Technical University of Crete

Department of

Electrical and Computer Engineering

Analog and Digital Quantum Neural Networks: Basic Concepts and Applications

by

Antonios Kastellakis

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DIPLOMA OF

ELECTRICAL AND COMPUTER ENGINEERING

October 15, 2022

THESIS COMMITTEE

Associate Professor Dimitris G. Angelakis, *Thesis Supervisor*

Professor Minos Garofalakis

Professor, Vice Rector, Michalis Zervakis

Abstract

In the scope of this thesis, we investigate how the rise of quantum computers can offer a new, potentially more powerful, way of machine learning. The study begins by defining the framework of quantum computation. This includes the building blocks of a quantum computer, such as the quantum bits and gates, but also the postulates of quantum mechanics, that determine their behaviour. Then we move the discussion to the field of machine learning, where we do a gentle introduction to the basic machine learning methods with the focal point being neural networks as generative models. To this end, we introduce a special type of energy based neural network, the Restricted Boltzmann machine (RBM). We discuss not only the theoretical background of the RBM, but also present an example, by coding and training on the MNIST data set of handwritten digits. Next, we examine Quantum Machine Learning (QML), the union of quantum computation with machine learning. There are two approaches of QML, the quantum advantage QML algorithms that have proven speed-ups over their classical counterparts but require fault-tolerant quantum devices, and hybrid classical-quantum variational models that can be executed on the Noisy Intermediate Scale Quantum (NISQ) devices of today. The QNNs models we implement for this study belong to the latter case. We present two QNN approaches, the digital approach that considers the Quantum Circuit Born Machines (QCBM) and an analog approach, which refers to qunatum information processing with analog quantum systems. These models are quantum analogues of classical neural networks that can be trained, using both classical and quantum resources, to learn target probability distributions. We demonstrate how they learn from classical data and at the end, we attempt to compare their capabilities their capabilities of learning the same dataset. Our novel algorithms have been implemented on classical simulators as well as real quantum hardware available in cloud from IBM.

Acknowledgements

Words cannot express my gratitude to my thesis supervisor Associate Prof. Dimitris G. Angelakis for his patience, guidance, and support. I have benefited greatly from his wealth of knowledge. Thank you for reading my numerous revisions and helped make some sense of the confusion. I am fortunate to have been a member of the Angelakis research group in Centre for Quantum Technologies at the National University of Singapore and I would like to thank the team for the much interesting and helpful talks and presentations. Especially, I am gratefully indebted to Gan Beng Yee for his very valuable comments and insights on this thesis that helped me in surpassing many road blocks I encountered along the way. My sincere thanks also go to the members of my thesis committee Professor Minos Garofalakis and Professor, Vice Rector, Michalis Zervakis for their patience and understanding for my effort that went into the production of this paper. Finally, I must express my very profound gratitude to my family and friends for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you.

Contents

List of Figures

Chapter 1

Introduction

> "Nature is not classical, dammit, and if you want to make a simulation of nature, you'd better make it quantum mechanical."
>
> *Richard P. Feynman*

Since the dawn of humanity, it was evident that people have an innate need to have explanations for the world around us. In the ancient days, phenomena such as droughts, diseases and other natural disasters were explained by myths. People usually attributed these natural disasters to the bad mood of the gods, and so, they would gave offerings and prayed to appease them. With the passage of time humans slowly started poking around and observing patterns and regularities. This really started to catch on during the enlightenment, were the modern definition of science was born. It became apparent that the way to systematically study the structure and behaviour of the natural world was through observation and experiment.

During the 18th and 19th centuries, a broad range of mathematical methods were discovered, which gave people the tools to analyse data and draw insight from them. But as the amount of data that was collected increased, it was becoming harder and harder to crunch the numbers, analyse them and create models to generalize the findings as it was all done by hand. Then, in midst of the 20th century, a miracle discovery happened. The invention of the computer. The computer swooped in and made things easier as now we could process, analyse and discover patterns in data much faster and much more efficiently.

For years, computers evolved to meet the increasing numbers of collected data, which they created the need for more computational power. However, nowadays we are facing yet another challenge. The amount of collected data is growing fast, while the advancements in computers are slowing down. From their invention until now, computers evolved according to Moore's law. Moore's law indicates that the number of transistors in a dense integrated system, doubles every two years, which

results in an increase to the computational power. But there's a pitfall that will will inevitably bring this evolution to an end. The issue is that we are running out of space to store transistors and as we continue to miniaturize the chips the size of the transistors are getting closer to the atomic size. This is a tipping point as quantum phenomena will start to kick in and the transistor won't be able to function as is.

Naturally, people started looking for different computational paradigms, with one of them being to utilize for our gain the same quantum phenomena that are causing us problems. Quantum computing attempts to harness the bizarre phenomena of quantum mechanics to perform computations that are faster and more efficient than on ordinary computers.

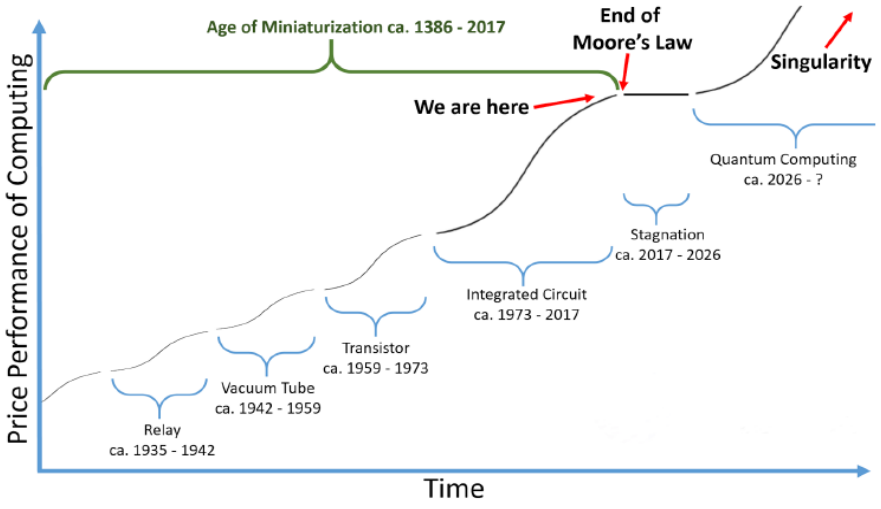


Figure 1.1: The evolution of computers.

But our interest in quantum computers does not only stems from the end of Moore's law. The entire field of quantum computation began back in the 80's, long before the miniaturization of the transistor posed any problems. It is no secret that conventional computers, due to their binary nature, lack the ability to practically simulate nature, regardless of the available resources. Roughly speaking, it is hard, time-wise, for a computer to keep up with problems that grow exponentially. A question, then, arises: If we can't use conventional computers, is there a way to simulate such problems? As Richard P. Feynman stated in 1981, when he alluded to a *quantum computer model* [1], should a computer be "build of quantum components" maybe it wouldn't fall behind.

On a general scope, a 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 assumption sparked the interest of the public, as it implies that it is possible to get a computational speedup, should we run a problem on a quantum computer instead of a classical.

Figure 1.2: Along with his work in theoretical physics, Feynman has been credited with pioneering the field of quantum computing.

In recent years, research in the field has intensified with investments coming from public and private sectors. Potential applications range from finance, to business, to system security and artificial intelligence. One of the fields that shows huge potential of exploiting the "quantum advantage" is machine learning. One may wonder though, why that is the case. The answer to this is pretty simple. Quantum mechanics are known for producing atypical patterns in data. Classical machine learning methods have the capability to not only recognize patterns in data, but also produce data that possess the same patterns. Thus, there is hope that if a quantum computer can produce patterns that are not tractable with a classical computer, then maybe quantum computers can also recognise patterns that a classical computer can not.

Currently, research in the *Quantum Machine Learning (QML)* field revolves around two approaches. The first approach research quantum algorithms that require a fully functional *fault-tolerant* quantum computer with millions of qubits. The problem with this approach is that we don't have quantum computers are still at their "infancy". The most advanced quantum devices today have a few dozen noisy and error-prone qubits. The second approach is a more pragmatic one, as the research is focused on utilizing the devices that we have to solve real problems. This is called *Noisy Intermediate Scale Quantum (NISQ)*.

Thesis Outline

This thesis pertains to the applications of quantum computation in machine learning and neural networks in NISQ devices. From the very beginning, we would like to treat this manuscript as a journey of quantum computation and neural networks. Thus, we start from the beginning and build towards our goal. A brief outline of this thesis is provided below.

Chapter 2. Quantum Computation. This thesis starts by introducing the basic principles of *quantum computation* and providing the necessary mathematical notation and background of the field.

Chapter 3. Machine Learning. The third chapter does a brief introduction to the basic methods of machine learning and focuses on generative generative modeling with neural networks. For this purpose we introduce a special type of energy based neural network called *restricted Boltzmann machine (RBM).*

Chapter 4. Quantum Machine Learning. This chapter is the main chapter of this thesis and regards QML, with *quantum neural networks (QNN)* on NISQ devices being the focal point. Specifically, we explain the basic concepts and applications of *digital* and *analog* quantum neural networks. The first approach, which we will call *digital*, views a parameterized quantum circuit as a neural network. The second approach, which we call *analog*, tunes the physical control parameters of the quantum device, such as the electromagnetic field strength or a laser pulse frequency, to train the system. We attempt a comparison in the training accuracy in terms of simulations and noisy real quantum processors.

Chapter 2

A Basic Introduction to Quantum Computation

Computers today are based on the work of the great mathematician *Alan Turing*. Turing developed an abstract model of computation, which is now called *Turing machine*, that aimed at resolving whether or not there are *computational problems* that are considered *intractable*. An intractable, or "unsolvable", computational problem is one that cannot be solved by an algorithm.

Since the days of Turing, computational problems has been studied extensively by theoretical computer science. There is an entire field, called *computational complexity*, that attempts to classify the problems based on the resources they require to solve them. Unfortunately, the reality of the computation has showed that even among the "solvable" problems, there are the ones that are solved *inefficiently*. But what does this actually mean? Simply put, an efficient algorithm is one that requires polynomial amount of computation time to solve a problem.

However, many problems that are found in nature are of higher complexity and as a result cannot be solved by a classical computer, no matter the resources available. It is evident that the problem lies with the binary nature of information in a typical computer. Thus, it comes as no surprise that a possible solution, that may offer computational speed-ups, is to move to a different computing paradigm.

One such paradigm is *quantum computing*, which utilizes the strange laws of quantum mechanics to perform computations, instead of the classical mechanics that are used in a typical microelectronic chip. A computer that uses the phenomena and laws of quantum mechanics can perform computations faster and more efficiently than an ordinary computer. This is due to the unconventional and weird way that information is being processed and stored in a quantum computer.

2.1 What is a qubit?

The elementary unit of information in a quantum computer is called a *quantum bit*, or *qubit* for short. Similar to a classical bit, that is either 0 or 1, a qubit has also two basic states, the $|0\rangle$ and $|1\rangle$. The notation '$|\cdot\rangle$' is called *ket*, and was introduced by Paul Dirac in 1958 to describe quantum states.

In classical systems, the space of states is a set that contains all the possible states. For example, the space of states for a bit is $\{0, 1\}$. In quantum systems, however, the space of states is a *vector space*.

A quantum state is represented by a vector in a complex vector space called *Hilbert space* and is denoted by $\mathcal{H}$. We don't need to give the mathematical definition for a Hilbert space here, we can just think of it as a generalization of a *Euclidean vector space* that may have either a finite or infinite number of dimensions.

In quantum mechanics, the vectors that compose the Hilbert space are represented by *kets*. The simplest Hilbert space we can study regards the two-level quantum systems, where the vector states $|0\rangle$ and $|1\rangle$ form an *orthonomal basis* for this vector space and are defined as

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad , \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

To make it more concrete we need to introduce some important definitions from linear algebra.

Definition 2.1.1 (*Linear combination*)**.** A vector $|u\rangle \in \mathbb{C}^n$ is a *linear combination* of vectors $|u_1\rangle$, $|u_2\rangle$, $\cdots$, $|u_n\rangle$, if $|u\rangle$ can be expressed as

$$|u\rangle = \sum_i^n c_i \, |u_i\rangle \tag{2.1}$$

where $c_i \in \mathbb{C}$

Definition 2.1.2 (*Linear Independence*)**.** A set of non zero vectors $|u_1\rangle$, $|u_2\rangle$, $\cdots$, $|u_n\rangle \in \mathbb{C}^n$ are *linearly independent* if

$$a_1 \, |u_1\rangle + a_2 \, |u_2\rangle + \cdots + a_N \, |u_n\rangle = 0 \;\Leftrightarrow\; a_i = 0 \quad \forall \, i = 1, 2, \cdots, n \tag{2.2}$$

Definition 2.1.3 (*Spanning set*)**.** A *spanning set* for a vector space is a set of vectors $|u_1\rangle$, $|u_2\rangle$, $\cdots$, $|u_n\rangle \in \mathbb{C}^n$, for which any other vector $|u\rangle \in \mathbb{C}^n$ can be written as a linear combination $|u\rangle = \sum_i c_i \, |u_i\rangle$ of that set of vectors.

Definition 2.1.4 (Basis)**.** A *basis* is a set of vectors that are a *spanning set* and *linearly independent*.

Definition 2.1.5 (Orthonomal basis)**.** An *orthonomal basis* is a basis whose vectors are all unit vectors and orthogonal to each other.

At this point it is easy to see that the vectors $|0\rangle$,$|1\rangle$ constitute an orthonomal basis in $\mathbb{C}^2$. This is the most frequent used basis for quantum computation and is called the *computational basis*. Some careful readers may notice that a spanning set is not unique for a given vector space ,and therefore, a basis is not unique either. In reality, there are many basis used in quantum computation and later on we will see how we can transition from one basis to another.

The main difference between a qubit and a classical bit, is that a qubit can be in more than one state at a time. This strange phenomenon is found only in quantum mechanics and is called *superposition*. In a mathematical context, superposition translates to a linear combination of states:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix}. \tag{2.3}$$

The components α and β are complex numbers, and while it is not yet clear why, they are called *probability amplitudes*. In fact, α and β by themselves, have no experimental meaning. However, their magnitudes do. In particular, when we *measure* a qubit, we either find it in the state $|0\rangle$ with probability $|\alpha|^2$, or in the state $|1\rangle$ with probability $|\beta|^2$. This is called the *Born rule* and says a lot about the nature of quantum mechanics. Before measurement, all we have is the state vector $|\psi\rangle$, which represents the possible states. The act of *measurement* in computational basis collapses the superposition and forces the qubit to one of the potential states with probabilities,

$$P_{|0\rangle} = |\alpha|^2 = \alpha^*\alpha \quad , \quad P_{|1\rangle} = |\beta|^2 = \beta^*\beta. \tag{2.4}$$

There is an important point hidden here that we need to pay attention to. Since we talk about probabilities, it is important that the total probability sums to unity. Hence, we must have

$$|\alpha|^2 + |\beta|^2 = 1. \tag{2.5}$$

More generally, an arbitrary quantum state in $\mathbb{C}^n$ can be written as

$$|\psi\rangle = \sum_i^n c_i\,|u_i\rangle \tag{2.6}$$

where $|u_1\rangle$, $|u_2\rangle$, $\cdots$, $|u_n\rangle$ form an orthonomal basis in $\mathbb{C}^n$. The corresponding probabilities make up a probability distribution, and as such, they must sum to unity $\sum_i^n |c_i|^2 = c_i^* c_i = 1$.

A geometric interpretation of this principle, is that the state of a quantum system is represented by a *normalized* vector in a vector space of states.

Definition 2.1.6 (*Unit vector*)**.** A *unit vector* or normalized vector, is a vector $|u\rangle$ in $\mathbb{C}^n$ that satisfies $\||u\rangle\| = 1$.

This geometric interpretation helps in visualizing such systems, but more on that later. As we have seen, the ket vectors are complex vectors, and as you may expect, they have a *complex conjugate* form. For every ket $|\psi\rangle$, there is a "*bra*" vector in the complex conjugate space, that is denoted by $\langle\psi|$. So, if a ket is represented by a column vector,

$$|\psi\rangle \mathrel{\widehat{=}} \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

then the corresponding bra is a row vector

$$\langle\psi| \mathrel{\widehat{=}} \begin{pmatrix} \alpha^* & \beta^* \end{pmatrix}.$$

This notation will help us define inner products in the form of *bra-kets*, $\langle\phi|\psi\rangle$, which play an essential part in quantum mechanics.

2.2 Visualizing a qubit

Besides their bizarre nature, qubits can actually be realized by various physical systems. We will reference two systems that also help with the visualisation. The first representation of a qubit that we will discuss considers quantum systems with two discrete energy levels. One such example is an electron that orbits an atom, as shown in the figure below. In the atomic model there are a lots discrete energy orbitals that an electron can occupy. To define a qubit, one just need to utilise the lowest two orbitals, the *ground state* and the *first excited state* of the orbitals. The *ground state* is the energy state that would be considered normal for an electron and we use it to represent the $|0\rangle$ state. Respectively, the higher energy states are called "*excited states*" and we use the first excited state to represent the $|1\rangle$ state. Should the electron absorb, or emit, energy that is equal to the energy gap between two energy states, it is can move from the state $|0\rangle$ to $|1\rangle$ and vice-versa. The simplest example we can discuss, that helps with the visualization, is the hydrogen atom, that has a single electron orbiting it's nucleus.

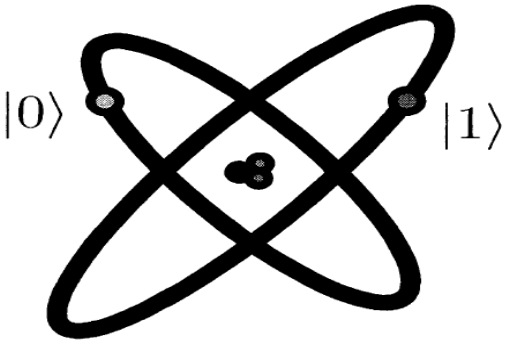


Figure 2.1: Qubit implementation by two electron orbitals in an atom.

The second and most common quantum system used as a qubit is the *isolated quantum spin*. The concept of spin is derived from particle physics and is one of the attributes that are attached to elementary particles. Naivly, we can picture the quantum state of the spin as an *arrow* that points to some direction. This is where the geometric interpretation we mentioned earlier fits perfectly. We can rewrite Equation 1.1 to represent a point in a 3-d unit sphere,

$$|\psi\rangle = cos\frac{\theta}{2}|0\rangle + e^{i\phi}sin\frac{\theta}{2}|1\rangle \tag{2.7}$$

where $0 \leq \theta \leq \pi$ and $0 \leq \phi \leq 2\pi$. This sphere is called *Bloch sphere*, after the physicist Felix Bloch, and serves as an excellent way of visualizing a quantum state as shown in the figure 1.2 below.

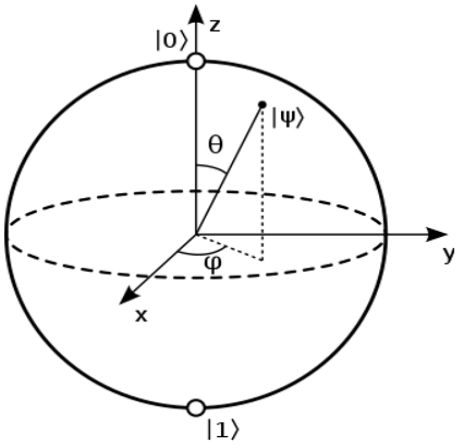


Figure 2.2: Bloch sphere representation of a quantum state $|\psi\rangle$.

Later on we will see that every action on a single qubit can be interpreted as a rotation on the Bloch sphere. However, there is a downside in the Bloch sphere picture, since it cannot be expanded to multiple qubits.

2.3 Inner products and orthonomal bases

Inner products play an important role in defining vector spaces. We are already familiar with the concept of dot product from linear algebra. The analogous operation in quantum computation is defined in the form of product of a bra with a ket

$$\langle\psi|\phi\rangle = \begin{pmatrix}\psi_1^* & \psi_2^*\end{pmatrix}\begin{pmatrix}\phi_1 \\ \phi_2\end{pmatrix} = \psi_1^*\phi_1 + \psi_2^*\phi_2 = \sum_i \psi_i^*\phi_i. \tag{2.8}$$

The result of an inner product is a complex number, and as with vectors in real spaces, it shows us the *vector projection* of one vector over another vector.

Apart from their definition, inner products should also satisfy some properties,

$$\textbf{Linearity} : \langle\psi| \left(|\phi\rangle + |\chi\rangle\right) = \langle\psi|\phi\rangle + \langle\psi|\chi\rangle \tag{2.9}$$

$$\textbf{complex conjugation} : \langle\psi|\phi\rangle^* = \langle\phi|\psi\rangle \tag{2.10}$$

However, the inner product between complex vectors is not commutative in general, as opposed to the dot product between two vectors in an Euclidean space.

As we previously mentioned, qubits live in Hilbert spaces, where the basis vectors, that span the space, are *orthonomal*. A set of vectors is said to be orthonormal if they are all normalized, and each pair of vectors in the set is orthogonal. This fits perfectly with the concept of the two mutually exclusive states of a qubit, $|0\rangle$ and $|1\rangle$.

The demand to be normalized practically means that

$$\langle 0|0\rangle = \begin{pmatrix}1 & 0\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} = 1 \quad \text{and} \quad \langle 1|1\rangle = \begin{pmatrix}0 & 1\end{pmatrix}\begin{pmatrix}0\\1\end{pmatrix} = 1. \tag{2.11}$$

While the demand to be orthogonal means

$$\langle 0|1\rangle = \begin{pmatrix}1 & 0\end{pmatrix}\begin{pmatrix}0\\1\end{pmatrix} = 0 \quad \text{and} \quad \langle 1|0\rangle = \begin{pmatrix}0 & 1\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix} = 0. \tag{2.12}$$

We can generalize the concept for a $\mathbb{C}^n$ space with an orthonomal basis, assuming n is the dimensions of the space. The basis vector are labeled as $|i\rangle$, where $i = 0, 1, \cdots, n-1$. In that case, we have

$$\langle i|j\rangle = 0 \quad , \quad \text{if i} \neq \text{j} \tag{2.13}$$

$$\langle i|j\rangle = 1 \quad , \quad \text{if i} = \text{j} \tag{2.14}$$

In a more elegant fashion, we can sum up these two equations into one as $\langle i|j\rangle = \delta_{ij}$, where δ_{ij} is the *Kronecker delta*.

2.4 Postulates of quantum mechanics

To better understand qubits and how we interact with them, we first need to understand the rules that govern the quantum world. Conversely to classical physics, we don't possess the intuition to understand and visualize quantum. Nevertheless, we overcame this lack of senses by turning to abstract mathematics.

Postulate 1.

The state of a physical system is represented by a complex vector $|\psi\rangle$ in the Hilbert space. This vector contains all the information, that are accessible to us, about the system.

Postulate 2.

The physical observables are described by *linear operators*, and specifically, *Hermitian operators*. The observables are quantities we can measure. For example, observables in quantum mechanics are things like the position, energy and angular momentum of a particle. But what are *Hermitian operators*?

Definition 2.4.1 (*Linear operators*)**.** A *linear operator* M is a linear transformation $M : V \to W$, between two vectors spaces V and W. Suppose $|\psi\rangle$ is a state vector in $\mathbb{C}^n$, then,

$$M\,|\psi\rangle = M\left(\sum_i^n c_i\,|u_i\rangle\right) = \sum_i^n c_i\; M\,|u_i\rangle\,, \tag{2.15}$$

where $c_i \in \mathbb{C}$.

Definition 2.4.2 (*Hermitian conjugate*)**.** The *Hermitian conjugate*, or *adjoint* (†), is the complex conjugate of a transposed matrix

$$M^\dagger = [M^*]^T \tag{2.16}$$

Definition 2.4.3 (*Hermitian Operator*)**.** *Hermitian Operators* are linear operators that are equal to their own Hermitian conjugate

$$M^\dagger = M \tag{2.17}$$

This postulate brings us back to the conversation about spin qubits. In the context of quantum mechanics, the quantum state of the spin is represented as a 3 dimensional vector in the Bloch sphere with basis $\hat{\sigma}_x, \hat{\sigma}_y$, and $\hat{\sigma}_z$. These operators are the *observable components of the spin* and thus, are described by *Hermitian operators*. Measuring the spin in one of these directions will yield either 1 or -1.

The measurement is performed by an apparatus that interacts with the system. For simplicity we can think of the apparatus as a black box, which we can orient along each axis to measure the respective spin component.

Postulate 3.

Should we measure an observable quantity, the possible measurement results are the eigenvalues λ_i of the operator that describes the observable. Moreover, the state for which we measure with certainty λ_i is the corresponding eigenstate $|\lambda_i\rangle$.

Definition 2.4.4 (*Eigenvalues and eigenvectors*)**.** An *eigenvector* $|\lambda_i\rangle \in \mathbb{C}^n$ of an operator $M \in \mathbb{C}^{n\times n}$, is a non-zero vector that satisfies

$$M\,|\lambda_i\rangle = \lambda_i\,|\lambda_i\rangle \tag{2.18}$$

where λ_i is a complex number that called *eigenvalue*. The number of eigenvalues depends to the dimensions of the operator. So, an operator of $n \times n$ dimensions will have n *eigenvectors* and n corresponding *eigenvalues*. Finding these *eigenvectors* and *eigenvalues* requires solving the so-called *characteristic equation*

$$det|M - \lambda I| = 0 \tag{2.19}$$

where I is the *identity matrix*.

