Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Dipartimento di Fisica e Astronomia
Corso di Laurea in Fisica

Machine Learning for Quantum Error Mitigation on NISQ devices

Relatore:
Prof. Daniele Bonacorsi

Correlatori:
Dott. Simone Gasperini
Dott. Marco Lorusso

Presentata da:
Marco Vassallo

Anno Accademico 2023/2024

Abstract

Il Quantum Error Mitigation (QEM) è un insieme di tecniche utilizzate per ridurre l'impatto del rumore nel calcolo quantistico senza la necessità di implementare specifici protocolli per una completa rimozione degli errori, come avviene invece nel caso del Quantum Error Correction (QEC). Questo approccio mira a migliorare le prestazioni degli algoritmi mitigando gli effetti indesiderati del rumore che caratterizzano i computer quantistici odierni, appartenenti alla cosidetta classe Noisy Intermediate-Scale Quantum (NISQ).

Nel contesto del calcolo quantistico *gate-based*, questa tesi introduce un innovativo approccio al QEM basato su tecniche di Machine Learning (ML). In particolare, viene proposta una semplice rete neurale classica (architettura dell'autoencoder) in grado di imparare uno specifico modello di rumore dai dati forniti. Dopo una prima fase, che prevede la preparazione di un dataset eseguendo la simulazione di diversi circuiti quantistici, la rete neurale viene allenata per l'identificazione e mitigazione degli effetti del rumore quantistico sull'istogramma misurato. L'efficacia del modello è poi validata calcolando una metrica per confrontare ogni coppia di istogrammi *rumoroso* e *mitigato* con la corrispondente distribuzione di probabilità ideale.

I promettenti risultati ottenuti aprono nuove prospettive per applicazioni del QEM in contesti sperimentali anche più avanzati, contribuendo alla crescita e alla maturazione della computazione quantistica.

Contents

Introduction **1**

1 Quantum Computing **3**

1.1 Quantum information theory 3

1.1.1 Single-qubit state 3

1.1.2 Multi-qubit state 5

1.1.3 Entanglement 6

1.1.4 Projective-Valued Measurement 7

1.2 Gate-based quantum computing 8

1.2.1 Quantum gates 8

1.2.2 Clifford Group 11

1.2.3 NISQ devices 12

1.2.4 Qiskit framework 13

2 Machine Learning **15**

2.1 Introduction to Machine Learning 15

2.1.1 Learning paradigms 15

2.1.2 Gradient-based optimization 16

2.1.3 Supervised learning 18

2.1.4 Training, validation, and testing 19

2.2 Artificial neural networks 20

2.2.1 Multi-Layer Perceptron 21

2.2.2 Autoencoder architecture 22

2.2.3 TensorFlow framework 23

3 Autoencoder for Error Mitigation **25**

3.1 Data preparation 25

3.1.1 Circuits generation 26

3.1.2 Clifford ideal simulation 27

3.1.3 Noisy simulation 27

3.2 Model definition 29

3.3 Data analysis 30

3.4 Possible applications 33

Conclusion **35**

Bibliography **37**

A Noisy circuit simulation in Qiskit **41**

B Circuit optimization in Pennylane **43**

C Denoising Autoencoder in Keras **45**

Introduction

Quantum computing stands at the frontier of technological advancement, promising computational power far surpassing classical systems. However, this potential is hindered by the inherent susceptibility of quantum systems to errors, stemming from decoherence and imperfections in hardware. Traditional methods such as Quantum Error Correction (QEC) mitigate errors by encoding quantum information redundantly, but they come at a high cost in terms of hardware resources. In contrast, Quantum Error Mitigation (QEM) seeks to mitigate errors directly within the measured results, presenting a more hardware-efficient alternative that has gained traction in recent experimental demonstrations.

This thesis introduces an innovative approach to QEM using machine learning techniques, specifically targeting the reduction of noise within quasi-probability distributions rather than focusing solely on correcting expected values. By harnessing the power of machine learning, we aim to transform noisy quantum distributions into states that closely resemble their ideal counterparts. This methodology not only addresses the practical challenges of error mitigation in quantum computing but also opens new avenues for applying QEM in advanced experimental and practical contexts.

In this study, we employ real-world quantum circuits as our initial dataset, incorporating both benchmarking circuits and simulations involving five qubits. Subsequently, we generate ideal circuits from Gaussian distributions, which serve as targets for our optimization algorithm. Central to our approach is the implementation of an autoencoder neural network with 13 layers, trained on a dataset comprising 4500 entries of noisy distributions paired with their corresponding ideal distributions, simulated using a custom noise model developed in Qiskit.

The efficacy of our approach is validated through Wasserstein distance, a comprehensive metric used for comparing distributions, which demonstrated the alignment between denoised data and the expected ideal distributions.

Overall, this thesis not only contributes to the theoretical understanding of Quantum Error Mitigation but also provides practical insights and methodologies that can significantly enhance the reliability and performance of quantum computing technologies in diverse application scenarios. By leveraging machine learning techniques, we pave the way for the broader integration of QEM in quantum computing research and development, ultimately advancing the maturity and applicability of quantum technologies in the coming years.

Chapter 1

Quantum Computing

1.1 Quantum information theory

Quantum information theory is a scientific field that combines quantum mechanics and information theory. It focuses on how quantum systems can be used to store, process, and transmit information. It extends classical information theory by incorporating the principles of quantum mechanics, which allows for new and more powerful methods of computation and communication [1]. In quantum mechanics, a physical system is usually represented by a probabilistic wave function in an Hilbert space. Dirac bra-ket notation can be employed to represent a quantum state as a vector of complex amplitudes, denoted as $|\psi\rangle$. This representation allows for the encoding of all the information associated with the quantum state. The exclusive features of quantum-mechanical systems, such as superposition and entanglement, can be exploited to theoretically solve certain classes of problems that would be impossible or too energy- and time-consuming for a classical computer. Some examples are the so called Shor's algorithm [2] for the large integer numbers factorization problem (exponentially complex for classical computing) or Grover's algorithm [3], which provide a theoretical quadratic speedup for searching problem in an unstructured database.

This relatively new field of research holds promise to completely change the study of many subjects like computational chemistry, machine learning and materials science, where complex simulations and optimizations are essential.

We are now going to provide a brief survey of all the basic concepts needed to understand quantum computing and its applications, starting with the fundamentals of quantum mechanics and the mathematical definition of the qubit, fundamental unit of quantum information.

1.1.1 Single-qubit state

A qubit is defined as a mathematical object used to describe a two-level quantum system. Its state $|\psi\rangle$ is a unit-norm complex vector that can be written as a linear

combination of the $|0\rangle$ and $|1\rangle$ basis vectors:

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \alpha\,|0\rangle + \beta\,|1\rangle \tag{1.1}$$

where α and β are complex coefficients such that $|\alpha|^2 + |\beta|^2 = 1$, and $|0\rangle$ and $|1\rangle$ are the two orthonormal vectors constituting the so called *computational basis* and spanning the whole single-qubit Hilbert space $\mathcal{H}$:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \qquad \mathcal{H} = span\{|0\rangle\,, |1\rangle\} \tag{1.2}$$

Analogously, we can define another basis denoted by the symbols $|+\rangle$ (*plus* state) and $|-\rangle$ (*minus* state) defined as:

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \qquad |-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{1.3}$$

This expression of a state in a linear combination of the basis, is the so-called *superposition* of multiple states that potentially leads to quantum advantage with respect to classical computation[4]. This is because, while operating on the quantum state to solve a given problem, we are effectively evolving over all the states represented by the superposition.

