



**UNIVERSIDAD  
DE GRANADA**

---

Facultad de Ciencias

GRADO EN FÍSICA

TRABAJO FIN DE GRADO

**Deep Learning in  
Classical and Quantum  
Neural Networks**

Presentado por:

**D. Eduardo Pérez Álvarez**

Curso Académico 2019/2020

## **Resumen**

Este trabajo trata sobre la combinación de inteligencia artificial y computación cuántica. En él se desarrollan dos redes neuronales: un modelo clásico y otro cuántico. Ambos son aplicados a una tarea de ajuste de curvas con el objetivo de compararlos. Finalmente, se hace uso de otra red neuronal para detectar entrelazamiento cuántico en sistemas de dos qubits.

## **Abstract**

This work is about combining artificial intelligence and quantum computing. Specifically, it develops a classical and a quantum neural network. Both are applied to a curve fitting task in order to perform a comparison. Finally, another neural network is used to detect quantum entanglement in two qubits systems.

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction.</b>  | <b>4</b>  |
| <b>2</b> | <b>Artificial Intelligence (AI).</b>  | <b>4</b>  |
| 2.1      | Historical context. . . . .   | 5         |
| 2.2      | Machine learning. . . . .   | 5         |
| 2.3      | Neural networks and deep learning. . . . .  | 6         |
| 2.4      | Perceptrons. . . . .  | 6         |
| 2.4.1    | Single-layer perceptron, activation function, gradient descent and loss function. . . . . | 6         |
| 2.4.2    | Stochastic Gradient Descent (SGD). . . . .  | 8         |
| 2.4.3    | Multilayer perceptron (MLP) and backpropagation. . . . .                                  | 8         |
| 2.4.4    | Optimizers and deep learning libraries. . . . .   | 11        |
| 2.5      | Application: curve fitting. . . . .   | 12        |
| <b>3</b> | <b>Quantum Computing.</b>   | <b>19</b> |
| 3.1      | Historical context. . . . .   | 20        |
| 3.2      | Fundamentals of Quantum Computing. . . . .  | 20        |
| 3.2.1    | The qubit. . . . .  | 20        |
| 3.2.2    | Quantum logic gates. . . . .  | 21        |
| 3.3      | Quantum machine learning and quantum neural networks. . . . .                             | 22        |
| 3.4      | Continuous-Variable Quantum Neural Network (CV QNN). . . . .                              | 23        |
| 3.5      | Curve fitting with CV quantum neural network. . . . .                                     | 26        |
| <b>4</b> | <b>Detecting quantum entanglement via neural networks.</b>                                | <b>30</b> |
| 4.1      | Quantum entanglement. . . . .   | 30        |
| 4.2      | Peres-Horodecki criterion. . . . .  | 32        |
| 4.3      | Two-qubits entanglement via neural networks. . . . .                                      | 33        |
| <b>5</b> | <b>Conclusions.</b>   | <b>35</b> |
|          | <b>References</b>   | <b>37</b> |

## 1 Introduction.

The famous Moore's law predicts that the number of transistors on a microchip doubles every two years, i.e., we can expect the speed and capability of our computers to increase every couple of years. Moore's law is being fulfilled from the 60s. However, transistors cannot shrink infinitely, so the difficulties to fulfill the law are increasing. After many years of advancements, quantum computing is presented as the key to continue the technological development in the long term. From an up-to-date point of view, quantum computers have proved that they are able to surpass classical ones in several tasks [1, 2, 3, 4].

On the other hand, artificial intelligence is experiencing a peak within the research and technology fields. Specifically, in the last years, there has been a revolution in machine learning techniques based on neural networks, building the new field of deep learning.

These progress lead naturally to the following question: Could artificial intelligence and quantum computing complement each other? The goal of this project is to combine them and to show how artificial intelligence can make use of quantum computing and vice versa. To do so, this project is organized as follows:

In Section 2, we place deep learning and neural networks within the artificial intelligence field and we explain their fundamentals. Furthermore, we develop a multilayer perceptron (the fundamental neural network in deep learning) by describing its algorithm and implementing it computationally in order to study a curve fitting task.

Section 3 shows the fundamentals of quantum computing and quantum machine learning. We develop and implement the continuous-variable quantum neural network from the recent paper [2] and we apply it to the same curve fitting task in order to compare the obtained results with the classical network ones, highlighting the features in which the quantum model is better than the classical one.

Section 4 deals with detecting entanglement by neural networks. Quantum entanglement plays an important role in quantum computing, however, detecting it with 100% accuracy in high dimensional quantum systems has not yet been achieved. We show that neural networks can detect entanglement.

In the conclusions (Section 5), the main ideas and the obtained results are brought together and the work carried out is valued.

The source codes used in each section can be found at [5]. All the results displayed in this project can be reproduced by means of them.

Finally, it should be added that this project also pursues the goal of encouraging the reader to be interested in artificial intelligence and quantum computing. Making progress in these fields could make many people's lives easier, so it is worth studying.

## 2 Artificial Intelligence (AI).

Artificial intelligence is the ability of a machine or computer system to imitate human intelligence processes, include experiences in its cognitive process, adapt to new information, and develop human-like tasks. The ideal feature of artificial intelligence is its capability to rationalize and take decisions that have the best chance of achieving a determined goal [6, 7].

The applications of artificial intelligence are endless, from playing chess to self-driving

cars or predicting the evolution of a economical system. This project is focused on one of the most important branches in every sector nowadays: machine learning.

## 2.1 Historical context.

The first idea about a “thinking machine” was presented in 1950 in a paper carried out by the English mathematician Alan Turing [8]. He established a test in order to observe the machine’s ability to show signs of intelligent conduct similar to humans. This test is known as the Turing test and it was a sort of game of three players having two humans and one computer. An interrogator (human) is apart from the rest of the players (the computer and the another human). The interrogator’s task is to try to guess which player is human and which one is the computer by making them questions [8, 9].

The field of AI has its origins at Dartmouth College in 1956, where the denomination "Artificial Intelligence" was coined by John McCarthy. From there, this branch had been developed through the years until the end of the 20th century when its progression became much faster [10].

Nowadays, AI is incorporated into plenty of different fields: natural language processing, computer vision, big data, etc. Every field makes progress by their own ways, but yet we are far away to get a “general artificial intelligence” which works similarly to a human, although it is an objective of some of the important research teams [6].

## 2.2 Machine learning.

Machine learning is an application of AI that provides the systems with the capability to use previous experience to improve in performing a task without being explicitly programmed for it. It focuses on algorithms that are able to find and apply patterns in data [10, 11].

Machine learning algorithms can present many differences in their approach, in the type of data they input and output, and in the type of task that they must solve. Generally, they can be divided into [11, 12]:

- **Supervised learning algorithms** make a mathematical model of a data set that contains inputs and their desired outputs. A straightforward example would be a computer programme that, after receiving several pictures of cat and non-cat followed by their respective answer to the picture (e.g. 1 if it is a cat, -1 if it is not), would learn it. Then, if presented related to the same kind of pictures, it would be able to classify them.
- **Unsupervised learning algorithms** take a data set that fits only inputs, and find patterns in the data, like conglomeration or clustering of data points. For instance, in marketing, unsupervised algorithms are used in order to find groups of clients with similar behaviours.
- **Reinforcement machine learning algorithms** work within its environment by producing movements and getting rewards or punishments. For example, this kind of algorithms is used to teach computers how to play a game against a human opponent, such as chess.

This is a general classification. Algorithms can be found falling between supervised and unsupervised learning, or algorithms that combine these ideas with other ones of AI.

In this project, we deal with supervised machine learning. To do so, we will carry out a curve fitting task in sections 2.5 and 3.5, and a classification task in Section 4.3.

### 2.3 Neural networks and deep learning.

Neural Networks (NN) are a kind of algorithms that have revolutionized machine learning. They are general function approximators. This is the reason why they can be applied to many of machine learning scenarios, as many problems can be reduced to learning a complicated mapping from the input to the output domain [13].

A neural network is based on a graph of connected nodes denominated artificial neurons, which is inspired in the topology of a human brain. Every connection, like the synapses in the brain, is able to pass a signal between the neurons. Each artificial neuron receives and processes the signal in order to provide the neurons connected to it with a new signal. To simulate a NN in a computer, neurons receive a real number, and the output of the neurons are computed by a non-linear function of the sum of its inputs.

Deep learning is a powerful set of methods for learning in NN which has shown quite efficient results. Some of them will be explained in this project. Within Deep Learning, the fundamental NN is the multilayer perceptron (MLP). It is the basis of more complex neural networks, such as Convolutional Neural Networks (CNN), which have applications in image and video recognition, medical image analysis, natural language processing, financial time series, etc [14].

### 2.4 Perceptrons.

Multilayer networks learning is based on backpropagation and gradient descent algorithms. We start by explaining what a single-layer perceptron is [15, 16, 17].

#### 2.4.1 Single-layer perceptron, activation function, gradient descent and loss function.

The most simple NN is known as a single-layer perceptron. It has a input layer with  $N$  neurons and an output layer with only one neuron. It takes a input  $\vec{X} = (x_1, \dots, x_N) \in \mathbb{R}^N$ , where every  $x_i$  goes to the output neuron ponderated by the internal weights  $\vec{W} = (w_1, \dots, w_N) \in \mathbb{R}^N$ . Thus, the input of the output neuron is:

$$z = \sum_{i=1}^N w_i x_i + b, \quad (2.1)$$

where  $b$  is equivalent to an extra weight with input  $x = 1$ . It is called bias or threshold. The output neuron processes the received information by the activation function,  $f(z)$ , and gives the output of the network. Figure 2.1 displays the single-layer perceptron diagram:

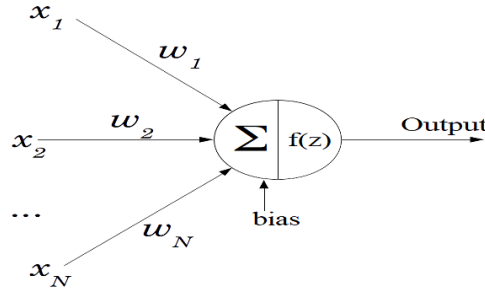


Figure 2.1: Single-layer perceptron diagram. See text.

The learning process consists in adjusting the weights and bias in order to obtain the correct output. The activation function can be simply the Heaviside function,  $\Theta(z)$ , but in the multilayer perceptron, as it will be seen later, it has to be differentiable. The most common activation functions in deep learning are:

$$\text{sigmoid}(z) = \frac{1}{1 + \exp(-z)}. \quad (2.2)$$

$$\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}. \quad (2.3)$$

$$\text{Linear} \quad f(z) = z. \quad (2.4)$$

$$\text{Rectified linear unit} \equiv \text{ReLU}(z) = \max(0, z) = z \Theta(z). \quad (2.5)$$

The last one is non-differentiable at zero. Nevertheless, the value of the derivative in that point can be chosen arbitrarily as 0 or 1.

The weights of a single perceptron are usually trained by using gradient descent, the most popular optimization algorithm in neural networks. This method tries to find a local minimum of a function by taking proportional steps to the negative of the gradient of the function at the current point [15]. The function to minimize is called *loss function*. It represents the exactitude of the results obtained with a determinate set of weights and biases (in single perceptron we only have one bias, but in the MLP there are more). Let  $y^{(k)}$  be the desired output of the input vector  $\vec{X}^{(k)}$  and  $a^{(k)} = f\left(\sum_{i=1}^N w_i x_i^{(k)} + b\right)$  the predicted value, with  $k = 1, \dots, m$ , being  $m$  the number of training examples. Typical loss functions are:

$$\text{Mean Square Error (MSE)} \quad L(a, y) = \frac{1}{m} \sum_{k=1}^m \left(y^{(k)} - a^{(k)}\right)^2. \quad (2.6)$$

$$\text{Cross Entropy Loss} \quad L(a, y) = -\frac{1}{m} \sum_{k=1}^m \left(y^{(k)} \log(a^{(k)}) + (1 - y^{(k)}) \log(1 - a^{(k)})\right). \quad (2.7)$$

The first one is used in regression tasks, while the second one is characteristic of classification tasks. To carry out the gradient descent we have to compute:

$$\nabla_w L = \left(\frac{\partial L}{\partial w_1}, \dots, \frac{\partial L}{\partial w_N}\right) \quad ; \quad \nabla_b L = \left(\frac{\partial L}{\partial b}\right). \quad (2.8)$$

Choosing the mean square error (2.6), every derivative is calculated as:

$$\begin{aligned}\frac{\partial L}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{m} \sum_{k=1}^m \left( y^{(k)} - a^{(k)} \right)^2 = \frac{2}{m} \sum_{k=1}^m \left( y^{(k)} - a^{(k)} \right) \frac{\partial a^{(k)}}{\partial w_i} = \\ &= \frac{2}{m} \sum_{k=1}^m \left( y^{(k)} - a^{(k)} \right) \frac{\partial}{\partial w_i} f \left( \sum_{i=1}^N w_i x_i^{(k)} + b \right).\end{aligned}\quad (2.9)$$

Taking  $z^{(k)} = \sum_{i=1}^N w_i x_i^{(k)} + b$ , using the chain rule, and applying the gradient descent, the weights and bias can be updated by

$$w_i^{new} = w_i^{old} + \Delta w_i; \quad \Delta w_i = -\alpha \frac{\partial L}{\partial w_i} = -\alpha \frac{2}{m} \sum_{k=1}^m \left( y^{(k)} - a^{(k)} \right) f'(z^{(k)}) x_i^{(k)}. \quad (2.10)$$

$$b^{new} = b^{old} + \Delta b; \quad \Delta b = -\alpha \frac{\partial L}{\partial b} = -\alpha \frac{2}{m} \sum_{k=1}^m \left( y^{(k)} - a^{(k)} \right) f'(z^{(k)}). \quad (2.11)$$

Where  $\alpha$  is called learning rate, a parameter that controls the size of the gradient descent steps. Too small  $\alpha$  implies that the algorithm converges very slowly and, if  $\alpha$  is too large, the minimum can be overstepped. The learning rate is usually  $0.0001 < \alpha < 0.5$ . The importance of  $\alpha$  will be discussed later [15].