Definition 2.4.5 (*Identity matrix*)**.** The *identity matrix* is a square matrix that when it acts on a vector $|\psi\rangle \in \mathbb{C}^n$, it leaves the vector it unaffected

$$I\,|\psi\rangle = |\psi\rangle \tag{2.20}$$

The matrix representation of the *identity matrix* is

$$I_{n\times n} \widehat{=} \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix} \tag{2.21}$$

Definition 2.4.6 (*Spectral decomposition*)**.** Any *normal operator* $M \in \mathbb{C}^{n\times n}$ has a *diagonal representation* in the following form

$$M = \sum_i \lambda_i\,|\lambda_i\rangle\,\langle\lambda_i| \tag{2.22}$$

where the eigenvectors $|\lambda_i\rangle$ form an orthonomal set and λ_i are their corresponding eigenvalues. In matrix form we can write this as

$$M = VDV^\dagger \tag{2.23}$$

where V is a matrix that has the eigenvectors as columns and D is a diagonal matrix with the eigenvalues as its entries.

Definition 2.4.7 (*Normal operator*)**.** A linear operator $M \in \mathbb{C}^{n\times n}$ is said to be normal if it commutes with its Hermitian conjugate

$$MM^\dagger = M^\dagger M \tag{2.24}$$

Definition 2.4.8 (*Commutator*)**.** The *commutator* between two operators A and B is defined as

$$[A, B] = AB - BA \tag{2.25}$$

If $[A, B] = 0$ we say that the two operators *commute*

As we mentioned earlier, when an observable component of the spin is measured, the measuring apparatus will yield nothing else but ± 1. Postulate 3 gives a new meaning to these measurement results. It implies that result of a measurement is always one of the eigenvalues of the corresponding operator. Therefore, the eigenvalues of the spin operators are ± 1.

Postulate 4.
Suppose that the state of a quantum system is $|\psi\rangle$. If we measure the observable M, we will obtain λ_i with a probability

$$pr(\lambda_i) = |\langle\psi|\lambda_i\rangle|^2 = \langle\psi|\lambda_i\rangle \langle\lambda_i|\psi\rangle \tag{2.26}$$

where λ_i is an eigenvalue and $|\lambda_i\rangle$ an eigenvector of the Hermitian operator that describes M. Since the operator is Hermitian, the possible measurement are *unambiguously distinct*. In general, we define the product $|\lambda_i\rangle \langle\lambda_i|$ as a *Projective Operator* P_i.

Definition 2.4.9. A *projective operator* P_i, also called *Von Neumann operator*, is a Hermitian operator that acts on the state space of the system $|\psi\rangle$, and affects it *irreversibly*. An operator is projective if

$$P^2 = P \tag{2.27}$$

Projective operators are also mutually orthogonal and they form a *complete set*. Meaning

$$\sum_i P_i = I \tag{2.28}$$

This postulate binds in a beautiful way everything we mentioned on quantum states and measurement. Everything we can learn about a quantum state is contained within a vector $|\psi\rangle$. Nevertheless, we are unable to access this information without interacting and permanently altering the state of the system. Thus, any kind of computation should occur before observing the system.

In the case of spin qubits specifically, the information we can observe about the system is the three components of the spin, $\sigma_z, \sigma_x, \sigma_y$. These components, as well as any observable quantity a quantum system may have, are described by *Hermitian operators*. Hermitian operators can be decomposed to a set of mutually orthogonal eigenvectors, where each eigenvector is associated with a different real number, called eigenvalue.

Every time we attempt to observe one of the components of the spin, we get as a measurement result one of the eigenvalues of the respective operator. Since each eigenvalue has a unique corresponding eigenvector, we can easily deduce the state the system was prior to observing it.

However, measurement of a specific observable can distinguish without a doubt only the eigenvector states. So is it possible to observe any state? The answer here is yes! The eigenvector of each operator constitute an orthonomal basis and hence, we can write each state as a linear combination of the eigenvectors we intent to measure. It turns out that, the *computational basis* $\{|0\rangle, |1\rangle\}$ that we defined earlier on, is simply the basis formed by the eigenvectors of the σ_z operator.

Postulate 5.
The time evolution of a closed quantum system is governed by the *Schrödinger equation*

$$i\hbar\frac{\partial}{\partial t}|\psi(t)\rangle = \hat{H}|\psi(t)\rangle \tag{2.29}$$

where $\hbar$ is the *Plank's constant* and $\hat{H}$ is called the *Hamiltonian* of the system, with which we will become very familiar later on. The *Hamiltonian* is a Hermitian operator that describes the energy structure of the system. Its eigenvalues are the values that would result from measuring the energy of a quantum system.

The solution of the *Schrödinger equation*, with a time independent Hamiltonian, describes how the quantum states of two different times t_1,t_0 are connected

$$|\psi(t_1)\rangle = e^{-iHt/\hbar}|\psi(t_0)\rangle \tag{2.30}$$

There is something really interesting and important to mention about the above equation. The operator that is responsible for the evolution of an isolated quantum system is *unitary*.

Definition 2.4.10. *Unitary operator* A linear operator $U \in \mathbb{C}^{n\times n}$ is said to be unitary if it satisfies

$$U^\dagger U = UU^\dagger = I \tag{2.31}$$

where $U^\dagger$ is the *adjoint* or *Hermitian conjugate* of U and I is the *identity operator*. A *unitary operator* can be expressed in the following form

$$U = e^{iK} \tag{2.32}$$

where K is some *Hermitian operator*.

This conclusion is important because it implies that the action of operators on quantum states is in fact *unitary transformations*. We will see in the next section how we utilize this to realize quantum circuits.

2.5 Single qubit operations

Classical computers consists of microelectronic circuits that work on *Boolean logic*. These circuits are composed of wires and *logic gates*. The wires transfer the electric signal from one gate to another and the gates process the information that arrives to them based on a *truth table*. The logic gates are actual physical systems, made of silicon, so is the information they process.

Similar to a classical computer, the building blocks of a quantum computers are wires and *elementary quantum gates*. However, there are some main differences. Conversely to a logic gate, a quantum gate is described by *unitary operators*.

One may wonder at this point, why a quantum gate needs to be a unitary operator. Remember that we defined a quantum state as a unit vector on the Hilbert space. Quantum gates act on qubits by rotating them. However, this rotation should not change the length of the qubit. That is, for a qubit $\alpha|0\rangle+\beta|1\rangle$, the normalization condition $|\alpha|^2 + |\beta|^2 = 1$, must hold after the action of the gate.

Moreover, the time evolution of closed quantum systems is *reversible*. This implies that if the action of a quantum gate G is rotating a qubit from a state $|s_1\rangle$ to a state $|s_2\rangle$, then by applying the inverse of the same gate, we rotate the qubit back from $|s_2\rangle$ to $|s_1\rangle$.

These two properties of quantum gates are perfectly encapsulated by unitary transformations, and hence, every unitary matrix can be used as a quantum gate.

2.5.1 Spin Operators

In spin qubits, we saw that we can express each quantum state as a sum of three observable components, $\sigma_z, \sigma_x, \sigma_y$. Now it is time to introduce the Hermitian operators that they are associated with. These operators are very famous and are called *Pauli operators*, after the physicist Wolfgang Pauli that discovered them. In literature, they can be referenced by are various notations

$$X \equiv \sigma_x \equiv \sigma_1 \equiv \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$Y \equiv \sigma_y \equiv \sigma_2 \equiv \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

$$Z \equiv \sigma_z \equiv \sigma_3 \equiv \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

The Pauli operators, as well as all the gates we will see later, have a circuit symbol. This is a way to compress and visualize the operations we perform on qubits. The symbols of the Pauli gates and their behavior is summed up on the following table

$$\alpha|0\rangle + \beta|1\rangle \quad —\boxed{X}— \quad \beta|0\rangle + \alpha|1\rangle$$

$$\alpha|0\rangle + \beta|1\rangle \quad —\boxed{Y}— \quad -i\beta|0\rangle + i\alpha|1\rangle$$

$$\alpha|0\rangle + \beta|1\rangle \quad —\boxed{Z}— \quad \alpha|0\rangle - \beta|1\rangle$$

Interestingly enough, the *Pauli matrices* can generate arbitrary rotations around the three coordinates axis *x,y* and *z*. To generate these rotation matrices, which we will call R_x, R_y and R_z, we need to exponentiate the matrices in the following way

$$R_X = e^{-i\theta X/2} = cos\frac{\theta}{2}\, I - sin\frac{\theta}{2}\, X \widehat{=} \begin{pmatrix} cos\frac{\theta}{2} & -isin\frac{\theta}{2} \\ -isin\frac{\theta}{2} & cos\frac{\theta}{2} \end{pmatrix} \tag{2.33}$$

$$R_Y = e^{-i\theta Y/2} = cos\frac{\theta}{2}\, I - sin\frac{\theta}{2}\, Y \widehat{=} \begin{pmatrix} cos\frac{\theta}{2} & -sin\frac{\theta}{2} \\ sin\frac{\theta}{2} & cos\frac{\theta}{2} \end{pmatrix} \tag{2.34}$$

$$R_Z = e^{-i\theta Z/2} = cos\frac{\theta}{2}\, I - sin\frac{\theta}{2}\, Z \widehat{=} \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix} \tag{2.35}$$

This may seem strange at first glance. However, there is a way to simplify things by defining functions for operators. These functions can be defined for normal matrices that have a spectral decomposition.

Definition 2.5.1 (*functions for operators*)**.** Let M be a normal operator that can be decomposed as $M = \sum_i \lambda_i |\lambda_i\rangle \langle\lambda_i|$. Then we can define

$$f(M) = \sum_i f(\lambda_i) |\lambda_i\rangle \langle\lambda_i| . \tag{2.36}$$

An example always helps at this point. Let's prove that R_Z has indeed the matrix representation that we claimed. Recall that the eigenvectors of σ_z are the basis vectors of the computational basis and that the corresponding eigenvalues are 1 and -1, respectively.

$$e^{-i\theta Z/2} = e^{-i\theta/2} |0\rangle \langle 0| + e^{i\theta/2} |1\rangle \langle 1| = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}$$

2.5.2 Hadamard gate

Another gate that is essential to quantum computation is the *Hadamard gate*. This is the gate that is responsible for creating superposition, given one of the basis states. The action of this gate can be summarized by the mapping

$$|0\rangle \mapsto \frac{|0\rangle + |1\rangle}{\sqrt{2}} \quad , \quad |1\rangle \mapsto \frac{|0\rangle - |1\rangle}{\sqrt{2}}. \tag{2.37}$$

These states are often denoted in bibliography as $|+\rangle$ and $|-\rangle$, respectively. These states are of particular interest because upon measurement in computational basis the qubit will be found 50% of the time in the state $|0\rangle$ and 50% of the time in the state $|1\rangle$.

Superposition is the first quantum mechanical phenomenon that quantum computation utilizes to it's advantage. Naively, we can think superposition as being at both states at the same time, and therefore, perform calculations for both states at the same time, which wouldn't be possible in a conventional computer.

The matrix representation, notation and circuit symbol is summarized below

$$H \equiv \quad -\boxed{H}- \quad \widehat{=} \quad \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{2.38}$$

2.5.3 Arbitrary single qubit gates

There are many more quantum gates that we could present here. However, as we seen previously, there are infinite quantum gates. All of them perform some type rotation on the Bloch sphere by changing the angles θ and ϕ that can be seen in the Fig(2.2).

It is easy to show that any single qubit quantum gate can be decomposed as

$$U = e^{i\alpha}\begin{pmatrix} e^{-i\beta/2} & 0 \\ 0 & e^{i\beta/2} \end{pmatrix}\begin{pmatrix} cos\frac{\gamma}{2} & -sin\frac{\gamma}{2} \\ sin\frac{\gamma}{2} & cos\frac{\gamma}{2} \end{pmatrix}\begin{pmatrix} e^{-i\delta/2} & 0 \\ 0 & e^{i\delta/2} \end{pmatrix} \tag{2.39}$$

The components $\alpha, \beta, \gamma, \delta$ are real numbers. Determining these components can generate an exact approximation of any unitary matrix, and hence, any quantum gate.

2.5.4 Changing Bases

All the definitions of gates we saw was based on the computational basis $\{|0\rangle, |1\rangle\}$. If we want to work on a different basis, we need to calculate the matrix representation ourselves. We use unitary transformations to find the matrix representation of an operator in a different basis.

Suppose that we want to shift from the orthonomal basis $\{|\psi_1\rangle, |\psi_2\rangle\} \in \mathbb{C}^2$ to the orthonomal basis $\{|\phi_1\rangle, |\phi_2\rangle\} \in \mathbb{C}^2$. The first step is to calculate the unitary matrix that is responsible for the change of basis

$$U = \begin{pmatrix} \langle\phi_1|\psi_1\rangle & \langle\phi_1|\psi_2\rangle \\ \langle\phi_2|\psi_1\rangle & \langle\phi_2|\psi_2\rangle \end{pmatrix} \tag{2.40}$$

Subsequently, we transform the operator M from the $\{|\psi_1\rangle, |\psi_2\rangle\}$ basis to the $\{|\phi_1\rangle, |\phi_2\rangle\}$ as follows

$$M' = UMU^{\dagger} \tag{2.41}$$

where M' is the matrix representation of M in the $\{|\phi_1\rangle, |\phi_2\rangle\}$ basis.

Besides operators, we also need to transform the state vectors. This is done by simply acting matrix to the state. Let $|u\rangle$ be a state vector in $\{|\psi_1\rangle, |\psi_2\rangle\}$, then

$$|u'\rangle = U|u\rangle \tag{2.42}$$

where $|u'\rangle$ is the state vector $|u\rangle$ in the $\{|\phi_1\rangle, |\phi_2\rangle\}$ basis.

The *Hadamard matrix* transforms the computational basis $\{|0\rangle, |1\rangle\}$ to the complementary basis $\{|+\rangle, \{|-\rangle\}$, where the complementary basis is the eigenvectors of the σ_x.

2.6 Tensor products

Single qubit systems are the simplest quantum systems we can study. However, the computational abilities of a single qubit are quite limited. To perform computations that a convectional computer is unable to do we need multiple qubits to interact with each other. But let's take one step at a time and see how two quantum systems can combine.

The *tensor product*, which sometimes also called *Kronecker product*, plays a crucial role in quantum mechanics, since it offers a way of combining quantum systems to create larger ones, which we call *composite systems*. Concisely, tensor products are used to merge vector spaces.

Definition 2.6.1 (*Tensor product*)**.** The *tensor product* of two vector spaces, namely $V_1 \in \mathbb{C}^{d_1}$ and $V_2 \in \mathbb{C}^{d_2}$, is defined as

$$V = V_1 \otimes V_2 \tag{2.43}$$

where V is of $d = d_1\, d_2$ dimensions. In general, *tensor products* are not limited with regard to the number of vector spaces we can put together at the same time.

In the case of qubits, suppose that we have 2 separate state vectors $|\psi\rangle$,$|\chi\rangle \in \mathbb{C}^2$. The column vector notation of the tensor product is defined as

$$|\psi\rangle \otimes |\chi\rangle = \begin{pmatrix}\psi_1\\ \psi_2\end{pmatrix} \otimes \begin{pmatrix}\chi_1\\ \chi_2\end{pmatrix} = \begin{pmatrix}\psi_1 \cdot \begin{pmatrix}\chi_1\\ \chi_2\end{pmatrix}\\ \psi_2 \cdot \begin{pmatrix}\chi_1\\ \chi_2\end{pmatrix}\end{pmatrix} = \begin{pmatrix}\psi_1\,\chi_1\\ \psi_1\,\chi_2\\ \psi_2\,\chi_1\\ \psi_2\,\chi_2\end{pmatrix} \tag{2.44}$$

The dimensions of the composite system grow exponentially with the number of qubits. Meaning that a Hilbert space, where n qubits interact, would be of 2^n dimensions.

Sometimes, to simplify things, we use abbreviations when we write down the tensor product of states. All the following notations are equivalent

$$|u\rangle \otimes |v\rangle = |u\rangle\,|v\rangle = |u, v\rangle = |uv\rangle \tag{2.45}$$

As every product in vector spaces, tensor product product has some properties that it must obey.

1. Tensor products between state vectors are *linear*. Let $|v_1\rangle, |v_2\rangle \in V$ and $|w\rangle \in W$, where V and W are two vector spaces. Then

$$|w\rangle \otimes (|v_1\rangle + |v_2\rangle) = |w\rangle \otimes |v_1\rangle + |w\rangle \otimes |v_2\rangle \tag{2.46}$$

2. Tensor products are *associative*. If a is a scalar and $|v\rangle \in V$, $|w\rangle \in W$. Then

$$a(|w\rangle \otimes |v\rangle) = a\,|w\rangle \otimes |v\rangle = |w\rangle \otimes a\,|v\rangle \tag{2.47}$$

3. Tensor products are *noncommutative*. Suppose two states $|v\rangle$ and $|w\rangle$, that belong to two different vector spaces, V and W, respectively. Then

$$|w\rangle \otimes |v\rangle \neq |v\rangle \otimes |w\rangle \tag{2.48}$$

Apart from vectors, tensor products can be applied to operators too. It is defined in a similar fashion so as to hold all the properties that we mentioned. Suppose that A and B are linear operators that act on V and W, respectively. Then if $|v\rangle \in V$ and $|w\rangle \in W$

$$A \otimes B(|v\rangle \otimes |w\rangle) = A\,|v\rangle \otimes B\,|w\rangle \tag{2.49}$$

In other words, $A \otimes B$ is a well defined linear operator that acts on the $V \otimes W$ vector space. I understand that this discussion might seem rather abstract. In matrix notation however, things are more concrete. Let A and B two matrices of $n \times n$ and $m \times m$ dimensions, respectively. The matrix representation the tensor product is

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ & \ddots & \\ a_{n1}B & \cdots & a_{nn}B \end{pmatrix}_{(nm_\times nm)} \tag{2.50}$$

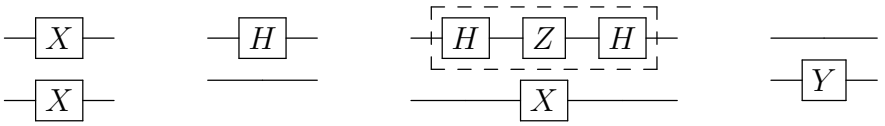
The resulting matrix is of $nm_\times nm$ dimensions.

2.7 Multi-particle gates

Tensor products play a fundamental role in generalizing quantum gates to multiple qubits. They allow us to combine any kind of single qubit gates together. These types of gates are called *local*.

Definition 2.7.1 (*Local operators*)**.** *Local operators* are unitary operators that can be factorized with a tensor product. In other words, an operator is local if it acts on only one part of the composite system.

Some characteristic examples of local gates and their circuit are shown below



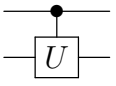
$X \otimes X$ $\quad H \otimes I$ $\quad (HZH) \otimes X$ $\quad I \otimes Y$

We can expand the tensor product to describe larger systems. Positioning the operators properly on the tensor product allows us to create any local gate we want. The wires in the quantum circuit illustration doesn't automatically imply a physical wire, but rather they represent the passage of time. We read the circuit from left to right. This is an important step in understanding how we can translate quantum operations into circuit illustrations.

However, local gates does not utilize fully the computational ability that quantum mechanics offer. The *non-local operators* take advantage of the strange quantum phenomena that rise through the interaction between particles. One such phenomenon is *entanglement*, that allows us to do calculations that otherwise are deemed impossible. Entanglement is such a bizarre and fascinating phenomenon that proves we don't have the intuition nor the senses to fully comprehend how reality actually works. We will devote the entire next section to elaborate more on this phenomenon, but for now that's all we need to know to properly introduce *non-local operators*.

Definition 2.7.2 (*Non-local operators*)**.** *Non-local operators* are unitary operators that *can't* be factorized into a tensor product of single qubit gates. This due to the fact that the action of the gate on one qubit is directly controlled by the state of another qubit.

The most typical class of *non-local operators* are the *control gates*. Control gates has two qubits as inputs, the control and the target. When the control qubit is in the state $|1\rangle$, we perform a unitary operation on the target qubit. While if the control qubit is in the state $|0\rangle$, we don't perform any actions on the second qubit. These gates are also called *Controlled-U*



$P_0 \otimes I + P_1 \otimes U$

An important example of these gates is the *Controlled-NOT (CNOT)*, where the unitary operation is the X gate as illustrated below

$$P_0 \otimes I + P_1 \otimes X$$

In order to reduce the abstraction around the CNOT, and get a better understanding of tensor products, it is also useful to calculate the matrix representation

$$CNOT = P_0 \otimes I + P_1 \otimes X = \begin{pmatrix} I & \mathbf{0} \\ \mathbf{0} & X \end{pmatrix} \widehat{=} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.51}$$

The *CNOT* is the most important multi-qubit gate, because as it turns out it has a *universal behaviour*. A universal set of quantum gates, is any set of gates that any other unitary operation can be expressed as a finite sequence of gates from that set. CNOT and arbitrary single qubit rotation gates form one such universal set of quantum gates.

2.8 Entanglement

So far we have seen that quantum mechanics are bizarre and counter-intuitive compared to the reality we are experiencing in a macroscopic level. However, things can get even more strange. In fact, there is a phenomenon in the quantum world that is so absurd, that has many physicists consider quantum theory incomplete to this day.

This phenomenon, that lies at the heart of quantum mechanics, is called *quantum entanglement*. We say that two particles are *entangled* when the quantum state of each particle cannot be described independently of the state of the other. This phenomenon was the subject of the famous *EPR* paper, which was published by *Albert Einstein*, *Boris Podolski* and *Nathan Rosen* in 1935. In this paper, they stated that if two particles entangle and then separate, measurement of one of the particles will also determine the state of the other particle. This holds true even when the particles are spatially separated and are no longer interacting at the time of measurement. They deemed that this was impossible, and Einstein even referred to entanglement as "spooky action at a distance".

Nevertheless, despite not fully understanding yet the mechanics behind it, we utilize entanglement as a valuable resource for computation.

Definition 2.8.1 (*Product states*)**.** A composite system is in a *Product state*, or separable state, if it can be expressed as a tensor product of the individual quantum states that compose it.

Definition 2.8.2 (*Entangled states*)**.** A composite system that is not in a *product state*, is *entangled*.

We can define a qualitative criterion to identify whether or not a composite system is entangled.

Definition 2.8.3 (*Qualitative measure of entanglement*)**.** Suppose that we have a bipartite composite system of two qubits, $|q\rangle_{\mathbf{A}}, |q\rangle_{\mathbf{B}}$, with respective Hilbert spaces $\mathcal{H}_A$ and $\mathcal{H}_B$. Individually, each qubit is in a 2-dimensional Hilbert space, while the composite system is in 2^2 dimensions. Any state vector in the composite system can be expanded, with respect to the computational basis, as

$$|\psi\rangle = \sum_{q_A=0}^{1} \sum_{q_B=0}^{1} a_{q_A} |q\rangle_{\mathbf{A}} \otimes b_{q_B} |q\rangle_{\mathbf{B}} = \begin{pmatrix} a_0\, b_0 \\ a_0\, b_1 \\ a_1\, b_0 \\ a_1\, b_1 \end{pmatrix} \tag{2.52}$$

where $a_{q_A}, b_{q_B} \in \mathbb{C}$ and, as probability amplitudes, satisfy

$$\sum_{q_1,q_2} |a_{q_1} b_{q_2}|^2 = 1. \tag{2.53}$$

Vectors and matrices have an isomorphic relation that allows us to define the *coefficient* matrix of the state $|\psi\rangle$

$$C = \begin{pmatrix} a_0\, b_0 & a_0\, b_1 \\ a_1\, b_0 & a_1\, b_1 \end{pmatrix} \tag{2.54}$$

We can positively say that the state $|\psi\rangle$ can be factored as a *product stated*, if $det(C) = 0$. We know from linear algebra that when the determinant of a matrix is zero, it implies that its columns are *linearly independent*. This can be generalized to apply to a composite system of more dimensions.

Apart from just declaring whether or not a set of subsystems is entangled, we can also quantify the amount of entanglement that is present system.

Definition 2.8.4 (*Quantitative measure of entanglement*)**.** Suppose that we have again a bipartite system. The *coefficients matrix* C, with elements $c_{ij} = a_i b_j$, can generate the probabilities of each state $|i\rangle_{\mathbf{A}} \otimes |j\rangle_{\mathbf{B}}$ of the system as $p(ij) = |c_{ij}|^2$. The *Shannon entropy* gives us a quantitative meausure of entanglement

$$S(p(ij)) = -\sum_{i=0}^{1} \sum_{j=0}^{1} p(ij)\ log_2\left(p(ij)\right) \tag{2.55}$$

Shannon entropy has its roots in information theory and is used as a measure of *uncertainty*. Uncertainty refers to how "surprising" the average outcome of a quantum measurement is. Entropy ranges from 0, for a separable product state, to 1 for a maximally entangled state.