To have a deeper comprehension of the qubit, it is possible to utilise the Bloch sphere that provides a geometrical representation of a single-qubit state. On the Bloch sphere, the state vector $|\psi\rangle$, can be written as:

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi}\sin\left(\frac{\theta}{2}\right)|1\rangle \tag{1.4}$$

where θ and ϕ are respectively the polar angle and the azimuthal angle defining the position in spherical coordinates. Any point on the surface of the sphere represents a valid unit-norm pure state of the qubit. The north pole of the sphere ($\theta = 0$) corresponds to the state $|0\rangle$; the south pole ($\theta = \pi$) corresponds to the state $|1\rangle$. All the equal superpositions states between $|0\rangle$ and $|1\rangle$ lie on the equatorial plane ($\theta = \pi/2$) , but they all have a different relative phases ϕ. In this representation the global phase of the state is omitted since it cancels out during normalization when measurement occurs. This means that the Bloch sphere encapsulates all the essential physical information about a qubit state. Figure 1.1 illustrates the Bloch sphere and the position of a general state vector $|\psi\rangle$.

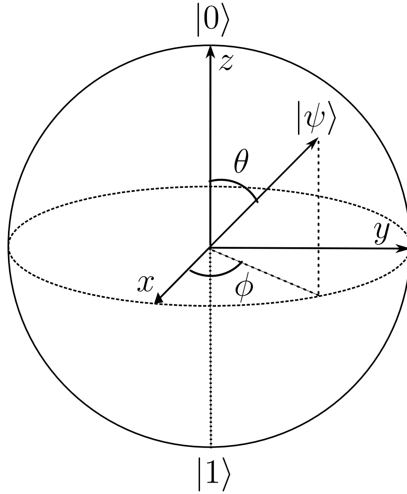


Figure 1.1: Representation of a general state vector $|\psi\rangle$ on the Bloch sphere.

1.1.2 Multi-qubit state

As an example for a multi-qubit state, consider the most simple case of two Hilbert spaces $\mathcal{H}_A$ and $\mathcal{H}_B$, corresponding to systems A and B. The space $\mathcal{H}$ of the composite system is the tensor product of the individual spaces, i.e. $\mathcal{H} = \mathcal{H}_A \otimes \mathcal{H}_B$. If systems A and B are prepared in states $|\psi\rangle_A$ and $|\psi\rangle_B$ respectively, then the state vector $|\psi\rangle = |\psi\rangle_A \otimes |\psi\rangle_B$ represents the state of the composite system [4].

With this formalism, we establish the basis of the two-qubits Hilbert space by combining the individual bases of each system. For instance, if each system's basis comprises $|0\rangle$ and $|1\rangle$, then the basis for the composite system includes all possible combinations of tensor products of these states:

$$\{|a\rangle_A \otimes |b\rangle_B \mid a, b \in \{0, 1\}\} \qquad |a\rangle \otimes |b\rangle = |ab\rangle \tag{1.5}$$

The general state of the composite system can then be expressed as:

$$\begin{aligned} |\psi\rangle &= (\alpha_A |0\rangle + \beta_A |1\rangle) \otimes (\alpha_B |0\rangle + \beta_B |1\rangle) \\ &= \alpha_A\alpha_B |00\rangle + \alpha_A\beta_B |01\rangle + \beta_A\alpha_B |10\rangle + \beta_A\beta_B |11\rangle \end{aligned} \tag{1.6}$$

In quantum computing, we categorize all possible states as either *pure* or *mixed*. When we have full knowledge of the quantum state, then it is defined a pure state, which can be always represented as a state vector $|\psi\rangle$, written as a linear combinations of tensor products of other state vectors.

However, a quantum system can also exist in a mixed state, namely a classical mixture of several pure states $|\psi_i\rangle$, each with a corresponding probability p_i [5]. This happens, for instance, when we don't have a full knowledge of the quantum state and the usual Dirac formalism is not enough to encode all the information. In fact, to fully represent a mixed state, we have to introduce the concept of density matrix ρ, a more powerful mathematical tool defined as:

$$\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i| \tag{1.7}$$

In general, ρ is a unit-trace ($\mathrm{tr}(\rho) = 1$) and hermitian ($\rho = \rho^\dagger$) matrix. Moreover, the purity of the quantum state is defined as $\mathrm{tr}(\rho^2) \leq 1$. For a pure state $|\psi\rangle$, the density matrix is simply $\rho = |\psi\rangle\langle\psi|$ and indeed we have:

$$\mathrm{tr}(\rho^2) = \mathrm{tr}(|\psi\rangle\langle\psi|\psi\rangle\langle\psi|) = \mathrm{tr}(|\psi\rangle\langle\psi|) = \mathrm{tr}(\rho) = 1 \tag{1.8}$$

since $\langle\psi|\psi\rangle = 1$ by definition of unit-norm state vector. For a mixed state ρ, the purity turns out to be always smaller than 1, in particular we have:

$$\mathrm{tr}(\rho^2) = \mathrm{tr}\left(\sum_{i,j} p_i p_j |\psi_i\rangle\langle\psi_i|\psi_j\rangle\langle\psi_j|\right) = \mathrm{tr}\left(\sum_i p_i^2 |\psi_i\rangle\langle\psi_i|\right) = \sum_i p_i^2 < 1 \tag{1.9}$$

since $\langle\psi_i|\psi_j\rangle = \delta_{ij}$ and $p_i \in [0, 1]$ are probabilities summing up to 1.

The density matrix formalism is crucial for quantifying noise sources like imperfect gates, decoherence from environmental interactions, and measurement errors. It allows researchers to simulate and analyze noisy quantum circuits, providing insights into how noise disrupts phase relationships and computational outcomes. Using detailed measurements and analysis enabled by the density matrix, scientists can study and possibly mitigate noise effects, optimizing algorithms for improved reliability of quantum computations [6].

1.1.3 Entanglement

Quantum entanglement is a phenomenon in quantum mechanics where the states of two or more systems (e.g. qubits) become correlated in such a way that the state of each individual system cannot be described independently of the state of the others, even when they are separated by large distances. Entanglement plays a crucial role in various quantum information and quantum computing applications, enabling phenomena such as quantum teleportation.