#### 2.4.2 Stochastic Gradient Descent (SGD).

In multilayer neural networks, SGD is usually used instead of gradient descent, because it is much more efficient. This method considers only the randomly chosen input  $k$  in order to update the weights and biases. Hence, in equations (2.10) and (2.11), updates become to:

$$\Delta w_i = -\alpha \frac{\partial l^{(k)}}{\partial w_i} = -2\alpha \left( y^{(k)} - a^{(k)} \right) f'(z) x_i^{(k)}. \quad \Delta b = -\alpha \frac{\partial l^{(k)}}{\partial b} = -2\alpha \left( y^{(k)} - a^{(k)} \right) f'(z). \quad (2.12)$$

The function that we want to minimize is the mean square error, so we express it by:

$$L(a, y) = \frac{1}{m} \sum_{k=1}^m l^{(k)}; \quad l^{(k)} = \left( y^{(k)} - a^{(k)} \right)^2. \quad (2.13)$$

In some references,  $l$  is called loss function and  $L$  is called cost function, also represented by  $J$ . Also, in some of them, there is a factor  $\frac{1}{2}$  multiplying in the MSE loss function. These aspects will not have any influence on the results [15].

#### 2.4.3 Multilayer perceptron (MLP) and backpropagation.

The single-layer model has two layers. Now the idea is to adjust it to 3 or more layers. The notation will be the following:

$a_i^{(k)}(n)$  will be the output of the  $n$  layer neuron  $i$ .  $k$  will be referred to the input vector  $\vec{X}^{(k)}$ .



$w_{j,i}(n)$  is the weight which connects the neuron  $i$  from the  $n$  layer to the neuron  $j$  from  $n + 1$  layer.

$b_i(n)$  corresponds to the bias of the neuron  $i$  from the layer  $n$ .

$N(n)$  is the numbers of neurons in the  $n$  layer.

The output of  $n + 1$  layer neuron  $j$  when the  $k$  data is introduced to the network is:

$$a_j^{(k)}(n + 1) = f \left( \sum_{i=1}^{N(n)} w_{j,i}(n) a_i^{(k)}(n) + b_j(n + 1) \right). \quad (2.14)$$

MLPs contain 3 types of layers:

- The **input layer** ( $n = 0$ ). It only transmits the data to the network as it was previously seen. So  $a_i^{(k)}(0) = x_i^{(k)}$ . It has as many neurons as the dimension of  $\vec{X}$ . Therefore, if we want to introduce a 2x2 matrix to the network, the input layer has to be formed by 4 neurons.
- The **hidden layers**, which process the received information displayed in Equation (2.14). The number of hidden layers, as the number of neurons in each hidden layer, are network parameters and there is no a general number that maximizes the performance. It mostly depends on each problem complexity, the input dimension, and the number ( $m$ ) of inputs available to train de network.
- The **output layer**, like the hidden ones has an output determined by (2.14), and it gives the final result of the network. In this section, we analyse problems of fitting 1-variable functions. Because of that, the solution will be a real number, so the output layer will have only one neuron.

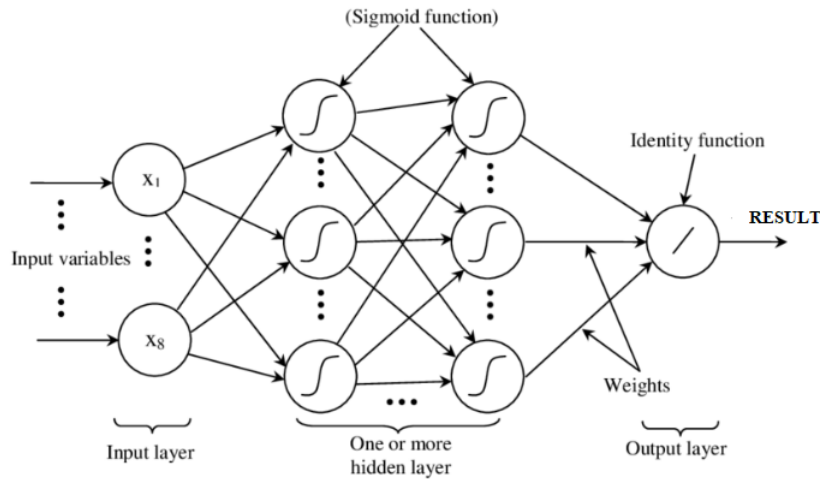


Figure 2.2: Architecture of an example of multilayer perceptron neural network. Note that every neuron also has its own bias or threshold (except the input ones). Picture from [18].

The MLP algorithm is the following:

1. Initialize all weights and biases randomly.
2. Propagate an input  $\vec{X}^{(k)}$  through the network by applying Equation (2.14) recursively. This is called “Forward Propagation”.
3. Evaluate the function  $l^{(k)}$  in order to determine the network accuracy related to the expected value  $y^{(k)}$ .
4. Update weights and biases by using Stochastic Gradient Descent and Backpropagation (more details below).
5. Repeat points 2, 3 and 4 a certain number of times (usually the number of inputs in the training set,  $m$ , times).
6. Evaluate the loss function  $L$ .
7. If  $L$  reaches its minimum value, the network training is completed. Otherwise, go back to point 2.

The weights and biases are updated following Eq. (2.12). If we label the output layer as  $n + 1$  and we define:

$$\delta_j^{(k)}(n+1) = 2 \left( y^{(k)} - a_j^{(k)}(n+1) \right) f' \left( \sum_{i=1}^{N(n)} w_{j,i}(n) a_i^{(k)}(n) + b_j(n+1) \right), \quad (2.15)$$

the updates are:

$$\Delta w_{j,i}(n) = -\alpha \delta_j^{(k)}(n+1) a_i^{(k)}(n). \quad \Delta b_j(n+1) = -\alpha \delta_j^{(k)}(n+1). \quad (2.16)$$

Note that for our fitting problem  $j = 1$ , since there is only one output neuron. However, we keep the general notation for convenience.

In order to update the weights and biases of the hidden layers, the algorithm used is known as Backpropagation (or Backward Propagation):

Taking neuron  $p$  of the  $n - 1$  layer,  $\Delta w_{i,p}(n - 1)$  is given by:

$$\Delta w_{i,p}(n - 1) = -\alpha \frac{\partial l^{(k)}}{\partial w_{i,p}(n - 1)} = -2\alpha \sum_{j=1}^{N(n+1)} \left( y^{(k)} - a_j^{(k)}(n+1) \right) \frac{\partial a_j^{(k)}(n+1)}{\partial w_{i,p}(n - 1)}. \quad (2.17)$$

Applying again the chain rule and taking into account Equation (2.14) :

$$\frac{\partial a_j^{(k)}(n+1)}{\partial w_{i,p}(n - 1)} = f' \left( \sum_{i=1}^{N(n)} w_{j,i}(n) a_i^{(k)}(n) + b_j(n+1) \right) w_{j,i}(n) \frac{\partial a_i^{(k)}(n)}{\partial w_{i,p}(n - 1)}. \quad (2.18)$$

Thus

$$\frac{\partial l^{(k)}}{\partial w_{i,p}(n - 1)} = \sum_{j=1}^{N(n+1)} \delta_j^{(k)}(n+1) w_{j,i}(n) \frac{\partial a_i^{(k)}(n)}{\partial w_{i,p}(n - 1)}. \quad (2.19)$$

Now, repeating Equation (2.18) for  $\frac{\partial a_i^{(k)}(n)}{\partial w_{i,p}(n-1)}$  instead of  $\frac{\partial a_j^{(k)}(n+1)}{\partial w_{i,p}(n-1)}$ , and defining

$$\delta_i^{(k)}(n) = f' \left( \sum_{p=1}^{N(n-1)} w_{i,p}(n-1) a_p^{(k)}(n-1) + b_i(n) \right) \sum_{j=1}^{N(n+1)} \delta_j^{(k)}(n+1) w_{j,i}(n), \quad (2.20)$$

the updated weights result

$$\begin{aligned} \frac{\partial l^{(k)}}{\partial w_{i,p}(n-1)} &= \delta_i^{(k)}(n) a_p^{(k)}(n-1) \implies \\ \implies w_{i,p}^{new}(n-1) &= w_{i,p}^{old}(n-1) - \alpha \delta_i^{(k)}(n) a_p^{(k)}(n-1). \end{aligned} \quad (2.21)$$

Computing  $\frac{\partial l^{(k)}}{\partial b_i(n)}$  in the same way, the updated biases are obtained:

$$\frac{\partial l^{(k)}}{\partial b_i(n)} = \delta_i^{(k)}(n) \implies b_i^{new}(n) = b_i^{old}(n) - \alpha \delta_i^{(k)}(n). \quad (2.22)$$

Finally, calculating  $\Delta w_{p,t}(n-2)$  and  $\Delta b_p(n-1)$  would be possible by:

$$\delta_p^{(k)} = f' \left( \sum_{t=1}^{N(n-2)} w_{p,t}(n-2) a_t^{(k)}(n-2) + b_p(n-1) \right) \sum_{i=1}^{N(n)} \delta_i^{(k)}(n) w_{i,p}(n-1). \quad (2.23)$$

And this would be the process to follow for every layer [15].

#### 2.4.4 Optimizers and deep learning libraries.

Until now, we have studied Backpropagation and Stochastic Gradient Descent in order to understand how neural networks work. However, within deep learning, a lot of techniques to improve the neural networks efficiency have been developed. They are known as optimization algorithms (or optimizers) and an important part of the deep learning field is focused on the study of these techniques. There are many papers and complete books where all of these techniques are developed [12, 14]. Some of the most popular optimizers are *Adam Optimizer* [19], *Dropout Regularization* [20] (patented by Google) and *Xavier Initialization* [21]. In this project, we do not analyse the different optimization algorithms because it would mean a too large extension in this section. Notwithstanding, in the Subsection 2.5 the importance of these algorithms will be discussed and a quite simple one will be implemented. Within the sections 3 and 4 several optimizers will be applied by means of the deep learning libraries TensorFlow and Keras.

There are many free and open-source software libraries where a deep neural network can be implemented in a few lines of code source, with as many different optimizers, neurons, and layers as are required. Some of the best deep learning frameworks are TensorFlow (developed by Google), Microsoft Cognitive Toolkit, Theano, Keras (it is capable of running on top of the previous ones), Pytorch and Caffe. There are also quantum deep learning libraries as TensorFlow Quantum (TFQ) and Qiskit (developed by IBM). They base their algorithms on quantum circuits. We will develop this within Section 3 and we will use the python library Strawberry Fields in order to study a quantum neural network.

## 2.5 Application: curve fitting.

A common application of neural networks is curve fitting: learning the established relation between an input set and its respective outputs. In this section, curve fitting is dealt with by a neural network developed by myself. The obtained results are analysed according to some network parameters such as the number of layers, learning rate, etc.

The test function is  $\sin(\pi x)$  modified by noise, in one cycle,  $(-1,1)$ . The training data is a set of  $m$  inputs  $x^{(k)} \in (-1,1)$  and their respective outputs  $y^{(k)} \equiv y(x^{(k)}) = \sin(\pi x^{(k)}) + \varepsilon(x^{(k)})$ , being  $\varepsilon(x^{(k)})$  random uniform numbers between -0.1 and 0.1. The goal of the network is to learn the function  $\sin(\pi x) \forall x \in (-1,1)$ . Note that the lower is the noise, the easier is the task. However, in real applications, there is always a stochastic factor, which is included in this kind of studies through the noise. When there is no noise, any function can be approximated with arbitrary precision by a neural network [22].

The used source code is in [5]. It is a neural network coded in C++ from scratch, where the described algorithms in Section 2.4 are developed. No specific machine learning module or framework is used.

Since the function to fit is a real function of real variable ( $f(x), x \in \mathbb{R}$ ), the input and output layers have only one neuron. The chosen activation function in hidden layers is a sigmoid, and in the output one it is a linear function ( $f(x) = x$ ). The loss function to minimize is:

$$L(a, y) = \frac{1}{m} \sum_{k=1}^m l^{(k)}(a^{(k)}, y^{(k)}) ; \quad l^{(k)}(a^{(k)}, y^{(k)}) = (y^{(k)} - a^{(k)})^2. \quad (2.24)$$

Where  $a^{(k)}$  is the network output for the input  $x^{(k)}$ . Weights and biases are randomly initialized in the range  $[-0.005, 0.005]$ .

In order to speed up the learning, I have implemented an optimizer. It consists in adding a second factor, so called *momentum*, when weights and biases are updated. This factor takes into account the change in the previous iteration. In this way, the update results:

$$w_{j,i}^{new}(n) = w_{j,i}^{old}(n) - \alpha \frac{\partial l^{(k)}}{\partial w_{j,i}} + \beta (w_{j,i}^{old}(n) - w_{j,i}^{old-old}(n)). \quad (2.25)$$

$$b_i^{new}(n) = b_i^{old}(n) - \alpha \frac{\partial l^{(k)}}{\partial b_i} + \beta (b_i^{old}(n) - b_i^{old-old}(n)). \quad (2.26)$$

Where  $\beta$  is the momentum rate, and it controls the relevance of the momentum factor. It may always be  $0 \leq \beta < 1$ , since  $\beta > 1$  can produce divergences. Common values are between 0.7 and 0.99 [15]. On the other hand,  $\alpha$  is the usual learning rate. As it was described at the end of Section 2.4.1, it is a parameter that regulates how much we are adjusting the weights and biases of the network according to the loss function gradient. The network has worked with  $\alpha = 0.001$  and  $\beta = 0.8$ . Later, in order to show their importance, their parameter-space will be analysed

In the simulations of the following sections, the best results are looked for by changing some parameters while the rest remain fixed. All the graphics are built using the program *Gnuplot*, version 5.2.7.

### Network Architecture.

The first thing to take into account is that the number of neurons and layers depends on the complexity of the problem to solve. Fitting a linear function requires much less neurons than an oscillation function, and more neurons would be needed in order to fit a 3 variables function. Furthermore, fitting the function  $\sin(\pi x)$  in  $(-1, 1)$  is not the same thing as doing it within a larger interval.

