Technical University of Crete
Department of Electrical and Computer Engineering

Hybrid Quantum - Classical Machine Learning and Applications

Diploma Thesis for School of Electrical and Computer Engineering

Author
Christos Michalopoulos

Supervisor
Associate Professor Dimitris Angelakis

Committee Members
Professor Michail Zervakis
Professor Dionissios T. Hristopulos

August 2022

Abstract

In this thesis, we study the interface between quantum computing and machine learning, and more specifically quantum machine learning (QML) algorithms and certain applications in financial problems. We start by defining the building blocks of quantum computers, such as quantum states and quantum gates, along with the analytic presentation of three key quantum algorithms for our work: the quantum Fourier transform, the quantum phase estimation and the quantum amplitude estimation algorithm. We continue with summarizing the basics of classical machine learning and analyze in detail the inner workings of neural networks and specifically, of the generative adversarial networks (GANs). Next, we discuss how quantum algorithms can be incorporated in classical machine learning approaches. We analyze the two leading areas of QML, the fault-tolerant QML and the Noisy-Intermediate Scale Quantum (NISQ) friendly QML algorithms; in the former case, QML algorithms are shown to have proven quantum speedups against its classical counterparts but require fault-tolerant quantum hardware that are yet-to-be setup, while the latter can be realized on the current available devices but quantum speedups, or provable quantum advantages is yet-to-be demonstrated. In the next and main part of the work, we analyze and build on some of the NISQ-friendly QML algorithms, i.e: hybrid classical-quantum variational models that consist of quantum and classical processing. Within this approach, we study in details the quantum version of GANs, QGANs, and show how they can be trained to produce quantum states that efficiently encode and learn target probability distributions. We compare the training performance of the QGANs using different initial input probability distributions in various settings, e.g: 3 and 4 qubits systems, and models with different numbers of quantum circuit repetitions. These trained quantum states are then used along with quantum amplitude estimation algorithms, to compute important quantities in the financial world, such as the European call option problem found in real world markets. We implement our quantum algorithms in classical simulators and benchmark the performance for different numbers of qubits and configurations and discuss possible follow up works and applications.

Περίληψη

Σε αυτή τη διατριβή, μελετάμε τη διεπαφή μεταξύ κβαντικού υπολογισμού και μηχανικής μάθησης, και πιο συγκεκριμένα αλγορίθμων κβαντικής μηχανικής μάθησης (QML) και ορισμένων εφαρμογών τους σε προβλήματα στην Οικονομία. Ξεκινάμε ορίζοντας τα δομικά στοιχεία των κβαντικών υπολογιστών, όπως οι κβαντικές καταστάσεις και κβαντικές πύλες, μαζί με την αναλυτική παρουσίαση τριών βασικών κβαντικών αλγορίθμων για την εργασία μας: τον κβαντικό μετασχηματισμό Fourier, την εκτίμηση κβαντικής φάσης και τον αλγόριθμο εκτίμησης κβαντικού πλάτους. Συνεχίζουμε συνοψίζοντας τα βασικά στοιχεία της κλασικής μηχανικής μάθησης και αναλύουμε λεπτομερώς τις εσωτερικές λειτουργίες των νευρωνικών δικτύων και συγκεκριμένα, των παραγωγικών αντιπαραθετικών δικτύων (GANs). Στη συνέχεια, συζητάμε πώς οι κβαντικοί αλγόριθμοι μπορούν να ενσωματωθούν σε κλασικές προσεγγίσεις μηχανικής μάθησης. Αναλύουμε τους δύο κορυφαίους τομείς της κβαντικής μηχανικής μάθησης, την κβαντική μηχανική μάθηση με δυνατότητα διόρθωσης κβαντικών σφαλμάτων, και τους κβαντικούς αλγόριθμους μηχανικής μάθησης φιλικούς με συσκευές που έχουν κβαντικό θόρυβο-NISQ (Noise Intermediate Scale Quantum). Στην πρώτη περίπτωση, οι αλγόριθμοι αναμένεται να έχουν εκθετικές κβαντικές επιταχύνσεις έναντι των κλασικών αντίστοιχων, αλλά απαιτούν κβαντικούς επεξεργαστές με δυνατότητα πλήρους διόρθωσης σε κβαντικά σφάλματα που δεν έχουν ακόμη εφευρεθεί, ενώ η δεύτερη περίπτωση μπορεί να υλοποιηθεί σε τρέχουσες διαθέσιμες κβαντικές συσκευές, χωρίς ωστόσο να έχουν αποδειχθεί κβαντικές επιταχύνσεις ή άλλα κβαντικά πλεονεκτήματα για την ώρα. Στο επόμενο και κύριο μέρος της εργασίας, αναλύουμε μερικούς πρόσφατους, φιλικούς προς τις NISQ συσκευές, αλγόριθμους κβαντικής μηχανικής μάθησης, δηλαδή: υβριδικά κλασικά-κβαντικά μεταβλητά μοντέλα που αποτελούνται από κβαντική και κλασική επεξεργασία. Στο πλαίσιο αυτής της προσέγγισης, μελετάμε λεπτομερώς την κβαντική έκδοση των GAN (QGAN) και δείχνουμε πώς μπορούν να εκπαιδευτούν ώστε να παράγουν κβαντικές καταστάσεις που κωδικοποιούν και μαθαίνουν αποτελεσματικά τις κατανομές πιθανοτήτων. Συγκρίνουμε την απόδοση εκπαίδευσης των QGAN χρησιμοποιώντας διαφορετικές αρχικές κατανομές πιθανοτήτων εισόδου σε διάφορες ρυθμίσεις, π.χ.: συστήματα 3 και 4 qubits και μοντέλα με διαφορετικούς αριθμούς επαναλήψεων κβαντικών κυκλωμάτων. Αυτές οι εκπαιδευμένες κβαντικές καταστάσεις χρησιμοποιούνται στη συνέχεια μαζί με αλγόριθμους εκτίμησης κβαντικού πλάτους, για τον υπολογισμό σημαντικών ποσοτήτων στον χρηματοοικονομικό κόσμο, όπως το Ευρωπαϊκό πρόβλημα προαίρεσης αγοράς. Εφαρμόζουμε τους κβαντικούς αλγόριθμους μας σε κλασικούς προσομοιωτές και συγκρίνουμε την απόδοση για διαφορετικούς αριθμούς qubits και διαμορφώσεις και συζητάμε πιθανές μελλοντικές εφαρμογές σε άλλους τομείς.

Acknowledgements

I would like first and foremost to thank my supervisor Prof. Dimitris Angelakis for the trust he has shown in me in the assignment and opportunity of writing the current thesis. His notes, corrections and talks played a huge part in finishing this work. Following, I would like to thank the members of Mr. Angelakis' group at CQT Singapore for the much interesting and helpful talks and presentations, I had the pleasure and the honor to watch throughout the last year. Especially, I want to give my sincerest gratitude to Gan Beng Yee for always correcting, supervising me and being there whenever I got lost in the abyss of information that is called quantum computing. To all my professors at Technical University of Crete, I would like to thank them, not only for sharing their knowledge on each one of the many different courses I had to take, but also for showing me that to be good at something, you have to love it and enjoy working on it. This has been the basis idea for me before starting the writing of this thesis. To the supervising committee members, Prof. Michail Zervakis and Prof. Dionissios Hristopulos, thank you for your interest and participation in this presentation.

I would also like to thank my friends at Chania. Their support through the years of my studying has been immeasurable and I will always cherish and thank them for all the good moments we had together. Finally, I want to thank my family. My mother and my father who have shown me their unconditional love, trust and support no matter the difficulties and my brother who has always been by my side, silently but meaningfully supporting me and pushing me to get through everything, good or bad. This work is dedicated to you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

'Computers are famous for being able to do complicated things starting from simple programs'

Seth Lloyd

For many years the evolution of computers has been providing great results when solving problems, in times that have been continuously getting better. But nowadays, people are facing a challenge that could potentially lead to a rise of a new era. According to Moore's law, the number of transistors in a dense integrated system, doubles every two years, meaning that every two years the computing power will be getting better. This also means that there will be a point in time where this evolution will not be able to occur, due to the fact that there will be no more physical space to store these transistors. As we continue to miniaturize chips, we'll have to face Heisenberg's uncertainty principle, which limits precision at the quantum level, thus limiting our computational capabilities. In other words, as the size of the transistors decreases and gets closer to the atomic size, the transistor will no longer be able to work as is, due to the appearance of quantum phenomena.

Therefore, people are looking to alternatives, with one of them being, the ever-gaining momentum quantum computing. It can be defined as the rapidly emerging technology that harnesses the laws of quantum mechanics to solve problems too complex for classical computers. The idea is that we use some hardware that enables us to have control over quantum systems. These systems enable us to perform quantum information processing, which is the study of quantum algorithms that rely on quantum properties, some of which we will see in the current thesis.

The most widely used model in quantum computing is the quantum circuit. On a general scope, it utilizes the most basic elements of quantum computing, which are the qubit and the quantum gates. The first is the quantum counterpart of the classical bit, but with a very important twist, which is the ability to be in two states at the same time. For example, a classical bit can be either at state 1 or at state 0 at a specific moment in time. The qubit can be at both states simultaneously providing ground for many quantum-only properties. The second element is the quantum gate, which, as with classical logic gates, can manipulate and change the current state of a qubit.

Quantum computing does in fact offer some very interesting premises. One of them is that any

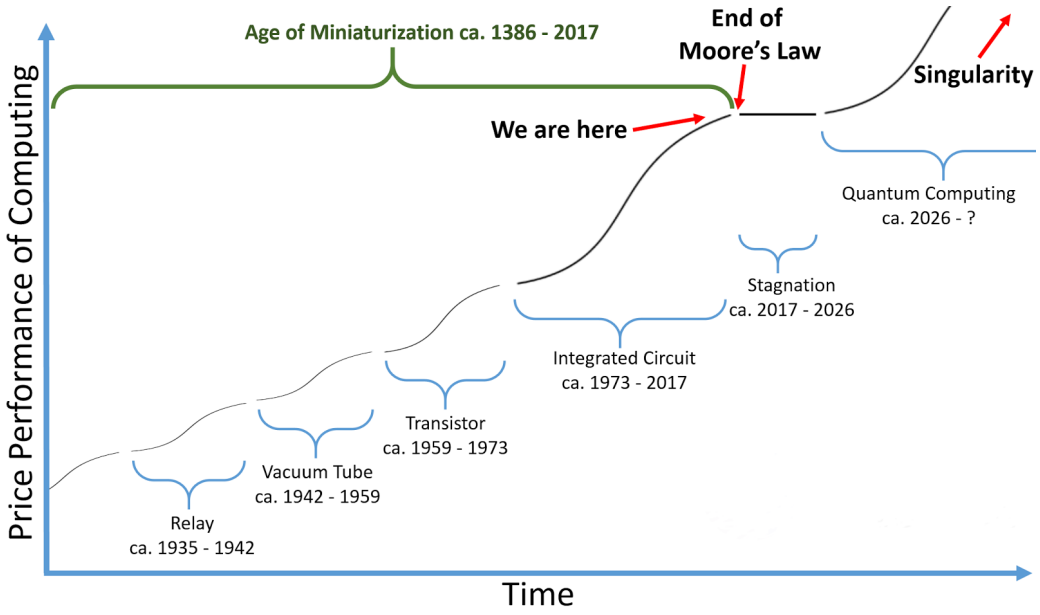


Figure 1.1: The evolution of Moore's law and the need for quantum computing.

computational problem that can be solved by a classical computer has the potential to be solved by a quantum computer as well. The other way around can in principle happen, given enough time. This is a very important statement, since it shows that it is possible to run a problem in a quantum computer and have a speedup in the total computation time. In some cases, we can say that quantum computers are able to solve problems that classical computers are not able to, in a feasible amount of time, which is known as "quantum supremacy". Some examples of this, include factorization in polynomial time [25], using Shor's algorithm. This algorithm is considered to be very powerful and it can also be used to break RSA cryptography [3]. Another very useful algorithm is Grover's, which enables us to search an unordered database with time proportional to $\sqrt{N}$ $(O(\sqrt{N}))$, while the classical approach takes time proportional to N $(O(N))$, where N: the number of entries in the database. We present the implementation of Grover's algorithm at *Appendix A*.

Furthermore, quantum computing finds use in the science of machine learning. As we know classical machine learning methods require huge computational resources, and in many cases, training costs a lot of time. The idea behind it, is that a machine learns from experience and builds its logic using data examples without a user or programmer giving explicit instructions. Quantum Machine Learning (QML) is a research field that combines quantum computing and machine learning technologies to push the boundaries of technological advancement. It utilizes quantum computers' processing power to process data faster than traditional computers.

There are two main waves in quantum machine learning. In the first one, introduced in 2009, implementing the QML algorithms required fault-tolerant quantum computers, which at this moment are not available and still need a lot of study and research. What we have now is the noisy version of these computers. This is the second wave, where we are utilizing these

noisy devises and design QML algorithms with some experimental constraints in mind. Noise is present due to the fragility of quantum phenomena in large scales and temperatures. Quantum systems need to be cooled down to close to absolute zero and isolate from the environment to stay quantum coherent which is important for the potential speed ups. The general idea however is that quantum machine learning algorithms hold in fact the possibility to offer important speedup when compared to the classical ones. For example, we have been able to prove that QML algorithms can provide exponential speedup in principle component analysis where finding the principle components takes $O(\log N^2)$ time, compared to classical's $O(N^2)$. Another example is the support vector machines method whose classical version has a time complexity of $O(poly(N))$, whereas the quantum algorithm achieves the same results in $O(\log N^2)$.

Currently, quantum computing is in the Noisy Intermediate-Scale Quantum (**NISQ**) era, which defines the second wave that we mentioned. While, as we have said, we are taking into account the noise factor, people have been able to achieve some great results using NISQ devices. As an example to illustrate this, in 2019 Google announced that its Sycamore quantum computer had completed a task that would take a conventional computer 10,000 years, in 200 seconds. Although there have been some arguments from the classical computing researchers, it is nonetheless a great showcase of the possibility of achieving quantum supremacy using NISQ devices. NISQ devices are useful tools for exploring many-body quantum physics, and may have other useful applications, but quantum technologists should continue to strive for more accurate quantum gates and, eventually, fully fault-tolerant quantum computing.

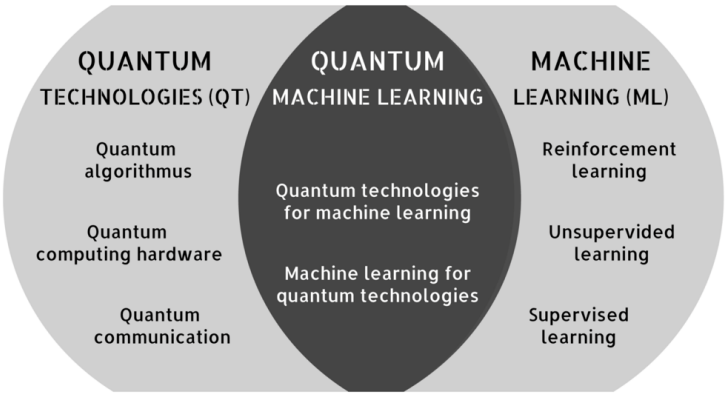


Figure 1.2: QML through quantum computing and classical ML.

Thesis Outline

In this thesis we present in *Chapter 1* the basic concept of quantum computing and quantum machine learning along with its current approaches. We continue in *Chapter 2* by presenting some basic quantum properties along with quantum algorithms that will be the basis for our computations. Following in *Chapter 3*, we will briefly study the basic methods of machine learning,

including neural networks and we will present a model that uses them to train itself in order to generate new data distributions, which is the generative adversarial network (GAN). In *Chapter 4*, we will present the main part of this thesis which is going to be about quantum machine learning with main focus on the quantum analog of the GANs (qGANs). We will describe how they differ from the classic ones, how they work and how one can train them. Finally in *Chapter 5*, we will conclude this work, with some real world applications in the field of Finance. More specifically, we will see how quantum machine learning can tackle every day financial problems, including the use of qGANs to compute some important values in a European Call Option problem.

Chapter 2

Introduction to Quantum Computing

In this chapter, we will introduce the basic idea and concepts of quantum computing. We will start by defining the qubit. Then we will see single and multiple quantum gates and finally we will introduce some of the most important quantum algorithms like the quantum Fourier transform, the quantum phase estimation and the quantum amplitude estimation algorithm that we will be using extensively in the main parts of the work [17].

2.1 The Qubit

A qubit is the quantum counterpart of the classic binary bit. It is the basic unit of quantum information, a two state system consisting of states 0 and 1 firstly introduced in 1992 by Ben Schumacher. The main difference between the quantum and the classical system, is that in the latter, a bit would have to be in one state or the other, while in the quantum system we can have both states simultaneously using a property called **superposition**.

2.2 Single Quantum States

The quantum states, use **Dirac's bra-ket** notation. A ket, $|v\rangle$, denotes a vector $\mathbf{v}$ in an abstract vector space $\mathbf{V}$, and represents a state of some quantum system. A bra, $\langle f|$, is a linear map that maps each vector in $\mathbf{V}$ to a number in the complex plane $\mathbb{C}$. Letting $\langle f|$ act on a vector $|v\rangle$ is written as $\langle f|v\rangle \in \mathbb{C}$.

Superposition

Superposition is one of the basic properties of quantum information. It is based on the idea that quantum states can be in states $|0\rangle$, $|1\rangle$, but also in a linear combination of those two, as:

$$|\psi\rangle = a\,|0\rangle + b\,|1\rangle\,,$$

where $a, b \in \mathbb{C}$ are the state amplitudes.

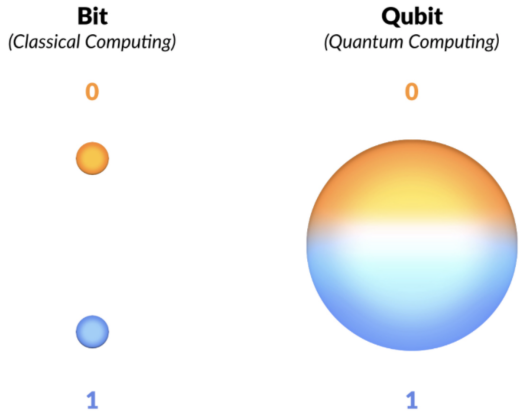


Figure 2.1: Difference between qubit and bit. While a bit can take one value (0 or 1) at a time, a qubit can take both values at the same time using superposition.

Matrix Representation

Qubit $|0\rangle$ can be written in a vector form as: $\begin{bmatrix}1\\0\end{bmatrix}$, while qubit $|1\rangle$ as: $\begin{bmatrix}0\\1\end{bmatrix}$. So we can write :

$$|\psi\rangle = a\,|0\rangle + b\,|1\rangle = a\begin{bmatrix}1\\0\end{bmatrix} + b\begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}a\\0\end{bmatrix} + \begin{bmatrix}0\\b\end{bmatrix} = \begin{bmatrix}a\\b\end{bmatrix}.$$

If we consider two quantum states:

$$|\psi\rangle = a\,|0\rangle + b\,|1\rangle \quad \text{and} \quad |x\rangle = c\,|0\rangle + d\,|1\rangle\,,$$

with $a, b, c, d \in \mathbb{C}$, we can define their inner and outer product as following:

Inner Product: $\langle\psi|x\rangle = \begin{bmatrix}a^* & b^*\end{bmatrix}\begin{bmatrix}c\\d\end{bmatrix} = a^*c + b^*d.$

Outer Product: $|\psi\rangle\langle x| = \begin{bmatrix}a\\b\end{bmatrix}\begin{bmatrix}c^* & d^*\end{bmatrix} = \begin{bmatrix}ac^* & ad^*\\bc^* & bd^*\end{bmatrix}.$

Numbers a^*, b^* are the complex conjugates of a,b. As a complex conjugate of a complex number $z = a + ib$, we define the complex number $z^* = a - ib$.

Orthogonal: For states $|x\rangle$, $|\psi\rangle$ to be orthogonal, they have to confirm the following equations:

$$\langle x|\psi\rangle = 0 \quad \text{and} \quad \langle\psi|x\rangle = 0$$

For example, for qubits $|0\rangle$ and $|1\rangle$ we have:

$$\langle 1|0\rangle = \begin{bmatrix}0 & 1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = 0 \text{ and } \langle 0|1\rangle = \begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = 0$$

Normalized: Similar for states $|x\rangle$, $|y\rangle$ to be normalized, they have to confirm the following equations:

$$\langle\psi|\psi\rangle = 1 \quad \text{and} \quad \langle x|x\rangle = 1$$

For qubits $|0\rangle$ and $|1\rangle$, we have:

$$\langle 1|1\rangle = \begin{bmatrix}0 & 1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} = 1 \text{ and } \langle 0|0\rangle = \begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = 1$$

Born's Rule

Consider a normalized state $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$. This means, as we have seen, that: $\langle\psi|\psi\rangle = 1$. The **Born rule** states, that if we measure state $|\psi\rangle$, the probability of finding the qubit in state $|0\rangle$ is $P_0 = |a|^2$, while the probability to find it in state $|1\rangle$ is $P_1 = |b|^2$. Since we are dealing with probabilities, there is a strict condition we should always follow, which is: $|a|^2 + |b|^2 = 1$. Of course, when we are working with more qubits, the same rule applies.