As an example, let's consider again the most simple case of a two-qubits state, living in the composite Hilbert space $\mathcal{H} = \mathcal{H}_A \otimes \mathcal{H}_B$, and defined by $|\psi\rangle = |\psi\rangle_A \otimes |\psi\rangle_B$. This composite state is called entangled if it cannot be written as a product of states of individual qubits. A well-known example of an entangled state is the Bell state:

$$|\Phi^+\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{1.10}$$

Consider measuring the state of one qubits in $|\Phi^+\rangle$. If we measure qubit A and find it in state $|0\rangle$, the state of qubit B also collapses to $|0\rangle$ with 100% probability. Similarly, if qubit A is found in $|1\rangle$, qubit B collapses to $|1\rangle$, as well. This strong correlation holds regardless of the distance between the qubits, illustrating the non-local nature of entanglement.

Moreover, a state is termed *maximally entangled* if its quantum correlations are maximal and cannot be further enhanced. This concept is pivotal for instance in quantum communication protocols, where maximizing entanglement is often desired for achieving robust and reliable quantum operations. Bell states represent the

most used maximally entangled states for a two-qubit system. There are actually three more Bell states:

$$|\Phi^-\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}} \qquad |\Psi^+\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}} \qquad |\Psi^-\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}} \tag{1.11}$$

They form a basis for the composite Hilbert space $\mathcal{H}_A \otimes \mathcal{H}_B$, which is the reason for their importance in quantum computation, communication, and fundamental tests of quantum mechanics.

1.1.4 Projective-Valued Measurement

In quantum computing, a Projective-Valued Measurement (PVM) is a fundamental technique used to extract information about the state of a quantum system. Given a pure quantum state $|\psi\rangle$, if we want to measure a physical observable, represented by a Hermitian operator O defined in the Hilbert space $\mathcal{H}$, a PVM provides a simple method to carry out this measurement effectively [4].

The core concept of a PVM involves projecting the state vector $|\psi\rangle$ onto the eigenstates $\{|\phi_i\rangle \mid O|\phi_i\rangle = \lambda_i|\phi_i\rangle\}$ of the observable operator. Since O is Hermitian, the spectral theorem assures that the eigenstates $|\phi_i\rangle$ form an orthonormal basis of $\mathcal{H}$ and that all the eigenvalues λ_i are real numbers. This also means that any quantum state $|\psi\rangle \in \mathcal{H}$ can be expressed as a linear combination:

$$|\psi\rangle = \sum_i c_i|\phi_i\rangle \tag{1.12}$$

where $c_i = \langle\phi_i|\psi\rangle$ are complex coefficients. To perform a measurement using a PVM, each eigenstate $|\phi_i\rangle$ is associated with a projection operator $P_i = |\phi_i\rangle\langle\phi_i|$. The probability p_i of observing the measurement outcome $\lambda_i \in \mathbb{R}$ corresponding to $|\phi_i\rangle$ is computed as:

$$p_i = \langle\psi|P_i|\psi\rangle = \sum_j c_j^*\langle\phi_j|\phi_i\rangle\langle\phi_i|\psi\rangle = \sum_j c_j^* c_i\langle\phi_j|\phi_i\rangle = |c_i|^2 \tag{1.13}$$

since $\langle\phi_j|\phi_i\rangle = \langle\phi_i|\phi_j\rangle = \delta_{ij}$ by definition of orthonormal basis. Upon conducting the measurement, the quantum state $|\psi\rangle$ collapses to the eigenstate $|\phi_i\rangle$ with probability p_i.

The expectation value $\langle O\rangle$ of the given observable can be obtained from repeated measurements on identical copies of the quantum state $|\psi\rangle$ and it is computed as:

$$\langle O\rangle = \langle\psi|O|\psi\rangle = \sum_i c_i\langle\psi|O|\phi_i\rangle = \sum_i c_i\langle\psi|\lambda_i|\phi_i\rangle = \sum_i |c_i|^2\lambda_i = \sum_i p_i\lambda_i \tag{1.14}$$

by the definition of eigenstates $|\phi_i\rangle$ with eigenvales λ_i, and the previous equation for probabilities $p_i = |c_i|^2$.

1.2 Gate-based quantum computing

The two main types of implementation of quantum computing are *quantum annealing* and gate-based Quantum Computing (QC). Annealing is a technique which uses the adiabatic theorem (which says that, if the evolution of the system is fairly small, the system will stay in the ground state) to transition from a initial ground state of a trivial Hamiltonian to a final state of an Hamiltonian whose ground state is the solution to our problem. The approach of gate-based quantum computing is instead made by operating unitary transformations to a series of qubits which encode the problem solution. The main difference is that annealing is used for optimization problems while gate-based is more versatile and can be implemented in various ways. Lastly, since we are in *NISQ* (Noisy Intermediate-Scale Quantum computing) regime for QC, the gate-based approach is more sensitive to the noise and decoherence, hence it needs a lot of error correction and error mitigation.

The introduction we provided about quantum information theory and its basic principles is quite general and applicable to the study of any system composed of qubits. Next, we will focus on the type of quantum computing that interests us: gate-based QC.

1.2.1 Quantum gates

Quantum computing hinges on the principle of reversible operations, a fundamental requirement rooted in quantum mechanics. Unlike classical computing, where irreversible operations are common, quantum computations must preserve information to uphold coherence and respect to quantum conservation laws.

Thus unitary gates are the only used implementation in quantum computation, because they enforce reversibility and preserve the norm of quantum states. It guarantees that quantum algorithms can manipulate quantum states accurately and reliably, utilizing phenomena such as superposition and entanglement.

Here are some of the fundamental single-qubit gates used in quantum computing:

- **X Gate**: Also known as the NOT gate, represented by the matrix:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{1.15}$$

  It flips the probabilities of the basis states $|0\rangle$ and $|1\rangle$.

- **Y Gate**: Represented by the matrix:

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \tag{1.16}$$

  It performs a rotation around the y-axis in the Bloch sphere, mapping $|0\rangle$ to $i|1\rangle$ and $|1\rangle$ to $-i|0\rangle$.

- **Z Gate**: Represented by the matrix:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{1.17}$$

It introduces a phase shift of π (180 degrees) to the $|1\rangle$ state.

- **Hadamard (H) Gate**: Changes the basis from $|0\rangle, |1\rangle$ to $|+\rangle, |-\rangle$ and is represented by:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{1.18}$$

  Applying the Hadamard gate twice results in the identity operator.

- **S Gate**: The phase gate represented by:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \tag{1.19}$$

  It introduces a phase shift of $\frac{\pi}{2}$ (90 degrees) to the $|1\rangle$ state, it is then equivalent to $\sqrt{Z}$.

- **T Gate**: Represents a phase gate with:

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix} \tag{1.20}$$

  It introduces a phase shift of $\frac{\pi}{4}$ (45 degrees) to the $|1\rangle$ state, it is then equivalent to $\sqrt{S}$.

These gates exemplify how quantum computations leverage unitary transformations to manipulate quantum information while preserving coherence.

Additionally, the general form of a single-qubit gate U can be expressed as:

$$U = e^{i\alpha} \begin{pmatrix} \cos\frac{\theta}{2} & -e^{i\phi}\sin\frac{\theta}{2} \\ e^{-i\phi}\sin\frac{\theta}{2} & e^{i(\alpha+\phi)}\cos\frac{\theta}{2} \end{pmatrix}, \tag{1.21}$$

where α is a global phase, θ is the rotation angle around an axis in the Bloch sphere, ϕ determines the relative phase, and U is unitary, ensuring the conservation of quantum state probabilities.