On the other hand, more neurons and, mainly, more layers implies an increase in the computational cost. Fitting  $\sin(\pi x)$  can be done with a lot of architectures with the same precision, but some of them take longer than others.

In order to study how the function is fitted by neural networks with different number of neurons and layers, I have prepared 50 points equally separated. Figure 2.3 displays the learned function by every model. All models have been trained the same number of cycles.

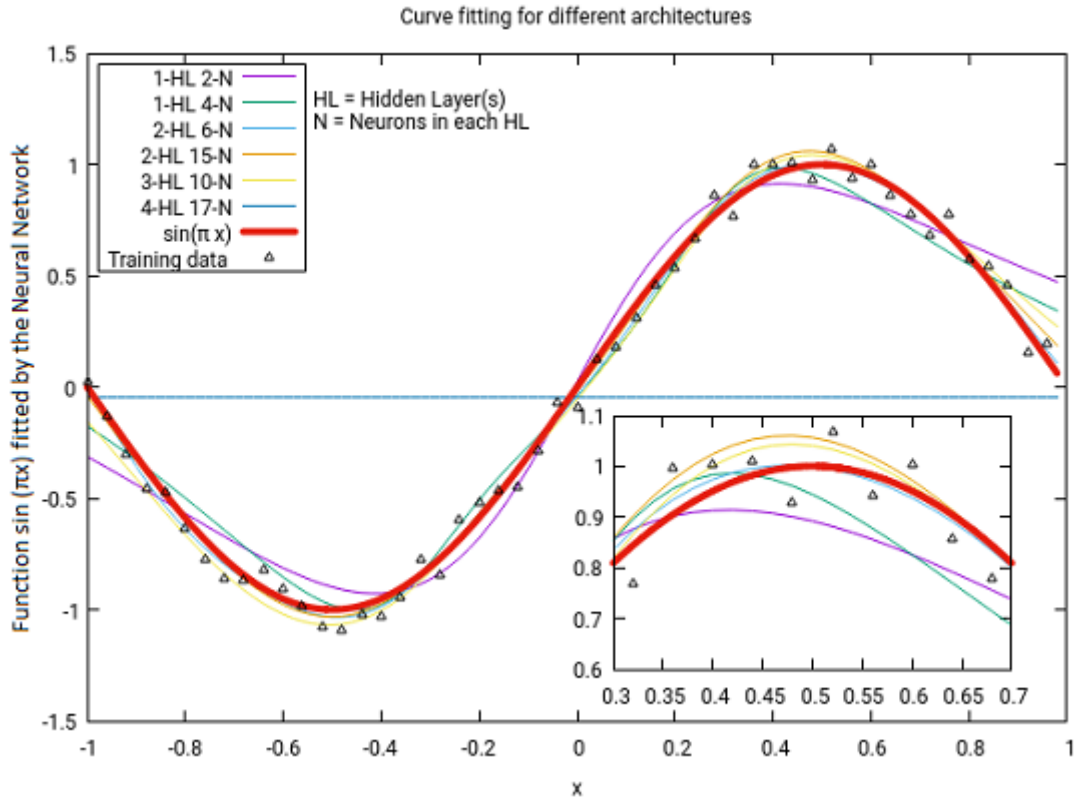


Figure 2.3: Performance of neural networks with different number of hidden layers and neurons. The training has been performed by 50 points belonging to the function  $y(x) = \sin(\pi x) + \varepsilon(x)$  in the range  $(-1, 1)$ , being  $\varepsilon(x)$  a different random number between  $-0.1$  and  $0.1$  for every  $x$ . The learning rate and momentum rate are respectively  $\alpha = 0.001$  and  $\beta = 0.8$ . In the inset, we can observe how the models with more than one hidden layer learn the curvature of the function better than the models with only one hidden layer.

The model with one hidden layer and 2 neurons has not got enough parameters to fit the function. If the number of neurons is increased to 4, the model has the double of parameters to adjust the function and the result is better. However, it is not good enough.

The rest of the models present correct results except the 4 hidden layers one, that remains as a constant function. This model has not been trained long enough. We can understand it better by looking at Figure 2.4, where the evolution of each model can be observed. Every epoch implies that  $m$  inputs ( $m = 50$  in this case) have been propagated through the network. Hence, one epoch is equivalent to  $m$  updates of weights and biases.

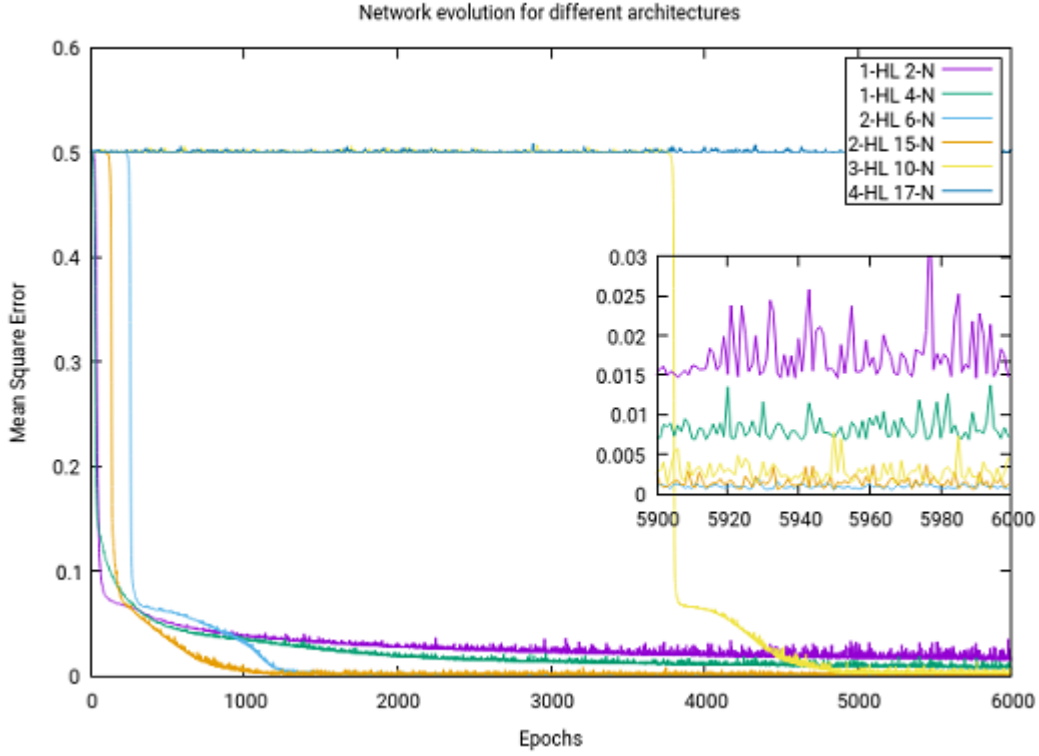


Figure 2.4: Mean square error as a function of epochs for each model from Figure 2.3.  $m$  updates of weights and biases are carried out in each epoch.

The graphic shows that one hidden layer models are quickly trained, but their results are worse. Generally, more neurons means more training time (it does not happen always; in fact, 2HL 6N model takes longer than 2HL 15N), but the computational cost is incremented mostly by the number of layers. A good example of this can be observed with the 2 hidden layers and 15 neurons model. It has the same number of neurons that 3 hidden layers and 10 neurons model, but the second one takes longer than the first one. However, results are similar. Lastly, note that the 4 hidden layers model is not a mistaken model. Seeing good results in it would be possible after 10.000 epochs.

As it has been seen in both figures, the best model to solve this problem after 6000 epochs is the 2 hidden layers and 6 neurons one. Note that its error in the inset of Figure 2.4 is about 0.001, which is mostly a consequence of the squared error between the points of  $\sin(\pi x)$  and the noisy points.

On the other hand, if we studied a more complex problem, we would probably obtain the best result by the 4 hidden layers model, since it has more neurons and layers to learn.

### Overfitting.

In machine learning, the goal is getting a model that, after the training, gets to solve situations that differ from the training ones. However, models that go through longer trainings can turn out the learning algorithm focused in too specific features [23]. In curve fitting, we can observe overfitting when the error's model is smaller with respect to the noisy points than the points of the function to fit. The deeper is the network, the more common is the issue. In this section, we study some overfitted models and how to avoid this problem. First, overfitting is analysed according to the number of data examples. After that, we study how to control the training time in order to evade this problem.

The best way to avoid overfitting is by increasing the training data. Figure 2.5 below shows a model with 4 hidden layers and 25 neurons in each one trained within different dataset, after  $5 \cdot 10^4$  epochs.

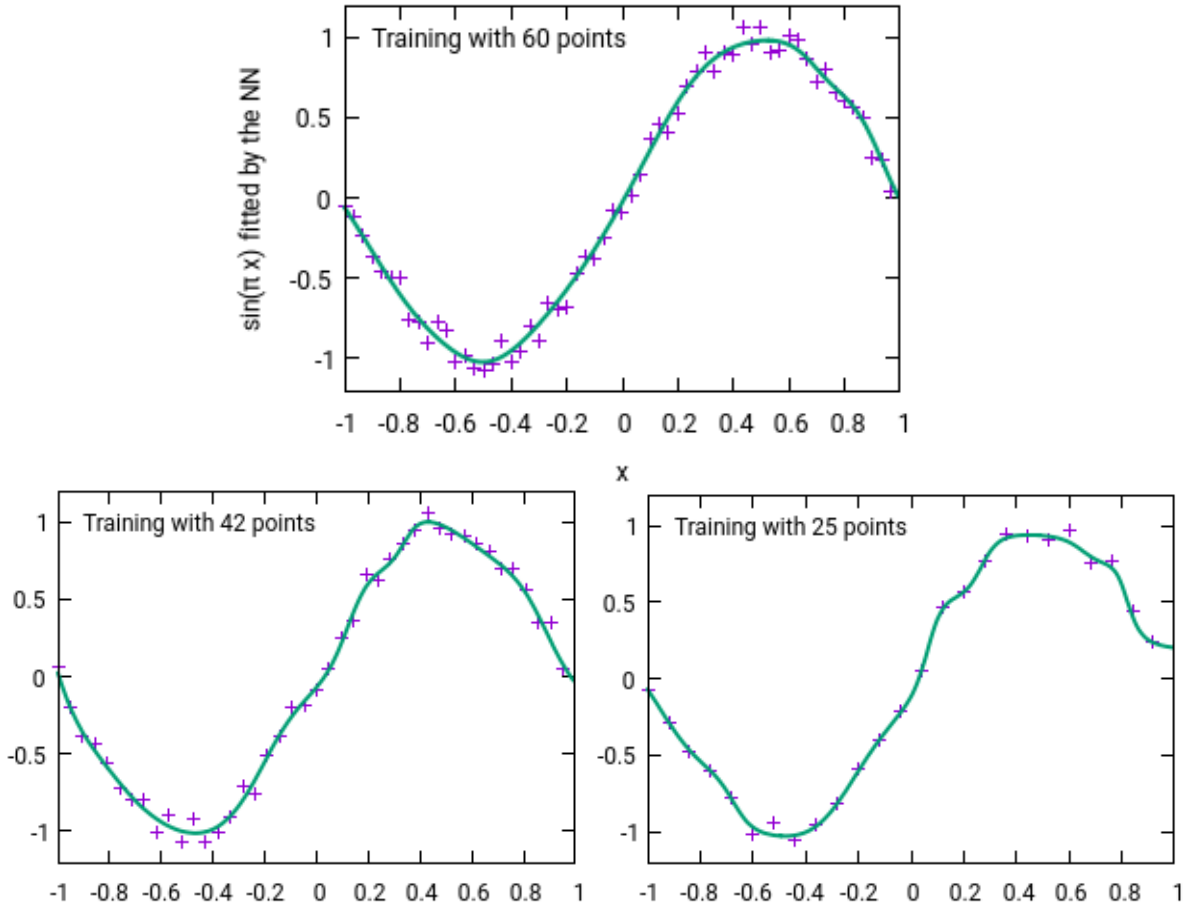


Figure 2.5: Neural network with 4 hidden layers and 25 neurons in each one trained with different sizes of data set, after  $5 \cdot 10^4$  epochs.

A low number of points implies that the network parameters can easily fit the points rather than fitting the trend. In the 25 points graphic, the fitted curve takes too many different curvatures in order to pass through the training points. This behaviour is reduced when the size of the training set is increased to 42 points. Finally, overfitting is practically avoided with 60 data points.



However, if a network has more parameters than needed to adjust the data and we train it too long, the model will undergo overfitting. This will happen even if we have a big number of data to train. Given this situation, training time is very important. Figure 2.6 displays a model with 4 layers and 35 neurons in each layer trained an optimal time and trained a longer time. The number of training points is 60.

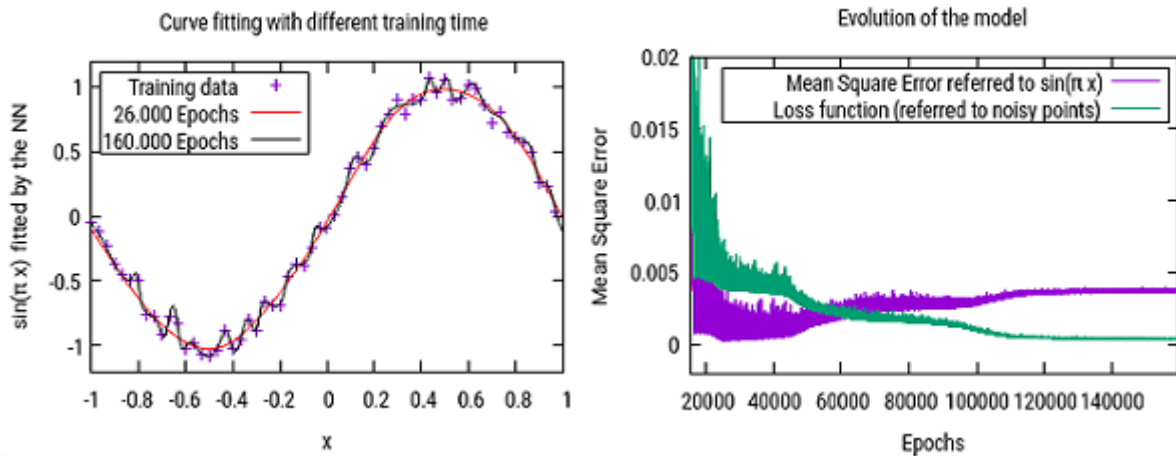


Figure 2.6: An optimal fitted and an overfitted curve got by the same model. Right graphic shows that the best results are between  $2 \cdot 10^4$  and  $4 \cdot 10^4$  epochs. Red curve on the left graphic corresponds with that interval. The black one has trained so much that it is totally overfitted.