Bloch Sphere

Named after physicist Felix Bloch, it is a geometrical representation of the quantum state of a quantum system. We can write a random state $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$ as:

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

Below is the sphere with the corresponding angles θ, ϕ:

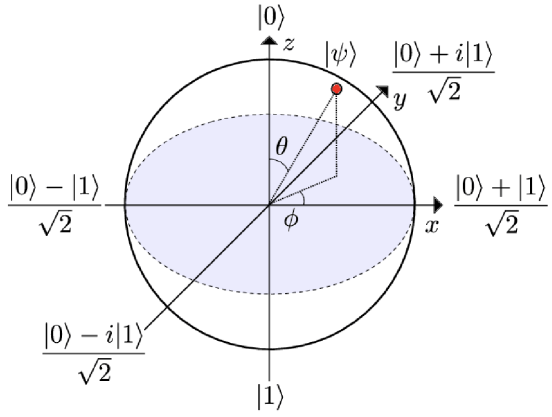


Figure 2.2: Bloch Sphere

2.3 Single Qubit Gates

After introducing quantum states, we need a way to change the qubit between them. This is possible with the use of quantum gates. Nowadays, the number of different gates has grown significantly, therefore we will present the basic ones in the following subsections.

The Identity Gate

It is a single-qubit operation that leaves any state that it applied on unchanged. The matrix representing this gate is the following:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

It doesn't change anything when acting on qubits, but is often used when we want to define a 'do nothing' operation.

The Pauli Gates

They act on a single qubit and change its state. They are based on the Pauli matrices, namely σ_x, σ_y, σ_z.

We have gates X, Y, Z which are based on those matrices and are as following:

- $\mathbf{X} = \sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, which acts on qubits as: $\mathbf{X}|0\rangle = |1\rangle$ and $\mathbf{X}|1\rangle = |0\rangle$
- $\mathbf{Y} = \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$, which acts on qubits as: $\mathbf{Y}|0\rangle = i|1\rangle$ and $\mathbf{Y}|1\rangle = \text{-}i|0\rangle$
- $\mathbf{Z} = \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$, which acts on qubits as: $\mathbf{Z}|0\rangle = |0\rangle$ and $\mathbf{Z}|1\rangle = \text{-}|1\rangle$

We can note one of the basic properties of Pauli matrices, which is that each matrix squared is equal to the identity matrix: $X^2 = Y^2 = Z^2 = I$

The Hadamard Gate

It is the main gate that is used to create a superposition of a state. The gate is: $\mathbf{H} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$, and acts as following: $\mathbf{H}|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $\mathbf{H}|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

The Phase Shift Gates

It is a parameterised gate, meaning that based on a value of a parameter ϕ the gate performs a phase shift as: $\mathbf{P}(\phi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix}$ and acts as: $\mathbf{P}|0\rangle = |0\rangle$ and $\mathbf{P}|1\rangle = e^{i\phi}|1\rangle$.

We have to note here that the amplitude (probability of qubit measurement) does not change, when performing a phase shift. Some other gates can be derived from this gate. For example, by setting $\phi = \pi$ and using the property $e^{i\pi} = -1$ we obtain the Z-gate as:

$$\mathbf{P}(\pi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \mathbf{Z}$$

The Rotation Gates

They use an angle θ and rotate the qubit in the Bloch sphere around the wanted axis. For the proof of how these gates are constructed, we refer you to **Ref**. [5].

- The *rotation around axis X* is performed using matrix:
  $R_X(\theta) = e^{i\frac{\theta}{2}X} = \cos\frac{\theta}{2}I - i\sin\frac{\theta}{2}X = \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$
- The *rotation around axis Y* is performed using matrix:
  $R_Y(\theta) = e^{-i\frac{\theta}{2}Y} = \cos\frac{\theta}{2}I - i\sin\frac{\theta}{2}Y = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$
- The *rotation around axis Z* is performed using matrix:
  $R_Z(\theta) = e^{i\frac{\theta}{2}Z} = \cos\frac{\theta}{2}I - i\sin\frac{\theta}{2}Z = \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix}$

The general U Gate

Finally we present the **U Gate**, which is the most general and parameterised gate, depending on the eigenvalues (λ), eigenvectors(ϕ) and angle (θ). Every gate that we presented, can be derived from the U-Gate. The gate is the following:

$$U(\theta,\lambda,\phi) = \begin{bmatrix} \cos\frac{\theta}{2} & -e^{i\lambda}\sin\frac{\theta}{2} \\ e^{i\phi}\sin\frac{\theta}{2} & e^{i(\phi+\lambda)}\cos\frac{\theta}{2} \end{bmatrix}$$

2.4 Multiple Qubit States

So far we have seen the single qubit gates, used on single quantum states. We can use a mathematical operation, the tensor product, to construct multiple qubit states. The **tensor product** is the way of putting vector spaces together to form larger vector spaces. For example, lets say we have vectors **v** and **w** of dimensions n and m respectively in the corresponding Hilbert spaces **V,W**. We define $V \otimes W$ as an nm dimensional vector space, as: $|v\rangle \otimes |w\rangle = |v\rangle\,|w\rangle = |vw\rangle$. For example, given the known qubits $|0\rangle$ and $|1\rangle$ we can get: $|0\rangle\otimes|0\rangle+|1\rangle\otimes|1\rangle = |0\rangle\,|0\rangle+|1\rangle\,|1\rangle = |00\rangle+|11\rangle$. One useful notation is: $|\psi\rangle^{\otimes n} = \underbrace{|\psi\rangle \otimes |\psi\rangle \ldots \otimes |\psi\rangle}_{n}$. From these properties we can obtain the following vectors:

$$|0\rangle \otimes |0\rangle = |00\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \qquad |0\rangle \otimes |1\rangle = |01\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$$

$$|1\rangle \otimes |0\rangle = |10\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \qquad |1\rangle \otimes |1\rangle = |11\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}.$$

So we can write the 2-qubit system as following: $|\psi\rangle = \alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|10\rangle + \delta\,|11\rangle$, where: $(\alpha,\beta,\gamma,\delta) \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$

2.4.1 Multiple Qubit Gates

There are many types of multiple qubit gates. One broad category of these, is the **controlled gates**, having the general **Control-U** gate and then more specifically the **Control - NOT (CNOT)**, **Control - Z (CZ)** gates. They act on 2 qubits, the **control** qubit and the **target** qubit. The idea is that, when the control qubit is at state $|0\rangle$, nothing happens, but when control qubit is $|1\rangle$ then the U-gate acts on target qubit. This means that the **Control-U** gate is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{11} & u_{12} \\ 0 & 0 & u_{21} & u_{22} \end{bmatrix} \quad \text{with} \quad \mathbf{U} = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{21} \end{bmatrix}, \tag{2.1}$$

the general U-gate we defined earlier.

The **Control-NOT** gate action is based on the XOR logic gate. As a reminder the XOR logic gate acts as following:

Input 1	Input 2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Table 1.1. XOR truth table

The **CNOT** gate, also known as CX gate is:

$$\mathbf{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & x_{11} & x_{12} \\ 0 & 0 & x_{21} & x_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \mathbf{0} & \mathbf{1} \\ 0 & 0 & \mathbf{1} & \mathbf{0} \end{bmatrix},$$

where $x_{11}, x_{12}, x_{21}, x_{22}$ the elements of X Pauli-matrix.
The way this gate acts on the control qubit is: $|x, y\rangle \to |x, x \oplus y\rangle$, meaning that:

$$\begin{aligned} |0,0\rangle \to |0, 0 \oplus 0\rangle = |0,0\rangle\,, \qquad |0,1\rangle \to |0, 0 \oplus 1\rangle = |0,1\rangle\,, \\ |1,0\rangle \to |1, 1 \oplus 0\rangle = |1,1\rangle\,, \qquad |1,1\rangle \to |1, 1 \oplus 1\rangle = |1,0\rangle\,. \end{aligned}$$

The **CZ** gate is:

$$\mathbf{CZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & z_{11} & z_{12} \\ 0 & 0 & z_{21} & z_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \mathbf{1} & \mathbf{0} \\ 0 & 0 & \mathbf{0} & \mathbf{-1} \end{bmatrix},$$

where $z_{11}, z_{12}, z_{21}, z_{22}$ the elements of Z Pauli-matrix.
The way this gate acts on the control qubit is: $|x, y\rangle \to |x, (-1)^x y\rangle$, meaning that:

$$\begin{aligned} |0,0\rangle \to |0, (-1)^0 0\rangle = |0,0\rangle\,, \qquad |0,1\rangle \to |0, (-1)^0 1\rangle = |0,1\rangle\,, \\ |1,0\rangle \to |1, (-1)^1 0\rangle = -|1,0\rangle\,, \qquad |1,1\rangle \to |1, (-1)^1 1\rangle = -|1,1\rangle\,. \end{aligned}$$

Finally, we introduce another type of two qubit gates which is the **SWAP** gate: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$.

The way it acts, is that it swaps the state of the two qubits involved in the operation.

2.5 Basic Quantum Algorithms

A quantum algorithm, is an algorithm that runs on a quantum computer, using gates and oracles that form what we call as quantum circuits. It is a step by step procedure that makes use of the quantum properties mention in the subsections above and can possibly produce results, much better than the corresponding classic ones. Some of the algorithms that we are going to study are **Quantum Fourier Transform (QFT), Quantum Phase Estimation (QPE)** and **Quantum Amplitude Estimation (QAE)**.

2.5.1 Quantum Circuit

A **quantum circuit** is a model, containing qubits and quantum gates, which combined produce the wanted results. It is read from left to right. In the example below, we can see the structure of a simple quantum circuit. The horizontal lines are the qubits, and the boxes are the single-qubit quantum gates. Also the circles are two qubit gates that are connected with the control-qubit. The box on the right end of the circuit indicates that a measurement of the first qubit occurs.

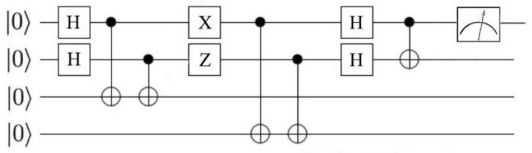


Figure 2.3: Simple quantum circuit

2.5.2 Quantum Fourier Transform

The *Fourier transform* in classic computation is a very powerful tool to transform and study a signal from time to frequency domain. One of the most important uses of it is the ability to remove noise, due to the fact that we are working with frequencies, and we can therefore eliminate the high ones. The mathematical formula this transformation follows is that given a set of variables x_j we can get:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{\frac{-2\pi i k j}{N}} x_j \tag{2.2}$$

There exists as well an *inverse Fourier transform* enabling to do the exact reverse process and move from frequency to time domain. It's formula is:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{\frac{2\pi i k j}{N}} x_j \tag{2.3}$$

As with many algorithms and processes, there is a quantum version of it as well, the **Quantum Fourier Transform** (QFT) and the **inverse Quantum Fourier Transform** (iQFT) with functionalities and results, that we will discuss in the following sections. We have to note that the quantum analog of the inverse Fourier transform is the QFT and obviously Fourier transform's analog is the inverse quantum Fourier transform.

Preliminaries

Let's consider a quantum state $|\psi\rangle = \sum_{j=0}^{N-1} a_j |j\rangle$. If we apply QFT on this state, we obtain

$$|\psi'\rangle = \sum_{k=0}^{N-1} = b_k |k\rangle \, , where : b_k = \frac{1}{\sqrt{N}} \frac{1}{N} \sum_{j=0}^{N-1} a_j e^{\frac{2\pi i k j}{N}}$$

and therefore:

$$|\psi'\rangle = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \sum_{j=0}^{N-1} a_j e^{\frac{2\pi i k j}{N}} |k\rangle$$

As with all quantum algorithms we have to define an operator that will do as we want. The operator for this transformation is $\hat{F}$. It has to be unitary, meaning that $\hat{F}^\dagger \hat{F} = I$, and can be defined as:

$$\hat{F} = \sum_{j,k=0}^{N-1} \frac{e^{\frac{2\pi i k j}{N}}}{\sqrt{N}} |k\rangle \langle j| \quad \text{and} \quad \hat{F^\dagger} = \sum_{j,k=0}^{N-1} \frac{e^{-\frac{2\pi i k j}{N}}}{\sqrt{N}} |j\rangle \langle k| \tag{2.4}$$

So for state $|\psi\rangle$, we have:

$$\hat{F} |\psi\rangle = \sum_{j,k=0}^{N-1} \frac{e^{\frac{2\pi i k j}{N}}}{\sqrt{N}} |k\rangle \langle j| \sum_{j=0}^{N-1} a_j |j\rangle = \sum_{j,k=0}^{N-1} \frac{e^{\frac{2\pi i k j}{N}}}{\sqrt{N}} a_j |k\rangle \tag{2.5}$$

The **Fast Fourier Transform** (FFT) is an algorithm that determines the Fourier transform of an input significantly faster. It uses a technique that splits the problem in pairs of two. After that the transformation occurs between these two and finally we merge the solutions to get the result we want. The quantum Fourier transform is based on the idea of FFT. We have to introduce two notations:

$$j = j_1 j_2 ... j_n = j_1 2^{n-1} + j_2 2^{n-2} + ... + j_n \tag{2.6}$$

$$0.j_l j_{l+1} ... j_m = \frac{j_l}{2} + \frac{j_{l+1}}{4} + ... + \frac{j_m}{2^{m-l+1}} \tag{2.7}$$

For example if we have $j = 2$, we can write j in a binary form 10 with $j_1 = 1$ and $j_2 = 0$, then the equations above give the result:

$$\begin{aligned} j &= j_1 j_2 \\ &= j_1 2^{n-1} + j_2 2^{n-2} \\ &= 1 * 2^{2-1} + 0 * 2^{2-2} \\ &= 1 * 2 + 0 * 2^0 = 1 * 2 = 2 \end{aligned}$$

and

$$\begin{aligned} 0.j_l j_{l+1} &= \frac{j_l}{2} + \frac{j_{l+1}}{4} \\ &= \frac{1}{2} + \frac{0}{4} = 0.5 \end{aligned}$$

The QFT maps the state $|\psi\rangle = |j_n ... j_1\rangle$ into:

$$\frac{(|0\rangle + e^{2\pi i 0.j_n} |1\rangle) \otimes (|0\rangle + e^{2\pi i 0.j_{n-1} j_n} |1\rangle) \otimes ... \otimes (|0\rangle + e^{2\pi i 0.j_1 j_2 ... j_n} |1\rangle)}{\sqrt{2^n}} \tag{2.8}$$

As we can see there is a superposition of states happening, along with a phase rotation of qubit $|1\rangle$. The first occurs when using a simple Hadamard gate (H) and the latter is possible using a phase shift gate, as seen on subsection 2.3 and is: $\hat{P}_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{2\pi i}{2^k}} \end{bmatrix}$. The way this rotation works is simple. When acting on qubit $|0\rangle$ we get: $P_k |0\rangle = |0\rangle$, while when acting on qubit $|1\rangle$ we get the wanted rotation: $P_k |1\rangle = e^{\frac{2\pi i}{2^k}} |1\rangle$.

Following is the circuit implementation of QFT:

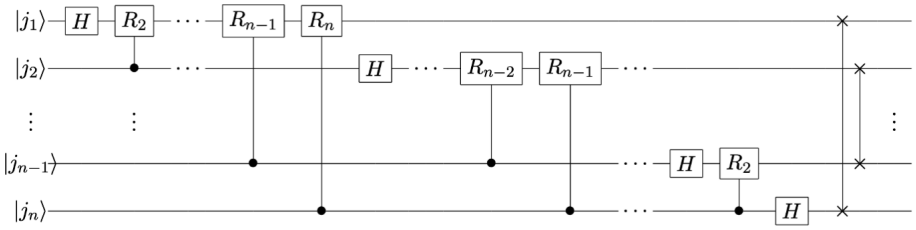


Figure 2.4: Quantum Fourier transform circuit

The Algorithm

Assume that we have two states of n qubits each: $|x\rangle = |x_{n-1} x_{n-2} ... x_0\rangle$ and $|y\rangle = |y_{n-1} y_{n-2} ... y_0\rangle$, where $x_i, y_i \in \{0, 1\}$. The QFT maps state $|x\rangle$ into the state:

$$\frac{1}{\sqrt{2^n}} (|0\rangle + e^{2\pi i [0.x_{n-1}]} |1\rangle) \otimes (|0\rangle + e^{2\pi i [0.x_{n-2} x_{n-1}]} |1\rangle) ... (|0\rangle + e^{2\pi i [0.x_0 ... x_{n-1} x_{n-2}]} |1\rangle) \tag{2.9}$$

As we mentioned above, the QFT has its corresponding operator: $\hat{F}$. This operator when acting on qubit $|x\rangle$ produces the result:

$$\begin{aligned} F|x\rangle &= \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{\frac{2\pi i x y}{2^n}} |y\rangle \\ &= \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{2\pi i (y_0(0.x_0x_1...x_{n-1}) + y_1(0.x_0x_1...x_{n-2}) + ... + y_{n-1}(0.x_0))} |y\rangle \\ &= \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{2\pi i (y_0(0.x_0x_1...x_{n-1})} |y_0\rangle \otimes e^{2\pi i (y_1(0.x_0x_1...x_{n-2}))} |y_1\rangle \otimes ... \otimes e^{2\pi i (y_{n-1}(0.x_0))} |y_{n-1}\rangle \end{aligned}$$

, where we have used the property that $e^{a+b} = e^{ae^b}$ and $|y\rangle = |y_0\rangle \otimes |y_1\rangle \otimes ... \otimes |y_{n-1}\rangle$ [17].

Since we have $|y_i\rangle \in \{0, 1\}$, we have the following cases

- If $y_i = 0$, then $e^{2\pi i (y_0(0.x_0x_1...x_{n-1})} = 1$
- If $y_i = 1$, then we reach equation (2.10).

2.5.3 Quantum Phase Estimation Algorithm

Firstly introduced in 1995, by Russian physicist Alexei Kitaev, **Quantum Phase Estimation** (QPE) is an algorithm used to estimate the phase (eigenvalue) of an eigenvector of a unitary operator and it is generally used as a powerful tool for many quantum algorithms.

Before we begin with the algorithm analysis, we remind the properties of eigenvectors and eigenvalues. Let there be a unitary matrix A, with the apparent property of $A^\dagger A = I$. An eigenvalue of the matrix (λ), also referred as its phase, is connected to its corresponding eigenvector (v) as:

$$A = \sum_{i=1}^{n} \lambda_i |v_i\rangle \langle v_i| , \tag{2.10}$$

where n the total number of eigenvalues. We can write that $\lambda_i = e^{2\pi i \theta}$, where θ the phase angle.

Let U be a unitary operator in a $d = 2^n$ Hilbert space and $|\phi_1\rangle, |\phi_2\rangle, ..., |\phi_d\rangle$ be eigenvectors of U. Operator U acts on $|\phi_n\rangle$ as:

$$U|\phi_n\rangle = e^{2\pi i \theta_n} |\phi_n\rangle \tag{2.11}$$

and consequently

$$U^j |\phi_n\rangle = U^{j-1} e^{2\pi i \theta_n} |\phi_n\rangle = (e^{2\pi i \theta_n})^j |\phi_n\rangle = e^{2\pi i \theta_n j} |\phi_n\rangle \tag{2.12}$$

The Problem

Given a unitary operator U and an input eigenvector $|\phi\rangle$ of U, estimate the angle θ in the associated eigenvalue that will have a form given by: $U|\phi_n\rangle = e^{2\pi i \theta_n} |\phi_n\rangle$.

We suppose that we want to know the phase angle θ to m bits of accuracy.

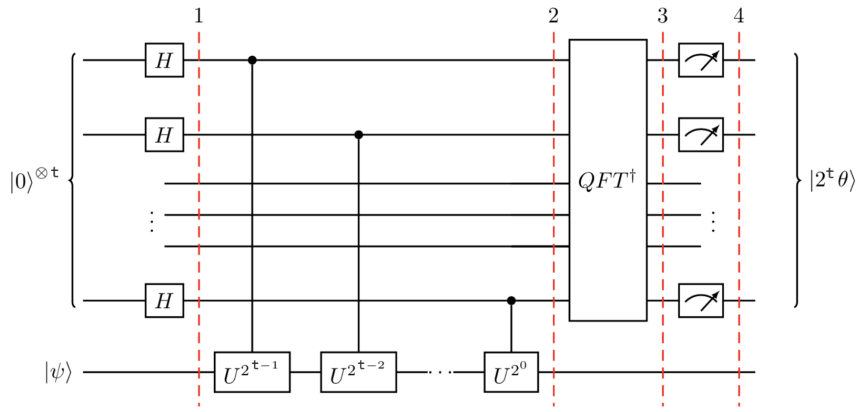


Figure 2.5: Quantum Phase Estimation Circuit

The Algorithm

The process involves two parts. A part of n-$|0\rangle$ qubits and a part with m remaining qubits in state $|\psi\rangle$. We suppose that we want to know θ to m bits of accuracy. Hence, the original state of the algorithm is $|\psi_{in}\rangle = |0\rangle^{\otimes n} |\phi\rangle_n$, with $|\phi\rangle_n = |\phi_{n-1}, \phi_{n-2}, ..., \phi_1, \phi_0\rangle$.