We can now introduce multi-qubit gates, starting with the simple case of two-qubit gates represented by $U \in SU(4)$. One of the most fundamental and important two-qubit gates is the CNOT (controlled-NOT), which flips the state of the second qubit (b) if and only if the first qubit (a) is in state $|1\rangle$. The CNOT gate acts as follows: $\text{CNOT}: \quad |a, b\rangle \to |a, a \oplus b\rangle$. In terms of basis states $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$, its matrix form is:

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{1.22}$$

We can now show the implementation to generate the maximally entangled states as described in equation 1.10 by implementing the Hadamard gate followed by a CNOT, as shown in Figure 1.2.

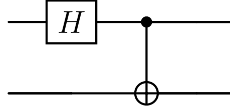


Figure 1.2: `H` and CNOT gate representation.

Wich translates in the following transformations:

$$(\hat{H} \otimes \hat{I})(|00\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) \tag{1.23}$$

$$\text{CNOT}\left(\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)\right) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{1.24}$$

Another important two-qubits implementation is the swap gate, Figure 1.3, which actively interchanges the states of the qubits:

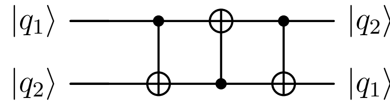


Figure 1.3: `SWAP` gate representation.

This can finally be generalised to a Controlled-U (CU), Figure 1.4, with U any unitary single-qubit gate which get‘s applied to only one qubit as CNOT:

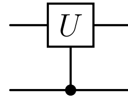


Figure 1.4: General two-qubit gate representation.

In quantum computing, the concept of C-U gates extends the idea of controlled operations, where U represents any single-qubit unitary gate applied similarly to CNOT. These gates operate within the $SU(2^n)$ space, enabling transformations across n qubits.

It can be demonstrated that any n qubit gate can be decomposed into a finite number two-qubit gates. An illustrative example of this concept is the Toffoli gate, often visualized as a controlled-controlled-not gate, Figure 1.5. This gate showcases how intricate quantum operations can be broken down into simpler, repeatable units that involve interactions between pairs of qubits:

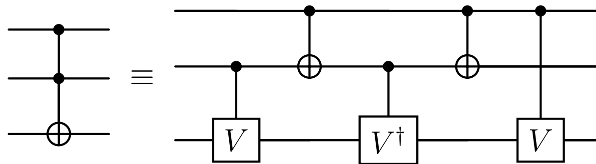


Figure 1.5: Toffoli gate representation (left) and its equivalent circuit (right).

with $V \equiv \frac{(1-i)(I+iX)}{2}$.

By generalizing to $\mathcal{C}^n(U)$ gates, where U denotes a single-qubit unitary gate, quantum circuits can be constructed using sequences of fundamental two-qubit gates. This approach builds upon foundational principles, facilitating scalable and efficient execution of quantum computations.

In classical computing, we can say that `AND, OR, NOT` gates can be used to compute any arbitrary classical function, constituting a universal set of gates. Continuing the analogy with quantum computation, we can state that a set of operators forms a *universal basis* if any unitary operation can be arbitrarily approximated by elements of the set.

$$E(U, V) \equiv \max_{|\psi\rangle} \|(E - V)\,|\psi\rangle\|\,, \tag{1.25}$$

where E represents a quantum gate or operator, V represents a target unitary operation that we aim to approximate, and $|\psi\rangle$ is any quantum state vector. The expression $(E - V)\,|\psi\rangle$ denotes the difference between the action of E and V on state $|\psi\rangle$, and $\|\cdot\|$ denotes the norm.

1.2.2 Clifford Group

An important example of basis gate is the set of *Clifford* gates. These gates are fundamental in quantum computing due to their ability to efficiently manipulate qubits while preserving certain properties. The Clifford gates can be generalized for the case of n-qubits using the Pauli matrices. The Pauli group for n-qubits, denoted as $\mathbf{P}_n$, consists of unitary operators of the form $e^{i\theta\pi/2}\sigma_{j_1} \otimes \cdots \otimes \sigma_{j_n}$, where $\theta = 0, 1, 2, 3$ and σ_{j_k} represents one of the Pauli matrices I, X, Y, Z acting on the k-th qubit.

The Clifford group $\mathbf{C}_n$ is explicitly defined as $\mathbf{C}_n = \{V \in U_{2^n} \mid V\mathbf{P}_nV^\dagger = \mathbf{P}_n\}$. This group consists of all unitary operators V in the $2^n \times 2^n$ unitary group U_{2^n} that satisfy the condition of normalizing the Pauli group $\mathbf{P}_n$.

Clifford gates are then the elements of the Clifford group and can be generated using the Hadamard, the Phase, and the CNOT gates [7]. It's easy to see that Clifford operations on n qubits form a discrete finite set, hence they cannot provide universal representation for arbitrary quantum computations. Indeed, Gottesman and Knill [8, 9] have shown that Clifford gates, along with specific additional elements such as state preparations in the computational basis and measurements in the Pauli group, allow for efficient simulation of quantum computations on a classical computer. This characteristic makes Clifford gates particularly valuable in quantum computing, enabling the simulation of ideal quantum circuit behavior efficiently.

In the realm of physical implementations, Clifford gates offer several distinct advantages. Firstly, they are relatively simple to implement compared to non-Clifford gates, requiring fewer physical resources and operational complexity. This simplicity enhances the feasibility of scaling quantum systems, as fewer error-prone elements are involved in their execution. Secondly, Clifford gates are intrinsically

robust against certain types of errors, making them suitable for error correction schemes such as the surface code. This resilience is critical for maintaining the fidelity of quantum states over extended periods, essential for performing complex computations reliably.

Moreover, beyond the Clifford gate set, the addition of T gates enables a universal approximation capability in quantum computation. The *Solovay-Kitaev* theorem [10] elucidates that any single-qubit quantum gate can be approximated with arbitrary precision using a discrete gate set like $\{H, T, \text{CNOT}\}$. This theorem ensures that even very small errors $\epsilon > 0$ can be managed with a sequence of gates whose length scales as $\log(1/\epsilon)$. This capability is crucial for constructing complex quantum circuits on actual quantum hardware, where precise and reliable gate operations are paramount. The practical application of the Solovay-Kitaev theorem are vast, as they create a way for the efficient implementation of quantum algorithms and the realization of quantum computational advantages across various quantum computing architectures.

1.2.3 NISQ devices

Noisy Intermediate-Scale Quantum (NISQ) technology is a term coined by John Preskill in 2018 [11] to indicate the modern noise-affected scale of quantum computers. One of the biggest problem with quantum computing is in fact the noise generated by unitary operations and the decoherence of the phase. Both get worse once we increase the number of qubits in our Quantum Processing Unit (QPC). The state of the art quantum computer, considering the most number of qubits in the QPC, is the 1121-qubit Condor processor from IBM which is already a big number of qubits and in 2025 are expected two more quantum computers from IBM with fully quantum connections between chips that allow the quantum information to flow between different processors unhindered, enabling truly large-scale quantum computation. These devices, while not yet capable of achieving fault-tolerant quantum computation due to current noise levels and limited qubit coherence times, still potentially promise solving computationally intense problems.