The function  $\sin(\pi x)$  and the red curve have an error lower than 0.001 between them. The black curve is the result of training too long. Four layers with 35 neurons in each one provide more parameters than needed in order to fit  $\sin(\pi x)$ . Thus, if the model is trained a considerable time, it will be able to get overfitted and it will pass through all the points. As we can see in the right graphic, this implies that the loss function is practically zero. However, the error referred to the function to minimize is bigger. In the same graphic, we can also observe that the results become worse due to overfitting after around 60.000 epochs.

Models with many neurons and layers can get better results, but it is needed to be very careful with the training time and overfitting. Some optimization techniques, as *Dropout Optimization* [20], can help to reduce overfitting in a very efficient way.

It should be mentioned that the loss function had been said to be the function to minimize. That is right, but in the event of a problem with a stochastic factor, noise avoids it to be completely correct. As it was seen, the optimal loss function value gets closer to the minimum and avoids overfitting. Within a classification task, such as cat and non-cat, it would be different and the minimum would be searched in the loss function.

### Learning rate.

One of the most important parameters in neural networks is the learning rate,  $\alpha$ . As it was previously commented, the learning rate controls the size of the gradient descent steps. It can be better understood by Figure 2.7 :



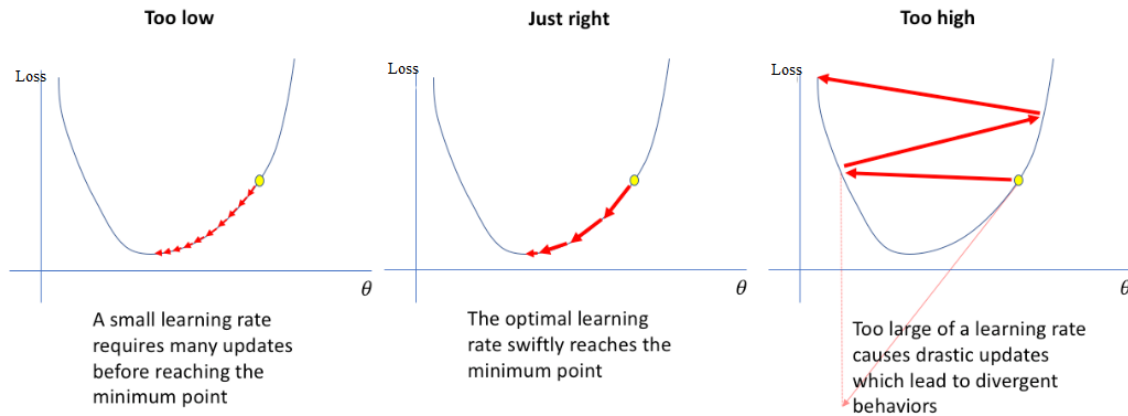


Figure 2.7: Gradient descent steps according to the learning rate.  $\theta$  represents the network weights and biases, which are updated in order to find the minimum in the loss function. Picture from [24].

Too small  $\alpha$  implies that the algorithm converges too slowly. On the other hand, if  $\alpha$  is too large, the minimum can be overstepped. As it was said before, the learning rate is usually chosen between  $0.0001 < \alpha < 0.5$ . This is a quite general range since every model has its own features and applications. Up to now, the network has been updating weights and biases taking  $\alpha = 0.001$ . In this section, this value is changed in order to study the importance of the learning rate.

Figure 2.8 shows the evolution of a network with 2 hidden layers and 25 neurons in each one according to different learning rates. The model has been trained until 2000 epochs with 50 data points.

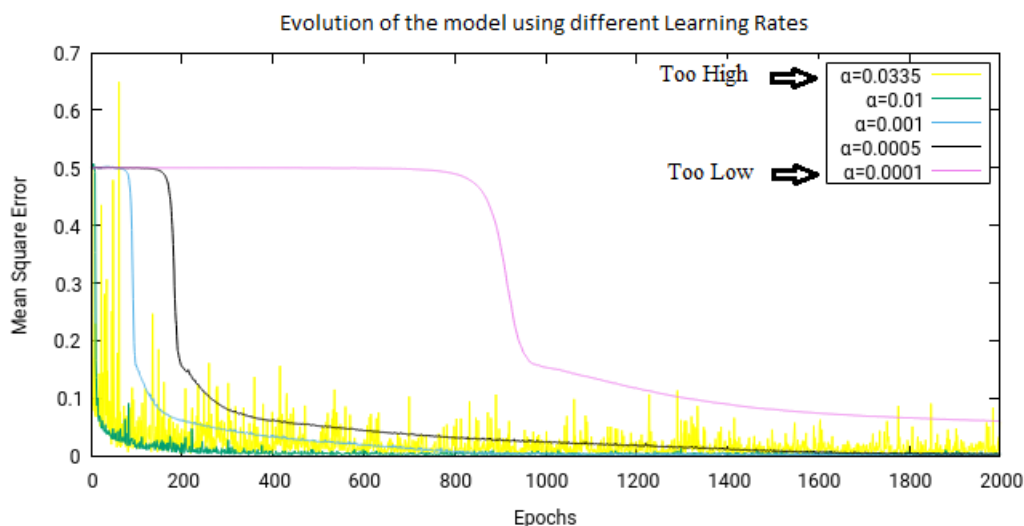


Figure 2.8: Mean square error as a function of epochs for different learning rates.. See text.

Learning rates higher than  $\alpha = 0.0335$  lead to divergent behaviours and the neural network does not fit anything. On the other hand,  $\alpha = 0.0001$  takes too long in comparison with the others. The bigger is the network, the more noticeable will be the temporal difference.

### Optimizers.

As it is described in the equations (2.25) and (2.26), the momentum optimizer speeds up the learning in the network. In Section 2.4.4, we commented that optimization algorithms are very important in deep learning. The deeper the network is, the more important optimizers are. In order to confirm this, the momentum rate  $\beta$  has been changed in two different models, one deeper than the other. The rest of parameters are the same in both networks. Figure 2.9 displays the evolution of each model.

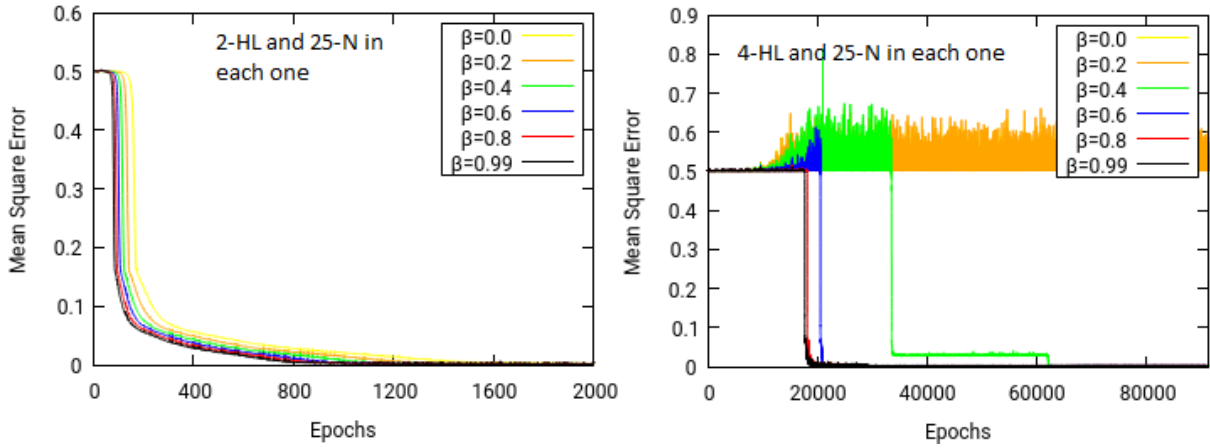


Figure 2.9: Evolution of the models using different Momentum Rates. Right model is deeper, so the optimizer algorithm is more important.

Momentum factor has been modified in the model that had been used in the previous section and in another one with twice its depth. In the 2 hidden layers one, the differences are small, but in the 4 hidden layers one, momentum factor is very important in order to reduce the computational cost. Note that this network would have to train many more epochs in order to get the results with  $\beta = 0.2$  or  $\beta = 0$ . What is more, this is a simple network with a quite simple optimizer. In a model with many layers, a powerful optimizer is fundamental.

One of the most powerful optimization algorithms is *Adam Optimizer* [19]. It combines the momentum idea with some others (*exponentially weighted averages* [25] and *RMSprop* [26]) in order to generate an *Adaptive Learning Rate*, which can be the difference between getting good results in minutes, hours, and days.

### Initialization.

Another important factor in neural networks is the weights and biases initialization. A common issue is the vanishing and exploding gradients. It means that the derivatives become too small or big and the training becomes more complicated. This issue can be often solved by choosing a good initialization. Weights and biases are usually randomly initialized within an interval centered in zero. So every neuron learns in a different way and the network symmetry is broken. In this section, we study a network changing the initializations. Figure 2.10 presents the obtained results with three different initializations in a network with 2 hidden layers and 25 neurons in each one.

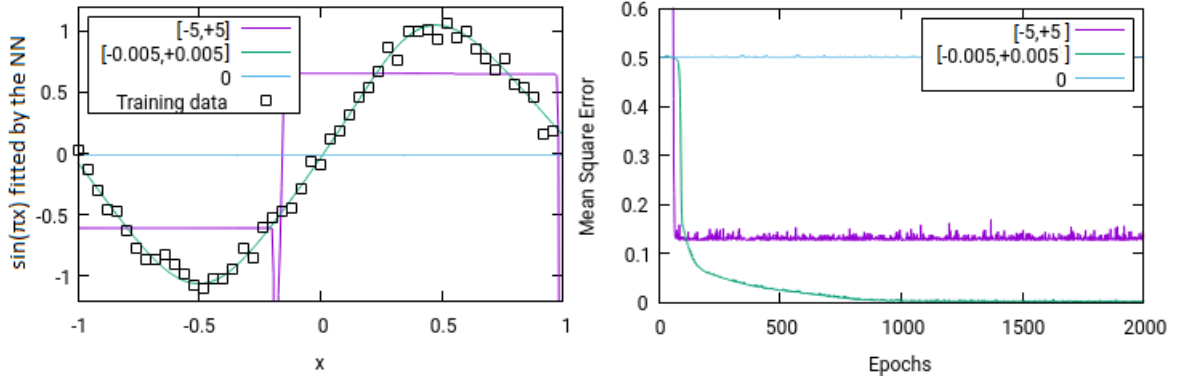


Figure 2.10: Network output (left) and evolution (right) when weights and biases are randomly initialized within different ranges. Blue curves are an example of vanishing gradients. In the violet curve, the gradient has exploded around the point  $x = -0.2$  (left), so the training does not get to converge (right). Finally, green curves correspond with a good performance.

To initialize in zero or in too small values implies that gradients are negligible. When it happens, it is said that the network does not get to break symmetry. On the other hand, too big initializations cause some weights (or biases) to be too different from others, which means sudden changes that would need a long training time to be damped.

### Conclusion.

Taking into account all the content within this section, it could be said that the good performance and efficiency of a neural network are completely conditioned by the network parameters. When a new application is started, it is almost impossible to correctly guess the ideal parameters in order to get good results.

Applying deep learning is a highly iterative process that begins with an idea according to the problem to solve. This idea is implemented, results are obtained and through them the idea is refined. This process is repeated until the best choice of parameters is found.

## 3 Quantum Computing.

Quantum computing basically consists in making use of quantum mechanics principles in order to process information. The functioning of a classical computer is based on “bits” (“binary digits”). This is the basic unit of measurement applied to computer data, and it has two possible values: 0 or 1. Conversely, a quantum computer uses quantum bits or “qubits”. A qubit is a quantum-mechanical system formed by the linear combination of two pure orthogonal states, 0 and 1. Since qubits behave quantumly, they are subjected to quantum phenomenons, such as superposition and entanglement. Thanks to this, a quantum computer computes a great number of operations at the same time. While a traditional computer operates with ones and zeros, the quantum one uses ones, zeros and their linear combinations. This makes possible that certain tasks that take too long for classical computers could be done faster by quantum computers [27].

In this section, we describe what is quantum computing in a way that can be understood by anyone without previous knowledge in this field. Afterwards, we study a neural network developed through quantum computing. Finally, we make use of this quantum network to deal with the same curve fitting task that we have analysed with the classical one.

### 3.1 Historical context.

Richard Feynman first suggested quantum computers in the paper *Simulating Physics with Computers*, published in 1982 [28]. He established that building quantum computers would be needed in order to simulate quantum systems, since tracking every possible configuration in that systems would require thousands of years of computations in a traditional computer. However, building hardware able to base its operations in quantum effects seemed to be impossible at that time. Nevertheless, quantum algorithms began to be developed. The most famous are Deutsch–Jozsa algorithm (1985), Shor’s algorithm (1994), and Grover’s algorithm (1996). These algorithms allows to carry out some tasks much faster than any classical algorithm. This landmark showed the scientist community that quantum computing was a very promising field [29].

Some years later, in 1998, the first 2-qubits machine was developed in the University of Berkeley. Since that moment, more advances have been achieved, such as storing for the first time a qubit in an atomic nucleus. It was carried out by the National Scientist Foundation of USA, and the information remained intact for 1.75 seconds [29].

