus has a stochastic nature, the high probability of the partial derivatives being zero derives from the statistical properties of partial derivatives.

3.2.4. Considerations on QNN's

As a conclusion for this section, we will reflect on a series of ideas and concerns to take into consideration regarding quantum neural networks.

1. A quantum neural network depends on a feature map, a variational form and a measurement operation. These are three choices we have to make when designing a variational circuit depending on the problem and the data we are dealing with.

2. The data is a really important part of any machine learning method. It is what feeds the quantum neural network. We should consider pre-processing techniques according to the nature of the data and the chosen feature map.

3. The design of the variational form and the number of optimizable parameters it works with is a determining factor for the power of the quantum neural network. If the variational form uses too many parameters, we risk overfitting, while with very few parameters, we would be threatened by underfitting.

4. Case study: entanglement detection

To continue further with our study of quantum machine learning methods, we will approach a binary classification problem using quantum support vector machines and quantum neural networks. First, let's get introduced to the problem.

4.1. Problem

Our goal for this chapter involved delving into the state-of-the-art quantum computing frameworks to translate the methods from chapter 3 from theory to practice. To do so, our chosen approach has been to tackle a machine learning problem. In order to continue with the discourse of this thesis, we picked up Feynman's initial proposal of simulating physics with computers and we have chosen the **quantum separability problem**. It addresses detection and classification of quantum entanglement, a fundamental feature of quantum mechanics presented in section 2.3. This problem is based on determining if a certain quantum state is entangled or not. There has been proposals of different criteria for dealing with the separability problem such as Bell's inequalities, the Peres-Horodecki positive partial transpose criterion (PPT) and entanglement witnesses ([MHH96, GT09]).

In particular, we will only focus on bipartite quantum systems, that is, systems composed of two distinct subsystems. The Hilbert space $\mathbb{H}$ associated with a bipartite quantum system is given by the tensor product $\mathbb{H}_A \otimes \mathbb{H}_B$ of the subsystems A and B. Those subsystems may not necessarily be the 2-dimensional Hilbert state space of qubits, but any n-dimensional Hilbert space. Our problem will be constrained to $\mathbb{H} = \mathbb{H}_A \otimes \mathbb{H}_B$, where $\dim(\mathbb{H}_A) = \dim(\mathbb{H}_B) = 3$. A quantum state is described by a density matrix: a positive semidefinite Hermitian matrix of trace one. Let's quickly remember the notion of entanglement: a state (that is, a density matrix) that cannot be written as a tensor product of quantum states of the subsystems. Now we can express the quantum separability problem in the terms we will work with. This problem consists in deciding whether a bipartite density matrix is entangled or separable. Formally, it can be defined as follows:

Quantum separability problem

Let $\mathbb{H} = \mathbb{H}_A \otimes \mathbb{H}_B$ be a bipartite quantum state space and ρ be a density matrix acting on $\mathbb{H}$. The quantum separability problem is to determine if $\rho \in S_+^p = \{\rho \in \mathbb{C}^{p\times p} : \rho \geq 0, \operatorname{Tr}(\rho) = 1\}$ admits a decomposition of the form

$$\rho = \sum_j q_j \sigma_{A,j} \otimes \sigma_{B,j} \text{ such that } q_j \in \mathbb{R}^+, \sum_j q_j = 1$$

where $\sigma_{A,j} \in S_+^{p_A}$, $\sigma_{B,j} \in S_+^{p_B}$ are density matrices acting on $\mathbb{H}_A$ and $\mathbb{H}_B$, respectively, and the variables $\{q_j\}$ form a probability distribution.
If ρ admits such a decomposition, then it is said to be separable, otherwise it is said to be entangled.

In the case of pure states, the Schmidt decomposition Theorem A.5 is a valid tool for detecting entanglement. However, for mixed states, the separability problem has been proved to be NP-hard [Gur03].

The starting point has been the papers [CDMAK23, USBM23], in which the separability problem is taken as a binary classification task that is approached with classical support vector machines and deep neural networks, respectively. To be able to do so, [CDMAK23] provide an efficient Frank-Wolfe based algorithm to approximately seek the nearest separable bipartite density matrix, derive a systematic way for labeling density matrices as separable or entangled and generate a labeled dataset with thousands of separable and entangled density matrices of sizes 9×9 (for the case $\mathbb{H} = \mathbb{H}_A \otimes \mathbb{H}_B$, where $\dim(\mathbb{H}_A) = \dim(\mathbb{H}_B) = 3$) and 49×49 ($\mathbb{H} = \mathbb{H}_A \otimes \mathbb{H}_B$, where $\dim(\mathbb{H}_A) = \dim(\mathbb{H}_B) = 7$). Additionally, the code and data of this work had been released at their gitlab repository, which has allowed us to take a subset of their data and manipulate it to obtain the dataset we will work with. In this study, we will take a step towards the "quantum" direction: we will apply the quantum machine learning methods that we have introduced in this thesis to the quantum separability problem.

4.2. Dataset

The dataset we have worked with in this case study comes from [CDMAK23] 6000 samples dataset of separable and entangled density matrices, released at their gitlab repository. The effort that stands out in their research is the creation of a well-labeled training dataset, specially for high-dimensional density matrices and the release of this data encouraging reproducibility and future work aimed at detecting quantum entanglement. Their approach is based on the positive partial transpose (PPT) criterion and optimal entanglement witness. Next, we will present these criteria for bipartite entanglement.

4.2.1. The PPT criterion

First, we will start with the partial transposition. If we have a state ρ that acts on $\mathbb{H}_A \otimes \mathbb{H}_B$, we can write ρ density matrix in a chosen product basis as

$$\rho = \sum_{i,j}^{p_A} \sum_{k,l}^{p_B} \rho_{ij,kl} \left|i\right\rangle \left\langle j\right| \otimes \left|k\right\rangle \left\langle l\right|$$

Given this decomposition, the partial transposition of ρ with respect to one subsystem (in this case B) is defined as

$$\begin{aligned} \rho^{T_B} &= \sum_{i,j}^{p_A} \sum_{k,l}^{p_B} \rho_{ij,kl} \left|i\right\rangle \left\langle j\right| \otimes (\left|k\right\rangle \left\langle l\right|)^T \\ &= \sum_{i,j}^{p_A} \sum_{k,l}^{p_B} \rho_{ik,kl} \left|i\right\rangle \left\langle j\right| \otimes \left|l\right\rangle \left\langle k\right| \\ &= \sum_{i,j}^{p_A} \sum_{k,l}^{p_B} \rho_{ik,lk} \left|i\right\rangle \left\langle j\right| \otimes \left|k\right\rangle \left\langle l\right| \end{aligned}$$

This definition becomes more apparent when representing the density matrix in the form of a block matrix:

$$\rho = \begin{bmatrix} \rho_{1,1} & \cdots & \rho_{1,p_A} \\ \vdots & & \vdots \\ \rho_{p_A,1} & \cdots & \rho_{p_A,p_A} \end{bmatrix}$$

where the blocks $\rho_{i,j}$ are $p_B \times p_B$ matrices $\forall i, j = 1, ..., p_A$.

Then, the partial transpose is

$$\rho^{T_B} = \begin{bmatrix} \rho_{1,1}^T & \cdots & \rho_{1,p_A}^T \\ \vdots & & \vdots \\ \rho_{p_A,1}^T & \cdots & \rho_{p_A,p_A}^T \end{bmatrix}$$

Similarly, we can define ρ^{T_A} by exchanging i and j instead of k and l.

A density matrix ρ has a positive partial transpose (or the matrix is PPT) if its partial transposition has no negative eigenvalues, thus it is positive semidefinite: $\rho^{T_B} \geq 0$. If a matrix is not PPT, we call it NPPT.

Theorem 4.1 (PPT Criterion)**.** *Let ρ be a bipartite separable state. Then ρ is PPT.*

Proof. The PPT criterion follows directly from the definition of separability. If ρ is a bipartite separable state, then

$$\rho = \sum_j q_j \sigma_{A,j} \otimes \sigma_{B,j}$$

and we have $\rho^{T_B} = \sum_j q_j \sigma_{A,j} \otimes (\sigma_{B,j})^T$

Since the transposition operation preserves eigenvalues, the spectrum of $(\sigma_{B,j})^T$ is the same as the spectrum of $\sigma_{B,j}$, therefore $(\sigma_{B,j})^T$ is positive semidefinite. Consequently, the partial transposition ρ^{T_B} is also positive semidefinite. □

For a given density matrix, this criterion guarantees us that if we can calculate its partial transpose and find a negative eigenvalue, then the state is entangled. The PPT criterion is a necessary condition, so it is natural to ask: is the PPT criterion sufficient for separability? That is, $\rho^{T_B} \geq 0$ implies separability?

Theorem 4.2 (Horodecki Theorem)**.** *If ρ is a bipartite density matrix such that $p_A = 2$ and $p_B = 2$ or $p_B = 3$, then $\rho^{T_B} \geq 0$ implies that ρ is separable. In other dimensions this is not the case.*

In 2×2 and 2×3 systems the PPT criterion is a necessary and sufficient condition for separability. When working with the separability problem in these dimensions from a machine learning approach, the PPT criterion is enough to label as separable or entangled the training density matrix set. However, in other dimensions there might be non-separable density matrices fulfilling the PPT condition. Note that PPT states contain separable and entangled states, while NPPT states are all entangled. Therefore, using the PPT criterion to label the dataset in these cases leads to a partial problem: being PPT or not, which is a different task from quantum separability and not really interesting.

According to the PPT criterion, we can classify quantum states in the following classes:

- **Separable**: density matrices that are separable fulfill the PPT criterion. In 2×2 and 2×3 all matrices that satisfy the PPT condition are separable.

- **PPT entangled**: density matrices that are entangled and fulfill the PPT condition. This class exists in systems of dimension different than 2×2 and 2×3 .
- **NPPT entangled**: density matrices which do not fulfill the PPT condition, that is, matrices whose partial transpose matrix has a negative eigenvalue and thus, are entangled. In 2×2 and 2×3 all the entangled states are matrices that do not satisfy the PPT condition.

The dataset includes separable, PPT entangled and NPPT entangled density matrices, each class stored in a file. The generation of PPT entangled density matrices is the great contribution of [CDMAK23], providing a practical and efficient method that relies on the notion of entanglement witness.

4.2.2. Methodology to obtain the dataset

Working with the code and data released by [CDMAK23], we took the following steps to create the dataset we will work with:

1. The density matrices for separable and entangled states were stored in files. From them we read:
   - 100 separable density matrices, labeled 0, and generated an array of separable density matrices
   - 100 entangled density matrices, labeled 1. For one dataset, we fixed a 0.5 PPT ratio, which means that we took 50% of the entangled class to be PPT entangled density matrices and the remaining 50% NPPT entangled states. For the other dataset, we took a PPT ratio of 1.0, that is, all the entangled matrices were entangled and fulfilling the PPT condition. There is no presence of NPPT matrices in this dataset.
2. Both arrays were stacked to create a 200 samples training dataset of bipartite 9×9 complex density matrices of a 3×3 system.
3. We transformed the array of complex density matrices (with shape $(200, 9, 9)$ into a real-valued vector of 200 samples and 80 attributes. This operation is based on the decomposition into the generalised Gell-Mann basis and provided in the repository of their paper.
4. Finally, we exported the array containing our training set into a *.csv* file. The training dataset with PPT ratio 0.5 is stored in *x_train_05.csv*. The dataset with PPT ratio 1.0 corresponds with the file *x_train_1.csv*.

The process for creating the test set was analogous: we read the density matrices for separable and entangled states from the test files. In this case we took 100 samples of each type: 100 samples of separable states, 100 PPT entangled density matrices and 100 NPPT entangled states. Resulting in a 300 samples test set.

Our aim was to reproduce the conditions under which the paper [CDMAK23] works to be able to obtain comparable results. We took the reduced version of the training set composed by 200 samples that they consider in the Appendix E.3. Their study is performed considering different PPT ratios, while we have constrained our research under the specific PPT ratios of 0.5 and 1.0. On the other hand, the test set used in the paper was composed of 3000 samples, 1000 samples per type (SEP, PPT-ent, NPPT-ent). Due to the long executing times of current quantum machine learning methods, we decided to work with a reduced version of the test set made of 300 samples, 100 samples per type. As a final consideration, we have not used data augmentation.

4.3. Implementation

We approached the implementation part of this work using a Jupyter notebook, powered by Python, and hosted on Google Colab. This choice of tools not only facilitated the smooth implementation of the project but also offered the advantage of accessibility, collaboration and the use of GPU acceleration. This notebook and data are available at the GitHub repository of this thesis.

4.3.1. PennyLane

It comes the moment to put into practice the quantum machine learning methods we acquainted in the previous chapter. Thankfully, for passing from theory to implementation there is currently a multitude of excellent software frameworks for quantum computing. Our chosen one is **PennyLane** [BIS+18], a powerful, widely used open-source software library specially targeted for quantum machine learning, built around the idea of training quantum circuits using automatic differentiation. It is being developed at Xanadu. It relies on Python as a host language. PennyLane software package allows you to implement quantum circuits, comes with several built-in simulators and allows you to train quantum machine learning models.

It is similar to **TensorFlow** and **PyTorch** for classical computation and indeed, PennyLane is highly inter-operable with classical machine learning frameworks such as **scikit-learn**, **keras** and the previously mentioned ones. This integration enables the combination of classical and quantum computations.

PennyLane comes with built-in support for several quantum devices and also a wide collection of plugins to run PennyLane quantum algorithms in external quantum devices: real quantum computers and a broad collection of simulators provided by various computing platforms. This allows researchers and developers to experiment and run quantum computations on a variety of backends.

In the end, the various quantum computing frameworks that are at our disposal, PennyLane among them, play a vital role in democratizing access to this emerging technology beyond its theoretical formulation and explore its potential.

4.3.2. Design choices

Our work in the realm of quantum machine learning starts with choosing an encoding technique. Our dataset contains classical data about the quantum separability problem, but nonetheless, classical data. The first step to work with quantum machine learning (CQ category) is to map the classical data into quantum states. There are different possible feature-embedding circuits built-in PennyLane. The amount of attributes of our dataset also conditions this choice. Working with 80 attributes makes us discard almost every PennyLane data embedding because they require the number of qubits of the system to be greater or equal to the number of attributes. Given that we have 80 attributes, or even if we reduced the number of attributes to half with dimensionality reduction techniques, the needed amount of qubits to encode the data is too high for the simulators, quantum computers and executing times we are working with in the current days. The only option is amplitude encoding, which allows us to encode $N \leq 2^n$ attributes in a system of n-qubits. This way, we would need a 7-qubit system, which can encode up to 128 attributes with amplitude encoding.

As soon as we started working with quantum machine learning models, we faced long training times. Consequently, it became apparent that we needed to reduce the dimensionality of the dataset. The dataset is formed by 200 samples characterized by 80 attributes. The challenge we had to deal with were the high number of attributes. We opted for Principal Component Analysis (PCA). This tecnique simplifies the complexity in high-dimensional data while retaining trends and patterns. It does this by transforming the data into fewer dimensions, which act as summaries of features. After a first phase of exploration of the techniques and their parameters, we decided to use PCA to 32 components for the training and test set. This way, we could reduce the number of qubits of the system to 5 and embedded the 32 features using amplitude encoding.

In practice, the roots of quantum machine learning models are quantum circuits, which are then optimised through training. To run a circuit in PennyLane you firstly need a *Device* object and a function that specifies the circuit. A *Device* object is PennyLane's virtual analog of a quantum device, provided with methods to run any given circuit. We have worked with **lightning.qubit**, a fast state-vector qubit simulator written with a C++ backend, since it is a good choice for optimizations with a moderate number of qubits and parameters. The number of qubits is a property of the device object itself that we fix when we initialize the device.

It is worth noting that we have worked with five fold cross validation. This approach is extensively used and it was also chosen in [CDMAK23] work. Data is divided in 5 disjoint subsets called folds. The goal is to validate the model with different combinations of such folds. In particular, the samples belonging to one of the five folds will become the validation set, while the union of the samples belonging to the remaining 4 folds will be used as the training set. This process is iterative such that each fold becomes once the validation set.

4.3.3. Development of the work

The development of this practical part has evolved in several consecutive phases driven by the findings and insights we gleaned.

4.3.3.1. Phase 1: Pre-work to get started

We first started working with **dataset 1.0**. The scope of this first phase was familiarizing with the framework PennyLane, the problem and the dataset, as well as building our first quantum machine learning methods and getting a broad idea of the effect of their parameters and performance. This phase determined the design choices stated previously. This section comprises the experimentation performed in section B.2.

Firstly, we proceeded with quantum support vector machines. Since quantum support vector machines are completely determined by the feature map and our feature embedding technique was limited to amplitude encoding due to the previous argumentation, there weren't many possible configurations beyond experimenting with the number of attributes of the dataset and the number qubits of the system. Our first trial with all the 80 attributes and amplitude encoding with 7 qubits, was not successful because of the training would take more than 1 hour and the kernel disconnected before completing the training and predictions. This

Nº of attributes	Nº of qubits	Device	Method	Runtime env.	Times	Accuracy test set
64	6	lightning.qubit	SVM	CPU	Training: 28 min Prediction: 44 min	1.0
32	5	lightning.qubit	SVM	CPU	Training: 15 min Prediction: 23 min	0.9667

Table 4.1.: Phase 1: Summary table of QSVM

showed us that it was mandatory to reduce the dimensionality of the dataset with Principal Component Analysis to be able to work with it. The results of this part are summarised in Table 4.1.

We noticed that the reduction in the number of attributes (from 64 to 32) implied a reduction by half in the executing time of both training and prediction stages, barely compromising the accuracy on the test set. Our first encounter with quantum support vector machines turned out to be really successful.

Next we continued with Quantum Neural Networks. We have worked with the whole dataset (with 80 attributes) and the dataset reduced with principal component analysis to 64 and 32 components for performing our experimentation. We also tested in some scenarios the effect of feeding the models with normalized data. The choice of the data encoding technique, the variational form and the number of repetitions of it (also understood as layers) are the determining factors of a quantum neural network architecture. Therefore, we have more conditions to play with. One of the most rigorous ways to select the best model and hyperparameters of a machine learning problem is using *GridSearchCV* class from scikit-learn. However, we cannot use it for tuning quantum neural network's parameters because they are part of the definition of the circuits. We have had to try by ourselves to find the most promising configuration by trial and error, observing the effects of different architectures in the performance. Although we could not perform "by hand" the exhaustive analysis that a method like *GridSearchCV* would do, we arrived to some potential results, that we will try further in the next phases, and gained a valuable insight understanding quantum neural network's architecture. The results of this part are summarised in Table 4.2.

We will point out the following aspects:

- The interface between PennyLane and automatic differentiation libraries relies on PennyLane's ability to compute or estimate gradients of quantum circuits. When creating a QNode, you can specify the differentiation method. When not specified, PennyLane will attempt to determine the best differentiation method given the device and interface. We have realised that using the device **lightning.qubit** and **adjoint** differentiation method we obtain faster training times and the accuracy does not change.
- We can appreciate that the increase in the number of repetitions seems to produce an increase in the accuracy. In particular, the last two entries of the table obtain around a 70% of accuracy.

4.3.3.2. Phase 2: Separability problem with dataset 0.5 PPT ratio

We have chosen 0.5 PPT ratio so that both types of entangled states are present in our dataset. Since the test dataset contains separable, PPT entangled and NPPT entangled matrices, we

Nº attr.	Nº of qubits	Device	Method	Nº reps	Diff. method	Runtime env.	Time	Accuracy test set
32	5	default.qubit	QNN TwoLocal	1	best	CPU	15 min	0.5967
32	5	default.qubit	QNN Strong-Entangling-Layers	2	best	CPU	25 min	0.5733
32	5	lightning.qubit	QNN Strong-Entangling-Layers	2	adjoint	CPU	11 min	0.5733
64	6	lightning.qubit	QNN Strong-Entangling-Layers	2	adjoint	CPU	13 min	0.63
80	7	lightning.qubit	QNN Strong-Entangling-Layers	2	adjoint	CPU	20 min	0.4833
80, n	7	lightning.qubit	QNN Strong-Entangling-Layers	2	adjoint	CPU	15 min	0.3333
80, n	7	lightning.qubit	QNN Strong-Entangling-Layers	4	adjoint	CPU	20 min	0.3867
32, n	5	lightning.qubit	QNN TwoLocal	1	adjoint	CPU	10 min	0.4733
32, n	5	lightning.qubit	QNN TwoLocal	5	adjoint	CPU	12 min	0.5633
32, n	5	lightning.qubit	QNN TwoLocal	10	adjoint	CPU	20 min	0.7033 SEP: 0.95 PPT: 0.57 NPPT: 0.59
32, n	5	lightning.qubit	QNN Strong-Entangling-Layers	8	adjoint	CPU	28 min	0.7233 SEP: 0.88 PPT: 0.69 NPPT: 0.6

Table 4.2.: Phase 1: Summary table of QNN
The number of attributes followed by the letter n stands for normalized.

decided that using a training dataset with 0.5 PPT ratio would be more representative of the problem and thus improve our results from phase 1. This section corresponds to section B.3 experimentation.

Our aim for this phase has been

- Study the most promising configurations from Phase 1:
    1. Quantum support vector machine, with 5 qubits amplitude encoding.
    2. Quantum neural network TwoLocal variational form with 10 repetitions.
    3. Quantum neural network StrongEntanglingLayers variational form with 8 repetitions
- Study the effect and improvements in the performance of the models using a training dataset with 0.5 PPT ratio for the quantum separability problem.
- Try out the effect of data normalization in the performance of our models.

Regarding quantum support vector machine, we have to remark that we have obtained again great results with this method. We have used a Support Vector Machine with a quantum kernel induced by Amplitude encoding with 5 qubits and 32 attributes.

We have run with GPU acceleration the training in 12 minutes and predictions on the training set in 13 minutes. We have obtained an accuracy of 1.0 on the training set. Then, we have performed the predictions on the test set, which took 19 minutes. We have obtained an accuracy of 0.9667 on the test set.

With the predictions of the test set, we have also studied the accuracy on the predictions of each type of data: Separable, PPT-entangled and NPPT-entangled:

- Accuracy of 0.97 on the separable data samples
- Accuracy of 0.95 on the PPT entangled data samples
- Accuracy of 0.98 on the NPPT entangled data samples.

Therefore, we can conclude that this QSVM performs great and is able to correctly clasify separable and entangled data, in particular PPT entangled data (which is the "hardest" to distinguish from separable) with a great accuracy. Additionally, we can state that quantum support vector machines are a robust method against the PPT ratio for the quantum separability problem.

On the other hand, as we can appreciate in the summary table Table 4.3 of quantum neural networks, we have obtained lower accuracy over the test set in our different trials, not higher than 65%, compared to our best neural networks from Phase 1 that were around 70% of accuracy. If we pay attention to the accuracy analysis broken down per type, we can observe that our models struggle to detect entanglement, specially for PPT states. The accuracy for PPT entangled states does not even reach 50%. Going back to the accuracy per type of the two last experiments from Phase 1, these models (trained with a PPT ratio 1.0 dataset) could predict entanglement better, in particular detecting entangled states of the class PPT, with up to 69% of accuracy.

We should remark that the most interesting and challenging matter of this problem is distinguishing separable states from PPT entangled states, since we can simply make use of the PPT criterion for predicting the class NPPT entangled. Therefore, after the worsening of the results of this phase, we realised that training our models with a dataset of 1.0 (thus,

Norm	Variational form	Nº reps	Time	Accuracy test set	Accuracy per type	F-1 test set
No	TwoLocal	10	17 min	0.633	SEP: 0.97 PPT 0.45 NPPT 0.48	0.628
Yes	TwoLocal	10	15 min	0.633	SEP: 0.86 PPT 0.49 NPPT 0.55	0.654
No	TwoLocal	20	24 min	0.64	SEP: 0.98 PPT 0.44 NPPT 0.5	0.635
No	StrongEntangling-Layers	8	26 min	0.64	SEP: 0.96 PPT 0.46 NPPT 0.5	0.64
Yes	StrongEntangling-Layers	8	26 min	0.56	SEP: 0.91 PPT 0.42 NPPT 0.35	0.538
No	BasicEntangler-Layers	10	20 min	0.52	SEP: 0.67 PPT 0.48 NPPT 0.43	0.56

Table 4.3.: Phase 2: Summary table of QNN

trained only with separable and PPT entangled samples) made our QNN better at detecting entanglement both for PPT entangled states (since it is trained with those samples) and for NPPT states (even if they are not present in the dataset). Our intuition of training with a dataset more representative of the test set was proved wrong. Instead, focusing on the most difficult task: separable versus PPT entangled, providing a dataset only formed by those two types of data, leads to better results.

Although in this phase we obtained worst results than we were expecting, it was a really interesting phase because it made us gain perspective on the problem and arrive to deeper a level of understanding of what we were facing beyond a simple classification problem. We acquired really valuable insights of the domain of the problem out of the initial counterintuitive worsening of the results that we did not expect. Therefore, we are leaving aside the 0.5 PPT ratio dataset to continue working with 1.0 PPT ratio dataset in the next phase.

4.3.3.3. Phase 3: Separability problem with dataset 1.0 PPT ratio

In this phase, we come back to the dataset with 1.0 PPT ratio to study the most promising configurations to approach the quantum separability problem. Our experimentation (section B.4) has been guided by the work done in Phase 2 and the goals we had set.

On the one hand, quantum support vector machine have obtained again great results with the same configuration: a quantum kernel induced by Amplitude encoding with 5 qubits and 32 attributes.

We have run with GPU acceleration the training in 12 minutes. Then, we have performed the predictions on the test set, which took 19 minutes. We have obtained an accuracy of

0.9667 on the test set.

The accuracy broken down per type of data (separable, PPT-entangled and NPPT-entangled) also shows that our quantum support vector machine can detect entanglement for both PPT and NPPT entangled states with great accuracy.

- Accuracy of 0.99 on the separable data samples
- Accuracy of 0.93 on the PPT entangled data samples
- Accuracy of 0.98 on the NPPT entangled data samples.

We conclude here our testing of QSVM, satisfied with the results obtained and the robustness of this method for detecting entanglement against the PPT ratio of the dataset.

Concerning quantum neural networks, our results are summarised in Table 4.4. We have achieved better performance overall, with better accuracy as the numper of repetitions increases as before, but what interests us is drawing a conclusion on the normalization question. In the previous phase, we left that matter unresolved because there was no clear tendency and we should have continued with more testing. We decided to leave that question for this phase, where we did see a clear effect of normalization of our data: both methods work better with normalized data, resulting in finally surpassing the 0.7 accuracy bound that it seemed to be. The quantum neural network with 10 repetitions of TwoLocal variational form has 0.703 of accuracy and with 8 repetitions of StrongEntanglingLayers variational form we have obtained an accuracy of 0.723. Finally, the best model we have tried has been the quantum neural network with 20 repetitions of TwoLocal variational form, which achieved up to 80.6% of accuracy over the test set and the greatest accuracy so far at detecting entanglement of PPT entangled states: 76%.

4.4. Results

To finalize the practical part of this thesis, we will go through the results of our approach to the quantum separability problem, where we have showcased the potential of quantum machine learning algorithms to effectively address entanglement detection.

Our findings indicate that the most difficult part of the quantum separability problem is detecting entanglement, not separability. Throughout our models we have always obtained a decent accuracy in detecting separable states, while entanglement detection has presented itself as the challenging task. The presence of PPT entangled samples in the dataset conditioned the performance of the models. A dataset formed by separable and PPT entangled samples is a better training set for quantum neural networks to face the problem, since the most interesting task is classifying separable and PPT entangled states, given that the PPT criteria is applicable for NPPT entangled states.

The challenge of our dataset has not been the number of data samples (since we kept it small to a couple of hundreds of data samples), but the number of attributes. When working with quantum support vector machines and quantum neural networks, the first step is encoding classical data into quantum states that our circuit will work with. Out of the possible data embedding techniques our choice was limited to one: amplitude encoding. This method encodes $N \leq 2^n$ attributes in a n-qubit system and allowed us to try out our methods in systems with a constrained number of qubits and execution time. We found that working with principal component analysis to reduce the number of attributes to 32 to work

Norm	Variational form	Nº reps	Time	Accuracy test set	Accuracy per type	F-1 test set
No	TwoLocal	10	14 min	0.67	SEP: 0.95 PPT 0.52 NPPT 0.54	0.68
Yes	TwoLocal	10	15 min	0.703	SEP: 0.95 PPT 0.57 NPPT 0.59	0.722
No	TwoLocal	20	24 min	0.636	SEP: 0.99 PPT 0.44 NPPT 0.49	0.628
Yes	TwoLocal	20	24 min	0.806	SEP: 0.97 PPT 0.76 NPPT 0.69	0.83
No	StrongEntangling-Layers	8	28 min	0.656	SEP: 0.97 PPT 0.5 NPPT 0.5	0.66
Yes	StrongEntangling-Layers	8	24 min	0.723	SEP: 0.88 PPT 0.69 NPPT 0.6	0.756

Table 4.4.: Phase 3: Summary table of QNN

with a 5-qubit system was a good compromise of results and execution time. However, this can be further explored as larger systems and more resources come available.

We can also state that the state-of-the-art technologies for the development of quantum machine learning methods are available and accessible. In particular, PennyLane's interoperability with Tensorflow and Pytorch opens future lines of this work exploring hybrid quantum neural networks: models in which classical and quantum layers are integrated or interleaved in some manner. This integration could involve a sequence of classical layers, which is then passed through quantum layers whose output might then be processed further by classical layers.

Quantum support vector machines are robust and our results using them are remarkable, achieving up to 96.7% of accuracy. Since we replicated the hypothesis of [CDMAK23] paper, a training dataset of 200 samples with 1.0 PPT ratio and a reduced test dataset of 300 samples with 100 samples per type (separable, PPT-entangled and NPPT-entangled), we can establish a comparison between their results with classical support vector machines (without data augmentation) and ours with a quantum support vector machine. In this case, we have achieved better broken down accuracy for separable, PPT and NPPT entangled states.