The most famous entangled states are the maximally entangled *Bell states*. Bell states form an orthonomal basis in $\mathcal{H}^4$ called *the Bell basis*. These states are special because by measuring the state of one qubit, we definitively know the state of the other!

$$\left|\Phi^{\pm}\right\rangle = \frac{|00\rangle \pm |11\rangle}{\sqrt{2}} \tag{2.56}$$

$$\left|\Psi^{\pm}\right\rangle = \frac{|01\rangle \pm |10\rangle}{\sqrt{2}} \tag{2.57}$$

Chapter 3

Machine Learning

Humanity throughout history strove to find patterns in nature. This quest, during the 19th and 20th centuries, gave birth to mathematical techniques for analyzing data to reveal patterns, such as learning optima via gradient decent and polynomial interpolation. The development of digital computers during the second half of the 20th century, allowed the mathematical methods of data analysis to be automated. *Machine learning (ML)* grew out of the field of *artificial intelligence* at the late 50's when some scientists attempted to allow computer systems to constantly improve their performance in certain tasks through the use of data and statistical techniques. It is the study of computer algorithms that pertains to searching for patterns and correlations in data. ML algorithms build a model based on data, in order to make decisions or predictions without being explicitly programmed to do so. The accuracy of the model is improved automatically through the data, albeit it requires a huge amount of data for a machine learning model to achieve an acceptable accuracy. The lack of data, back in the 60's, create a hurdle that forced machine learning to stay in purely academic cycles for some decades.

The emergence of the *Big Data* reignited the interest for machine learning. Many major technological companies saw great success from storing, processing and extracting value from their huge volume of data. More and more companies start to follow that path as of late. However, the data sets are becoming too large or too complex to be dealt with by traditional data-processing methods and analytics. This is where machine learning fits perfectly, since the more data a system receives, the more it learns recognize patterns, and thus, offer helpful insights for business operations.

Besides businesses, machine learning algorithms are slowly integrating to a variety of applications, ranging from medicine and stock market trading to security systems and speech recognition. This variety of emerging applications is due to the fact that machine learning employs different approaches to teach computers. In the theory of machine learning, the *learning* is traditionally divided into three broad categories, *supervised learning*, *unsupervised learning* and *reinforcement learning*.

In *supervised learning*, the goal is to develop a predictive model that maps an input to an output. The data set that is used to train the model is *labeled* and consists of example input-output pairs. To evaluate the training process, the model is put to test by feeding it unknown data, with hidden labels, and is instructed to predict the output based on its knowledge from the training.

Conversely, in *unsupervised learning*, we don't have labeled pairs of inputs-outputs, and hence, the goal is to group and interpret the correlations of the input data. The figure below sums up the different machine learning approaches. Finally, *reinforcement learning* is the mechanism behind the development and study of *intelligent agents*. In the field of artificial intelligence an agent is a computer program that acts as a player in a game. Agents learn by rewarding (reinforcing) or punishing the strategy they use to win.

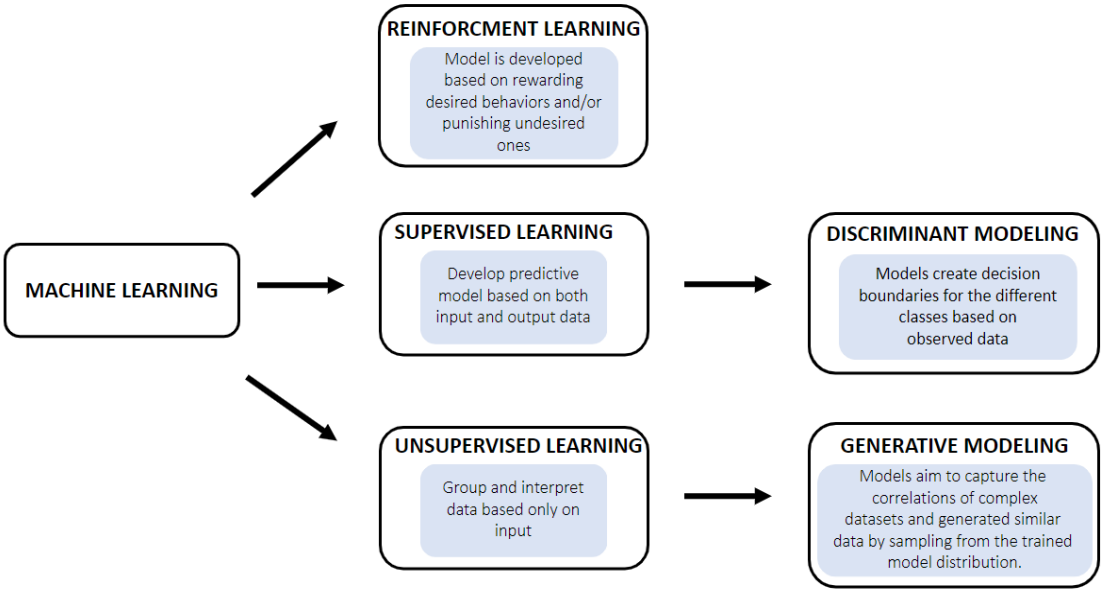


Figure 3.1: Map of machine learning.

This chapter of the thesis is aimed at introducing *generative modeling* using *neural networks*. For the latter, we will use a special type of neural networks that are very popular in generative modeling, called *Restricted Boltzmann Machines (RBMs)*. To make the idea more concrete we will start from a single neuron and build our way up to restricted Boltzmann machines and how we can utilize them for generative modeling.

3.1 Artificial neurons

Artificial neural networks (ANN) are computing systems inspired by the model of the human brain. They are a collection of connected units or nodes called *artificial neurons*, which loosely model the neurons in a biological brain. Basically, we can

think of them as functions that transforms input data to output based on their training on many data samples.

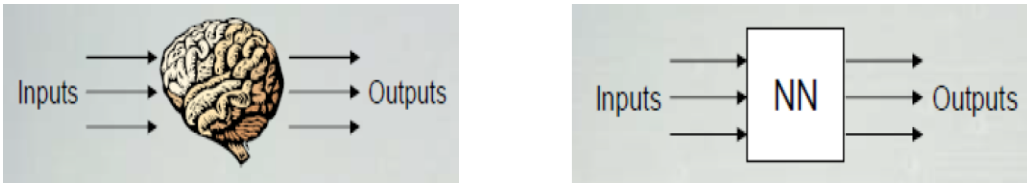


Figure 3.2: Neural networks are inspired by the brain model and behave as a function

Similar to a biological neuron, an *artificial neuron*, has weighted inputs from other neurons. Also, it may have a threshold such that a signal is sent only if the weighted sum of it's input signals crosses that threshold. Mathematically the neuron activation is based on,

$$z = b + \sum_j w_j \cdot x_j \tag{3.1}$$

where w_j are the weights of the inputs, x_j are the inputs and b is a constant called *bias*. The bias allows to shift the activation by adding a constant to the input. The output of the neuron is some non-linear function $f(z)$ of the weighted sum of it's input signals. There are many functions that can be used at this point. However, we only need to introduce the one we will use later on, the *sigmoid function*.

Definition 3.1.1 (*Sigmoid function*)**.** A sigmoid function is a mathematical function having a characteristic "S"-shaped curve and is defined as

$$f(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1} \tag{3.2}$$

The output of the sigmoid function is in the $[0, 1]$ range.

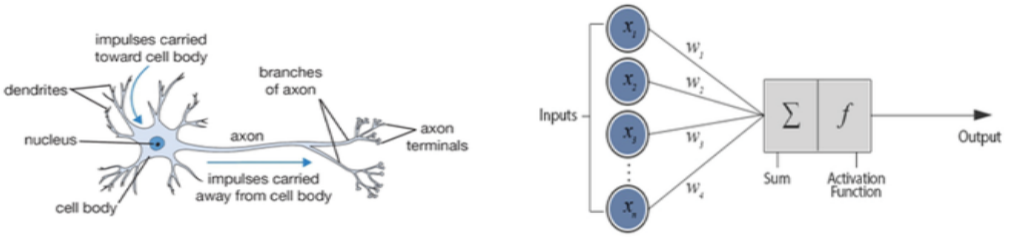


Figure 3.3: Computational Model of a Biological Neuron

A collection of connected neurons builds up what we call a neural network. Typically, the neurons are organised into layers. Neurons of one layer connect only to neurons of the immediately preceding and immediately following layers. The layer that receives external data is the input layer, while the the layer that produces the

ultimate result is the output layer. The layers that are stacked between the input and output layers are called *hidden* layers. Any neural network with more than one hidden layer is called a *deep* neural network.

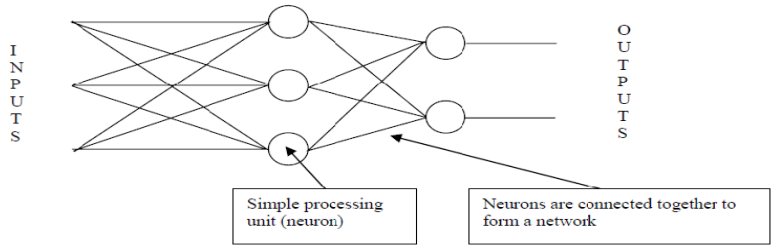


Figure 3.4: Structure of a simple neural network.

Depending on the way the neurons are connected it can give rise to various types of neural networks that have different applications. In general, neural networks are widely used in supervised learning for classification tasks, where they can reach an accuracy $> 98\%$. The mathematical description of the learning process of artificial neuron can be found in Appendix A.

3.2 Restricted Boltzmann Machines

Apart from supervised learning, neural networks can also be used in unsupervised learning, where the data is unlabeled. *Restricted Boltzmann Machines (RBMs)* are a special type of neural networks that was widely used as building blocks in deep learning architectures and they continue to play an important role in applied and theoretical machine learning. In this work we will train a RBM to perform *generative modeling*.

Generative modeling is a very useful application of neural networks that falls under the category of unsupervised learning. The model automatically discovers and learns the correlations of the input data, so as to be able to generate new instances that could have been possibly drawn from the original data set. Simply put, the model captures the distribution of the data.

3.2.1 Historical background

In 1982 Hopfield introduced a fully connected network of interacting units that have the ability to store and retrieve binary patterns [2]. Hopfield networks can be considered as a dynamical system in which the stable states of the system correspond to the patterns we want to store. They belong to the category of energy based models where the desired patterns are associated with the minima of a suitably defined energy. In the figure below we can see a Hopfield network with 5 neurons.

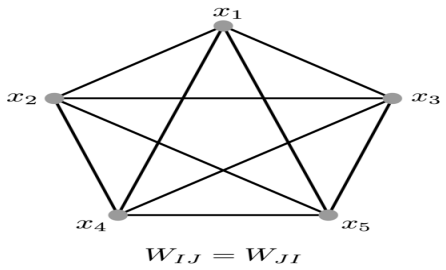


Figure 3.5: A graphical representation of the fully-connected Hopfield network consisting of five neurons. The synaptic weights are described by a symmetric matrix $\mathbf{W_{IJ}}$

The network is initialized randomly and each unit updates its state through a simple rule that depends on the units it is connected to. Mimicking nature, the evolution of a Hopfield network constantly decreases its energy. An interesting question to be raised at this point is where did the inspiration for this "energy" concept came from?

Hopfield mentioned a *spin glass* system based on the *Ising model* [3]. In physics, a spin glass is a magnetic state characterized by randomness. In contrast with a ferromagnet where all the spins align, the spins in a spin glass are aligned randomly without a regular pattern. This is depicted in the figure below.

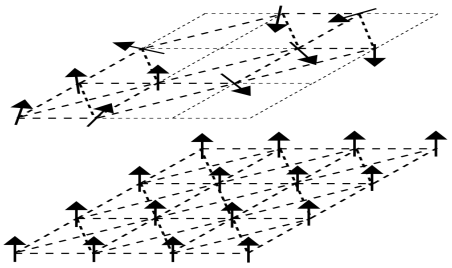


Figure 3.6: At the top we have a schematic of random spins in a spin glass and at the bottom we have the alignment at a ferromagnet

When a external field is applied each magnetic spin tries to align itself to the local field and in doing so it may flip. However this causes a chain reaction because it will change the fields at other dipoles. A change at the magnetic fields of other dipoles may cause them to flip which in turn changes the field at the current dipole. This process is what we call the evolution of the system and it continues until the system reaches a minimum energy state. The dipoles stop flipping if any flips results in increase of energy.

The Hopfield network was very influential in the development of neural network models in the 80's. Though successful in storing/retrieving the desired patterns, it was observed that the Hopfield model has various problems such as limited storing capacity and spurious minima. In an attempt to mitigate these issues, a stochastic

version of the Hopfield model, called *Boltzmann machine*, was proposed [4]. In contrast to Hopfield networks, where the units are deterministic, Boltzmann machine units are stochastic. Each unit updates its state over time in a probabilistic way depending on the states of the neighboring units. The units of a Boltzmann machine are divided into 'visible' units, V, and 'hidden' units, H. The visible units are the ones that interact with the environment and work as information input to the machine, while the hidden units are latent variables forming a conditional hidden representation of the data.

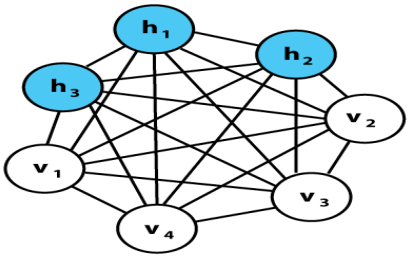


Figure 3.7: This is a connectivity diagram of a Boltzmann machine. Each undirected edge represents dependency. In this example there are 3 hidden units and 4 visible units.

However, learning the parameters of the Boltzmann Machine model is computationally intensive. To reduce the complexity of learning, a restricted communication structure was introduced [5, 6, 7]. This model is called the *Restricted Boltzmann Machine*.

3.2.2 Structure

A restricted Boltzmann machine is a probabilistic energy based model that pertains to the mathematical model of interacting particles, like the Ising model. Similar to a Boltzmann machine, the architecture is organised into two layers, that are most commonly referred to as the "visible" and "hidden" units respectively. However, unlike Boltzmann machines, lateral connections within a layer are prohibited to make the computations faster, and hence, the edges are defined as $e = \{\{v, h\} : v \in V, h \in H\}$. This is illustrated in the figure below.

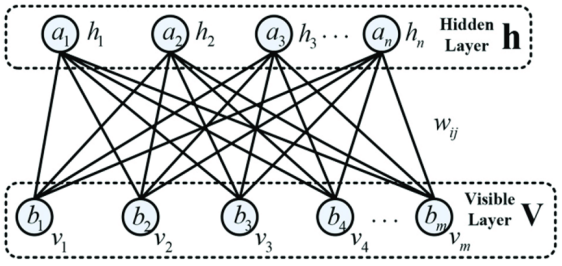


Figure 3.8: The connections between the two unit layers of a RBM form a bipartite graph.

The units of a restricted Boltzmann machine are stochastic binary units that can be either 0 or 1. The joint state of each layer is represented by vectors $\mathbf{v} \in \{0,1\}^V$ and $\mathbf{h} \in \{0,1\}^H$. Generally, there are other types of units that can be used, such as Gaussian units, but the most widely used are the stochastic binary units that we see here.

Each unit has an associated *bias* and each connection has an associated *weight*. As a generative model, a restricted Boltzmann machine represents a probability distribution. The network assigns a probability to every possible pair of a visible and a hidden vector as

$$p(\mathbf{v}, \mathbf{h}) = \frac{1}{Z}\, e^{-E(\mathbf{v},\mathbf{h})}. \tag{3.3}$$

The probability of a visible sample vector, $\mathbf{v}$, is derived by summing over all possible hidden vectors

$$p(\mathbf{v}) = \frac{1}{Z} \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}. \tag{3.4}$$

Each joint configuration of all units $(\mathbf{v}, \mathbf{h})$ has an energy value $\mathrm{E}(\mathbf{v}, \mathbf{h})$ which depends on the pairwise interactions of units and biases

$$E(\mathbf{v}, \mathbf{h}) = -\sum_{i \in visible} a_i v_i \; - \sum_{j \in hidden} b_j h_j \; - \sum_{i,j} w_{ij} v_i h_j \tag{3.5}$$

$$= -\mathbf{a}^T \mathbf{v} - \mathbf{b}^T \mathbf{h} - \mathbf{v}^T W \mathbf{h} \tag{3.6}$$

where v_i, h_j are the binary states of the visible unit i and hidden unit j, a_i, b_j are their biases and w_{ij} is the weight between them.

The *'partition function'*, Z, is calculated by summing over all possible pairs of visible and hidden vectors

$$Z = \sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}. \tag{3.7}$$

The partition function can be interpreted as a normalizing constant to ensure that the probabilities sum to 1. However, there is a problem hidden here. With a careful observation we can see that it is computationally expensive to calculate the partition function, since the integration over all possible states is only computable for small toy problems. The *partition function*, Z, is exponential w.r.t to the number of the hidden and visible units of the model and complexity theory tells us that any problem that is exponential in nature should be regarded intractable on any conventional computer. This is a major issue because it makes the joint probability distribution $p(\mathbf{v}, \mathbf{h})$ intractable.

The solution to this problem is found in the field of statistics with an algorithm called *Gibbs sampling*, which is employed when direct sampling is difficult. In statistics, Gibbs sampling is a *Markov chain Monte Carlo* algorithm that constructs a Markov chain whose values converge towards a target distribution. In other words, instead of calculating the joint configuration $(\mathbf{v}, \mathbf{h})$, alternate transitions between $\mathbf{v}$ and $\mathbf{h}$ states and sample from $p(\mathbf{h}|\mathbf{v})$ and $p(\mathbf{v}|\mathbf{h})$ that are easy to calculate. We can simultaneously and independently sample from all the elements of $\mathbf{h}$ given $\mathbf{v}$ and the same holds for $\mathbf{v}$ given $\mathbf{h}$. To get an unbiased estimate, we need the Markov chain to converge to a stationary distribution.

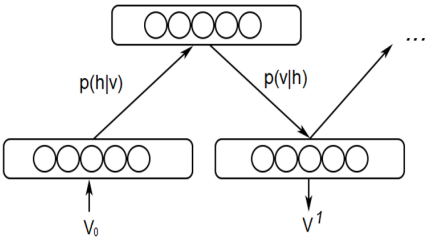


Figure 3.9: Given some visible vector $\mathbf{v}$, calculate the probabilities $p(h_j = 1|\mathbf{v})$. Then, sample according to these probabilities, to obtain a new hidden state vector $\mathbf{h}$. Similarly, go from $\mathbf{h}$ to a new $\mathbf{v}^{'}$, using $p(v_i = 1|\mathbf{h})$. This process is repeated until the chain converges.

Using Bayes theorem we can derive the probability of the hidden units conditioned on all visible units,

$$p(\mathbf{h}|\mathbf{v}) = \frac{p(\mathbf{h},\mathbf{v})}{p(\mathbf{v})} = \frac{p(\mathbf{v},\mathbf{h})}{\sum_{\mathbf{h}} p(\mathbf{v},\mathbf{h})} = \frac{e^{-E(\mathbf{v},\mathbf{h})}}{\sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}} \tag{3.8}$$

$$= \frac{e^{\mathbf{a}^T\mathbf{v}} \cdot e^{\mathbf{b}^T\mathbf{h}+\mathbf{v}^TW\mathbf{h}}}{\sum_{\mathbf{h}} e^{\mathbf{a}^T\mathbf{v}} \cdot e^{\mathbf{b}^T\mathbf{h}+\mathbf{v}^TW\mathbf{h}}} = \frac{e^{\mathbf{b}^T\mathbf{h}+\mathbf{v}^TW\mathbf{h}}}{\sum_{\mathbf{h}} e^{\mathbf{b}^T\mathbf{h}+\mathbf{v}^TWh}} \tag{3.9}$$

$$= \frac{e^{\sum_j (b_j h_j + \sum_i w_{ij} v_i h_j)}}{\sum_{\mathbf{h}} e^{\sum_j (b_j h_j + \sum_i w_{ij} v_i h_j)}} = \prod_j \frac{e^{(b_j h_j + \sum_i w_{ij} v_i h_j)}}{\sum_{\mathbf{h}} e^{(b_j h_j + \sum_i w_{ij} v_i h_j)}} \tag{3.10}$$

$$= \prod_j \frac{e^{(b_j h_j + \sum_i w_{ij} v_i h_j)}}{1 + e^{(b_j + \sum_i w_{ij} v_i)}} = \prod_j p(h_j|\mathbf{v}). \tag{3.11}$$

Similarly,

$$p(\mathbf{v}|\mathbf{h}) = \prod_i p(v_i|\mathbf{h}). \tag{3.12}$$

An important point to notice here is that we have a product of probabilities. This implies conditional independence of visible units conditioned on all hidden units and vice-versa. The conditional independence is also confirmed by the bipartite structure of the model.

The above conditional distributions can be expressed as

$$p(h_j = 1|\mathbf{v}) = \sigma\left(b_j \ + \sum_i w_{ij} v_i\right) \quad , \quad p(h_j = 0|\mathbf{v}) = 1 \ - \ p(h_i = 1|\mathbf{v}) \tag{3.13}$$

$$p(v_i = 1|\mathbf{h}) = \sigma\left(a_i \ + \sum_j w_{ij} h_j\right) \quad , \quad p(v_i = 0|\mathbf{h}) = 1 \ - \ p(v_i = 1|\mathbf{h}) \tag{3.14}$$

where $\sigma(x)$ is the *sigmoid function* that we saw earlier.

3.2.3 Training a Restricted Boltzmann Machine

Training a restricted Boltzmann machine is finding the parameters θ, such that the distribution represented by the model $p(\mathbf{v}|\theta)$ closely matches the desired distribution as indicated by the training data. The parameters, θ, of the RBM include all the connection weights and the biases.

To find the parameters, θ of the model, we have to minimize the *Kullback-Leibler divergence* between the model and the data distribution $p_{data}(\mathbf{v})$. KL-divergence is a popular distribution distance measure and is defined as,

$$d_{KL}(p_{data}(\mathbf{v}) \ || \ p(\mathbf{v}|\theta)) = \sum_{\mathbf{v}\in\mathcal{S}} p_{data}(\mathbf{v}) \ log\left(\frac{p_{data}(\mathbf{v})}{p(\mathbf{v}|\theta)}\right) \tag{3.15}$$

where $\mathcal{S} = \{\mathbf{v^{(1)}}, \mathbf{v^{(2)}}, \cdots, \mathbf{v^{(N)}}\}$ are the training samples.