- **Step 1**: Apply Hadamard gate to the first m qubits. The target here is to generate a superposition states:

$$|\psi'\rangle = H^{\otimes m} |\psi_{in}\rangle = (\frac{|0\rangle + |1\rangle}{\sqrt{2}})^{\otimes m} |\phi\rangle_n = \frac{1}{2^{m/2}} \sum_{x=0}^{2^m-1} |x\rangle |\phi\rangle_n \tag{2.13}$$

- **Step 2**: Apply the unitary operator U^x to state $|\phi\rangle$ as:

$$|\psi''\rangle = \frac{1}{2^{m/2}} \sum_{x=0}^{2^m-1} |x\rangle (U^x |\phi\rangle_n) = \frac{1}{2^{m/2}} \sum_{x=0}^{2^m-1} |x\rangle (e^{2\pi i\theta x} |\phi\rangle_n) = \frac{1}{2^{m/2}} \sum_{x=0}^{2^m-1} (|x\rangle e^{2\pi i\theta x}) |\phi\rangle_n \tag{2.14}$$

- **Step 3**: We apply the inverse Fourier transform ($F^\dagger$) to convert it to the computational basis. This action yields:

$$F^\dagger |\psi_{out}\rangle = \frac{1}{2^m} \sum_{x=0}^{2^m-1} \sum_{y=0}^{2^m-1} e^{(-2\pi xy/2^m)+(2\pi i\theta x)} |x\rangle |\phi\rangle_n = \frac{1}{2^m} \sum_{x,y=0}^{2^m-1} e^{2\pi ix((2^m\theta - y)/2^m)} |x\rangle |\phi\rangle_n \tag{2.15}$$

The Measurement

From the above equations, we want to find an estimation of θ and therefore we need to perform a measurement. To do this, we have to check two possible scenarios:

- If $(2^m\theta)$ is an integer. Then by measuring the probability $Pr(2^m\theta)$ we get:

$$\begin{aligned} Pr(2^m\theta) &= |\langle 2^m\theta| \frac{1}{2^m} \sum_{x,y=0}^{2^m-1} e^{2\pi i x(2^m\theta - y)/2^m} |x\rangle|^2 \\ &= |\langle 2^m\theta| \frac{1}{2^m} \sum_{x=0}^{2^m-1} e^{2\pi i x(2^m\theta - 2^m\theta)/2^m} |2^m\theta\rangle|^2 \\ &= \frac{1}{2^{2m}} | \sum_{x=0}^{2^m-1} e^{2\pi i x(0)/2^m} \langle 2^m\theta|2^m\theta\rangle |^2 \\ &= \frac{1}{2^{2m}} \sum_{x,y=0}^{2^m-1} e^0 = \frac{2^{2m}}{2^{2m}} = 1 \end{aligned}$$

We can note that with probability equal to 1 we can make a measurement of θ to m bits of precision with every single bit correct.

- If $(2^m\theta)$ is not an integer. Then we can approximate the value of θ with a certain probability, by rounding $2^m\theta$ to the nearest integer. We can write: $2^m\theta = 2^m\delta + a$, where: a is the nearest integer to $2^m\theta$ and $0 \leq |2^m\delta| \leq \frac{1}{2}$.

We have then the final state as:

$$|\psi_{out}\rangle = \frac{1}{2^m} \sum_{x,y=0}^{2^m-1} e^{2\pi i x((2^m\delta + a) - y)/2^m} |y\rangle = \frac{1}{2^m} \sum_{x,y=0}^{2^m-1} e^{2\pi i x(a-y)/2^m} e^{2\pi i x\delta} |x\rangle . \qquad (2.16)$$

Now, we find the probability of finding the best possible approximation a:

$$\begin{aligned}
Pr(|a\rangle) &= |\langle a| \frac{1}{2^m} \sum_{x,y=0}^{2^m-1} e^{2\pi i x(a-y)/2^m} e^{2\pi i x\delta} |x\rangle|^2 \\
&= |\langle a| \frac{1}{2^m} \sum_{x=0}^{2^m-1} e^{2\pi i x(a-a)/2^m} e^{2\pi i x\delta} |a\rangle |^2 \\
&= \frac{1}{2^{2m}} | \sum_{x=0}^{2^m-1} e^{2\pi i x(0)/2^m} e^{2\pi i x\delta} \langle a|a\rangle |^2 \\
&= \frac{1}{2^{2m}} | \sum_{x,y=0}^{2^m-1} e^{2\pi i x\delta}|^2 \\
&= \frac{1}{2^{2m}} |\frac{1-e^{2\pi i\delta 2^m}}{1-e^{2\pi i\delta}}|^2 \\
&= \frac{1}{2^{2m}} |\frac{2\sin(\pi 2^m\delta)}{2\sin(\pi\delta)}|^2 \\
&= \frac{1}{2^{2m}} |\frac{\sin(\pi 2^m\delta)}{\sin(\pi\delta)}|^2 \\
&\geq \frac{1}{2^{2m}} |\frac{\sin \pi 2^m\delta}{\pi\delta}|^2 \\
&\geq \frac{1}{2^{2m}} |\frac{2\cdot 2^m\delta}{\pi\delta}|^2 = \frac{4}{\pi^2} \approx 0.4
\end{aligned}$$

The bottom line of this calculation is that the probability is greater than 0.4 that a measurement of θ to m bits of precision is obtained with every single bit correct.

2.6 Quantum Amplitude Estimation Algorithm

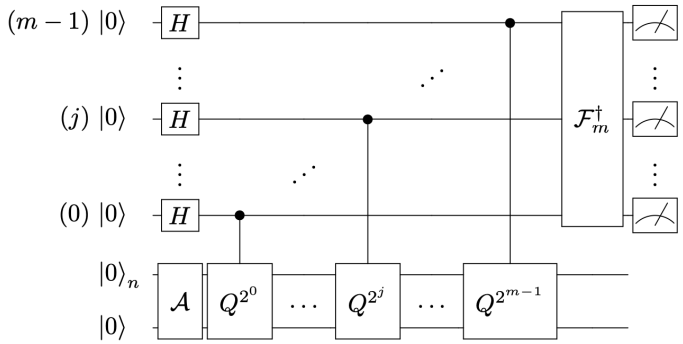


Figure 2.6: Quantum Circuit for Amplitude Estimation

We suppose a unitary operator A, acting on a register of $n+1$ qubits such that

$$A\left|0\right\rangle_n\left|0\right\rangle = \sqrt{1-a}\left|\psi_0\right\rangle_n\left|0\right\rangle + \sqrt{a}\left|\psi_1\right\rangle_n\left|1\right\rangle. \tag{2.17}$$

The quantum states $\left|\psi_0\right\rangle_n$ and $\left|\psi_1\right\rangle_n$ are normalized states but not necessary orthogonal. Typically, one would want to estimate amplitude of $\left|\psi_1\right\rangle_n$, and this can be achieved using the Quantum Amplitude Estimation (QAE) algorithms. Below we outline the steps of the QAE algorithms to estimate the amplitude and the circuit representation of QAE is shown in Fig.2.6 where its circuit is closely related to the circuit in quantum phase estimation algorithm.

First, we initialize $(m+n+1)$ qubits in zero states, then apply Hadamard gates on the first m qubits where these m qubits will serve as control qubits while applying the operator A to the last $n+1$. A control unitary operator Q is then applied iteratively to the $n+1$ qubits, with the power of the operators conditioning on the qubit numbers of the first m qubits. The operator Q is composed of the operators A, S_0 and S_{ψ_0}.

$$Q = AS_0A^\dagger S_{\psi_0}, \tag{2.18}$$

where

$$S_0 = (\mathbf{I} - 2\left|0\right\rangle_{n+1}\left\langle 0\right|_{n+1}) \quad \text{and} \quad S_{\psi_0} = (\mathbf{I} - 2\left|\psi_0\right\rangle_n\left|0\right\rangle\left\langle\psi_0\right|_n\left\langle 0\right|),$$

and every application of Q corresponds to one quantum sample. If we take $n=0$, we will have: $Q = A(\mathbf{I} - 2\left|0\right\rangle\left\langle 0\right|)A^\dagger(\mathbf{I} - 2\left|0\right\rangle\left\langle 0\right|))$, reducing the operator Q to Pauli Z-gate, so we can write the operator Q as: $Q = AZA^\dagger Z$. In general, the operator Q requires very deep circuit to implement. Fortunately, a simplification can be made by letting $A = R_Y(\theta)$, and the operator Q will reduce to $Q = R_Y(2\theta)$, which is relatively easier to implement on current NISQ devices. Finally, an inverse Quantum Fourier Transform will be applied to the first m qubits, followed by measurements in the computational basis. The result of the measurement, is an integer $y \in \{0, ..., 2^m - 1\}$, which is classically mapped to the estimator as:

$$\bar{a} = sin^2(y\pi/M) \in [0, 1] \tag{2.19}$$

According to **Ref.**[4], estimation $\bar{a}$ along with a satisfies:

$$|a - \bar{a}| \leq \frac{2\sqrt{a(1-a)}}{\pi}M + \frac{\pi^2}{M^2} \leq \frac{\pi}{M} + \frac{\pi^2}{M^2} = O(M^{-1})$$

with probability $\frac{8}{\pi^2}$.

Quantum Monte Carlo Algorithm

Monte Carlo (MC) is a well established method that is used to estimate statistical quantities of some complex systems. It is ubiquitous across science and finds application in various areas, ranging from the evaluation of integrals, the pricing of financial derivatives, and the computation of distances between probability distributions. Here, we will focus on one specific use of Monte Carlo methods, i.e: estimating the expected value of a function $f(x)$ of one (or more) variable over some probability distribution ($E[f(x)]$). The MC methods first sample M samples from the target probability distribution, and then use those samples to perform an empirical estimation of the statistical quantities associated to the probability distribution with the estimation error scaling

as $O(1/\sqrt{M})$. In this subsection, we will present the use of QAE algorithm as the sub-routine of quantum Monte Carlo methods, i.e: the quantum generalization of the classical Monte Carlo methods, to estimate the expectation and variance of random variables, using quadratically less samples if compared with its classical counterpart, i.e: it scales as $O(1/M)$ [9, 20, 28, 29].

First, we must represent X as a quantum state. Using n-qubits we map X to the interval $\{0, ..., 2^n - 1\}$, where $N = 2^n$. The classic variable X can be represented by quantum state:

$$|\psi\rangle_n = \sum_{i=0}^{N-1} \sqrt{p_i}\,|i\rangle_n \tag{2.20}$$

with

$$\sum_{i=0}^{N-1} p_i = 1$$

Here, p_i is the probability of measuring the state $|i\rangle$. This state $|i\rangle$ is one of the N realizations of X. Remembering we want to create a superposition of $|0\rangle$ and $|1\rangle$ and from that we will measure the probability of being in state $|1\rangle$. In order to achieve this, we define firstly a function $f : \{0, ..., N-1\} \to [0, 1]$ with its corresponding operator F that acts as:

$$F : |i\rangle_n |0\rangle \to |i\rangle_n \sqrt{1 - f(i)}\,|0\rangle + \sqrt{f(i)}\,|1\rangle \tag{2.21}$$

for all $i \in \{0, ..., 2^n - 1\}$. Now, applying F to state $|\psi\rangle_n |0\rangle$ as:

$$F\,|\psi\rangle_n |0\rangle = \sum_{i=0}^{2^n-1} \sqrt{1 - f(i)}\sqrt{p_i}\,|i\rangle_n |0\rangle + \sqrt{f(i)}\sqrt{p_i}\,|i\rangle_n |1\rangle \tag{2.22}$$

So now, we can approximate the wanted probability of measuring $|1\rangle$, which is:

$$\sum_{i=0}^{2^n-1} f(i)p_i$$

By definition of the expected value, we have: $E[X] = \sum_i p_i X(i)$. So we can say that we can compute the expected value of $f(X)$: $\mathbf{E[f(X)]} = \sum_{i=0}^{2^n-1} f(i)p_i$.

- By choosing $f(i) = \frac{i}{N-1}$, we estimate $E[\frac{X}{N-1}]$ and hence $E[X]$.
- By choosing $f(i) = \frac{i^2}{(N-1)^2}$, we estimate $E[\frac{X}{N-1}^2]$ and hence $E[X^2]$.

So we can compute the variance of X as well, which is: $\mathbf{var(X)} = E[X^2] - E[X]^2$

Example

We will now present an example, where we use the quantum amplitude estimation algorithm to estimate the probability of $p = 0.6$

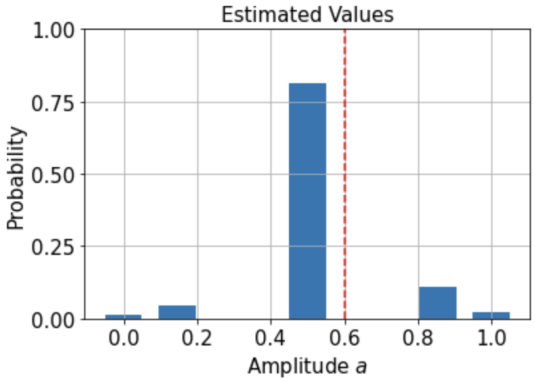
We will see again about **QAE** in *Chapter 6*, as we will see some applications of this algorithm in the field of finance. We will define operators A, Q as described in the previous section:

$$A = R_Y(\theta_p) \quad \text{and} \quad Q = R_Y(2\theta_p),$$

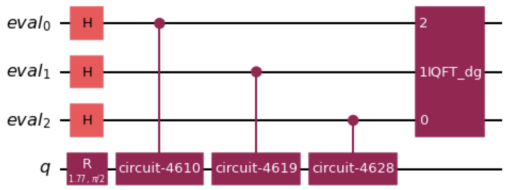
with

$$\theta_p = 2\sin(\sqrt{p}).$$

We use 3 evaluation qubits, but the more we use the better the estimation is going to be. The estimated result after running the algorithm with these two circuits we reach a result of 0.5. This can be illustrated below. The red dotted line is the probability we want to reach and as we can see the estimation produced by the algorithm is at 0.5.



(a)



(b)

Figure 2.7: Figure (a) shows the amplitude estimation with QAE algorithm. In figure (b) you can see the quantum circuit of the QAE algorithm. The code for this example can be seen in the appendix.

If we use 4 evaluation qubits, we can reach an estimation at about 0.69 and with 5 qubits the estimation gets even better at 0.59. However, since we want to use only 3 qubits, we want to be able to estimate better results than 0.5.

Due to the fact that the classical QAE uses phase estimation and inverse Quantum Fourier transformation, this can become very complex. Therefore, there are some new variations that will help reduce gates and computational cost, such as *iterative quantum amplitude estimation*, *maximum likelihood amplitude estimation* and *faster amplitude estimation* [10, 18].

Chapter 3

Classical Machine Learning

Since the beginning of the information age, there has been great speculation on how good computers can imitate the human behaviour. This can be observed through the evolution in the field of **Artificial Intelligence** (AI). An AI system is any system that perceives its environment and takes actions that maximise its chance of achieving its goals, meaning that implementing and testing it can lead to great results. **Machine Learning** (ML) is a subfield of artificial intelligence. It is a category of algorithms that allow software applications to become more accurate in predicting outcomes without being explicitly programmed. The basic goal is to build algorithms that can receive input data and use statistical analysis to predict an output while updating outputs as new data becomes available. Its uses range from face and voice recognition, to traffic and weather prediction, to fraud detection and stock prediction in the field of Finance. In the following chapter, we will introduce some of the basic methods of machine learning, along with a basic machine learning model for the current thesis, the Generative Adversarial Network (GAN), which is based on the neural network technology.

3.1 Machine Learning Methods

Machine learning can be classified into 4 major types of algorithms, supervised learning, semi-supervised learning, unsupervised learning and reinforcement learning [16].

Supervised Learning

It is a training method, where a computer algorithm is trained on input data that has been labeled for a particular output. The goal is that, for a given input X, the model we will be able to predict the output Y. During its training the model receives labeled data sets that instruct the system on what output is related to a specific input. Then, the model is being given a test data set, with hidden labels, and is instructed to predict the output based on its knowledge from the training. The most prominent types of supervised learning are: **classification** and **regression**.

- **Classification**: The task of this algorithm is to map the input value(x) with the discrete output variable(y).

- **Regression**: The task of this algorithm is to map the input value (x) with the continuous output variable(y).

Semi-Supervised Learning

It refers to a learning problem that involves a small portion of labeled examples and a large number of unlabeled examples from which a model must learn and make predictions on new examples.

Unsupervised Learning

It is a machine learning technique in which the user does not need to supervise the model. Instead, it allows the model to work on its own to discover patterns and information that was previously undetected. Unlike supervised learning, this method deals with unlabelled data. The most known types of unsupervised learning are: **clustering** and **association**.

- **Clustering**: It is the task of creating groups of objects in such a way that objects in the same group (called a cluster) are more similar and have similar properties to each other than to those in other groups.
- **Association**: It is based on the idea of discovering rules that define large portions of data samples.

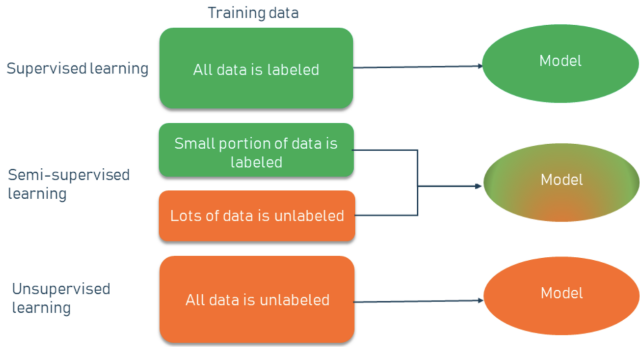
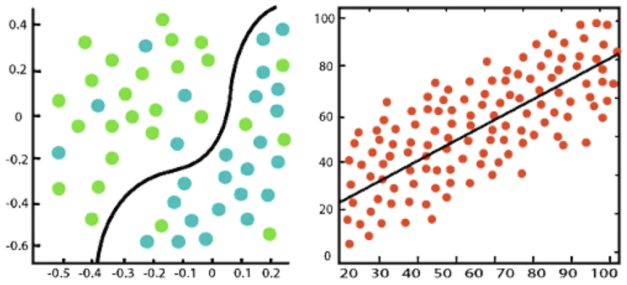
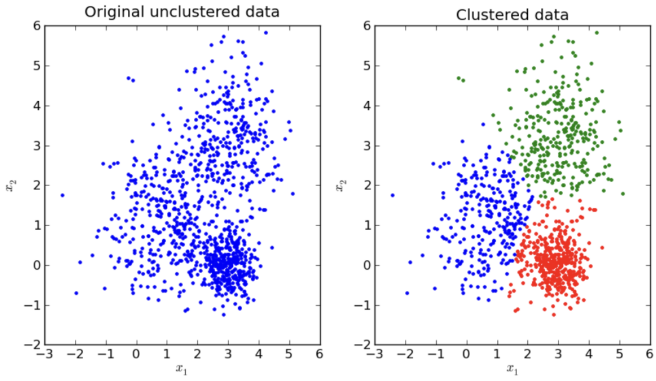


Figure 3.1: Differences between supervised, unsupervised and semi-supervised learning.



(a)



(b)

Figure 3.2: Figure (a) shows an example of classification (left) and regression (right) and figure (b) an example of clustering.

Reinforcement Learning

It is a machine learning training method based on rewarding desired behaviors and/or punishing undesired ones, using penalties. It is an agent based technique, where the agent learns how to achieve a goal in an uncertain, potentially complex environment using trial and error to come up with a solution to the problem. Then he either gets rewards or penalties for the actions it performs. Its goal is to maximize the total reward, or minimize the total penalty.

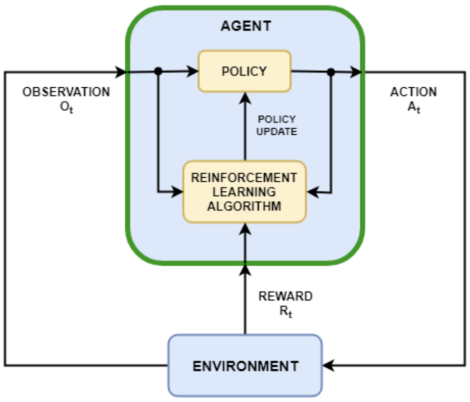


Figure 3.3: Agent reinforcement learning model.

3.2 Neural Networks

3.2.1 Introduction to Neural Networks

Inspired by the human brain and the way biological neurons signal one another, they are a subset of machine learning in the center of deep learning algorithms. They consist of three layers the input, one or more hidden layers and an output layer. Each one contains nodes, which are connected to each other using edges. Each edge has an associated threshold and weight. The way the model works is that if the output of a node is greater than the threshold, then this node is activated and sends the data to the next layer, otherwise nothing happens. Neural Networks (NN) through fine tuning and data learning, are able to produce results in regression analysis, classification and data processing, in significantly lower time than normal human based methods. Some examples of NN applications are, among others, time-series production, stock market prediction, facial recognition, security and defence applications and ultrasound and X-ray detection in healthcare systems.

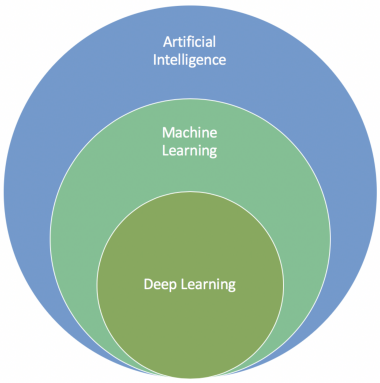


Figure 3.4: Hierarchy in AI.

3.2.2 Training the Neural Network Models

Each node can be considered as a linear regression model, composed of input data, weights, a threshold (*bias*) and an output:

$$\sum_{i=1}^{m} w_i x_i + bias = w_1 x_1 + ... + w_m x_m + bias \tag{3.1}$$

Weights determine the importance of any given variable, with longer ones contributing more significantly to the output compared to other inputs. Inputs are multiplied with their weight and then summed up. After that, the output is passed through an *activation function*. An activation function in a neural network defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network. It is used within or after the internal processing of each node in the network, although networks are designed to use the same activation function for all nodes in a layer. The choice of activation function will control how well the network model learns the training data set and will define the type of predictions it can make.

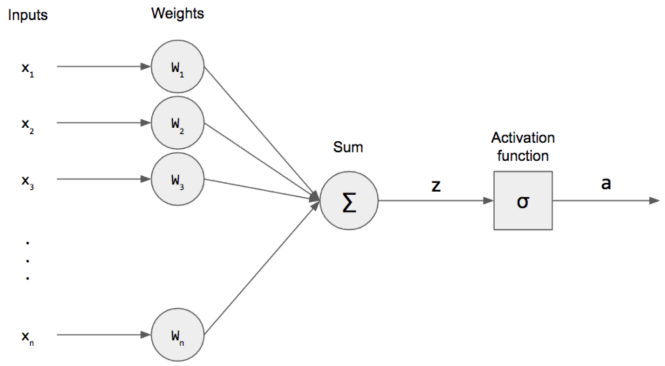


Figure 3.5: Workflow inside a perceptron.