In 2009, the first state solid quantum processor was developed. Two years later the enterprise D-Wave Systems sold the first quantum annealer. After that, D-Wave Systems and other companies such as IBM, Google, and Intel have been progressing. On October 23, 2019, Google announced that they had achieved quantum supremacy, getting their 53-qubits (they were 54, but one failed) computer Sycamore to carry out in 200 seconds a task which would have taken about 10.000 years in a classical supercomputer [1, 30].

### 3.2 Fundamentals of Quantum Computing.

#### 3.2.1 The qubit.

As we have said, a qubit can be in the state 0, 1 or in a state which is a superposition of them. Using Dirac’s notation, we describe a qubit as:

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle. \quad (3.1)$$

Where  $\alpha$  and  $\beta$  are complex numbers such that  $|\alpha|^2 + |\beta|^2 = 1$ . Thus, a qubit is a state  $|\psi\rangle \in H_2$ , being  $H_2$  a complex Hilbert space of two dimensions. The states  $|0\rangle$  and  $|1\rangle$  form an orthonormal basis of  $H_2$ . If we measure a qubit in the computational basis, we can obtain 0 or 1 with probabilities  $|\alpha|^2$  and  $|\beta|^2$  respectively. A common way to visualize a qubit is by the *Bloch Sphere*. If we write Equation (3.1) using Euler angles  $\theta, \delta$  and  $\varphi$ , it results:

$$|\psi\rangle = \left[ \cos(\theta/2) |0\rangle + e^{i\varphi} \sin(\theta/2) |1\rangle \right] e^{i\delta}. \quad (3.2)$$

Where  $\theta \in [0, \pi]$ ,  $\varphi \in [0, 2\pi]$ . The term  $e^{i\delta}$  can be ignored because it is a global phase and it has *no observable effects*. With that, the equation defines a unit 3-dimensional sphere. Figure 3.1 displays the Bloch sphere:

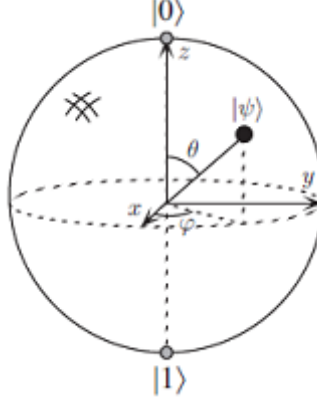


Figure 3.1: A qubit represented by the Bloch sphere. Picture from [31].

The figure shows that a qubit has infinite possible values which are superpositions of  $|1\rangle$  and  $|0\rangle$ . When we measure it in the computational basis, the state  $|\psi\rangle$  collapse into  $|1\rangle$  or  $|0\rangle$ , as the postulates of quantum mechanics establish. However, the power of quantum computing is in the information represented by a qubit if we do not collapse it. This *hidden information* increases exponentially with the number of qubits.

If we consider a system composed of 2 qubits, the state vector corresponding to the system is:

$$|\psi\rangle = \alpha_{00} |00\rangle + \alpha_{01} |01\rangle + \alpha_{10} |10\rangle + \alpha_{11} |11\rangle. \quad (3.3)$$

Now,  $|\psi\rangle \in H_4$ , the four kets form a orthonormal basis of the Hilbert space and the sum of the four probabilities has to be 1. In a system with  $N$  qubits, the Hilbert space has dimension  $2^N$ .

The behaviour of a  $H_2O$  molecule cannot be described by classical thermodynamics. Studying it as a quantum system requires such a quantity of information - we can say  $2^{500}$  single values - that it is impossible for any conceivable classical computer. However, manipulating that information would be possible with the Hilbert space corresponding to 500 qubits. Here is the computation power that we talked about [31].

### 3.2.2 Quantum logic gates.

A classical algorithm is carried out by a circuit composed of logic gates. Likewise, in quantum computing, algorithms go through the different quantum gates of a quantum circuit. We are able to manipulate the information via them. Figure 3.2 displays the main quantum gates. We can understand quantum gates as unitary operators which act on one or more quantum states (qubits).




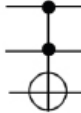

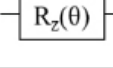

| Gate name       | # Qubits | Circuit Symbol  | Unitary Matrix   | Description   |
|-----------------|----------|---|--|---|
| Hadamard        | 1        |    | $\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$   | Transforms a basis state into an even superposition of the two basis states.  |
| T               | 1        |    | $\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$  | Adds a relative phase shift of $\pi/4$ between contributing basis states. Sometimes called a $\pi/8$ gate, because diagonal elements can be written as $e^{-i\pi/8}$ and $e^{i\pi/8}$ . |
| CNOT            | 2        |    | $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$   | Controlled-not; reversible analogue to classical XOR gate. The input connected to the solid dot is passed through to make the operation reversible.                                     |
| Toffoli (CCNOT) | 3        |    | $\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$ | Controlled-controlled-not; a three-qubit gate that switches the third bit for states where the first two bits are 1 (that is, switches $ 110\rangle$ to $ 111\rangle$ and vice versa).  |
| Pauli-Z         | 1        |  | $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$  | Adds a relative phase shift of $\pi$ between contributing basis states. Maps $ 0\rangle$ to itself and $ 1\rangle$ to $- 1\rangle$ . Sometimes called a "phase flip."                   |
| Z-Rotation      | 1        |  | $\begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}$  | Adds a relative phase shift of (or rotates state vector about z-axis by) $\theta$ .   |
| NOT             | 1        |  | $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$   | Analogous to classical NOT gate; switches $ 0\rangle$ to $ 1\rangle$ and vice versa.  |

Figure 3.2: Most common quantum gates used on 1, 2 and 3 qubits, along with their respective circuit symbols, matricial representations and descriptions. Picture from [4].

Like classical logic, gates with many inputs are hard to deal with. They are usually synthesized through a set of simpler gates. Typical quantum gates operate with 1, 2, or 3 qubits, as Figure 3.2 shows. All possible quantum functions can be built from a small set of quantum gates. A universal quantum gate set is any set of gates to which every possible task can be reduced on a quantum computer. That is to say, a quantum computer built only with this set gates could approximate any arbitrary function with the desired precision. The gates T, Hadamard, and CNOT, together with single qubits rotations, are an example of an universal quantum gate set [4, 31].

### 3.3 Quantum machine learning and quantum neural networks.

Quantum machine learning is the application of quantum algorithms in order to solve machine learning tasks. This includes hybrid methods with quantum and classical processes [32]. We have seen in Section 2 that machine learning techniques are a powerful tool in



many fields. However, there are domains where these techniques are not able to perform as we desire. The goal of quantum machine learning is to improve the applications in that domains via quantum algorithms and quantum computers [4].

Quantum Neural Networks (QNN) are the generalization of the classical neural networks through quantum computing. The term is stated by many different approaches, adding the implementation and extension of neural networks using layered variational circuits, photons, or quantum Ising-type models [32, 33]. Current studies, such as [2], show that QNN are able to reduce the computational cost and to obtain better results in several aspects in relation to classical NN. In the many attempts made over the years to implement QNN, two approaches can be clearly distinguished: discrete-variable QNN and continuous-variable QNN [34].

- A **discrete-variable quantum neural network** is a quantum circuit of perceptrons - we defined perceptrons in Section 2.4.1 - distributed in several hidden layers of unitary operations, performing on input quantum states, and producing the respective output quantum states. The network works with separable qubits in order to parameterize the inputs and outputs of quantum circuits, defining the quantum neuron as a positive function between both. It is a kind of quantum circuits composed of classical neurons. It allows the network to carry out the backpropagation in a very efficient way [34].
- A **continuous-variable (CV) quantum neural network** consists in the use of continuous-variable quantum systems - instead of qubits - in order to define a quantum perceptron. The harmonic oscillator is the most common CV quantum system. The CV models are based on the denominated *Gaussian states*, which has a certain probability distribution on the phase space picture. They can be uniquely determined by  $x$  and  $p$ , the position coordinate and its conjugate momentum, respectively [2, 34].

In this project, we have studied how to build a CV quantum neural network and we have done some simulations with it in order to compare it with the classical network studied in Section 2.5.

### 3.4 Continuous-Variable Quantum Neural Network (CV QNN).

This section explains the CV QNN developed in the novel paper [2]. The network is built and optimized by the quantum photonic company Xanadu, which has achieved to provide practical quantum results [35]. The network is designed in order to be executed on a photonic computer. However, for Gaussian states it can also be run on classical computers by the library Strawberry Fields [36]. This is a software platform for photonic quantum computing built in Python.

As we have mentioned in the previous section, in the CV models the quantum information is not processed by qubits. Instead of that, the information is transported in the quantum states of bosonic modes, denominated *qumodes*. They are the “wires” of the quantum circuit. We do not discuss why it is better to use this formalism rather than the discrete-variable (with qubits) one. That discussion can be found in several papers, such as [34, 37]. To continue, we define the elements of CV systems:

The bosonic harmonic oscillator is the basic CV system. It is defined through the

canonical mode operators  $\hat{a}$  and  $\hat{a}^\dagger$ . Quadrature operators can be expressed by them as:

$$\hat{x} = \sqrt{\frac{\hbar}{2}} (\hat{a} + \hat{a}^\dagger). \quad (3.4)$$

$$\hat{p} = -i\sqrt{\frac{\hbar}{2}} (\hat{a} - \hat{a}^\dagger). \quad (3.5)$$

They satisfy the commutation relation  $[\hat{a}, \hat{a}^\dagger] = \mathbb{I}$  and  $[\hat{x}, \hat{p}] = i\hbar\mathbb{I}$ . A qumode is defined as:

$$|\psi\rangle = \int \psi(x) |x\rangle dx. \quad (3.6)$$

Equation (3.1) describes a qubit using the discrete coefficients  $\alpha$  and  $\beta$  and the two states  $|0\rangle$  and  $|1\rangle$ . On the contrary, in CV systems we have a *continuum*, since the states  $|x\rangle$  are eigenstates of the position operator,  $\hat{x}|x\rangle = x|x\rangle$ ,  $x \in \mathbb{R}$ . The qumode state is described by the wavefunction  $\psi(x)$ . Alternately, it can also be described by a wavefunction based on a conjugate momentum variable  $\phi(p)$  and the states  $|p\rangle$ . The qumodes are initially in the fundamental state, denominated the *vacuum state*. The system evolves to other states by applying the *time evolution operator*:

$$|\psi\rangle = e^{-itH} |0\rangle, \quad (3.7)$$

where  $t$  is time and  $H$  is a bosonic Hamiltonian, which can be defined by a combination of the operators  $\hat{a}$  and  $\hat{a}^\dagger$ . *Gaussian states* are defined as those states where the Hamiltonian is at most quadratic in the operators  $\hat{a}$  and  $\hat{a}^\dagger$  (equally, in  $\hat{x}$  and  $\hat{p}$ ). They are so-named because every Gaussian state can be identified by a Gaussian distribution on the phase space picture, and it can be uniquely determined by the position and momentum. The qumode states are special samples of the Gaussian states. Therefore, the N-qumode Gaussian state is represented by  $2N$  variables,  $(x, p) \in \mathbb{R}^{2N}$  [2, 36].

In the same way that we have the quantum gates (Figure 3.2) in order to act on qubits, we can define the Gaussian gates to operate on the Gaussian states. The simplest single-mode Gaussian gates are the following:

$$\text{Displacement} \equiv D(\alpha) : \begin{pmatrix} x \\ p \end{pmatrix} \mapsto \begin{pmatrix} x + \text{Re}(\alpha) \\ p + \text{Im}(\alpha) \end{pmatrix}, \alpha \in \mathbb{C}. \quad (3.8)$$

$$\text{Rotation} \equiv R(\phi) : \begin{pmatrix} x \\ p \end{pmatrix} \mapsto \begin{pmatrix} \cos(\phi) & \sin(\phi) \\ -\sin(\phi) & \cos(\phi) \end{pmatrix} \begin{pmatrix} x \\ p \end{pmatrix}, \phi \in [0, 2\pi]. \quad (3.9)$$

$$\text{Squeezing} \equiv S(r) : \begin{pmatrix} x \\ p \end{pmatrix} \mapsto \begin{pmatrix} e^{-r} & 0 \\ 0 & e^r \end{pmatrix} \begin{pmatrix} x \\ p \end{pmatrix}, r \in \mathbb{R}. \quad (3.10)$$

In addition, the elementary two-mode Gaussian gate is a rotation between two qumodes. It is denominated *beamsplitter (BS)* and it produces the following transformation:

$$BS(\theta) : \begin{pmatrix} x_1 \\ x_2 \\ p_1 \\ p_2 \end{pmatrix} \mapsto \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & \cos(\theta) & -\sin(\theta) \\ 0 & 0 & \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ p_1 \\ p_2 \end{pmatrix}, \theta \in [0, 2\pi]. \quad (3.11)$$



In view of the above, the CV quantum neural network is created from several layers  $\mathcal{L}$ . Each  $\mathcal{L}$  – layer is formed by an arrangement of Gaussian gates:

$$\mathcal{L} = \Phi \circ D \circ U_1 \circ S \circ U_2, \quad (3.12)$$

where  $U_i = U_i(\theta, \phi)$  includes a single-mode rotation gate,  $R(\phi)$ , and a two-mode rotation gate,  $BS(\theta)$ .  $S = S(r)$  and  $D = D(\alpha)$  are respectively the squeezing and the displacement transformation gates. Finally,  $\Phi = \Phi(\lambda)$  represents a non-Gaussian gate, that corresponds to a non-linear function on the phase space  $(x, p)$ . It performs locally on every mode, like the non-linear activation function from the classical neural networks. In this model, the used non-Gaussian gate is the Kerr gate,  $K(\lambda) = e^{i\lambda\hat{n}^2}$ . Operator  $\hat{n}$  is defined on the *Fock States* basis and it is related to the number of qumodes [2].