Interestingly minimizing KL-divergence results to maximizing the *log-likelihood*,

$$arg\min_\theta d_{KL} = arg\min_\theta \sum_{\mathbf{v}\in\mathcal{S}} (p_{data}(\mathbf{v}) \ log\, p_{data}(\mathbf{v}) - p_{data}(\mathbf{v}) \ log\, p(\mathbf{v}|\theta)) \tag{3.16}$$

$$= arg\max_\theta \sum_{\mathbf{v}\in\mathcal{S}} p_{data}(\mathbf{v}) \ log\, p(\mathbf{v}|\theta). \tag{3.17}$$

Maximizing the *log-likelihood* is usually done using gradient based optimization methods. The log-likelihood of θ for a given vector, $\mathbf{v}$, can be calculated by summing out the hidden units from the joint distribution,

$$\ln\, p(\mathbf{v}|\theta) = \ln\left(\frac{1}{Z}\sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}\right) \tag{3.18}$$

$$= \ln\sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} - \ln\sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}. \tag{3.19}$$

The gradient of the log-likelihood with respect to the parameters we want to optimize is evaluated as,

$$\boldsymbol{\nabla}_\theta \ \ln p(\mathbf{v}|\theta) = \boldsymbol{\nabla}_\theta \ln \sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} - \boldsymbol{\nabla}_\theta \ln \sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \tag{3.20}$$

$$= \frac{\sum_{\mathbf{h}} \boldsymbol{\nabla}_\theta \ e^{-E(\mathbf{v},\mathbf{h})}}{\sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}} - \frac{\sum_{\mathbf{v},\mathbf{h}} \boldsymbol{\nabla}_\theta \ e^{-E(\mathbf{v},\mathbf{h})}}{\sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}} \tag{3.21}$$

$$= -\frac{\sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \boldsymbol{\nabla}_\theta \ E(\mathbf{v},\mathbf{h})}{\sum_{\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}} + \frac{\sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})} \boldsymbol{\nabla}_\theta \ E(\mathbf{v},\mathbf{h})}{\sum_{\mathbf{v},\mathbf{h}} e^{-E(\mathbf{v},\mathbf{h})}} \tag{3.22}$$

$$= -\sum_{\mathbf{h}} p(\mathbf{h}|\mathbf{v}) \boldsymbol{\nabla}_\theta \ E(\mathbf{v},\mathbf{h}) + \sum_{\mathbf{v},\mathbf{h}} p(\mathbf{v},\mathbf{h}) \boldsymbol{\nabla}_\theta \ E(\mathbf{v},\mathbf{h}) \tag{3.23}$$

For a given set of training samples, $\mathcal{S} = \{\mathbf{v^{(1)}}, \mathbf{v^{(2)}}, \cdots, \mathbf{v^{(N)}}\}$, the log-likelihood gradient is,

$$\frac{1}{N} \sum_{l=1}^{N} \boldsymbol{\nabla}_\theta \ \ln p(\mathbf{v}^{(l)}|\theta) = \frac{1}{N} \sum_{l=1}^{N} \left(-\sum_{\mathbf{h}} p(\mathbf{h}|\mathbf{v}) \boldsymbol{\nabla}_\theta \ E(\mathbf{v},\mathbf{h}) + \sum_{\mathbf{v},\mathbf{h}} p(\mathbf{v},\mathbf{h}) \boldsymbol{\nabla}_\theta \ E(\mathbf{v},\mathbf{h}) \right) \tag{3.24}$$

$$= \mathbb{E}_{p_{model}} \left[\boldsymbol{\nabla}_\theta \ E(\mathbf{v},\mathbf{h}) \right] - \mathbb{E}_{p_{data}} \left[\boldsymbol{\nabla}_\theta \ E(\mathbf{v},\mathbf{h}) \right]. \tag{3.25}$$

Specifically, the updates for weights and biases are,

$$\Delta w_{ij} = \mathbb{E}_{p_{data}} \left[v_i h_j \right] - \mathbb{E}_{p_{model}} \left[v_i h_j \right] \tag{3.26}$$

$$\Delta a_i = \mathbb{E}_{p_{data}} \left[v_i \right] - \mathbb{E}_{p_{model}} \left[v_i \right] \tag{3.27}$$

$$\Delta b_j = \mathbb{E}_{p_{data}} \left[h_j \right] - \mathbb{E}_{p_{model}} \left[h_j \right]. \tag{3.28}$$

The first term is easy to be obtained and is called the *positive phase*. It calculates the expectation of the hidden probabilities given the data under the current model. The second term is called the *negative phase* and calculates the expectation of the joint probability of visible and hidden units under the current model. However, the second term is exponential in the size of the smallest layer $2^{\min(m,n)}$. The following equations shine some light as to why this the case

$$\sum_{\mathbf{v},\mathbf{h}} p(\mathbf{v},\mathbf{h}) \boldsymbol{\nabla}_\theta \ E(\mathbf{v},\mathbf{h}) = \sum_{\mathbf{v}} p(\mathbf{v}) \sum_{\mathbf{h}} p(\mathbf{h}|\mathbf{v}) \boldsymbol{\nabla}_\theta \ E(\mathbf{v},\mathbf{h}) \tag{3.29}$$

$$= \sum_{\mathbf{h}} p(\mathbf{h}) \sum_{\mathbf{v}} p(\mathbf{v}|\mathbf{h}) \boldsymbol{\nabla}_\theta \ E(\mathbf{v},\mathbf{h}) \tag{3.30}$$

Thus, once again Markov chain Monte Carlo sampling methods are used to obtain expectation under the model distribution. The samples are obtained when the

Markov chain converges to the stationary distribution. But if one waits for the chain to converge at each iteration, the computational cost becomes large again. To avoid this large computational cost, we use an algorithm called *Contrastive Divergence* [7].

The idea is to reduce the computational cost by initializing the Markov chain close to the desired distribution. This is achieved by initializing the chain with samples from the data set. Moreover, the expectation is replaced by a single sample v^k that is obtained after running a Markov chain for k steps. This way we eliminate almost all of the computation that is required to get samples from the converged distribution.

According to Hinton, we can interpret 'contrastive divergence' as the difference between two KL-divergences. He states that instead of simply minimizing the KL-divergence between the data distribution and the converged distribution of the Markov chain, we can minimize the following,

$$d_{KL}(p_{data}(\mathbf{v}) \mid\mid p(\mathbf{v}|\theta)) = d_{KL}(p_{data}(\mathbf{v}) \mid\mid p(\mathbf{v}|\theta)) - d_{KL}(p_k(\mathbf{v}) \mid\mid p(\mathbf{v}|\theta)) \qquad (3.31)$$

where $p_k(\mathbf{v})$ is the distribution of the chain after k steps. In practice, only one step of Gibbs sampling will suffice for most problems.

3.2.4 Learning the MNIST digits

To demonstrate how generative modeling works with a restricted Boltzmann machine, we coded and trained one to learn the distribution of handwritten digits. Most people effortlessly recognize handwritten digits, however the ease is deceptive. Our brains has millions of neurons, that are fine tuned with by thousands of years of evolution. Practically, we walk around with a supercomputer in our heads, that is stupendously good at making sense of the visual signals that reach our eyes. So, it should come at no surprise that recognizing handwritten digits is not trivial for a convectional computer, let alone generating new samples. Recognizing handwritten digits is a widely known problem in the field of machine learning and is now considered solved since there has been networks that can tackle the problem with an accuracy of over 99 percent. In our experiment, we will not attempt to recognise but rather generate new samples that could have possibly drawn from the initial data.

Figure 3.10: A random selection of MNIST digits

As data we utilized the MNIST digits database that is easily accessible and tailor made for machine learning experiments. The MNIST database consists 60,000 training images and 10,000 testing images. Each image is a 28 by 28 pixel scanned image of a single handwritten digit. The values of each pixel range from 0 to 255. However, a RBM has binary units, and thus, we convert the the grayscale images with values [0,255] to binary with values [0,1]. The mapping is done by setting a threshold at 128. If the pixel value is below 128 we set the value to 0, otherwise we set the value to 1.

The visible layer of the network contains units that encode the values of the input pixels, and hence, we will have $28 \times 28 = 784$ units. For the hidden layer we found that 100 units will suffice to get good approximations of the digits. To train the network we implemented a mini-batch version of the Contrastive Divergence algorithm according to the Hinton practical guide [8]. Hinton suggests that for data sets that contain a small number of equiprobable classes, the ideal mini-batch size is often equal to the number of classes. So, in this case we chose each batch to contain 10 images. Furthermore, in order to avoid changing the learning rate when the size of a mini-batch is changed, we use as learning rate 1/mini-batch size. The code that we implemented for our RBM can be found in Appendix A.

Fig(3.11) illustrates how the RBM generates images that bare the same statistical patterns as the MNIST images.

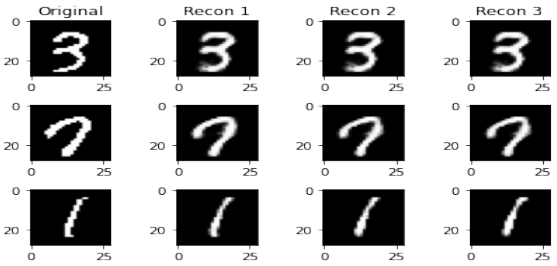


Figure 3.11: Reconstructions of random MNIST digits. Left column indicates a number to the RBM. Going from left to right we can see new images of the same digit. Each column is a reconstruction of the previous and is obtain by a step of Gibbs sampling.

Chapter 4

Quantum Machine Learning

4.1 Introduction to Quantum Machine Learning

The rapid technological advancements of the last couple of decades has helped machine learning to escape the academic cycles and slowly integrate into our everyday lives. However, there are still challenges to overcome, such as the large computational cost of training or deriving insight from problems that are exponentially complex. This is where quantum computation is thought that can be of help. Data analysis and machine learning are based around statistics and linear algebra. But quantum computation in its core is also linearly algebraic. Thus, the scientific community thought that research in integrating those two fields may yield computational benefits. The merging of quantum computation within machine learning algorithms gave rise to a new field that is called *quantum machine learning (QML)*. This field promises advantages such as speed-ups in complex matrix and tensor operations as well as in the training process, while also allows the handling of more complex network topologies.

Research in the field is divided into two major directions. The first one, pertains to the theoretical study of algorithms that run on a *fault-tolerant* quantum computer. In this approach, the assumption is that the speed-up will be derived from executing the subroutines of ML algorithms on quantum computers without changing the structure of the algorithms. In fact, all the evidence that quantum computation will yield speed-ups to ML comes from this approach. Most proposed quantum algorithms however require *fault-tolerant* quantum computers that can be scaled to a few thousand qubits. Unfortunately, the state of the art quantum processors that exist today have a few dozens qubits that are noisy and error-prone.

So, have we reached an impasse for the time being? The hurdles of the *fault-tolerant QML* sparked a new research interest. After the realization that *fault-tolerant quantum computers* won't be available in the near future, people started to investigate whether or not we can utilize these noisy devices to solve various general problems efficiently. This approach is called *noisy intermediate scale quantum* or

simply *NISQ*.

Regardless of the era and the type of quantum computers we possess, QML algorithms can be also classified by the way they fuse together the disciplines of machine learning with quantum computation. The most common combination is to exploit quantum computers to analyse classical data. This is motivated by the belief that quantum computers would in theory outperform their classical counterparts at solving specific problems. This is what we consider *quantum advantage* or speed-up. Alternatively, one can harness machine learning algorithms in order to process data from quantum experiments. In total, there are four different ways of merging the two fields as demonstrated in Fig.(4.1) below. In this thesis, we will be working on the upper squares, developing hybrid quantum classical algorithms for classical data

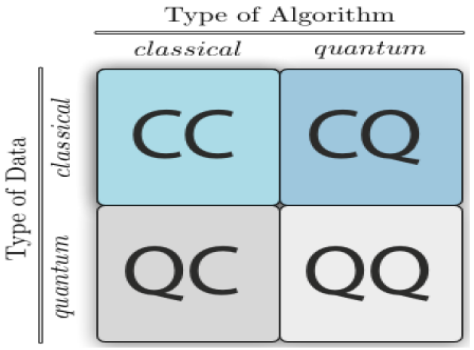


Figure 4.1: The first letter shows whether the type of data that are studied are classical or quantum, where the second letter refers to the nature of the algorithm that will process the data.

4.1.1 Fault-tolerant Quantum Machine Learning

The *fault-tolerant QML* was the one that started the whole QML field back in 2009. It works under the premise that the whole algorithm will run on a quantum computer, and so, those algorithms are evaluated by the speed-up they demonstrate compared to their classical counterparts. Where does this speed-up comes from though?

To understand this, we must first introduce a very important algorithm in the quantum computation field. This algorithm is named after its inventors *Harrow, Hassidim and Lloyd* or simply *HHL* [9]. The algorithm attempts to solve the linear equation $A\boldsymbol{x} = \boldsymbol{b}$ in a quantum computer. This problem is actually very simple to solve in a classical computer but it can get very time consuming. Suppose that A is an $n \times n$ real matrix. The best classical algorithm would require $\mathcal{O}(n \log n)$ to go through the entries of A and output the solution for $\boldsymbol{x}$. *HHL* promises to compute the solution in $\mathcal{O}((\log n)^2)$ by exploiting the exponential character of the wave function.

Unfortunately, in order for *HHL* to achieve this, there are some limitations that need to be overcome. The first major issue is that the algorithm requires a quantum

RAM (qRAM). Vector b needs to be encoded quickly into the quantum computer's memory as the quantum state $|b\rangle$ before the execution of the algorithm. The qRAM allows to store the classical entry values of $\boldsymbol{b}$ and read them all at once as a superposition of states,

$$b = [b_1, b_2, \cdots, b_n]^T \mapsto |b\rangle = \sum_{i=1}^{n} b_i |i\rangle . \tag{4.1}$$

The catch is that qRAMs does not exist currently, as the technology has yet to catch up to theory. However, there is no physical no go theorem why we cannot build one, it is just quite tough. Should we not use a qRAM and attempt to let the quantum computer do the encoding itself, we risk loosing the speed-up. The encoding from $\boldsymbol{b}$ to $|b\rangle$ needs to be done in less than than n^c, where c is some constant.

Besides the encoding, the decoding can also pose problems. Once the quantum computer has finished processing the algorithm, the n values of $\boldsymbol{x}$ are encoded in the $log_2(n)$ amplitudes of the quantum state $|x\rangle$. However, in order to extract all the information out of the state we are forced to repeat the process n times, and as a result we kill the speed-up. This is related to the probabilistic nature of measurement and it is unavoidable, regardless of the basis we choose to measure against.

The third and final issue is the ability of the quantum computer to simulate efficiently unitary operations of the form $exp(-iAt)$, for various values of t. Similar to the encoding case, in order to maintain the speed-up, the simulation of the unitary needs to be done in less than n^c. Having a qRAM could help the system apply the aforementioned unitary transformation in the special case of sparse matrices. A sparse matrix is a matrix in which most of the its elements are zero. Storing the non-zero elements and their positions in a qRAM can boost the quantum device to apply the transformation in time that evolves linearly to the number of the non-zero elements.

To summarize, it seems like *HHL* has a lot of drawbacks for us to be excited about it. The truth is that the excitement is not because *HHL* is an algorithm that solves linear equations in logarithmic time. The excitement arise when we view *HHL* as a model for other quantum algorithms. It provides a template of to carefully encode and prepare $\boldsymbol{b}$, apply unitary transformations and measure $|x\rangle$. But most importantly it indicates us how to derive the cost of the operations and compare it against the best classical algorithms. Moreover, should the constraints of the *HHL* are carefully met, it can be used as a core subroutine to achieve speed-ups compared to the classical ML algorithm as shown in the Fig.(4.2) below.

Algorithm	Speed-up
Bayesian Inference	$O(\sqrt{N})$
Least-squares fitting	$O(log\, N)$
Quantum PCA	$O(log\, N)$
Quantum SVM	$O(log\, N)$

Figure 4.2: Speedups of various QML algorithms compared to the classical ML algorithms. [10, 11, 12, 13]

4.1.2 Noisy-Intermediate Scale Quantum (NISQ) Quantum Machine Learning

The problem with *fault-tolerant QML* is that even if we satisfy the constraints that algorithms such as the HHL bare, we don't have yet the quantum computers that they require to yield the speed-up. The hurdles emerge from the fact that most proposed quantum algorithms require error-correcting quantum computers that can be scaled to a few millions of qubits. This is a far cry from the state of the art quantum processors that exist today that offer just a few dozen noisy qubits.

So, have we reached an impasse for the time being? ...No! The *NISQ-era* is better suitable for the so-called *Variational models.* They are the leading proposal for achieving quantum advantage using near-term quantum computers. *Variational models* can be thought as hybrid learning that includes both classical and quantum information processing where we outsource some of the computational expensive subroutines to a classical computer while keeping the tasks that only quantum computers can do to the quantum devices.

Similar to the *fault-tolerant* approach, an issue emerges when we have classical data that are being processed by a quantum computer *(CQ)*. More and more people are trying to find real world applications for *Variational model*, and hence, it is apparent that the data encoding problem cannot be overlooked. Encoding classical data into quantum states can be quite challenging as there is no systematic methodology of doing it. In practise, encoding is usually tailored to a specific problem or it is done heuristically. There are a few techniques though for someone to consider, the *Basis encoding*, *Angle encoding* and *Amplitude encoding*.

In the *Basis encoding* scheme, we map our data in the computational basis states. This is convenient when our data are binary in nature because the mapping is straightforward $b_1\, b_2 \cdots b_n \mapsto |b_1\, b_2 \cdots b_n\rangle$. In the case the data are arithmetic but not binary, we can still convert and map them to the basis states. For example, suppose that we have the following classical data with binary representations $x_1 \longrightarrow 01,\ x_2 \longrightarrow 11$. The mapping would be $|00\rangle \mapsto 0,\ |01\rangle \mapsto x_1,\ |10\rangle \mapsto 0,\ |11\rangle \mapsto x_2$.

Amplitude encoding is an alternative approach, where we map our data into amplitude vectors rather than basis states. A quantum state of a system with n qubits has 2^n amplitudes in the state vector and so it can encode 2^n-dimensional

classical data point $\boldsymbol{x}$. The data are encoded to the qubits by applying some unitary operation as shown in Fig.(4.3).

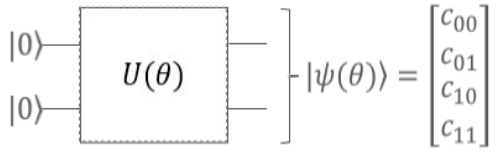


Figure 4.3: Amplitude encoding by applying some unitary $U(\boldsymbol{\theta})$. The $U(\boldsymbol{\theta})$ is such that the output vector matches the classical data $\boldsymbol{x}$.

Angle encoding requires that we have as many qubits as the dimensions of our classical data. The data are encoded by rotating each qubit about some chosen axis by some angle that is dependent on the features in our classical data as shown in the Fig(4.4).

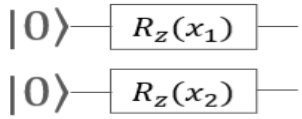


Figure 4.4: Angle encoding by applying some rotation around the z' axis, where the angle of rotation is depended on the classical data.

Besides the encoding process, the way variational models work is quite simple. Firstly, we have a parametrized quantum circuit $U(x;\theta)$, where the data are mapped into quantum states and passed through the circuit. Then a measurement is made at the output state to obtain an expectation value. A cost function is then used to quantify the quality of the estimation, i.e: better prediction will have a lower cost. The optimal parameters are then found by minimizing the cost function via gradient descent or gradient-free optimization methods, and the training process is repeated until the convergence of the cost function. Note that this classical optimization step is outsourced to the classical optimizer to reduce the workload of quantum devices as can see in Fig.(4.5).

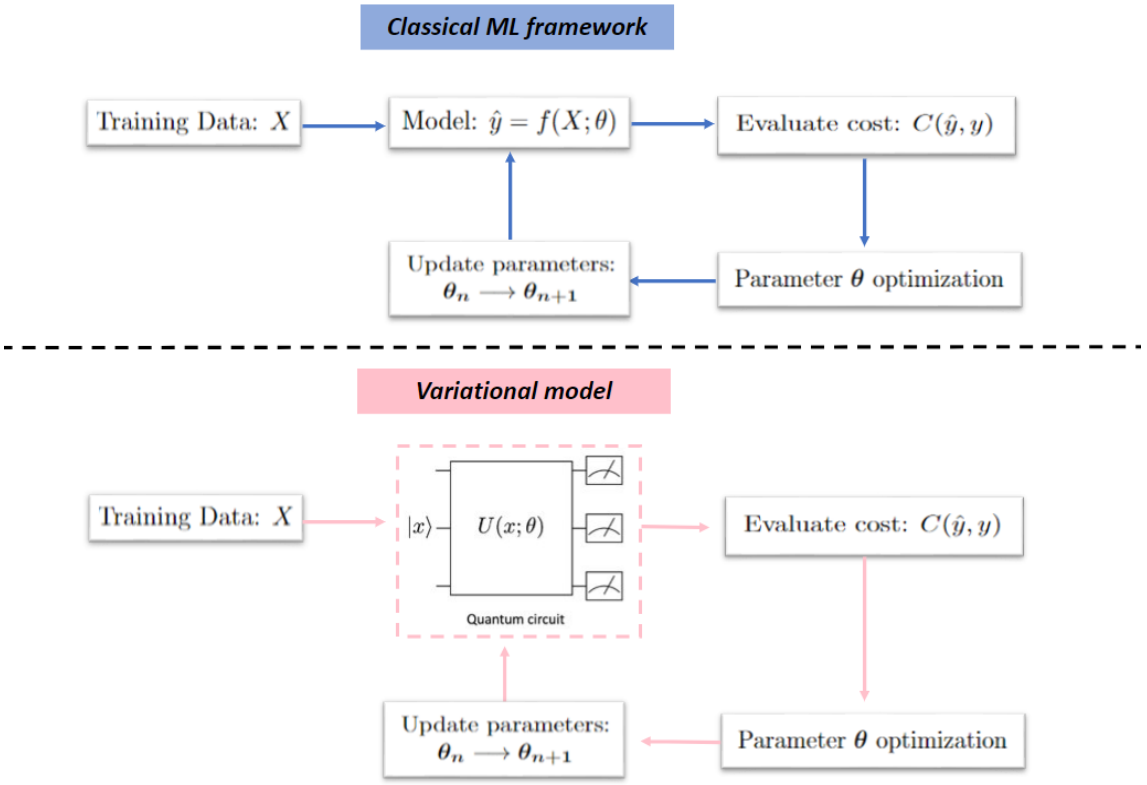


Figure 4.5: Variational quantum-classical approach compared to a classical ML model

The adaptive nature of *Variational models* is well suited to handle the constraints of near-term quantum computers. *Variational models* have now been proposed for essentially all applications that researchers have envisioned for quantum computers as they appear to be the best hope for obtaining quantum advantage.

4.2 Quantum Neural Networks

Research around *Quantum Neural Network (QNN)* models does not have a unified focus, but rather it is a very broad field. Most QNN proposal involve combining classical artificial neural networks with the resources of quantum information in order to develop more efficient algorithms. The motivation behind this research direction lied heavily on the difficulty of training neural networks in Big Data applications.

One such proposal is the Hybrid quantum autoencoders (HQA) [14]. This is a hybrid model that incorporates both classical machine learning, in the form of neural networks, and quantum machine learning, using parameterised quantum circuits (PQC).

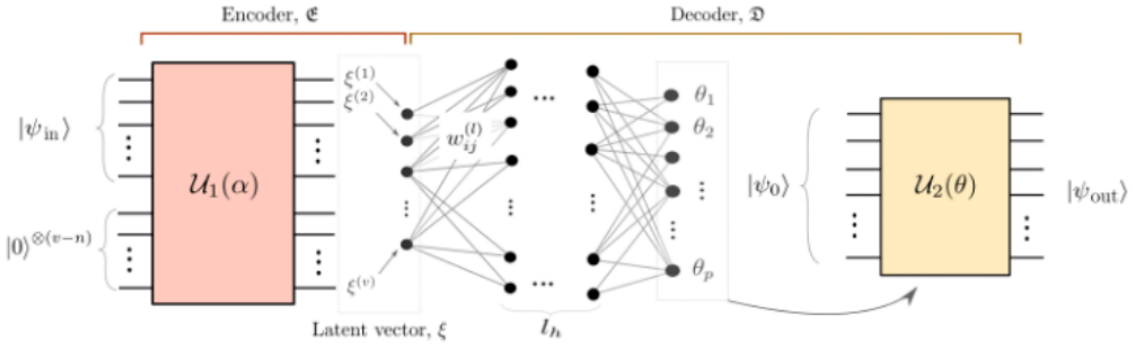


Figure 4.6: Hybrid Quantum Autoencoder (HQA).

Fig(4.6) shows the structure of the model. The encoder and the decoder are quantum in nature. The encoder takes an input state $|\psi_{in}\rangle$ from the $H_2^{\otimes n}$ space and maps it to a subset of the real vector space V of dimension v. The decoder performs the inverse operation. Though the functional form of the encoder and decoder are defined, the models themselves are not specified. The encoder receives some state $|\psi_{in}\rangle$, and applies unitary $U1(\boldsymbol{\alpha})$ on the combined system of the input state with $(v-n)$ ancilla qubits. The latent vector ξ is formed by measuring the qubits on the computational basis. Training of the model changes only the weights of the ANN. The output of the ANN is given as a parameter to the unitary $U_2(\boldsymbol{\theta})$ of the decoder.

Another QNN proposal that is drawing attention is the Quantum Generative Adversarial Networks (QGANs) [15]. This model retains the general structure of classical generative adversarial networks (GANs) but changes the implementation of the generator and/or the discriminator. An example is shown in Fig(4.7).

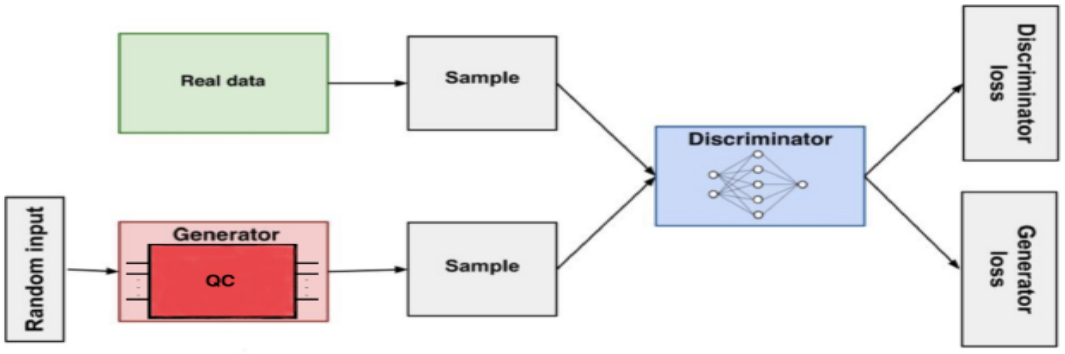


Figure 4.7: Quantum GAN with a generator that is a parametrized quantum circuit and a classical discriminator, which is a classical neural network acting as classifier.

Note that we can replace any classical neural network with a parametrized quantum circuit, as they can be used and trained like neural networks even though they bare only little resemblance with the multi-layer perceptron structure. We can adjust the control parameters of a quantum computer to train it to learn a pattern. The QNN models we will study in this thesis, are based on this premise ,and thus,

are staying true to the *variational model* approach. These models will potentially excel at the hard tasks of unsupervised generative modeling, since they exploit the probabilistic nature of quantum mechanics to learn classical data.