As the training proceeds, we will want to evaluate its accuracy using a loss function, also referred as **Mean Squared Error** (MSE), in this case:

$$MSE = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y} - y)^2 \tag{3.2}$$

where: $\hat{y}$: predicted outcome, y: actual outcome, m: number of samples.

Naturally the goal is to minimise the cost function. The model uses the cost function and reinforcement learning to reach the point of convergence, or the local minimum. The method during which the algorithm adjusts its weights is gradient descent , allowing it to determine the direction needed to take to reduce the error rate, meaning to minimise the cost function.

Gradient Descent

We use gradient descent to find a local minimum of a differentiable function as in figure 3.6. It is based on the idea of taking repeated steps in the opposite direction of the gradient of the

function. The size of these steps is called learning rate. The higher the learning rate is the faster we can reach the lowest point. This is a bit ambiguous, since with low learning rate we have more precise results but we spend more time, while with high learning rate we spend less time, but we achieve the result with less precision. Given the cost function:

$$f(w,b) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - (w_i x_i + b))^2, \tag{3.3}$$

where $y_i = w_i x_i + b$, w the weight of sample i and b the bias, the gradient can be calculated as:

$$f'(w,b) = \begin{bmatrix} \frac{df}{dw} \\ \frac{df}{db} \end{bmatrix} = \begin{bmatrix} \frac{1}{N}\sum_{i=1}^{N} -2x_i(\hat{y}_i - (w_i x_i + b)) \\ \frac{1}{N}\sum_{i=1}^{N} -2(\hat{y}_i - (w_i x_i + b)) \end{bmatrix} \tag{3.4}$$

To solve for the gradient, we iterate through our data points using our new w, b values and compute the partial derivatives. This new gradient tells us the slope of our cost function of our current position and the direction we should move to update the parameters.

Most networks are *feedforward*, meaning that they flow from input to output. However they can also be trained through *backpropagation*. It is an algorithm for supervised learning of artificial neural networks using gradient descent. The method calculates the gradient of the error function with respect to the neural network's weights. The difference here is that the gradient of the final layer of weights is being calculated first and the gradient of the first layer of weights is being calculated last, meaning that we start from the output layer and go through the NN to the input layers.

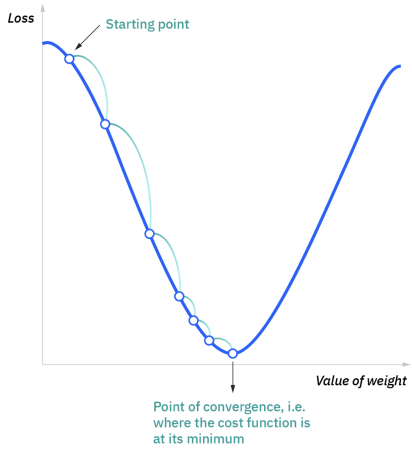


Figure 3.6: Minimizing the Cost Function

3.2.3 Types of Neural Networks

There are many variations of the neural network model. Some differ on the structure and others on the functions they implement.

Perceptron

The perceptron is the oldest neural network, with its discovery dating back to 1958. It consists of a single neuron with input and output layers only and is the simplest form of a neural network. In figure 3.7a, the **yellow** nodes indicate the input layers and the **red** indicate the output layer.

Multilayer Perceptrons (MLP)

The model we have been studying so far in this thesis consists, as mentioned, of an input layer, one or more hidden layers and an output layer. Data is fed into them to train them and are used in computer vision and language processing applications among others.
In figure 3.7b, the **green** nodes indicate the hidden layers.

Convolutional Neural Netwroks (CNN)

They are very similar to MLPs but include many linear algebra properties making them useful for image and pattern recognition.
In figure 3.7c, the **purple** nodes indicate the kernels while the **purple circles** indicate the process of convolution.

Recurrent Neural Networks (RNN)

They are distinguishable from the other types due to their feedback loops. They are used when decisions from past iterations or samples can influence current ones. They are met when dealing with time series. They can be used to make predictions for future outcomes, such as stock market predictions and sales forecasting.
In figure 3.7d, the **blue** nodes indicate a recurrent cell.

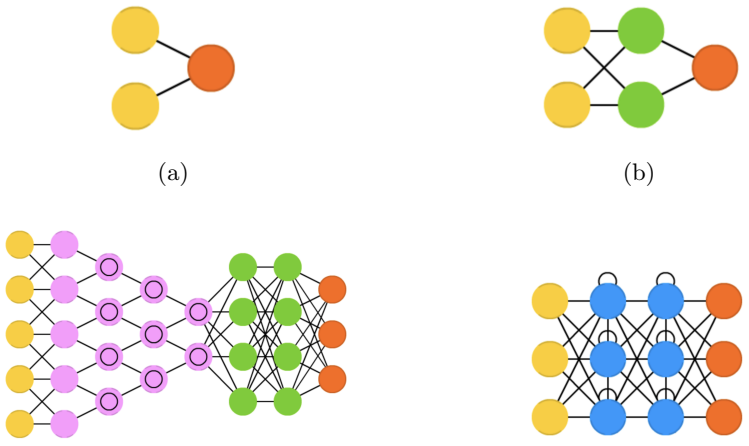


Figure 3.7: Neural network variations. Figure (a) shows a perceptron, (b) a multilayer perceptron, (c). a convolutional neural network and (d) a recurrent neural network.

3.3 Classical Generative Adversarial Networks

Generative Adversarial Networks (GANs) belong to the family of generative models in machine learning [8]. Generative models are processes that can generate new data instances. Given an observable variable X and a target Y, a generative model is a statistical model of the joint probability distribution $P(X|Y)$. The basic goal of GANs is to discover patterns in the input data and learn them in a way that the model can then generate new samples that retain characteristics of the original data set. A GAN is a model that is composed of two neural networks, **the Generator (G)** which captures the data distribution from the input data and generates new data and **the Discriminator (D)** which estimates the probability that a sample came from the training data and not from G. The generator is responsible for the generation of data, trying to deceive the discriminator, who has the task of identifying if the data have originated from the training data set or from the generated data set. The structure of a GAN model is shown in Fig.3.8.

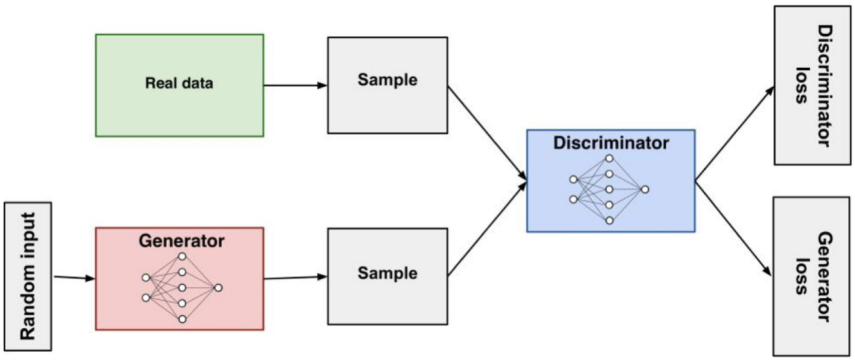


Figure 3.8: Classical Generative Adversarial Network

The discriminator acts as a classifier where it receives real and synthetic data (i.e. information that's artificially manufactured rather than generated by real world events) from the generator and attempts to identify their origin. On its output, it connects two loss functions the **Discriminator Loss** and the **Generator Loss**. After the classification, the discriminator loss, penalises it for the misclassifications and updates its weights through backpropagation.

The generator uses feedback from the discriminator to learn how to generate synthetic data that are similar, if not identical, to those of the real data set. Its goal is that this data will not be able to be distinguished from the real one. It receives a random input and the result is evaluated by the discriminator, where the generator loss function is computed. The penalty that the generator receives through this function is because the data it produced were not good enough to deceive the discriminator.

3.3.1 Training the GANs

To quantify the performances of the generator and the discriminator, a loss function that measures the distance between the distributions of the generated and the real data sets is defined

as a *minimax loss function*:

$$min_G max_D V(D,G) = E_x[logD(x)] + E_z[log(1 - D(G(z)))],$$

where:

- $D(x)$ is the discriminator's estimate of the probability that real data instance x is real
- E_x is the expected value over all real data instances.
- $G(z)$ is the generator's output when given input z.
- $D(G(z))$ is the discriminator's estimate of the probability that a fake instance is real.
- E_z is the expected value over all generated (fake) instances $G(z)$.

and X is the real data set while Z is the generator's input. Basically the generator tries to minimise the function above, while the discriminator tries to maximise it. For the generator minimising the loss is equivalent to minimising the $log(1-D(G(z)))$ term, since it can't directly affect the $log(D(x))$ term.

This loss function is then optimized by a classical optimizer, which will update the model in response to the output of the loss function. It has control over the learning process, since its target is to find the values that lead to the lowest loss function. Most models use a gradient descent based optimiser. To simplify the notation, from this point, generator will also be referred as G, and the discriminator as D.

Now we will outline the training procedure of the GANs. Firstly, we sample from the random input 'Z' samples and so new samples are produced in G. Then D classifies the samples as "real" or "fake" and a loss function is calculated from the classification. Afterwards, a backpropagation occurs through D and G to obtain gradients and those are then used to change the weights of generator's neural network. Following, the same process occurs again. The discriminator gets from its input the real and the fake data and classifies them, whether it thinks they are real or generated. Then the discriminator loss penalises D for misclassifying real and fake instances. Finally as in the generator, the discriminator's weights are updated through backpropagation from the discriminator loss inside the D network.

Since training two neural networks at the same time can be very challenging, GAN training proceeds in alternating periods. The discriminator and the generator are trained for one or more epochs repeatedly. An epoch is defined as one complete pass of the training data set. While the discriminator is trained, the generator is kept at a halt, as the first has to figure out how to distinguish the two types of data and how to recognise the latter's flaws. Similarly the discriminator is kept constant while the generator goes through its training phase, otherwise the generator would always try to reach a constantly moving target, making it impossible to converge.

3.3.2 Challenges and variations of GANs

There are two main challenges, one has to face when working with GANs. The first one is **vanishing gradients** and the second one is **mode collapse**. The *vanishing gradient* occurs when the discriminator gets too good, so that the training of the generator can fail. The gradient gets so small that it doesn't change the weight values of the generator's initial layers, leading to a very slow learning rate and possibly a halt. This means that an optimal discriminator is not useful since

it doesn't provide enough feedback for the generator. The *mode collapse* occurs when the generator maps several input values to the same output, or when the generator ignores a region of the target data distribution. This means that the goal of the training process of the generator is not only to generate realistic samples, but also to produce a wide variety of them.

To solve these challenges there have been proposed two different variation types of GANs. The first one is **architecture variants** and the second one is **loss variants**. In *architecture variants*, structural changes are made to adapt the GAN to a certain purpose or to improve the overall performance. On the other hand in *loss variants*, different approaches to loss functions occur, which try to improve stability and performance while training, usually aiming to solve the issue of non-convergence. Namely some variants are: *FIN-GANs* (2019), *Conditional-GANs* (2019), *WGAN-GP* (2019), *Corr-GAN* (2019), *Quant-GAN* (2020), *MAS-GAN* (2021) [8].

Chapter 4

Quantum Machine Learning

So far we have seen the most prominent classical machine learning methods, namely supervised, semi-supervised, unsupervised and reinforcement learning methods. We have also studied about neural networks, along with a model that uses them for its implementation, the generative adversarial network. We have also described the basics of quantum computing, and some of the basic quantum subroutines for allowing possible speed ups. In this chapter, we will combine the above and we will introduce the basic concepts and ideas behind Quantum Machine Learning (QML). We will see why it is such a promising concept and present some quantum machine learning algorithms. We will end the chapter by defining the quantum counterpart of GANs, which is the quantum Generative Adversarial Network (qGAN).

4.1 Introduction to Quantum Machine Learning

Quantum machine learning can be defined as the field where classical machine learning meets the quantum computing properties and algorithms, in order to push the boundaries and the computational capabilities on a way to technological advancement. It aims to take advantage of the computational power that quantum computers provide to solve machine learning problems much more efficiently. Some of the key benefits of QML are among others the speedup in the training of the model, the ability to handle complex network topology and the computational power to perform complex matrix and tensor manipulation at high speeds.

The current state of the art of quantum machine learning can be naively categorized into two approaches. The first one is the theoretical study of QML algorithms for **fault-tolerant computers**. This holds the premise of providing exponential speedup for certain problems. The idea is that we don't change the structure of the ML algorithm, rather than we use quantum computers to speedup its subroutine. In table 4.1 we present some of the most important quantum machine learning algorithms that have been proven to provide quantum speedup. The label (*) labels the algorithms that have to take into account certain caveats to achieve this speedup and N is the number of samples. $O(\sqrt{N})$ corresponds to quadratic speedup, while $O(\log N)$ to exponential speedup, both compared to their classical counterparts. For more information on the following methods, we cite you to their respective papers.

Method	Speedup
Bayesian Inference [15]	$O(\sqrt{N})$
Online Perceptron [12]	$O(\sqrt{N})$
Least Squares Fitting [27]	$O(\log N^{(*)})$
Quantum Boltzmann Machines [1]	$O(\log N^{(*)})$
Quantum PCA [13]	$O(\log N^{(*)})$
Quantum SVM [24]	$O(\log N^{(*)})$
Quantum Reinforcement Learning [7]	$O(\sqrt{N})$

Table 4.1: Quantum speedup provided by quantum machine learning methods.

During the previous years there have been studies to find fault-tolerant algorithms that provide these speedups and also to be able to implement them on machine learning problems as well. The most important discovery is probably the **HHL** algorithm, named after scientists Aram Harrow, Avinatan Hassidim and Seth Lloyd [11]. It is considered to be the basis of many quantum machine learning algorithms, where it is used either as a subroutine or as an extension. What HHL does, is that it tackles the basic problem of solving linear equations in potentially exponential time when compared to the classical ways. We will discuss about this algorithm and its importance later in this chapter.

The problem is that the aforementioned machines are far from being realized, and what we have now are the noisy (not error corrected) quantum devices that contain about a few hundred qubits. These noisy-intermediate scale quantum (NISQ) devices impose constraints on the depths of quantum algorithms, inspiring the development of hardware-aware quantum algorithms that utilise classical resources to unload some computational weights off the quantum devices, i.e: *variational models* for quantum chemistry problem in 2014 and has since adapted to QML problems in 2018. They are based on the idea of using a classical computer to train the parameters that are afterwards input to a quantum circuit. There are four different approaches to combine the disciplines of quantum computing and machine learning and are presented in Figure 4.1. We can have variations on the type of data that we use and on the type of algorithm we implement, i.e. we can use classical data or quantum states to analyse, and of course we can choose between classical or quantum algorithms. The most common case is to perform analysis on classical data while using quantum computers.

4.1.1 Quantum Machine Learning Algorithms

In this section we will study some of the quantum machine learning algorithms that, as we mentioned, have the potential to achieve very good results exponentially faster. These belong and have been evolved around the fault-tolerant approach. We start with an extensive study on possibly the, currently, most important algorithm (HHL), which is not a QML algorithm by its nature but can be the basis on QML algorithms to achieve speedup and then we present the theoretical quantum analog of two very important classical machine learning methods, namely principle component analysis and support vector machines.

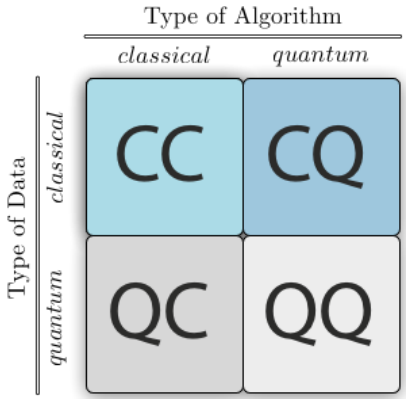


Figure 4.1: Different QML approaches

Quantum Machine Learning Algorithm to Solve Linear Algebraic Problems

As we have already mentioned, solving linear algebraic problems is not a machine learning problem. While this problem may sound simple, it can get really complex very fast. However it can be be the basis or a very helpful tool on our way of solving bigger more complex ML problems. The HHL algorithm is the key algorithm in this study where it basically attacks the problem of solving a system of linear equations. The problem description is as following: Given an $n \times n$ real matrix A and a vector b, the goal of HHL is to solve the system: $Ax = b$ for x in time that scales logarithmically with n , where n is the number of equations and unknowns. The current classical approach requires n^2 steps to examine all entries of A and n more steps to write the solution vector x. HHL promises to do this process in $log(n)$ steps. But this can occur only if some caveats are taken into account. Firstly, we need to be able to encode vector $b = [b_1, ..., b_n]^T$ in the quantum computer's memory and encode b's entries as $|b\rangle = \sum_{i=1}^{n} b_i |i\rangle$. This could be done by using a *quantum RAM (qRAM)*, which is a memory that stores the classical values of b and then allows them to be read at once in quantum superposition. However, we have to note that currently there doesn't exist any physical qRAM, rather than a more theoretical approach to it. Otherwise, if we are lucky enough and b is described by a simple formula, the quantum computer could possibly be able to prepare $|b\rangle$ quickly by itself. In general, we can say that if preparing $|b\rangle$ takes more than n^c, with c some constant, then we lose the promised exponential speedup.

Furthermore, the quantum computer must be able to apply unitary transformations of the form e^{-iAt} for various values of t. Should we encounter the case where matrix A is sparse, meaning it has at most s non-zero entries per row, for $s << n$, and we have a qRAM that can store all the non-zero values and locations for each row, it is proven that we can apply the unitary transformation mentioned, in time that grows linearly with s [2]. Again, if this process requires n^c time then the speedup disappears.

Finally, it is obviously important to be able to export the results. After the quantum computations of the algorithm, solution vector x is at a quantum state $|x\rangle$ of $log(n)$ qubits, which approximately encodes the entries of x in its amplitudes. We can measure $|x\rangle$ in the basis of our choice to reveal some limited statistical information about x like the location of any very large entry of x, but should we like to learn the value of a specific entry x_i, it would require n algorithm

repetitions, leading inevitably to the killing of the speedup.

Summing up, the best way to see the HHL algorithm is as a template for other quantum algorithms. It tackles the problems of preparing the states, applying unitaries and measuring the output and after that one has to analyze the total cost to see if it is faster than the respective classical process. In the years since the publication of the HHL algorithm there have been proposed QML algorithms that achieve exponential speedups over classical ML algorithms. However, it is important to always have in mind that if we want to use the HHL algorithm for QML problems and to achieve that speedup, we must track that the caveats that we mentioned are met.

Quantum Principal Component Analysis

Classical principal component analysis (PCA) is a technique that reduces the dimensionality of large data sets. It is a process during which, we have to decide which variables we should keep and which we can eliminate without losing important information. Through this we can make the machine learning task easier and faster since we will be dealing with a smaller data set. This dimensionality reduction makes use of the eigenvectors and the eigenvalues. The problem classical PCA faces, is that when dealing with high dimensions it is natural that the set of eigenvalues and eigenvectors will be very big. From the classical PCA, we compute the covariance matrix of the data of a vector v_j as: $C = \sum_j v_j v_j^T$. The classical analysis operates by diagonalizing the covariance matrix: $C = \sum_k e_k c_k c_k^\dagger$, where c_k the eigenvectors and e_k the corresponding eigenvalues. The eigenvectors whose eigenvalues are large, are called principle components.

For quantum PCA of classical data, we firstly choose a data vector v_j and use a quantum random access memory (qRAM) to map that vector into a quantum state: $v_j \rightarrow |v_j\rangle$. The quantum state that summarizes the vector has $log(n)$ qubits and the operation of the qRAM requires $O(n)$ operations divided by $O(log(n))$ steps that can be performed in parallel, where n is the dimension of the vector space. The resulting quantum state has a density matrix $\rho = (1/N)\sum_j |v_j\rangle \langle v_j|$, where N is the number of data vectors. So we can actually see that the density matrix is the covariance matrix C, up to a factor. By repeatedly sampling the data and using density matrix exponentiation combined with the quantum phase estimation algorithm (which finds eigenvectors and eigenvalues), we can take the quantum version of any data vector $|v\rangle$ and decompose it into the principle components $|c_k\rangle$ as: $|v\rangle = \sum_k v_k |c_k\rangle |\hat{e_k}\rangle$. Finally, the principle components of C can be probed by making measurements on the quantum representation of the eigenvectors of C. This algorithms has complexity of $O(log(N)^2)$ which is exponentially more efficient than classical PCA ($O(N)$) [13].

Quantum Support Vector Machines

Classical Support Vector Machines (SVM) aim to find an optimal separating hyperplane between two classes of data in a data set, such that with high probability all training examples of one class are only found on one side of the hyperplane. The most robust classifier is obviously when the margin between the hyperplane and the data are maximized. The problem is that classical SVM can be performed only up to a certain number of dimensions. After a particular limit, it will be hard because classical computers do not have enough processing power.

On the other hand quantum SVM has the potential to be much faster. The data input can come from various sources, like a qRAM accessing classical data or a quantum subroutine that

produces quantum states. Once they are ready, they are processed with quantum phase estimation and matrix inversion (HHL algorithm). The total operational time, to construct the hyperplane and to test whether a vector lies on one side or the other, requires $O(logN)$ time, where N is the dimension of the matrix required to prepare a quantum version of the hyperplane vector [24].

4.1.2 Data Encoding Techniques and Variational Models

Following, we will see how the noisy devices work, in order to solve ML problems in the NISQ era. As we have seen one of the most prominent challenges we have to deal when studying QML is the data encoding process. This applies of course to the case when we are using classical data and quantum algorithms (CQ) and not when we have from the beginning only quantum states. While this can become a prohibitive reason to work with quantum algorithms, we will present three data encoding techniques along with the idea behind the variational models.