The main problem with quantum computing is, in fact, their susceptibility to the external factors like changing in temperature, electromagnetic fields, vibrations and, in general, any kind of perturbation that can change the state of single qubits.

This perturbations usually cause:

- **Bit-flip errors**: These errors flip the state of a qubit from $|0\rangle$ to $|1\rangle$ or vice versa.
- **Phase-flip errors**: These errors change the phase of a qubit, flipping the sign of the $|1\rangle$ component.
- **Depolarizing errors**: A combination of bit-flip and phase-flip errors.

The majority of research in quantum computing is, in this particular period, concentrated on the development of new techniques for *error correction* and *error mitigation*.

The first one is focused on resolving problems in the real-time computing part of the computation. This is achieved by utilising logic qubits (analogous to logic bits in classical computing), which act in a manner analogous to normal qubits, with the exception that they are composed of numerous physical ones whose collective properties define the logic's state.

The second technique is implemented as a post-analysis of the output generated by the quantum computer. It frequently employs machine learning, algorithmic techniques, system modelling of the noise in order to extract the clean data. One of the principal challenges in this field is the scalability of the technique in relation to the number of qubits and the complexity of the system.

1.2.4 Qiskit framework

Qiskit (Quantum Information Science Kit) is a Python open-source software framework for quantum computing developed by IBM. It allows to create and execute quantum algorithms on IBM Quantum simulators as well as real processors. Qiskit provides tools for various stages of quantum computing, including designing circuits, running experiments, and analyzing results. It's aimed at making quantum computing accessible for researchers, developers, and educators.

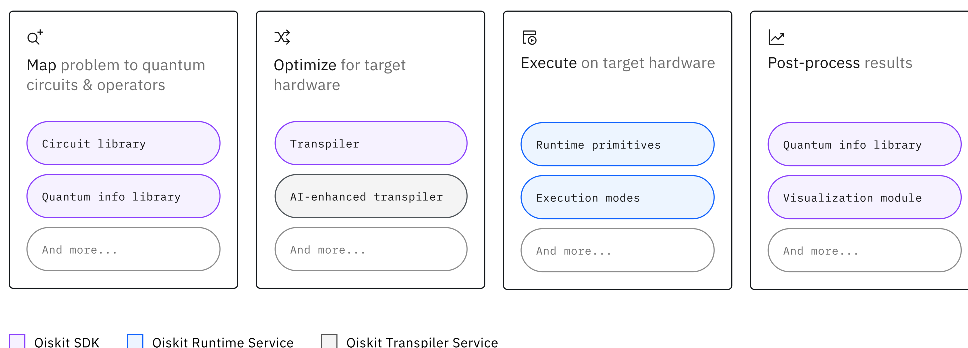


Figure 1.6: Overview of the Qiskit framework architecture [12].

Qiskit library consists of several main modules, as depicted in Figure 1.6:

- **Circuit library**: it offers a comprehensive set of tools to create and manipulate quantum circuits, registers, gates, and instructions, including parameterized and conditional operations. It also provides a diverse collection of pre-built quantum algorithms, serving as fundamental building blocks for more advanced quantum programs;
- **Quantum info module**: it provides functionalities for precise manipulation and analysis of quantum states, operators, and channels, enabling exact quantum evolution of state vectors and density matrices;
- **Circuit transpiler**: it compiles quantum circuits to run on a specific target backend, adapting them to device-specific properties and enhancing performance through various optimization passes;

- **Quantum Primitives**: it contains foundational components such as the *Sampler* (drawing samples from a quantum state) and the *Estimator* (estimating expectation values of quantum operators) primitives, serving as base definitions and reference implementations for quantum computation tasks;
- **Qiskit Runtime**: it is a cloud-based service acting as a client to enable the remote execution of quantum algorithms on real IBM Quantum devices.

Chapter 2

Machine Learning

2.1 Introduction to Machine Learning

Machine Learning (ML for short) is a field of computer science that develops algorithms capable of self-improving through experience. In other words, "Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed" (Arthur Samuel, 1959) [13].

In recent years, this kind of programming has spread across various fields thanks to the increased volume of data available for training, better computing power, and improvements in training algorithms. Machine learning is often developed in data science for its ability to analyze large datasets and extract information that would otherwise be inaccessible [14, 15, 16].

The problems one encounters in the development of ML algorithms usually involve the training process. There is a substantial amount of theory behind choosing the right type of ML implementation, the kind of optimization process, dataset preparation, and the libraries and APIs to use. In this section, we will provide a general overview of all these topics, with particular attention to those relevant to the purpose of this thesis.

2.1.1 Learning paradigms

The name Machine learning is actually used for a large set of different types of algorithm. In particular we can enclose them in four categories depending on whether or not, and to what extent, they need human supervision during training:

1. Supervised learning: this type of algorithm learns from labeled data, meaning each input data point is paired with an output label that represents the correct answer[17]. The algorithm's goal is to learn a mapping from inputs to outputs that can be applied to new, unseen data. Common tasks in supervised learning include:

    - Classification: assigning input data into predefined categories or classes;
    - Regression: predicting a continuous output value based on input data.

2. Unsupervised Learning: in this approach, the training dataset is unlabeled, and the algorithm attempts to identify patterns and structures in the input data, finding connections and relationships within it. Some application are:
    - Clustering: grouping data points into clusters based on their similarities;
    - Anomaly Detection: identifying outliers or unusual data points;
    - Dimensionality Reduction: reducing the number of features in the data while preserving the information the data represent.
3. Semi-supervised learning: This approach combines labeled and unlabeled data, typically with a larger portion being unlabeled. It involves an unsupervised algorithm, or a stack of unsupervised algorithms, which is fine-tuned with supervised learning [18, 19]. Common methods include:
    - Self-training: the algorithm generates pseudo-labels using the labeled data and then proceeds to train on those.
    - Co-training: two classifiers are trained on labeled data and enhance each other's predictions through iterative retraining.
    - Graph-based self-training: uses graph-based label propagation to label all data with an accuracy coefficient.
4. Reinforcement learning: which is fundamentally different since we use a system, called agent, that can "observe" the environment and perform action which cause a penalty/reward system to find the optimal policy for the task, by maximising the rewards. This type of learning is often used in robotics, for example in learning how to walk or how to play games.

2.1.2 Gradient-based optimization

We discussed various types of learning algorithms without delving into how they actually "learn". This can be implemented by evaluating some kind of function that represents how well our model completes the tasks and varying the function parameters in order to find the optimal ones.