There are a few reasons on why we choose to concentrate on unsupervised generative modeling rather than discriminative supervised task. First of all, they bypass the data encoding issue as what we do here is just performing sampling on the output state and train it to get to some target quantum state. Moreover, it is a well known fact that the classical computers cannot simulate arbitrary quantum systems as the resources required for the simulation grows exponentially with the system size. By turning the argument around, people has conjecture that the quantum computer can learn some classical unlearnable distribution, and this might leads to potential quantum advantage.

The following sections presents and compares the performance of two different approaches, the *digital* and the *analog*. Both behave as quantum analogues of classical neural networks that utilize quantum algorithms to process classical data (CQ). Generally, the data can be either quantum in nature, or classical that ends up encoded as quantum states. The data set that we to chose to test the two approaches is the *Bars and Stripes* data set, which is a classical set of synthetic data.

4.2.1 Quantum Digital Circuit Born Machines

The digital approach to QNNs that we are referring to regards a type of quantum circuits that are called *Quantum circuit Born machines*. The *Quantum Circuit Born Machine (QCBM)* is a recently proposed parametrized quantum circuit model, that is used for unsupervised generative modeling [16, 17, 18]. The term QCBM was originally used to describe quantum wave-functions from tensor networks. For distinguishing purposes, the latter are now referred to as tensor networks Born machines (TNBMs).

Quantum circuit Born machines fall under the umbrella of the more general class of Born Machines. These quantum circuits directly generate samples via projective measurement on the qubits $\mathbf{x} \sim P_{\theta(\mathbf{x})} = |\langle \mathbf{x}|\psi(\boldsymbol{\theta})\rangle|^2$ according to the Born rule. The wavefunction $|\psi(\boldsymbol{\theta})\rangle$ is prepared by applying a parametrized unitary,

$$|\psi(\boldsymbol{\theta})\rangle = U(\boldsymbol{\theta})\,|0\rangle \tag{4.2}$$

Generally, $U(\boldsymbol{\theta})$ is chosen based on the capabilities of the NISQ devices that we have in our possession. As we see in Fig.(4.8), the parametrized unitary $U(\boldsymbol{\theta})$ can be decomposed in a series of adjustable gates, such as qubit rotations gates or parametric coupling gates that generate selective entangling interactions. A benefit of this approach is that these circuits are naturally implemented on a noisy intermediate-scale quantum (NISQ) device. However, we need to bare in mind that the noise in these devices can pose challenges to the training.

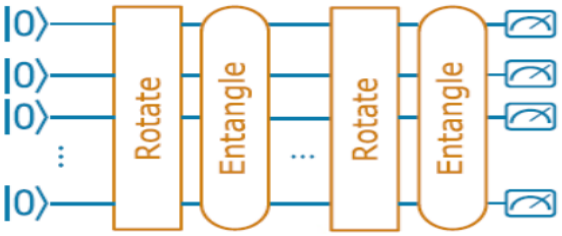


Figure 4.8: The ansatz for the QCBM is divided into layers of parametrized gates, where two types of layers are used. One with *arbitrary single qubit rotations* and one with *entangling gates*. We can stack alternating layers until the QCBM learns the target distribution adequately.

Training a QCBM

Given a dataset D with independent and identically distributed (i.i.d.) samples from a target distribution $P_{\mathcal{D}}$, the goal is to train a QCBM to generate samples close to the unknown target distribution. Quantum circuit Born Machines are trained in the hybrid quantum-classical variational model framework that we mentioned earlier. The model is trained by sampling the output of a quantum computer and updating the circuit parameters using a classical optimizer. The training pipeline is illustrated in the Fig.(4.9) below.

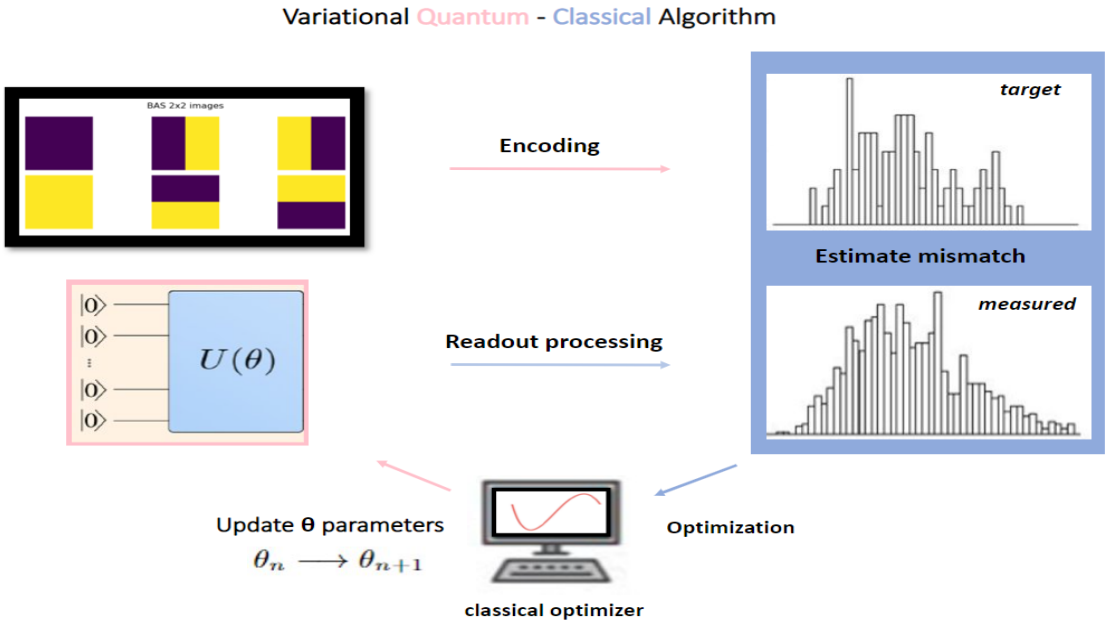


Figure 4.9: Illustration of the QCBM's training pipeline.

At the start, all the parameters of the unitary are initialized with random values. Once the initialization is done, the quantum circuit is executed and sampled M times, in order to reconstruct the state distribution P_θ. A cost function is then used to quantify the estimated mismatch between the target distribution $P_{\mathcal{D}}$ and the measured distribution P_θ. The optimal parameters are then found by minimizing the

cost function via gradient descent or gradient-free optimization methods, and the training process is repeated until the convergence of the cost function. After convergence, the quantum circuit with the optimized parameters produces a wavefunction $|\psi(\boldsymbol{\theta}_{opt}))\rangle$ that encapsulates in the amplitudes of states the probability distribution.

Since we are doing generative modeling, the typical choice of cost function is the *Kullback-Leibler (KL) divergence*, which quantifies how much two distributions differ. KL divergence is defined as,

$$D_{KL}\left(P_{\mathcal{D}}||P_{\theta}\right) = \sum_{x\in\{0,1\}^N} P_{\mathcal{D}}(x)\cdot log\left(\frac{P_{\mathcal{D}}}{P_{\theta}}\right) \tag{4.3}$$

In terms of machine learning, $D_{KL}\left(P_{\mathcal{D}}||P_{\theta}\right)$ is often called the *information gain*. We can think of it as the amount of information gained if we use $P_{\mathcal{D}}$ instead of P_{θ}, or conversely, the amout of information lost when P_{θ} is used to approximate $P_{\mathcal{D}}$.

Nevertheless, while it is a statistical distance, it's not a metric. In mathematics, a metric or distance function is a function that gives a distance between each pair of point elements of a set. In general, KL divergence is asymmetric and does not obey the triangle inequality.

Thus, a better approach would be to instead use a variant of the *negative log-likelihood cost function*,

$$C_{nll} = -\sum_{x\in\{0,1\}^N} P_{\mathcal{D}}(x)\cdot log\left\{max(\epsilon, P_{\theta}(x))\right\}. \tag{4.4}$$

This variant has a quantity ϵ to avoid singularities. A singularity can occur when the estimated probability $P_{\theta}(\mathbf{x})$, of some data $\mathbf{x}$, turns out 0. This would lead the log function to infinity, and hence infinite cost. The ϵ is generally chosen to be very small and close to zero. Typical values are in the range of $\epsilon = 0,001$ or $\epsilon = 0,01$.

The Bars and Stripes dataset

In this thesis, we considered a specific dataset known as *Bars and stripes* (BAS) dataset. *Bars and stripes* (BAS) is a set of images that has been widely used to study generative models. The images that belong to the dataset, are a subset of $n\times m$ black and white pictures. Each image consists of any number of either horizontal stripes or vertical bars. From simple combinatorics it is easy to derive that for $n\times m$ pixels, there are NBAS(n,m) $= 2^n + 2^m - 2$ images. The simpler case that we can study is the NBAS(2,2) which is illustrated in the figure below.

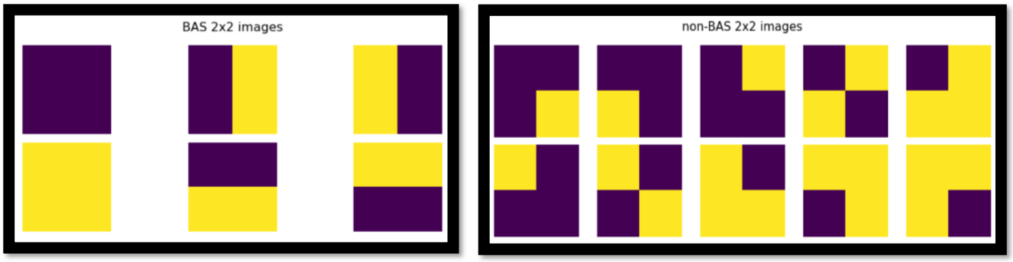


Figure 4.10: The NBAS(2,2) data set.

Each image in the BAS data set can be viewed as a vector $\mathbf{x}$ of $N \times 1$ dimensions, where $N = n \cdot m$. The elements of the vectors correspond to the pixels of the images taken from top to bottom, and each row from left to right. Furthermore, since the images are black and white, the vectors are binary vectors $\mathbf{x} \in \{0,1\}^N$. The N-dimensional binary vectors $\mathbf{x}$ allow us to make a one-to-one mapping with the computational basis of the N-qubit quantum systems. That is $\mathbf{x} \leftrightarrow |x_1 x_2 \cdots x_N\rangle$. This method of encoding classical data into quantum states is called *Basis encoding*. In the case of the *Bars and Stripes* dataset with 2×2 images, we can directly map the images into a 4 qubit system as we see in the figure below.

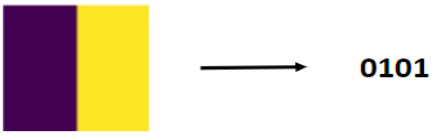
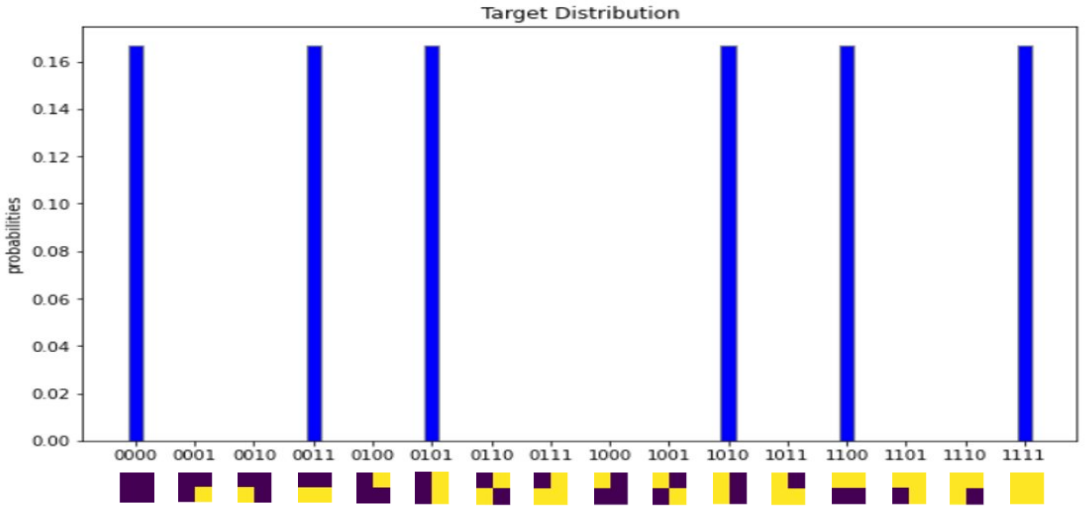


Figure 4.11: Encoding classical data into quantum states.

The data set $D = (\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(6)})$ of the NBAS(2,2) images consists of 6 *independent* and *identically distributed* images. For simplicity we assume that they are uniformly distributed,



Using a Quantum Circuit Born Machine to learn a data set

In this section, we will use the QCBM as a digital quantum generative model to learn the BAS dataset. We considered three different ansatz for the QCBM: Chain, star and all-to-all topology Fig(4.13). The ansatz for the QCBM is divided into layers of parametrized gates, where two types of layers are used. One with *arbitrary single qubit rotations* and one with *entangling gates* between two qubits. The layer of single qubit rotations involves three rotation gates, that are applied to each qubit i, specifically,

$$U^{(l)}(i) = R_z^{(l,i)}(\alpha_i) \cdot R_x^{(l,i)}(\beta_i) \cdot R_z^{(l,i)}(\gamma_i) \tag{4.5}$$

where l is the layer index. Each arbitrary rotation layer requires 3N trainable parameters, where N is the number of qubits. An important thing to note here is that we can remove the first Z rotation in the first layer, as indicated by the dashed box in the image below, since the circuits are executed from an initial $|0\cdots0\rangle$ state. This would reduce the number of parameters at the first layer to 2N.

Figure 4.12: Arbitrary rotations applied on a single qubit.

Moreover, should we use an odd number of layers, we can reduce slightly further the number of parameters. In this case, the last set of R_Z rotations would only add a phase, which is not measurable when we square the amplitudes with the Born rule. This reduction in parameters is important, because the more parameters we have, the more complex and hard the optimization process gets.

The entangling layer applies entangling gates between two qubits. The type of gates used in this layer is dependent on the experimental platform the we have on our hands. In previous implementations on trapped-ion quantum computers the entangling layers consisted of Mølmer-Sørensen gates,

$$XX(\theta_{ij}) = exp\left(-\frac{i}{2}\,\theta_{ij}\; X_i \otimes X_j\right) \tag{4.6}$$

$$= \begin{pmatrix} \cos\left(\frac{\theta_{ij}}{2}\right) & 0 & 0 & -i\sin\left(\frac{\theta_{ij}}{2}\right) \\ 0 & \cos\left(\frac{\theta_{ij}}{2}\right) & -i\sin\left(\frac{\theta_{ij}}{2}\right) & 0 \\ 0 & -i\sin\left(\frac{\theta_{ij}}{2}\right) & \cos\left(\frac{\theta_{ij}}{2}\right) & 0 \\ -i\sin\left(\frac{\theta_{ij}}{2}\right) & 0 & 0 & \cos\left(\frac{\theta_{ij}}{2}\right) \end{pmatrix} \tag{4.7}$$

where X is the Pauli gate and θ_{ij} is a parameter.

In this work, we will execute the circuit on the superconducting quantum processors offered by IBM. The quantum processors are available to the public through

the cloud. In this technology, the Mølmer-Sørensen gates, that are native only to trapped-ions, are replaced by R_{XX} gates,

$$R_{XX}(\theta_{ij}) = exp\left(-\frac{i}{2}\,\theta_{ij}\; X_i \otimes X_j\right). \tag{4.8}$$

Notice that a R_{XX} gate has the same mathematical behaviour as a Mølmer-Sørensen gate. The only difference is the experimental platform. As for the complexity of this layer, the number of entangling gates differs, depending on the conectivity we want to implement.

Before we proceed to numerical simulations, we also need to decide which optimization algorithm we will use in order to find the parameters that minimize the said cost function. We opt to go with the the *Particle Swarm Optimization* (PSO) implemented by the PYSWARMS library [19]. This algorithm has been used for similar simulations with very good results.

PSO is a gradient-free optimization method that imitates the social behaviour of animals. Particles, where each one represents a possible solution, are scattered around the search-space. Each particle moves around in the parameter-space searching for an optima. The movement of each particle is affected by both it's *current performance* and the *performance of the swarm*. In our problem, each particle represents a point in the parameter-space of the quantum circuit. The performance of the optimization is affected by the number of particles, as well as their position and velocity. The value of these parameters are usually determined empirically. There are three hyper-parameters controlling the swarm dynamics: a cognition coefficient c_1, a social coefficient c_2 and an inertia coefficient w. After testing different values, we conclude that setting $c_1 = c_2 = 1$ and $w = 0.5$ works well for our case. We also set the number of particles to twice the number of the parameter's of the circuit.

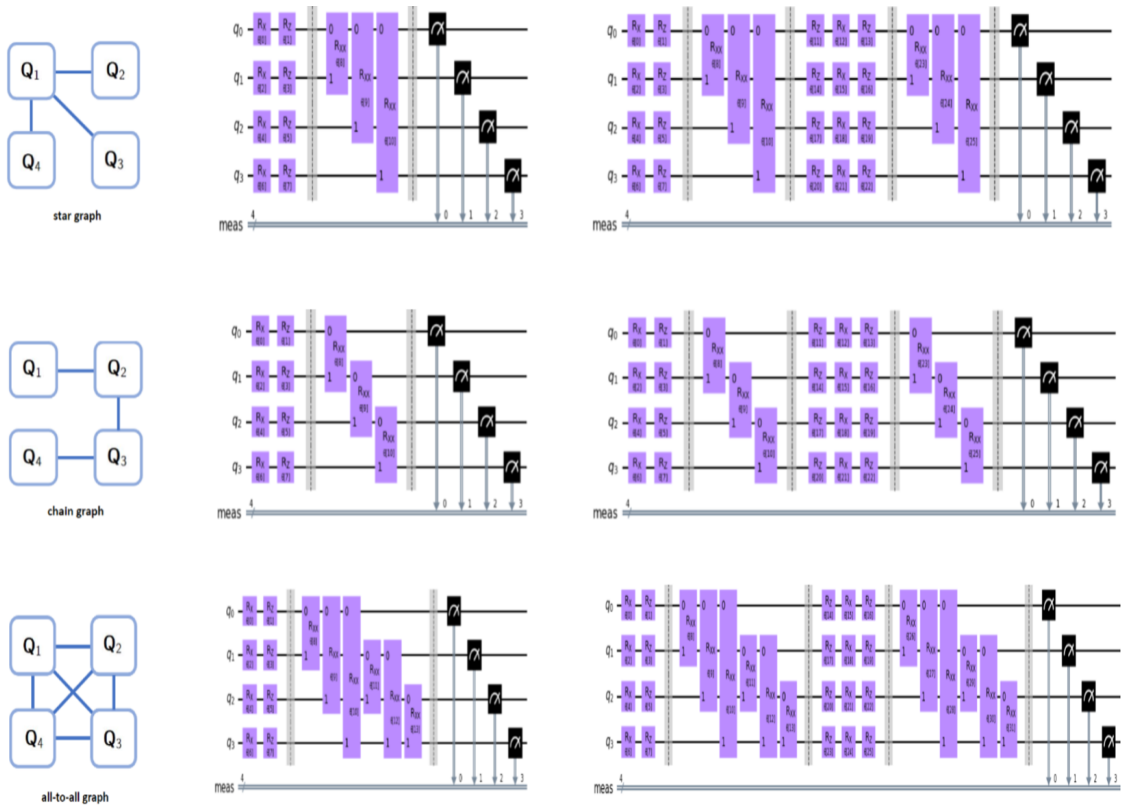


Figure 4.13: Three different ansatz of the QCBM. On the left column we can see the connectivity graph of each ansantz, while middle column and right column illustrate the QCBMs with 2 and 4 layers, respectively.

For the QCBM, we will perform numerical simulation on a classical computer for all three topologies, and for different number of layers. We then select the best model and implement it on the IBM quantum device to see how the generator model perform under a realistic setup. Note that the training process is done classically and only the final optimized parameters are implemented on the real hardware. The numerical simulations are done using *Qiskit*, which is an open-source software development kit created by IBM [20]. It provides tools for creating and manipulating quantum programs and running them on prototype quantum devices on IBM Quantum Experience or on simulators on a local computer.

As Particle Swarm Optimization (PSO) method is sensitive to the initial "seed" values of the particles, so we simulate for many different random seeds and keep the one with the best performance. In practise, we don't know which seeds will yield a good result, and thus, we need to try different seeds. This slightly increases the computational cost, since we need to repeat the training process a few times. The results from the PSO optimization as well as the distributions generated by the circuits can be seen in Fig.(4.14) with the training cost history shown in Fig.(4.15).

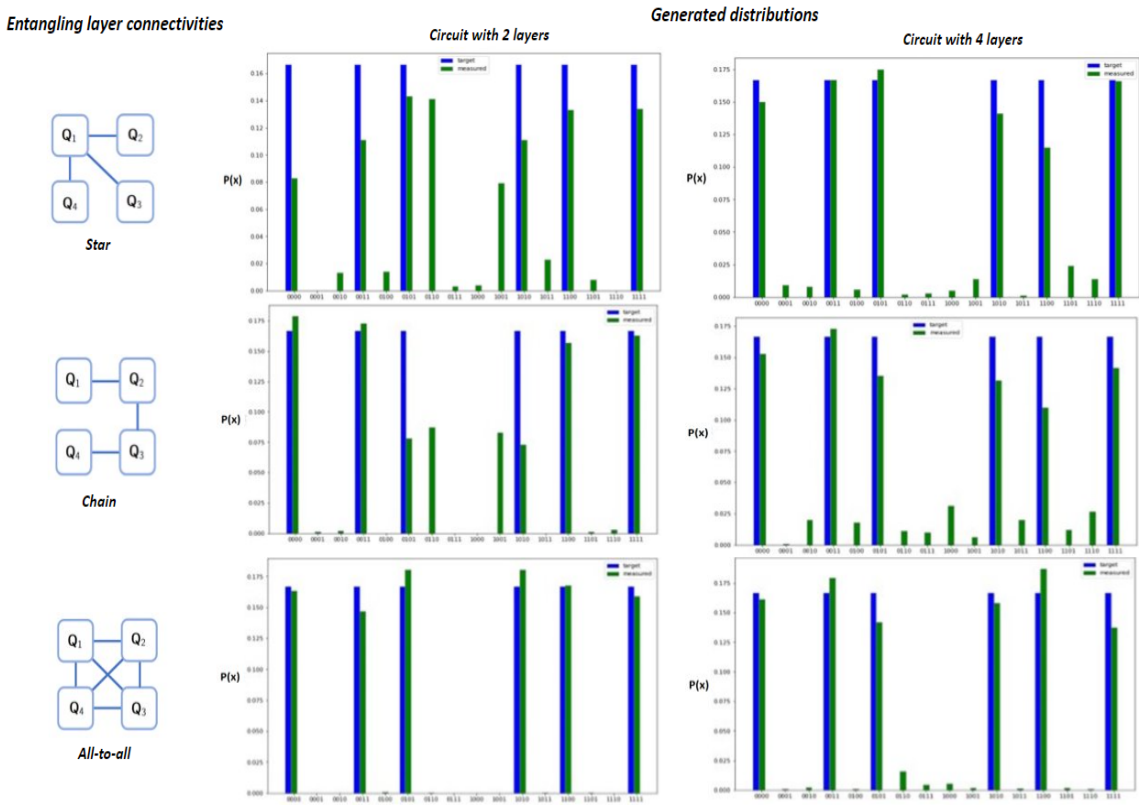


Figure 4.14: Distributions generated by the circuits. To create the distributions each circuit was sampled 1000 using the Born rule. The target distribution is the blue, while the measured distribution is the green.

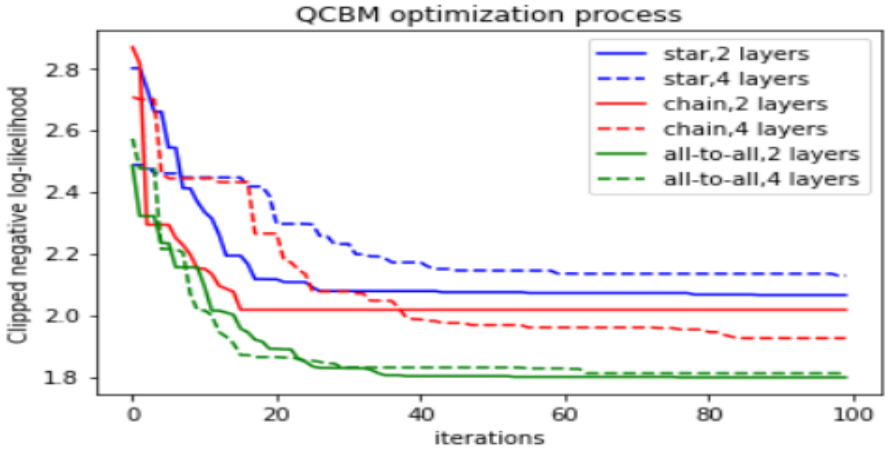


Figure 4.15: The graph shows the minimization of the clipped negative log-likelihood cost function for each circuit.