Data Encoding

While there isn't any systematic global methodology for this, there are three main ways of data encoding. **Basis encoding** is primarily used when real numbers have to be arithmetically manipulated in a quantum algorithm. Such an encoding represents real numbers as binary numbers and then transforms them into a quantum state on a computational basis. For example, for 2 qubits, these states would be: $|00\rangle, |01\rangle, |10\rangle, |11\rangle$ and considering that we have as real data values $x_1 = 2, x_2 = 3$, we map $|00\rangle \to 0, |01\rangle \to 0, |10\rangle \to x_1, |11\rangle \to x_2$. This is not efficient in terms of the required number of qubits but is good for arithmetic operations as we mentioned. **Amplitude encoding** is a method where we encode our data into amplitude vectors rather than basis states. A normalized N-dimensional data point x is represented by the amplitudes of an n-qubit quantum state $|\psi\rangle = \sum_{i=0}^{N} x_i |i\rangle$, where x_i is the i-th element of x and $|i\rangle$ is the i-th computational basis state. Finally, for **angle encoding** we need a number of qubits equal to the dimensionality of our data. From there, we encode our data values as rotation angles in rotation gates. Therefore, each qubit is rotated about a given axis (x,y, or z) by an angle that is dependent on the classical data value.

Variational Model

As we have mentioned, at present we don't have the hardware to build large scale, fault tolerant quantum computers that are able to execute all the quantum algorithms. The current solution to this are the variational models. The way they work is fairly simple. It is a hybrid process. Firstly we have a quantum circuit, where the classical data are mapped in to the input state which are then passed though a parameterized unitary $V(\theta)$. Afterwards, a measurement is made on the output state and we obtain an expectation value. Then, this value, classicaly, is evaluated through a cost function and optimized using optimization methods like gradient descent and after that the parameters are updated accordingly and given as an input to $V(\theta)$ to go through the process again.

One of the most common problem one faces when dealing with variational models is the loading of classical data to quantum states, which potentially kills the speedup. Furthermore, another process that may take a lot of time and resources is the training of the circuit. Therefore, while they are possibly the best hardware technology we can use for QML processing, we must note that they are not panacea.

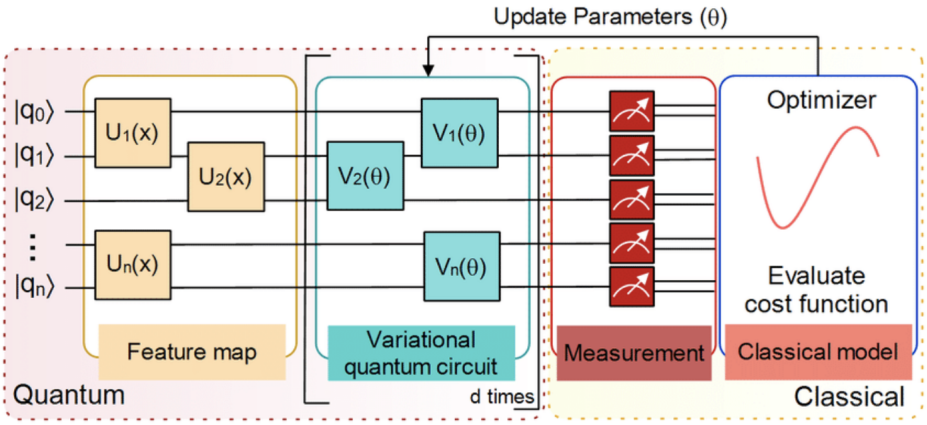


Figure 4.2: Variational model

So far we have seen how QML works and what it can potentially offer. We have presented some very powerful algorithms and we have showcased how real data can be encoded to variational circuits along with the workflow of the latter. In the remaining of this chapter we will study the quantum analog of the generative adversarial networks, which we introduced in *chapter 3.3*, for learning and loading the probability distribution to quantum states using variational model in the generator network.

4.2 Quantum Generative Adversarial Networks

As we have discussed in the **Chapter 3**, generative adversarial networks are used to discover patterns in the input data and try to recreate data sets with same characteristics. In order to do so, it consists of a "fake data" *Generator* and a *Discriminator*, which decides if the data that are input to it, are real or generated [30]. When working with quantum generative adversarial networks there are different implementations of the generator and the discriminator. Normally, in the classical model they would be neural networks but in our case we have models where they consist of either a *quantum generator* and a *quantum discriminator*, a *quantum generator* and a *classical discriminator* and a *classical generator* and a *quantum discriminator*. Furthermore, we can load to the model, either classical or quantum data. In the case that we are going to study, we choose a *quantum generator*, which is a quantum circuit and a *classical discriminator*, which is a classical neural network acting as classifier. Our goal is to train the quantum generator to create a quantum state, which represents the classical training sample's underlying probability distribution [6, 14]. The qGAN we will study can be seen in figure 4.3.

4.2.1 The Quantum Generator

Generally, we can consider the **quantum generator** as a black box, which we define as $U(\theta)$. As an input to this box we can consider n-qubits in state $|0\rangle$, which can be written as: $|0\rangle^{\otimes n}$. Its output is the application of this box to the input state, written as: $|\psi(\theta)\rangle = U(\theta)\,|0\rangle^{\otimes n}$.

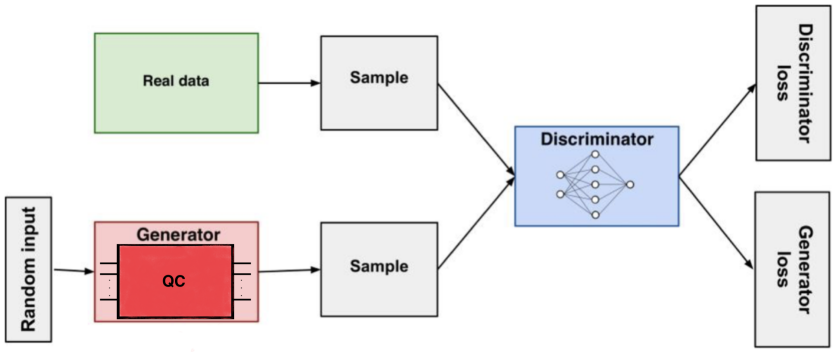


Figure 4.3: Quantum GAN with quantum generator and classical discriminator

Implementing Born's rule to compute the probability of measuring state i, we have:

$$p(i) = |\langle i|\psi(\theta)\rangle|^2$$

Using this probability distribution we can obtain some samples, while for the quantum generator we train $U(\theta)$ until the circuit can reproduce the real data probability distribution. The way we load the random data into the quantum circuit is through amplitude encoding.

The quantum generator in its general form is a parameterised quantum circuit, trained to transform an n-qubit input state $|\psi_{in}\rangle$ to an n-qubit output state: $|g_\theta\rangle = \sqrt{p_\theta{}^j}\,|j\rangle$, where p_θ^j describe the probability of the occurrence of state $|j\rangle$. We will show how this circuit is implemented. It consists of alternating layers of single qubit rotations around the Y-axis (R_Y) and blocks of controlled-Z gates (**CZ**), called entangled blocks, denoted as U_{ent}. We define a parameter k, called **depth of the circuit**, to define how many repetitions of R_Y and U_{ent} we will use. The rotation on the i-th qubit on the j-th layer is parameterised by $\theta^{i,j}$. It is obvious that increasing the depth k leads to more complex, but more accurate, structures.

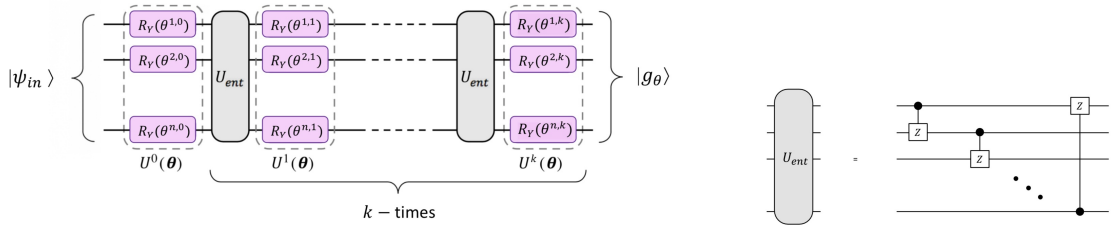
The reason we use Y-rotations, instead of X or Z, is that in contrast to the latter, for $\theta^{i,j} = 0$, the circuit does not have any effect on the amplitude of the state, but only flips the phase. These flips do not affect the probability distribution, which depends on the amplitude, meaning that if the state is loaded efficiently, the circuit allows its exploitation. Each entangling block applies **CZ** gates from qubit i to qubit $((i+1) \mod (n))$ to create entanglement between the different qubits.

- We remind that R_Y and CZ gates are as following:

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

$$\mathbf{CZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}.$$

Below we present the quantum circuit of the quantum generator, and the U_{ent} blocks.



(a) (b)

Figure 4.4: Figure (a) shows the variational circuit of the quantum generator and figure (b) takes us inside the U_{ent} block.

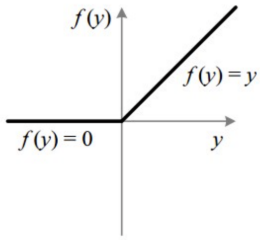
4.2.2 The Classical Discriminator

The discriminator is a classical neural network, consisting of several layers that apply non-linear activation functions. It processes the data samples and labels them as real or generated. It consists of a 50-node input layer, a 20-node hidden layer and a single node output layer. As with all neural network models, the nodes must have an activation function. As the name suggests an activation function decides whether a node should be activated or not. The role of the activation function is to derive output from a set of input values fed to that node. The input and the hidden layers apply linear transformations followed by Leaky ReLu functions, while the output layer implements another linear transformation and applies sigmoid function, to be able to define the distinguishing between the real or fake result. For the input and hidden layers, we use Leaky ReLu activation function and not a simple ReLu function and the reason is that the latter is faulty. As you can see from the figure 4.5a, all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. This means that any negative input given to the ReLU activation function turns into zero immediately, which in turns affects the resulting graph by not mapping the negative values appropriately. On the contrary with a Leaky ReLu activation function, we can increase the range of the ReLU function. We use sigmoid function for the output layer because it exists between (0.0 to 1.0). Since the discriminator acts as a classifier, which outputs "real" or "fake", the sigmoid activation function is very good since, using a confidence *'c'* the sample could correspond to *"real"* and for *'1-c'* to *"fake"*.

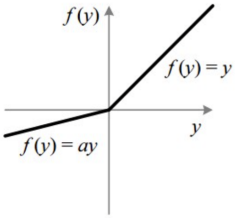
4.2.3 The Training

Given m data samples g^l from the quantum generator and m randomly chosen real training data samples x^l, where $l : 1, ..., m$ we can compute the **loss functions** for the generator and the discriminator. Assuming ϕ is the parameter of the discriminator and θ the parameter of the generator we have:

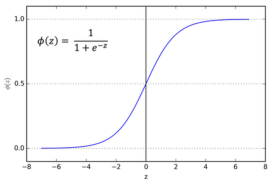
$$L_G(\phi, \theta) = -\frac{1}{m}\sum_{l=1}^{m}[log(D_\phi(g^l))] \tag{4.1}$$



(a) ReLu activation function.



(b) Leaky ReLu activation function.



(c) Sigmoid activation function.

Figure 4.5: Activation functions.

$$L_D(\phi, \theta) = -\frac{1}{m}\sum_{l=1}^{m}[log(D_\phi(x^l)) + log(1 - D_\theta(g^l))] \tag{4.2}$$

From these functions we can see that the loss function of the generator depends on the sum of the generated data samples, the discriminator has predicted correctly that are fake, while the loss function of the discriminator depends on the sum of the real data samples that it has predicted correctly, plus the false classifications of the generated data samples as real.

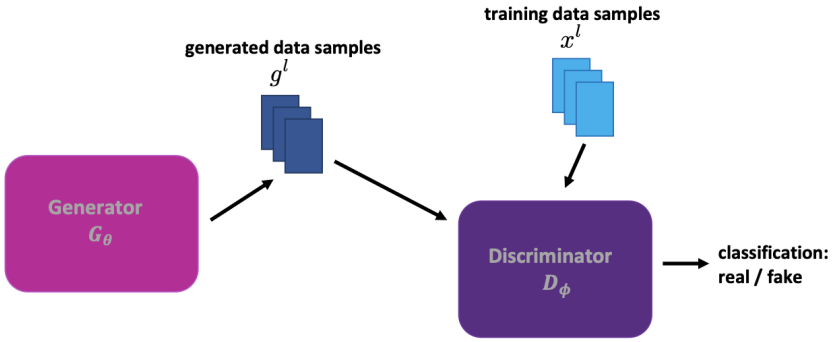


Figure 4.6: Quantum GAN workflow. The quantum generator produces samples and the classical discriminator classifies them, along with real training data as real or fake.

To train the qGAN, samples are drawn from the generator, by measuring the output state $|g_\theta\rangle$. The measurements are classical information, namely p_j, which is defined as the measurement frequency (probability) of $|j\rangle$. A carefully chosen input state $|\psi_{in}\rangle$ can help reduce the complexity of the quantum generator and the number of training epochs. We should note here, that the

preparation of the generator's input should not dominate the overall complexity. This means that the input should be loadable with at most $O(poly(n))$ gates.

Quantum GANs are trained using **AMSGRAD**. AMSGRAD is an extension to the **Adam** version of gradient descent that attempts to improve the convergence properties of the algorithm, avoiding large abrupt changes in the learning rate for each input variable. It is a robust optimization technique for non-stationary objective functions as well as for noisy gradients, making it very suitable for running the algorithm on real quantum hardware, where the noise presence is extremely high.

Optimization Method - Gradient Descent

We want to calculate the generator's loss function gradients. Firstly, applying the n-qubit generator to input state $|\psi_{in}\rangle$ produces the state:

$$|g_\theta\rangle = G_\theta |\psi_{in}\rangle = \prod_{p=1}^{k} (\otimes_{q=1}^{n} (R_Y(\theta^{q,p})) U_{ent}) \otimes_{q=1}^{n} (R_Y(\theta^{q,0})) |\psi_{in}\rangle$$

$$\implies |g_\theta\rangle = \sum_{j=0}^{2^n-1} \sqrt{p_\theta^j} |j\rangle \tag{4.3}$$

We measure $|g_\theta\rangle$ m times to obtain data samples g^l, where $l \in \{1, ..., m\}$ which can take 2^n. different values.

The generator's loss function is:

$$L_G(\phi, \theta) = -\frac{1}{m} \sum_{l=1}^{m} [log(D_\phi(g^l))] \implies L_G(\phi, \theta) = -\sum_{j=0}^{2^n-1} p_\theta^j [log(D_\phi(g^l))] \tag{4.4}$$

with $p_j^\theta = |\langle j|g_\theta\rangle|^2$.
Since θ is the generator's parameter, in order to realise a gradient descent on L_G, we differentiate L_G with respect to θ. This gives us:

$$\frac{\partial L_G(\phi, \theta)}{\partial \theta^{i,l}} = -\sum_{j=1}^{m} \frac{\partial p_\theta^j}{\partial \theta^{i,l}} log(D_\phi(g^l)) \tag{4.5}$$

According to [31], the term $\frac{\partial p_\theta^j}{\partial \theta^{i,l}}$ can be evaluated as:

$$\frac{\partial p_\theta^j}{\partial \theta^{i,l}} = \frac{1}{2}(p^j_{\theta^{i,l}_+} - p^j_{\theta^{i,l}_-}) \tag{4.6}$$

where we have: $\theta_\pm^{i,l} = \theta^{i,l} \pm \frac{\pi}{2} e_{i,l}$, with $e_{i,l}$ denoting the (i,l)-unit vector of the respective parameter space.

4.2.4 Evaluation Statistics

The Kolmogorov - Smirnov Statistic

The **Kolmogorov - Smirnov** statistic is defined by the distance of two probability distributions P and Q. It is based on the cumulative distribution function $P(X \leq x)$ and $Q(X \leq x)$ and is given by:

$$D_{KS}(P||Q) = \sup_{x \in X} |P(X \leq x) - Q(X \leq x)|. \tag{4.7}$$

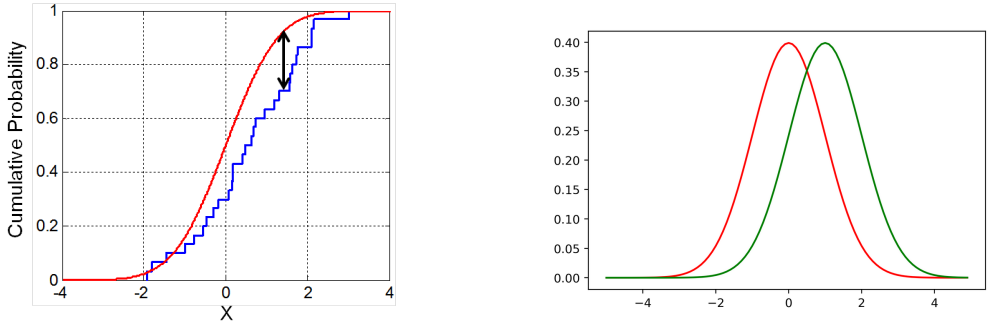
In our case, the null hypothesis we want to check is if the probability distribution from $|g_\theta\rangle$ (denoted as P) is equivalent to the probability distribution of the real data (denoted as Q), with a confidence level set at 95%.

The Relative Entropy Statistic

Another measure used to characterise the closeness of the discrete probability distributions $P(x), Q(x)$ is the **relative entropy**, also known as **Kullback - Leibler divergence**. It is given by:

$$D_{(RE)}(P||Q) = \sum_{x \in X} P(x) log(\frac{P(x)}{Q(x)}). \tag{4.8}$$

This metric represents a non negative quantity, meaning that $D_{(RE)}(P||Q) \geq 0$, with the equality true if $P(x) = Q(x)$. This means that we want to get as close to 0 as possible, which would mean that the generated data and the real data distributions would match.



(a) (b)

Figure 4.7: Figure (a) offers a Kolmogorov - Smirnov representation indicating the distance between two distributions. Figure (b) offers relative entropy graphical explanation as we see the PDF of two distributions. The closer the distributions get, the closer to zero the relative entropy gets.

4.2.5 Testing Setup

Next we are going to present the results of qGANs training with different generator input distributions and different target distributions. The quantum circuit of the generator is implemented using the QiSkit simulator. QiSkit is an open-source software development kit (SDK) suitable for implementing and working with quantum circuits and algorithms. These programs are run on

prototype quantum devices on IBM Quantum Experience or on simulators on a local computer [23].

We will use 20.000 real samples with 3 different real data distributions, namely a log-normal, a triangular and a bimodal distribution (consists of two superimposed Gaussian distributions). The quantum generator we are going to use is going to be acting on $n = 3$ qubits, meaning that it can represent $2^3 = 8$ values and will be implemented using uniform, normal and random data initialization. Each training epoch consists of batches of 2000 samples and the generated data are created by measuring the quantum generator. Now, we are going to show how to prepare the input state $|\psi_{in}\rangle$ so we can have a uniform and a normal distribution on the 3 qubits.

Uniform Initialization

To prepare a uniform distribution on 3 qubits, we simply have to apply 3 Hadamard gates, one for each qubit. So this leads to the following simple circuit:

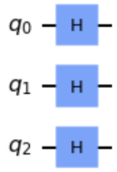


Figure 4.8: Preparing Uniform Distribution on $n = 3$ qubits.

Normal Initialization

Although using normal distribution on qubits can potentially require the use of advanced techniques, it is sufficient in our case to use a low-level variational model to load an approximate normal distribution as the initialization state. This can be achieved by fitting the parameters of the 3-qubit generator circuit with depth 1 with at least squares loss function. This means that we minimize the distance between the measurement probabilities of the circuit output p_j^i and the probability of a discretized normal distribution q^i, which is equal to:

$$\min_{\zeta} \sum_i ||p_j^i - q^i||^2 \tag{4.9}$$

The circuit implementing the normal distribution on 3 qubits can be seen in figure 4.9.

Log-Normal Data Distribution

The first test we are going to present is with **log-normal** real data distribution, with $\mu = 1$ and $\sigma = 1$, while the generator will follow a **normal** initialization.

The target line corresponds to the random data distribution in the real input, while the trained data are those generated. We can see that the generated data match extremely well with the real data, with very small differences that could be neglected.

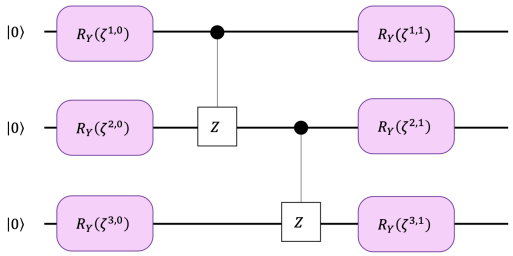
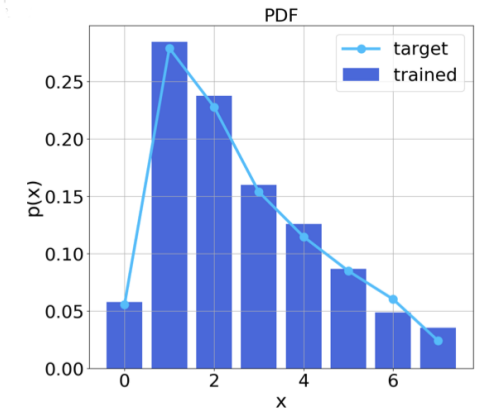
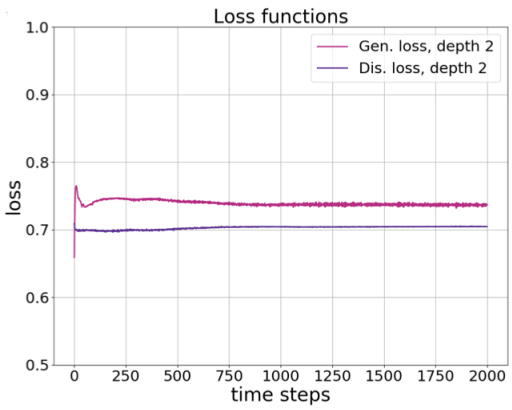


Figure 4.9: Variational Circuit for approximate loading of normal distribution.