A general implementation is made considering a function $\vec{y} = f(\vec{x}, \vec{\theta})$, where $\vec{\theta}$ represents a series of parameters, whose value we want to be close to the truth label value associated to the input $y_{truth} = g(x)$. To achieve this, we define a loss function $J(\vec{\theta})$ that we want to minimize to enhance our model (for supervised regression it usually is the mean squared error or MSE). One widely used method to optimize these parameters involves Gradient Descent (GD), initialized randomly with a vector of random parameters $\vec{\theta}$ [17, 20]. This approach hinges on computing the gradient of the loss function relative to the parameters, which guides adjustments to $\vec{\theta}$ in a direction that minimizes J. Through iterative refinement, we progressively approach a minimum of the loss function.

In this approach, for which a graphical implementation is shown in Figure 2.1, we can consider the directional derivative with versor $\hat{u}$ of the function $J(\vec{\theta} + \alpha\hat{u})$

with respect to α in $\alpha = 0$

$$\frac{\partial}{\partial \alpha} J(\vec{\theta} + \alpha \hat{u}). \tag{2.1}$$

To minimize J, with this step-iteration method, we want to find the fastest decreasing direction for J:

$$\begin{aligned} & \min_{u,\, u^\top u = 1} \quad u^\top \nabla_\theta J(\vec{\theta}) \\ = & \min_{u,\, u^\top u = 1} \quad \|u\|_2 \|\nabla_\theta J(\vec{\theta})\|_2 \cos \phi \end{aligned} \tag{2.2}$$

with ϕ the angle between $\hat{u}$ and the gradient. And to calculate the next vector:

$$\vec{\theta}' = \vec{\theta} - \lambda \nabla_{\vec{\theta}} J(\vec{\theta}) \tag{2.3}$$

where λ is a hyper-parameter that controls the step length of parameter variations.

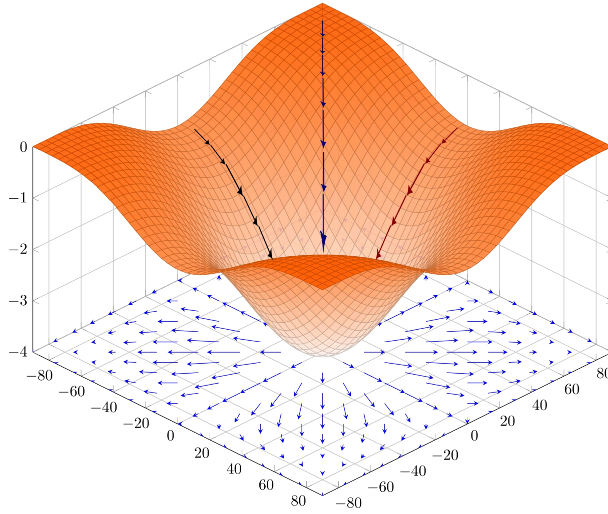


Figure 2.1: Visualization of a function landscape explored by Gradient Descent [21].

We can now show an implementation using MSE and $h(\vec{x})$ as the true labeled value:

$$J(\vec{\theta}) = \frac{1}{2N} \sum_{i=1}^{N} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{2.4}$$

then

$$\theta_j = \theta_j - \lambda \frac{\partial}{\partial \theta_j} J(\vec{\theta}) \qquad \frac{\partial}{\partial \theta_j} J(\vec{\theta}) = \frac{\partial}{\partial \theta_j} \left[\frac{1}{2N} \sum_{i=1}^{N} (h_\theta(x^{(i)}) - y^{(i)})^2 \right] \tag{2.5}$$

Iterating this algorithm allows us to converge to a minimum, not necessarily a global minimum, of the loss function and thus to an optimal set of parameters. Even though we cannot be sure we reached a global minimum, we still are able to utilize

this procedure as long as the cost function is small enough, as shown in Figure 2.2. We can still make better model by implementing methods like stochastic GD, considering the momentum of the gradient, hyper-parameters tuning and utilizing some instances of the dataset to select the best performing ones for validation and testing.

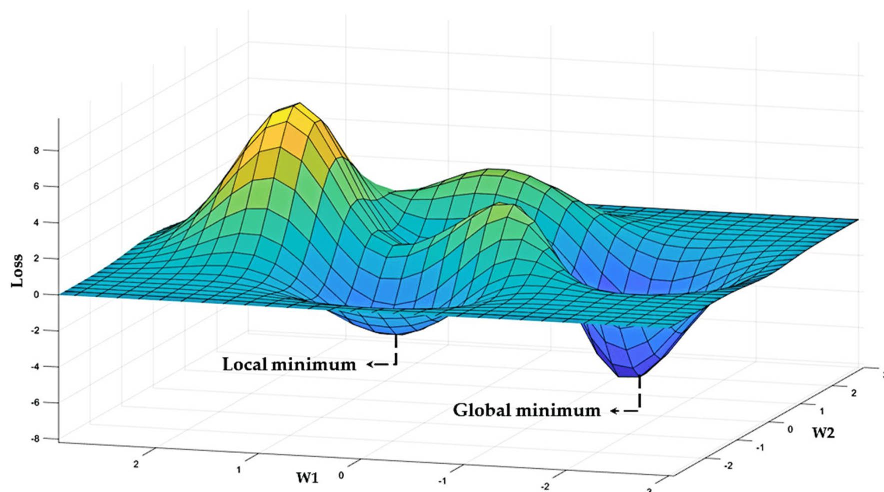


Figure 2.2: Global and local minima.

2.1.3 Supervised learning

As anticipated supervised machine learning is a technique based on the training of our model with labeled data. The aim of the algorithm is then to understand and emulate the labeling process used in the dataset and to generalize to new unseen data [16]. This approach is mainly used in regression, where the output to reconstruct is a continuous one, and classification where the model has to predict on a discrete set of values. These tasks find applications in many fields of science such as medical diagnosis, image recognition, financial forecasting, data science and many more, and they are mostly implemented as:

- Linear regression: it consists in studying the data assuming a true form of a straight line, often trained with GD and RMSE for the loss function. The aim is simply to compute a weighted sum, plus a constant for the inputs.
- Support Vector Machines: SVMs are powerful implementations of ML capable of linear and nonlinear classification and regression. Used for complex but medium-sized datasets, they consist of the identification of linear decision boundaries (lines that separate the data with different labels).
- Decision Trees: DTs are versatile models, used for classification and regression, characterized by a series of "questions" in the decision nodes, consisting of testing the data for a particular property until reaching a leaf node which is going to be the final label for the data.
- Artificial Neural Networks: ANNs encompass various architectures with developed with supervised learning , such as Feed-Forward Neural Networks

(FFNNs), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs). These models differ in structure and application: FFNNs have data flowing in one direction, CNNs are optimized for visual recognition with a filter mechanism, and RNNs handle sequential data and can process inputs of arbitrary lengths. Some other details about ANNs will be discussed in Section 2.2.

2.1.4 Training, validation, and testing

When training a machine learning program, our goal is to minimize loss and so maximize the accuracy of our algorithm. However, this process primarily evaluates how well our algorithm performs on the data it has been trained on and not how it generalise to new data.