Quantum neural networks had more aspects to investigate with and were more susceptible to the PPT ratio of the dataset, number of attributes, number of repetitions of the variational form while dealing with longer training times. The best configuration provided us over 80.6% of accuracy, which we considered a satisfactory result to conclude our experimentation with quantum neural networks. However, this work paves the way for further exhaustive exploration of quantum neural network's application for the quantum separability problem.

5. Feasibility of Quantum Machine Learning problems

The target question, specifically on the economic and business point of view, in the field of quantum machine learning is: **What difference can quantum computers actually make for machine learning?**

The potential power of quantum machine learning algorithms would be a decisive argument to justify the development, and therefore, the investment on quantum computing technologies. Machine learning stands at the forefront of state-of-the-art technology. With the availability of vast amounts of data, powerful computational resources, and advancements in algorithms, machine learning is transforming various industries. After this incentives, researchers hunt for impressive advantages of adding "quantum" into the already well established field of machine learning.

The commercial expectations frequently come into conflict with the complex reality of scientific research, where highly complex questions rarely yield straightforward yes-or-no answers.

Firstly, it is not possible to compare in general quantum machine learning and classical machine learning. It would be like comparing a car and a bike, while they both are means of transportation if we only consider their speed, the comparison may favor the car but if we focus on factors like enviromental impact, cost or ease of maneuverability in traffic the comparison could favor the bike. To be able to talk about the advantage of quantum machine learning regarding classical machine learning, we would need to fix some parameters: precise what kind of quantum computer we are considering- NISQ, ideal and error corrected-, the task in machine learning- supervised, unsupervised-, the data- size, density-, and understand what we mean by advantage, which is not simply a question of speed. According to [SP21], we can identify four important markers on quantum advantage:

1. **Performance.** The algorithm effectively addresses a learning task, demonstrating its ability to generalize from data samples to previously unseen data.

2. **Speedup.** The algorithm shows a faster execution compared to classical competitors that could be used in practice.

3. **Relevance.** The algorithm solves a problem that is relevant to machine learning.

4. **Availability.** There is a reasonable likelihood that the technology required to implement the algorithm will become available in the near future.

All four criteria are important in a way to answer the question on how could quantum computing revolutionize machine learning, since there cannot be a true advantage if we match one criteria but not the others. Consider a quantum algorithm with a remarkable generalisation performance but that does not provide any speedup to the classical methods, then we do not need a quantum computer, it provided us a new better classical algorithm. Instead, if there is a speedup, but only for problems of pathological relevance, thus it does not provide much. Finally, considering an algorithm that exceeds in its generalisation performance, execution speed for a relevant problem but only for a type of computer that may be available in a distant future, it is more likely that in the meantime machine learning would progress to a point that it would be needed to reevaluate in the future the advantage of the algorithm against a more advanced machine learning status.

Currently there is no quantum machine learning algorithm that fulfills all four of this requirements successfully.

Next, we will discuss more especifically on some matters we have previously mentioned.

5.1. Generalisation performance of quantum models

The generalisation performance, or also known as generalisation power, of a machine learning algorithm refers to its ability to learn from a limited set of data samples and be able to generalise it to previously unseen data. Generalisation requires a machine learning model to be expressive enough to learn a pattern without overfitting. A good model is able to minimise the training error while having the smallest possible expressivity (complexity of functions that can possibly be computed by a parametric function such as a neural net). Then, the expressivity is a crucial factor on the generalisation performance. It is not just a property of the model, it needs to take into account the training procedure.

In the case of kernel theory, the expressivity of a quantum model is determined by the data encoding feature map, which induces a quantum kernel κ that is used to measure distances between data points. Remembering the relationships studied in section 3.1 between feature maps, kernels and reproducing kernel Hilbert, we can deduce that the espressivity directly depends on the size of the reproducing kernel Hilbert space, formed by the linear combinations of functions $\{\kappa(x, \cdot)\}$ centred in all points $x \in \mathcal{X}$ of the input domain.

The quantum support vector machine obtained using basis encoding, which maps a n bit binary string $x = (b_1, ...b_n)$ with $b_i \in \{0, 1\}, \quad i = 1, ..., n$ to a computational basis state, gives us an example of overy expressive kernel. This data encoding maps data to the corners of a hypercube in $\mathbb{C}^{2^n}$. Two classes of data always become linearly separable, but the inner-product between all data points is the same, thus it does not provide useful statements about unseen data points because they uniformly have a zero distance to the training points. A support vector machine will still be able to learn to classify the training data well, by learning the label of each training data point as a weight α_i in Theorem 3.4. Therefore, the kernel induced by a basis encoding feature map will have zero training error and very large test error. This is shown in the reproducing kernel Hilbert space induced by such kernel: it consists of linear combinations of Kronecker delta functions centred in all points $x \in \mathcal{X}$ of the input set, hence functions that are not smooth, in other words, that can show big variations in $f(x)$ for small variations in x. Consequently, it is likely to end up in overfitting.

The main question about generalisation performance of quantum models is still unresolved: How can quantum models strike a balance between flexibility to achieve a low training error and simplicity to prevent overfitting? Furthermore, how might genuine quantum phenomena, such as coherent superpositions of entire datasets or interference in quantum kernels, enhance generalization for specific tasks? Addressing these inquiries requires a genuinely interdisciplinary research approach that incorporates the insights from theory-driven aspects of quantum mechanics and classical machine learning. It is being researched the implications of adapting quantum machine learning algorithms to open quantum systems, aiming to open pathways enhancing the potential of noise and decoherence ([OALCP23]).

5.2. Speedup of quantum machine learning

There are two interpretations of speedup. On the one hand, there is the speedup whose goal is to take a specific classical machine learning algorithm and implement it faster on a quantum computer. This approach would consider it as a potential or limited speedup. According to this interpretation of speedup, the possible quantum speedups derive directly from known results in quantum computing research. These speedups become notably interesting when the subroutine represents a significant bottleneck in the runtime, as the speedup can lead to an overall improvement in the algorithm's runtime. This is often feasible when we make assumptions about the speed at which data can be loaded into the quantum computer.

On the other hand, the second interpretation of speedup refers to the speedup that quantum computers can provide to the process of learning: provided with a dateset, can a quantum computer help the process of identifying patterns in this data, achieving equivalent performance to classical computing but at a faster pace? For this we need to compare the speed and performance of the quantum algorithm to any classical machine learning algorithm. This approach presents greater challenges. Firstly, we need to know the performance of the quantum model. Secondly, according to the no-free-lunch theorems [WM97], no machine learning method can universally excel on every dataset, therefore we need to impose assumptions on the data itself. Consequently, we can only study such speedups under highly controlled conditions.

5.3. The size of the dataset

As we have seen, feature maps are a common element of quantum support vector machines and quantum neural networks. They tackle the problem of "translating" the classical input data into quantum states. In many situations, this stage is the bottleneck of the algorithm since in general, data encoding costs linear time in the data size because every feature has to be addressed. If we think of basis encoding, it requires significant spatial constraints concerning data dimension. If each feature is encoded as a n-bit binary sequence, we can encode only $100/n$ features within 100 qubits.

The bottleneck of data encoding implies that, with the exception of a few specific scenarios, quantum machine learning is unlikely to provide practical solutions for processing big data in the near future. It is needed to adopt solutions to deal with high-dimensional data. However, it's worth noting that certain algorithms, particularly hybrid training approaches, can overcome this limitation by processing a subset of samples from the dataset at a time, which alleviates the constraints on the number of data samples.

The promise of quantum techniques for handling big data sound appealing but the challenges are indeed daunting. There is an undeniable global enthusiasm surrounding big data, nonetheless, there are numerous scenarios where data collection is costly or naturally constrained; hence, predictors have to be built for extremely small datasets. If quantum computing can demonstrate a qualitative advantage, it will undoubtedly find valuable applications in the realm of "small data".

5.4. The best of both worlds

Hybrid schemes in which only a small part of the model is processed by a quantum device, align better with intermediate-term technologies. Hybrid quantum algorithms offer the advantage of employing quantum devices for relatively short tasks, alternating with classical computations. Another significant advantage of hybrid methods is that their parameters are accessible as classical data, making them straightforward to store and use for predicting multiple inputs.

Amongst hybrid algorithms, variational circuits are particularly promising: the combination of quantum neural networks with classical neural networks, since they are two models that naturally fit together. The general approach involves incorporating a quantum neural network as a layer within a conventional classical neural network. In this way, the "quantum layer" will take as input the outputs of the previous layer (or the inputs to the model, if there's no layer before it) and will forward its output to the next layer (if there are any). The output of the quantum neural network will be a numerical array of length n; thus, in the eyes of the next layer, the quantum layer will behave as if it were a classical layer with n neurons.

Variational algorithms in the context of machine learning open a rich research area, with the prospect of developing an entirely new field within classical machine learning.

5.5. Future perspectives

Quantum computing is a field with strong roots in theory. While the mathematical foundations were essentially established in the 1930s, we continue to explore their practical applications up to today. Algorithmic development in quantum computing often relies on theoretical proofs to demonstrate the potential of quantum algorithms, especially when hardware constraints make numerical arguments difficult to employ. On the other hand, machine learning methods are largely of practical nature and breakthroughs are often the result of huge numerical experiments. The merits of machine learning range from its computational complexity to the generalisation power of a model, its ease of use, mathematical and algorithmic simplicity and wide applicability.

When trying to put together these two fields, we suffer some limitations due to the reduced scale and resilience to the noise of current quantum computers, which in turn affects possible results we can obtain executing quantum machine learning algorithms on real quantum hardware. The goal of demostrating superior performance of methods that—at least at this stage—we only have limited practical access to is challenging. We have to take into account that quantum machine learning is a relatively young research discipline, although it has already come a long way and we are not pessimistic. Scientist will continue researching and expanding the boundaries of our understanding, exploring new techniques for processing information with quantum devices.

Our discussion so far has been oriented on what can quantum computing provide to machine learning. However, quantum machine learning may gain perspective by shifting the roles and asking what machine learning can offer to quantum computing. As we had mentioned before, machine learning can be the "greatest application" that makes quantum computing commercially viable by connecting the emerging fields of quantum computing with an already established wealthy market. But beyond the economic interests, perhaps machine learning can inspire quantum computing in NISQ era to add methods to its "toolbox", looking for more practical methods even if they are less rigorous. Machine learning is a field of computer science that revolves about the idea of what it means to learn. Quantum machine learning extends the idea of learning into the realm of quantum information processing, raising numerous abstract questions that aim our knowledge and research rather than finding commercial applications.

Limiting the scope of quantum machine learning may help obtaining better results, more solid and powerful models and addressing some matters such as what makes a good data embedding or a good trainable variational form. Models use designs where circuits inherit traditional quantum gates whose goal is to enhance the expressive power. However, this approach may be more driven by our limited understanding rather than any other fundamental reason. In classical machine learning, neural networks do not rely on arbitrary structures, but are based on fundamental mechanisms derived form biological processes. If we take into account Gaussian kernels, they rise from the natural distance measures of Gaussian functions. Although both can be used to asymptotically approximate any function, they are based on specific computations, far away from being an ansatz for "general classical computations". To succeed in building powerful models from quantum circuits we need to find a basic structure of quantum models equivalent to what the perceptron and the Gaussian are to classical machine learning.

The restriction of the scope of quantum machine learning should also be applied to the problems, datasets and applications that it is targeted to. Most quantum computing researchers agree that it is unlikely that quantum computers will supersede classical computers as a whole, instead, quantum computers could turn out to be convenient accelerators or special-purpose hardware devices for very specific applications.

Optimistically quantum machine learning may be one of the first applications for quantum computers, but pessimistically, it may also just remain as a research discipline because machine learning on classical computers has already an impressive performance. Quantum machine learning is not just a collection of applications, but a new exciting, challenging and intriguing approach to learning from data when it is processed according to the laws of quantum mechanics, which draws a relevant connection between the quantum nature of the world and the abstract concept of learning. In any case, the possibilities offered by quantum computing deserve the intense research activity that is currently carried on. It is important leaving aside sensationalist claims on quantum computing and being realistic with the possibilities of it at its current near-term noisy state.

6. Conclusion

In this thesis, we have explored from a mathematical point of view supervised machine learning principles, focusing on support vector machines and neural networks. We have presented the quantum mechanics model and formulated the theoretical framework of quantum machine learning, targeting quantum support vector machines and quantum neural networks with whom we have also worked in the practical part. The experimentation and implementation section of this thesis was fruitfully completed. We believe that we've largely achieved our primary goals as set out initially.

Throughout chapters 1 and 2 we have acquired a solid understanding of the two main components of the puzzle: supervised machine learning and quantum computing. We have followed the objectives we had set for each chapter. This has allowed us to later focus on quantum machine learning on chapter 3. Understanding this field from CQ perspective (classical data and quantum computation) we have studied quantum support vector machines and quantum neural networks mathematical framework, completing the objectives for this chapter. This effort towards understanding supervised machine learning, quantum computing and its synergies in quantum machine learning was rewarded for the elaboration of chapter's 4 practical work.

Chapter 4 has been a really stimulating chapter. Not only it served its purpose of getting acquainted with PennyLane and the implementation of the quantum machine learning models studied in theory, but also the quantum nature of the problem and the results obtained were remarkable. We understood the non-trivial problem of detecting quantum entanglement and its subtleties through looking at it from a quantum machine learning lens. As Richard Feynman suggested at his opening speech "Simulating Physics with computers" [Fey82], by looking at the problem from a quantum computing perspective, we gained some valuable insights about the problem itself. Additionally, we achieved up to 96.9% of accuracy with quantum support vector machines and 80.6% of accuracy with our best configuration of quantum neural network proved to be satisfactory results for our first interaction with a quantum machine learning application on a real world scenario. This proved that not only the theory of quantum machine learning is developed, but also the tools and resources are ready to be used for their implementation at least at a small scale due to the constraints by simulators and limited number of qubits.

Regarding future areas of development of this work, we present several approaches. Firstly, this work could be continued shifting the focus to unsupervised machine learning and its translation to quantum computing to complete the panorama of machine learning and its quantum counterpart.

Secondly, the case study on detection of quantum entanglement has room for further research on several ways. Quantum neural networks have more aspects to consider (its architecture and variational form) and it can be performed a more exhaustive analysis, especially if working with a device with GPU acceleration and more resources to deal with the long training times. The availability of more powerful simulators and larger quantum systems (with more qubits) will also open new possibilities for the difficulties we had in trying data embedding techniques, reducing the dimensionality of the dataset and training our models.

It would also add a supplementary dimension to the problem to run the quantum machine learning models on real quantum hardware such as IBM's quantum computers.

Lastly, PennyLane's great interoperability with TensorFlow unlocks the possibility of building hybrid architectures: models that combine classical models with other quantum-based models by joining them together and training them as a single unit. In particular, we have hybrid quantum neural networks at one's fingertips thanks to Keras which is a valuable tool for their implementation since they are classical neural networks in which one or more of its layers have been replaced by quantum layers.

To conclude, it is deserved to carry on with the research activity in quantum machine learning and we encourage to take part in it, taking advantage of current frameworks such as PennyLane based on this thesis promising results. With research and the evolution of quantum resources, we will build and find out the role of this discipline, keeping in mind that there is no fight for the future between classical and quantum machine learning. Quoting Nelson Mandela, "I never lose, I either win or learn"

Glossary

$\mathcal{X}$	Input domain.
$\mathcal{Y}$	Output domain.
$\mathcal{D}$	Dataset $\{(x_1, y_1), ..., (x_M, y_M)\} \subset \mathcal{X} \times \mathcal{Y}$
$\mathcal{F}$	Feature space.
$\mathbb{H}$	Hilbert space
$\mathcal{H}$	Hypothesis space
$\{f\}$	family of model functions
κ	Kernel function.
$\mathcal{R}$	Reproducing kernel Hilbert space
$\mathcal{R}_\kappa$	Reproducing kernel Hilbert space induced by the kernel κ.
$\hat{\mathcal{R}}_f$	Empirical risk a model f over a dataset $\mathcal{D}$
Φ	Feature map from a data input space to a feature space.
θ	Set of parameters of a machine learning model.

A. Appendix: Auxiliary theorems

Theorem A.1 (Spectral theorem). *Let $\mathbb{H}$ be a separable Hilbert space and let T be a compact self-adjoint operator. Then there exists a Hilbert basis (a orthonormal basis) composed of eigenvectors of T. [Bre11] Theorem 6.11*

Theorem A.2 (Hausdorff theorem). *Every linear biyective application between finite dimension normed spaces is an isomorphism. [Pay]*

Theorem A.3 (Riesz-Fréchet representation theorem). *Let $\mathbb{H}$ be a Hilbert space and $\Lambda \in \mathbb{H}^*$. Then it exists a unique $y \in \mathbb{H}$ such that $\Lambda = \Lambda_y$, that is, $\Lambda(x) = \langle x, y \rangle \quad \forall x \in \mathbb{H}$.*

Moreover, the application

$$\begin{aligned} \mathbb{H} &\longrightarrow \mathbb{H}^* \\ y &\longmapsto \Lambda_y \end{aligned}$$

is an isometry (isomorphism that preserves the norm: $||\Lambda|| = ||y||$) and it is linear conjugate. [Bre11] Theorem 5.5

Theorem A.4 (Orthogonal projection theorem). *Given $\mathbb{H}$ a Hilbert space and $\mathcal{Y}$ a closed subset in $\mathbb{H}$. Then,*

1. *The orthogonal projection $P_{\mathcal{Y}}$ from $\mathbb{H}$ to $\mathcal{Y}$ verifies that $\forall x \in \mathbb{H}$, $P(x)$ is the only point in $\mathcal{Y}$ so that $||P(x) - x|| = \text{dist}(x, \mathcal{Y})$. Therefore, $P(x) = P_{\mathcal{Y}}(x)$*
2. *$\forall x \in \mathbb{H}$, $||x||^2 = ||P_{\mathcal{Y}}(x)||^2 + ||x - P_{\mathcal{Y}}(x)||^2$. In particular, $P_{\mathcal{Y}}$ is continuous.*
3. *$\mathcal{X} = \mathcal{Y} \oplus \mathcal{Y}^{\perp}$*
4. *$\mathcal{Y}^{\perp\perp} = \mathcal{Y}$ and $P_{\mathcal{Y}^{\perp}} = \text{id}_{\mathcal{H}} - P_{\mathcal{Y}}$*

[Med23]

Theorem A.5 (Schmidt decomposition). *Suppose $|\psi\rangle$ is a pure state of a composite sytem, AB. Then there exist orthonormal states $|i_A\rangle$ for system A, and orthornormal states $|i_B\rangle$ of system B such that*

$$|\psi\rangle = \sum_i \lambda i_i \, |i_A\rangle \, |i_B\rangle$$

where λ_i are non-negative real numbers satisfying $\sum_i \lambda_i^2 = 1$.
[NC10] Theorem 2.7

B. Appendix: Notebook and implementation

This appendix contains the notebook elaborated for chapter 4. It contains the code, output and notes. The notebook and data are released at `https://github.com/anamarsabi/tfg` for reproducibility.

Case study: binary classification problem of entanglement detection

Ana Martínez Sabiote

The quantum separability problem consists in deciding whether a bipartite density matrix is entangled or separable. Finding the Schmidt decomposition of a state to determine if its separable or not is an NP-hard problem. In this notebook we will propose QSVM and QNN methods to tackle problem as a binary classification task using the data produced by https://gitlab.lis-lab.fr/balthazar.casale/ML-Quant-Sep

This repository publishes two fully labeled datasets of 6,000 bipartite density matrices. The first contains density matrices of dimension 9×9 of a bipartite quantum system $H = H_a \otimes H_b$, with $\rho\_a = \dim(H_a) = 3$ and $\rho_b = \dim(H_b) = 3$. The second is composed of density matrices of size 49×49, thus $\rho_a = \rho_b = 7$. Each dataset is a collection of pairs of input density matrices and labels indicating whether the corresponding density matrix is separable or entangled, and contains separable (SEP), PPT entangled (PPT-ENT) and non-PPT (NPPT-ENT) density matrices with 2,000 examples each.

We will constraint our experimentation to the first dataset for a bipartite system of $\rho_a = \rho_b = 3$.

We will start taking 100 samples of each label to form our training dataset, resulting in a perfectly balanced dataset. Using the code from the repository, we have read the separable data, PPT entangled data and NPPT entangled data and joined them to make a np.ndarray of 200 density matrix of dimensions 9 x 9. Then, the representations.py module has been used to transform the np.ndarray of density matrices into a real-valued vector of 200 samples and 80 attributes. We have saved this arrays into a csv file. An analogous process has been performed for creating the test set.

We can deal with 200 data instances, but the challenging aspect of this problem is the amount of attributes. For applying the quantum machine learning techniques we have studied, we firstly need to encode this great number of features.

1 Work setup

```
[ ]: from google.colab import drive
     drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

```
[ ]: pip install pennylane==0.26
```

```
Collecting pennylane==0.26
  Downloading PennyLane-0.26.0-py3-none-any.whl (1.0 MB)
```

```
[ ]: pip install tensorflow==2.9.1
```

```
Collecting tensorflow==2.9.1
  Downloading
tensorflow-2.9.1-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl
(511.7 MB)
```

```
[ ]: pip install pyyaml h5py
```

```
[ ]: import pandas as pd
     import numpy as np
     import pennylane as qml
     import tensorflow as tf
     import matplotlib.pyplot as plt

     from sklearn.decomposition import PCA
     from sklearn.svm import SVC
     from sklearn.metrics import accuracy_score
     from sklearn.metrics import f1_score
     from sklearn.metrics import confusion_matrix
     from sklearn.metrics import ConfusionMatrixDisplay
     from sklearn.preprocessing import MaxAbsScaler
     from sklearn.model_selection import KFold

     import seaborn as sns

     import joblib
```

```
[ ]: # pennylane works with doubles and tensorflow works with floats.
     # We ask tensorflow to work with doubles

     tf.keras.backend.set_floatx('float64')
```

2 Pre-work to get started

2.1 Data preprocessing

Overview of the dataset description. Should we normalise or scale the values? they are all really close to 0.

```
[ ]: training_data = pd.read_csv("/content/drive/MyDrive/tfg/x_train_1.csv", header=None)
     training_data.head()
```

```
[ ]:          0         1         2         3         4         5         6  ...
     0  0.002405 -0.013551  0.014313  0.007182 -0.008649 -0.003936 -0.002174
     1 -0.006249  0.010679 -0.000343 -0.003259 -0.001029 -0.008732 -0.002129
     2  0.012010  0.006747  0.006879  0.000910 -0.001017 -0.003304 -0.003681
     3 -0.147372 -0.009079 -0.008336 -0.021830 -0.012960 -0.004989  0.054233
     4 -0.010161 -0.021211 -0.009640 -0.004726  0.000182 -0.013206 -0.001762
```

```
         74        75        76        77        78        79
0  0.045722  0.076358  0.083020  0.093225  0.105654  0.109198
1  0.061884  0.080446  0.095130  0.103344  0.103996  0.110223
2  0.028129  0.060973  0.073608  0.078672  0.095634  0.102708
3  0.105213  0.087479  0.105942  0.130937  0.111882  0.118560
4  0.055780  0.075803  0.095085  0.105444  0.107857  0.114460

[5 rows x 80 columns]
```

```
[ ]: training_data.describe()
```

```
[ ]:              0           1           2           3           4           5  ...
count  200.000000  200.000000  200.000000  200.000000  200.000000  200.000000
mean     0.004797    0.000064    0.000108   -0.000144    0.000498    0.001381
std      0.028860    0.029620    0.036638    0.025325    0.026439    0.032549
min     -0.147372   -0.081901   -0.187091   -0.069290   -0.080485   -0.159721
25%     -0.011422   -0.019337   -0.015301   -0.011488   -0.010467   -0.014435
50%      0.005595    0.000423    0.000219   -0.000011   -0.000762    0.001258
75%      0.024328    0.016108    0.018386    0.007331    0.010476    0.018243
max      0.078962    0.099702    0.170843    0.073404    0.087689    0.114851


               77          78          79
count  200.000000  200.000000  200.000000
mean     0.097276    0.106195    0.112494
std      0.013057    0.010411    0.008404
min      0.057148    0.059018    0.070539
25%      0.091099    0.100307    0.108088
50%      0.096756    0.107006    0.112603
75%      0.104528    0.111522    0.116679
max      0.162879    0.162101    0.152014

[8 rows x 80 columns]
```

```
[ ]: x_train = np.genfromtxt("/content/drive/MyDrive/tfg/x_train_1.csv",␣
      ↪delimiter=",",dtype=None)
     y_train = np.genfromtxt("/content/drive/MyDrive/tfg/y_train_1.csv",␣
      ↪delimiter=",",dtype=None)

     x_test_small = np.genfromtxt("/content/drive/MyDrive/tfg/x_test_small.csv",␣
      ↪delimiter=",",dtype=None)
     y_test_small = np.genfromtxt("/content/drive/MyDrive/tfg/y_test_small.csv",␣
      ↪delimiter=",",dtype=None)
```

```
[ ]: print(type(x_train))
     print(type(y_train))
     print(x_train.shape)
     print(y_train.shape)

     print(type(x_test_small))
     print(type(y_test_small))
     print(x_test_small.shape)
     print(y_test_small.shape)
```

```
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
(200, 80)
(200,)
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
(300, 80)
(300,)
```

2.2 Quantum support vector machines

```
[ ]: # Amplitude encoding of 80 variables using 7 qubits (can encode up to 128 inputs)

     # Number of qubits of the system
     nqubits = 7
     # We define a device
     dev = qml.device("lightning.qubit", wires = nqubits)

     # We define de circuit of our kernel. We use AmplitudeEmbedding which returns an
     # operation equivalent to amplitude encoding of the first argument
     # Since the vector has 80 components which is not a power of 2, we extend the vector
     # to 128 components using padding with 0.
     @qml.qnode(dev)
     def kernel_circ(a,b):
         qml.AmplitudeEmbedding(a, wires=range(nqubits), pad_with=0, normalize=True)
         # Computes the adjoint (or inverse) of the amplitude encoding of b
         qml.adjoint(qml.AmplitudeEmbedding(b, wires=range(nqubits), pad_with=0,␣
      ↪normalize=True))     # We return an array with the probabilities fo measuring each␣
      ↪possible state in the
         # computational basis
         return qml.probs(wires=range(nqubits))
```

```
[ ]: # Check that the circuit works as expected
     k=kernel_circ(x_train[0], x_train[1])
     print(k.shape)
```

```
(128,)
```

```
[ ]: # Checking we get a 1 in the first entry of the returned array when the two arguments␣
      ↪are the same
     kernel_circ(x_train[0], x_train[0])
```

```
[ ]: tensor([1.00000000e+00, 7.40406749e-32, 1.66038413e-32, 1.38834029e-32,
             4.92602587e-33, 1.42263375e-32, 1.88747586e-31, 4.36044173e-32,
             ...,
             2.52937313e-34, 2.07789578e-33, 5.23592488e-35, 1.95060277e-34],
     requires_grad=True)
```

Even if amplitude encoding can be computed, then it takes more than 1h execture the following cell to train the SVM (16384 iterations).

[ ]:
```
"""
from sklearn.svm import SVC
def qkernel(A, B):
  return np.array([[kernel_circ(a,b)[0] for b in B] for a in A])

svm = SVC(kernel = qkernel).fit(x_train, y_train)
"""
```

[ ]:
```
"""
from sklearn.metrics import accuracy_score
print(accuracy_score(svm.predict(x_test), y_test))
"""
```

[ ]:
```
'\nfrom sklearn.metrics import
accuracy_score\nprint(accuracy_score(svm.predict(x_test), y_test))\n'
```

Therefore, its needed to apply some dimensionality reduction techniques to work with less attributes such as principal component analysis or autoencoding.

Then, regarding embeddings (encoding input features into the quantum state of the circuit) - Amplitude encoding: we need n qubits for $N \leq 2^n$ attributes. - Angle enconding: it encodes N features into the rotation angles of n qubits, where $N \leq n$. The other built in encodings that PennyLane offers need a number of qubits greater or equal to the number of features. Since we have so many attributes, even if we reduce its dimensionality, I think we would need too many qubits for applying this kind of encoding.

2.2.1 PCA (64) with amplitude encoding

Next, we are goint to try a different approach: reducing the dimensionality of a dataset while minimizing information loss. We will use principal component analysis to reduce the number of variables in our dataset from 80 to 64, to apply amplitude enconding on 6 qubits instead of 7.

[ ]:
```
pca = PCA(n_components = 64)

xs_train = pca.fit_transform(x_train)
xs_test = pca.transform(x_test)
```

[ ]:
```
print(xs_train.shape)
```

```
(200, 64)
```

[ ]:
```
# Amplitude encoding of 64 variables using 6 qubits (can encode up to 64 inputs)

# Number of qubits of the system
nqubits = 6
# We define a device
dev = qml.device("lightning.qubit", wires = nqubits)

# We define de circuit of our kernel. We use AmplitudeEmbedding which returns an
# operation equivalent to amplitude encoding of the first argument
```

```
@qml.qnode(dev)
def kernel_circ(a,b):
    qml.AmplitudeEmbedding(a, wires=range(nqubits), pad_with=0, normalize=True)
    # Computes the adjoint (or inverse) of the amplitude encoding of b
    qml.adjoint(qml.AmplitudeEmbedding(b, wires=range(nqubits), pad_with=0,␣
↪normalize=True))     # We return an array with the probabilities fo measuring each␣
↪possible state in the
    # computational basis
    return qml.probs(wires=range(nqubits))
```

[ ]:
```
# Check that the circuit works as expected
ks=kernel_circ(xs_train[0], xs_train[1])
print(ks.shape)
```

```
(64,)
```

[ ]:
```
def qkernel(A, B):
  return np.array([[kernel_circ(a,b)[0] for b in B] for a in A])
```

[ ]:
```
svm = SVC(kernel = qkernel).fit(xs_train, y_train)
```

[ ]:
```
xs_test_small = pca.transform(x_test_small)
print(accuracy_score(svm.predict(xs_test_small), y_test_small))
```