(a) PDF of log-normal real data distribution and normal generator initialization.



(b) Loss functions.

Following, the table contains information on the Kolmogorov-Smirnov and the relative entropy statistics. Specifically we can see $\mu_{KS}, \sigma_{KS}, \mu_{RE}, \sigma_{RE}$. The null hypothesis here as described in Chapter 4.2.4, is that the distribution of the generated data is equal to the distribution of the real data. Each test was run 10 times, so the metric $n_{\leq b}$ shows how many times, during these 10 tests, the null hypothesis was accepted. Considering a confidence level of 95%, meaning $\alpha = 0.05$. For $s = 500$ samples, we define an acceptance bound b, which can be calculated as:

$$b = \sqrt{\frac{ln\frac{2}{\alpha}}{s}} = 0.0859 \tag{4.10}$$

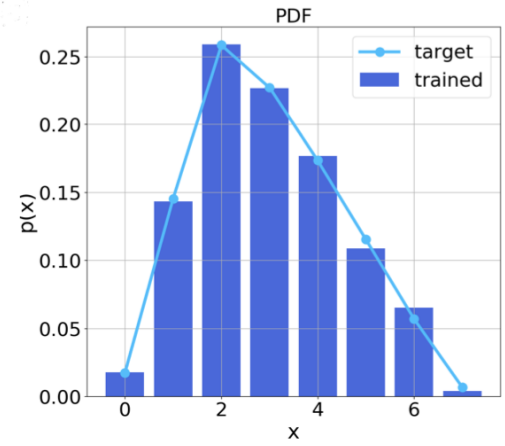
We can see that increasing the depth k of the circuit we can achieve better results, since at most cases we get the most runs accepted while having $k = 3$.

Data	Initialization	k	μ_{KS}	σ_{KS}	$n_{\leq b}$	μ_{RE}	σ_{RE}
Log-Normal	Uniform	1	0.0522	0.0214	9	0.0454	0.0856
		2	0.0699	0.0204	7	0.0739	0.0510
		3	0.0576	0.0206	9	0.0309	0.0206
	Normal	1	0.1301	0.1016	5	0.1379	0.1449
		2	0.1380	0.0347	1	0.1283	0.0716
		3	0.0810	0.0491	7	0.0435	0.0560
	Random	1	0.0821	0.0466	7	0.0916	0.0678
		2	0.0780	0.0337	6	0.0639	0.0463
		3	0.0541	0.0174	10	0.0436	0.0456

Table 4.2: Mean values and variances of KS and RE of log-normal real data distribution with different generator initializations.

Triangular Data Distribution

In the second test it is presented a **triangular** real data distribution, with lower level $l = 0$, upper level $u = 7$ and mode $m = 2$, with a **random** generator initialization. Again the same table as before is presented, for this case and showcases the metrics regarding the KS and Relative Entropy statistics.



(a) PDF of triangular real data distribution and random generator initialization.

(b) Loss functions

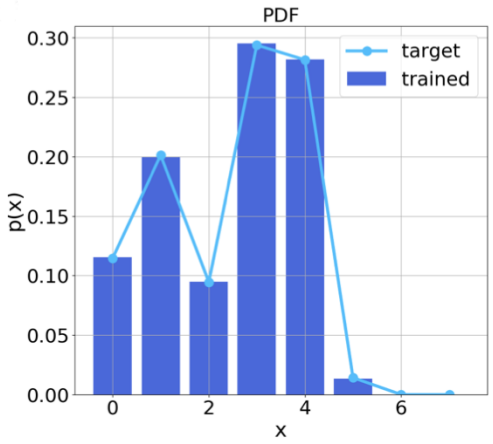
In this case as well the bigger the depth of the circuit, the more accepted tests we have. The best combination as seen is a triangular distribution of real data with a normal initialization of the generator.

Data	Initialization	k	μ_{KS}	σ_{KS}	$n_{\leq b}$	μ_{RE}	σ_{RE}
Triangular	Uniform	1	0.0880	0.0632	6	0.0624	0.0535
		2	0.0336	0.0174	10	0.0091	0.0042
		3	0.0695	0.1028	9	0.0760	0.1929
	Normal	1	0.0288	0.0106	10	0.0038	0.0048
		2	0.0484	0.0424	9	0.0210	0.0315
		3	0.0251	0.0067	10	0.0033	0.0038
	Random	1	0.0843	0.0635	7	0.1050	0.1387
		2	0.0538	0.0294	9	0.0387	0.0486
		3	0.0438	0.0163	10	0.0201	0.0194

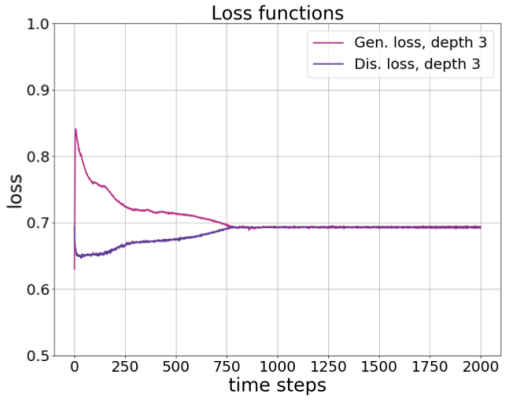
Table 4.3: Mean values and variances of KS and RE of triangular real data distribution with different generator initializations.

Bimodal Data Distribution

Finally, for the third test we test a **bimodal** real data distribution, with a **uniform** generator initialization. The bimodal distribution consists of two superimposed Gaussian distributions with $\mu_1 = 0.5, \sigma_1 = 1$ and $\mu = 3.5$ and $\sigma_2 = 0.5$. The statistics are as following, showing that the best combination of the two distributions are bimodal distribution for the real data with a normal initialization for the generator's data.



(a) PDF of bimodal real data distribution and uniform generator initialization



(b) Loss Functions

Data	Initialization	k	μ_{KS}	σ_{KS}	$n_{\leq b}$	μ_{RE}	σ_{RE}
Bimodal	Uniform	1	0.0.1288	0.259	0	0.3254	0.0146
		2	0.0358	0.0206	10	0.0192	0.0252
		3	0.0278	0.0172	10	0.0127	0.0040
	Normal	1	0.0509	0.0162	9	0.3417	0.0031
		2	0.0406	0.0135	10	0.0114	0.0094
		3	0.0374	0.0067	10	0.0018	0.0041
	Random	1	0.2432	0.0537	0	0.5813	0.2541
		2	0.0279	0.0078	10	0.0088	0.0060
		3	0.0318	0.0133	10	0.0070	0.0069

Table 4.4: Mean values and variances of KS and RE of bimodal real data distribution with different generator initializations.

4.2.6 Conclusions

As seen in this section, the quantum version of GANs, qGANs are able to reproduce very good results. We have seen that we can use them to learn and load the probability distributions into quantum states. These probabilities are able to match the distributions of the real data, if we choose the right initialization distribution and tune the best way possible the corresponding parameter values, presenting the ability that qGANs can find use in real world problems as well in many scientific fields.

Chapter 5

Quantum Machine Learning for Finance

In this chapter, we will see how **finance** can benefit from Quantum Machine Learning Algorithms. We will see how classical machine learning techniques apply on quantum computing, and finally we will go through some applications of qGANs in the field of finance, mainly for the task of option pricing [19, 26].

5.1 Classical Machine Learning Methods in Finance

Finance is defined as the management of money and includes activities such as investing, borrowing, lending, budgeting, saving, and forecasting. In this context, we must note that as it is natural, this field of science deals with many sensitive data, like personal data, investments and contracts among others. It also means that it can become extremely difficult to deal with some of the aforementioned activities, since a restriction in the data, can possibly lead to a halt in computations. This can potentially be solved with the use of machine learning algorithms and methods. This type of research has the ability to lead to reduced operational costs, due to more automated processes, increased revenues thanks to better productivity and a more satisfying user experience. One good thing, though, about the financial industry is the huge amount of historical data, which can be used to train the ML algorithms and to help in producing some impressive results. As of today, there are four key use cases in financial machine learning. Firstly, **process automation** is one of the most common applications, allowing the replacement of manual work, with automated repetitive tasks. As a result companies are enabled to optimize costs, improve customer experiences and enhance provided services. Then, we have **security threats** which, as normal, grow very fast with the increase of daily transactions, users and third-party integrations. Machine learning algorithms produce very good results in fraud detection. Namely, they can be used for *transaction monitoring*, where the algorithm supervises the actions of a user and assesses if they are characteristic of that user or they are irregular and for *network security*, where the models can be trained to analyze thousands of parameters in order to detect anomalies and possible cyber-threats. Next, we tackle the problem of **credit scoring**. Machine learning models are trained with

historical data on customers' information such as personal information like age, income, financial history etc. and then they are fed with such information of a new customer, with the task to classify them as high or low risk customers when it comes to loan granting. Finally, we have **algorithmic trading** where ML models track the news and trade results in real-time and detect patterns that can force stock prices to go up or down. It can then act proactively to sell, hold, or buy stocks according to its predictions. So, as we can easy see, machine learning algorithms can analyze thousands of data sources simultaneously, something that human traders cannot possibly achieve.

5.2 Quantum Machine Learning Methods in Finance

Quantum Machine Learning contains some of the most prominent techniques of classical machine learning, but has some properties that otherwise wouldn't be accessible. They can be used in the field of Finance in order to provide some very useful results. We will briefly discuss about the quantum machine learning applications in finance using the methods of **classification**, **regression** and **principle component analysis**.

Data Classification

Classification can tackle the problem of *credit scoring*. Each data point (customer) can be expressed as a vector, inside a vector space of all considered attributes (characteristics). Each vector belongs to a class (loan risk). The goal is that when given a new vector, the program must determine the class it is most likely going to belong to. One way to determine this is by returning the class which occurs most frequently among the k vectors which are closest to the new vector. Depending on the size of the training set and the number of attributes considered, finding a new vector's class can mean performing a large number of projections. Quantum Computers can perform these projections efficiently. The idea is casting each data point as a quantum state. Then we will estimate the classical distance $|\langle a|\,|b\rangle\,|$, between states $|a\rangle\,, |b\rangle$ by repeatedly performing swap tests. A **swap test** is a procedure in quantum computing that is used to check how much two states differ. This technique is a lot faster than the classical algorithms, even with taking into consideration the preparation of the states. The quantum circuit that implements this process is the following:

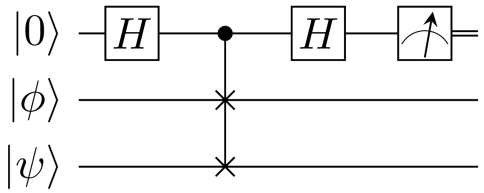


Figure 5.1: Quantum SWAP circuit

Regression

One common financial regression problem is the supply chain management. This is the art of meeting customer demand while avoiding unwanted stock. The idea behind regression is that given a new data point, instead of attributing to classes like in classification, we want to learn a

function from the trained data set. Regression algorithms are used to understand how the typical value of a response variable changes as an attribute is varied. The optimal fit parameters are found by minimizing the least-square error (LSE) between the training data and the values predicted by the model, which can become very computationally expensive when working with big data sets. There exists a linear algebra toolbox which allows to diagonalize matrices on quantum computers, exponentially faster than on classical computers. Researches have shown that for a sparse training data matrix, one could encode the model's optimal fit parameters into the amplitudes of a quantum state in time exponentially faster than the fastest classical algorithm.

Principle Component Analysis

It is very important in portfolio optimization to have a global vision of interest rates paths. This can be achieved through principle component analysis. As we have seen in Chapter 4, it can find the dominant eigenvalues and eigenvectors of a large matrix. The cost can reach $O(N^2)$. In the case though, that the matrix is sparse, or it has a big amount of data, we can say that the cost will be prohibitive. The quantum version of this algorithm, on the other hand, qPCA, can achieve exponentially better times. This algorithms finds approximations to the principal components of a correlation matrix, with a computational cost of $O(logN)^2$

In the following section, we will define a core financial problem, namely **European call option** and we will show how a qGAN can train a data loading unitary, in order to facilitate a financial derivative pricing, along with a QAE simulation to compare the results [22, 20].

5.3 Applications of qGANs in Finance: European Call Option

Firstly, we will start with some definitions in order to understand what a European call option is. **Financial derivative pricing** is a financial contract with a value that is based on some underlying asset. Derivative pricing models are techniques used by investors to try to find an objective measure of a derivative's true value. **Spot price** is the current price at which an asset can be bought or sold. On the other hand, **strike price** is a price at which a derivative contract can be bought or sold at its maturity date. Generally we can define **European call option** as a contract where the owner is permitted but not obliged to buy an underlying asset for a given strike price K at a predefined future maturity date T, where the asset's spot price at maturity S_T is assumed to be uncertain. We have two scenarios:

- If $S_T \leq K$, meaning that the current price at maturity is less than the predicted price, then there will be no payoff and there is no reason to exercise the option.
- If $S_T > K$, then buying the asset for price K and then immediately selling it for S_T can produce a payoff value of: $S_T - K$.

The payoff function is equal to $max\{0, S_T - K\}$. The goal is to evaluate the expected payoff $E[max\{0, S_T - K\}]$.

The Problem

In modern financial history, one of the most widely used concepts is the Black-Scholes model. It basically assumes that spot prices will have a log-normal distribution following a random walk. This assumption and with the use of some other variables can produce the fair price of a European call option, which corresponds to the expected payoff ($E[max\{0, S_T - K\}]$). Classically, we would use a Monte Carlo method, which it is not that efficient as we have already mentioned. We will use a qGAN to train the quantum generator to learn and load a log-normal probability distribution. After that we will implement two methods to estimate the payoff. The first one is through a simple sampling process, and the second is through a quantum amplitude estimation algorithm. We know from (2.22) that QAE on real data produces this state:

$$F\left|\psi\right\rangle_n\left|0\right\rangle = \sum_{i=0}^{2^n-1} \sqrt{1-f(i)}\sqrt{p_i}\left|i\right\rangle_n\left|0\right\rangle + \sqrt{f(i)}\sqrt{p_i}\left|i\right\rangle_n\left|1\right\rangle.$$

We will measure $|1\rangle$ to find the wanted estimation as:

$$\sum_{i=0}^{2^n-1} f(i)p_i,$$

where $f(i)$ derives from the payoff function and $p(i)$ from the probability distribution of the generator.

5.3.1 The Setup

For the following experiment we will test 10000 real samples with log-normal distributions, in combination with log-normal, uniform and normal initializations for the generator. The number of qubits tested here are mainly 3, meaning that our bounds are 0 and 7 but we have also tested for 4 qubits as well. Furthermore, the generator is a variational circuit consisting of R_Y rotations and CZ gates as described in *Chapter 4*. Then we define θ, which is the default parameter for generator circuits, and the final circuit we will use is shown below:

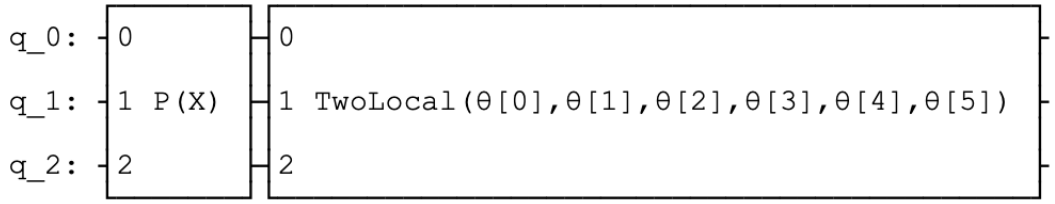


Figure 5.2: Generator circuit

The first block (**"P(X)"**) is the circuit used to encode the distribution into qubit amplitudes.

The "**TwoLocal**" block is a quantum circuit that implements the rotations along with entanglements. It can be implemented as a python class in QiSkit and for our case it is very useful, since

the circuit of the quantum generator consists of Y-rotations and an entanglement block (U_{ent}). Below, you can see a decomposition of the TwoLocal circuit. The black boxes are implementing the CZ gates.

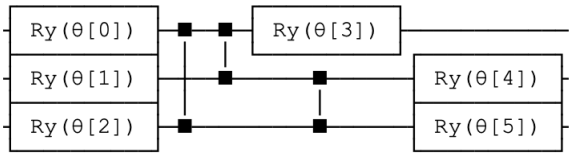


Figure 5.3: TwoLocal circuit

For the case that we will study, we will set the strike price = 2, meaning that after that point there can be a payoff, but before that payoff is equal to 0. The payoff function is shown in the following figure.

Figure 5.4: Payoff Function

Next, we will use the quantum amplitude estimation algorithm to estimate the payoff value. We will use the European option pricing class in that is implemented in the **"qiskit.finance.applications"**. For this we define the number of qubits, the bounds and the strike price, which are the same as before, and we will load the uncertainty model that we implemented earlier, which is the quantum generator circuit that loads the distribution. After that we define a confidence level α and run an iterative amplitude estimation algorithm to compare the results.

5.3.2 Training the qGAN

The first step is to generate some trained data to input in the circuit that estimates the payoff. To train the qGAN, to be able to generate samples that reach the same values as those of the real data, we will use a quantum generator and a classic discriminator as described in chapter 4. For the quantum generator we will implement the circuit that we described in the qGAN section, which is a quantum circuit consisting of R_Y and CZ gates. For our tests we have set the training epochs to be 20. The classical discriminator will be implemented using the NumPyDiscriminator class.

In the following sections, we will see how well the generated data distributions reach the real data ones.

Log-Normal Initialization

For the first test we will use a log-normal initialization of the generator's input data, with $\mu = 1$ and $\sigma = 1$. As initialization parameters we will be using random values. Since the number of parameters (θ) is the same as the number of integral values within the bounds, for $n = 3$ qubits we will have $\theta(0), ..., \theta(7)$.

Normal Initialization

For a normal initialization of the generator's input with $\mu = 1, \sigma = 1$, we can see that the training doesn't require many runs, since the CDF of the real and the trained data are very close comparing them after 1 and 6 training runs.

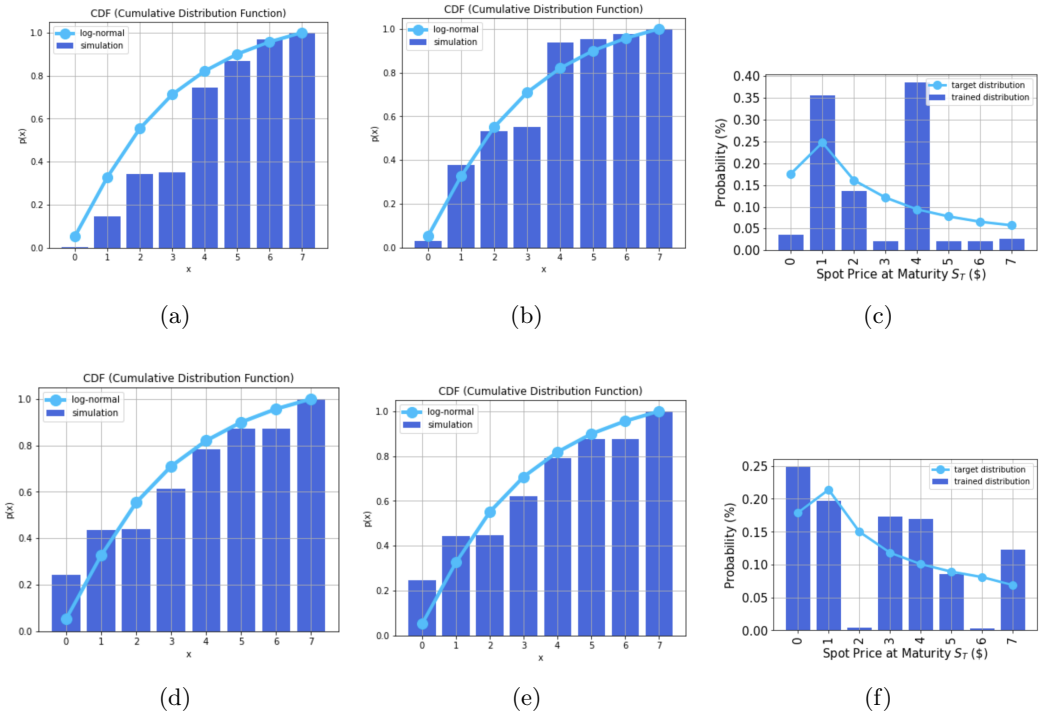


Figure 5.5: Figures (a) and (b) are the CDF of real and trained data with log-normal generator initialization after 1 and 5 training runs respectively. Figures (d) and (e) are the respective CDFs for normal generator initialization. Figures (c) and (f) show the PDF of real and trained data that are used in the payoff calculations for the European call option problem for the log-normal and normal-initializations.

The analytically computed payoff values are found by taking for trained and target data, their values and multiplying them with array $[0, 0, 0, 1, 2, 3, 4, 5]$, since we have set strike price $= 2$, while

the payoff computed through QAE is found as described earlier. The values in the table show that we were able to achieve very close results from the trained data using the generator. As for the payoff values from the normal initialization, they are very close, better than in the log-normal case meaning that it is better to chose the current initialization over the first one. It also is good to note that there have been many tests run with different parameters, and the results illustrated, are the best matches we could produce.

Initialization	Payoff w.r.t. **target data**	Payoff w.r.t. **trained data**	Payoff computed with **QAE**
Log-normal	1.0573	1.3115	1.3327
Normal	1.2532	1.3893	1.4206

Table 5.1: Payoff values calculated with respect to target (real) and trained (generated) data as well as with QAE, for log-normal and normal generator initializations.