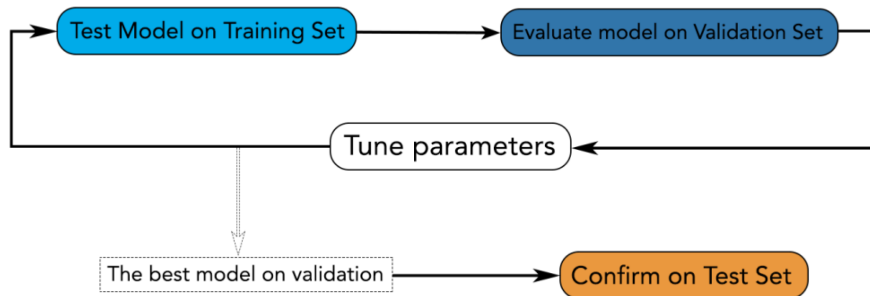


Figure 2.3: Training procedure with training, validation and test sets.

To effectively select the best parameters and hypotheses on the training set and to mitigate overfitting — where the model memorizes noise and specific dataset characteristics rather than learning underlying patterns — we must evaluate its performance on new, independent data [17, 14], as shown inFigure 2.3.
One of the most used solution is to split our training data into three different part: the training, the validation and the test set:

- the first one is used in the actual learning algorithm to find the best parameters and minimise the loss function;
- the validation set is crucial for tuning hyperparameters, which directly influence the training process and ultimately the performance of the final model, helping us select the best-performing algorithm;
- the test set remains unseen until the final evaluation phase, providing an unbiased evaluation of the model's performance after it has been tuned and validated on the best-performing algorithms.

2.2 Artificial neural networks

Artificial Neural Networks (ANNs) are a particular set of ML algorithms that try to mimic the functioning of the brain. This attempt at emulating real neurons is made by creating a closed set of deeply interconnected processing units that, as a whole, exhibit some features of real neural networks[17, 22].

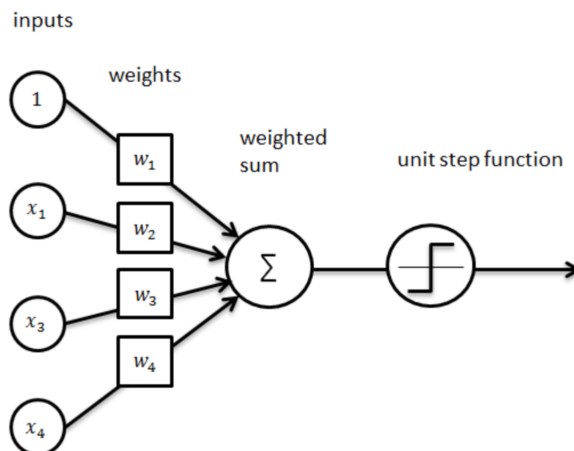


Figure 2.4: Threshold Logic Unit (TLU) representation.

One of the first and simplest forms of ANNs is the Threshold Logic Unit (TLU), shown in Figure 2.4, which acts like a simple linear binary classifier by taking numeric inputs with associated weights and outputs a step function based on their sum. Using TLUs, we can create a perceptron by adding a bias neuron to the input, which always outputs one, and connecting the inputs to a layer of TLUs. The bias neuron is practically used to shift the activation function of TLUs in different ways, depending on the associated weight and data, enhancing the learning process.

To train a perceptron, we take inspiration from real neural networks. When two neurons in our brain interact with each other, their connection gets stronger, which is akin to "training" the neurons. For a perceptron, we want to strengthen the connections that would have made the correct prediction on the input data. We do this by changing the values of the weights as described in general gradient-based optimization 2.5.

However, this type of implementation is limited to linear decision boundaries. With just a single TLU, we cannot even learn slightly more complex functions, like for instance the XOR (exclusive-OR). This is due to the fact that the classification is made with a straight line decision boundary and gives a binary result. Nevertheless, we can attempt to stack many perceptrons to create a Multi-Layer Perceptron (MLP) and to chose an optimal activation function in order to be able to analyze more complex data.

2.2.1 Multi-Layer Perceptron

A Multi-Layer Perceptron (MLP) is a Feed-Forward Neural Network (FFNN) made up of a series of fully connected layers of TLUs, as is qualitatively shown in Figure 2.5. The layers between the input and the outputs are called the hidden layers, and each one has a bias neuron.

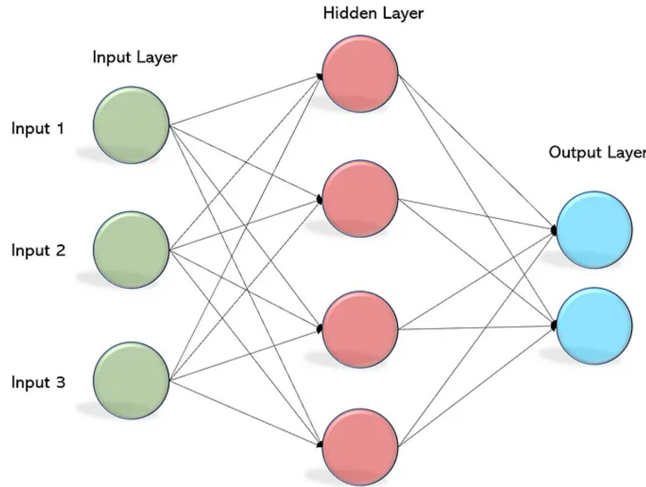


Figure 2.5: Multi-Layer Perceptron (MLP) architecture.

Let's consider a general MLP with $a_i^{(j)}$ representing the activation (value passed forward) of the ith neuron in the jth layer, $\mathbf{H}^{(j)}$ representing the weight matrix (with dimensions depending on the number of layers and neurons per layer) from layer $j-1$ to j, g is the activation function, and N as the number of TLUs of the $j-1$ layer:

$$a_i^{(j)} = g\left(\sum_{k=1}^{N} H_{i,k}^{(j)} \cdot a_k^{(j-1)}\right) \tag{2.6}$$

The output will then be the activation of the final layer of the network. It is now straightforward to consider the inputs $\vec{x}$ as the activation of the first layer:

$$\vec{a}^{(0)} = g(\mathbf{H}^{(0)}\vec{x}) \tag{2.7}$$

This allows us to iterate the substitution and calculate the final output $h = H(x)$; this process is called forward propagation. Thanks to this process, we can compute complex operations, where each layer acts like a tensor operation that can be seen as a geometric transformation of the input data, making the algorithm flexible.

To train MLPs, we use gradient-based optimization. The challenge with this approach is that our structure is implemented as a FFNN, and performing the iteration required for this method can be complex and time-consuming. Instead, we can utilize the fact that every operator used is differentiable, allowing us to compute the gradient of the operator and apply the chain rule:

$$\frac{\partial}{\partial x} f(g(x)) = f'(g(x))g'(x) \tag{2.8}$$

The Back-Propagation (BP) is an efficient technique for implementing GD optimization. It begins with a forward- followed by a backward- propagation of the information trough the layers. In order to compute the gradient of the loss function relative to each parameter of the model, reverse mode auto-differentiation (automatically computing gradients) is used. Practically it calculates how a small variation in each parameter affects the final output so it can be changed to the parameter with the lowest loss function. It is important to choose a proper activation function for the TLU units when using GD and BP, particularly since we compute the gradient. It is best to have an activation function that is continuous and differentiable, and to randomly initialize the weights, in order to break the symmetry and enhance the training process.