For the all-to-all connectivity, one rotation layer and one entangling layer suffice in catching the patterns and correlations of the dataset. That's not the case for the star and chain connectivities though. Quantum circuits with star and chain topologies demonstrate better accuracy with the addition of another pair layers, one

rotational and one entangling. Yet it still seems that these connectivities are not complex enough to represent the dataset. Regardless of the connectivity, we stack alternative layers of arbitrary qubit rotations and entangling layers until the QCBM can reproduce the target dataset adequately. For this work, we limit ourselves to 4 layers. Deeper circuits oftentimes come with an excess number of parametrized quantum gates that enhance expressivity but can compromise trainability as well as create a model that strongly overfits training data.

Execution on an IBM quantum processor on the cloud

At the moment, quantum computers are not yet commercial. A few prototypes are being developed by some big companies, such are Google or IBM, and some elite universities around the world,including the Centre for Quantum Technologies where the author had a chance to spend a few weeks during the work on this thesis. IBM has launched since 2016 an online platform, the *IBM Quantum Experience*, that allows the public access to cloud-based quantum computing services. This includes access to a set of IBM's prototype quantum processors, six of which are freely available for the public.

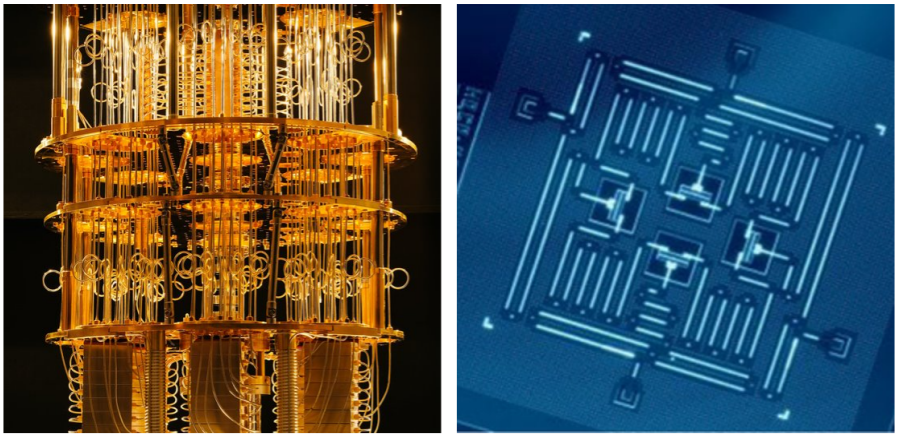


Figure 4.16: IBM's quantum processors are made up of superconducting qubits, located in dilution refrigerators at the IBM Research headquarters at the Thomas J. Watson Research Center. At the left we can see the dilution refrigerator that cools down qubits at below 15mK (milli-Kelvin). On the right we can see a quantum chip with four superconducting qubits.

From the simulations we can clearly see that the circuit with the best recreation of the Bars and Stripes distribution, is the circuit with the all-to-all qubit connectivity in the entangling layer. Shallow quantum circuits are generally more preferable for NISQ quantum devices, as deeper circuits can significantly decrease the *fidelity* of the quantum states due to quantum decoherence.

Fidelity, in quantum information theory, is a measure that quantifies how close are two quantum states. Specifically, it describes the probability that one state will pass a test to identify as the other. In quantum computation, we use fidelity to

express the quality of states that a quantum gate generate. Unfortunately, even the state of the art quantum gates do not have a fidelity of 1. Meaning that the more gates we apply, the more decoherence we add to the system and we risk a complete loss of information. Until fault-tolerant quantum computing is on our hands, one should take into account the fidelity of the gates that a specific quantum computer offers when designing the depth of the circuit.

Therefore, since longer circuits are subject to greater noise, we are focusing only on the shallow circuits. If the performance of the shallow circuit turns out bad, we can directly infer that the result for the longer circuit will also be bad. The ansatz that we choose to implement on a real IBM quantum processor in the cloud is the all-to-all topology, as it is the best performing among all three topologies. Note though that we only implement the optimal parameters on the IBM's device. The training is done on a classical computer.

Nonetheless, before running the circuit on a real device there is a few things we need to consider. First of all, some of the gates that we want to apply may not be available at the quantum device. Meaning that the gate has to be decomposed to a series of gates that would overall produce the the same effect. Secondly, we have to convert one set of gate operations to another set of gate operations, since the qubit connectivity, i.e. the geometry of the architecture, varies from one quantum computer to the another.

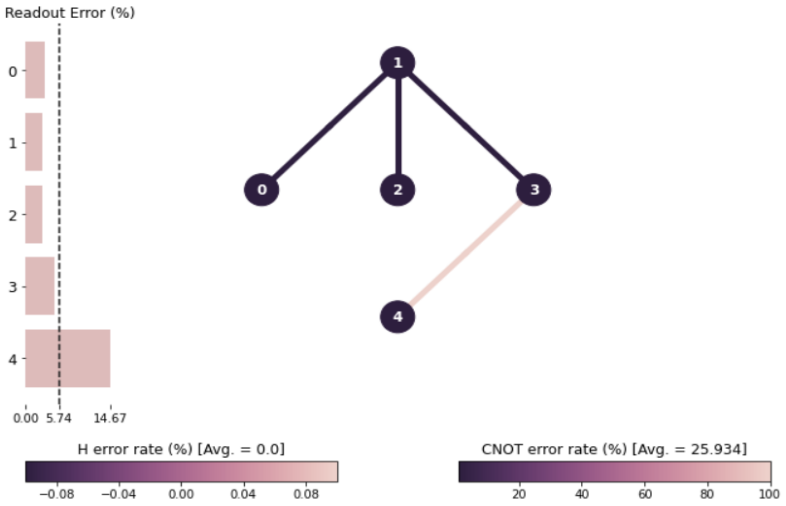


Figure 4.17: IBMQ-Belem quantum processor. We can see the allowed qubit interaction connectivities along with the fidelity of gates and the readout errors of the chip.

Fig.(4.17) shows the connectivity of the IBMQ-Belem device where the qubits are not all-to-all connected. Implementing the all-to-all topology on this device will therefore requires the utilization of lots of swap operators. Additionally, R_{XX} gates are not available and thus they are decomposed into a series of native gates of the hardware, which as a result increases the depth of the circuit. The illustration in

Fig.(4.18) bellow demonstrate the discrepancies between the theoretical model we wanted to implement and the actual circuit on the quantum computer.

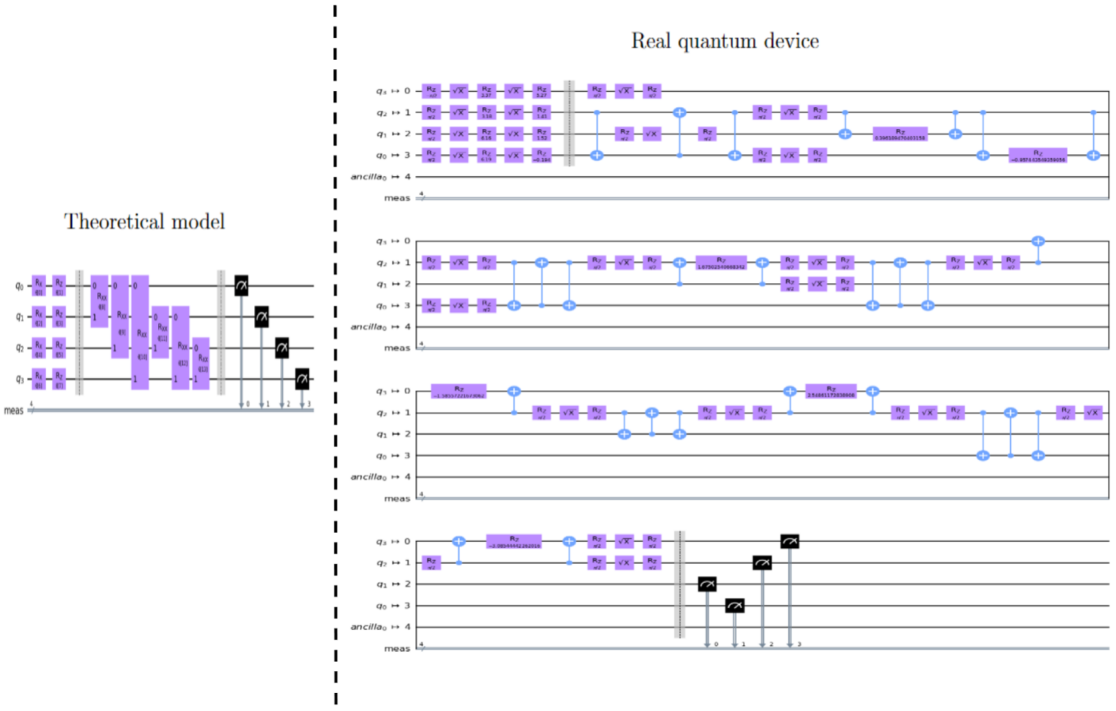


Figure 4.18: Comparison between the theoretical model and the one that is implemented on the IBM quantum computer.

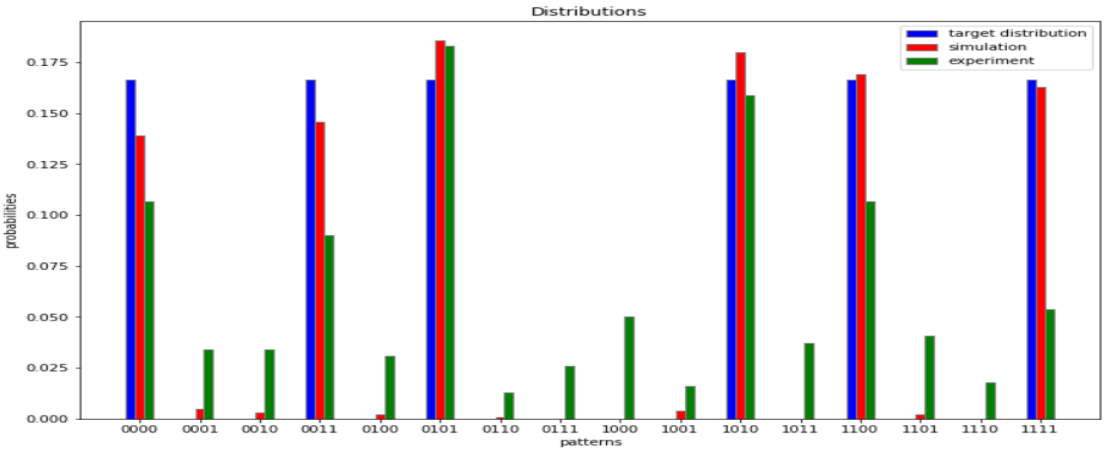


Figure 4.19: QCBM training results with PSO, with simulations (red) and IBM superconducting quantum computer (green).

After executing the circuit on the quantum computer it is obvious that noise is prevalent and messes with the results. The poor results compared to the simulation can be attributed to various reasons. As we said simulated circuits may use gates that are *not native* to the real device. Each of these gates are decomposed to a

series of native gates that results to the *increase of the circuit depth*. The *fidelity of the gates* is not perfect, and thus, for each gate the accuracy drops! In addition to that, the theoretical model considers the quantum device a closed quantum system. However, the existing quantum devices are *susceptible to disturbance and noise from their environment*, which can lead to *loss of information* from a system into the environment. This phenomenon is amplified by the fact that the equipment that are used to control the qubits are imperfect, causing errors such as the *initialization error* or the *readout noise*.

It is important to note that this noisy results are expected and shouldn't be discouraging us from continuing to work with NISQ. Working with a quantum computer that allow us to implement the connectivity we need is essential to reducing the errors and noise. Previous implementations of this problem in an ion trap quantum computer has shown very promising results for the NISQ-era and the abilities of variational models.

4.2.2 Analog Quantum Machine Learning

The random parametrized quantum circuits can express complex distributions that are intractable by a classical computer. Although the digital quantum generative models are performing well in the noiseless simulation, they under-performed in the real hardware because of the noise that is prevalent in the NISQ devices. Adding to that, should we want to increase the learning capabilities of a QCBM, we are required to increase the number of layers of the quantum circuit. The Bars and Stripes dataset is a simple dataset that needed only two layers. However, more complex problems require a much larger number of gate operations, which exceeds the capabilities of NISQ devices without error correction.

For the near-term quantum computing, an alternate and more 'native' approach would be to built a scale model of the system in an analogue quantum simulator Fig(4.20). The major advantage is that these analogue simulators can readily scale to large system sizes, making them a natural frontier to search for a practical quantum advantage. Analogue quantum simulators require less precise control and are more easier to realize than the digital quantum computer. The model is built to simulate the rules that describe physical microscopic quantum systems and thus we can tune the physical control parameters, such as an electromagnetic field strength or a laser pulse frequency, to simulate the system. The downside is that they are tailor made to specific problems, i.e: non fully configurable like a digital quantum computer, and therefore different quantum simulators need to be built for different problems. There are various experimental platforms that can be used for analogue quantum simulations, such as *trapped atoms*, *superconducting circuits*, *photonic arrays* and *spin systems*. In this thesis we will utilize the Ising spin systems for our analogue quantum neural network, as the topology we considered is native to the Ising model. Also, we are considering the model to be a closed quantum system. When the system is open then its means noises kick in. In principle open quantum systems can also be used to do quantum simulation.

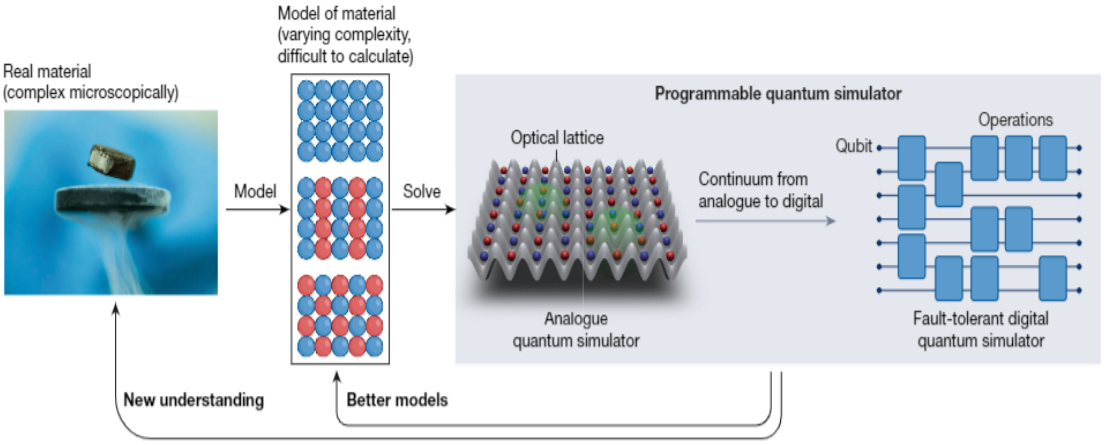


Figure 4.20: Problems could be solved on a fault-tolerant digital quantum computer or we can build a scale model of the problem in an analogue quantum simulator [21].

Evolution of closed quantum system

To understand how we can achieve calculations with such systems, we first need to understand their behaviour. Similar to classical mechanics and the laws of motion, quantum mechanics has equations that fully describe the evolution of a quantum state in time. While this is not surprising, what may come as a surprise is the fact that the evolution is *deterministic*. The basic postulation is that if we know the quantum state at one time, we can calculate what the state will be in a later time. But conversely to classical mechanics, knowing the quantum state does not imply that we can predict the outcome of an experiment with certainty. So, it is not the same as classical determinism.

The quantum counterpart of the classical laws of motion is the famous *Schrödinger equation*, a linear partial differential equation that describes the behaviour and evolution of an undisturbed quantum system. The most general form of the *Schrödinger equation* is defined as,

$$i\hbar\frac{\partial}{\partial t}\left|\psi(t)\right\rangle = \hat{H}\left|\psi(t)\right\rangle . \tag{4.9}$$

In this equation $\hbar$ is a physical constant, called the Plank constant. In practise it can be absorbed in the $\hat{H}$ when doing calculations and hence it's exact value is not important to us here. The operator $\hat{H}$ is a Hermitian operator called the *Hamiltonian*. The Hamiltonian describes the dynamics of the system as its eigenstates are the possible energy states that the system can occupy. If we know the Hamiltonian, the solution of the Schrödinger equation tells us how the state of an undisturbed system evolves with time.

Yet, figuring out the Hamiltonian that completely describes a system is considered a very difficult problem in general. In this work we will confine ourselves in the simple case of two-level spin systems. Spin systems as a two-level system has the

advantage that are simple to explain and can be transformed and implemented on other experimental apparatus that are based on two-level systems.

But before we say anything about spins, there are more things to unpack from the *Schrödinger equation*. The notation $|\psi(t)\rangle$, besides refering to the state $|\psi\rangle$ at time t, it also suggests that the state is changing over time. Hence, $|\psi(t)\rangle$ can be thought of as an expression of the system's entire history. Solving the *Schrödinger equation* will reveal to how two states of the the system, from different points in time, are connected with each other. Without loss of generality, we can consider the initial time as zero and the latter time as t. This yields,

$$|\psi(t)\rangle = e^{-i\cdot\hat{H}\cdot t/\hbar}\cdot|\psi(0)\rangle = U(t)\cdot|\psi(0)\rangle \tag{4.10}$$

where $U(t)$ is unitary operator known as the *evolution operator*. There is something important here to unpack. We defined the action of the quantum gates as unitary operations. Now, quantum mechanics indicates that unitary operations are just representations of the systems evolution in time under some specific dynamics. Meaning that what we perceive as quantum gates on the algorithmic level, is just the controlled evolution of the quantum system. Now we are in a position to compare the analog approach with the circuit approach. The quantum circuit is as one big parametrized unitary operation that is implemented on a higher level of the quantum computer's architecture, while the analog approach goes into the hardware and tries to implement the same unitary by tuning directly the physical control parameters of the system (Fig(4.21)).

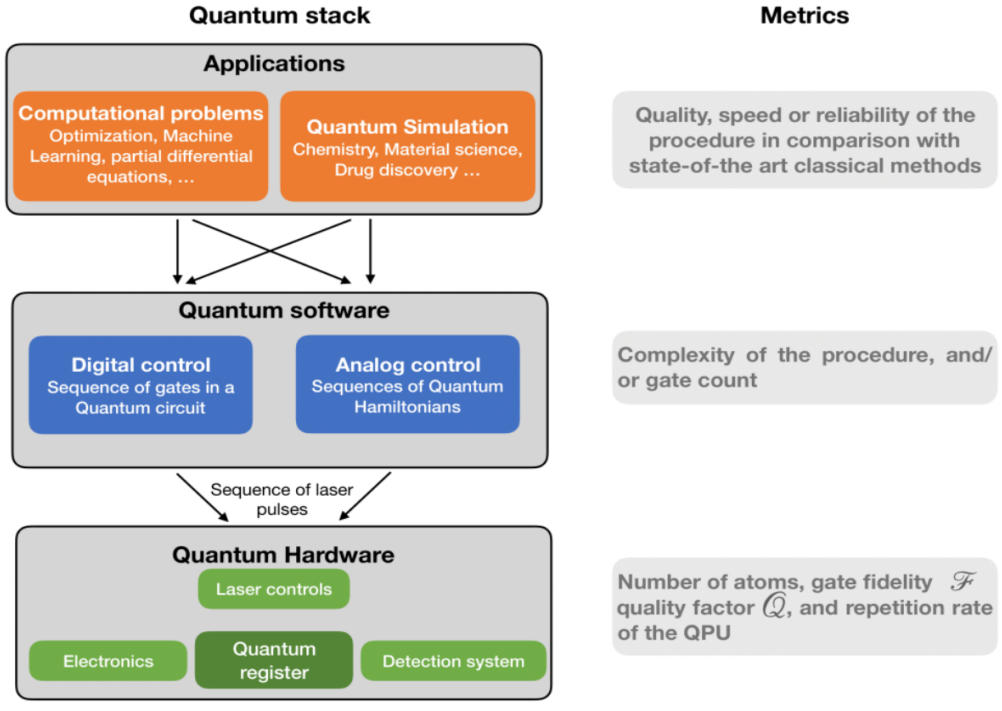


Figure 4.21: General architecture of a quantum computer.

Spin system - the simplest possible qubit

To understand which are the physical parameters of the system that we can adjust to achieve learning we need to explain the basics of *spin systems*. Spin is an intrinsic form of angular momentum that is carried by elementary particles. Electrons belong in a class of elementary particles, called *fermions*, that have a spin of $\frac{1}{2}$. The spin is commonly described as the electron spinning around its axis. It is symbolized as $\vec{S}$ and is mathematically described as a vector with definite values. Quantum mechanics states that the component of spin angular momentum of an electron measured along any direction is,

$$\vec{S} = \hbar \, s \, \vec{\boldsymbol{\sigma}} \tag{4.11}$$

where $\vec{\boldsymbol{\sigma}} = (\sigma_x, \sigma_y, \sigma_z)$ is the spin operator and s is the spin quantum number. The electron as a fermion has a only two possible spin values, 1/2 and -1/2.

An electron in a magnetic field has a magnetic dipole moment, just like a rotating electrically charged body in classical electrodynamics. The magnetic dipole moment $\vec{\mu}$ is proportional to its spin angular momentum $\vec{S}$,

$$\vec{\mu} = \gamma \vec{S} \tag{4.12}$$

where γ is the *gyromagnetic ratio* of the electron.

Thus, the energy of the system, which is associated with the torque of the magnetic dipole moment $\vec{\mu}$ in a uniform magnetic field $\vec{B}$, is proportional to the dot product of the spin and the magnetic field,

$$\hat{H} = -\vec{\mu} \cdot \vec{B} = \gamma \vec{S} \cdot \vec{B} = \frac{1}{2} \, \hbar \, \gamma \, \vec{\boldsymbol{\sigma}} \cdot \vec{B}. \tag{4.13}$$

Concisely, the Hamiltonian for a spin in a magnetic field can be expressed as,

$$\hat{H} = \vec{\boldsymbol{\sigma}} \cdot \vec{\tilde{B}} \tag{4.14}$$

where $\vec{\tilde{B}} = \frac{1}{2} \, \hbar \, \gamma \vec{B}$. Notice that the only thing that is not a constant is the amplitude and the direction of the magnetic field B.

In the *Ising model* case, we have many spins that interact under an external magnetic field. The spins are arranged in a graph, allowing each spin to interact with its neighbors. The *Ising model* is the simpler interaction model we can explore as the spins interact only on the z'z axis. For any interaction pair i, j, between two spins in the graph, there is a coupling constant J_{ij} that determines the strength of the interaction. The energy of a given configuration is thus given by,

$$\hat{H}_{z'z} = \sum_{i=1}^{n-1} \left(\sum_{j=i+1}^{n} J_{ij} \cdot \sigma_i^z \otimes \sigma_j^z \right) \tag{4.15}$$

where n is the number of spins in the graph. The complete dynamics of the system are described if we add in the factors that regard the interaction of each spin with the external magnetic field B,

$$\hat{H} = \sum_{k=1}^{n-1} \vec{\boldsymbol{\sigma}}_k \cdot \vec{\tilde{B}}_k + \sum_{i=1}^{n-1} \left(\sum_{j=i+1}^{n} J_{ij} \cdot \sigma_i^z \otimes \sigma_j^z \right) \tag{4.16}$$

Spin systems can be a good and simple implementation setup for quantum computation. We can think of a qubit as an electron in a magnetic field. The electron's spin may be either in alignment with the field, which is known as a spin-up state $|0\rangle$, or opposite to the field, which is known as a spin-down state $|1\rangle$. This mapping between spins and qubits can be seen in the Fig.(4.22) below. The internal states of an ion can be considered as a spin, and the same holds for the two circulating states in a superconducting circuit. Similar quantum hardware implementation are now in the forefront of technological developments. IBM, Google, Riggetti as well as several leading labs in EU, China and US are using superconducting qubits, where as ION Q, Inssbruck Austria, Honeywell and others are using ions. There are also photonic efforts by several groups and recent multi million dollar funded spin offs like PSI Q and Xanadu, where the photons play the role of the qubit.

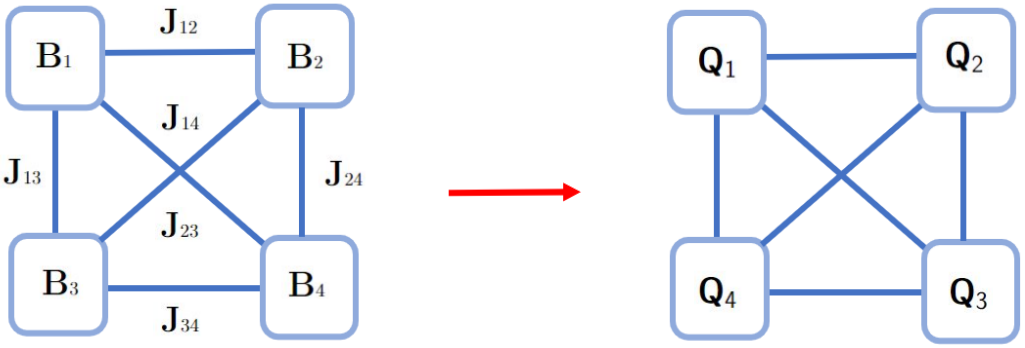


Figure 4.22: A spin is a two-state (or two-level) quantum-mechanical system so it can be utilized as a qubit.