```
1.0
```

Summary of this case

PCA 64 attributes, amplitude encoding, lightning qubit device, 6 qubits

Accuracy on a test set of 300 instances is 1.0. Suspicious?

Training took 28 min

Prediction took 44 min

2.2.2 PCA (32) with amplitude encoding

[ ]:
```
pca = PCA(n_components = 32)

xs_train = pca.fit_transform(x_train)
xs_test_small = pca.transform(x_test_small)
```

[ ]:
```
# Amplitude encoding of 64 variables using 6 qubits (can encode up to 64 inputs)

# Number of qubits of the system
nqubits = 5
# We define a device
dev = qml.device("lightning.qubit", wires = nqubits)

# We define de circuit of our kernel. We use AmplitudeEmbedding which returns an
# operation equivalent to amplitude encoding of the first argument
@qml.qnode(dev)
```

```
def kernel_circ(a,b):
    qml.AmplitudeEmbedding(a, wires=range(nqubits), pad_with=0, normalize=True)
    # Computes the adjoint (or inverse) of the amplitude encoding of b
    qml.adjoint(qml.AmplitudeEmbedding(b, wires=range(nqubits), pad_with=0,␣
↪normalize=True))     # We return an array with the probabilities fo measuring each␣
↪possible state in the
    # computational basis
    return qml.probs(wires=range(nqubits))
```

[ ]:
```
svm = SVC(kernel = qkernel).fit(xs_train, y_train)
```

[ ]:
```
print(accuracy_score(svm.predict(xs_test_small), y_test_small))
```

```
0.9666666666666667
```

Summary of this case

PCA 32 attributes, amplitude encoding, lightning qubit device, 5 qubits

Accuracy on a test set of 300 instances is 0.9666

Training took 15 min

Prediction took 23 min

2.3 Quantum neural networks

[ ]:
```
# We set a seed for the packages so the results are reproducible
seed=4321
np.random.seed(seed)
tf.random.set_seed(seed)
```

[ ]:
```
def plot_losses(history):
  tr_loss = history.history["loss"]
  epochs = np.array(range(len(tr_loss))) + 1
  plt.plot(epochs, tr_loss, label="Training loss")
  plt.xlabel("Epoch")
  plt.show()
```

2.3.1 PCA (32) with amplitude encoding, TwoLocal variational form, 5 qubits, default.qubit

[ ]:
```
pca = PCA(n_components = 32)

xs_train = pca.fit_transform(x_train)
xs_test_small = pca.transform(x_test_small)
```

[ ]:
```
# Two local variational form
def TwoLocal(nqubits, theta, reps=1):
  for r in range(reps):
    for i in range(nqubits):
      qml.RY(theta[r*nqubits+i], wires=i)
    for i in range(nqubits-1):
```

```
    qml.CNOT(wires=[i,i+1])

  for i in range(nqubits):
    qml.RY(theta[reps*nqubits+i], wires=i)
```

[ ]:
```
# Hermitian matrix
state_0 = [[1], [0]]
M = state_0 * np.conj(state_0).T
```

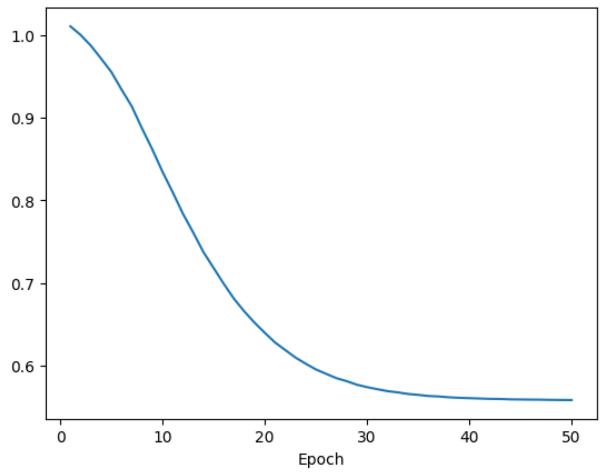
[ ]:
```
nqubits=5
dev=qml.device("default.qubit", wires=nqubits)

def qnn_circuit(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  TwoLocal(nqubits=nqubits, theta=theta, reps=1)
  return qml.expval(qml.Hermitian(M, wires=[0]))

qnn = qml.QNode(qnn_circuit, dev, interface="tf")
```

[ ]:
```
weights={"theta": 10}
# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:
```
# Training our model
history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 27s 3s/step - loss: 1.0103
Epoch 2/50
10/10 [==============================] - 35s 3s/step - loss: 0.9998
.
.
.
Epoch 49/50
10/10 [==============================] - 15s 2s/step - loss: 0.5584
Epoch 50/50
10/10 [==============================] - 15s 1s/step - loss: 0.5583
```

[ ]:
```
plot_losses(history)
```



```
[ ]: # Check accuracy

     tr_acc = accuracy_score(model.predict(xs_train) >= 0.5, y_train)
     test_acc = accuracy_score(model.predict(xs_test_small) >= 0.5, y_test_small)
```

```
7/7 [==============================] - 10s 1s/step
10/10 [==============================] - 13s 1s/step
```

```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.715
Test accuracy:  0.5966666666666667
```

Test accuracy is not great.

Aspects to consider - Training time 15 min - We can try other NN architectures, looking at paper Entanglement detection using Deep Neural networks - We don't have validation set!

2.3.2 PCA (32) with amplitude encoding, StronglyEntanglingLayers, default.qubit

```
[ ]: pca = PCA(n_components = 32)

     xs_train = pca.fit_transform(x_train)
     xs_test_small = pca.transform(x_test_small)
```

```
[ ]: # Hermitian matrix
     state_0 = [[1], [0]]
     M = state_0 * np.conj(state_0).T
```

```
[ ]: nqubits=5
     dev = qml.device("default.qubit", wires=nqubits)
```

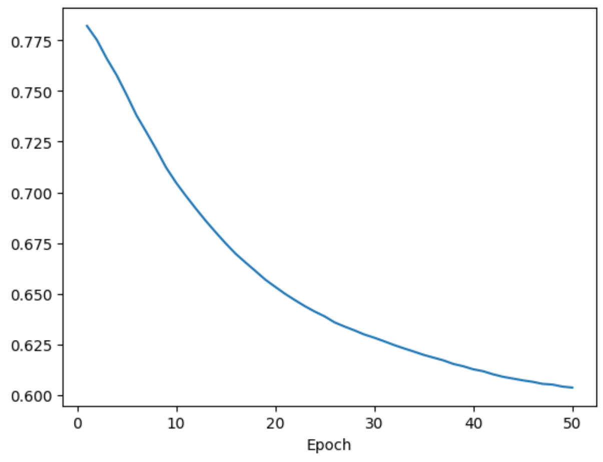
```
# number of repetitions that we want in each instance of the variational form
nreps = 2
#  dimensions of the input that the variational form expects
weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
#  number of inputs that each instance of the variational form will take
nweights = 3*nreps*nqubits

def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))

qnn_strong = qml.QNode(qnn_circuit_strong, dev)

# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

```
[ ]: # Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 31s 3s/step - loss: 0.7821
Epoch 2/50
10/10 [==============================] - 31s 3s/step - loss: 0.7751
.
.
.
Epoch 49/50
10/10 [==============================] - 31s 3s/step - loss: 0.6042
Epoch 50/50
10/10 [==============================] - 31s 3s/step - loss: 0.6037
```

```
[ ]: plot_losses(history)
```



```
[ ]: # Check accuracy

     tr_acc = accuracy_score(model.predict(xs_train) >= 0.5, y_train)
     test_acc = accuracy_score(model.predict(xs_test_small) >= 0.5, y_test_small)
```

```
7/7 [==============================] - 14s 2s/step
10/10 [==============================] - 22s 2s/step
```

```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.715
Test accuracy:  0.5733333333333334
```

Using StronglyEntanglingLayers the accuracy is basically identical to the previous case.

Observation, lightning.qubit in this case raises unknown error in tensorflow.

Training 25 minutes

2.3.3 PCA (32) with amplitude encoding, StronglyEntanglingLayers, lightning.qubit and adjoind differentiation method

```
[ ]: nqubits=5
     dev = qml.device("lightning.qubit", wires=nqubits)

     # number of repetitions that we want in each instance of the variational form
     nreps = 2
     #  dimensions of the input that the variational form expects
     weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
     #  number of inputs that each instance of the variational form will take
     nweights = 3*nreps*nqubits
```

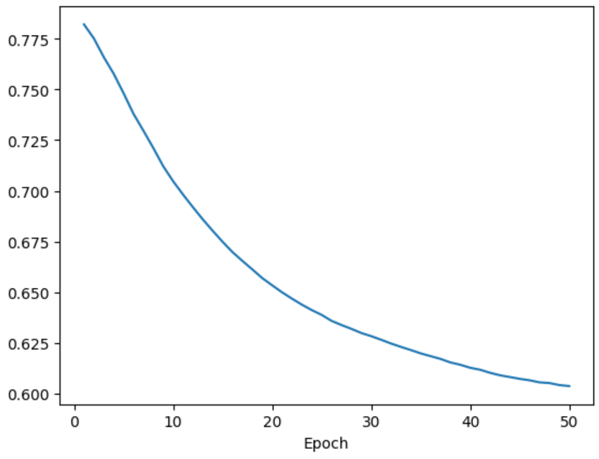
```
def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))


# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

[ ]:
```
method= "adjoint"

tf.random.set_seed(seed)

qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:
```
# Training our model
history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 15s 2s/step - loss: 0.7821
Epoch 2/50
10/10 [==============================] - 8s 835ms/step - loss: 0.7751
.
.
.
Epoch 49/50
10/10 [==============================] - 14s 1s/step - loss: 0.6042
Epoch 50/50
10/10 [==============================] - 13s 1s/step - loss: 0.6037
```

[ ]:
```
plot_losses(history)
```



```
[ ]: # Check accuracy

     tr_acc = accuracy_score(model.predict(xs_train) >= 0.5, y_train)
     test_acc = accuracy_score(model.predict(xs_test_small) >= 0.5, y_test_small)
```

```
7/7 [==============================] - 5s 660ms/step
10/10 [==============================] - 12s 1s/step
```

```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.715
Test accuracy:  0.5733333333333334
```

No improvements on accuracy, exact performance, which makes sense because we are running the exact same configuration. Although using lightning.qubit with adjoint differenciation method increases significantly the speed (averagely per epoch 30s -> 14s)

Training 11 minutes

2.3.4 PCA (64) with amplitude encoding, StronglyEntanglingLayers, lightning.qubit and adjoint differentiation method

```
[ ]: pca = PCA(n_components = 64)

     xs_train = pca.fit_transform(x_train)
     xs_test_small = pca.transform(x_test_small)
```

```
[ ]: # Hermitian matrix
     state_0 = [[1], [0]]
     M = state_0 * np.conj(state_0).T
```

```
[ ]: nqubits=6
     dev = qml.device("lightning.qubit", wires=nqubits)

     # number of repetitions that we want in each instance of the variational form
     nreps = 2
     #  dimensions of the input that the variational form expects
     weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
     #  number of inputs that each instance of the variational form will take
     nweights = 3*nreps*nqubits

     def qnn_circuit_strong(inputs, theta):
       qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
      ↪normalize=True)
       # reshape the theta array of parameters to make it fit into the shape that
       # the variational form expects
       theta1 = tf.reshape(theta, weights_dim)
       qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

       return qml.expval(qml.Hermitian(M, wires = [0]))

     # dictionary we would send to TensorFlow when constructing the Keras layer
     weights_strong = {"theta": nweights}
```

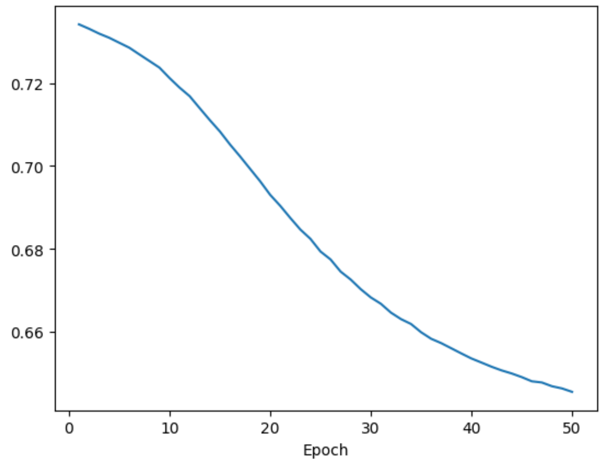
```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

     # keras model
     model = tf.keras.models.Sequential([qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                         validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 18s 2s/step - loss: 0.7342
Epoch 2/50
10/10 [==============================] - 16s 2s/step - loss: 0.7331
.
.
```

```
.
Epoch 49/50
10/10 [==============================] - 16s 2s/step - loss: 0.6463
Epoch 50/50
10/10 [==============================] - 16s 2s/step - loss: 0.6455
```

```
[ ]: plot_losses(history)
```



```
[ ]: # Check accuracy

tr_acc = accuracy_score(model.predict(xs_train) >= 0.5, y_train)
test_acc = accuracy_score(model.predict(xs_test_small) >= 0.5, y_test_small)
```

```
7/7 [==============================] - 11s 1s/step
10/10 [==============================] - 15s 2s/step
```

```
[ ]: print("Train accuracy: ", tr_acc)
print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.66
Test accuracy:  0.63
```

Still not great accuracies even if we are working with more attributes (64). Since the training time was only 13 min thanks to the configuration of lightning.qubit + adjoint differentiation method, we can afford to try with 7 qubits and the original attributes, without reducing dimensionality with PCA.

2.3.5 80 attributes, 7 qubits, lightning.qubit and adjoint differentiation method

```
[ ]: nqubits=7
dev = qml.device("lightning.qubit", wires=nqubits)

# number of repetitions that we want in each instance of the variational form
nreps = 2
```

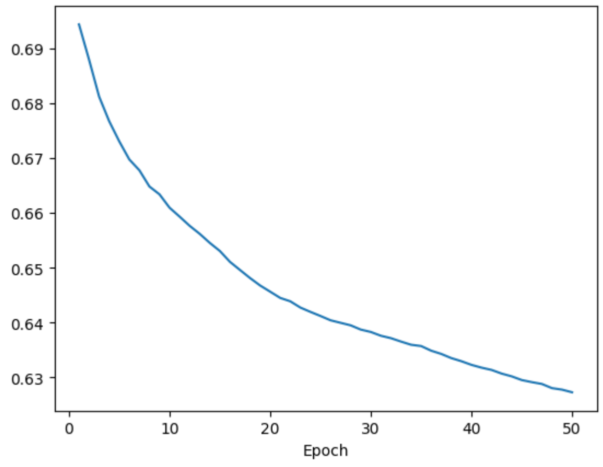
```
#  dimensions of the input that the variational form expects
weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
#  number of inputs that each instance of the variational form will take
nweights = 3*nreps*nqubits

def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))

# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

     # keras model
     model = tf.keras.models.Sequential([qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model.fit(x_train, y_train, epochs = 50, shuffle = True,
                         validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 17s 2s/step - loss: 0.6943
Epoch 2/50
10/10 [==============================] - 17s 2s/step - loss: 0.6879
.
.
.
Epoch 49/50
10/10 [==============================] - 17s 2s/step - loss: 0.6278
Epoch 50/50
10/10 [==============================] - 22s 2s/step - loss: 0.6273
```

```
[ ]: plot_losses(history)
```



```
[ ]: # Check accuracy

     tr_acc = accuracy_score(model.predict(xs_train) >= 0.5, y_train)
     test_acc = accuracy_score(model.predict(xs_test_small) >= 0.5, y_test_small)
```

```
7/7 [==============================] - 10s 1s/step
10/10 [==============================] - 16s 2s/step
```

```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.475
Test accuracy:  0.48333333333333334
```

Terrible results

Possible directions to continue
- Incorporating data preprocessing, since all the attributes have values really really close to 0 - Using other possible feature maps and dimensionality reduction techniques - Study the accuracy of each type of data (SEP, PPT-ENT and NPPT-ENT) - Generate dataset with different proportions of PPT-ENT and NPPT-ENT as done in the paper, train svm's with those variations of the dataset and compare the accuracy of those svm's. - another test_small dataset because it is not balanced. We are training with a perfectly balanced set and testing with a 1:2 dataset. - a different QNN architecture

```
[ ]: scaler = MaxAbsScaler()
     x_train_norm = scaler.fit_transform(x_train)

     x_test_norm = scaler.transform(x_test_small)

     # Restrict all the values to be between 0 and 1
     x_test_norm = np.clip(x_test_norm,0,1)
```

2.3.6 Normalized data, 80 attributes, 7 qubits, lightning.qubit and adjoint differentiation method

[ ]:
```
# Hermitian matrix
state_0 = [[1], [0]]
M = state_0 * np.conj(state_0).T
```

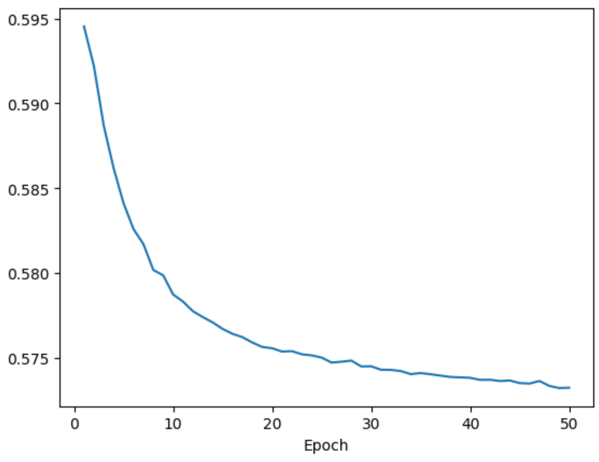
[ ]:
```
nqubits=7
dev = qml.device("lightning.qubit", wires=nqubits)

# number of repetitions that we want in each instance of the variational form
nreps = 2
#  dimensions of the input that the variational form expects
weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
#  number of inputs that each instance of the variational form will take
nweights = 3*nreps*nqubits

def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))

# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

[ ]:
```
method= "adjoint"

tf.random.set_seed(seed)

qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:
```
# Training our model
history = model.fit(x_train_norm, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 81s 8s/step - loss: 0.5945
Epoch 2/50
10/10 [==============================] - 16s 2s/step - loss: 0.5922
.
.
.
Epoch 49/50
10/10 [==============================] - 16s 2s/step - loss: 0.5732
Epoch 50/50
10/10 [==============================] - 16s 2s/step - loss: 0.5732
```

```
[ ]: plot_losses(history)
```



```
[ ]: # Check accuracy

     tr_acc = accuracy_score(model.predict(x_train_norm) >= 0.5, y_train)
     test_acc = accuracy_score(model.predict(x_test_norm) >= 0.5, y_test_small)
```

```
7/7 [==============================] - 12s 1s/step
10/10 [==============================] - 13s 1s/step
```

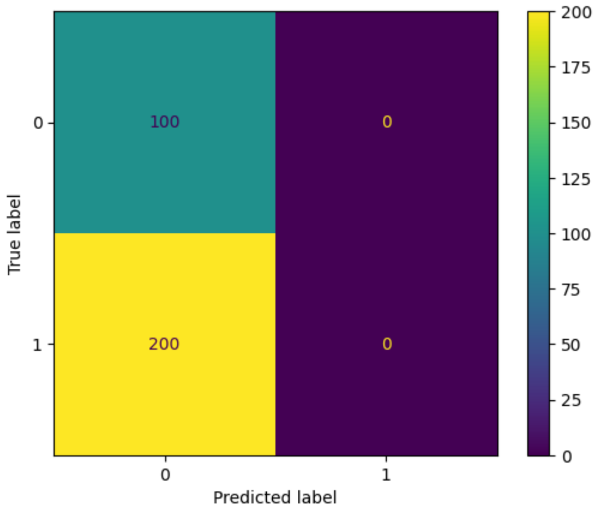
```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.545
Test accuracy:  0.3333333333333333
```

Even worst!

```
[ ]: cm = confusion_matrix(y_test_small, model.predict(x_test_norm) >= 0.5)

     cm_display = ConfusionMatrixDisplay(cm).plot()
```

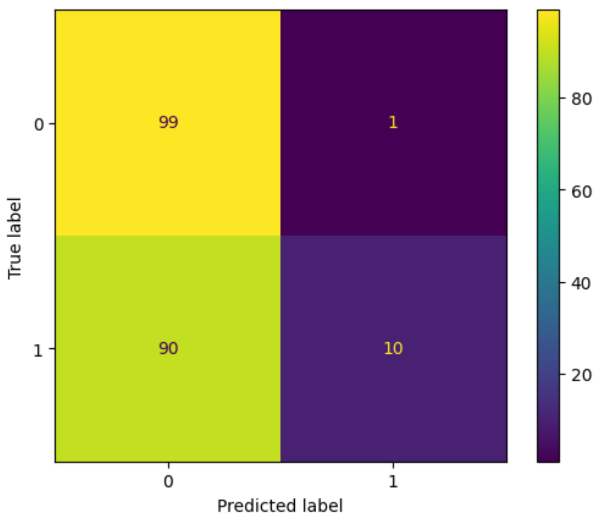
```
10/10 [==============================] - 16s 1s/step
```



It predicts all states separable, it doesn't detect entanglement at all. Let's see it it is because overfitting

```
[ ]: cm = confusion_matrix(y_train, model.predict(x_train_norm) >= 0.5)

     cm_display = ConfusionMatrixDisplay(cm).plot()
```

```
7/7 [==============================] - 8s 1s/step
```



With the training set it does not predict entanglement either

2.3.7 Normalized data, 80 attributes, 7 qubits, lightning.qubit,adjoint differentiation method and 4 repetitions in each instance of the variational form

```
[ ]: # number of repetitions that we want in each instance of the variational form
nreps = 4
#  dimensions of the input that the variational form expects
weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
#  number of inputs that each instance of the variational form will take
nweights = 3*nreps*nqubits

def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))

# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

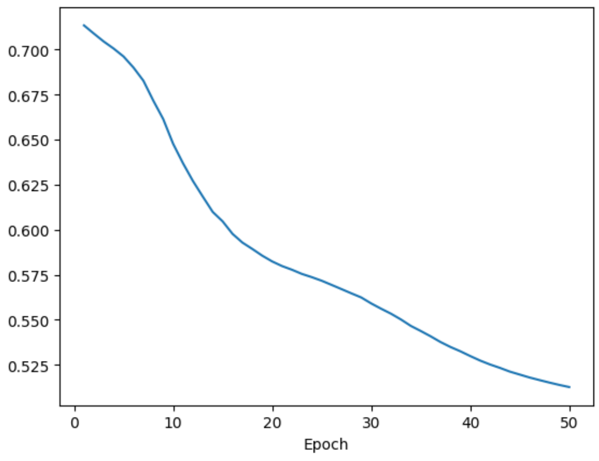
```
[ ]: method= "adjoint"

tf.random.set_seed(seed)

qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
history = model.fit(x_train_norm, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
```

```
10/10 [==============================] - 28s 3s/step - loss: 0.7134
Epoch 2/50
10/10 [==============================] - 27s 3s/step - loss: 0.7089
.
.
.
Epoch 49/50
10/10 [==============================] - 27s 3s/step - loss: 0.5139
Epoch 50/50
10/10 [==============================] - 22s 2s/step - loss: 0.5127
```

[ ]:
```
plot_losses(history)
```



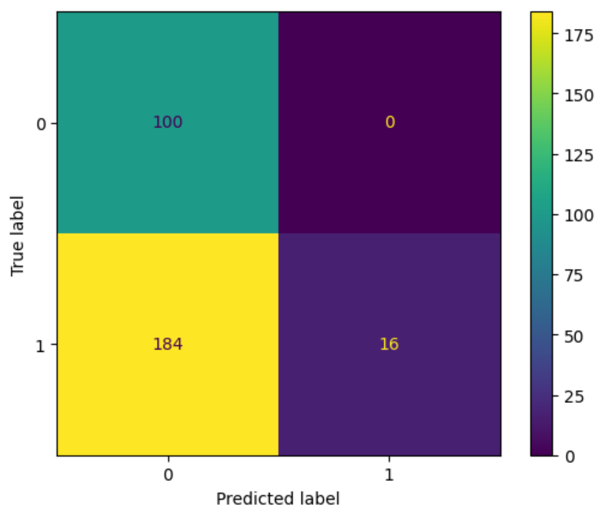
[ ]:
```
# Check accuracy
y_train_pred=model.predict(x_train_norm) >= 0.5
y_test_pred=model.predict(x_test_norm) >= 0.5

tr_acc = accuracy_score(y_train_pred, y_train)
test_acc = accuracy_score(y_test_pred, y_test_small)
```

```
7/7 [==============================] - 12s 1s/step
10/10 [==============================] - 17s 2s/step
```

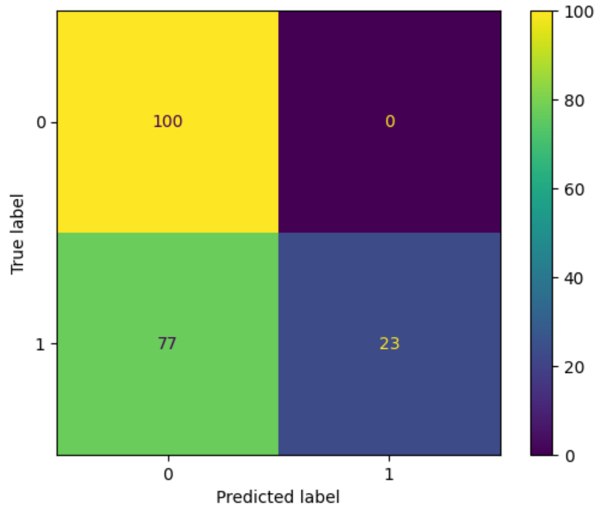
[ ]:
```
print("Train accuracy: ", tr_acc)
print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.615
Test accuracy:  0.38666666666666666
```

[ ]:
```
cm = confusion_matrix(y_test_small, y_test_pred)

cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: cm = confusion_matrix(y_train, y_train_pred)

     cm_display = ConfusionMatrixDisplay(cm).plot()
```



Not much improvement, still absolutely terrible accuracy. Let's discard this QNN with 80 attributes and 7 qubits

2.3.8 PCA (32) with amplitude encoding, Twolocal variational form, 5 qubits, lightning.qubit, adjoint differentiation method

[ ]:
```
pca = PCA(n_components = 32)

xs_train = pca.fit_transform(x_train_norm)
xs_test_small = pca.transform(x_test_norm)
```

[ ]:
```
# Two local variational form
def TwoLocal(nqubits, theta, reps=1):
  for r in range(reps):
    for i in range(nqubits):
      qml.RY(theta[r*nqubits+i], wires=i)
    for i in range(nqubits-1):
      qml.CNOT(wires=[i,i+1])

  for i in range(nqubits):
    qml.RY(theta[reps*nqubits+i], wires=i)
```

[ ]:
```
# Hermitian matrix
state_0 = [[1], [0]]
M = state_0 * np.conj(state_0).T
```

[ ]:
```
nqubits=5
dev=qml.device("lightning.qubit", wires=nqubits)

def qnn_circuit(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
↪normalize=True)
  TwoLocal(nqubits=nqubits, theta=theta, reps=1)
  return qml.expval(qml.Hermitian(M, wires=[0]))
```

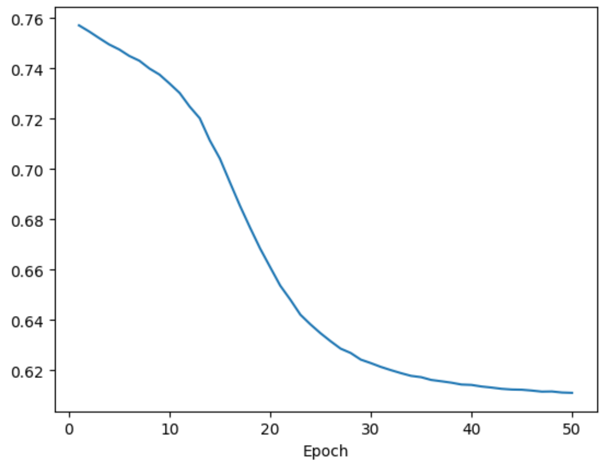
[ ]:
```
method= "adjoint"

tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

weights={"theta": 10}

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
```

```
model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                         validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 8s 657ms/step - loss: 0.7572
Epoch 2/50
10/10 [==============================] - 8s 892ms/step - loss: 0.7548
.
.
.
Epoch 49/50
10/10 [==============================] - 6s 439ms/step - loss: 0.6113
Epoch 50/50
10/10 [==============================] - 4s 441ms/step - loss: 0.6111
```

```
[ ]: plot_losses(history)
```



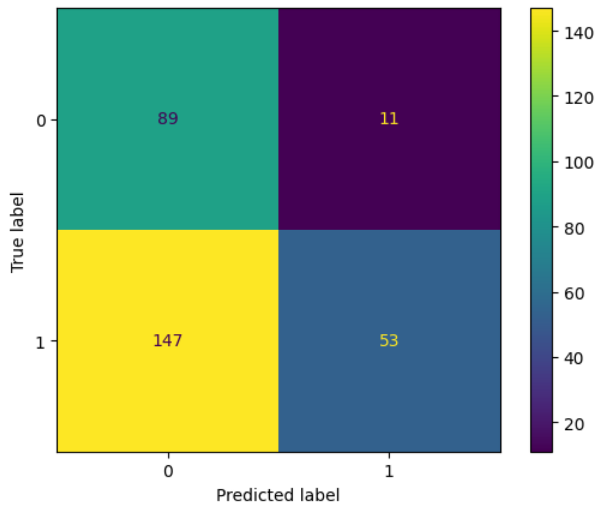
```
[ ]: # Check accuracy
     y_train_pred=model.predict(xs_train) >= 0.5
     y_test_pred=model.predict(xs_test_small) >= 0.5

     tr_acc = accuracy_score(y_train_pred, y_train)
     test_acc = accuracy_score(y_test_pred, y_test_small)
```

```
7/7 [==============================] - 3s 369ms/step
10/10 [==============================] - 4s 391ms/step
```