Uniform Initialization

For the uniform initialization of the generator with 3 input qubits, we simply have to apply 3 Hadamard gates, one for each qubit, leading to:

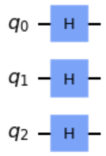


Figure 5.6: Preparing uniform distribution on $n = 3$ qubits

As far as the PDF of the real and generated samples is concerned we obtain the best results in this case, which showcases as well in the payoff values.

Uniform Initialization with 2 Circuit Repetitions

Since the results with uniform generator initialization are very satisfying, we can try and test how good the circuit behaves if we increase its repetitions to 2. The new generator circuit will be:

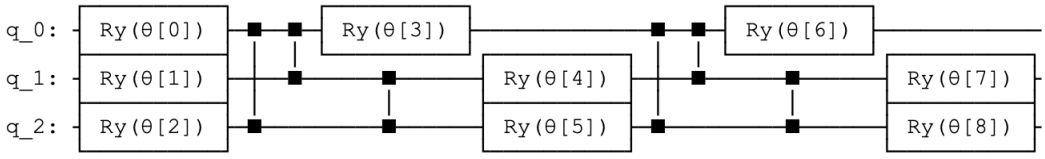


Figure 5.7: Quantum circuit for the generator with 2 repetitions

This model doesn't require much training since the qGAN matches very good from the first run the real data, while the payoff values are also very close.

Uniform Initialization with 4 Input Qubits

This time we will see how good the model works when increasing the number of input qubits. Again we will use uniform initialization leaving everything as is. In this case, because we will be using 4 qubits and therefore $2^4 = 16$ samples, for the payoff computation we will need to add more elements in the array to be able to do the multiplications, i.e. [0,0,0,1,2,3,4,5,6,7,8,9,10,11,12,13]. The payoff function therefore can be seen as:

Figure 5.8: Payoff function for 4-qubits

In the following table, we can see the different payoff values for the 3 aforementioned cases.

Initialization	Payoff w.r.t. **target data**	Payoff w.r.t. **trained data**	Payoff computed with **QAE**
Uniform-3 qubits	1.0593	1.0370	1.0986
Uniform-2 reps.	1.1264	1.2196	1.4209
Uniform-4 qubits	2.0255	2.0723	2.2327

Table 5.2: Payoff values calculated with respect to target (real) and trained (generated) data as well as with QAE, for uniform generator initialization variations.

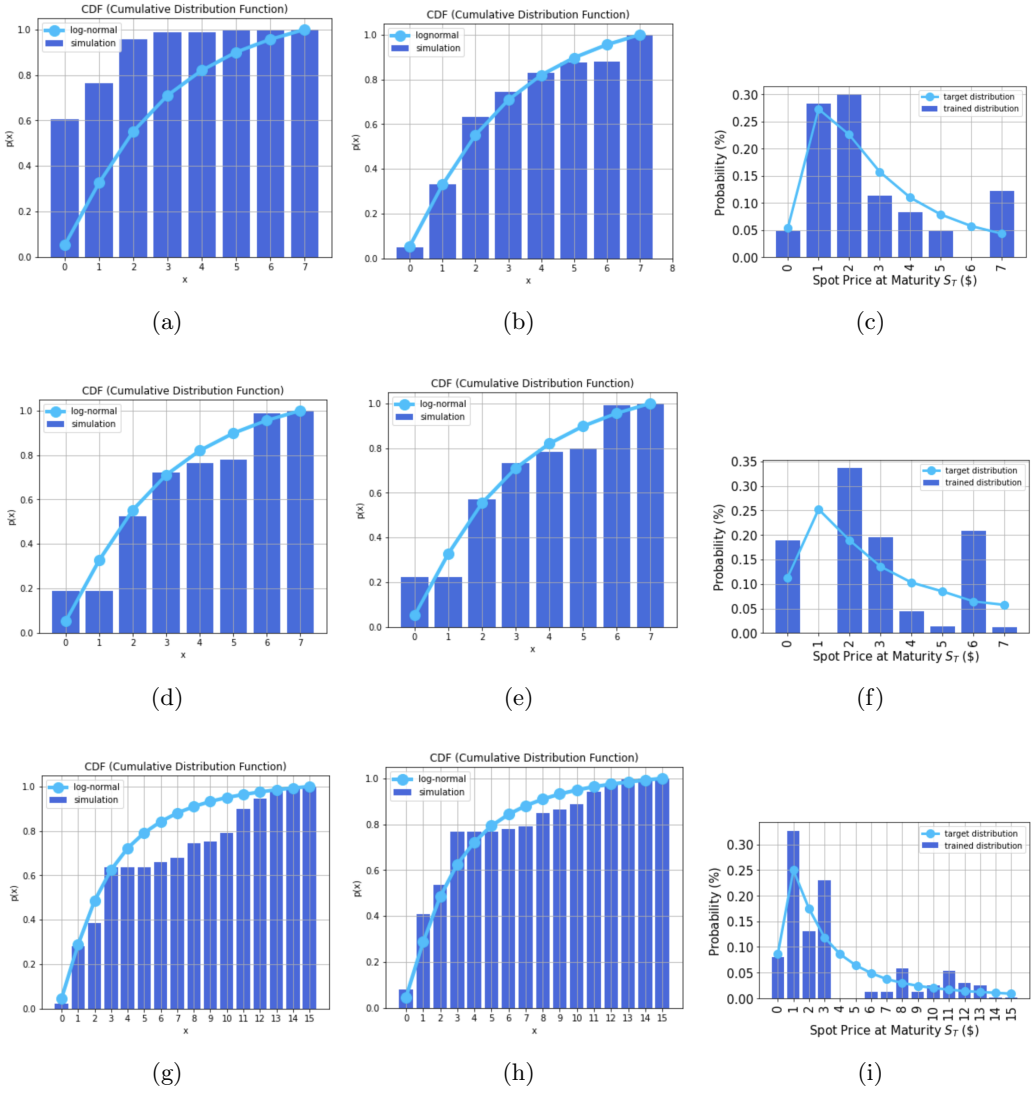


Figure 5.9: Results of different uniform initializations. (a) and (b) are the CDF of real and trained data with uniform generator initialization on 3 qubits after 1 and 5 training runs respectively. (d) and (e) are the respective CDFs for uniform generator initialization with 2 circuit repetitions while (g) and (h) are the CDFs for uniform generator initialization with 4 qubits. (c), (f) and (i) show the PDF of real and trained data that are used in the payoff calculations for the European call option problem for each one of the 3 cases.

5.4 Conclusion

As we have seen from the results of the tests, we can train the qGANs' generator, so that we can achieve distributions very close to those of the real data. In our tests, not only were we able to illustrate the similarities of the distributions, but in practice we could compute the expected values of payoff in a European call option, with great accuracy. We should note here that the qGAN model doesn't offer any speedup compared to the classical method of data loading. The real advantage comes through QAE, which offers important speedup compared to its classical analog, the Monte Carlo method.

Conclusion and Future Work

In the course of this thesis, we have studied the way that hybrid machine learning works. Using the classical ideas and methods of machine learning, combined with quantum properties and quantum computation, we have seen that one can benefit in producing very good and prominent results. Quantum computing has seen a huge rise during the last decade with the introduction of new algorithms and concepts. Its ability to pull through the same tasks as many classical algorithms and methods, but in significantly less computational time, has produced the premise that through quantum computing, the research and development in many fields could see a huge rise.

We have studied the two main approaches of quantum machine learning, namely the idea of fault-tolerant devices and the use of NISQ devices. While the first one is far from being utilized, studies have shown that there exists in fact the potential of implementing QML algorithms and achieving exponential speedups compared to classical ML algorithms. Currently, what we are using are the NISQ devices, which define noisy hybrid approaches to solve machine learning problems. We use quantum and classical computing to train and run the algorithms' subroutines. Studies have shown that there has been progress in this field as well, by reducing the noise and correcting the error, but it is widely expected that the quantum supremacy will occur when we will be able to implement the first type of devices.

One specific machine learning model that we studied in this thesis, is the general adversarial network. Since it is a generative model, it can be used in problems in many fields of every day life, such as weather prediction, financial stock predictions, image generation and others. GANs can be used, based on some real data to generate very similar new ones and therefore help in the research process in the aforementioned fields. We have studied the quantum analog of the GANs, the quantum adversarial networks. While they can be fully quantum, meaning that all of their elements can be quantum circuits, we focused on one variation of combining quantum with classical methods. In order to learn and load data distributions and produce some samples we used a quantum circuit and then these samples would be classified by a classical neural network, along with some real ones, as *"real"* or *"fake"*. By training the qGAN, we want to see how accurately the quantum circuit can create a probability distribution that can match the real one, needing $O(poly(n))$ gates. We have furthermore tested some qGANs in a specific real world application in the field of Finance. That is, we have used them to generate probability distributions and compare them with the real distributions and from there compute a very important value which is the expected payoff. We have presented the European call option problem, how it works and the importance of the payoff term for day to day investments. We have seen that with the combination of qGANs and quantum amplitude estimation, we can produce very satisfying results meaning that with some future work,

we could potentially rely on quantum generative models for the day to day computations.

Future work on qGANs can include the implementations of different variations of classical GANs. The goal is and will be to find the optimal quantum generator and discriminator structures, since it is not yet clear which structure is the most suitable for each problem we study. Also worth studying is the training process and how we could implement different ones, in order to possibly achieve better results. Finally, a potentially interesting variation worth studying is the representation of the capabilities of qGANs with other data types. Encoding data into qubit basis states naturally induces a discrete and equidistantly distributed set of represented data values. However, we could see and study the results of the compatibility of qGANs with continuous or non-equidistantly distributed values. Finally, we can try and see what other machine learning problems and applications in the field of Finance can benefit from the use of qGAN models.

Bibliography

[1] Mohammad H Amin, Evgeny Andriyash, Jason Rolfe, Bohdan Kulchytskyy, and Roger Melko. Quantum boltzmann machine. *Physical Review X*, 8(2):021050, 2018.

[2] Dominic W Berry, Andrew M Childs, and Robin Kothari. Hamiltonian simulation with nearly optimal dependence on all parameters. In *2015 IEEE 56th annual symposium on foundations of computer science*, pages 792–809. IEEE, 2015.

[3] Vaishali Bhatia and KR Ramkumar. An efficient quantum computing technique for cracking rsa using shor's algorithm. In *2020 IEEE 5th international conference on computing communication and automation (ICCCA)*, pages 89–94. IEEE, 2020.

[4] Gilles Brassard, Peter Hoyer, Michele Mosca, and Alain Tapp. Quantum amplitude amplification and estimation. *Contemporary Mathematics*, 305:53–74, 2002.

[5] Gavin E Crooks. Gates, states, and circuits. 2020.

[6] Pierre-Luc Dallaire-Demers and Nathan Killoran. Quantum generative adversarial networks. *Physical Review A*, 98(1):012324, 2018.

[7] Vedran Dunjko, Jacob M Taylor, and Hans J Briegel. Quantum-enhanced machine learning. *Physical review letters*, 117(13):130501, 2016.

[8] Florian Eckerli and Joerg Osterrieder. Generative adversarial networks in finance: an overview. *arXiv preprint arXiv:2106.06364*, 2021.

[9] Daniel J Egger, Ricardo García Gutiérrez, Jordi Cahué Mestre, and Stefan Woerner. Credit risk analysis using quantum computers. *IEEE Transactions on Computers*, 70(12):2136–2145, 2020.

[10] Dmitry Grinko, Julien Gacon, Christa Zoufal, and Stefan Woerner. Iterative quantum amplitude estimation. *npj Quantum Information*, 7(1):1–6, 2021.

[11] Aram W Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical review letters*, 103(15):150502, 2009.

[12] Ashish Kapoor, Nathan Wiebe, and Krysta Svore. Quantum perceptron models. *Advances in neural information processing systems*, 29, 2016.

[13] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. Quantum principal component analysis. *Nature Physics*, 10(9):631–633, 2014.

[14] Seth Lloyd and Christian Weedbrook. Quantum generative adversarial learning. *Physical review letters*, 121(4):040502, 2018.

[15] Guang Hao Low, Theodore J Yoder, and Isaac L Chuang. Quantum inference on bayesian networks. *Physical Review A*, 89(6):062315, 2014.

[16] Batta Mahesh. Machine learning algorithms-a review. *International Journal of Science and Research (IJSR).[Internet]*, 9:381–386, 2020.

[17] David McMahon. *Quantum computing explained*. John Wiley & Sons, 2007.

[18] Kouhei Nakaji. Faster amplitude estimation. *arXiv preprint arXiv:2003.02417*, 2020.

[19] Román Orús, Samuel Mugel, and Enrique Lizaso. Quantum computing for finance: Overview and prospects. *Reviews in Physics*, 4:100028, 2019.

[20] QiSkit. Credit risk analysis. `https://qiskit.org/documentation/finance/tutorials/09_credit_risk_analysis.html#Expected-Loss`.

[21] QiSkit. Grover's algorithm. `https://qiskit.org/textbook/ch-algorithms/grover.html`. Accessed: 2010-09-30.

[22] QiSkit. Option pricing with qgans. `https://qiskit.org/documentation/stable/0.25/tutorials/finance/10_qgan_option_pricing.html`.

[23] QiSkit. qgans for loading random distributions. `https://qiskit.org/documentation/machine-learning/tutorials/04_qgans_for_loading_random_distributions.html`.

[24] Patrick Rebentrost, Masoud Mohseni, and Seth Lloyd. Quantum support vector machine for big data classification. *Physical review letters*, 113(13):130503, 2014.

[25] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[26] Nikitas Stamatopoulos, Daniel J Egger, Yue Sun, Christa Zoufal, Raban Iten, Ning Shen, and Stefan Woerner. Option pricing using quantum computers. *Quantum*, 4:291, 2020.

[27] Nathan Wiebe, Daniel Braun, and Seth Lloyd. Quantum algorithm for data fitting. *Physical review letters*, 109(5):050505, 2012.

[28] Stefan Woerner and Daniel J Egger. Quantum risk analysis. *npj Quantum Information*, 5(1):1–8, 2019.

[29] Stefan Woerner and Daniel J Egger. Supplementary information to quantum risk analysis. *npj Quantum Information*, 5(1):1–8, 2019.

[30] Christa Zoufal, Aurélien Lucchi, and Stefan Woerner. Quantum generative adversarial networks for learning and loading random distributions. *npj Quantum Information*, 5(1):1–9, 2019.

[31] Christa Zoufal, Aurélien Lucchi, and Stefan Woerner. Supplementary information for quantum generative adversarial networks for learning and loading random distributions. *npj Quantum Information*, 5(1):1–9, 2019.

Appendices

Appendix A

Grover's Algorithm

Grover's algorithm, also known as quantum search algorithm, is an algorithm for unstructured search that finds with high probability the unique input to a black box function that produces a particular output value. It can produce the result in $O(\sqrt{N})$ evaluations of the function, while the best classical result can be obtained after $O(N)$ evaluations. As we can see, the algorithm provides quadratic speedup, which is very important, especially when N is very big. Now, we will deconstruct the algorithm to understand the way it works. We will split the problem into two subsections, the problem description and the algorithm analysis [21].

The Problem

Suppose we have a list of N unsorted numbers as seen in the figure below, and let's assume that we want to find the number in the purple box. A classical approach would have us search an average of $N/2$ items of the list, meaning that the in the worst case scenario, we would have to go through all N numbers in order to find the one we want.

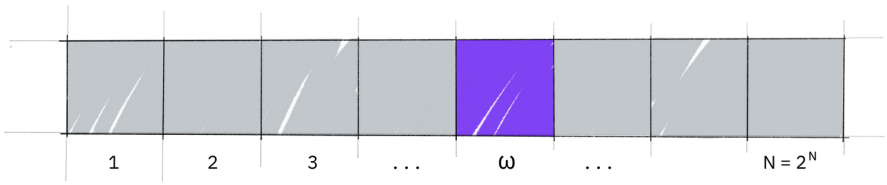


Figure A.1: List of N numbers

The mathematical approach to this problem is that we firstly consider a function $f : \{0, 1, ..., N-1\} \to \{0, 1\}$. The function returns value $f(x) = 1$ if and only if the data that x points, satisfies the search criterion. We assume that this is true only for one value, namely w.

Suppose also that all the items in the list can be encoded to all the possible states the number of qubits can be in, meaning that for N different items we have 2^N qubits. For example

if $n = 4$ (n: number of qubits), the list has all the states that can be encoded using 4 qubits: $|0000\rangle, |0001\rangle, ..., |1111\rangle$ (the states $|0\rangle \rightarrow |16\rangle$).

We can define a unitary operator U_w to implement the action of $f(x)$ for any state $|x\rangle$ as:

$$\begin{cases} U_w |x\rangle = -x & \text{for } x = w \implies f(x) = 1 \\ U_w |x\rangle = x & \text{for } x \neq w \implies f(x) = 0 \end{cases}$$

This action can also be written as: $U_w |x\rangle = (-1)^{f(x)} |x\rangle$. This is the oracle we will use and its matrix representation is:

$$U_w = \begin{bmatrix} (-1)^{f(0)} & 0 & ... & 0 \\ 0 & (-1)^{f(1)} & ... & 0 \\ 0 & 0 & ... & 0 \\ 0 & 0 & ... & (-1)^{f(2^n-1)} \end{bmatrix}$$

At the beginning of the search, we have the same knowledge of the location of every item in the list, which can be expressed as a superposition of item $|x\rangle$: $|s\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{N-1} |x\rangle$. A very useful procedure is **amplitude amplification**, where in our case we use it to take advantage of the fact that the amplitude of the qubit is also its probability. By enhancing the amplitude of the marked item, and at the same time by shrinking the other amplitudes, we can measure the correct state with near perfect certainty.

The Algorithm

The circuit that we want to implement to run this algorithm is the following:

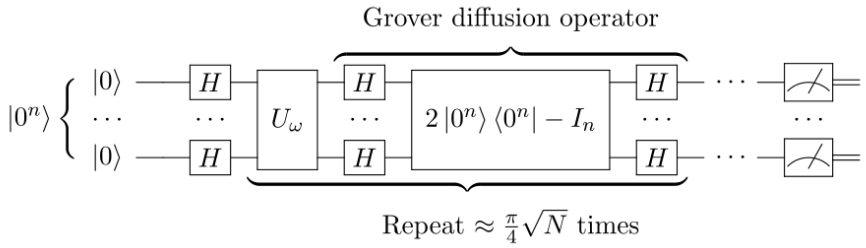


Figure A.2: Grover's algorithm implementation circuit

We will use three states. State $|w\rangle$ which stands for the winner qubit, the one we search, $|s\rangle$ which is the same state as we mentioned above and we will introduce a new state $|s'\rangle$ which is state $|s\rangle$ by removing $|w\rangle$. The algorithm consists of three major steps:

- Initialize the system to the superposition over all states.

  We start from state $|0\rangle^{\otimes x}$ and then by applying the Hadamard gate, we can create the superposition state, meaning at the same time that we are initializing the amplitude amplification process.

$$|s\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{N-1} |x\rangle \tag{A.1}$$

.

(a) Initial state of superposition

(b) Amplitude of superposition state $|s\rangle$

Figure A.3: Initialization step

Then we apply the oracle of the Grover's algorithm which takes place through steps 2 through 3.

- Apply operator U_w.

  The way this operator works, is that it reflects state $|s\rangle$ around $|s'\rangle$, making its amplitude negative and at the same time, decreasing the average amplitude of all the states. As we have discussed above, the unitary U_w changes the sign of the qubit it acts on based on whether it is the qubit we search or not. So by applying it on each superpositioned state $\frac{|0\rangle - |1\rangle}{\sqrt{2}}$ we get the following results:

$$U_w |s\rangle \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |s\rangle \frac{|f(x) \oplus 0\rangle - |f(x) \oplus 1\rangle}{\sqrt{2}} \tag{A.2}$$

  If $f(x) = 1$ the result we get after the application of the unitary is as expected:

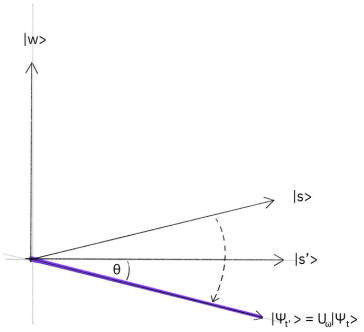
$$U_w |s\rangle \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |s\rangle \frac{|1 \oplus 0\rangle - |1 \oplus 1\rangle}{\sqrt{2}} = - |s\rangle \frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{A.3}$$

  If $f(x) = 0$ the result we get after the application of the unitary is as expected:

$$U_w |s\rangle \frac{|0\rangle - |1\rangle}{\sqrt{2}} = |s\rangle \frac{|0 \oplus 0\rangle - |0 \oplus 1\rangle}{\sqrt{2}} = |s\rangle \frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{A.4}$$

  We can write this operation as:

$$U_w |x\rangle \rightarrow (-1)^{f(x)} |s\rangle \tag{A.5}$$



(a) Rotation of state $|s\rangle$ around $|s'\rangle$

(b) Lowered average amplitude (dash line)

Figure A.4: Applying unitary U_w

- Apply the Grover diffusion operator $U_s = 2\,|s\rangle\,\langle s| - I$.

  Now we apply the operator U_s on the rotated state $|s\rangle$ as we have produced during step 2. The result is given by:

$$U_s = (2\,|s\rangle\,\langle s| - I)U_w \tag{A.6}$$

  We basically perform a total of U_sU_w rotation around the initial state $|s\rangle$.

(a) Grover Operator on $U_w\,|s\rangle$

(b) Lowered Average Amplitude (dash line)

Figure A.5: Applying unitary U_S

The transformation U_sU_w rotates the initial state $|s\rangle$ closer towards the winner $|w\rangle$. The action of the reflection U_s in the amplitude bar diagram can be understood as a reflection about the average amplitude. We then go to step 2 to repeat the application. This procedure will be repeated several times to zero in on the winner.