It must be highlighted that in CV quantum network the layers are formed by the introduced CV-model gates instead of by the classical neurons. The sequence  $D \circ U_1 \circ S \circ U_2$  Gaussian gates corresponds to the weights,  $W$ , and biases,  $B$ , in classical models. They do an affine transformation, which is a lineal operation followed by a translation,  $|x\rangle \mapsto |Wx + B\rangle$ . Later, the application of  $\Phi$  leads to the desired transformation  $|x\rangle \mapsto |\Phi(Wx + B)\rangle$ . In addition, CV-model gates can act on fixed basis states  $|x\rangle$ , as well as on superpositions of them,  $|\psi\rangle = \int \psi(x) |x\rangle dx$ . Figure 3.3 displays the structure of a single layer in the CV quantum neural network, according to all mentioned elements:

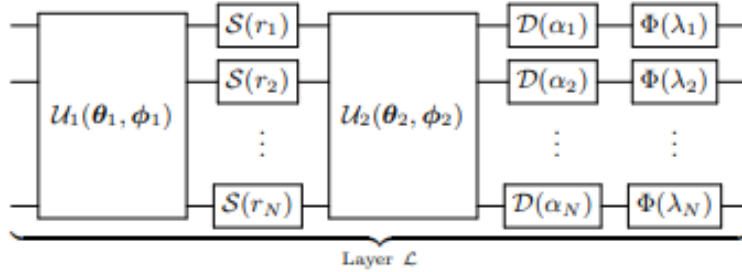


Figure 3.3: Single-layer structure in the CV quantum neural network. The first four operators effect an affine transformation, followed by a non-linear one, like an activation function in classical neural networks. Picture from [2].

The quantum state (formed by a set of qumodes) coming out from a layer is the input for the following one. The input of the first layer is created by applying the displacement operator  $D(x)$  to the state  $|0\rangle$ , according to the external information  $x$  that we want to introduce in the network. In the last layer, we read the output through an ideal homodyne detection in each qumode. Homodyne detection, in quantum tomography, is a way to determine the quantum state of a quantum system, or equally, to evaluate the expectation value of an operator on that system [38]. Figure 3.4 shows the structure of a six layers quantum neural network:

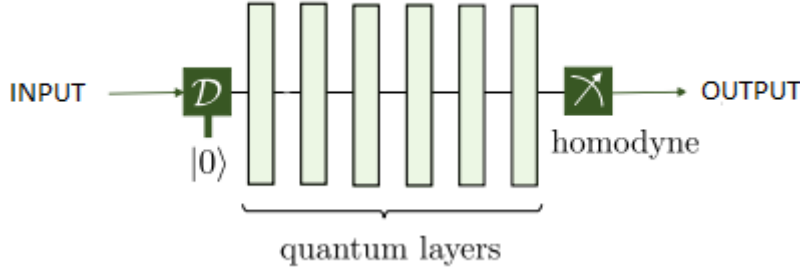


Figure 3.4: Structure of a multilayer continuous-variable quantum neural network composed of six layers. All elements have been described in the text. Picture from [2].

This model behaves equivalently to the classical neural network, where the qumodes propagate through the network and the Gaussian parameters  $\alpha, \theta, \phi$ , and  $r$  are updated in order to learn the relation between the inputs and outputs. The learning algorithm is similar to the studied one in Sections 2.4.1 and 2.4.3. The stochastic gradient descent is carried out by computing the loss function derivative with respect to the squeezing parameter ( $r$ ), beamsplitter parameter ( $\theta$ ), rotation parameter ( $\phi$ ), or displacement parameter ( $\alpha$ ). Taking  $\mu$  as any of them and using the Figure 3.3 notation, the updates in that layer are:

$$\mu_i^{\text{new}} = \mu_i^{\text{old}} + \Delta\mu_i. \quad \Delta\mu_i = -\eta \frac{\partial l^{(k)}}{\partial \mu_i}. \quad (3.13)$$

Where  $\eta$  is the learning rate. The updates are carried out from the output layer to the input one, according to the backpropagation algorithm (Section 2.4.3). Note that we do not have mentioned anything about the non-Gaussian gate parameter,  $\lambda_i$ . It can be understood simply like a constant of Kerr gate,  $K(\lambda_i) = e^{i\lambda_i \hat{n}^2}$ , which remains fixed over the entire algorithm. More details can be found in [2, 36].

### 3.5 Curve fitting with CV quantum neural network.

In this section, as we did in Section 2.5 with the classical network, we address the curve fitting task using the continuous-variable quantum neural network. For that, we make use of Strawberry Fields library. As it has been commented before, this is a Python open-source library for simulation, design, optimization, and quantum machine learning of continuous-variable circuits [36]. Strawberry Fields is equipped with TensorFlow, the Google's platform for machine learning, and Numpy, the Python library to work with multidimensional arrays. We also use the Python library Matplotlib in order to build the graphics. The implemented source code is in [5]. Specifically, the used support has been Python 3.6.10, TensorFlow 1.3.0, StrawberryFields 0.10.0, Numpy 1.18.1, and Matplotlib 3.1.3.

The test function is again  $\sin(\pi x)$  in a cycle,  $(-1, 1)$ . The training data is a set of  $m$  inputs  $x^{(k)} \in (-1, 1)$  and their respective outputs modified by noise  $y^{(k)} \equiv y(x^{(k)}) = \sin(\pi x^{(k)}) + \varepsilon(x^{(k)})$ , being  $\varepsilon(x^{(k)})$  random uniform numbers between -0.1 and 0.1. The goal of the network is to learn the function  $\sin(\pi x) \forall x \in (-1, 1)$ .

The input  $x^{(k)} \equiv x$  is encoded as  $D(x)|0\rangle$ , according to the Figure 3.4. If  $|\psi_x\rangle$  is the output of that state and  $f(x)$  is the function to fit, the objective is to achieve that the

expectation value of the quadrature operator,  $\hat{x}$ , coincides with  $f(x)$ . That is  $\langle \psi_x | \hat{x} | \psi_x \rangle = f(x)$ ,  $\forall x$ . Thus, in the same way that the classical network, we define the loss function as:

$$L = \frac{1}{m} \sum_{k=1}^m \left( y^{(k)} - \langle \psi_{x^{(k)}} | \hat{x} | \psi_{x^{(k)}} \rangle \right)^2. \quad (3.14)$$

We have implemented the *Adam Optimizer* [19] by the use of TensorFlow, which implies that we do not have a fixed learning rate as in the classical network.

The following simulations present some interesting results. They allow to compare the classical and the quantum neural network. Comparisons are based in the number of layers and the overfitting issue, since the rest of the parameters (such as the learning rate or the initializations) are not comparable between classical and quantum models.

### Enhancement with depth.

We analyse the behaviour of the network according to the number of layers. By means of Equation (3.12), a quantum layer is defined as a sequence of Gaussian operations followed by a non-Gaussian operation.

Like in the classical network problem, I have prepared 50 points equally separated in order to study how the function is fitted by neural networks with different number layers. Figure 3.5 displays the different approximations with each architecture. The four models have been training for 1000 epochs.

Curve fitting for different number of layers.

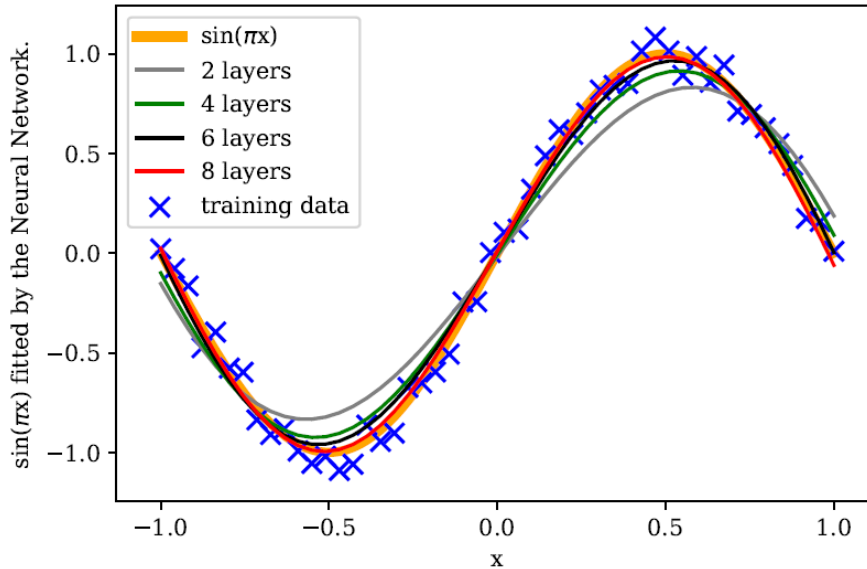


Figure 3.5: Continuous-variable quantum neural networks with different number of layers have been trained with 50 points belonging to the function  $y(x) = \sin(\pi x) + \varepsilon(x)$  within the range  $(-1, 1)$ , being  $\varepsilon(x)$  a random number between -0.1 and 0.1 for every  $x$ . This Figure corresponds to Figure 2.3 from the classical network.

We can observe that the eight layers model shows very good results. On the other

hand, the learned function by the 2 layers model does not have enough parameters to adjust the curve path. This behaviour is similar to the learned one in Figure 2.3 by models with one layer. Figure 3.5 also manifests no signal of overfitting.

It has been tried to improve the eight layers result by adding more layers, but results are similar. So we can affirm that for curve fitting tasks, the network enhancements saturates at eight layers.

Figure 3.6 presents the evolution of each model according to the epochs. It should be mentioned that, although the epochs have a similar meaning here and in the classical networks, they are not comparable because the updates in the CV quantum neural network are different from the classical one.

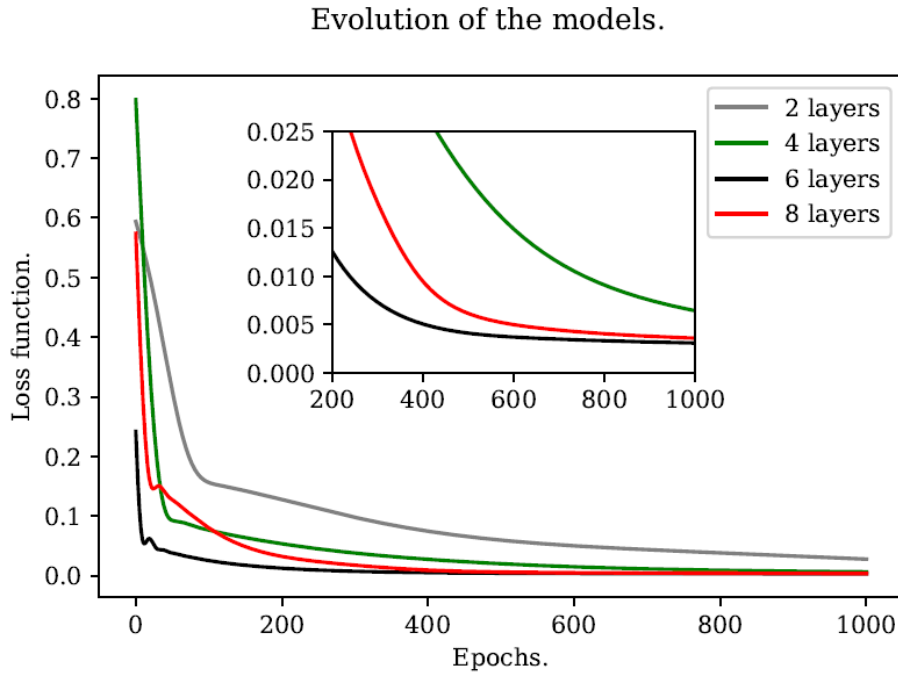


Figure 3.6: Evolution of each model from Figure 3.5.

The most noticeable difference in relation to the evolution in the classical network displayed in Figure 2.4 is in the first epochs: While in the classical case (Figure 2.4) some models need more epochs than others in order to get the parameter values that begin to minimize the loss function, here all the models start to minimize the function quickly. Although Figure 3.6 shows that the loss function of 6 layers model is lower than the 8 layers one, we have seen in Figure 3.5 that 8 layers model provides a better approximation. This happens due to the fact that the 6 layers model has an error referred to noisy points that is smaller than the 8 layers one. However, as Figure 3.5 displays, the 8 layers model is closer to  $\sin(\pi x)$  than the 6 layers one.

Figure 3.6 inset also displays that the loss functions do not reach the zero value, which suggests that the models do not undergo overfitting. The next point is focused on this.

### Overfitting.

Theoretically, continuous-variable models produce smooth outputs, i.e., they are not affected by noise. This happens because the quantum states are so close to each other that their expectation values are similar. Quantitatively, this smoothness property of quantum neural networks can be understood through the Hölder's inequality, which affirms that for any two states  $\xi$  and  $\xi'$ ,

$$|Tr((\xi - \xi') A)| \leq \|\xi - \xi'\|_1 \|A\|_\infty. \quad (3.15)$$

for any operator  $A$  [2]. Figure 3.7 shows that, effectively, the CV quantum neural network is not subject to overfitting despite a long training. It presents a model constituted by 10 layers and trained with 50 points.

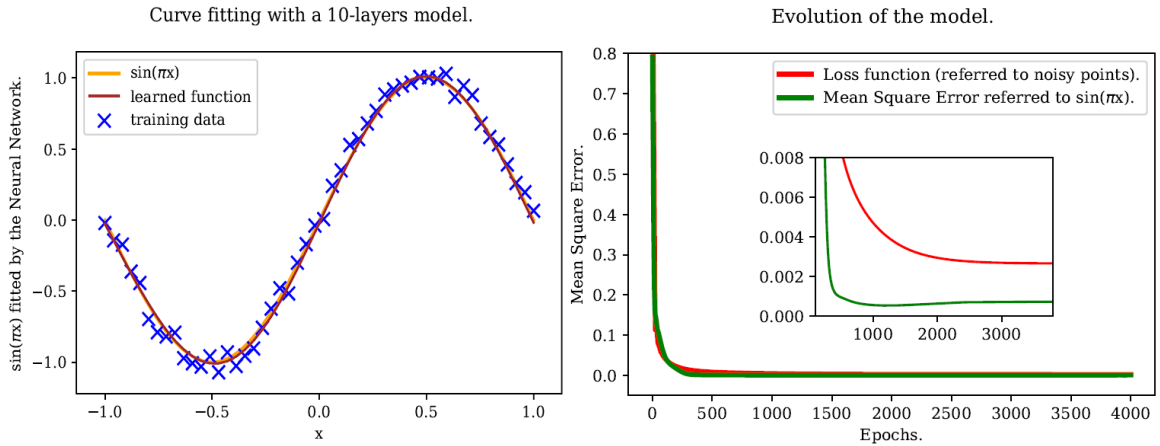


Figure 3.7: Left: an optimal fitted curve after 4000 epochs. Right: the model remains in the optimal configuration and it does not get to be overfitted by increasing the epochs. Although red and green curves look similar, the zoom exhibits that the fitted function has a lower error referred to  $\sin(\pi x)$  than the noisy points.