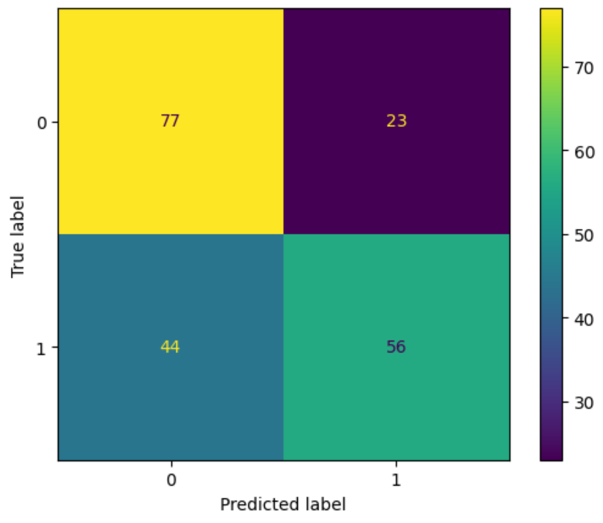
```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.665
Test accuracy:  0.47333333333333333
```

```
[ ]: cm = confusion_matrix(y_test_small, y_test_pred)

     cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: cm = confusion_matrix(y_train, y_train_pred)

     cm_display = ConfusionMatrixDisplay(cm).plot()
```



Normalization doesn't improve the original

2.3.9 PCA (32) with amplitude encoding, Twolocal variational form 5 REPS, 5 qubits, lightning.qubit, adjoint differentiation method

```
[ ]: nqubits=5
     dev=qml.device("lightning.qubit", wires=nqubits)

     nreps=5

     def qnn_circuit(inputs, theta):
       qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
      ↪normalize=True)
       TwoLocal(nqubits=nqubits, theta=theta, reps=nreps)
       return qml.expval(qml.Hermitian(M, wires=[0]))
```

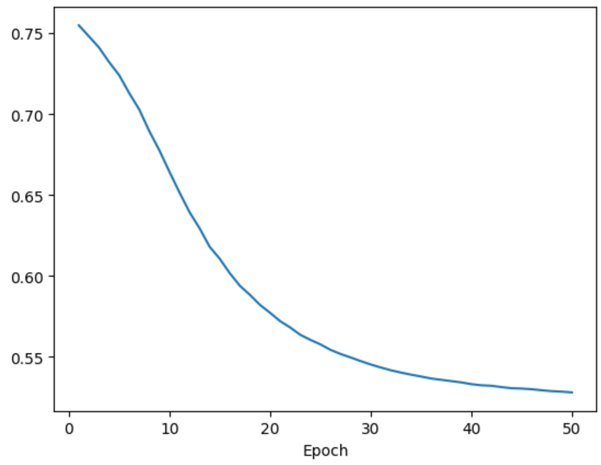
```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

     nweights = 3*nreps*nqubits

     weights={"theta": nweights}

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

     # keras model
     model = tf.keras.models.Sequential([qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                         validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 72s 8s/step - loss: 0.7548
Epoch 2/50
10/10 [==============================] - 13s 1s/step - loss: 0.7480
.
.
.
Epoch 49/50
10/10 [==============================] - 13s 1s/step - loss: 0.5286
```

```
Epoch 50/50
10/10 [==============================] - 13s 1s/step - loss: 0.5281
```

```
[ ]: plot_losses(history)
```



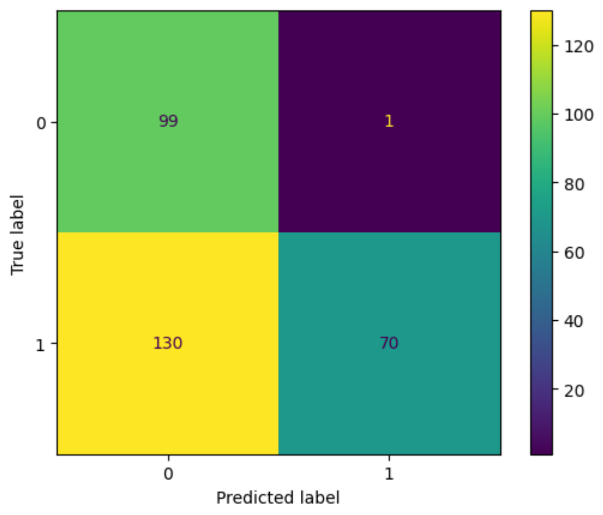
```
[ ]: # Check accuracy
     y_train_pred=model.predict(xs_train) >= 0.5
     y_test_pred=model.predict(xs_test_small) >= 0.5

     tr_acc = accuracy_score(y_train_pred, y_train)
     test_acc = accuracy_score(y_test_pred, y_test_small)
```

```
7/7 [==============================] - 9s 1s/step
10/10 [==============================] - 14s 1s/step
```

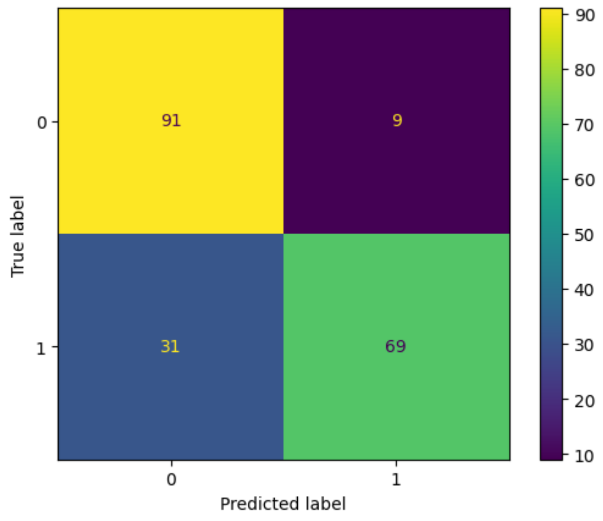
```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.8
Test accuracy:  0.5633333333333334
```

```
[ ]: cm = confusion_matrix(y_test_small, y_test_pred)

     cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: cm = confusion_matrix(y_train, y_train_pred)

     cm_display = ConfusionMatrixDisplay(cm).plot()
```



Increase in the number of repetitions produces an increase in the accuracy of the model. Let's try with more repetitions

2.3.10 PCA (32) with amplitude encoding, Twolocal variational form 10 REPS, 5 qubits, lightning.qubit, adjoint differentiation method

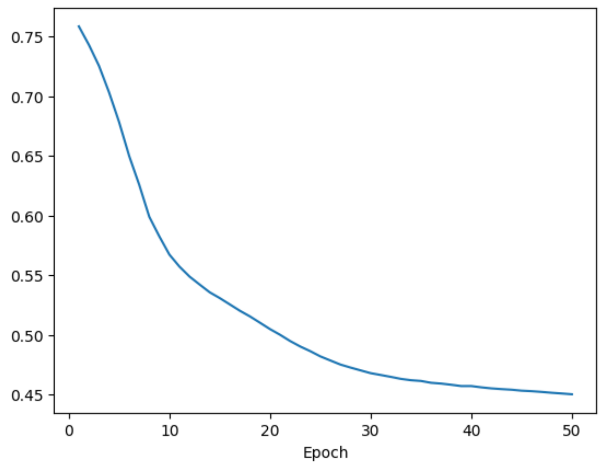
```
[ ]: nqubits=5
     dev=qml.device("lightning.qubit", wires=nqubits)

     nreps=10

     def qnn_circuit(inputs, theta):
       qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
      ↪normalize=True)
       TwoLocal(nqubits=nqubits, theta=theta, reps=nreps)
       return qml.expval(qml.Hermitian(M, wires=[0]))
```

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

     nweights = 3*nreps*nqubits

     weights={"theta": nweights}

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

     # keras model
     model = tf.keras.models.Sequential([qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                         validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 18s 2s/step - loss: 0.7584
Epoch 2/50
10/10 [==============================] - 18s 2s/step - loss: 0.7429
.
.
.
Epoch 49/50
10/10 [==============================] - 18s 2s/step - loss: 0.4507
Epoch 50/50
10/10 [==============================] - 18s 2s/step - loss: 0.4501
```

```
[ ]: plot_losses(history)
```



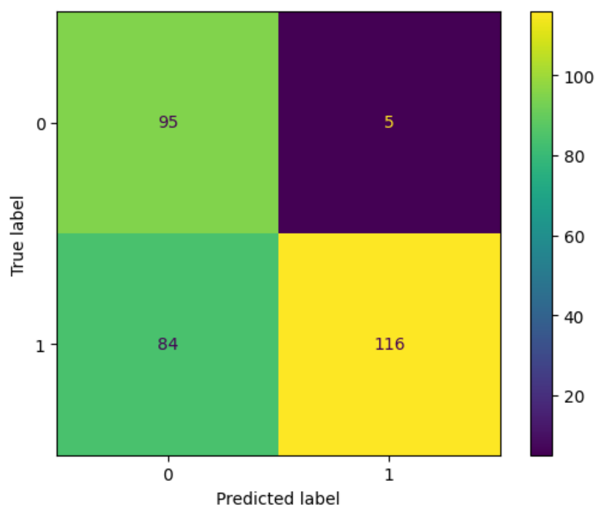
```
[ ]: # Check accuracy
     y_train_pred=model.predict(xs_train) >= 0.5
     y_test_pred=model.predict(xs_test_small) >= 0.5

     tr_acc = accuracy_score(y_train_pred, y_train)
     test_acc = accuracy_score(y_test_pred, y_test_small)
```

```
7/7 [==============================] - 6s 846ms/step
10/10 [==============================] - 13s 1s/step
```

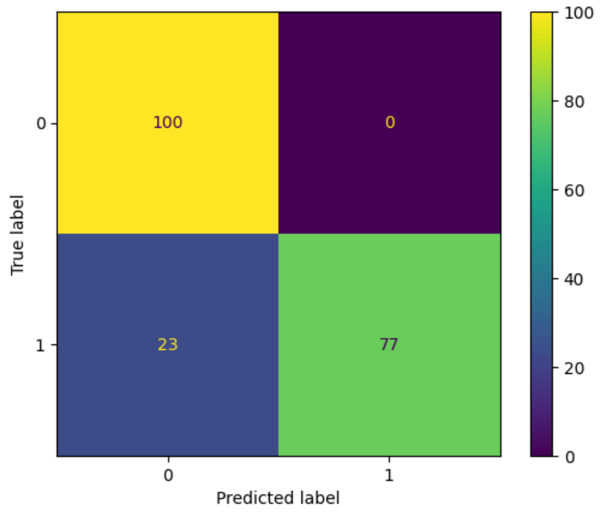
```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.885
Test accuracy:  0.7033333333333334
```

```
[ ]: cm = confusion_matrix(y_test_small, y_test_pred)

     cm_display = ConfusionMatrixDisplay(cm).plot()
```



[ ]:
```
cm = confusion_matrix(y_train, y_train_pred)

cm_display = ConfusionMatrixDisplay(cm).plot()
```



[ ]:
```
test_size=100
pred_split=np.array_split(y_test_pred,3)
y_sep_pred=pred_split[0]
y_ppt_pred=pred_split[1]
y_nppt_pred=pred_split[2]
```

```
score_sep = accuracy_score(y_sep_pred, np.full(test_size, 0))
score_ppt= accuracy_score(y_ppt_pred, np.full(test_size,1))
score_nppt= accuracy_score(y_nppt_pred, np.full(test_size,1))

print("SEP accuracy: ", score_sep)
print("PPT accuracy: ", score_ppt)
print("NPPT accuracy: ", score_nppt)
```

```
SEP accuracy:  0.95
PPT accuracy:  0.57
NPPT accuracy:  0.59
```

GREAT IMPROVEMENT!!! Increasing the number of repetitions of TwoLocal variational form produces better accuracy. In this case with 10 repetitions we obtain 70% of accuracy on the test set, the highest so far for QNN.

Observing the confussion matrix of the test set, there is almost a 1:1 proportion of true possitives to false negatives. Remembering that 1 represents entanglement and those samples come from PPT-ENT and NPPT-ENT we can investigate if that amount of false negatives comes from our model making errors when predicting one of the types of entanglement

2.3.11 PCA (32) with amplitude encoding, StrongEntanglingLayers 8 reps, lightning.qubit, 5 qubits, adjoint diff method

```
[ ]: # Hermitian matrix
     state_0 = [[1], [0]]
     M = state_0 * np.conj(state_0).T
```

```
[ ]: pca = PCA(n_components = 32)

     xs_train = pca.fit_transform(x_train_norm)
     xs_test_small = pca.transform(x_test_norm)
```

```
[ ]: nqubits=5
     dev = qml.device("lightning.qubit", wires=nqubits)

     # number of repetitions that we want in each instance of the variational form
     nreps = 8
     #  dimensions of the input that the variational form expects
     weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
     #  number of inputs that each instance of the variational form will take
     nweights = 3*nreps*nqubits

     def qnn_circuit_strong(inputs, theta):
       qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
      ↪normalize=True)
       # reshape the theta array of parameters to make it fit into the shape that
       # the variational form expects
       theta1 = tf.reshape(theta, weights_dim)
       qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))
```

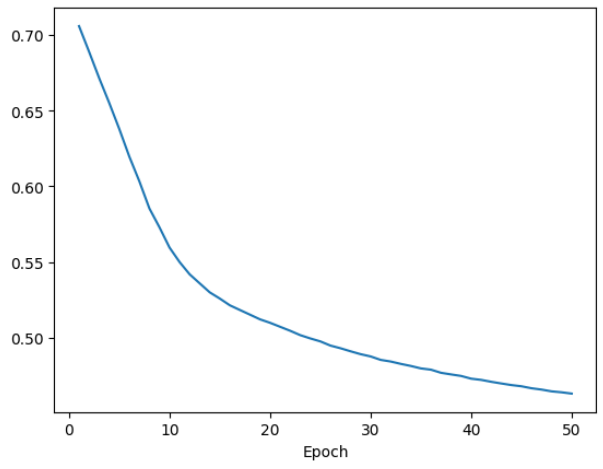
```
  return qml.expval(qml.Hermitian(M, wires = [0]))

# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

     # keras model
     model = tf.keras.models.Sequential([qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model.fit(xs_train, y_train, epochs = 50, shuffle = True,
                         validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 39s 4s/step - loss: 0.7059
Epoch 2/50
10/10 [==============================] - 33s 3s/step - loss: 0.6890
.
.
.
Epoch 49/50
10/10 [==============================] - 32s 3s/step - loss: 0.4639
Epoch 50/50
10/10 [==============================] - 32s 3s/step - loss: 0.4631
```

```
[ ]: plot_losses(history)
```



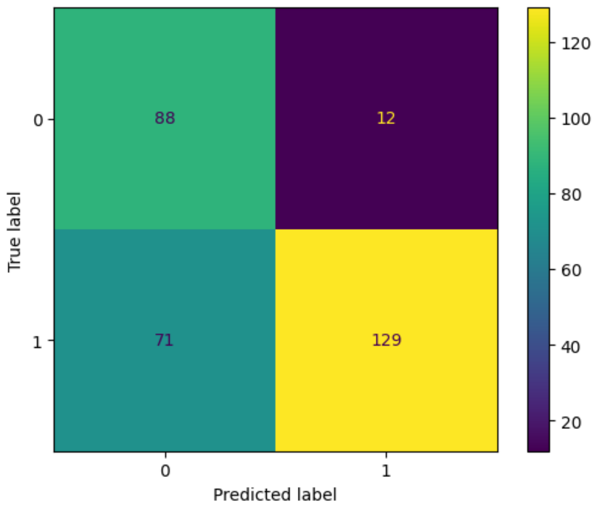
```
[ ]: # Check accuracy
     y_train_pred=model.predict(xs_train) >= 0.5
     y_test_pred=model.predict(xs_test_small) >= 0.5

     tr_acc = accuracy_score(y_train_pred, y_train)
     test_acc = accuracy_score(y_test_pred, y_test_small)
```

```
7/7 [==============================] - 24s 3s/step
10/10 [==============================] - 20s 2s/step
```

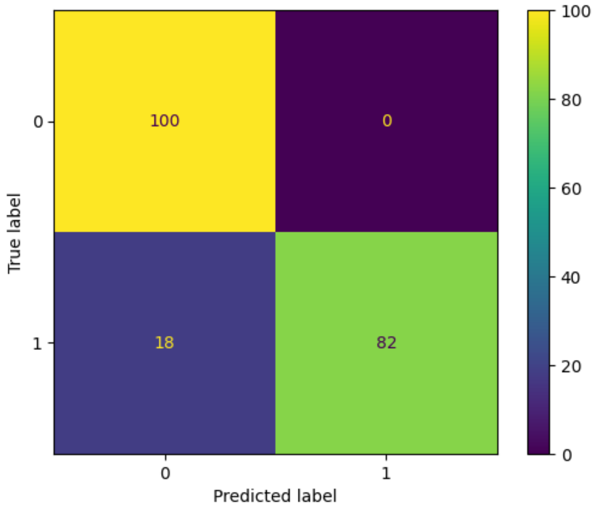
```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.91
Test accuracy:  0.7233333333333334
```

```
[ ]: cm = confusion_matrix(y_test_small, y_test_pred)

     cm_display = ConfusionMatrixDisplay(cm).plot()
```



[ ]:
```
cm = confusion_matrix(y_train, y_train_pred)

cm_display = ConfusionMatrixDisplay(cm).plot()
```



Improvement with more repetitions (8) of strong entangling layers with normalized data PCA 32 attributes Training 28 min

[ ]:
```
test_size=100
pred_split=np.array_split(y_test_pred,3)
y_sep_pred=pred_split[0]
```

```
y_ppt_pred=pred_split[1]
y_nppt_pred=pred_split[2]

score_sep = accuracy_score(y_sep_pred, np.full(test_size, 0))
score_ppt= accuracy_score(y_ppt_pred, np.full(test_size,1))
score_nppt= accuracy_score(y_nppt_pred, np.full(test_size,1))

print("SEP accuracy: ", score_sep)
print("PPT accuracy: ", score_ppt)
print("NPPT accuracy: ", score_nppt)
```

```
SEP accuracy:  0.88
PPT accuracy:  0.69
NPPT accuracy:  0.6
```

3 Case study: dataset with 0.5 PPT ratio

Replicating the working conditions from paper LARGE-SCALE QUANTUM SEPARABILITY THROUGH A REPRODUCIBLE MACHINE LEARNING LENS, we will work with - A training set of 200 samples: 100 separable and 100 entangled (with 0.5 ppt ratio, thus 50 samples ppt-ent and 50 samples nppt-ent). File *train_ set_ 05.csv* - A test set of 300 samples, 100 separable, 100 ppt-entangled and 100 nppt-entangled. It is a reduced version of the papers test set consisting in 1000 samples per type. File *test_ set_ small.csv*

Additionally, we will use 5 fold cross validation for the training process of the models.

```
[ ]: training_data = pd.read_csv("/content/drive/MyDrive/tfg/x_train_05.csv", header=None)
     training_data.head()
```

```
[ ]:          0         1         2         3         4         5         6  ...
     0  0.139035  0.008456 -0.082087 -0.019300  0.009335  0.050326  0.010913
     1  0.027434 -0.024593 -0.004040 -0.009824  0.003811  0.012059  0.013763
     2  0.009734  0.003539  0.016788  0.005305  0.014389  0.013836  0.011906
     3  0.007171  0.000200  0.006113 -0.011893 -0.008216  0.047268 -0.002918
     4  0.016754  0.088911  0.170843  0.007160  0.037591 -0.159721 -0.014413


              74        75        76        77        78        79
     0  0.061386  0.077893  0.107214  0.120851  0.119377  0.126888
     1  0.045268  0.065257  0.075945  0.086621  0.088141  0.100372
     2  0.017251  0.065321  0.076891  0.090936  0.113257  0.113306
     3  0.043498  0.080326  0.100310  0.101018  0.113511  0.120475
     4  0.052332  0.105790  0.112462  0.076749  0.107003  0.119324

     [5 rows x 80 columns]
```

```
[ ]: training_data.describe()
```

```
[ ]:                 0           1           2           3           4           5  ...
     count  200.000000  200.000000  200.000000  200.000000  200.000000  200.000000
     mean     0.003763   -0.002582   -0.001872    0.003203   -0.002515   -0.001205
     std      0.035739    0.033113    0.033377    0.026297    0.026577    0.035588
```

```
min     -0.170321   -0.166935   -0.136005   -0.078350   -0.080244   -0.159721
25%     -0.016411   -0.022544   -0.016625   -0.007574   -0.014764   -0.015894
50%      0.001116    0.000261    0.000438    0.001154   -0.000650   -0.002426
75%      0.022831    0.013867    0.016098    0.011541    0.007547    0.017720
max      0.139035    0.096240    0.170843    0.102711    0.077133    0.117559


               77          78          79
count  200.000000  200.000000  200.000000
mean     0.097140    0.105141    0.111583
std      0.012903    0.011354    0.008890
min      0.064312    0.083020    0.092286
25%      0.088503    0.098364    0.105646
50%      0.096861    0.104572    0.111667
75%      0.104330    0.111644    0.116109
max      0.162879    0.162430    0.152799

[8 rows x 80 columns]
```

```
[ ]: x_train = np.genfromtxt("/content/drive/MyDrive/tfg/x_train_05.csv",␣
      ↪delimiter=",",dtype=None)
     y_train = np.genfromtxt("/content/drive/MyDrive/tfg/y_train_05.csv",␣
      ↪delimiter=",",dtype=None)

     x_test_small = np.genfromtxt("/content/drive/MyDrive/tfg/x_test_small.csv",␣
      ↪delimiter=",",dtype=None)
     y_test_small = np.genfromtxt("/content/drive/MyDrive/tfg/y_test_small.csv",␣
      ↪delimiter=",",dtype=None)
```

```
[ ]: print("Training set:")
     print("Size of the training set ", x_train.shape)
     print("Size of the corresponding labels ", y_train.shape)

     print("\nTest set:")
     print("Size of the test set ", x_test_small.shape)
     print("Size of the corresponding labels ", y_test_small.shape)
```

```
Training set:
Size of the training set  (200, 80)
Size of the corresponding labels  (200,)

Test set:
Size of the test set  (300, 80)
Size of the corresponding labels  (300,)
```

3.1 Exploratory data analysis

We will use Principal Component Analysis to reduce the dimensionality of the dataset. Since we will use quantum machine learning techniques to build a classifier on this data, our first step in that process will be data embedding. There are different data encoding built-in functions in Pennylane, however, the only suitable one for our case is Amplitude embedding due to the constraint in the number of qubits we can use in the current simulators. Amplitude encoding allows us to codify $N \leq 2^n$ attributes with n qubits. Our dataset has 80 attributes, with 7 qubits amplitude encoding could codify up to 128 attributes. In the initial testing, we have obtained best performance reducing the dimensionality of the dataset to 32 attributes using principal component analysis. This allows us to reduce to 5 qubits.

We will also check the effect of normalization on our dataset.

We will use the class *sklearn.decomposition.PCA*. First we will work with direclty with our data (no normalization)

```
[ ]: pca = PCA(n_components = 32)

     xs_train = pca.fit_transform(x_train)
     xs_test = pca.transform(x_test_small)
```

Once trained, the PCA object contains all the information about the components. We fixed the number of components to 32. Next, we will analyse the three main components.

These three rows correspond to the three principal axes in feature space, representing the directions of maximum variance in the data. The components are sorted by decreasing *explained_variance_* (the amount of variance explained by each of the selected components)

```
[ ]: print(pca.components_.shape)
```

```
(32, 80)
```

```
[ ]: three_pc = pd.DataFrame(
         data=pca.components_[0:3],
         index = ['PC1', 'PC2', 'PC3']
     )
```

```
[ ]: three_pc
```

```
[ ]:            0         1         2         3         4         5         6  ...
     PC1 -0.078923  0.138744  0.347397 -0.082156 -0.010084 -0.157959  0.018403
     PC2 -0.046127  0.056054 -0.006340  0.038842  0.018734  0.008609 -0.047358
     PC3 -0.261291 -0.176917  0.008989  0.012620 -0.057868  0.357191  0.029919


                73        74        75        76        77        78        79
     PC1 -0.052625 -0.025488  0.013009 -0.000893 -0.013329 -0.002820 -0.008094
     PC2 -0.040785 -0.016024  0.008447 -0.013036 -0.008851  0.007211 -0.004051
     PC3  0.003536 -0.072748 -0.062288 -0.055809 -0.027001 -0.009286 -0.014112

     [3 rows x 80 columns]
```

```
[ ]: print('Porcentaje de varianza explicada por cada componente')
     print(pca.explained_variance_ratio_)
```

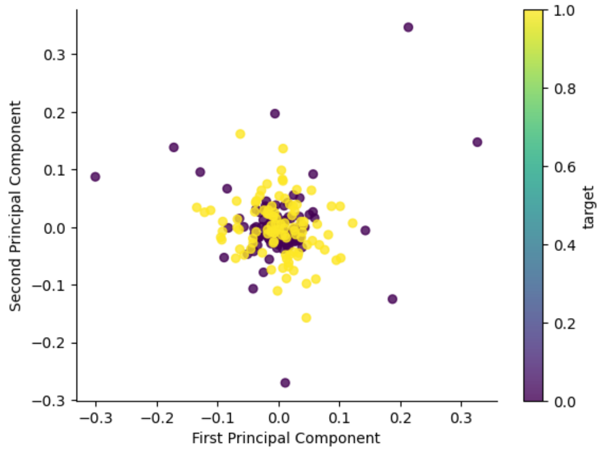
```
Porcentaje de varianza explicada por cada componente
[0.05263381 0.0494832  0.04331597 0.03946737 0.03424476 0.03411636
 0.03268133 0.02931714 0.02839679 0.02712453 0.02636718 0.02459969
 0.02388274 0.02320542 0.02227055 0.02117816 0.02100179 0.02023511
 0.01923234 0.01888048 0.01842044 0.01773041 0.01685939 0.0162822
 0.01530907 0.01474871 0.01440209 0.01387777 0.01355994 0.01286114
 0.01243935 0.01218028]
```

[ ]:
```
pca_df = pd.DataFrame(
    data = xs_train[:,0:3],
    columns = ['PC1', 'PC2', 'PC3']
)
pca_df = pd.concat([pca_df, pd.DataFrame(y_train, columns =['target'])[['target']]],␣
 ↪axis=1)
```

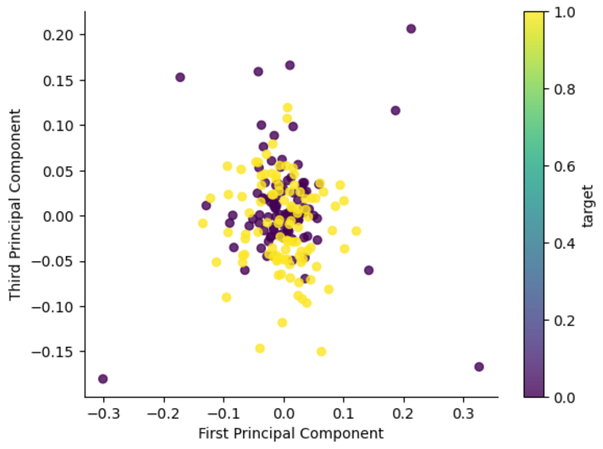
[ ]:
```
pca_df
```

[ ]:
```
          PC1       PC2       PC3  target
0   -0.089289 -0.052919 -0.008266     0.0
1   -0.016367 -0.006689 -0.012467     0.0
2    0.014952  0.042296 -0.025777     0.0
3   -0.031557 -0.017658  0.018171     0.0
4    0.326876  0.147360 -0.167084     0.0
..        ...       ...       ...     ...
195  0.032934 -0.044987 -0.048340     1.0
196  0.030219  0.039094 -0.039554     1.0
197  0.067205 -0.030824  0.026076     1.0
198  0.011818 -0.048264 -0.068982     1.0
199 -0.041580 -0.027801  0.058748     1.0

[200 rows x 4 columns]
```

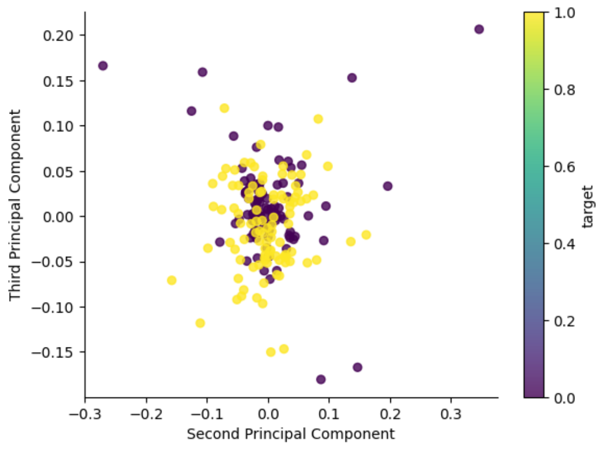
[ ]:
```
pca_df.plot(kind='scatter', x='PC1', y='PC2', c='target', s=32, alpha=.8)
plt.xlabel('First Principal Component')

plt.ylabel('Second Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```



```
[ ]: pca_df.plot(kind='scatter', x='PC1', y='PC3', c='target', s=32, alpha=.8)
     plt.xlabel('First Principal Component')

     plt.ylabel('Third Principal Component')

     plt.gca().spines[['top', 'right',]].set_visible(False)
```