Conclusions

As we can see, after the rotation the amplitude of the wanted qubit is amplified compared to the other's. Roughly $\sqrt{N}$ rotations are enough to achieve the result we want. Therefore we can say that the total time needed to implement and get results from the Grover's algorithm is $O(\sqrt{N})$ compared to classical $O(N)$.

Appendix B

Applications of QAE in Finance

B.1 Quantum Risk Analysis

Consider a portfolio of K assets. A portfolio can be defined as a set of investments. Then, we define a multivariate random variable $(L_1, ..., L_K) \in \mathbf{R}^K_{\geq 0}$, which denotes each possible loss associated to each asset. The total loss can be calculated as:

$$L = \sum_{k=1}^{K} L_k$$

A key metric for risk analysis, used by many companies, is Value at Risk (VaR). For a given confidence level (a), it is defined as the smallest total loss that still has a probability greater than or equal to a. The mathematical definition of VaR is:

$$VaR_a[L] = \underset{x \geq 0}{inf}\{x | P[L \leq x] \geq a\} \tag{B.1}$$

We assume that all losses are independent and can be expressed as: $L_k = \lambda_k X_k$, where λ_k is a loss-given default, and X_k corresponds to Bernoulli variable, with:

$$\Pr\{X_k = x\} = \begin{cases} p_k & , x = 1 \\ 1 - p_k & , x = 0 \end{cases}$$

Here we can calculate the expected value of the total loss as:

$$E[L] = \sum_{k=1}^{K} \lambda_k p_k,$$

whereas $VaR_a[L]$ requires a Monte Carlo method.

In a more realistic scenario though, X_k are not independent but follow a conditional independence scheme. The default probability of every asset k follows a Gaussian conditional independence

model, that is, for a given z sampled from a latent variable Z, which follows a normal distribution, the default probability is given through:

$$p_k(z) = F(\frac{F^{-1}(P_k^0) - \sqrt{\rho_k^z}}{\sqrt{1-\rho_k}}) \tag{B.2}$$

where **F**: is the cumulative distribution function (**CDF**) of the standard normal distribution and ρ_k is the sensitivity of X_k to Z, meaning that it shows how the default probability of k behaves with respect to Z.

To estimate VaR, we use QAE to efficiently evaluate the CDF of the total loss. We will find, through operator A, a value a such that $a = P[L \leq x]$ for a given $x \geq 0$ and then apply a bisection search to find the smallest $x_a \geq 0$, such that $P[L \leq x_a] \geq a$, implying that: $x_a = VaR_a[L]$. The way we do this is by using 3 operators, that combined implement the operator A. These are:

- **Operator U**: It is used to load the uncertainty model.
- **Operator S**: It computes the total loss into a quantum register with n_s qubits
- **Operator C**: It flips a target qubit if the total loss is less or equal to a given level x which is then used to search for the VaR.

Thus, we can define operator A as: $A = CSU$. Below we present a high level model of operator A and then we will break down and see how the 3 operators work separately.

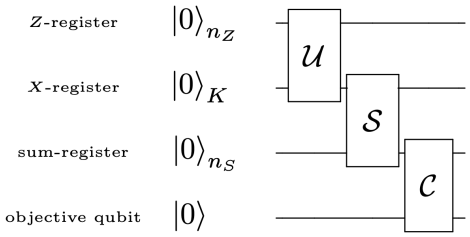


Figure B.1: High level quantum circuit of operator A

The first qubit register contains n_Z qubits that represent variable Z, the second qubit register contains K qubits that represent variable X_k, the third qubit register contains n_s qubits that represent the sum of losses and finally we have a single $|0\rangle$ that acts as a control qubit for the operator C. As you can see, and we will discuss in detail later, the starting from the top the output of each operator acts as an input to the following, meaning that the output of operator U is one of the inputs of operator S and the output of S is one of the inputs of operator C.

B.1.1 The Operator U

The work of **Operator U** is to load the uncertainty model that we described above. When the default events $\{X_1, X_2, ..., X_K\}$ are uncorrelated, we can encode the X_k of each asset in the state of a corresponding qubit, by applying Y-Rotations $R_Y(\theta_p^k)$ with $\theta_p^k = 2\arcsin(\sqrt{p_k})$. Therefore we

can have as a first step for U:

$$U = \bigotimes_{k=1}^{K} R_Y(\theta_p^k)$$

This prepares k in the state $\sqrt{1-p_k}\,|0\rangle + \sqrt{p_k}\,|1\rangle$.

Next we want to create a quantum state in a register of n_Z qubits that represents variable Z and as explained is used to control k. We can encode the realizations of Z as:

$$|i\rangle = \sum_{i=0}^{2^{n_Z}-1} \sqrt{p_z^i}\,|z_i\rangle$$

Combining these two equations and keeping in mind that Z-registers control the X-registers, we can write the final output state of operator U as:

$$|\psi\rangle = \sum_{i=0}^{2^{n_Z}-1} \sqrt{p_z^i}\,|z_i\rangle \bigotimes_{k=1}^{K} (\sqrt{1-p_k(z_i)}\,|0\rangle + \sqrt{p_k(z_i)}\,|1\rangle) \tag{B.3}$$

The $|1\rangle$ state of qubit k corresponds to a loss for asset k.

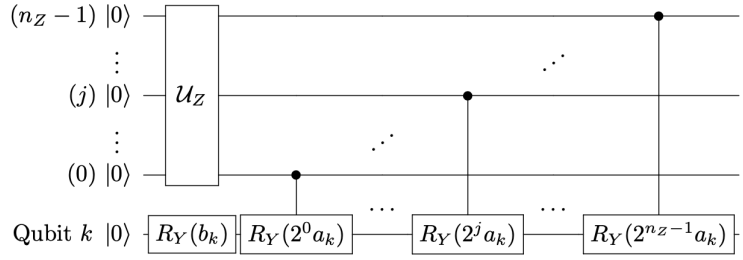


Figure B.2: Quantum Circuit of Operator Z

Note: Operator U_z creates the state $|i\rangle$.

B.1.2 The Operator S

In this operator, we need to compute the resulting total loss for every realization of X_k. The way we do this is by using a **weighted sum operator**.

The general way this operator works is: for given quantum states $|q_1\rangle, ..., |q_{n-1}\rangle$ and for non-negative integer weights $w_0, ..., w_n - 1$ the circuit performs the operation:

$$|q_0 ... q_{n-1}\rangle\,|0\rangle_s \rightarrow |q_0 ... q_{n-1}\rangle\,|\sum_{j=0}^{n-1} w_j q_j\rangle_s \tag{B.4}$$

where s is the number of qubits required. This can be computed as:

$$s = \lfloor log(\lambda_1 + ... + \lambda_k) \rfloor + 1$$

So, in the case we study we have:

$$|x_1, ..., x_K\rangle_K |0\rangle_{n_s} \rightarrow |x_1, ..., x_K\rangle_K |\lambda_1 x_1 + ... + \lambda_K x_K\rangle_{n_s} \tag{B.5}$$

Once the total loss is computed, we can map it through to the amplitude of an objective qubit, leading to final output:

$$|L\rangle_{n_s} |0\rangle \rightarrow |L\rangle_{n_s} (\sqrt{1 - \frac{L}{2^{n_s} - 1}} |0\rangle + \sqrt{\frac{L}{2^{n_s - 1}}} |1\rangle) \tag{B.6}$$

This allows to run amplitude estimation to evaluate the total loss.

B.1.3 The Operator C

Finally, we need an operator that compares a particular loss realization to a given x. Then it flips the target qubit from $|0\rangle$ to $|1\rangle$ if the loss is less or equal to x. It basically acts as a comparator and works as following:

$$C : |L\rangle_n |0\rangle \rightarrow \begin{cases} |L\rangle_n |1\rangle & , if L \leq x \\ |L\rangle_n |0\rangle & , otherwise \end{cases}$$

B.1.4 Computing VaR

After constructing the operator A, we can use QAE to estimate value $a = P[L \leq x]$. Then we use a classical method called bisection search. In general it is used to determine the smallest level such that the objective value is still larger than the target. In our case we use it to find the smallest x_a such that $P[L \leq x_a] \geq a$.
Value x_a is the wanted result and: $x_a = VaR_a[L]$.

Classically we could use Monte Carlo method, but as explained it would need $O(M^{-1/2})$ steps while QAE requires $O(M^{-1})$, which is a quadratic speedup.

Computation of CVaR

Derived from VaR, **Conditional Value at Risk** is another metric used for risk analysis. CVaR is the expected value of the loss, conditional to it being larger than or equal to the VaR. To calculate it we first define a function f such that:

$$f(L) = \begin{cases} 0 & , if L \leq VaR \\ L & , if L > VaR \end{cases}$$

Appendix C

Code

C.1 Code used for QAE example

```python
import numpy as np
from qiskit.circuit import QuantumCircuit

#define circuits A, Q
class BernoulliA(QuantumCircuit):
    """A circuit representing the Bernoulli A operator."""

    def __init__(self, probability):
        super().__init__(1)  # circuit on 1 qubit

        theta_p = 2 * np.arcsin(np.sqrt(probability))
        self.ry(theta_p, 0)


class BernoulliQ(QuantumCircuit):
    """A circuit representing the Bernoulli Q operator."""

    def __init__(self, probability):
        super().__init__(1)  # circuit on 1 qubit

        self._theta_p = 2 * np.arcsin(np.sqrt(probability))
        self.ry(2 * self._theta_p, 0)

    def power(self, k):
        # implement the efficient power of Q
        q_k = QuantumCircuit(1)
        q_k.ry(2 * k * self._theta_p, 0)
        return q_k
```

Figure C.1: Defining the operators.

```
p =0.6
A = BernoulliA(p)
Q = BernoulliQ(p)

from qiskit.algorithms import EstimationProblem

#define the estimation problem
problem = EstimationProblem(
    state_preparation=A,  # A operator
    grover_operator=Q,  # Q operator
    objective_qubits=[0],  # the "good" state Psi1 is identified as measuring |1> in qubit 0
)

from qiskit import BasicAer
from qiskit.utils import QuantumInstance

#use qiskit's statevector simulator
#The StatevectorSimulator supports CPU and GPU simulation methods and additional configurable options.
backend = BasicAer.get_backend("statevector_simulator")
quantum_instance = QuantumInstance(backend)

#solve the problem using QAE
from qiskit.algorithms import AmplitudeEstimation

ae = AmplitudeEstimation(
    num_eval_qubits=3,  # the number of evaluation qubits specifies circuit width and accuracy
    quantum_instance=quantum_instance,
)
```

Figure C.2: Defining the problem and setting the simulator.

```
#run the problem
ae_result = ae.estimate(problem)
print(ae_result.estimation)

import matplotlib.pyplot as plt

# plot estimated values
gridpoints = list(ae_result.samples.keys())
probabilities = list(ae_result.samples.values())

plt.bar(gridpoints, probabilities, width=0.5 / len(probabilities))
plt.axvline(p, color="r", ls="--")
plt.xticks(size=15)
plt.yticks([0, 0.25, 0.5, 0.75, 1], size=15)
plt.title("Estimated Values", size=15)
plt.ylabel("Probability", size=15)
plt.xlabel(r"Amplitude $a$", size=15)
plt.ylim((0, 1))
plt.grid()
plt.show()

#draw the circuit
ae_circuit = ae.construct_circuit(problem)
ae_circuit.decompose().draw(
    "mpl", style="iqx"
)  # decompose 1 level: exposes the Phase estimation circuit!
```

Figure C.3: Running the problem and getting results.

C.2 Code used for qGAN Training & Applications

The code for the training of the qGAN and the financial example was written in Python and implemented using QiSkit libraries.

```
import matplotlib.pyplot as plt
import numpy as np

from qiskit import Aer, QuantumRegister, QuantumCircuit
from qiskit.circuit import ParameterVector
from qiskit.circuit.library import TwoLocal
from qiskit.quantum_info import Statevector

from qiskit_machine_learning.algorithms import NumPyDiscriminator, QGAN
from qiskit.algorithms.optimizers import ADAM, Optimizer, GradientDescent


from qiskit.utils import QuantumInstance
from qiskit.algorithms import IterativeAmplitudeEstimation, EstimationProblem
from qiskit_finance.applications.estimation import EuropeanCallPricing
from qiskit_finance.circuit.library import LogNormalDistribution, NormalDistribution, UniformDistribution

seed = 71
np.random.seed = seed
algorithm_globals.random_seed = seed
```

Figure C.4: Imports

```
# Number training data samples
N = 10000

# Load data samples from log-normal distribution with mean=1 and standard deviation=1
mu = 1
sigma = 1
real_data = np.random.lognormal(mean = mu, sigma = sigma, size = N)

# Set the data resolution
# Set upper and lower data values as list of k min/max data values [[min_0,max_0],...,[min_k-1,max_k-1]]
bounds = np.array([0.0, 7.0])
# Set number of qubits per data dimension as list of k qubit values[#q_0,...,#q_k-1]
num_qubits = [3]
k = len(num_qubits)
```

Figure C.5: Setting Real Data Distribution and number of qubits

```
# Set number of training epochs
# Note: The algorithm's runtime can be shortened by reducing the number of training epochs.
num_epochs = 20
# Batch size
batch_size = 500

# Initialize qGAN
qgan = QGAN(real_data, bounds, num_qubits, batch_size, num_epochs, snapshot_dir=None)
qgan.seed = 1
# Set quantum instance to run the quantum generator
quantum_instance = QuantumInstance(
    backend=BasicAer.get_backend("statevector_simulator"), seed_transpiler=seed, seed_simulator=seed
)

# Set entangler map
#entangler_map = [[0, 1]]


# Set an initial state for the generator circuit as a uniform distribution
# This corresponds to applying Hadamard gates on all qubits
#init_dist = NormalDistribution(3, mu = 1, sigma = 1, bounds = bounds)
init_dist = QuantumCircuit(sum(num_qubits))
init_dist.h(init_dist.qubits)

# Set the ansatz circuit
ansatz = TwoLocal(int(np.sum(num_qubits)), "ry", "cz", entanglement="circular", reps=1)

# Set generator's initial parameters - in order to reduce the training time and hence the
# total running time for this notebook
#init_params = [3.0, 1.0, 0.6, 1.6]

# You can increase the number of training epochs and use random initial parameters.
init_params = np.random.rand(ansatz.num_parameters_settable) * 2 * np.pi

# Set generator circuit by adding the initial distribution infront of the ansatz
g_circuit = ansatz.compose(init_dist, front=True)

# Set quantum generator
qgan.set_generator(generator_circuit=g_circuit, generator_init_params=init_params)
# The parameters have an order issue that following is a temp. workaround
qgan._generator._free_parameters = sorted(g_circuit.parameters, key=lambda p: p.name)
# Set classical discriminator neural network
discriminator = NumPyDiscriminator(len(num_qubits))
qgan.set_discriminator(discriminator)

ansatz.decompose().draw()
```

Figure C.6: qGAN Initialization

```
# Run qGAN
result = qgan.run(quantum_instance)
```

Figure C.7: Running the qGAN training

```
# Plot the CDF of the resulting distribution against the target distribution, i.e. log-normal
triangular = np.random.triangular(0, 2, 7, size=100000)
triangular = np.round(triangular)
triangular = triangular[triangular <= bounds[1]]
temp = []
for i in range(int(bounds[1] + 1)):
    temp += [np.sum(triangular == i)]
triangular = np.array(temp / sum(temp))

plt.figure(figsize=(6, 5))
plt.title("CDF (Cumulative Distribution Function)")
samples_g, prob_g = qgan.generator.get_output(qgan.quantum_instance, shots=10000)
samples_g = np.array(samples_g)
samples_g = samples_g.flatten()
num_bins = len(prob_g)
plt.bar(samples_g, np.cumsum(prob_g), color="royalblue", width=0.8, label="simulation")
plt.plot(
    np.cumsum(triangular), "-o", label="log-normal", color="deepskyblue", linewidth=4, markersize=12
)
plt.xticks(np.arange(min(samples_g), max(samples_g) + 1, 1.0))
plt.grid()
plt.xlabel("x")
plt.ylabel("p(x)")
plt.legend(loc="best")
plt.show()
```

Figure C.8: Plotting the result CDF

```
# Set upper and lower data values
bounds = np.array([0.0, 15.0])
# Set number of qubits used in the uncertainty model
num_qubits = 4

# Load the trained circuit parameters

#g_init = [0.000, 0.368, 0.191, 0.189, 0.000, 0.123, 0.063, 0.063]
#g_init = [3.27684777, 1.65042226, 2.00504232, 0.70448162, 2.38175055, 4.83904844,
# 2.18861211, 4.76652129]
g_params =  [0.88364636 ,6.13704522,5.55994511, 5.28773269, 0.44667517, 1.45761677 , 1,1,1,1,1,1,1,1]
# Set an initial state for the generator circuit
#init_dist = LogNormalDistribution(num_qubits, mu = 1, sigma = 1, bounds = bounds)
init_dist = UniformDistribution(num_qubits)
#init_dist = NormalDistribution(num_qubits, mu = 1, sigma = 1, bounds = bounds)

# construct the variational form
var_form = TwoLocal(num_qubits, "ry", "cz", entanglement="circular", reps=1)
#var_form.decompose().draw()
# keep a list of the parameters so we can associate them to the list of numerical values
# (otherwise we need a dictionary)
theta = var_form.ordered_parameters
#print(theta)

# compose the generator circuit, this is the circuit loading the uncertainty model
g_circuit = init_dist.compose(var_form)

var_form.decompose().draw()
```

Figure C.9: Setting the Generator parameters for Option Pricing Problem

```
# set the strike price (should be within the low and the high value of the uncertainty)
strike_price = 2

# set the approximation scaling for the payoff function
c_approx = 0.25
```

Figure C.10: Setting the strike price and the approximation scaling for the problem

```
# construct circuit for payoff function
european_call_pricing = EuropeanCallPricing(
    num_qubits,
    strike_price=strike_price,
    rescaling_factor=c_approx,
    bounds=bounds,
    uncertainty_model=uncertainty_model,
)
```

Figure C.11: Defining the European Call Option Problem

```
# Evaluate trained probability distribution
values = [
    bounds[0] + (bounds[1] - bounds[0]) * x / (2**num_qubits - 1) for x in range(2**num_qubits)
]
uncertainty_model = g_circuit.assign_parameters(dict(zip(theta, g_params)))
amplitudes = Statevector.from_instruction(uncertainty_model).data
print(uncertainty_model.decompose().draw())
print(amplitudes)

x = np.array(values)
y = np.abs(amplitudes) ** 2
plt.bar(x, y, width=0.4, label="trained distribution", color="royalblue")
```

Figure C.12: Evaluating the trained probability distribution

```
# Sample from target probability distribution

N = 200000
log_normal = np.random.lognormal(mean = 1, sigma = 1.5, size=N)
log_normal = np.round(log_normal)
log_normal = log_normal[log_normal <= (2**num_qubits-1)]
log_normal_samples = []
for i in range(2**num_qubits):
    log_normal_samples += [np.sum(log_normal == i)]
log_normal_samples = np.array(log_normal_samples / sum(log_normal_samples))

#N = 100000
#uniform = np.random.uniform(low = 0, high = 8, size = N)
#uniform = np.round(uniform)
#uniform = uniform[uniform <= 7]
#uniform_samples = []
#for i in range(8):
#   uniform_samples += [np.sum(uniform == i)]
#uniform_samples = np.array(uniform_samples / sum(uniform_samples))

#N = 10000
#normal = np.random.normal(loc =.75, scale = 4.5, size=N)
#normal = np.round(normal)
#normal = normal[normal <= 7]
#normal_samples = []
#for i in range(8):
#    normal_samples += [np.sum(normal == i)]
#normal_samples = np.array(normal_samples / sum(normal_samples))


# Plot distributions
plt.bar(x, y, width=0.8, label="trained distribution", color="royalblue")
plt.xticks(x, size=15, rotation=90)
plt.yticks(size=15)
plt.grid()
plt.xlabel("Spot Price at Maturity $S_T$ (\$)", size=15)
plt.ylabel("Probability ($\%$)", size=15)
plt.plot(
    (log_normal_samples),
    "-o",
    color="deepskyblue",
    label="target distribution",
    linewidth=3,
    markersize=10,
)
plt.legend(loc="best")
plt.show()
```

Figure C.13: Plotting the target and trained PDF

```
# Evaluate payoff for different distributions
payoff = np.array([0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13])
ep = np.dot(log_normal_samples, payoff)
print("Analytically calculated expected payoff w.r.t. the target distribution:  %.4f" % ep)
ep_trained = np.dot(y, payoff)
print("Analytically calculated expected payoff w.r.t. the trained distribution: %.4f" % ep_trained)

# Plot exact payoff function (evaluated on the grid of the trained uncertainty model)
x = np.array(values)
y_strike = np.maximum(0, x-strike_price)
plt.plot(x, y_strike, "ro-")
plt.grid()
plt.title("Payoff Function", size=15)
plt.xlabel("Spot Price", size=15)
plt.ylabel("Payoff", size=15)
plt.xticks(x, size=15, rotation=90)
plt.yticks(size=15)
plt.show()
```

Figure C.14: Evaluating payoff

```
# set target precision and confidence level
epsilon = 0.01
alpha = 0.05

qi = QuantumInstance(Aer.get_backend("aer_simulator"), shots=500)
problem = european_call_pricing.to_estimation_problem()
# construct amplitude estimation
ae = IterativeAmplitudeEstimation(epsilon, alpha=alpha, quantum_instance=qi)
```

Figure C.15: Setting parameters for QAE and defining the problem

```
result = ae.estimate(problem)

conf_int = np.array(result.confidence_interval_processed)
print("Exact value:        \t%.4f" % ep_trained)
print("Estimated value:    \t%.4f" % (result.estimation_processed))
print("Confidence interval:\t[%.4f, %.4f]" % tuple(conf_int))
```

Figure C.16: Running QAE