As Figure 2.6, Figure 3.7 presents a model composed of more parameters than can be justified by the data - we have seen in Figure 3.5 that 8 layers network can perfectly adapt to the data -, and it has been trained so long. However, opposed to Figure 2.6, where results started to become worse due to overfitting after many epochs, the quantum model does not undergo overfitting despite the long training.

Now, we analyse how the network works when there is not a big number of data points to learn. Figure 2.5 shows that the classical network results are exposed to overfitting in the mentioned situation. Figure 3.8 displays the same previous (10 layers) quantum network training with 25 and 15 data points. Training time is 4000 epochs again.

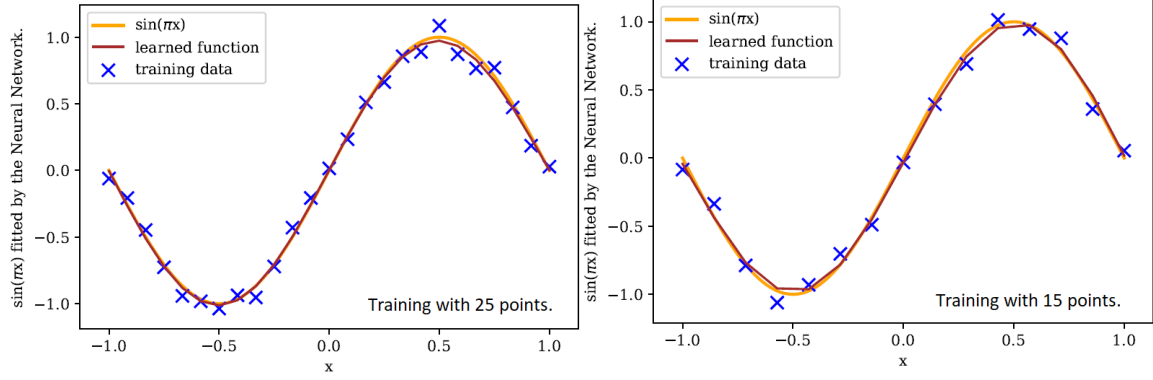


Figure 3.8: 10 layers quantum network after 4000 epochs. Different sizes of data set.

Figure 3.8 verifies that the continuous-variable quantum neural network avoids the overfitting issue much better than the classical model. In spite of a low number of data points and a long training, the learned relation by the network does not turn out focused in too specific features, such as passing through all training points. Note that results with 15 points within the CV quantum network are better than results with 42 points in the classical case displayed in Figure 2.5.

### Conclusions.

We have seen that building a continuous-variable quantum neural network is a wide task and it requires a much more complex formalism than classical networks. However, they undoubtedly lead to better results in several aspects, such as overfitting.

This landmark suggests a machine learning development based on hybrid models, where the quantum part can be specialized in those aspects which present worse results through classical algorithms. To carry it out in practical terms, a hardware capable of getting quantum algorithms to run in an efficient way would be necessary.

## 4 Detecting quantum entanglement via neural networks.

As we have mentioned in Section 3, the quantum computing power is based on the superposition and entanglement. An unsolved problem in the quantum information theory is the detection of entanglement. To determine if a given quantum state is separable or entangled and to define a level of entanglement (i.e., to quantify the entanglement) is a difficult task which has been dealt with by different approaches, without achieving a close-form solution.

In this section, we introduce the detecting quantum entanglement task and we show how classical neural networks are able to determine if the 2-qubits system density matrix corresponds to an entangled or separable state.

### 4.1 Quantum entanglement.

Quantum entanglement is one of the most striking quantum phenomenon, which does not have classical analogous. It was introduced in 1935 by Einstein, Podolsky and Rosen

in their famous “EPR” paper [39]. They suggested that quantum mechanics does not provide a complete description of reality. The term “entanglement” was coined by Schrödinger that same year, who classified the entanglement as the most important quantum property, since it separates the classical world from the quantum one.

The mathematical property which quantum entanglement is based on is inseparability. Using the structure of Hilbert spaces in quantum mechanics, a composite systems is described by the tensor products of Hilbert spaces corresponding to the subsystems. In this context, inseparability allows the existence of pure states of composite systems that cannot be factorized (separated) as the tensor products of states associated with each subsystem [3]. Two qubits can form the simplest entangled quantum system. Every qubit is described by a two-dimensional Hilbert space (Eq. (3.1)), so the composite system is based in a four-dimensional Hilbert space (Eq. (3.3)). If we denote by  $\{|0\rangle_1, |1\rangle_1\}$  and  $\{|0\rangle_2, |1\rangle_2\}$  the orthonormal basis for the two qubits, two examples of separable states in that system are:

$$|\psi\rangle = |00\rangle = |0\rangle_1 \otimes |0\rangle_2. \quad (4.1)$$

$$|\phi\rangle = \frac{1}{2} (|00\rangle + |01\rangle + |10\rangle + |11\rangle) = \frac{1}{\sqrt{2}} (|0\rangle_1 + |1\rangle_1) \otimes \frac{1}{\sqrt{2}} (|0\rangle_2 + |1\rangle_2). \quad (4.2)$$

Conversely, examples of 2-qubits entangled states are the Bell states:

$$|\psi^\pm\rangle = \frac{1}{\sqrt{2}} (|01\rangle \pm |10\rangle), \quad (4.3)$$

$$|\phi^\pm\rangle = \frac{1}{\sqrt{2}} (|00\rangle \pm |11\rangle), \quad (4.4)$$

where it is not possible to express any of them as a sequence of tensor products. The name *Bell states* is due to the Irish physicist John S. Bell, who pushes the development of this field along the 60s.

Let us consider a pure n-partite state  $|\psi\rangle_{A_1 \dots A_n} \in H_{A_1 \dots A_n} = H_{A_1} \otimes \dots \otimes H_{A_n}$ , where the Hilbert space of the composed system  $H_{A_1 \dots A_n}$  is the tensor product of the Hilbert spaces  $H_{A_i}$  of all subsystems, being  $A_i$  the set of operators which act on each  $H_{A_i}$ . It is a fully separable state iff it can be expressed in a factorized way:

$$|\psi\rangle_{A_1 \dots A_n} = |\psi\rangle_{A_1} \otimes \dots \otimes |\psi\rangle_{A_n}. \quad (4.5)$$

Thus, a pure state which cannot be factorized is an entangled state. Equally, a general mixed state is fully separable iff it can be decomposed as:

$$\rho^{A_1 \dots A_n} = \sum_{i=1}^k w_i \rho_i^{(A_1)} \otimes \dots \otimes \rho_i^{(A_n)}, \quad (4.6)$$

where  $\rho$  is the density matrix [40]. The  $w_i$  terms form a convex set, i.e.,  $\sum_{i=1}^k w_i = 1$ . The density matrices that do not fulfill Equation (4.6) correspond to entangled states [3]. On the other hand, it should be recalled that a density matrix is a complex matrix such that:

$$\text{Tr}(\rho) = 1. \quad \rho \geq 0. \quad \rho = \rho^\dagger. \quad (4.7)$$

In addition, if the density matrix corresponds to pure states, it is also idempotent,  $\rho^2 = \rho$ .



## 4.2 Peres-Horodecki criterion.

The fact that quantum systems can be in entangled states leads naturally to the next problem: given a composed quantum system in a mixed state described by density matrix  $\rho$ , determine if  $\rho$  represents a separable state or not. The Peres–Horodecki criterion is a necessary condition for the joint density matrix  $\rho$  of two quantum systems to be factorized as Equation (4.6) points, i.e., it provides a necessary condition for the separability of bipartite quantum systems [41]. The condition is also sufficient for the  $2 \times 2$  and  $2 \times 3$  dimensional cases [42]. The criterion is obtained by doing a partial transpose on  $\rho$ , as it is shown below.

Consider a bipartite quantum system composed of two subsystems A and B in a separable state:

$$\rho = \sum_{i=1}^k w_i \rho_i^{(A)} \otimes \rho_i^{(B)}, \quad (4.8)$$

where  $\rho_i^{(A)}$  and  $\rho_i^{(B)}$  are density matrices for the subsystems A and B, and  $\sum_{i=1}^k w_i = 1$ . In order to obtain the separability condition, the first is to write the Equation (4.8) explicitly:

$$\rho_{m\mu, nv} = \sum_{i=1}^k w_i \left( \rho_i^{(A)} \right)_{mn} \left( \rho_i^{(B)} \right)_{\mu\nu}. \quad (4.9)$$

Now, we define the following matrix:

$$\sigma_{m\mu, nv} \equiv \rho_{n\mu, mv}. \quad (4.10)$$

where the indices  $m$  and  $n$  from Equation (4.9) have been transposed. The density matrix  $\sigma$  is Hermitian. Thus, if  $\rho$  is separable,  $\sigma$  can be expressed as Equation (4.8) transposing the first subsystem:

$$\sigma = \sum_{i=1}^k w_i \left( \rho_i^{(A)} \right)^T \otimes \rho_i^{(B)}. \quad (4.11)$$

Note that the partial transpose (regarding the A party) of  $\rho$  can be expressed by  $(T \otimes \mathbb{I}) \rho$ , where  $T$  is the transposition map and  $\mathbb{I}$  is the identity map.

Since the transposed matrices fulfill  $\text{Tr} \left[ \left( \rho_i^{(A)} \right)^T \right] = 1$  and their eigenvalues are non-negative,  $\left( \rho_i^{(A)} \right)^T$  are also density matrices. Hence, none of the eigenvalues of  $\sigma$  is negative. This allows us to affirm that if we transpose partially a separable density matrix, their eigenvalues remain non-negatives. This is the necessary condition of separability [41]. In addition, it is proved that the condition is also sufficient for composed systems with dimensions  $2 \times 2$  and  $2 \times 3$  [42]. In this way, given the density matrix of a system composed of two qubits ( $2 \times 2$ ), the Peres-Horodecki criterion determines unambiguously if the system is in an entangled state or in a separable state.

A common example of the criterion is to apply it to the 2-qubits family of Werner states [43]:

$$\rho = x |\psi\rangle \langle \psi| + \frac{1-x}{4} \mathbb{I}, \quad (4.12)$$

where  $x \in [0, 1]$  and  $|\psi\rangle = \frac{1}{\sqrt{2}} [|01\rangle - |10\rangle]$  is a Bell state. The density matrix results:

$$\rho = \frac{1}{4} \begin{pmatrix} 1-x & 0 & 0 & 0 \\ 0 & x+1 & -2x & 0 \\ 0 & -2x & x+1 & 0 \\ 0 & 0 & 0 & 1-x \end{pmatrix}, \quad (4.13)$$



and its partial transpose:

$$\rho^{T_B} = (\mathbb{I} \otimes T) \rho = \frac{1}{4} \begin{pmatrix} 1-x & 0 & 0 & -2x \\ 0 & x+1 & 0 & 0 \\ 0 & 0 & x+1 & 0 \\ -2x & 0 & 0 & 1-x \end{pmatrix}. \quad (4.14)$$

The lowest eingevalue of this matrix is  $(1 - 3x)/4$ . Thus, by virtue of Peres-Horodecki criterion, the state is entangled for  $\frac{1}{3} < x \leq 1$  and the state is separable for  $0 \leq x \leq \frac{1}{3}$ .

### 4.3 Two-qubits entanglement via neural networks.

In this section, we study the possibility of detecting entanglement in systems composed of two qubits using neural networks. Specifically, four sets of 10.000 density matrices have been prepared. The first set corresponds to separable pure states, the second is for entangled pure states, the third is for separable mixed states, and the fourth is for entangled mixed states. Note that they are  $4 \times 4$ -dimensional complex matrices and they have the properties mentioned in Equation (4.7). The hermiticity allows us to encode every matrix by 16 real numbers<sup>1</sup>. In this way, we build a neural network in which an input of 16 real numbers is introduced and the network output is 0 or 1. The goal is to achieve the output network to be 0 (1) when the input corresponds to a separable (entangled) density matrix.

This study is not focused on the network analysis, i.e., we do not study the network behaviour according to the parameters (layers, neurons, learning rate, etc). Here, we are testing if the network is able to learn the Peres-Horodecki criterion. Therefore, we have built a model that carries out the task and we have analysed the results by introducing in it the different data sets. The network is implemented in Python by means of TensorFlow and Keras. Specifically, the used support has been Python 3.7.6, Keras 2.3.1, TensorFlow 2.1.0, Numpy 1.18.1, Pandas 1.0.1 and Matplotlib 3.1.3. The code-source can be found in [5].

The network is composed of an input layer of 16 neurons (inputs are 16-dimensional vectors), two hidden layers with 32 and 8 neurons respectively, and an output layer with one neuron (outputs are a single number). The activation function in the hidden layers is the *Rectified linear unit* (Eq. (2.5)). The one in the output layer is a sigmoid. The gradient descent steps are regulated by the *Adam Optimizer* [19]. Finally, unlike curve fitting, now we have a classification task, so the loss function to minimize is the binary cross-entropy (it had already been mentioned in Eq. (2.7)):

$$L(a, y) = -\frac{1}{m} \sum_{k=1}^m \left( y^{(k)} \log(a^{(k)}) + (1 - y^{(k)}) \log(1 - a^{(k)}) \right), \quad (4.15)$$

where  $a^{(k)}$  is the network output for the input  $\vec{X}^{(k)} \in \mathbb{R}^{16}$  and  $y^{(k)}$  is the correct output. First, we have tested the network with pure states. To do so, we have randomly mixed the 2 sets composed of pure states and we have used the 80% of data in order to train the network. Thus, in Equation (4.15)  $m = 16.000$  input vectors. The remaining 20% of data (4.000 matrices) is used to test that the network is able to classify correctly the density matrices which have not trained with, i.e., we check that there is not overfitting. Secondly, we have repeated the process using the 2 sets composed of mixed states. The data has

<sup>1</sup>Four real numbers from the diagonal and six complex numbers from one side.

been equally distributed. Figure 4.1 displays the learning process with each couple of data sets.