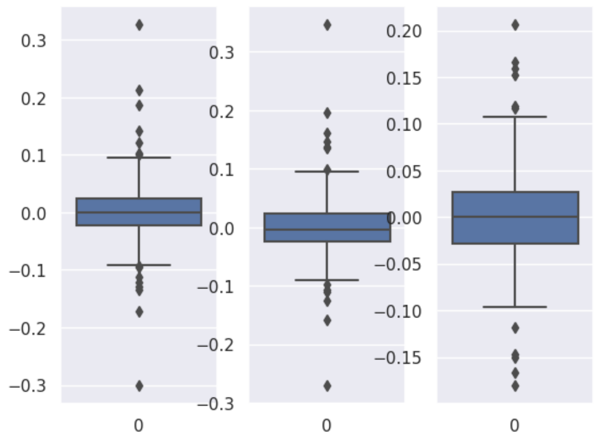


```
[ ]: pca_df.plot(kind='scatter', x='PC2', y='PC3', c='target', s=32, alpha=.8)
     plt.xlabel('Second Principal Component')

     plt.ylabel('Third Principal Component')

     plt.gca().spines[['top', 'right',]].set_visible(False)
```



```
[ ]: # Box Plot

     #set seaborn plotting aesthetics as default
     sns.set()

     #define plotting region (2 rows, 2 columns)
     fig, axes = plt.subplots(1, 3)

     #create boxplot in each subplot
     sns.boxplot(data=pca_df['PC1'], ax=axes[0])
     sns.boxplot(data=pca_df['PC2'], ax=axes[1])
     sns.boxplot(data=pca_df['PC3'], ax=axes[2])
```

```
[ ]: <Axes: >
```



3.1.1 MaxAbsScaler normalization

```
[ ]: scaler = MaxAbsScaler()
     x_train_norm = scaler.fit_transform(x_train)

     x_test_norm = scaler.transform(x_test_small)

     # Restrict all the values to be between 0 and 1
     x_test_norm = np.clip(x_test_norm,0,1)
```

```
[ ]: pca = PCA(n_components = 32)

     xs_train_norm = pca.fit_transform(x_train_norm)
     xs_test_norm = pca.transform(x_test_norm)
```

```
[ ]: three_pc_norm = pd.DataFrame(
         data=pca.components_[0:3],
         index = ['PC1', 'PC2', 'PC3']
     )
```

```
[ ]: three_pc_norm
```

```
[ ]:           0         1         2         3         4         5         6   ...
PC1  0.024188  0.056195  0.037531  0.013884 -0.109444 -0.014466 -0.072613
PC2  0.000420  0.083452  0.119313 -0.104092  0.167000 -0.124731  0.058960
PC3 -0.123593 -0.040699 -0.073086  0.014512  0.088001  0.059851 -0.071889


           73        74        75        76        77        78        79
PC1 -0.085012 -0.015312  0.021536 -0.008675 -0.020402 -0.003547 -0.002765
PC2 -0.077326 -0.000914  0.024644  0.020783  0.008202  0.004571 -0.003491
PC3  0.005548 -0.023724 -0.019626 -0.016679  0.010794  0.012590 -0.003202

[3 rows x 80 columns]
```

```
[ ]: print('Porcentaje de varianza explicada por cada componente')
     print(pca.explained_variance_ratio_)
```

```
Porcentaje de varianza explicada por cada componente
[0.04667584 0.04370644 0.038514   0.03810517 0.03644829 0.03399039
 0.03291459 0.03232812 0.03079872 0.02836721 0.02774176 0.02559811
 0.02475009 0.02444303 0.02398773 0.02280793 0.02158439 0.02110549
 0.02043725 0.0198008  0.01908729 0.0179154  0.01718435 0.01674891
 0.01561015 0.01516731 0.01468464 0.01395486 0.01385951 0.01316067
 0.01292767 0.01250232]
```

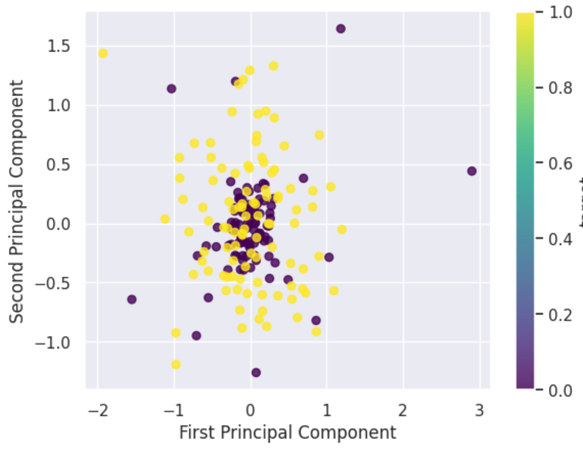
```
[ ]: pca_norm_df = pd.DataFrame(
         data = xs_train_norm[:,0:3],
         columns = ['PC1', 'PC2', 'PC3']
     )
     pca_norm_df = pd.concat([pca_norm_df, pd.DataFrame(y_train, columns␣
      ↪=['target'])[['target']]], axis=1)
```

```
[ ]: pca_norm_df
```

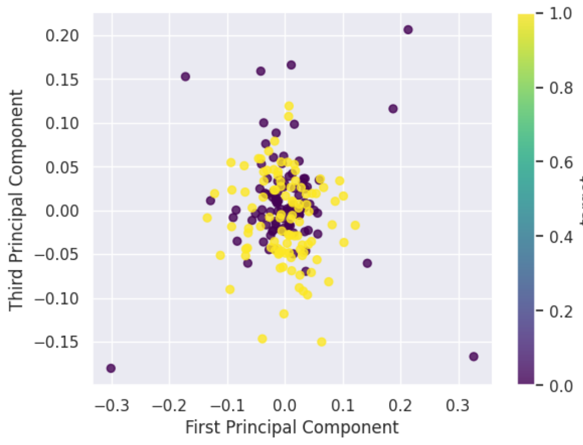
```
[ ]:          PC1       PC2       PC3  target
0   -0.128141 -0.391311 -0.041418     0.0
1   -0.043548  0.094400  0.170526     0.0
2    0.180909  0.189010  0.086033     0.0
3   -0.198316 -0.271490 -0.151484     0.0
4    1.186498  1.642871 -0.648775     0.0
..        ...       ...       ...     ...
195 -0.028142  0.488250 -0.709229     1.0
196  0.580187 -0.000676 -0.809034     1.0
197  0.176704  0.513842 -0.930971     1.0
198 -0.520711  0.679900 -0.256616     1.0
199 -0.629642 -0.320575 -0.166607     1.0
```

```
[200 rows x 4 columns]
```

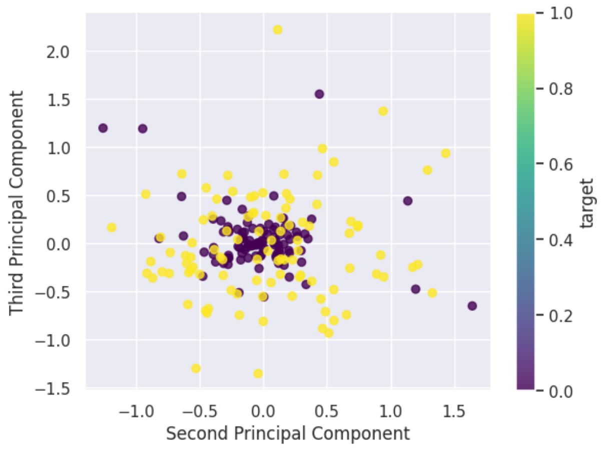
```
[ ]: pca_norm_df.plot(kind='scatter', x='PC1', y='PC2', c='target', s=32, alpha=.8,␣
      ↪cmap='viridis')
     plt.xlabel('First Principal Component')

     plt.ylabel('Second Principal Component')

     plt.gca().spines[['top', 'right',]].set_visible(False)
```



```
[ ]: pca_df.plot(kind='scatter', x='PC1', y='PC3', c='target', s=32, alpha=.8,␣
      ↪cmap='viridis')
     plt.xlabel('First Principal Component')

     plt.ylabel('Third Principal Component')

     plt.gca().spines[['top', 'right',]].set_visible(False)
```

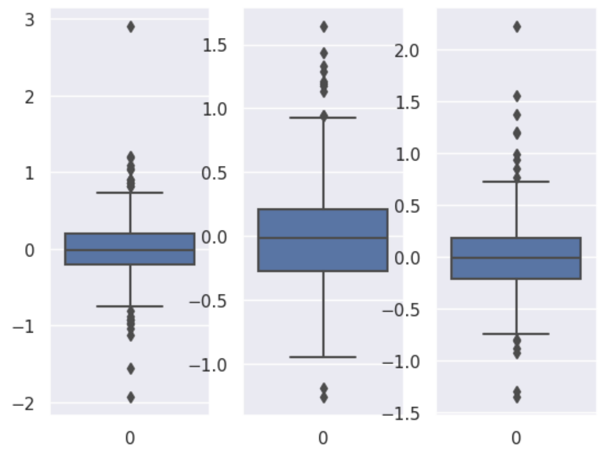


```
[ ]: pca_norm_df.plot(kind='scatter', x='PC2', y='PC3', c='target', s=32, alpha=.8,␣
      ↪cmap='viridis')
     plt.xlabel('Second Principal Component')

     plt.ylabel('Third Principal Component')

     plt.gca().spines[['top', 'right',]].set_visible(False)
```



```
[ ]: # Box Plot

     #set seaborn plotting aesthetics as default
     sns.set()

     #define plotting region (2 rows, 2 columns)
     fig, axes = plt.subplots(1, 3)

     #create boxplot in each subplot
     sns.boxplot(data=pca_norm_df['PC1'], ax=axes[0])
     sns.boxplot(data=pca_norm_df['PC2'], ax=axes[1])
     sns.boxplot(data=pca_norm_df['PC3'], ax=axes[2])
```

```
[ ]: <Axes: >
```



3.2 Quantum Support Vector Machines

```
[ ]: def detailed_accuracy(y_pred, size):
       test_size=size
       pred_split=np.array_split(y_test_pred,3)
       y_sep_pred=pred_split[0]
       y_ppt_pred=pred_split[1]
       y_nppt_pred=pred_split[2]

       score_sep = accuracy_score(y_sep_pred, np.full(test_size, 0))
       score_ppt= accuracy_score(y_ppt_pred, np.full(test_size,1))
       score_nppt= accuracy_score(y_nppt_pred, np.full(test_size,1))

       print("SEP accuracy: ", score_sep)
       print("PPT accuracy: ", score_ppt)
       print("NPPT accuracy: ", score_nppt)
```

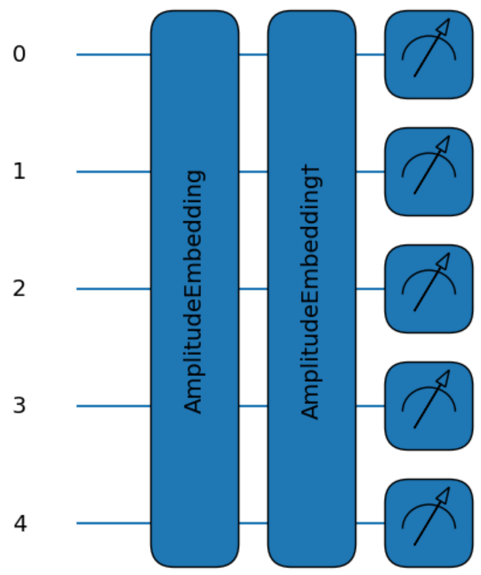
```
[ ]: def save_model(model, filename):
       dir="/content/drive/MyDrive/tfg/"+filename
       joblib.dump(model, dir)
       print("Model saved")
```

```
[ ]: def load_model(model, filename):
       dir="/content/drive/MyDrive/tfg/"+filename
       return joblib.load(dir)
```

```
[ ]: # Amplitude encoding of 64 variables using 5 qubits (can encode up to 32 inputs)

     # Number of qubits of the system
     nqubits = 5
     # We define a device
     dev = qml.device("lightning.qubit", wires = nqubits)
```

```
# We define de circuit of our kernel. We use AmplitudeEmbedding which returns an
# operation equivalent to amplitude encoding of the first argument
@qml.qnode(dev)
def kernel_circ(a,b):
    qml.AmplitudeEmbedding(a, wires=range(nqubits), pad_with=0, normalize=True)
    # Computes the adjoint (or inverse) of the amplitude encoding of b
    qml.adjoint(qml.AmplitudeEmbedding(b, wires=range(nqubits), pad_with=0,␣
 ↪normalize=True))     # We return an array with the probabilities fo measuring each␣
 ↪possible state in the
    # computational basis
    return qml.probs(wires=range(nqubits))
```

```
[ ]: fig, ax = qml.draw_mpl(kernel_circ)([1],[1])
     fig.show()
```



```
[ ]: def qkernel(A, B):
       return np.array([[kernel_circ(a,b)[0] for b in B] for a in A])
```

Now, we will activate GPU acceleration and check it

```
[ ]: gpu_info = !nvidia-smi
     gpu_info = '\n'.join(gpu_info)
     if gpu_info.find('failed') >= 0:
```

```
  print('Not connected to a GPU')
else:
  print(gpu_info)
```

```
Mon Nov 13 14:59:13 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.105.17   Driver Version: 525.105.17   CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   47C    P8    10W /  70W |      0MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

```
[ ]: svm = SVC(kernel = qkernel).fit(xs_train, y_train)
```

```
[ ]: y_train_pred=svm.predict(xs_train)
```

```
[ ]: tr_acc=accuracy_score(y_train_pred, y_train)
     print("Train accuracy: ", tr_acc)
```

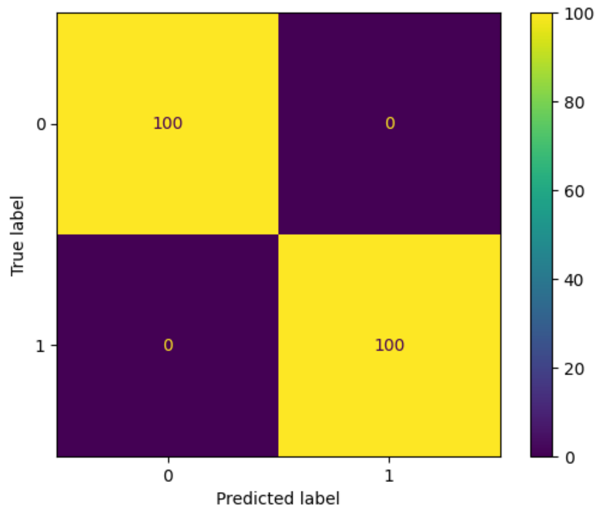
```
Train accuracy:  1.0
```

```
[ ]: y_test_pred=svm.predict(xs_test)
```

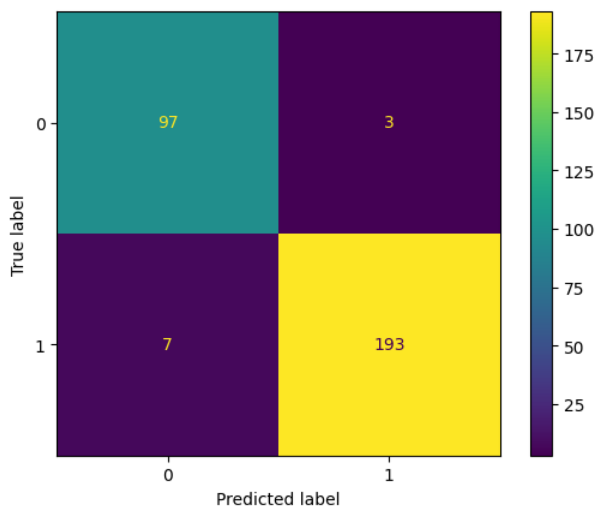
```
[ ]: test_acc=accuracy_score(y_test_pred, y_test_small)
     print("Test accuracy: ", test_acc)
```

```
Test accuracy:  0.9666666666666667
```

```
[ ]: # Train confussion matrix
     cm = confusion_matrix(y_train, y_train_pred)
     cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: # Test confussion matrix
     cm = confusion_matrix(y_test_small, y_test_pred)
     cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: # Test accuracy per type
     detailed_accuracy(y_test_pred, 100)
```

```
SEP accuracy:  0.97
PPT accuracy:  0.95
NPPT accuracy:  0.98
```

```
[ ]: from sklearn.model_selection import GridSearchCV, StratifiedKFold
```

```
[ ]: model_type = SVC()
     model_params = [{'kernel': [qkernel]}]
     model = GridSearchCV(model_type, model_params, cv=StratifiedKFold(shuffle=True)).
      ↪fit(xs_train, y_train)
     print('Training results :')
     print(model.best_params_)
     print(model.best_score_)
```

```
Training results :
{'kernel': <function qkernel at 0x788877ed0dc0>}
0.9650000000000001
```

Using a Support Vector Machine with a quantum kernel induced by Amplitude encoding with - 5 qubits - 32 attributes We have run with GPU acceleration the training in 12 minutes and predictions on the training set in 13 minutes. We have obtained an accuracy of 1.0 on the training set

Then, we have performed the predictions on the test set, which took 19 minutes. We have obtained an accuracy of 0.9667 on the test set.

With the predictions of the test set, we have also studied the accuracy on the predictions of each type of data: Separable, PPT-entangled and NPPT-entangled. - Accuracy of 0.97 on the separable data samples - Accuracy of 0.95 on the PPT entangled data samples - Accuracy of 0.98 on the NPPT entangled data samples.

Therefore, we can conclude that this QSVM performs great and is able to correctly clasify separable and entangled data, specially PPT entangled data (which is the "hardest" to distinguish from separable) with a great accuracy.

Above we can see the confussion matrices for classification of the training set and the test set.

Finally, we have run trained this model using GridSearchCV (just with this model exact configuration) with five fold cross validation using. This took more than 50 minutes to execute. We obtained 0.965 as best accuracy. We need to discard using GridSeachCV to perform an exhaustive search over differnet parameter values for a model due to the long executing time obtained just for one configuration and because the different configurations of a QSVM are not externally parametrised, but defined in the quantum circuit

Next we will train and predict this same model configuration on our normalized dataset to check if there is any difference in the accuracy

```
[ ]: svm = SVC(kernel = qkernel).fit(xs_train_norm, y_train)
```

```
[ ]: y_train_pred=svm.predict(xs_train_norm)
```

```
[ ]: tr_acc=accuracy_score(y_train_pred, y_train)
     print("Train accuracy: ", tr_acc)
```

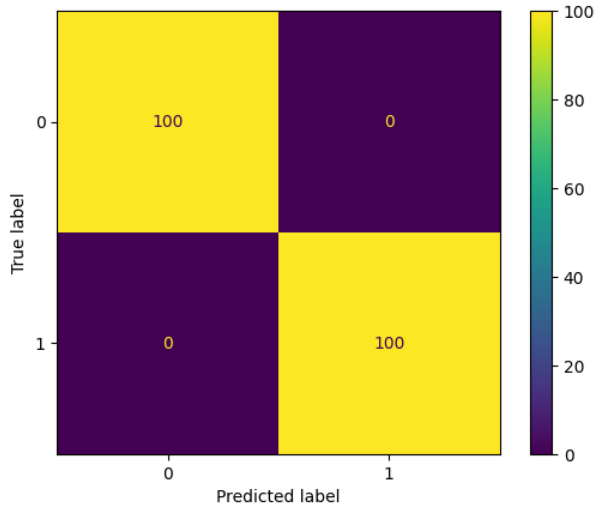
```
Train accuracy:  1.0
```

```
[ ]: y_test_pred=svm.predict(xs_test_norm)
```

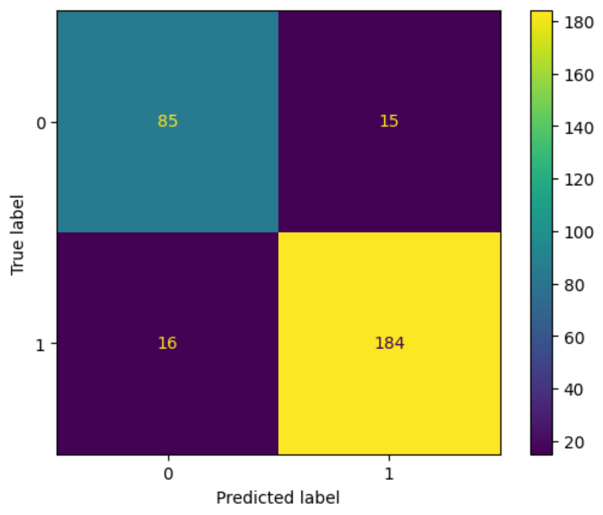
```
[ ]: test_acc=accuracy_score(y_test_pred, y_test_small)
     print("Test accuracy: ", test_acc)
```

```
Test accuracy:  0.8966666666666666
```

```
[ ]: # Train confussion matrix
     cm = confusion_matrix(y_train, y_train_pred)
     cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: # Test confussion matrix
     cm = confusion_matrix(y_test_small, y_test_pred)
     cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: # Test accuracy per type
     detailed_accuracy(y_test_pred, 100)
```

```
SEP accuracy:  0.85
PPT accuracy:  0.94
NPPT accuracy:  0.9
```

The use of normalized data (with MaxAbsScaler) does not improve the model neither the execution time. Indeed, the accuracy worsens from 0.9667 to 0.8967. From now on, we will continue working only with the not normalized dataset

3.3 3. Quantum Neural Networks

We will try different configurations of neural networks based on the promising results obtained in the first phase of trials. However, we cannot use GridSeachCV to select the best model because the parameters and changes that we apply in each configuration are performed in the quantum neural network circuit definition. We have created the function *fivefoldCV_qnn(circuit)* to perform five fold cross validation for every QNN circuit that we will test. The five fold cross validation code is inspired by https://github.com/christianversloot/machine-learning-articles/blob/main/how-to-use-k-fold-cross-validation-with-keras.md

```
[ ]: # We set a seed for the packages so the results are reproducible
     seed=4321
     np.random.seed(seed)
     tf.random.set_seed(seed)
```

```
[ ]: # pennylane works with doubles and tensorflow works with floats.
     # We ask tensorflow to work with doubles

     tf.keras.backend.set_floatx('float64')
```

```
[ ]: def fivefoldCV_qnn(circuit, x, y):
       # Define the K-fold Cross Validator
       kfold = KFold(n_splits=5, shuffle=True)
       acc_per_fold = []
       loss_per_fold = []

       # K-fold Cross Validation model evaluation
       fold_no = 1
       for train, test in kfold.split(x, y):

         # Define the model architecture
         method= "adjoint"
         tf.random.set_seed(seed)
         qnn = qml.QNode(circuit, dev, interface="tf", diff_method=method)
         nweights = 3*nreps*nqubits
         weights={"theta": nweights}
         # Keras layer containing qnn
         qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)
         # keras model
         model = tf.keras.models.Sequential([qlayer])
```

```
    # we choose adam optimizer with a learning rate of 0.005
    opt = tf.keras.optimizers.Adam(learning_rate=0.005)


    # binary cross entropy loss, because we are training a binary classifier
    model.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())

    # Generate a print
    print('------------------------------------------------------------------------')
    print(f'Training for fold {fold_no} ...')

    # Training our model
    history = model.fit(x[train], y[train], epochs = 50, shuffle = True,
                        validation_data = None, batch_size = 20, verbose=1)

    # Generate generalization metrics
    score = model.evaluate(x[test], y[test], verbose=0)
    # Check accuracy
    y_test_pred=model.predict(x[test]) >= 0.5
    test_acc = accuracy_score(y_test_pred, y[test])

    print(f'Score for fold {fold_no}: {model.metrics_names[0]} of {score}; accuracy of␣
 ↪{test_acc}')
    loss_per_fold.append(score)
    acc_per_fold.append(test_acc)


    # Increase fold number
    fold_no = fold_no + 1

  # == Provide average scores ==
  print('------------------------------------------------------------------------')
  print('Score per fold')
  for i in range(0, len(loss_per_fold)):
    print('------------------------------------------------------------------------')
    print(f'> Fold {i+1} - Loss: {loss_per_fold[i]} - Accuracy: {acc_per_fold[i]}')
  print('------------------------------------------------------------------------')
  print('Average scores for all folds:')
  print(f'> Loss: {np.mean(loss_per_fold)}')
  print(f'> Accuracy: {np.mean(acc_per_fold)} (+- {np.std(acc_per_fold)})')
  print('------------------------------------------------------------------------')
```

```
[ ]: def performance(y_train_pred, y_train, y_test_pred, y_test):
       tr_acc = accuracy_score(y_train_pred, y_train)
       test_acc = accuracy_score(y_test_pred, y_test_small)

       tr_f1 = f1_score(y_train, y_train_pred)
       test_f1 = f1_score(y_test, y_test_pred)

       print("Train accuracy: ", tr_acc)
```

```
  print("Train F-1 score: ", tr_f1)

  print("\nTest accuracy: ", test_acc)
  print("Test F-1 score: ", test_f1)

  # Test accuracy per type
  print("\nTest accuracy broken down per type")
  detailed_accuracy(y_test_pred, 100)

  print("\n")
  # Train confusion matrix
  cm = confusion_matrix(y_train, y_train_pred)
  cm_display = ConfusionMatrixDisplay(cm).plot()
  cm_display.ax_.set_title("Train confusion matrix")

  # Test confusion matrix
  cm = confusion_matrix(y_test, y_test_pred)
  cm_display = ConfusionMatrixDisplay(cm).plot()
  cm_display.ax_.set_title("Test confusion matrix")
```

3.3.1 Twolocal variational form

```
[ ]: # Hermitian matrix
     state_0 = [[1], [0]]
     M = state_0 * np.conj(state_0).T
```

```
[ ]: # Two local variational form
     def TwoLocal(nqubits, theta, reps=1):
       for r in range(reps):
         for i in range(nqubits):
           qml.RY(theta[r*nqubits+i], wires=i)
         for i in range(nqubits-1):
           qml.CNOT(wires=[i,i+1])

       for i in range(nqubits):
         qml.RY(theta[reps*nqubits+i], wires=i)
```

```
[ ]: nqubits=5
     dev=qml.device("lightning.qubit", wires=nqubits)

     nreps=10

     def qnn_circuit(inputs, theta):
       qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
      ↪normalize=True)
       TwoLocal(nqubits=nqubits, theta=theta, reps=nreps)
       return qml.expval(qml.Hermitian(M, wires=[0]))
```

```
[ ]: # Model 1 five fold cross validation
     fivefoldCV_qnn(qnn_circuit,xs_train, y_train)
```

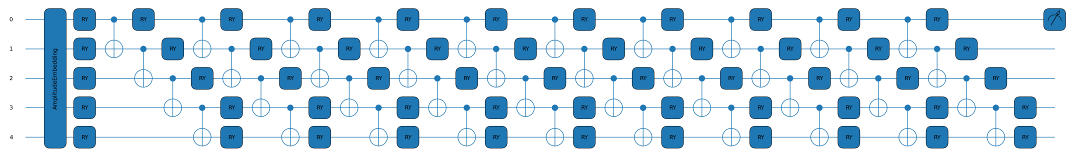
```
------------------------------------------------------------------------
Training for fold 1 ...

Score for fold 1: loss of 0.5885045654400779; accuracy of 0.675
------------------------------------------------------------------------
Training for fold 2 ...

Score for fold 2: loss of 0.5742032743541083; accuracy of 0.75
------------------------------------------------------------------------
Training for fold 3 ...

Score for fold 3: loss of 0.5726296517871493; accuracy of 0.675
------------------------------------------------------------------------
Training for fold 4 ...

Score for fold 4: loss of 0.6207334711746321; accuracy of 0.65
------------------------------------------------------------------------
Training for fold 5 ...

Score for fold 5: loss of 0.5579251616057197; accuracy of 0.825
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.5885045654400779 - Accuracy: 0.675
------------------------------------------------------------------------
> Fold 2 - Loss: 0.5742032743541083 - Accuracy: 0.75
------------------------------------------------------------------------
> Fold 3 - Loss: 0.5726296517871493 - Accuracy: 0.675
------------------------------------------------------------------------
> Fold 4 - Loss: 0.6207334711746321 - Accuracy: 0.65
------------------------------------------------------------------------
> Fold 5 - Loss: 0.5579251616057197 - Accuracy: 0.825
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5827992248723375
> Accuracy: 0.7150000000000001 (+- 0.06442049363362559)
------------------------------------------------------------------------
```

Since the performance we have obtained in the five fold cross validation process looks promising, we will finalise it by re-training our model with all the training data to make predictions on the test set.

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

     nweights = 3*nreps*nqubits

     weights={"theta": nweights}
```

```
# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model_1 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_1.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:
```
nweights = 3*nreps*nqubits
theta=np.random.rand(nweights)

fig, ax = qml.draw_mpl(qnn)(xs_train,theta)
fig.show()
```



[ ]:
```
history = model_1.fit(xs_train, y_train, epochs = 50, shuffle = True,
                      validation_data = None, batch_size = 20)
```

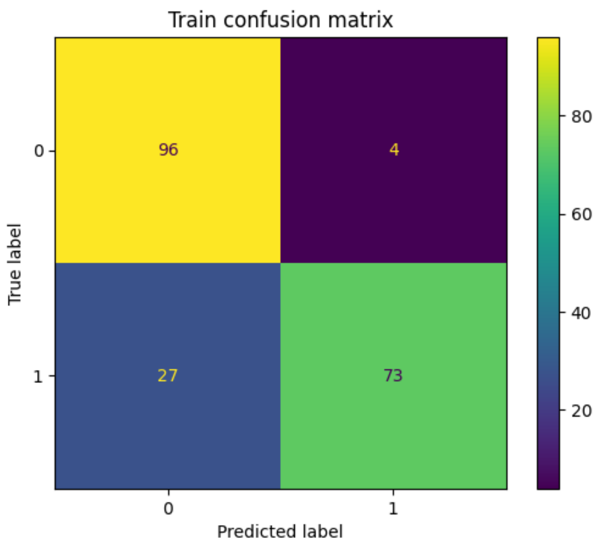
```
Epoch 1/50
10/10 [==============================] - 21s 2s/step - loss: 0.9890
Epoch 2/50
10/10 [==============================] - 19s 2s/step - loss: 0.9460
.
.
.
Epoch 49/50
10/10 [==============================] - 18s 2s/step - loss: 0.4731
Epoch 50/50
10/10 [==============================] - 16s 2s/step - loss: 0.4723
```

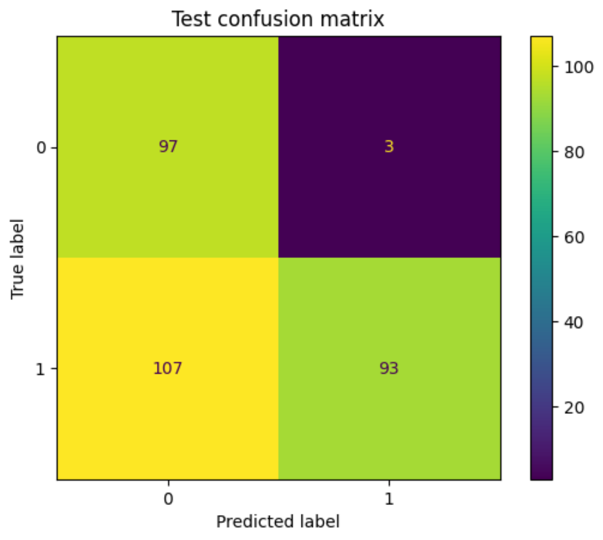
[ ]:
```
# Compute predictions
y_train_pred=model_1.predict(xs_train) >= 0.5
y_test_pred=model_1.predict(xs_test) >= 0.5
```

```
7/7 [==============================] - 8s 1s/step
10/10 [==============================] - 10s 960ms/step
```

[ ]:
```
performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.845
Train F-1 score:  0.824858757062147

Test accuracy:  0.6333333333333333
Test F-1 score:  0.6283783783783784

Test accuracy broken down per type
SEP accuracy:  0.97
PPT accuracy:  0.45
NPPT accuracy:  0.48
```





Training full dataset 17 min

Normalized data

```
[ ]: # Model 2 five fold cross validation
     fivefoldCV_qnn(qnn_circuit,xs_train_norm, y_train)
```

```
------------------------------------------------------------------------
Training for fold 1 ...