Training and Numerical Simulations

Analog quantum systems can be used as an unsupervised generative model in the *Variational model* framework. Similar to the quantum circuit approach, the information is encoded to the amplitudes of the wavefunction and the statistics are then extracted using the *Born* rule. The difference is that to match the system's distribution to the target, we need to find the amplitudes and directions of all the magnetic fields B, all the coupling constants J and the time t we need to let the system evolve. Note that the Hamiltonian of the system in question is time-independent, and thus, does not change while the system evolves in time.

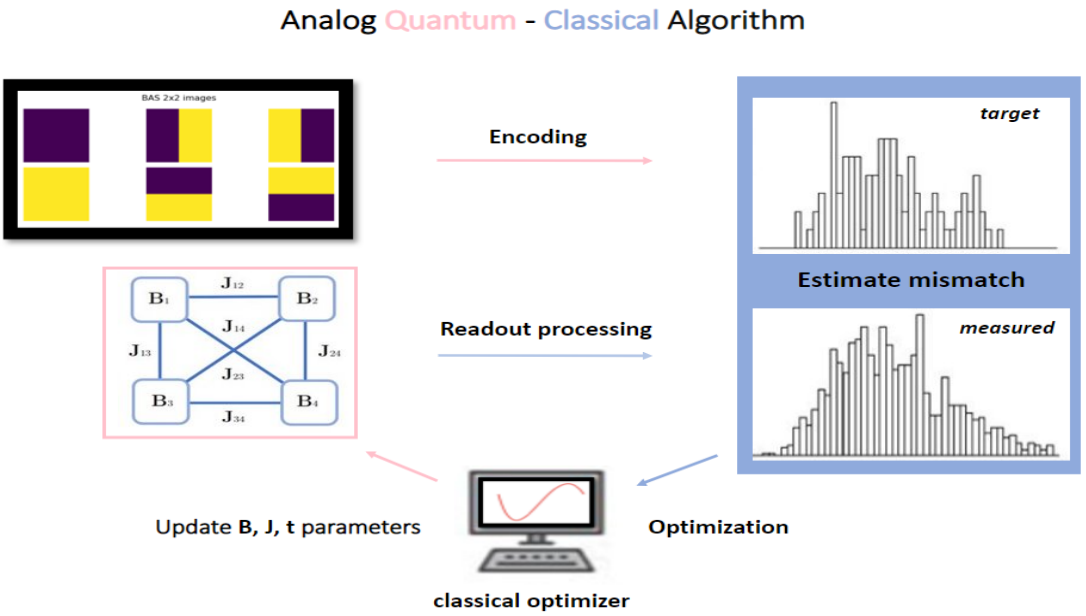


Figure 4.23: Generative modelling using an analog quantum system.

We choose the initial state to be $|++++\rangle$, as we found empirically that it yields better optimization than $|0000\rangle$ or $|1111\rangle$. Furthermore, we considered to have our optimization process bounded. This approach has a few benefits. First of all, bounding the optimization process will make it more stable as it will constrain the PSO from doing large value "jumps" when searching the parameter space. Secondly, by setting thresholds of [-50 MHz, 50 MHz] for J, B and evolution time up 300 ns, we can later implement our system in real hardware. Previous experiments have shown that these values can be applied on a real experimental setup [22]. In the classical simulation, we adapted the re-scaled Hamiltonian $\bar{H} = H/c$ and the re-scaled time $\bar{t} = ct$, where $c = 7 \cdot 10^6$. These re-scaled quantities will give us the same unitary evolution, i.e: $U = e^{\frac{-i\bar{H}\bar{t}}{\hbar}} = e^{\frac{-i(H/c)\cdot ct}{\hbar}} = e^{\frac{-iHt}{\hbar}}$, where $H(\vec{B}, \vec{J})$ is the Ising Hamiltonian in Eqn.(4.16) while $H(\vec{\bar{B}}, \vec{\bar{J}})$ is the re-scaled Hamiltonian. This was done so as to avoid numerical errors and to make the optimization less volatile to small changes in the parameters.

The distribution generated by the Ising spin system can be seen in Fig.(4.24) with the physical control parameters of the system shown in Fig.(4.25).

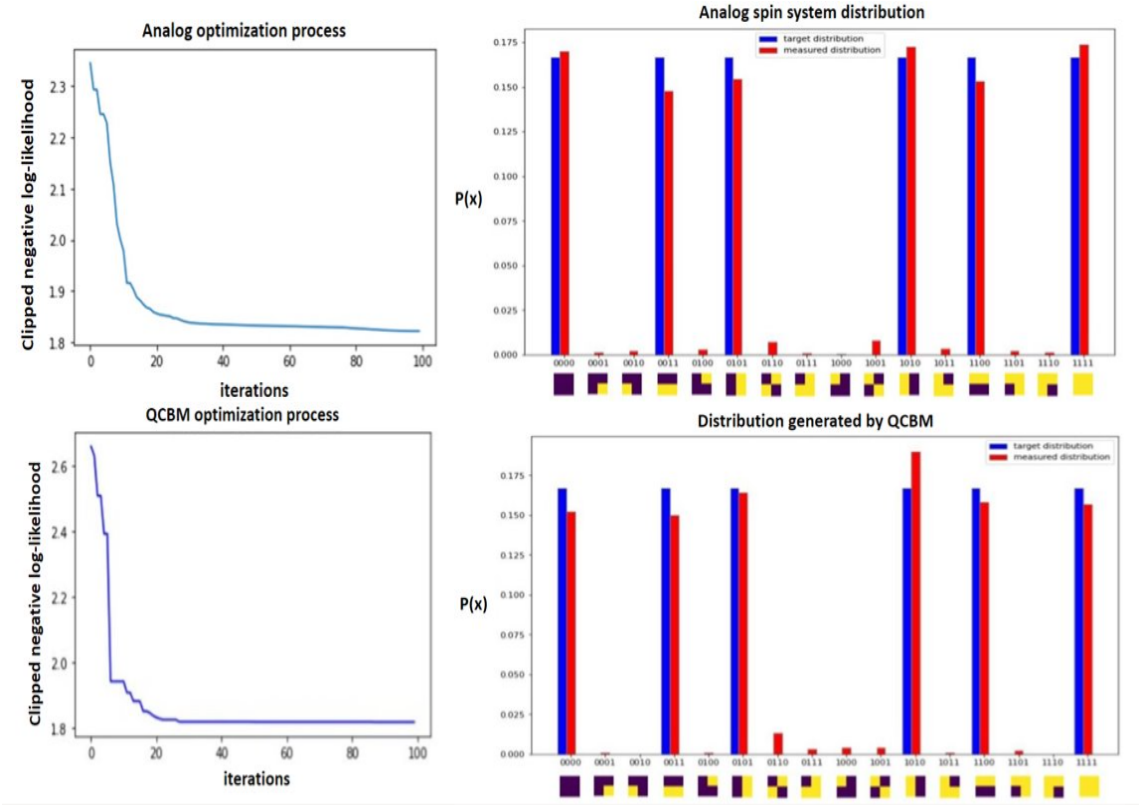


Figure 4.24: On the left we can see how the cost function is minimized during the optimization process. On the right we see the distributions generated by both models after 100 iterations. Blue is the target distribution and red is the measured distribution.

B(MHz)		J(MHz)		t(ns)
$B_{z'z\_1}$:	20.12	J_{12}:	5.313	188.77
$B_{z'z\_2}$:	23.317	J_{13}:	24.224	
$B_{z'z\_3}$:	-15.176	J_{14}:	-21.25	
$B_{z'z\_4}$:	15.724	J_{23}:	1	
$B_{x'x\_2}$:	-24.879	J_{24}:	1.28	
$B_{x'x\_2}$:	-25.654	J_{34}:	23.158	
$B_{x'x\_3}$:	28.367			
$B_{x'x\_4}$:	-27.894			

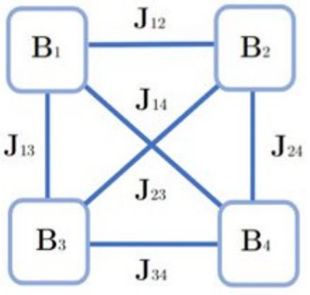


Figure 4.25: The all-to-all topology for Ising models and the corresponding trained parameters for the generative modelling on the Bar and Stripes example. These parameters are within the physically implementable range.

As we have seen from the results of the tests, the simulations of both models are pretty good. This is to be expected as we implement the same model, that is

described by the same dynamics, with the only difference being the level of implementation. The next step would be to implement the analog model on real experimental platform and see how close to the simulations we can go as decoherence, noise and calibration errors would take effect. The ultimate limit for analogue quantum simulation is set by the decoherence timescale, which provides an upper bound on the timescale on which we can controllably observe coherent quantum dynamics.

Conclusion and Future Work

The quantum computation revolution has already begun. What started as an brainchild of the genius physicist Richard Feynman back in 1981, has now materializing and open endless new possibilities that we can't help but be excited about. In the past decade, quantum algorithms such as Shor's and HHL's algorithms, with computational "advantange" compared to their classical counterparts have been proposed. During the course of this thesis, we saw that quantum computers have, in theory, the ability to tackle the same tasks as many classical algorithms, but with a significantly less computational cost. This was the premise that ignited the interest on quantum computers in the 21st century.

Realizing these quantum algorithms, however, requires fault-tolerant quantum computers with millions of high quality qubits to implement error-correcting codes that prevent errors to corrupt the results, ensuring the quantum computers can process quantum information reliably and smoothly. Yet, the many engineering challenges of creating these fault-tolerant quantum computers still need to be tackled in order to convert dreams into reality. What is available now is the Noisy-Intermediate Scale-Quantum (NISQ) devices: the noisy version of quantum computers with up to hundreds of qubits. *Chapter 3* introduced a special hybrid quantum-classical algorithm dedicated to the NISQ devices as this algorithm reduces the quantum resources by passing some computational workload to classical computer. In this work, we show the potential of using this hybrid quantum-classical algorithm to solve some unsupervised machine learning problems.

Specifically, we compared the performance of digital and analog quantum neural networks (QNN) approaches in solving an unsupervised machine learning task, i.e: to learn the probability distribution of the Bars and Stripes dataset, and show that the latter could potentially be better than the former. We numerically compared the learning performance of the noiseless digital QNNs of different number of layers for three different topology, i.e: star, chain and all-to-all and showed that the QNNs with all-to-all connectivity have the best performance. We then implemented the digital QNN with all-to-all ansatz on the real IBM quantum processor, but with a small number of layers to reduce the circuit depth, yet the results produced from the real hardware are very noisy even if we keep the circuit short. This is an expected result because the circuit depth is increased by two reasons (1) decomposition of the gates in our theoretical model into the hardware's native gates (2) the qubits on the quantum processors are not all-to-all connected so additional SWAP gates

are implemented to swap the qubits. Nevertheless, it was an amazing experience to have cloud access to a superconducting based quantum computer chip at 0.01 Kelvin sitting somewhere in the US. Similarly, we trained the analog QNN - the analog magnetic spin based approach - with all-to-all topology to learn the probability distribution of the Bar and Stripes dataset where the training of this magnetic spin based model could be done by adjusting the control parameters of the system. The analog system has less control parameters and is easier to train at the near-term. The ultimate quantitative limitation of the analog system is set by the calibration accuracy of the model that is implemented.

As follow up works, we will use the analog QNN to learn some practical probability distribution instead of synthetic datasets like the Bars and Stripes dataset. These probability distribution could be dataset from industry such as finance and supply chain industry. On the other hand, we will also study more basic properties of the model such as the expressibity and trainability, to understand the limitation and capability of the model. While we focus on the spin model in this thesis, our method could also be implemented on other quantum simulation platform such as photonic systems that are currently being developed in several labs and companies worldwide such as Xanadu, PsiQ and etc.

Appendix A

Learning process of a neural network

Once we have defined a design for the neural network, we can proceed to the training process. Neural networks generally perform supervised learning tasks like classification. In supervised learning the neural network is building knowledge from data sets where the right answer is provided in advance. The idea is to find an algorithm that compares the network output with the desired output and tune the weights and biases accordingly. The adjustment of the network parameters aims to approximate the correct output for all training samples. To measure the deviation from the target output we define a *cost function*. In literature sometimes it is also referred to also as a *loss* or *objective function*. One commonly used cost function is the *Mean Squared Error* or *MSE* ,

$$C(w) = \frac{1}{2} \left\langle \sum_j \left\| y_j^{(n)} - F_j(y^{in}) \right\|^2 \right\rangle = \left\langle C(w, y^{in}) \right\rangle \tag{A.1}$$

where $C(w, y^{in})$ is the cost for one particular input. It is important at this point to keep track of the indices carefully.

Definition A.0.1.

y_j^n : Value of neuron j in layer n, $y_j^{(n)} = F(z_j^{(n)})$
$F_j(y^{in})$: Desired output of neuron j in output layer
$z_j^{(n)}$: Input value for " y = f(z) "
$w_{jk}^{(n,n-1)}$: Weight (neuron k in layer n-1 feeding into neuron j in layer n)

There are various cost functions that we could use but this quadratic function is perfectly suited for understanding the basics of learning in neural networks. The quadratic cost allow us monitor easily how small changes in the parameters affect the cost. If we carefully inspect the definition of the MSE we can see that the cost

is equal to 0 when the output of the network matches the desired output. Thus, the aim of the training should be to minimize the cost function. Simply put, training is the process of finding a set of weights and biases which make the cost as small as possible. Typically, a method called *Gradient Descend* is used to minimize the cost function.

The idea behind this method is intuitive. Think of the cost function as a valley with slopes. Suppose there is a hiker that slowly make his way down the slope trying to reach the bottom of the valley. The derivatives of the cost function would tell us everything we need to know about the local "shape" of the valley, and hence how the hiker should move.

In a mathematical context, the gradient vector $\boldsymbol{\nabla} f$, is a vector that points in the direction of the steepest ascent of a function. We start at a random initial point in the function and calculate the *negative gradient* of the cost function w.r.t. the model parameters, since we want to move towards the minimum error. We move towards the minimum by a small step (negative gradient),

$$(x, y)_{k+1} = (x, y)_k - \rho \boldsymbol{\nabla} f(x, y)_k \tag{A.2}$$

where $(x, y)_k$ and $(x, y)_{k+1}$ are the coordinate points on the function at a time k and $k + 1$ respectively. The step we take in the direction of the minimum is denoted as ρ and in literature is called *learning step*. Usually we choose a small learning step to ensure convergence to at least a local minimum. This process is repeated, over and over, until we hopefully reach a global minimum.

However, Gradient Descent can be computationally expensive for large training sets because it goes through all the training set to do a single update on a parameter. Alternatively, we use a method called *Stochastic Gradient Descent (SGD)*. Stochastic gradient descent calculates an approximate value of the cost function by averaging over only a few samples. But how can we apply stochastic gradient descent to learn in a neural network?

For each step evaluate a few samples and update the weights and biases according to,

$$w_j \longrightarrow w_j - \rho \cdot \frac{\partial \tilde{C}(w)}{\partial w_j} \tag{A.3}$$

$$b_j \longrightarrow b_j - \rho \cdot \frac{\partial \tilde{C}(w)}{\partial b_j} \tag{A.4}$$

where $\tilde{C}(w)$ is the approximate version of $C(w)$.

Specifically, the learning rule of a neuron or a single-layer neural network is called a *Delta Rule*. Updating the weights and biases that lead to the output neuron layer is simple. The gradient of the cost function with respect to a weight (or bias) of an output neuron layer w_*, is,

$$\frac{\partial \tilde{C}(w)}{\partial w_*} = \frac{\partial}{\partial w_*}\left(\frac{1}{b}\sum_b C(w, y^{in})\right) \tag{A.5}$$

where b is the batch size of the samples that we use to approximate the cost function. This yields,

$$\frac{\partial \tilde{C}(w)}{\partial w_*} = \frac{1}{b}\sum_b \left(y_j^{(n)} - F_j(y^{in})\right) \cdot \frac{\partial y_j^{(n)}}{\partial z_j^{(n)}} \cdot \frac{\partial z_j^{(n)}}{\partial w_*}. \tag{A.6}$$

Here y=f(z) is the sigmoid activation function, that we defined in equation (2.2). Thus, the gradient that we are looking for is,

$$\frac{\partial \tilde{C}(w)}{\partial w_*} = \frac{1}{b}\sum_b \underbrace{\left(y_j^{(n)} - F_j(y^{in})\right) \cdot f'(z_j^{(n)})}_{\Delta_j} \cdot \frac{\partial z_j^{(n)}}{\partial w_*}. \tag{A.7}$$

The Δ_j is the same for all weights and biases in the output layer. The only things that changes is the partial derivative of w_*. If w_* was really a weight then we get,

$$\frac{\partial z_j^{(n)}}{\partial w_{jk}^{(n,n-1)}} = y_k^{(n-1)}. \tag{A.8}$$

While if w_* was really a bias we get,

$$\frac{\partial z_j^{(n)}}{\partial b_j^{(n)}} = 1. \tag{A.9}$$

Unfortunately, things are more complicated for parameters that are further back in the network. To update a weight (or bias) that is not on the output layer is not obvious, since we can't directly measure the impact this parameter has on the output. The trick here is to propagate the error backwards through the network. This is called *backpropagation*.

The derivative with respect to some parameter, somewhere in the network, for a specific input is,

$$\frac{\partial C(w, y^{in})}{\partial w_*} = \sum_j \underbrace{\left(y_j^{(n)} - F_j(y^{in})\right) \cdot f'\left(z_j^{(n)}\right)}_{\Delta_j} \cdot \frac{\partial z_j^{(n)}}{\partial w_*} \tag{A.10}$$

where,

$$\frac{\partial z_j^{(n)}}{\partial w_*} = \sum_k \frac{\partial z_j^{(n)}}{\partial y_k^{(n-1)}} \cdot \frac{\partial y_k^{(n-1)}}{\partial z_k^{(n-1)}} \cdot \frac{\partial z_k^{(n-1)}}{\partial w_*} \tag{A.11}$$

$$= \sum_k w_{jk}^{(n,n-1)} \cdot f'\left(z_k^{(n-1)}\right) \cdot \frac{\partial z_k^{(n-1)}}{\partial w_*}. \tag{A.12}$$

This is because we know that the input to a neuron is the weighted sum of it's input signals,

$$z_j^{(n)} = \sum_k y_k^{(n-1)} \cdot w_{jk}^{(n,n-1)}. \tag{A.13}$$

We apply this chain rule repeatedly until we reach w_*. If, we look closely, we can see that the derivative of $z_j^{(n)}$ with respect to w_* has two indices. The left index j is a free index, while k is being summed over. This means that we have a matrix-vector multiplication. Meaning that each pair of layers (n,n-1) contributes multiplication with the following matrix,

$$M_{jk}^{(n,n-1)} = w_{jk}^{(n,n-1)} \cdot f'\left(z_k^{(n-1)}\right). \tag{A.14}$$

Hence, we have a repeated matrix multiplication, going down the network,

$$\frac{\partial z_j^{(n)}}{\partial w_*} = \sum_{k,l,\ldots,u,v} M_{jk}^{(n,n-1)} \cdot M_{kl}^{(n-1,n-2)} \cdots M_{uv}^{(\tilde{n},\tilde{n}-1)} \cdot \frac{\partial z_u^{(\tilde{n})}}{\partial w_*}. \tag{A.15}$$

This repeated procedure stops when we finally encounter the parameter with respect to which we wanted to calculate the derivative of the cost function.

Appendix B

Code

B.1 Code used for the RBM

```
class RBM(object):

    def __init__(self, num_visible, num_hidden):
        # The number of neurons the visible layer has
        self.num_visible = num_visible
        # The number of neurons the hidden layer has
        self.num_hidden = num_hidden
        # We randomly iniatialize the biases with a Gausian with mean 0 and standard deviation 1
        self.a = np.zeros((1, num_visible))
        self.b = -4*np.ones((1, num_hidden))
        # The weights is a matrix
        # dimesions -> num_visible x num_hidden
        # We randomly iniatialize the weights with a Gausian with mean 0 and standard deviation 0.01
        self.weights = 0.01 * np.random.randn(num_visible, num_hidden)
        # The following parameters are used to implement momentum in training as suggested by Hinton
        self.w_inc = np.zeros(self.weights.shape)
        self.a_inc = np.zeros(self.a.shape)
        self.b_inc = np.zeros(self.b.shape)


    def train_RBM(self, training_data, epochs, mini_batch_size, eta):
        # The purpose of the learning is to create a good generative model of the set
        # of training vectors. We train the model using the Contrastive Divergence algorithhm.

        for epoch in range(epochs):
            # randomly shuffling the training data,
            # and then partitions it into mini-batches of the appropriate size
            error = 0
            training_data_size = len(training_data)
            mini_batches = [training_data[k:k+mini_batch_size] for k in range(0, training_data_size, mini_batch_size)]
            for batch in mini_batches:
                err = self.Contrastive_Divergence(batch, mini_batch_size, epoch, eta)
                error += err
            print("Epoch %s complete. Reconstruction error is %0.2f" % (epoch+1, error))

        print ("Training completed.\n")
```

```
def Contrastive_Divergence(self, v0, mini_batch_size, epoch, eta):
    # Update the network's weights and biases with mini-batch Contrastive Divergence.
    # mini_batch is a list of training data
    # eta is the learning rate.

    ## Positive phase of contrastive divergence ## -- Data driven phase

    # dimensions: [batchsize] x [num_hidden]
    z_h0 = np.dot(v0, self.weights) + np.tile(self.b,(v0.shape[0],1))
    prob_h0 = sigmoid(z_h0)

    # sample the states of hidden units based on prob_h0
    h0 = prob_h0 > np.random.rand(v0.shape[0], self.num_hidden)

    # Positive Phase product, needed to update the weights.
    # Using probability instead of the sampled hidden states usually has less sampling noise
    # which allows slightly faster learning
    vihj_data = np.dot(v0.T, prob_h0)

    # values needed to update biases
    vi_data = np.sum(v0, axis=0)
    hj_data = np.sum(prob_h0, axis=0)

    ## End of Positive Phase ##
```

Figure B.1: Positive phase of the Contrastive Divergence algorithm.

```
## Negative phase of contrastive divergence ## -- Reconstruction driven phase

# reconstruct the visible states from hidden states
z_v1 = np.dot(h0, self.weights.T) + np.tile(self.a,(h0.shape[0],1))
# for the reconstruction we do not sample the visible states
v1 = sigmoid(z_v1)

# sample the hidden states from the visible states
z_h1 = np.dot(v1, self.weights) + np.tile(self.b,(v1.shape[0],1))
prob_h1 = sigmoid(z_h1)

# Negative Phase products, needed to update the weights.
vihj_model = np.dot(v1.T, prob_h1)

# values needed to update biases
vi_model = np.sum(v1, axis=0)
hj_model = np.sum(prob_h1, axis=0)

## End of Negative Phase ##

# set momentum as per Hinton's practical guide to training RBMs
m = 0.5 if epoch > 5 else 0.9

self.w_inc = self.w_inc * m + (eta/mini_batch_size)*(vihj_data - vihj_model)
self.a_inc = self.a_inc * m + (eta/mini_batch_size)*(vi_data - vi_model)
self.b_inc = self.b_inc * m + (eta/mini_batch_size)*(hj_data - hj_model)

# Updating weights and biases
self.weights += self.w_inc
self.a += self.a_inc
self.b += self.b_inc

# Updating weights and biases
#self.weights += (eta/mini_batch_size)*(vihj_data - vihj_model)
#self.a += (eta/mini_batch_size)*(vi_data - vi_model)
#self.b += (eta/mini_batch_size)*(hj_data - hj_model)

err = np.sum((v0 - v1) ** 2)

return err
```

Figure B.2: Negative phase of the Contrastive Divergence algorithm.

```
    def Reconstruct(self, V_data, recon_setps):
        # Reconstruct the input vector
        # Sample hidden states from the input vector
        # and then sample visible states from the hidden states.
        # Returns a vector of dimensions: [1] x [num_visible]

        V_rec = np.copy(V_data)

        for i in range(recon_setps):
            # dimensions: [1] x [num_hidden]
            z = np.dot(V_rec,self.weights) + self.b
            prob_h = sigmoid(z)
            # sample hidden states
            h = prob_h > np.random.rand(1, self.num_hidden)

            z_v1 = np.dot(h, self.weights.T) + self.a
            # for the reconstruction we do not sample the visible states
            V_rec = sigmoid(z_v1)


        return V_rec
##############################################################################

def sigmoid(z):
    # The sigmoid function
    sig = 1.0/(1.0 + np.exp(-z))
    return sig
```

```
from tensorflow import keras
from keras.datasets import mnist
from matplotlib import pyplot as plt


# LOADING MNIST dataset
# We can verify that the split between train and test is 60,000 and 10,000
# respectively

mnist = keras.datasets.mnist
(X_train,Y_train),(X_test,Y_test) = mnist.load_data()

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')


# X_train is 60,000 rows of 28 x 28 values
# We reshape it to 60,000 x 784
# X_test is 10,000 rows of 28 x 28 values
# We reshape it to 10,000 x 1 x 784

training_data = np.reshape(X_train,(X_train.shape[0],784))
test_data = np.reshape(X_test, (X_test.shape[0], 784))

# Normalize inputs to be within [0,1] so as to behave like a probability
# Convert the grayscale images with values [0,255] to binary with values [0,1]
threshold = 128
training_data = (training_data > np.ones(training_data.shape)*threshold) *1
test_data = (test_data > np.ones(test_data.shape)*threshold) *1
```

Figure B.3: Preparing the MNIST dataset.

```
Num_visible_units = 784
Num_hidden_units = 100

rbm = RBM(Num_visible_units, Num_hidden_units)

# training parameters

EPOCHS = 50
# For datasets that contain a small number of equiprobable classes, the ideal mini-batch size is often
# equal to the number of classes
BATCH_SIZE = 10
# It is helpful to divide the total gradient computed on a mini-batch by the size of the mini-batch,
# so when talking about learning rates we will assume that they multiply the average,
# per-case gradient computed on a mini-batch, not the total gradient for the mini-batch.
ETA = 1/BATCH_SIZE

rbm.train_RBM(training_data, EPOCHS, BATCH_SIZE, ETA)
```

Figure B.4: Defining the model hyperparameters.