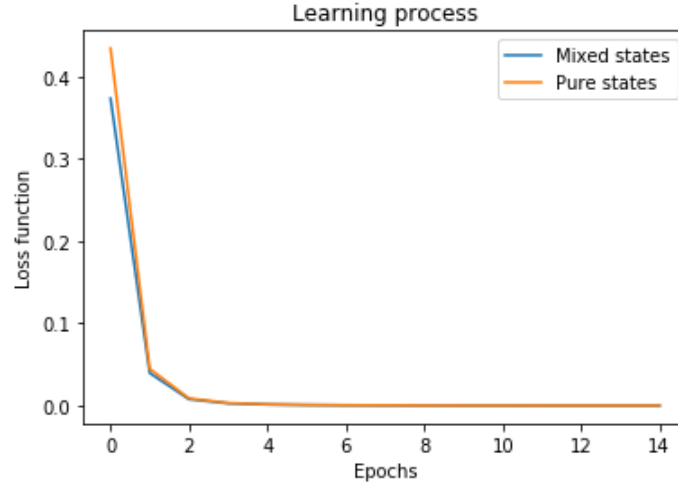


Figure 4.1: Change of the loss functions (Eq. (4.15)) with the number of epochs. Training data of blue curve: 16.000 density matrices correspond to mixed-entangled or mixed-separable states of 2 qubits. Training data of orange curve: the same with pure states.

Figure 4.1 shows that the network learns the difference between entangled and separable states, for both mixed and pure states. The network evolution is similar in the two cases. Figure 4.1 also shows a very low number of epochs in comparison with the other similar graphics from this project. This is because the network is training with a lot of data. Every epoch implies 16.000 weights and biases updates. In the curve fitting task (Section 2.5), we have generally worked with 50 data points, so one epoch is 50 updates.

Once the training process is completed, we have introduced to the network the remaining density matrices and the network has been able to classify them correctly. Note that these matrices are “unknown” for the network, since it has not trained with them. Specifically, the following has been tested:

- We train the network with pure states (orange curve) and we test it with pure and mixed states.
- We train the network with mixed states (blue curve) and we test it with pure and mixed states.

The obtained result is that the 100% of test data has been correctly classified by the network. This allows us to conclude that neural networks can learn the Peres-Horodecki criterion and they are effective in order to detect entanglement.

## 5 Conclusions.

In the introduction, we established complementing artificial intelligence and quantum computing as the main goal of this project.

In Section 2, we have developed the fundamental neural network model in deep learning: the multilayer perceptron. We have studied the algorithms that allow it to learn the right relation between inputs and outputs, analysing the limitations and the issues which can appear. Once its operations were understood, in Section 3, we have presented a quantum neural network based on continuous-variable formalism of quantum computing and based on the perceptron algorithms. Both (classical and quantum networks) have been explored in detail by applying them to a curve fitting task. This led us to conclude that quantum algorithms are effective in machine learning and able to obtain better results in some aspects. In particular, we have shown that quantum neural networks avoid the overfitting issue better than the classical ones. Finally, in Section 4, we have made use of the knowledge about neural networks obtained in the previous sections to tackle an unsolved problem related to quantum information and computation theory: the detection of quantum entanglement. We have found that neural networks can be helpful in this kind of tasks. Therefore, we can affirm that we have successfully complemented quantum computing and artificial intelligence, so the main goal of this project has been achieved.

In addition, I have acquired important knowledge about classical and quantum machine learning. Specifically referred to the first field, I have learned how to program a neural network from scratch and how to find the parameters to optimize it. I have also learned the Python programming language and the usage of classical and quantum machine learning libraries. Referred to quantum computing, this project provided me with the key concepts about continuous-variable systems, as well as about quantum entanglement. Besides, this last one allowed me to notice that there are more machine learning applications in Physics than I could imagine. Last but not least, I would like to highlight that carrying out this thesis in English has improved significantly my skills in the use of this language.

We have also seen through a lot of papers that, indeed, the combination of artificial intelligence and quantum computing is a new research branch which is noticeably growing. Quantum machine learning is how this field is widely referred to. Nowadays, its research is headed by the private sector, where companies such as Google, IBM, Intel and D-Wave Systems are the most prominent. Some companies from the automotive sector are also investing in quantum technologies, such as Volkswagen, which is developing a route optimizer for its buses by means of quantum computers [44].

However, in spite of the pointed advantages of quantum computing, it also presents drawbacks. Hardware is the biggest issue. Determining what is the best collection of physical parts for a quantum computer is not clear yet. The most popular candidates are based on superconducting electronic circuits, trapped ions, and quantum dots. The first one implements qubits as the state of a small superconducting circuit, the second one does it through the internal state of trapped ions, and the third one is based on the spin states of trapped electrons. Nevertheless, there are many other approaches that have shown their efficacy, as nuclear magnetic resonance quantum computers or diamond-based quantum computers. Specifically, the famous Google quantum computer, Sycamore, consists of an aluminium and silicon chip connected to a superconducting circuit plate. Liquid helium

holds the system temperature lower than 1 Kelvin [1, 30]. On the other hand, the company Xanadu, which provides the Strawberry Fields framework, designs quantum silicon photonic chips, since photons are more stable to heat than electrons [35].

The main problem that the different hardware try to remove is quantum decoherence. A definite phase relationship is necessary to perform quantum computing on the information encoded in quantum states. A perfectly isolated quantum system maintains the coherence indefinitely. However, when it is manipulated - when we measure - coherence is shared with the environment and quantum decoherence increases with time. Thus, quantum computers require quantum circuits to act faster than decoherence.

Taking into account all of this, although quantum machine learning is a very promising branch that shows considerable results in some applications, we will not see its full potential until quantum computers hardware is more developed. Thereafter, a new world of possibilities will open up. The society we live in is strongly influenced by the computer sciences, which began to take its first great steps in the 50s. Quantum computing is now in a similar situation to the classical one then, and, in the same way, the applications could be endless, as it is the work to be done ahead.

## References

- [1] F. Arute, K. Arya, R. Babbush *et al.*, *Quantum supremacy using a programmable superconducting processor*. Nature 574, 505–510 (2019). [Link](#).
- [2] N. Killoran *et al.*, *Continuous-variable quantum neural networks*. Physical Review Research 1, 033063 (2019). [Link](#).
- [3] Daniel Manzano, *Information and Entanglement Measures in Quantum Systems with Applications to Atomic Physics*. PhD Thesis, Universidad de Granada (2010). [Link](#).
- [4] National Academies of Sciences, Engineering, and Medicine, *Quantum Computing: Progress and Prospects*. Washington, DC: The National Academies Press (2019). [Link](#).
- [5] Eduardo Pérez, GitHub: Source codes.  
<https://github.com/edu-perez/Trabajo-de-Fin-de-Grado>
- [6] J. Franknfield, Investopedia: Artificial Intelligence,  
<https://www.investopedia.com/terms/a/artificial-intelligence-ai.asp> . Accessed 5 Apr. 2020.
- [7] M. Rose, Tech Target: Artificial Intelligence,  
<https://searchenterpriseai.techtarget.com/definition/AI-Artificial-Intelligence>. Accessed 5 Apr. 2020.
- [8] A. M. Turing, *Computing Machinery and Intelligence*. Mind 49: 433-460 (1950). [Link](#)
- [9] GeeksforGeeks: Turing Test, <https://www.geeksforgeeks.org/turing-test-artificial-intelligence/>. Accessed 5 Apr. 2020.
- [10] History of Computers: LISP Programming,  
<https://history-computer.com/ModernComputer/Software/LISP.html>. Accessed 6 Apr. 2020.
- [11] Expertsystem: Machine Learning, <https://expertsystem.com/machine-learning-definition/>. Accessed 6 Apr. 2020.
- [12] Ethem Alpaydin, *Introduction to Machine Learning*. 2nd Edition, The MIT Press (2010).
- [13] James Le: *A Gentle Introduction to Neural Networks*,  
[https://www.codementor.io/@james\\_aka\\_yale/a-gentle-introduction-to-neural-networks-for-machine-learning-hkijvz7lp](https://www.codementor.io/@james_aka_yale/a-gentle-introduction-to-neural-networks-for-machine-learning-hkijvz7lp). Accessed 9 Apr. 2020.
- [14] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*. The MIT Press (2016).
- [15] Andrew Ng, K. Katanforoosh and Y. B. Mourri, *Deep Learning Specialization*. deeplearning.ai Coursera, <https://www.deeplearning.ai/deep-learning-specialization/>. Accessed March-Apr. 2020.
- [16] Micha Wetzel, *Audio Segmentation for Robust Real-Time Speech Recognition Based on Neural Networks*. Bachelor's thesis, Institute for Anthropomatics and Robotics, Interactive Systems Labs (2016). [Link](#).

- [17] J. Pérez V.: Perceptrón Multicapa, <http://bibing.us.es/proyectos/abreproy/12166/fichero>. Accessed 15 Apr. 2020.
- [18] H. Bouzgou. *Advanced Methods for the Processing and Analysis of Multidimensional Signals: Application to Wind Speed*. PhD Thesis, University of Hadj Lakhdar, Batna (2012). [Link](#).
- [19] D. P. Kingma and J. L. Ba, *Adam: A Method for Stochastic Optimization*. arXiv:1412.6980 (2014). [Link](#).
- [20] N. Srivastava *et al.*, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. Journal of Machine Learning Research. 15. 1929-1958 (2014). [Link](#).
- [21] Xavier Glorot and Yoshua Bengio, *Understanding the difficulty of training deep feedforward neural networks*. In "Proceedings of the thirteenth international conference on artificial intelligence and statistics" p. 249-256 (2010).
- [22] R. Hecht-Nielsen, *Kolmogorov's Mapping Neural Network Existence Theorem*. Conference proceedings (1987).
- [23] Everitt B.S. and Skrondal A., *Cambridge Dictionary of Statistics*, Cambridge University Press (2010).
- [24] Jeremy Jordan's Blog, *Setting the learning rate of your neural network*. <https://www.jeremyjordan.me/nn-learning-rate/>. Accessed 15 Apr. 2020.
- [25] Marcus Perry, *The Exponentially Weighted Moving Average*. 10.1002/9780470400531.eorms0314, (2010). [Link](#).
- [26] Alex Graves, *Generating Sequences With Recurrent Neural Networks*. arXiv:1308.0850v5 [cs.NE] (2014). [Link](#).
- [27] Institute for Quantum Computing, University of Waterloo. <https://uwaterloo.ca/institute-for-quantum-computing/quantum-computing-101#What-is-quantum-computing>. Accessed 3 May 2020.
- [28] R. P. Feynman, *Simulating physics with computers*. Int. J. Theor. Phys. 21, 467–488 (1982). [Link](#).
- [29] Jack Hidary, *A Brief History of Quantum Computing*. doi: 10.1007/978-3-030-23922-0\_2 (2019).
- [30] B. Varona, *Techedge: Supremacía cuántica: Google vs IBM*, <https://www.techedgegroup.com/es/blog/supremacia-cuantica-google-vs-ibm>. Accessed 1 June 2020.
- [31] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. 10th Anniversary Edition, Cambridge University Press (2009).
- [32] IEEE: Quantum Machine Learning, <https://edu.ieee.org/in-dscepes/2019/12/11/quantum-machine-learning/>. Accessed 25 May 2020.

- [33] T. J. Ackermann, *D-Wave Annealing Quantum Computers Tackle Big Data With A ML Quantum Boltzmann Machine (Artificial Neural Network)*. <https://www.bgp4.com/2018/10/16/>. Accessed 25 May 2020.
- [34] P. Qiu, X. Chen and Y. Shi, *Detecting Entanglement With Deep Quantum Neural Networks*. IEEE Access, vol. 7, pp. 94310-94320, doi: 10.1109/ACCESS.2019.2929084 (2019).
- [35] Xanadu: a quantum photonic company, <https://www.xanadu.ai/>. Accessed 1 June 2020.
- [36] N. Killoran *et al.*, *Strawberry fields: A software platform for photonic quantum computing*. arXiv:1804.03159 (2018). [Link](#).
- [37] U. Andersen *et al.*, *Hybrid discrete- and continuous-variable quantum information*. Nature Phys 11, 713–719 (2015). [Link](#).
- [38] G. M. D’Ariano, L. Maccone and M. F. Sacchi, *Homodyne tomography and the reconstruction of quantum states of light*. arXiv:quant-ph/0507078 (2005). [Link](#).
- [39] A. Einstein, B. Podolsky and N. Rosen, *Can quantum-mechanical description of physical reality be considered complete?*. Phys. Rev., 47(10) 777 (1935). [Link](#).
- [40] A. Galindo, P. Pascual, *Mecánica Cuántica*. Ed. Eudema Universidad (1989).
- [41] A. Peres. *Separability criterion for density matrices*. Phys. Rev. Lett., 77(8) 1413 (1996). [Link](#).
- [42] ] M. Horodecki, P. Horodecki, and R. Horodecki, *Separability of mixed states: Necessary and sufficient conditions*. Phys. Lett. A, vol. 223, nos. 1–2, pp. 1–8, (1996). [Link](#).
- [43] R. F. Werner. *Quantum states with Einstein-Podolsky-Rosen correlations admitting a hidden-variable model*. Phys. Rev. A, 40(8) 4277 (1989). [Link](#).
- [44] Proyecto Volkswagen computación cuántica, <https://noticias-renting.aldautomotive.es/volkswagen-trafico-lisboa/>. Accessed 1 June 2020.