Score for fold 1: loss of 0.4644733953246071; accuracy of 0.875
------------------------------------------------------------------------
Training for fold 2 ...

Score for fold 2: loss of 0.5745343208440845; accuracy of 0.75
------------------------------------------------------------------------
Training for fold 3 ...

Score for fold 3: loss of 0.5817230830156055; accuracy of 0.75
------------------------------------------------------------------------
Training for fold 4 ...

Score for fold 4: loss of 0.5524740872067613; accuracy of 0.7
------------------------------------------------------------------------
Training for fold 5 ...

Score for fold 5: loss of 0.5461485356295814; accuracy of 0.825
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.4644733953246071 - Accuracy: 0.875
```

```
------------------------------------------------------------------------
> Fold 2 - Loss: 0.5745343208440845 - Accuracy: 0.75
------------------------------------------------------------------------
> Fold 3 - Loss: 0.5817230830156055 - Accuracy: 0.75
------------------------------------------------------------------------
> Fold 4 - Loss: 0.5524740872067613 - Accuracy: 0.7
------------------------------------------------------------------------
> Fold 5 - Loss: 0.5461485356295814 - Accuracy: 0.825
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5438706844041279
> Accuracy: 0.78 (+- 0.062048368229954284)
------------------------------------------------------------------------
```

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

     nweights = 3*nreps*nqubits

     weights={"theta": nweights}

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

     # keras model
     model_2 = tf.keras.models.Sequential([qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model_2.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: history = model_2.fit(xs_train_norm, y_train, epochs = 50, shuffle = True,
                           validation_data = None, batch_size = 20)
```

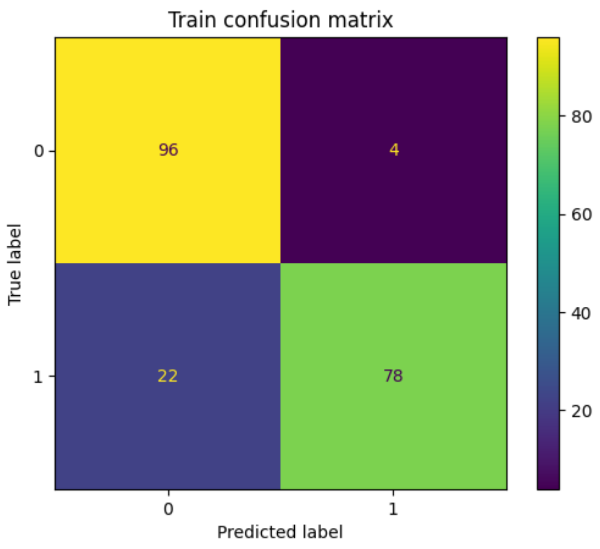
```
Epoch 1/50
10/10 [==============================] - 19s 2s/step - loss: 0.7627
Epoch 2/50
10/10 [==============================] - 28s 3s/step - loss: 0.7409
.
.
.
Epoch 49/50
10/10 [==============================] - 18s 2s/step - loss: 0.4778
Epoch 50/50
10/10 [==============================] - 18s 2s/step - loss: 0.4776
```

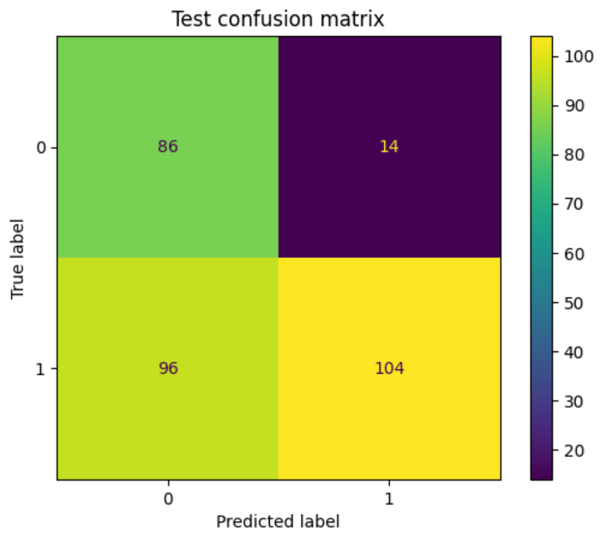
```
[ ]: # Check accuracy
     y_train_pred=model_2.predict(xs_train_norm) >= 0.5
     y_test_pred=model_2.predict(xs_test_norm) >= 0.5
```

```
7/7 [==============================] - 9s 1s/step
10/10 [==============================] - 11s 1s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.87
Train F-1 score:  0.8571428571428571

Test accuracy:  0.6333333333333333
Test F-1 score:  0.6540880503144655

Test accuracy broken down per type
SEP accuracy:  0.86
PPT accuracy:  0.49
NPPT accuracy:  0.55
```





Training full dataset 15 min

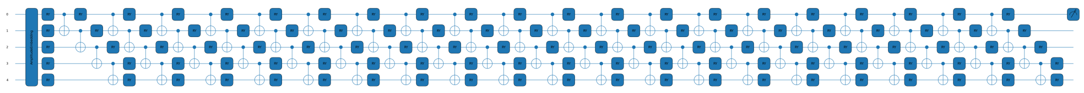
20 repetitions Next, we will try again with this variational form seeking for better results by increasing the number of repetitions. We need to improve our QNN to detect entanglement better, because in both previous experiments there is a high number of false negatives.

[ ]:
```
nqubits=5
dev=qml.device("lightning.qubit", wires=nqubits)

nreps=20

def qnn_circuit(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  TwoLocal(nqubits=nqubits, theta=theta, reps=nreps)
  return qml.expval(qml.Hermitian(M, wires=[0]))
```

[ ]:
```
method= "adjoint"

tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

nweights = 3*nreps*nqubits

weights={"theta": nweights}

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)
```

```
# keras model
model_3 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_3.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: nweights = 3*nreps*nqubits
     theta=np.random.rand(nweights)

     fig, ax = qml.draw_mpl(qnn)(xs_train,theta)
     fig.show()
```



```
[ ]: # Training our model
     history = model_3.fit(xs_train, y_train, epochs = 50, shuffle = True,
                           validation_data = None, batch_size = 20)
```

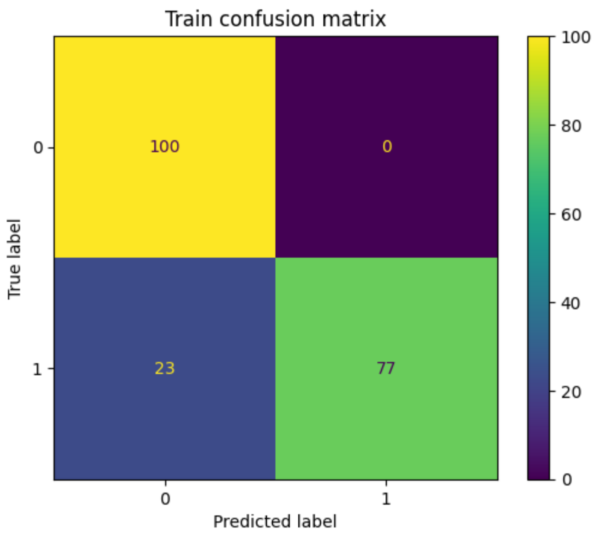
```
Epoch 1/50
10/10 [==============================] - 29s 3s/step - loss: 0.9886
Epoch 2/50
10/10 [==============================] - 25s 3s/step - loss: 0.9227
.
.
.
Epoch 49/50
10/10 [==============================] - 25s 3s/step - loss: 0.4093
Epoch 50/50
10/10 [==============================] - 30s 3s/step - loss: 0.4074
```

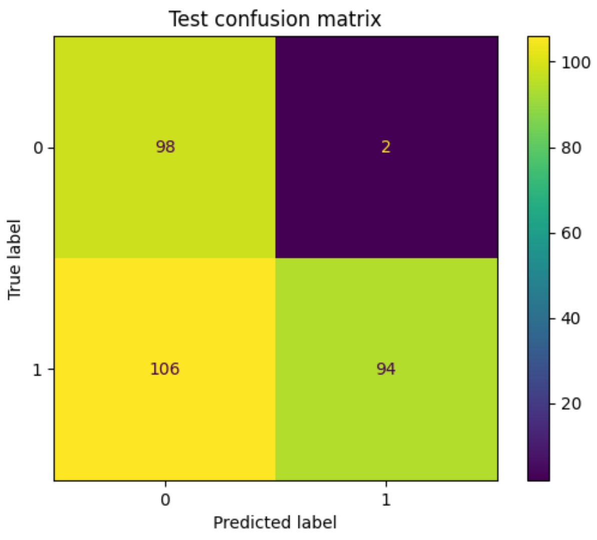
```
[ ]: # Check accuracy
     y_train_pred=model_3.predict(xs_train) >= 0.5
     y_test_pred=model_3.predict(xs_test) >= 0.5
```

```
7/7 [==============================] - 14s 2s/step
10/10 [==============================] - 19s 2s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.885
Train F-1 score:  0.8700564971751412

Test accuracy:  0.64
Test F-1 score:  0.6351351351351351
Test accuracy broken down per type
SEP accuracy:  0.98
PPT accuracy:  0.44
NPPT accuracy:  0.5
```





Training full dataset 24 min

We have tried with 15 and 20 repetitions and while it is true that the increase in the numer of repetitions slightly increases the accuracy, the tradeoff performance-executing time is not worth. The accuracy on the test set, specially detecting entanglement is really poor and the training time doubles, making it really costly to perform five fold cross validation.

3.3.2 StrongEntanglingLayers

```
[ ]: nqubits=5
     dev = qml.device("lightning.qubit", wires=nqubits)

     # number of repetitions that we want in each instance of the variational form
     nreps = 8
     #  dimensions of the input that the variational form expects
     weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)

     def qnn_circuit_strong(inputs, theta):
       qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
      ↪normalize=True)
       # reshape the theta array of parameters to make it fit into the shape that
       # the variational form expects
       theta1 = tf.reshape(theta, weights_dim)
       qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

       return qml.expval(qml.Hermitian(M, wires = [0]))
```

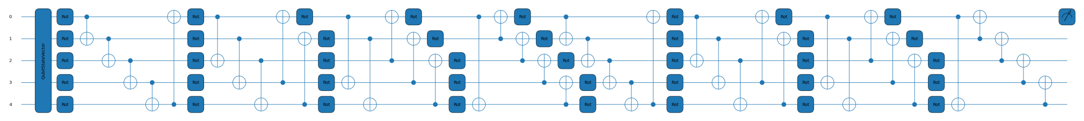
```
[ ]: # Model 4 five fold cross validation
     fivefoldCV_qnn(qnn_circuit_strong,xs_train, y_train)
```

```
------------------------------------------------------------------------
Training for fold 1 ...

Score for fold 1: loss of 0.5192811509641961; accuracy of 0.725
------------------------------------------------------------------------
Training for fold 2 ...

Score for fold 2: loss of 0.5206793387352479; accuracy of 0.775
------------------------------------------------------------------------
Training for fold 3 ...

Score for fold 3: loss of 0.502250655284547; accuracy of 0.8
------------------------------------------------------------------------
Training for fold 4 ...

Score for fold 4: loss of 0.5195175620979109; accuracy of 0.775
------------------------------------------------------------------------
Training for fold 5 ...

Score for fold 5: loss of 0.5569835664692083; accuracy of 0.725
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
```

```
> Fold 1 - Loss: 0.5192811509641961 - Accuracy: 0.725
------------------------------------------------------------------------
> Fold 2 - Loss: 0.5206793387352479 - Accuracy: 0.775
------------------------------------------------------------------------
> Fold 3 - Loss: 0.502250655284547 - Accuracy: 0.8
------------------------------------------------------------------------
> Fold 4 - Loss: 0.5195175620979109 - Accuracy: 0.775
------------------------------------------------------------------------
> Fold 5 - Loss: 0.5569835664692083 - Accuracy: 0.725
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5237424547102221
> Accuracy: 0.76 (+- 0.030000000000000023)
------------------------------------------------------------------------
```

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

     nweights = 3*nreps*nqubits

     weights={"theta": nweights}

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

     # keras model
     model_4 = tf.keras.models.Sequential([qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model_4.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: nweights = 3*nreps*nqubits
     theta=np.random.rand(nweights)

     fig, ax = qml.draw_mpl(qnn,expansion_strategy="device")(xs_train,theta)
     fig.show()
```



```
[ ]: # Training our model
     history = model_4.fit(xs_train, y_train, epochs = 50, shuffle = True,
                           validation_data = None, batch_size = 20)
```

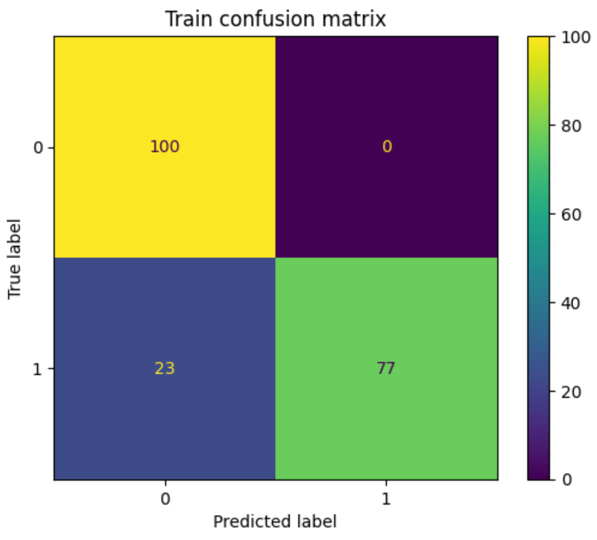
```
Epoch 1/50
10/10 [==============================] - 36s 4s/step - loss: 0.7253
Epoch 2/50
10/10 [==============================] - 28s 3s/step - loss: 0.7125
.
.
.
Epoch 49/50
10/10 [==============================] - 33s 3s/step - loss: 0.4707
Epoch 50/50
10/10 [==============================] - 33s 3s/step - loss: 0.4697
```

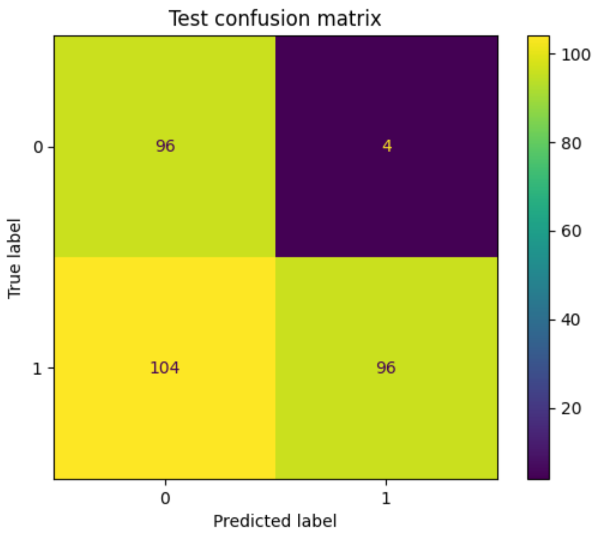
```
[ ]: # Check accuracy
     y_train_pred=model_4.predict(xs_train) >= 0.5
     y_test_pred=model_4.predict(xs_test) >= 0.5
```

```
7/7 [==============================] - 15s 2s/step
10/10 [==============================] - 20s 2s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.885
Train F-1 score:  0.8700564971751412

Test accuracy:  0.64
Test F-1 score:  0.64

Test accuracy broken down per type
SEP accuracy:  0.96
PPT accuracy:  0.46
NPPT accuracy:  0.5
```





Training time full dataset 26 min

StrongEntanglingLayers variational form training time increases as the numper of repetitions increases. We have observed in previous tests that the increase in the number of repetitions improves the performance of the QNN. For 8 repetitions, it took 2h for training with five fold cross validation and 26 minutes for training with the entire training set. Therefore, we cannot try with a higher number of repetitions because of the high training time. With this model we have still obtained a poor entanglement detection.

Normalized data

```
[ ]: # Model 4n five fold cross validation (normalized data)
     fivefoldCV_qnn(qnn_circuit_strong,xs_train_norm, y_train)
```

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

     nweights = 3*nreps*nqubits

     weights={"theta": nweights}

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

     # keras model
     model_4n = tf.keras.models.Sequential([qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model_4n.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model_4n.fit(xs_train_norm, y_train, epochs = 50, shuffle = True,
                         validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 33s 3s/step - loss: 0.7903
Epoch 2/50
10/10 [==============================] - 34s 4s/step - loss: 0.7656
.
.
.
Epoch 49/50
10/10 [==============================] - 33s 3s/step - loss: 0.4556
Epoch 50/50
10/10 [==============================] - 28s 3s/step - loss: 0.4550
```

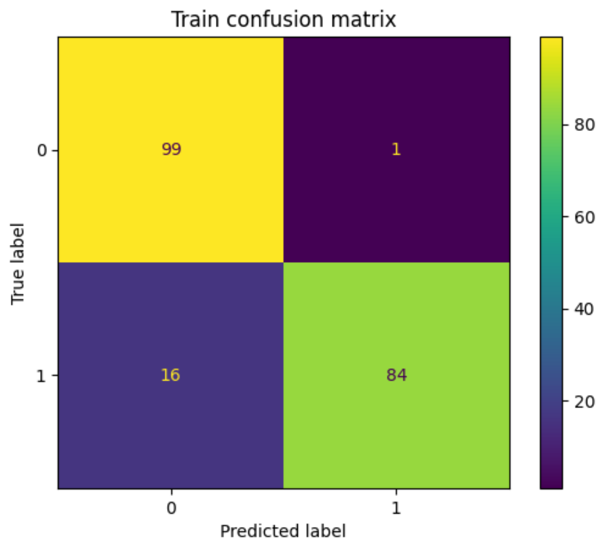
```
[ ]: # Check accuracy
     y_train_pred=model_4n.predict(xs_train_norm) >= 0.5
     y_test_pred=model_4n.predict(xs_test_norm) >= 0.5
```

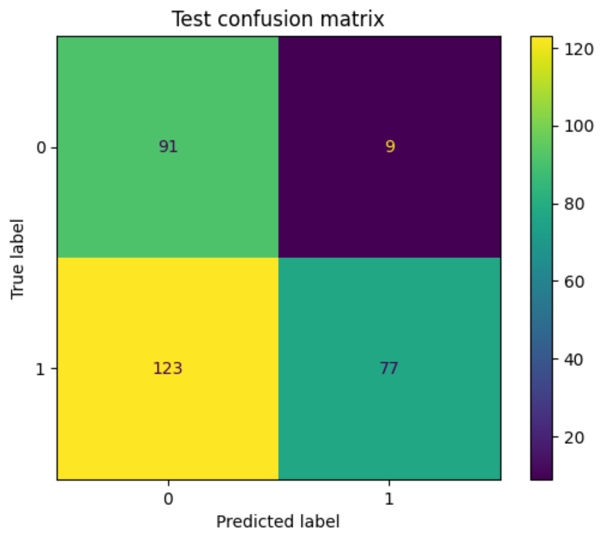
```
7/7 [==============================] - 12s 2s/step
10/10 [==============================] - 20s 2s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.915
Train F-1 score:  0.9081081081081082
```

```
Test accuracy:  0.56
Test F-1 score:  0.5384615384615384

Test accuracy broken down per type
SEP accuracy:  0.91
PPT accuracy:  0.42
NPPT accuracy:  0.35
```





Training full dataset 26 min

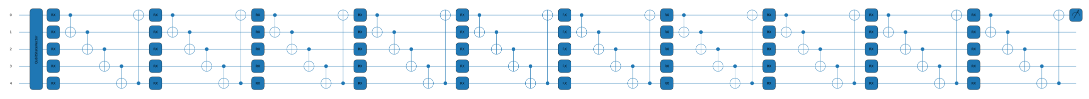
3.3.3 BasicEntanglerLayers

It uses by default the single qubit X rotation gate

[ ]:
```
nqubits=5
dev = qml.device("lightning.qubit", wires=nqubits)

# number of repetitions that we want in each instance of the variational form
nreps = 10


def qnn_circuit_basic(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  qml.BasicEntanglerLayers(weights=theta, wires=range(nqubits))
  return qml.expval(qml.Hermitian(M, wires = [0]))

n_layers = 6
weights = {"theta": (nreps, nqubits)}
```

[ ]:
```
method= "adjoint"

tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit_basic, dev, interface="tf", diff_method=method)

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model_5 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_5.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:
```
theta=np.random.rand(nreps,nqubits)

fig, ax = qml.draw_mpl(qnn,expansion_strategy="device")(xs_train,theta)
fig.show()
```



```
[ ]: # Training our model
     history = model_5.fit(xs_train, y_train, epochs = 50, shuffle = True,
                           validation_data = None, batch_size = 20)
```

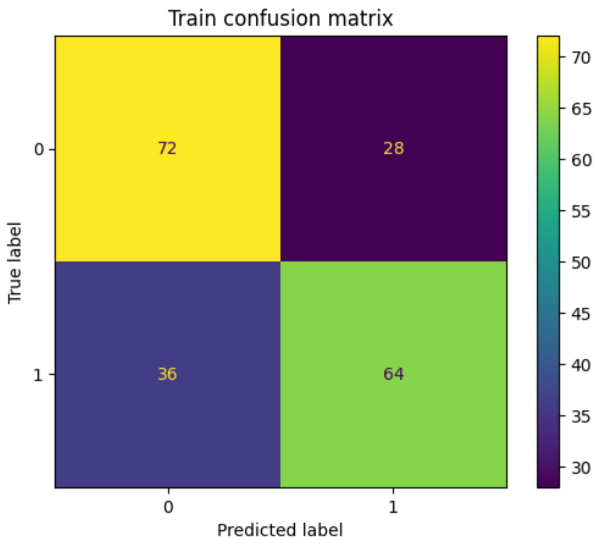
```
Epoch 1/50
10/10 [==============================] - 23s 2s/step - loss: 0.6810
Epoch 2/50
10/10 [==============================] - 27s 3s/step - loss: 0.6728
.
.
.
Epoch 49/50
10/10 [==============================] - 15s 2s/step - loss: 0.6456
Epoch 50/50
10/10 [==============================] - 16s 1s/step - loss: 0.6458
```

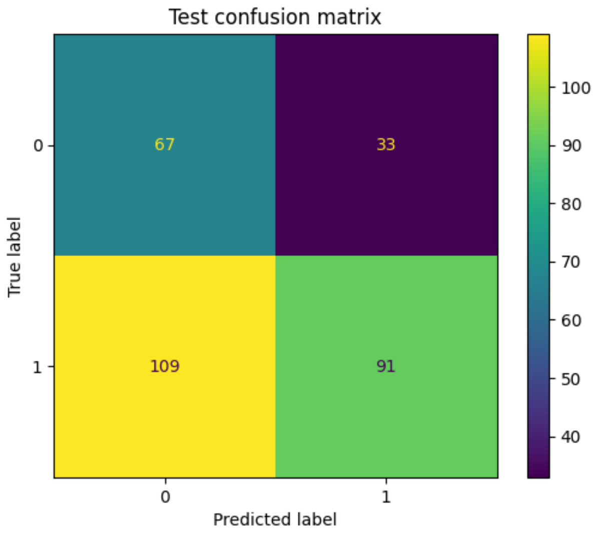
```
[ ]: y_train_pred=model_5.predict(xs_train) >= 0.5
     y_test_pred=model_5.predict(xs_test) >= 0.5
```

```
7/7 [==============================] - 7s 966ms/step
10/10 [==============================] - 10s 938ms/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.68
Train F-1 score:  0.6666666666666666

Test accuracy:  0.5266666666666666
Test F-1 score:  0.5617283950617283

Test accuracy broken down per type
SEP accuracy:  0.67
PPT accuracy:  0.48
NPPT accuracy:  0.43
```





Training full dataset 20 min

We cannot apply our five fold cross validation function because of the shape of the weights of this variational form. Anyway, this is the worst accuracy

3.3.4 Hybrid model

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit_basic, dev, interface="tf", diff_method=method)

     clayer1 = tf.keras.layers.Input(32)
     clayer2 = tf.keras.layers.Dense(32, activation="sigmoid")

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

     # keras model
     model_6 = tf.keras.models.Sequential([clayer1, clayer2, qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model_6.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model_6.fit(xs_train, y_train, epochs = 50, shuffle = True,
                          validation_data = None, batch_size = 20)
```

```
Epoch 1/50

WARNING:tensorflow:Gradients do not exist for variables ['dense_2/kernel:0',
'dense_2/bias:0'] when minimizing the loss. If you're using `model.compile()`,
did you forget to provide a `loss`argument?

 1/10 [==>...] - ETA: 1:33 - loss: 0.6914

WARNING:tensorflow:Gradients do not exist for variables ['dense_2/kernel:0',
'dense_2/bias:0'] when minimizing the loss. If you're using `model.compile()`,
did you forget to provide a `loss`argument?

 2/10 [=====>...] - ETA: 15s - loss: 0.6921

WARNING:tensorflow:Gradients do not exist for variables ['dense_2/kernel:0',
'dense_2/bias:0'] when minimizing the loss. If you're using `model.compile()`,
did you forget to provide a `loss`argument?
```

```
[ ]: # Check accuracy
     y_train_pred=model_6.predict(xs_train) >= 0.5
     y_test_pred=model_6.predict(xs_test) >= 0.5

     tr_acc = accuracy_score(y_train_pred, y_train)
     test_acc = accuracy_score(y_test_pred, y_test_small)
```

```
7/7 [==============================] - 10s 1s/step
10/10 [==============================] - 9s 884ms/step
```

```
[ ]: print("Train accuracy: ", tr_acc)
     print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.555
Test accuracy:  0.46
```

```
[ ]: nqubits=4
     dev = qml.device("lightning.qubit", wires=nqubits)

     # number of repetitions that we want in each instance of the variational form
     nreps = 2
     #  dimensions of the input that the variational form expects
     weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
     #  number of inputs that each instance of the variational form will take
     nweights = 3*nreps*nqubits

     def qnn_circuit_strong(inputs, theta):
       qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
      ↪normalize=True)
       # reshape the theta array of parameters to make it fit into the shape that
       # the variational form expects
       theta1 = tf.reshape(theta, weights_dim)
       qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

       return qml.expval(qml.Hermitian(M, wires = [0]))

     # dictionary we would send to TensorFlow when constructing the Keras layer
     weights_strong = {"theta": nweights}
```

```
[ ]: tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit_strong, dev, interface="tf")

     input = tf.keras.Input(shape=(32,))
     clayer = tf.keras.layers.Dense(16, use_bias=False)

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn, weights_strong, output_dim=1)

     # keras model
     model_7 = tf.keras.models.Sequential([input, clayer, qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model_7.compile(opt, loss='binary_crossentropy')

     model_7.summary()
```

```
Model: "sequential_10"
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_16 (Dense)            (None, 16)                512

 keras_layer_12 (KerasLayer)  (None, 1)                0 (unused)

=================================================================
Total params: 512
Trainable params: 512
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
# Training our model
history = model_7.fit(xs_train, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50

WARNING:tensorflow:Gradients do not exist for variables ['dense_16/kernel:0']
when minimizing the loss. If you're using `model.compile()`, did you forget to
provide a `loss`argument?

 1/10 [==>...] - ETA: 4:20 - loss: 0.7640

WARNING:tensorflow:Gradients do not exist for variables ['dense_16/kernel:0']
when minimizing the loss. If you're using `model.compile()`, did you forget to
provide a `loss`argument?

 2/10 [=====>...] - ETA: 26s - loss: 0.7501
```

[ ]:
```
# Check accuracy
y_train_pred=model_7.predict(xs_train) >= 0.5
y_test_pred=model_7.predict(xs_test) >= 0.5

tr_acc = accuracy_score(y_train_pred, y_train)
test_acc = accuracy_score(y_test_pred, y_test_small)
```

```
7/7 [==============================] - 14s 2s/step
10/10 [==============================] - 6s 620ms/step
```

[ ]:
```
print("Train accuracy: ", tr_acc)
print("Test accuracy: ", test_acc)
```

```
Train accuracy:  0.445
Test accuracy:  0.4666666666666667
```

4 Case study: dataset with 1.0 PPT ratio

- A training set of 200 samples: 100 separable and 100 entangled (with 1.0 ppt ratio, thus 100 samples ppt-ent and 0 samples nppt-ent). File *train_ set.csv*
- A test set of 300 samples, 100 separable, 100 ppt-entangled and 100 nppt-entangled. It is a reduced version of the papers test set consisting in 1000 samples per type. File *test_ set_ small.csv*