B.2 Code used for the QCBM

```
import random
import sys
import math
import numpy as np
import itertools
import matplotlib.pyplot as plt
# Import Qiskit
from qiskit.circuit import ParameterVector
from qiskit import QuantumCircuit, execute, Aer, transpile
from qiskit.visualization import plot_histogram
# Import PySwarms
import pyswarms as ps

class BAS_QCBM(object):

    def __init__(self, num_rows, num_columns, num_layers, topology):
        # The dimensions of the Bars and Stripes dataset.
        self.num_rows = num_rows
        self.num_columns = num_columns
        # This dataset is the Bars and Stripes (BAS) dataset.
        self.data = self.generate_BAS_dataset()
        # The target distribution in a dictionary form.
        self.target_distribution = self.generate_BAS_target_distribution()
        # The number of layer of the QCBM
        self.num_layers = num_layers
        # The topology of the entangling layer
        self.topology = self.qubit_topology(topology)
        # The number of paramaters the QCBM has
        self.num_params = self.count_params()
        # The QCBM
        self.qcbm = self.create_QCBM()
```

```
def qubit_topology(self, topology):
    # Choose the connectivity of the entangling layer

        try:
            if topology == 'chain':
                return 0
            elif topology == 'star':
                return 1
            elif topology == 'all-to-all':
                return 2
            else:
                raise ValueError
        except ValueError:
            sys.exit("\nInvalid choice. This topology is not applicable.\n")


def generate_BAS_dataset(self):
    # This dataset is the Bars and Stripes (BAS) dataset.
    # It is comprised by black and white images of [num_rows] x [num_columns] dimensions.
    # They contain any number of either horizontal stripes or vertical bars.
    data = []

    for h in itertools.product([0,1], repeat = self.num_columns):
        pic = np.repeat([h], self.num_rows, 0)
        data.append(pic.ravel().tolist())

    for h in itertools.product([0,1], repeat = self.num_rows):
        pic = np.repeat([h], self.num_columns, 1)
        elem = pic.ravel().tolist()
        if elem not in data:
            data.append(pic.ravel().tolist())

    return data
```

```
def generate_BAS_target_distribution(self):
    # Create a dictionary that holds the target distribution

    nBAS = (2**self.num_rows) + (2**self.num_columns) - 2

    distribution_dict = {}
    dataset = self.data

    for image in dataset:
        bitstring = ""
        for qubit in image:
            bitstring += str(qubit)

        distribution_dict[bitstring] = 1/nBAS

    return distribution_dict

def plot_BAS_target_distribution(self):
    # Plot the target distribution
            # set width of bar
    barWidth = 0.25
    fig = plt.subplots(figsize =(10, 6))

    # set height of bar and xticks
    n = 2**(self.num_rows + self.num_columns)
    xticks = []
    target = []

    for i in range(n):
        num = "{0:04b}".format(i)
        xticks.append(num)
        bitstring = str(num)
        target.append(self.target_distribution.get(bitstring,0))

    # Set position of bar on X axis
    br1 = np.arange(n)

    # Make the plot
    plt.bar(br1, np.asarray(target), color ='b', width = barWidth,
    edgecolor ='grey')

    # Adding Xticks
    plt.title('Target Distribution')
    plt.xlabel('patterns')
    plt.ylabel('probabilities')

    # Adding Xticks
    plt.xticks([r for r in range(n)],xticks)
    plt.show()
```

```
def visualize_non_BAS_images(self):

    #the BAS images
    dataset = self.data

    # Generate the non bas images
    non_bas_data = []

    num_of_BAS_patterns = (2**self.num_rows) + (2**self.num_columns) -2

    num_of_non_BAS_patterns = 2**(self.num_rows+self.num_columns) - num_of_BAS_patterns

    for i in range(16):
        num_str = "{0:04b}".format(i)
        num = [int(x) for x in num_str]
        if num not in dataset:
            non_bas_data.append(num)


    # plot the non-bas images
    fig = plt.figure(figsize=(10,4))

    fig.suptitle('non-BAS {0}x{1} images'.format(self.num_rows, self.num_columns),fontsize = 16)

    for idx in range(num_of_non_BAS_patterns):

        non_bas_img = np.asarray(non_bas_data[idx])
        # Adds a subplot at the 1st position
        fig.add_subplot(2, 5, idx+1)

        # showing image
        plt.imshow(np.reshape(non_bas_img, (2,2)), cmap = 'viridis', vmin = 0 , vmax = 1)
        plt.axis('off')


    plt.tight_layout()
    plt.show()


def clipped_negative_log_likelihood(self, measured_distribution):
    # Compute the value of the clipped negative log likelihood between a target
    # bitstring distribution and a measured bitstring distribution.
    epsilon =  0.0001
    value = 0.0

    target_keys = self.target_distribution.keys()
    measured_keys = measured_distribution.keys()
    all_keys = set(target_keys).union(measured_keys)

    for bitstring in all_keys:
        target_bitstring_value = self.target_distribution.get(bitstring, 0)
        measured_bitstring_value = measured_distribution.get(bitstring, 0)

        value += target_bitstring_value * math.log(max(epsilon, measured_bitstring_value))

    return -value
```

```
def count_params(self):
    # Count the parameters of the quantum circuit.

    num_qubits = self.num_rows + self.num_columns
    num_layers = self.num_layers
    num_params = 0

    for layer in range(num_layers):

        # entangling layer
        if (layer % 2) == 1:
            if self.topology == 2:
                # all-to-all connectivity
                # The entangling layer has 6 parameters in a fully connected topology
                num_params += 6
            else:
                # star or chain connectivity
                # The entangling layer has 3 parameters in a star or chain connected topology
                num_params += 3

        # the single qubit rotation layer
        if (layer % 2) == 0:
            # The single qubit rotations layer has 3 gates, Rz Rx Rz, on each qubit.
            # With the execption of the first layer, since it starts at |0..0>, and thus
            # an Rz rotation has no effect.
            # If we have an odd number of layers, we can omitt
            # the Rz rotations on the last layer as well
            if (layer == 0) or (layer == num_layers-1):
                num_params += num_qubits*2
            else:
                num_params += num_qubits*3

    return num_params
```

```
def create_QCBM(self):
    # Create a Quantum Circuit Born Machine with fully connected topology.
    # The quantum circuit is structured as layers of parameterized gates.
    # Each layer is consisted of a sigle-qubit rotation layer and an entagling layer.
    # The number of layers is defined by num_layer.

    num_qubits = self.num_rows + self.num_columns

    qcbm = QuantumCircuit(num_qubits)

    num_params = self.num_params
    num_layers = self.num_layers

    # define your parameters
    theta = ParameterVector('θ', num_params)

    self.theta = theta

    # index of paramter in the parameter vector
    idx = 0;

    for layer in range(num_layers):

        # entangling layer
        if (layer % 2) == 1:
            if self.topology == 2:
            # all-to-all topology\n",
                for qubit1 in range(num_qubits):
                    for qubit2 in range(qubit1+1,num_qubits):
                        qcbm.rxx(theta[idx],qubit1,qubit2)
                        idx += 1
            elif self.topology == 1:
                # star topology
                for qubit in range(1,num_qubits):
                    qcbm.rxx(theta[idx],0,qubit)
                    idx += 1
            else:
            # chain topology
                for qubit in range(num_qubits-1):
                    qcbm.rxx(theta[idx],qubit,qubit+1)
                    idx += 1

            # add a barrier to distinguish the layers\n",
            if layer < num_layers-1:
                qcbm.barrier()

        # the single qubit rotation layer
        if (layer % 2) == 0:
            if layer == 0 :
                for qubit in range(num_qubits):
                    qcbm.rx(theta[idx],qubit)
                    idx +=1
                    qcbm.rz(theta[idx],qubit)
                    idx +=1
            elif layer == num_layers-1:
                for qubit in range(num_qubits):
                    qcbm.rz(theta[idx],qubit)
                    idx +=1
                    qcbm.rx(theta[idx],qubit)
                    idx +=1
            else:
                for qubit in range(num_qubits):
                    qcbm.rz(theta[idx],qubit)
                    idx +=1
                    qcbm.rx(theta[idx],qubit)
                    idx +=1
                    qcbm.rz(theta[idx],qubit)
                    idx +=1

            # add a barrier to distinguish the layers\n",
            if layer < num_layers-1:
                qcbm.barrier()

    # measure all qubits
    qcbm.measure_all()

    return qcbm
```

```
def visualize_circuit(self):
    # visualize the QCBM
    qc = self.qcbm
    display(qc.draw(output = 'mpl'))

def f(self, x):
    # Higher-level method to do execute the QCBM for the whole swarm.

    # Inputs
    # ------
    # x: numpy.ndarray of shape (n_particles, dimensions)
    #      The swarm that will perform the search

    # Returns
    # -------
    # KL_divergence: numpy.ndarray of shape (n_particles, )
    #    The computed loss for each particle

    n_particles = x.shape[0]
    KL_divergence = [self.execute_QCBM(x[i]) for i in range(n_particles)]
    return np.array(KL_divergence)

def execute_QCBM(self, params):
    # Execute the quantum circuit for a specific set of parameters
    qc = self.qcbm

    theta = self.theta

    bind_dict = {}

    for i in range(self.num_params):
        # Need to use the defined ParameterVector label.
        key = theta[i]
        bind_dict[key] = params[i]

    # Assigned parameter without modifying the original circuit
    qc_params = qc.assign_parameters(bind_dict)

    job = execute(qc_params,Aer.get_backend('qasm_simulator'), shots=1000)

    # we can get the result of the outcome as follows
    counts = job.result().get_counts(qc_params)

    # Create a dictionary for the measured distribution
    measured_distribution = {}
    measured_keys = counts.keys()

    for bitstring in measured_keys:
        measured_value = counts.get(bitstring)
        measured_distribution[bitstring] = measured_value/1000

    loss = self.clipped_negative_log_likelihood(measured_distribution)

    return loss

def optimize_system_parameters(self, c1, c2, w, k):
    # Performing PSO on the given BAS dataset
    # Initialize swarm
    options = {'c1': c1, 'c2': c2, 'w': w}

    # Set the dimensions to the same size as the paramaters of the QCBM
    dimensions = self.num_params

    # Create bounds
    max_bound = math.pi * np.ones(dimensions)
    min_bound = - max_bound
    bounds = (min_bound, max_bound)

    # Call instance of PSO
    optimizer = ps.single.GlobalBestPSO(n_particles = k*dimensions, dimensions = dimensions, options=options, bounds=bounds)

    # Perform optimization
    cost, pos = optimizer.optimize(self.f, iters=100)

    #The optimized paramaters
    self.optimal_thetas = pos

    self.cost_history = optimizer.cost_history
```

In [ ]:
```
# Experiment on a real IBM quantum device
```

In [ ]:
```
from qiskit import IBMQ
import qiskit.tools.jupyter # import handy jupyter tools for viewing backend details and monitoring job status
%qiskit_job_watcher # You should see a floating tab "IBMQ Jobs" appear on the top left corner
IBMQ.save_account("0f6330b426e330eb2a9fd742c8768cc8cdfdc119bf8e998acee08a685c03ed3c164ea1890f94e5d95511004b8e514940785be06ba4900

IBMQ.load_account()
provider = IBMQ.get_provider(hub='ibm-q')
provider.backends()
```

In [ ]:
```
# get the least-busy backend at IQX, this step may take up to one minute
from qiskit.providers.ibmq import least_busy

device = least_busy(provider.backends(filters=lambda b: b.configuration().n_qubits >= 4 and
                                   not b.configuration().simulator and b.status().operational==True))

device
```

In [ ]:
```
# run on real quantum device and extract the statistics
# Execute the quantum circuit for the optimal set of parameters
# Calculate the qBAS score
# Plot the distribution that was measeaured after the optimization process

qc = qcbm_all_to_all1.qcbm

theta = qcbm_all_to_all1.theta
params = qcbm_all_to_all1.optimal_thetas

bind_dict = {}

for i in range(qcbm_all_to_all1.num_params):
    # Need to use the defined ParameterVector label.
    key = theta[i]
    bind_dict[key] = params[i]

# Assigned parameter without modifying the original circuit
qc_params = qc.assign_parameters(bind_dict)

optimized_circuit = qiskit.transpile(qc_params, device)
optimized_circuit.draw(output = 'mpl')
```

B.3 Code used for Analog simulation

```
import random
import math
import numpy as np
import itertools
import matplotlib.pyplot as plt
from scipy import linalg
# Import constants
import scipy.constants as sc
# Import PySwarms
import pyswarms as ps

class BAS_ANALOG_QML(object):

    def __init__(self, num_rows, num_columns):
        # The dimensions of the Bars and Stripes dataset.
        self.num_rows = num_rows
        self.num_columns = num_columns
        # This dataset is the Bars and Stripes (BAS) dataset.
        self.data = self.generate_BAS_dataset()
        # The target distribution in a dictionary form.
        self.target_distribution = self.generate_BAS_target_distribution()
        # The number of qubits the system has
        self.num_qubits = num_rows + num_columns
        # The parameters of the hamiltonian

        # create only 1d array

        self.B = np.random.randint(-10, 10, 2*self.num_qubits)
        self.J = np.random.randint(-10, 10, 6)
        self.t = np.random.randint(0,3)
```

```
def create_Hamiltonian(self):
    # All interactions of the system are on the z axis

    # The pauli Z matrix
    z = np.array([[1,0],[0,-1]])
    # The pauli X matrix
    x = np.array([[0,1],[1,0]])
    # The identity matrix
    I = np.array([[1,0],[0,1]])

    # The spin in the z and x axis
    sz, sx = 0.5*z, 0.5*x


    # Defining the hamiltonian
    H1 = -self.B[0]*np.kron(np.kron(np.kron(sz,I),I),I)
    H2 = -self.B[1]*np.kron(np.kron(np.kron(I,sz),I),I)
    H3 = -self.B[2]*np.kron(np.kron(np.kron(I,I),sz),I)
    H4 = -self.B[3]*np.kron(np.kron(np.kron(I,I),I),sz)

    # sigma_x: applying magnetic field in x direction
    # This is required to change the probability distribution
    # If we consider only sigma_z we will only get a additional phase for each of the computational basis
    # This is the reason why we keep getting the same probability distribution and not be able to train it.
    Hm1 = -self.B[4]*np.kron(np.kron(np.kron(sx,I),I),I)
    Hm2 = -self.B[5]*np.kron(np.kron(np.kron(I,sx),I),I)
    Hm3 = -self.B[6]*np.kron(np.kron(np.kron(I,I),sx),I)
    Hm4 = -self.B[7]*np.kron(np.kron(np.kron(I,I),I),sx)

    H12 = np.kron(np.kron(np.kron(sz,sz),I),I)
    H13 = np.kron(np.kron(np.kron(sz,I),sz),I)
    H14 = np.kron(np.kron(np.kron(sz,I),I),sz)
    H23 = np.kron(np.kron(np.kron(I,sz),sz),I)
    H24 = np.kron(np.kron(np.kron(I,sz),I),sz)
    H34 = np.kron(np.kron(np.kron(I,I),sz),sz)

    Hint  = (-self.J[0]*H12 + -self.J[1]*H13 + -self.J[2]*H14
             -self.J[3]*H23 -self.J[4]*H24 -self.J[5]*H34)

    self.Hamiltonian = (H1 + H2 + H3 + H4 +
                        Hm1 + Hm2 + Hm3 + Hm4 + Hint)
```

```
def update_Hamiltonian_parameters(self,params):
    # Update the parameters with their optimal values
    # Multiple self.num_qubits by 2
    self.B = params[0:2*self.num_qubits]
    self.J = params[2*self.num_qubits:params.shape[0]-1]
    self.t = params[-1]
    self.create_Hamiltonian()

def time_evolution(self):
    # Time evolution of the quantum system
    # which is described by the given Hamiltonian

    # Create the systems Hamiltonian
    self.create_Hamiltonian()

    # The initial state of the system is |++++>
    state_0 = np.array([[1],[0]])
    state_1 = np.array([[0],[1]])
    state_0000 = np.kron(state_0,np.kron(state_0,np.kron(state_0,state_0)))
    state_1111 = np.kron(state_1,np.kron(state_1,np.kron(state_1,state_1)))
    psi_0 = (state_0000 + state_1111)/math.sqrt(2)


    # Time evolution of the state
    # Removed hbar
    exp = linalg.expm((-(1j)*self.Hamiltonian*self.t))
    self.psi_t = np.dot(exp,psi_0)
```

```
def evaluate_params(self, params):
    # apply time evolution for a specific set of parameters

    # Multiple self.num_qubits by 2
    self.B = params[0:2*self.num_qubits]
    self.J = params[2*self.num_qubits:params.shape[0]-1]
    self.t = params[-1]

    # let the system evolve in time with the specific parameters
    self.time_evolution()

    # Extract the probabilities from the state vector
    # There are 16 possible states
    dist_len = 2**self.num_qubits

    # Create a dictionary for the measured distribution
    # that the 16 possible states create
    measured_distribution = {}

    for i in range(dist_len):
        s = np.zeros((16,1))
        s[i] = 1
        c = np.dot(np.conj(s).T,self.psi_t)

        prob = c[0,0].real**2 + c[0,0].imag**2
        bitstring = "{0:04b}".format(i)
        measured_distribution[bitstring] = prob
    #print(self.psi_t)
    #print(measured_distribution)
    loss = self.clipped_negative_log_likelihood(measured_distribution)

    return loss
```

```python
def optimize_system_parameters(self, c1, c2, w):
    # Performing PSO on the given BAS dataset
    # Initialize swarm
    options = {'c1': c1, 'c2': c2, 'w':w}

    # Set the dimensions to the same size as the paramaters of the QCBM

    # Multiple self.num_qubits by 2
    dimensions = 2*self.num_qubits + 6 + 1

    # Create bounds
    # B and J is bound at [-7.15,-7.15]
    bound_val = 7.15
    max_B_bound = bound_val*np.ones(8)
    min_B_bound = - max_B_bound
    # J is bound at [0,10]
    max_J_bound = bound_val*np.ones(6)
    min_J_bound = -max_J_bound
    # t is bound at [0,2.1]
    max_t_bound = [2.1]
    min_t_bound = [0]

    max_bound = np.array(max_B_bound.tolist() + max_J_bound.tolist() + max_t_bound)
    min_bound = np.array(min_B_bound.tolist() + min_J_bound.tolist() + min_t_bound)

    bounds = (min_bound, max_bound)

    # Call instance of PSO
    optimizer = ps.single.GlobalBestPSO(n_particles=6*dimensions, dimensions=dimensions, options=options, bounds=bounds)

    # Perform optimization
    cost, pos = optimizer.optimize(self.f, iters=100)

    #The optimized paramaters
    self.update_Hamiltonian_parameters(pos)

    self.cost_history = optimizer.cost_history
```

B(Hz)		J(Hz)		t(s)
$B_{z'z\_1}$:	2.87376581	J_{12}:	0.75910833	1.321437421605972
$B_{z'z\_2}$:	3.33100868	J_{13}:	3.46054473	
$B_{z'z\_3}$:	-2.16800743	J_{14}:	-3.03641454	
$B_{z'z\_4}$:	2.24630931	J_{23}:	0.1366996	
$B_{x'x\_2}$:	-3.5541876	J_{24}:	0.1829026	
$B_{x'x\_2}$:	-3.66487226	J_{34}:	3.30827745	
$B_{x'x\_3}$:	4.05247379			
$B_{x'x\_4}$:	-3.98481415			

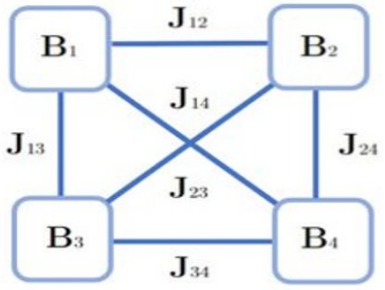


Figure B.5: The parameters implemented in the classical simulation for analog generative modelling. In the classical simulation, we adapted the re-scaled Hamiltonian $\bar{H} = H/c$ and the re-scaled time $\bar{t} = ct$, where $c = 7 \cdot 10^6$. These re-scaled quantities will give us the same unitary evolution, i.e: $U = e^{\frac{-i\bar{H}\bar{t}}{\hbar}} = e^{\frac{-i(H/c)\cdot ct}{\hbar}} = e^{\frac{-iHt}{\hbar}}$, where $H(\vec{B}, \vec{J})$ is the Ising Hamiltonian in Eqn.(4.16) while $H(\vec{\bar{B}}, \vec{\bar{J}})$ is the re-scaled Hamiltonian.

Bibliography

[1] Richard P Feynman. "Simulating physics with computers". In: *Feynman and computation*. CRC Press, 2018, pp. 133–153.

[2] John J Hopfield. "Neural networks and physical systems with emergent collective computational abilities." In: *Proceedings of the national academy of sciences* 79.8 (1982), pp. 2554–2558.

[3] Stephen G Brush. "History of the Lenz-Ising model". In: *Reviews of modern physics* 39.4 (1967), p. 883.

[4] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. "A learning algorithm for Boltzmann machines". In: *Cognitive science* 9.1 (1985), pp. 147–169.

[5] Paul Smolensky. *Information processing in dynamical systems: Foundations of harmony theory*. Tech. rep. Colorado Univ at Boulder Dept of Computer Science, 1986.

[6] Y Freund and D Haussler. "Unsupervised learning of distributions on binary vectors using two layer networks (Technical Report UCSC-CRL-94-25)". In: *University of California, Santa Cruz* (1994).

[7] Geoffrey E Hinton. "Training products of experts by minimizing contrastive divergence". In: *Neural computation* 14.8 (2002), pp. 1771–1800.

[8] Geoffrey E Hinton. "A practical guide to training restricted Boltzmann machines". In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 599–619.

[9] Aram W Harrow, Avinatan Hassidim, and Seth Lloyd. "Quantum algorithm for linear systems of equations". In: *Physical review letters* 103.15 (2009), p. 150502.

[10] Guang Hao Low, Theodore J Yoder, and Isaac L Chuang. "Quantum inference on Bayesian networks". In: *Physical Review A* 89.6 (2014), p. 062315.

[11] Nathan Wiebe, Daniel Braun, and Seth Lloyd. "Quantum algorithm for data fitting". In: *Physical review letters* 109.5 (2012), p. 050505.

[12] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. "Quantum principal component analysis". In: *Nature Physics* 10.9 (2014), pp. 631–633.

[13] Patrick Rebentrost, Masoud Mohseni, and Seth Lloyd. "Quantum support vector machine for big data classification". In: *Physical review letters* 113.13 (2014), p. 130503.

[14] Maiyuren Srikumar, Charles D Hill, and Lloyd CL Hollenberg. "Clustering and enhanced classification using a hybrid quantum autoencoder". In: *Quantum Science and Technology* 7.1 (2021), p. 015020.

[15] Christa Zoufal, Aurélien Lucchi, and Stefan Woerner. "Quantum generative adversarial networks for learning and loading random distributions". In: *npj Quantum Information* 5.1 (2019), pp. 1–9.

[16] Jin-Guo Liu and Lei Wang. "Differentiable learning of quantum circuit born machines". In: *Physical Review A* 98.6 (2018), p. 062324.

[17] Marcello Benedetti et al. "A generative modeling approach for benchmarking and training shallow quantum circuits". In: *npj Quantum Information* 5.1 (2019), pp. 1–9.

[18] Daiwei Zhu et al. "Training of quantum circuits on a hybrid quantum computer". In: *Science advances* 5.10 (2019), eaaw9918.

[19] LJV Miranda, A Moser, and SK Cronin. "PySwarms: A Research Toolkit for Particle Swarm Optimization in Python 2017". In: *URL https://zenodo.org/badge/latestdoi/97002861* ().

[20] Andrew Cross. "The IBM Q experience and QISKit open-source quantum computing software". In: *APS March meeting abstracts*. Vol. 2018. 2018, pp. L58–003.

[21] Andrew J Daley et al. "Practical quantum advantage in quantum simulation". In: *Nature* 607.7920 (2022), pp. 667–676.

[22] Pedram Roushan et al. "Spectroscopic signatures of localization with interacting photons in superconducting qubits". In: *Science* 358.6367 (2017), pp. 1175–1179.