Additionally, we will use 5 fold cross validation for the training process of the models.

```
[ ]: training_data = pd.read_csv("/content/drive/MyDrive/tfg/x_train_1.csv", header=None)
     training_data.head()
```

```
[ ]:          0         1         2         3         4         5         6   ...
     0  0.002405 -0.013551  0.014313  0.007182 -0.008649 -0.003936 -0.002174
     1 -0.006249  0.010679 -0.000343 -0.003259 -0.001029 -0.008732 -0.002129
     2  0.012010  0.006747  0.006879  0.000910 -0.001017 -0.003304 -0.003681
     3 -0.147372 -0.009079 -0.008336 -0.021830 -0.012960 -0.004989  0.054233
     4 -0.010161 -0.021211 -0.009640 -0.004726  0.000182 -0.013206 -0.001762


              74        75        76        77        78        79
     0  0.045722  0.076358  0.083020  0.093225  0.105654  0.109198
     1  0.061884  0.080446  0.095130  0.103344  0.103996  0.110223
     2  0.028129  0.060973  0.073608  0.078672  0.095634  0.102708
     3  0.105213  0.087479  0.105942  0.130937  0.111882  0.118560
     4  0.055780  0.075803  0.095085  0.105444  0.107857  0.114460

     [5 rows x 80 columns]
```

```
[ ]: training_data.describe()
```

```
[ ]:                 0           1           2           3           4           5  ...
     count  200.000000  200.000000  200.000000  200.000000  200.000000  200.000000
     mean     0.004797    0.000064    0.000108   -0.000144    0.000498    0.001381
     std      0.028860    0.029620    0.036638    0.025325    0.026439    0.032549
     min     -0.147372   -0.081901   -0.187091   -0.069290   -0.080485   -0.159721
     25%     -0.011422   -0.019337   -0.015301   -0.011488   -0.010467   -0.014435
     50%      0.005595    0.000423    0.000219   -0.000011   -0.000762    0.001258
     75%      0.024328    0.016108    0.018386    0.007331    0.010476    0.018243
     max      0.078962    0.099702    0.170843    0.073404    0.087689    0.114851


                    77          78          79
     count  200.000000  200.000000  200.000000
     mean     0.097276    0.106195    0.112494
     std      0.013057    0.010411    0.008404
     min      0.057148    0.059018    0.070539
     25%      0.091099    0.100307    0.108088
     50%      0.096756    0.107006    0.112603
     75%      0.104528    0.111522    0.116679
     max      0.162879    0.162101    0.152014

     [8 rows x 80 columns]
```

```
[ ]: x_train = np.genfromtxt("/content/drive/MyDrive/tfg/x_train_1.csv",␣
      ↪delimiter=",",dtype=None)
     y_train = np.genfromtxt("/content/drive/MyDrive/tfg/y_train_1.csv",␣
      ↪delimiter=",",dtype=None)
```

4.1 Exploratory data analysis

```
[ ]: pca = PCA(n_components = 32)

     xs_train = pca.fit_transform(x_train)
     xs_test = pca.transform(x_test_small)
```

```
[ ]: three_pc = pd.DataFrame(
         data=pca.components_[0:3],
         index = ['PC1', 'PC2', 'PC3']
     )
```

```
[ ]: three_pc
```

```
[ ]:           0         1         2         3         4         5         6  ...
     PC1 -0.161845 -0.022361 -0.034527  0.045538  0.044388  0.110899  0.046586
     PC2 -0.041790  0.087543 -0.464655  0.028212 -0.038200  0.201786  0.081180
     PC3 -0.124399  0.134612 -0.014800  0.033860 -0.003348 -0.382316  0.007777


                73        74        75        76        77        78        79
     PC1  0.057525  0.096779  0.060026  0.031667  0.003717 -0.010086  0.007219
     PC2  0.033076  0.007223  0.007607  0.009609  0.023536  0.023750  0.023123
     PC3  0.035481  0.079102  0.033847  0.058791  0.098434  0.067680  0.049390

     [3 rows x 80 columns]
```

```
[ ]: print('Porcentaje de varianza explicada por cada componente')
     print(pca.explained_variance_ratio_)
```

```
Porcentaje de varianza explicada por cada componente
[0.05654583 0.05482265 0.04150263 0.03946556 0.03682448 0.03483101
 0.0316728  0.02962571 0.0272871  0.02612473 0.0254304  0.02540558
 0.02431726 0.02338352 0.02163313 0.02123271 0.02030116 0.01983655
 0.01857565 0.01796784 0.01784044 0.01704973 0.01620598 0.01576475
 0.01498603 0.01467931 0.01376626 0.01358834 0.01350699 0.01288878
 0.01268894 0.01245    ]
```

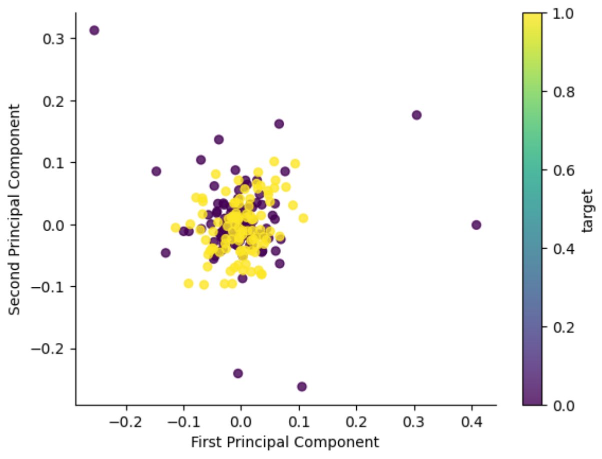
```
[ ]: pca_df = pd.DataFrame(
         data = xs_train[:,0:3],
         columns = ['PC1', 'PC2', 'PC3']
     )
     pca_df = pd.concat([pca_df, pd.DataFrame(y_train, columns =['target'])[['target']]],␣
      ↪axis=1)
```

```
[ ]: pca_df
```

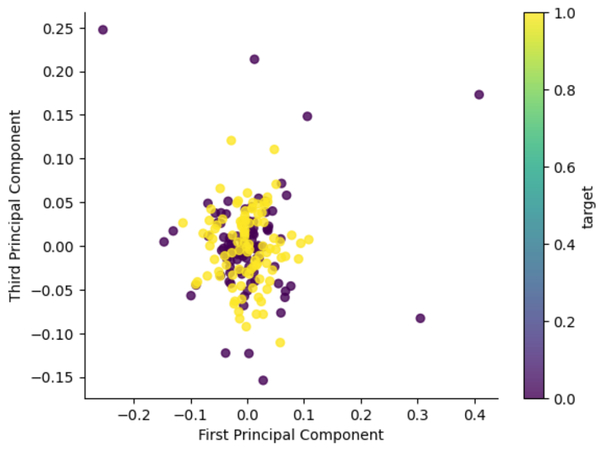
```
[ ]:          PC1       PC2       PC3  target
     0   -0.033562 -0.021375  0.001327     0.0
     1   -0.007987 -0.004189  0.006105     0.0
     2   -0.025172 -0.000218 -0.001940     0.0
```

```
3    0.408645 -0.001203  0.173139     0.0
4    0.019979  0.027201  0.054462     0.0
..        ...       ...       ...     ...
195  0.024591 -0.045316 -0.063818     1.0
196 -0.015872 -0.024501  0.051496     1.0
197  0.009933  0.010972  0.049546     1.0
198 -0.065871  0.036739 -0.000013     1.0
199 -0.025628 -0.039407 -0.033502     1.0

[200 rows x 4 columns]
```

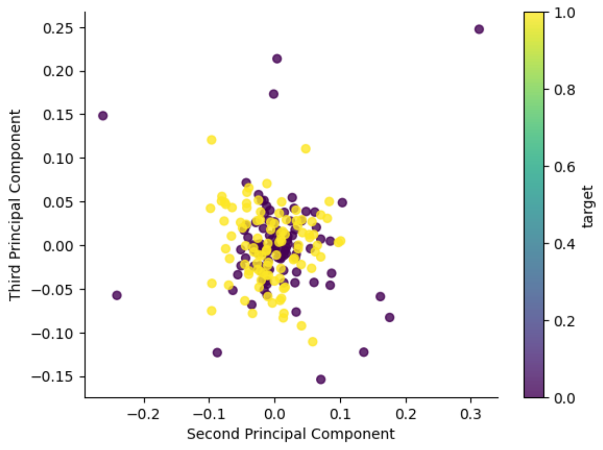
[ ]:
```
pca_df.plot(kind='scatter', x='PC1', y='PC2', c='target', s=32, alpha=.8)
plt.xlabel('First Principal Component')

plt.ylabel('Second Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```



[ ]:
```
pca_df.plot(kind='scatter', x='PC1', y='PC3', c='target', s=32, alpha=.8)
plt.xlabel('First Principal Component')

plt.ylabel('Third Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```



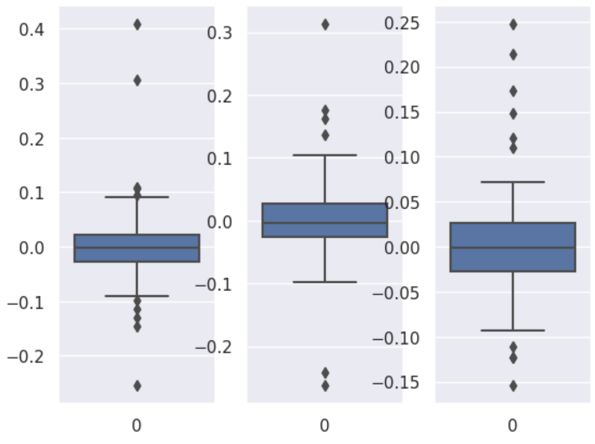
```
[ ]: pca_df.plot(kind='scatter', x='PC2', y='PC3', c='target', s=32, alpha=.8)
     plt.xlabel('Second Principal Component')

     plt.ylabel('Third Principal Component')

     plt.gca().spines[['top', 'right',]].set_visible(False)
```



```
[ ]: # Box Plot
     import seaborn as sns

     #set seaborn plotting aesthetics as default
     sns.set()

     #define plotting region (2 rows, 2 columns)
```

```
fig, axes = plt.subplots(1, 3)

#create boxplot in each subplot
sns.boxplot(data=pca_df['PC1'], ax=axes[0])
sns.boxplot(data=pca_df['PC2'], ax=axes[1])
sns.boxplot(data=pca_df['PC3'], ax=axes[2])
```

```
[ ]: <Axes: >
```



MaxAbsScaler normalization

```
[ ]: scaler = MaxAbsScaler()
     x_train_norm = scaler.fit_transform(x_train)

     x_test_norm = scaler.transform(x_test_small)

     # Restrict all the values to be between 0 and 1
     x_test_norm = np.clip(x_test_norm,0,1)
```

```
[ ]: pca = PCA(n_components = 32)

     xs_train_norm = pca.fit_transform(x_train_norm)
     xs_test_norm = pca.transform(x_test_norm)
```

```
[ ]: three_pc_norm = pd.DataFrame(
         data=pca.components_[0:3],
         index = ['PC1', 'PC2', 'PC3']
     )
```

```
[ ]: three_pc_norm
```

```
[ ]:             0         1         2         3         4         5         6   ...
     PC1 -0.088289  0.179963 -0.152478  0.172708  0.004456  0.057968  0.148815
```

```
PC2  0.037075  0.007439  0.065940  0.042914  0.202427  0.005615 -0.033994
PC3  0.027372  0.114360 -0.066136  0.126286  0.053287 -0.102089 -0.061799


           73        74        75        76        77        78        79
PC1  0.059820  0.079378  0.056832  0.027311  0.021976  0.013616  0.018079
PC2 -0.005179 -0.017332  0.010894  0.011215 -0.009000 -0.010088 -0.007949
PC3 -0.006481  0.000265  0.009991  0.015081  0.023333  0.024105  0.017907

[3 rows x 80 columns]
```

```
[ ]: print('Porcentaje de varianza explicada por cada componente')
     print(pca.explained_variance_ratio_)
```

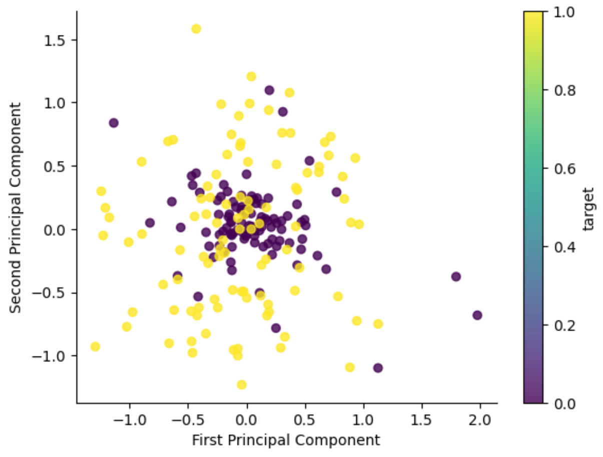
```
Porcentaje de varianza explicada por cada componente
[0.04519081 0.04305087 0.04040429 0.03852013 0.0358103  0.03383065
 0.03208468 0.03121227 0.03024195 0.02948633 0.02856054 0.02657779
 0.02529048 0.02499907 0.02368502 0.02338174 0.02270439 0.02130465
 0.02036222 0.01929434 0.01862555 0.01833084 0.0176221  0.01709922
 0.01623152 0.015521   0.01503582 0.01450377 0.01365743 0.01317983
 0.01273295 0.01217936]
```

```
[ ]: pca_norm_df = pd.DataFrame(
         data = xs_train_norm[:,0:3],
         columns = ['PC1', 'PC2', 'PC3']
     )
     pca_norm_df = pd.concat([pca_norm_df, pd.DataFrame(y_train, columns␣
      ↪=['target'])[['target']]], axis=1)
```

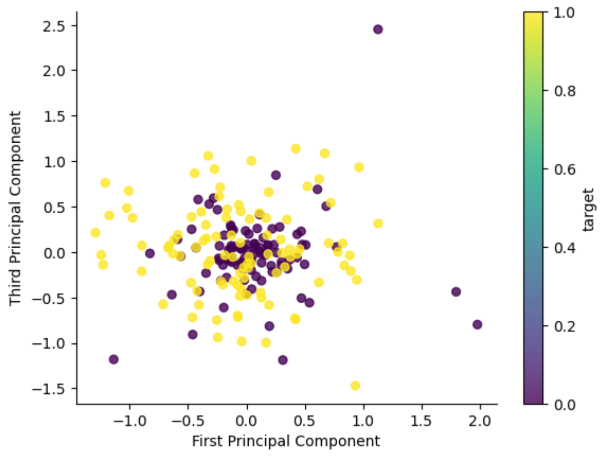
```
[ ]: pca_norm_df
```

```
[ ]:          PC1       PC2       PC3  target
     0   -0.230297 -0.045281  0.067306     0.0
     1   -0.016277  0.117728 -0.071714     0.0
     2   -0.122600 -0.141607  0.226814     0.0
     3    1.981290 -0.681880 -0.798570     0.0
     4    0.114172 -0.504560  0.411078     0.0
     ..        ...       ...       ...     ...
     195 -0.215890  0.986376 -0.324199     1.0
     196  0.295155 -0.938925  0.137995     1.0
     197  0.456633 -0.306292  0.033884     1.0
     198 -0.662429 -0.903509  0.070498     1.0
     199 -0.461377 -0.978318 -0.719641     1.0

     [200 rows x 4 columns]
```

```
[ ]: pca_norm_df.plot(kind='scatter', x='PC1', y='PC2', c='target', s=32, alpha=.8,␣
      ↪cmap='viridis')
     plt.xlabel('First Principal Component')

     plt.ylabel('Second Principal Component')
plt.gca().spines[['top', 'right',]].set_visible(False)
```

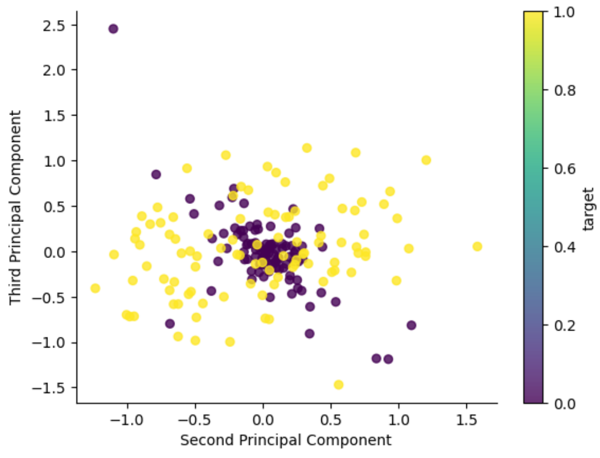


```
[ ]: pca_norm_df.plot(kind='scatter', x='PC1', y='PC3', c='target', s=32, alpha=.8,␣
      ↪cmap='viridis')
     plt.xlabel('First Principal Component')

     plt.ylabel('Third Principal Component')

     plt.gca().spines[['top', 'right',]].set_visible(False)
```



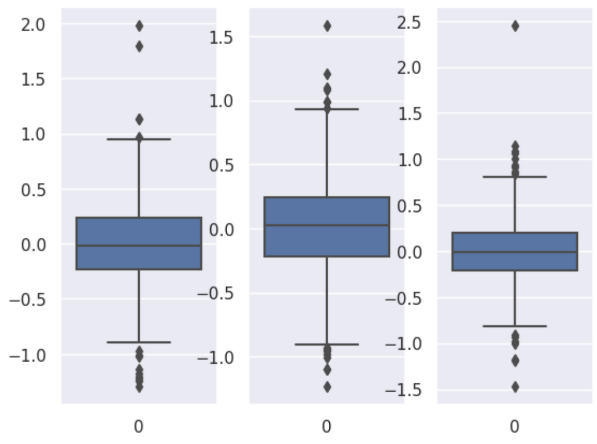
```
[ ]: pca_norm_df.plot(kind='scatter', x='PC2', y='PC3', c='target', s=32, alpha=.8,␣
      ↪cmap='viridis')
```

```
plt.xlabel('Second Principal Component')

plt.ylabel('Third Principal Component')

plt.gca().spines[['top', 'right',]].set_visible(False)
```



```
[ ]: # Box Plot

     #set seaborn plotting aesthetics as default
     sns.set()

     #define plotting region (2 rows, 2 columns)
     fig, axes = plt.subplots(1, 3)

     #create boxplot in each subplot
     sns.boxplot(data=pca_norm_df['PC1'], ax=axes[0])
     sns.boxplot(data=pca_norm_df['PC2'], ax=axes[1])
     sns.boxplot(data=pca_norm_df['PC3'], ax=axes[2])
```

```
[ ]: <Axes: >
```



4.2 Quantum Support Vector Machines

[ ]:
```
svm = SVC(kernel = qkernel).fit(xs_train, y_train)
```

[ ]:
```
y_train_pred=svm.predict(xs_train)
```

[ ]:
```
tr_acc=accuracy_score(y_train_pred, y_train)
print("Train accuracy: ", tr_acc)
```

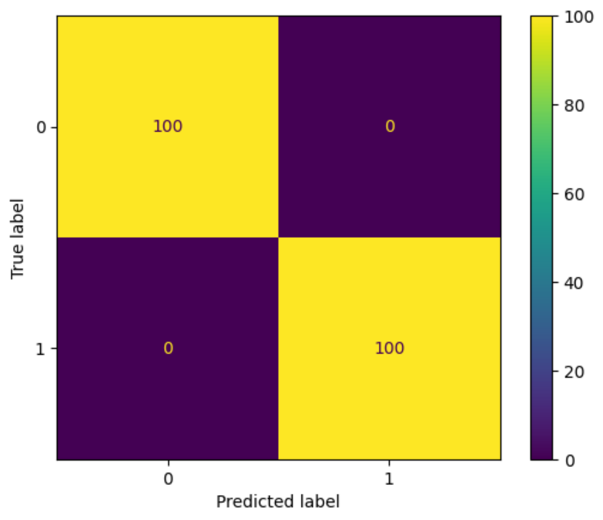
```
Train accuracy:  1.0
```

[ ]:
```
y_test_pred=svm.predict(xs_test)
```

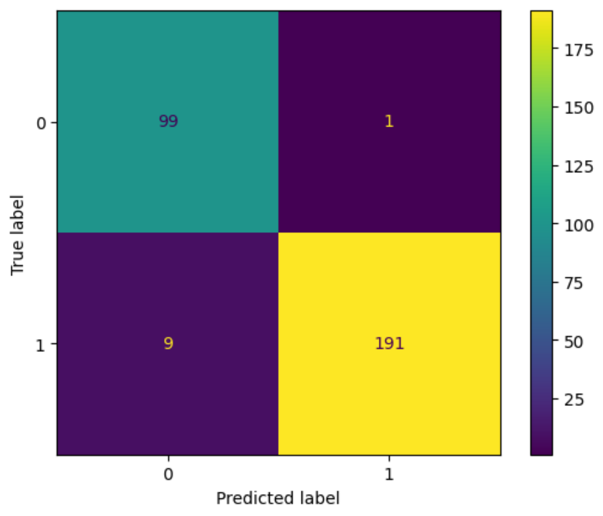
[ ]:
```
test_acc=accuracy_score(y_test_pred, y_test_small)
print("Test accuracy: ", test_acc)
```

```
Test accuracy:  0.9666666666666667
```

[ ]:
```
# Train confussion matrix
cm = confusion_matrix(y_train, y_train_pred)
cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: # Test confussion matrix
     cm = confusion_matrix(y_test_small, y_test_pred)
     cm_display = ConfusionMatrixDisplay(cm).plot()
```



```
[ ]: # Test accuracy per type
     detailed_accuracy(y_test_pred, 100)
```

```
SEP accuracy:  0.99
PPT accuracy:  0.93
NPPT accuracy:  0.98
```

```
[ ]: model_type = SVC()
     model_params = [{'kernel': [qkernel]}]
     model = GridSearchCV(model_type, model_params, cv=StratifiedKFold(shuffle=True)).
      ↪fit(xs_train, y_train)
     print('Training results :')
     print(model.best_params_)
     print(model.best_score_)
```

```
Training results :
{'kernel': <function qkernel at 0x7cd0d8e4b6d0>}
0.985
```

Training time 12 minutes

Train Prediction time 11 minutes

Test prediction time 18 min

Five fold cross validation 1 hour

4.3 Quantum Neural Networks

```
[ ]: def save_modelkeras(model,filename):
       dir="/content/drive/MyDrive/tfg/models/"+filename
       model.save(dir)
       print("Keras model saved")
```

```
[ ]: def load_modelkeras(filename):
       # Recreate the exact same model, including its weights and the optimizer
       dir="/content/drive/MyDrive/tfg/models/"+filename
       new_model = tf.keras.models.load_model(dir)
       return new_model
```

4.3.1 TwoLocal variational form

```
[ ]: nqubits=5
     dev=qml.device("lightning.qubit", wires=nqubits)

     nreps=10

     def qnn_circuit(inputs, theta):
       qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
      ↪normalize=True)
       TwoLocal(nqubits=nqubits, theta=theta, reps=nreps)
       return qml.expval(qml.Hermitian(M, wires=[0]))
```

```
[ ]: # Model 1 five fold cross validation
     fivefoldCV_qnn(qnn_circuit,xs_train, y_train)
```

```
------------------------------------------------------------------------
Training for fold 1 ...

Score for fold 1: loss of 0.4691675038124646; accuracy of 0.9
```

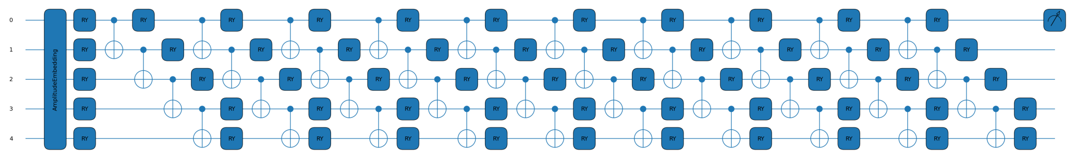
```
------------------------------------------------------------------------
Training for fold 2 ...

Score for fold 2: loss of 0.5366220189702486; accuracy of 0.7
------------------------------------------------------------------------
Training for fold 3 ...

Score for fold 3: loss of 0.5642844875142827; accuracy of 0.7
------------------------------------------------------------------------
Training for fold 4 ...

Score for fold 4: loss of 0.53884107944004; accuracy of 0.75
------------------------------------------------------------------------
Training for fold 5 ...

Score for fold 5: loss of 0.6026014746873601; accuracy of 0.65
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.4691675038124646 - Accuracy: 0.9
------------------------------------------------------------------------
> Fold 2 - Loss: 0.5366220189702486 - Accuracy: 0.7
------------------------------------------------------------------------
> Fold 3 - Loss: 0.5642844875142827 - Accuracy: 0.7
------------------------------------------------------------------------
> Fold 4 - Loss: 0.53884107944004 - Accuracy: 0.75
------------------------------------------------------------------------
> Fold 5 - Loss: 0.6026014746873601 - Accuracy: 0.65
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5423033128848792
> Accuracy: 0.74 (+- 0.08602325267042628)
------------------------------------------------------------------------
```

[ ]:

```
method= "adjoint"

tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

nweights = 3*nreps*nqubits

weights={"theta": nweights}

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model_1 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
```

```
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_1.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: nweights = 3*nreps*nqubits
     theta=np.random.rand(nweights)

     fig, ax = qml.draw_mpl(qnn)(xs_train,theta)
     fig.show()
```



```
[ ]: history = model_1.fit(xs_train, y_train, epochs = 50, shuffle = True,
                          validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 15s 1s/step - loss: 0.9957
Epoch 2/50
10/10 [==============================] - 17s 1s/step - loss: 0.9520
.
.
.
Epoch 49/50
10/10 [==============================] - 18s 2s/step - loss: 0.4449
Epoch 50/50
10/10 [==============================] - 19s 2s/step - loss: 0.4435
```

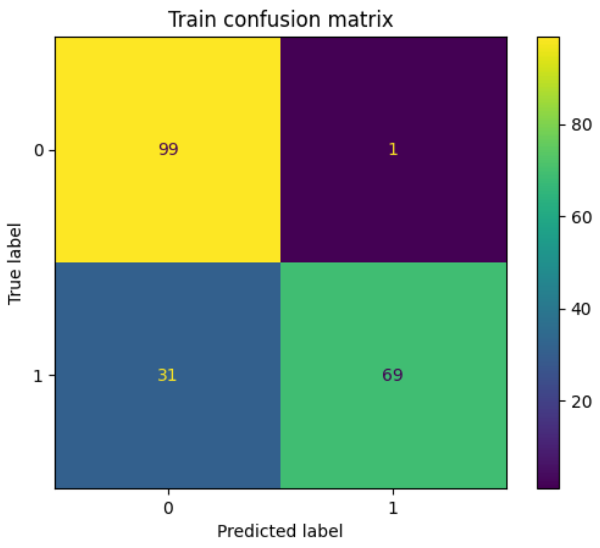
```
[ ]: y_train_pred=model_1.predict(xs_train) >= 0.5
     y_test_pred=model_1.predict(xs_test) >= 0.5
```

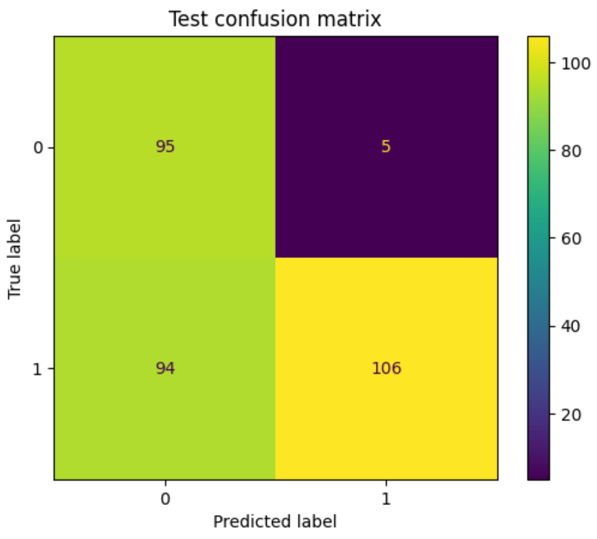
```
7/7 [==============================] - 9s 1s/step
10/10 [==============================] - 15s 2s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.84
Train F-1 score:  0.8117647058823529

Test accuracy:  0.67
Test F-1 score:  0.6816720257234727

Test accuracy broken down per type
SEP accuracy:  0.95
```

```
PPT accuracy:  0.52
NPPT accuracy:  0.54
```





Training full dataset 14 min

Normalized data

```
[ ]: # Model 2 five fold cross validation (normalized data)
     fivefoldCV_qnn(qnn_circuit,xs_train_norm, y_train)
```

```
------------------------------------------------------------------------
Training for fold 1 ...

Score for fold 1: loss of 0.4813411885921739; accuracy of 0.875
------------------------------------------------------------------------
Training for fold 2 ...

Score for fold 2: loss of 0.4781813496467676; accuracy of 0.85
------------------------------------------------------------------------
Training for fold 3 ...

Score for fold 3: loss of 0.5783924202387007; accuracy of 0.725
------------------------------------------------------------------------
Training for fold 4 ...

Score for fold 4: loss of 0.5303979587338639; accuracy of 0.8
------------------------------------------------------------------------
Training for fold 5 ...

Score for fold 5: loss of 0.5760894933973543; accuracy of 0.675
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.4813411885921739 - Accuracy: 0.875
------------------------------------------------------------------------
> Fold 2 - Loss: 0.4781813496467676 - Accuracy: 0.85
------------------------------------------------------------------------
> Fold 3 - Loss: 0.5783924202387007 - Accuracy: 0.725
------------------------------------------------------------------------
> Fold 4 - Loss: 0.5303979587338639 - Accuracy: 0.8
------------------------------------------------------------------------
> Fold 5 - Loss: 0.5760894933973543 - Accuracy: 0.675
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5288804821217721
> Accuracy: 0.7849999999999999 (+- 0.07516648189186453)
------------------------------------------------------------------------
```

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

     nweights = 3*nreps*nqubits

     weights={"theta": nweights}
```

```
# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model_2 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_2.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: history = model_2.fit(xs_train_norm, y_train, epochs = 50, shuffle = True,
                         validation_data = None, batch_size = 20)
```

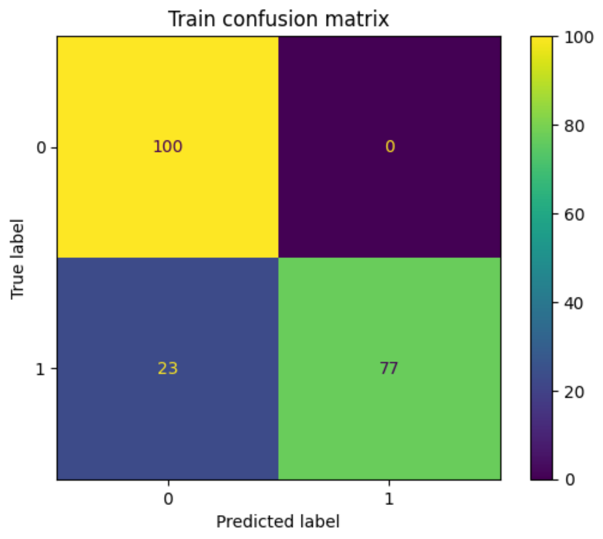
```
Epoch 1/50
10/10 [==============================] - 17s 1s/step - loss: 0.7584
Epoch 2/50
10/10 [==============================] - 18s 2s/step - loss: 0.7429
.
.
.
Epoch 49/50
10/10 [==============================] - 18s 2s/step - loss: 0.4507
Epoch 50/50
10/10 [==============================] - 18s 2s/step - loss: 0.4501
```

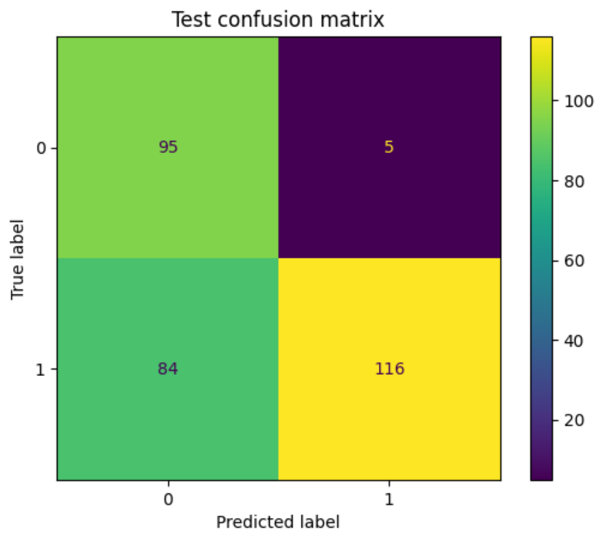
```
[ ]: y_train_pred=model_2.predict(xs_train_norm) >= 0.5
     y_test_pred=model_2.predict(xs_test_norm) >= 0.5
```

```
7/7 [==============================] - 17s 3s/step
10/10 [==============================] - 13s 1s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.885
Train F-1 score:  0.8700564971751412

Test accuracy:  0.7033333333333334
Test F-1 score:  0.7227414330218069

Test accuracy broken down per type
SEP accuracy:  0.95
PPT accuracy:  0.57
NPPT accuracy:  0.59
```





```
[ ]: save_modelkeras(model_3,'model_twolocal10reps_norm.h5')
```

```
Keras model saved
```

15 min training whole dataset

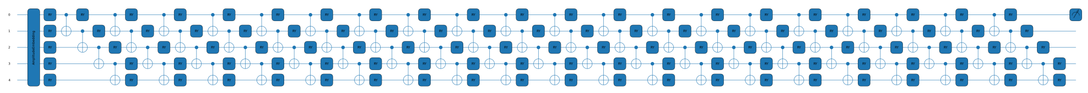
20 repetitions

```
[ ]: nqubits=5
     dev=qml.device("lightning.qubit", wires=nqubits)
```

```
nreps=20

def qnn_circuit(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
↪normalize=True)
  TwoLocal(nqubits=nqubits, theta=theta, reps=nreps)
  return qml.expval(qml.Hermitian(M, wires=[0]))
```

[ ]:
```
method= "adjoint"

tf.random.set_seed(seed)

qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

nweights = 3*nreps*nqubits

weights={"theta": nweights}

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

# keras model
model_3 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_3.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

[ ]:
```
nweights = 3*nreps*nqubits
theta=np.random.rand(nweights)

fig, ax = qml.draw_mpl(qnn)(xs_train,theta)
fig.show()
```



[ ]:
```
# Training our model
history = model_3.fit(xs_train, y_train, epochs = 50, shuffle = True,
                    validation_data = None, batch_size = 20)
```

```
Epoch 1/50
10/10 [==============================] - 28s 3s/step - loss: 0.9938
Epoch 2/50
```

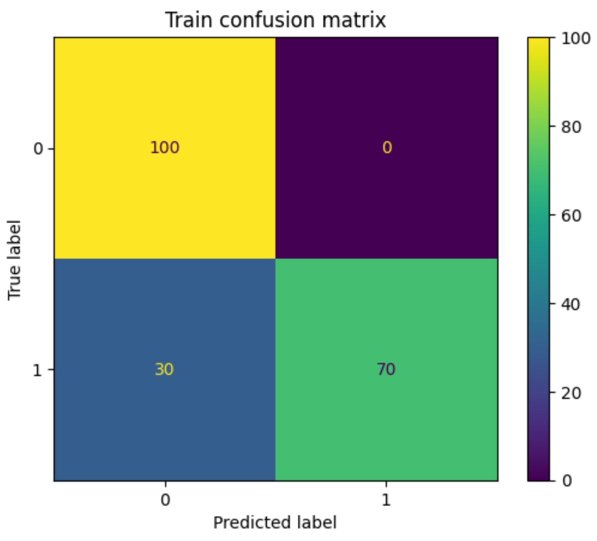
```
10/10 [==============================] - 32s 3s/step - loss: 0.9200
.
.
.
Epoch 49/50
10/10 [==============================] - 29s 3s/step - loss: 0.4082
Epoch 50/50
10/10 [==============================] - 27s 3s/step - loss: 0.4066
```

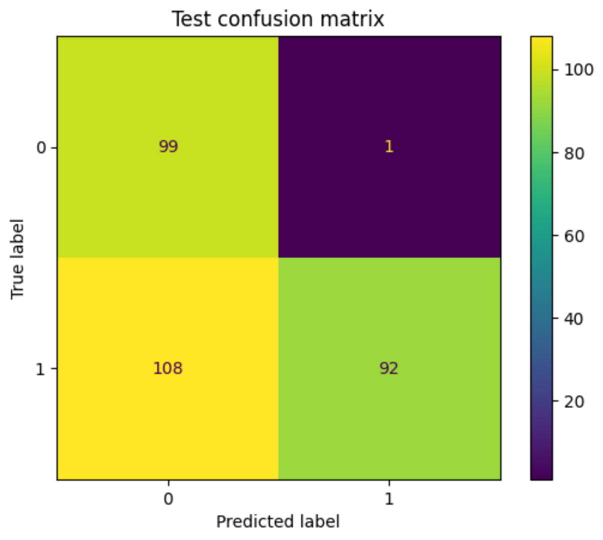
```
[ ]: y_train_pred=model_3.predict(xs_train) >= 0.5
     y_test_pred=model_3.predict(xs_test) >= 0.5
```

```
7/7 [==============================] - 14s 2s/step
10/10 [==============================] - 19s 2s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.85
Train F-1 score:  0.8235294117647058

Test accuracy:  0.6366666666666667
Test F-1 score:  0.6279863481228669

Test accuracy broken down per type
SEP accuracy:  0.99
PPT accuracy:  0.44
NPPT accuracy:  0.48
```





Training time 24 minutes

20 repetitions, normalized data

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn = qml.QNode(qnn_circuit, dev, interface="tf", diff_method=method)

     nweights = 3*nreps*nqubits

     weights={"theta": nweights}

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn, weights, output_dim=1)

     # keras model
     model_3n = tf.keras.models.Sequential([qlayer])

     # we choose adam optimizer with a learning rate of 0.005
     opt = tf.keras.optimizers.Adam(learning_rate=0.005)

     # binary cross entropy loss, because we are training a binary classifier
     model_3n.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model_3n.fit(xs_train_norm, y_train, epochs = 50, shuffle = True,
                         validation_data = None, batch_size = 20)
```

```
Epoch 1/50
```

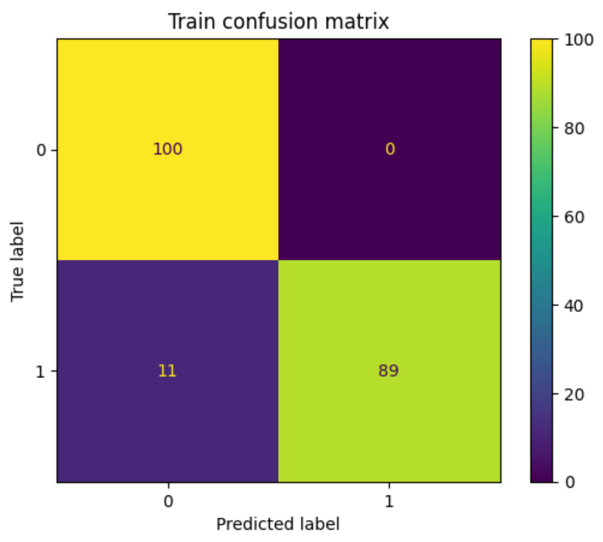
```
10/10 [==============================] - 32s 3s/step - loss: 0.7562
Epoch 2/50
10/10 [==============================] - 26s 3s/step - loss: 0.7098
.
.
.
Epoch 49/50
10/10 [==============================] - 31s 3s/step - loss: 0.3772
Epoch 50/50
10/10 [==============================] - 26s 3s/step - loss: 0.3763
```

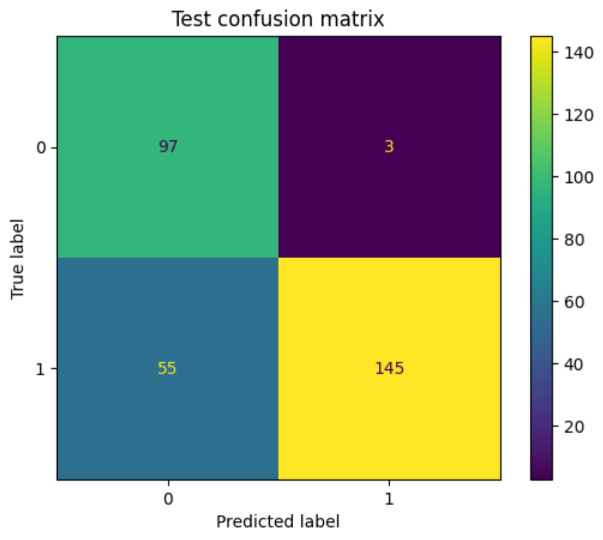
```
[ ]: y_train_pred=model_3n.predict(xs_train_norm) >= 0.5
     y_test_pred=model_3n.predict(xs_test_norm) >= 0.5
```

```
7/7 [==============================] - 14s 2s/step
10/10 [==============================] - 19s 2s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.945
Train F-1 score:  0.9417989417989417

Test accuracy:  0.8066666666666666
Test F-1 score:  0.8333333333333334

Test accuracy broken down per type
SEP accuracy:  0.97
PPT accuracy:  0.76
NPPT accuracy:  0.69
```





```
[ ]: save_modelkeras(model_3n,'model_twolocal20reps_norm.h5')
```

```
Keras model saved
```

Training time 24 minutes

4.3.2 StrongEntanglingLayers

```
[ ]: nqubits=5
dev = qml.device("lightning.qubit", wires=nqubits)

# number of repetitions that we want in each instance of the variational form
nreps = 8
#  dimensions of the input that the variational form expects
weights_dim = qml.StronglyEntanglingLayers.shape(n_layers = nreps, n_wires = nqubits)
#  number of inputs that each instance of the variational form will take
nweights = 3*nreps*nqubits

def qnn_circuit_strong(inputs, theta):
  qml.AmplitudeEmbedding([a for a in inputs], wires=range(nqubits), pad_with=0,␣
 ↪normalize=True)
  # reshape the theta array of parameters to make it fit into the shape that
  # the variational form expects
  theta1 = tf.reshape(theta, weights_dim)
  qml.StronglyEntanglingLayers(weights = theta1, wires = range(nqubits))

  return qml.expval(qml.Hermitian(M, wires = [0]))

# dictionary we would send to TensorFlow when constructing the Keras layer
weights_strong = {"theta": nweights}
```

```
[ ]: # Model 4 five fold cross validation
     fivefoldCV_qnn(qnn_circuit_strong,xs_train, y_train)
```

```
------------------------------------------------------------------------
Training for fold 1 ...

Score for fold 1: loss of 0.5199374013200586; accuracy of 0.775
------------------------------------------------------------------------
Training for fold 2 ...

Score for fold 2: loss of 0.5743251901298562; accuracy of 0.675
------------------------------------------------------------------------
Training for fold 3 ...

Score for fold 3: loss of 0.6113768385617595; accuracy of 0.65
------------------------------------------------------------------------
Training for fold 4 ...

Score for fold 4: loss of 0.49767407140878933; accuracy of 0.725
------------------------------------------------------------------------
Training for fold 5 ...

Score for fold 5: loss of 0.6144068848443738; accuracy of 0.675
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.5199374013200586 - Accuracy: 0.775
------------------------------------------------------------------------
> Fold 2 - Loss: 0.5743251901298562 - Accuracy: 0.675
------------------------------------------------------------------------
> Fold 3 - Loss: 0.6113768385617595 - Accuracy: 0.65
------------------------------------------------------------------------
> Fold 4 - Loss: 0.49767407140878933 - Accuracy: 0.725
------------------------------------------------------------------------
> Fold 5 - Loss: 0.6144068848443738 - Accuracy: 0.675
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5635440772529675
> Accuracy: 0.7 (+- 0.044721359549995579)
------------------------------------------------------------------------
```

```
[ ]: method= "adjoint"

     tf.random.set_seed(seed)

     qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

     # Keras layer containing qnn
     qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

     # keras model
```

```
model_4 = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)

# binary cross entropy loss, because we are training a binary classifier
model_4.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: # Training our model
     history = model_4.fit(xs_train, y_train, epochs = 50, shuffle = True,
                           validation_data = None, batch_size = 20)
```

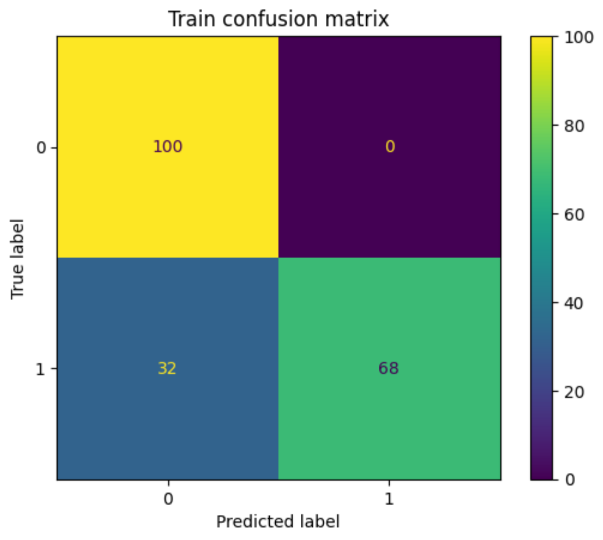
```
Epoch 1/50
10/10 [==============================] - 44s 5s/step - loss: 0.7748
Epoch 2/50
10/10 [==============================] - 60s 6s/step - loss: 0.7602
.
.
.
Epoch 49/50
10/10 [==============================] - 29s 3s/step - loss: 0.4824
Epoch 50/50
10/10 [==============================] - 33s 3s/step - loss: 0.4819
```

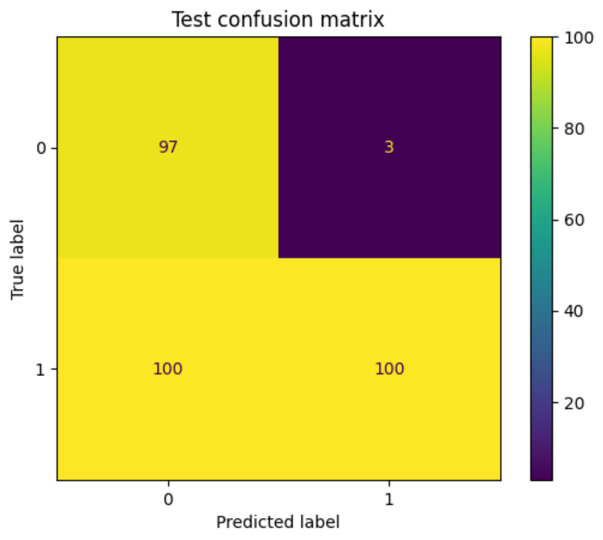
```
[ ]: y_train_pred=model_4.predict(xs_train) >= 0.5
     y_test_pred=model_4.predict(xs_test) >= 0.5
```

```
7/7 [==============================] - 15s 2s/step
10/10 [==============================] - 20s 2s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.84
Train F-1 score:  0.8095238095238095

Test accuracy:  0.6566666666666666
Test F-1 score:  0.6600660066006601

Test accuracy broken down per type
SEP accuracy:  0.97
PPT accuracy:  0.5
NPPT accuracy:  0.5
```





five fold cross validation 2h

28 min training full training set

Normalized data

```
[ ]: # Model 4n five fold cross validation (normalized data)
     fivefoldCV_qnn(qnn_circuit_strong,xs_train_norm, y_train)
```

```
------------------------------------------------------------------------
Training for fold 1 ...
```

```
Score for fold 1: loss of 0.5191338239214546; accuracy of 0.775
------------------------------------------------------------------------
Training for fold 2 ...

Score for fold 2: loss of 0.575823766732432; accuracy of 0.7
------------------------------------------------------------------------
Training for fold 3 ...

Score for fold 3: loss of 0.5508142099930897; accuracy of 0.75
------------------------------------------------------------------------
Training for fold 4 ...

Score for fold 4: loss of 0.5358043361919507; accuracy of 0.75
------------------------------------------------------------------------
Training for fold 5 ...

Score for fold 5: loss of 0.5572398545035854; accuracy of 0.8
------------------------------------------------------------------------
Score per fold
------------------------------------------------------------------------
> Fold 1 - Loss: 0.5191338239214546 - Accuracy: 0.775
------------------------------------------------------------------------
> Fold 2 - Loss: 0.575823766732432 - Accuracy: 0.7
------------------------------------------------------------------------
> Fold 3 - Loss: 0.5508142099930897 - Accuracy: 0.75
------------------------------------------------------------------------
> Fold 4 - Loss: 0.5358043361919507 - Accuracy: 0.75
------------------------------------------------------------------------
> Fold 5 - Loss: 0.5572398545035854 - Accuracy: 0.8
------------------------------------------------------------------------
Average scores for all folds:
> Loss: 0.5477631982685025
> Accuracy: 0.7550000000000001 (+- 0.033166247903554026)
------------------------------------------------------------------------
```

[ ]:
```
method= "adjoint"

tf.random.set_seed(seed)

qnn_strong = qml.QNode(qnn_circuit_strong, dev, interface="tf", diff_method=method)

# Keras layer containing qnn
qlayer=qml.qnn.KerasLayer(qnn_strong, weights_strong, output_dim=1)

# keras model
model_4n = tf.keras.models.Sequential([qlayer])

# we choose adam optimizer with a learning rate of 0.005
opt = tf.keras.optimizers.Adam(learning_rate=0.005)
```

```
# binary cross entropy loss, because we are training a binary classifier
model_4n.compile(opt, loss=tf.keras.losses.BinaryCrossentropy())
```

```
[ ]: history = model_4n.fit(xs_train_norm, y_train, epochs = 50, shuffle = True,
                           validation_data = None, batch_size = 20)
```

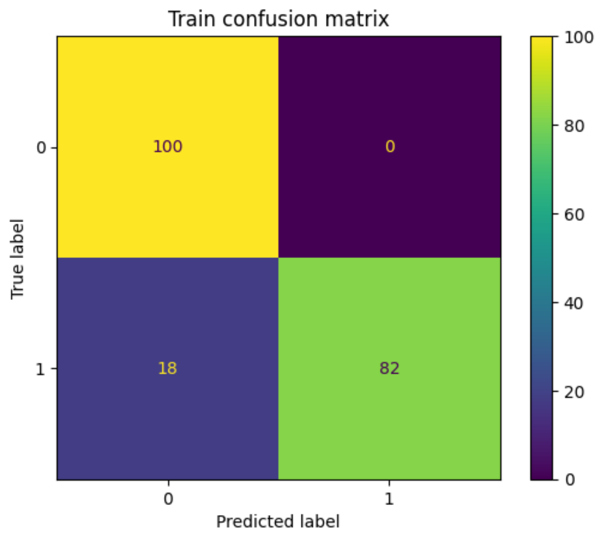
```
Epoch 1/50
10/10 [==============================] - 31s 3s/step - loss: 0.7059
Epoch 2/50
10/10 [==============================] - 34s 4s/step - loss: 0.6890
.
.
.
Epoch 49/50
10/10 [==============================] - 33s 3s/step - loss: 0.4639
Epoch 50/50
10/10 [==============================] - 30s 3s/step - loss: 0.4631
```

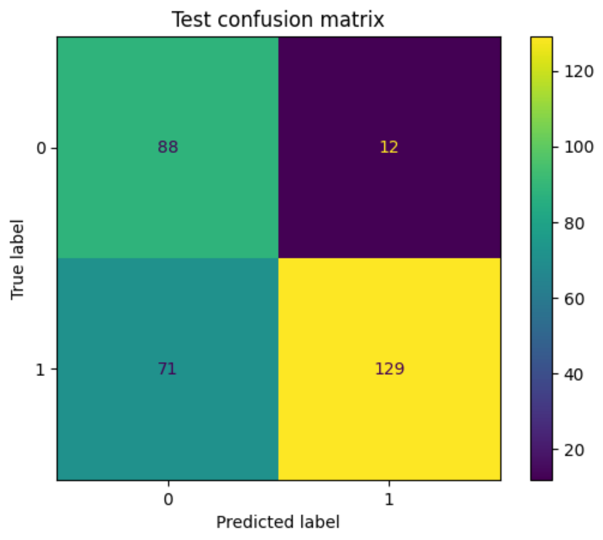
```
[ ]: y_train_pred=model_4n.predict(xs_train_norm) >= 0.5
     y_test_pred=model_4n.predict(xs_test_norm) >= 0.5
```

```
7/7 [==============================] - 13s 2s/step
10/10 [==============================] - 21s 2s/step
```

```
[ ]: performance(y_train_pred, y_train, y_test_pred, y_test_small)
```

```
Train accuracy:  0.91
Train F-1 score:  0.9010989010989011

Test accuracy:  0.7233333333333334
Test F-1 score:  0.7565982404692083

Test accuracy broken down per type
SEP accuracy:  0.88
PPT accuracy:  0.69
NPPT accuracy:  0.6
```





```
[ ]: save_modelkeras(model_4n,'model_strongentangling8reps_norm.h5')
```

```
Keras model saved
```

Five fold cross validation 2h

Training time full training set 24 min

Bibliography

[A$^+$18] Charu C Aggarwal et al. *Neural networks and deep learning*, volumen 10. Springer, 2018.

[AAES$^+$23] Sajid Ali, Tamer Abuhmed, Shaker El-Sappagh, Khan Muhammad, Jose M Alonso-Moral, Roberto Confalonieri, Riccardo Guidotti, Javier Del Ser, Natalia Díaz-Rodríguez, y Francisco Herrera. Explainable artificial intelligence (xai): What we know and what is left to attain trustworthy artificial intelligence. *Information Fusion*, 99:101805, 2023.

[Bar93] Andrew R Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information theory*, 39(3):930–945, 1993.

[BIS$^+$18] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M Sohaib Alam, Guillermo Alonso-Linaje, B AkashNarayanan, Ali Asadi, et al. Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*, 2018.

[BP02] Heinz-Peter Breuer y Francesco Petruccione. *The theory of open quantum systems*. Oxford University Press, USA, 2002.

[Bre11] Haim Brezis. *Functional analysis, Sobolev spaces and partial differential equations*, volumen 2. Springer, 2011.

[CDMAK23] Balthazar Casalé, Giuseppe Di Molfetta, Sandrine Anthoine, y Hachem Kadri. Large-scale quantum separability through a reproducible machine learning lens. *arXiv preprint arXiv:2306.09444*, 2023.

[CGCDM23] E Combarro, S Gonzalez-Castillo, y A Di Meglio. *A Practical Guide to Quantum Machine Learning and Quantum Optimization: Hands-on Approach to Modern Quantum Algorithms*. Packt Publishing, 2023.

[Con19] John B Conway. *A course in functional analysis*, volumen 96. Springer, 2019.

[Fey82] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, (21):467–488, 1982.

[Gar22] José Dueñas García. Puntos excepcionales en sistemas cuánticos markovianos. aplicaciones en termodinámica cuántica, 2022. Bacherlor's Thesis, Universidad de Granada.

[Gér22] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

[GGKC21] Benyamin Ghojogh, Ali Ghodsi, Fakhri Karray, y Mark Crowley. Reproducing kernel hilbert space, mercer's theorem, eigenfunctions, nyström method, and use of kernels in machine learning: Tutorial and survey, 2021.

[GLH15] Salvador García, Julián Luengo, y Francisco Herrera. *Data preprocessing in data mining*, volumen 72. Springer, 2015.

[GT09] Otfried Gühne y Géza Tóth. Entanglement detection. *Physics Reports*, 474(1-6):1–75, 2009.

[Gur03] Leonid Gurvits. Classical deterministic complexity of edmonds' problem and quantum entanglement, 2003.

[Heb05] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology press, 2005.

[Ill22] José Ignacio Illana. Quantum mechanics, 2022. Universidad de Granada.

[Jun22] A. Jung. *Machine Learning: The Basics*. Machine Learning: Foundations, Methodologies, and Applications. Springer Nature Singapore, 2022.

[Kay18] Alastair Kay. Tutorial on the quantikz package, 2018.

[Lid19] Daniel A Lidar. Lecture notes on the theory of open quantum systems. *arXiv preprint arXiv:1902.00967*, 2019.

[Med23] María Medina. Apuntes de análisis funcional, 2023. Universidad Autónoma de Madrid.

[MHH96] Paweł Horodecki Michał Horodecki y Ryszard Horodecki. Separability of mixed states: necessary and sufficient conditions. *Physics Letters A*, 223(1):1–8, 1996.

[MW$^+$22] Chao Ma, Lei Wu, et al. The barron space and the flow-induced function spaces for neural network models. *Constructive Approximation*, 55(1):369–406, 2022.

[NC10] Michael A. Nielsen y Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

[Nie15] Michael A Nielsen. *Neural networks and deep learning*, volumen 25. Determination press San Francisco, CA, USA, 2015.

[OALCP23] María Laura Olivera-Atencio, Lucas Lamata, y Jesús Casado-Pascual. Benefits of open quantum systems for quantum machine learning. *arXiv preprint arXiv:2308.02837*, 2023.

[Oce21] José A. Alvarez Ocete. De novo genome assembly using quantum annealing, sep 2021. Bacherlor's Thesis, Universidad de Granada.

[Pas22] Davide Pastorello. *Concise Guide to Quantum Machine Learning*. Springer Nature, 2022.

[Pay] Rafael Payá. Apuntes de análisis funcional. Universidad de Granada.

[PYF21] Elija Perrier, Akram Youssry, y Chris Ferrie. Qdataset: Quantum datasets for machine learning, 8 2021.

[RL18] R. Feynman R. Leighton. *Surely you're joking, Mr. Feynman*. WW Norton & Co, 2018.

[Ros58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[Rud12] Cynthia Rudin. Prediction: Machine learning and statistics (mit 15.097), lecture 13: kernels, 2012. Massachusetts Institute of Technology.

[SACC21] Louis Schatzki, Andrew Arrasmith, Patrick J. Coles, y M. Cerezo. Entangled datasets for quantum machine learning, 2021.

[SBG$^+$19] Maria Schuld, Ville Bergholm, Christian Gogolin, Josh Izaac, y Nathan Killoran. Evaluating analytic gradients on quantum hardware. *Physical Review A*, 99(3):032331, 2019.

[Sch16] W. Scherer. *Mathematics of Quantum Computing An Introduction*. Springer Cham, 2016.

[SK19] Maria Schuld y Nathan Killoran. Quantum machine learning in feature hilbert spaces. *Physical review letters*, 122(4):040504, 2019.

[sl] scikit learn. 3.3. metrics and scoring: quantifying the quality of predictions. `https://scikit-learn.org/stable/modules/model_evaluation.html#classification-metrics`. Accessed: 30th October 2023.

[SP18] Maria Schuld y Francesco Petruccione. *Supervised learning with quantum computers*, volumen 17. Springer, 2018.

[SP21] Maria Schuld y Francesco Petruccione. *Machine learning with quantum computers*. Springer, 2021.

[USBM23] Julio Ureña, Antonio Sojo, Juani Bermejo, y Daniel Manzano. Entanglement detection with classical deep neural networks. *arXiv preprint arXiv:2304.05946*, 2023.

[Wei20] E Weinan. Machine learning and computational mathematics. *arXiv preprint arXiv:2009.14596*, 2020.

[WM97] D.H. Wolpert y W.G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[WMWW20] E Weinan, Chao Ma, Stephan Wojtowytsch, y Lei Wu. Towards a mathematical understanding of neural network-based machine learning: what we know and what we don't. *arXiv preprint arXiv:2009.10713*, 2020.

[YSAML12] M. Magdon-Ismail Y. S. Abu-Mostafa y H.-T. Lin. *Learning from data*, volumen 4. AMLBook, 2012.