Implementing complex MLP with many layers (called *deep NNs*) capable of complex classification and regression tasks can be challenging because of training. Possible problems linked to the training of deep neural networks are the phenomena of exploding and vanishing gradients that make the training of lower layers very harsh. A simple solution has been shown to be the optimal choice of activation functions, which can affect the first part of training. To further solve it, we can implement a method called batch normalization, which consists of normalizing and centering the input of the activation function, then scaling it and shifting it with just 2 parameters per layer (this way, it is not necessary to standardize the dataset). It is called batch normalization since we normalize by calculating the means and variances of inputs divided into mini-batches. This type of implementation has been shown to avoid the poor solutions one gets using only random initialization [23].

2.2.2 Autoencoder architecture

A widely-used type of FFNNs is the so-called Autoencoder (AE). This neural network architecture is characterized by an input and an output layer of the same size, as well as two symmetric series of hidden layers usually denoted as encoder and decoder (see Figure 2.6) [24]. The aim of the AE is to compress the input essential information into a lower-dimensional representation (latent space) and try to reconstruct it.

Formally, the encoder and the decoder are general parameterized functions such that:

$$e_\theta : \mathcal{S} \longrightarrow \mathcal{L} \qquad d_\phi : \mathcal{L} \longrightarrow \mathcal{S} \tag{2.9}$$

where θ and ϕ represents the trainable parameters of the encoding and decoding functions respectively, $\mathcal{S}$ is the input and output space, and $\mathcal{L}$ is the low-dimensional latent space. Training the autoencoder means to search for the optimal set of parameters to minimize a loss function F which depend on a given distance metric computed between the input and the output of the AE itself. In particular:

$$\theta, \phi = \arg\min_{\theta,\phi} \left(F(|x - d_\phi(e_\theta(x))|) \right) \tag{2.10}$$

where $x \in \mathcal{S}$ is the input vector and $d_\phi(e_\theta(x)) = d_\phi(z) = \hat{x} \in \mathcal{S}$ is the output, with $z \in \mathcal{L}$ denoting the compressed representation in the latent space.

Over-training the AE network would lead to $\hat{x} = x$ but we would have a useless model that simply outputs a copy of the input. The output of the composite function is then to be thought of not as an exact reconstruction of x but as a probabilistic representation centered on x but not matching it exactly. To achieve this, we exploit the autoencoder bottleneck to produce a compressed and under-complete representation of the input, thus forcing this latent vector z to encode only the essential information about x and then trying to reconstruct it.

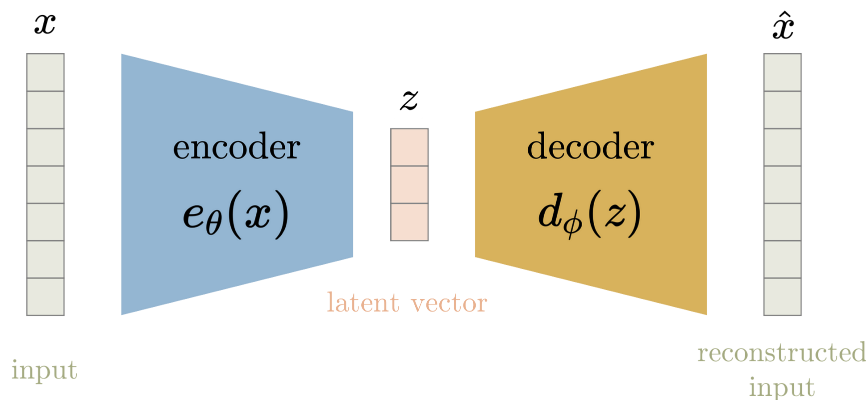


Figure 2.6: General architecture of an autoencoder neural network.

A Denoising Autoencoder (DAE) is a task-specific type of autoencoder whose aim is to reconstruct an original input from its noisy version. In this supervised context, the noisy vector gets labeled with the noiseless data so that a good denoised output can be obtained from the corrupted input. So, the model is supposed to learn how to extract the relevant information from the input and filter out the effects due to noise.

2.2.3 TensorFlow framework

Implementing a machine learning algorithm has become relatively easy thanks to powerful end-to-end libraries that have been developed in recent years like TensorFlow and high-level API like Keras [25]. In this section, we briefly present the TensorFlow framework and introduce the basic concepts underlying its development.

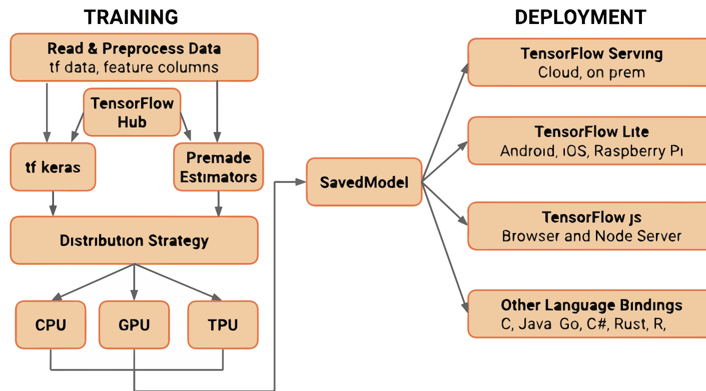


Figure 2.7: Simplified scheme of the TensorFlow architecture.

TensorFlow is a powerful tool designed for heavy computation, and in particular, well-suited and tuned for machine learning algorithms. It is being developed by the Google Brain team and published as open-source in late 2015 [26, 14]. Its architecture is implemented as shown in Figure 2.7.

At the lowest level, TF is made with C++ with its operations (or ops) transparently portable to different kernels for different processing units like CPUs (Control Processing Units), GPUs (Graphic Processing Units), and TPUs (a particular type of chip made by Google which is optimized for tensorial calculus). It is based on graph modeling (data flow graph), which consist in a data structure that defines the sequence and relationships between operators in a graph, that can be implemented using:

- **Tensors**: Tensors represent data as n-dimensional arrays of `int32`, `float32` or `string`. All tensors in TF are *dense* (dense, as opposed to sparse tensors where we store only non zero values, are multi-dimensional arrays where all element are explicitly stored), in order to facilitate simple implementation and iteration at the lowest level.

- **Operations**: An operation takes one or more tensors as input and produces one or more tensors as output. Each operation has a specified type and may have compile-time attributes that determine its behavior. Additionally, there are variable operations that have a mutable state, which is read or written each time the operation executes.

Chapter 3

Autoencoder for Error Mitigation

As anticipated in the subsection 1.2.3, one of the most important objectives regarding quantum computing research is error correction and, in parallel, error mitigation. The aim of this thesis is to utilise the concepts described thus far to implement a functioning autoencoder, thereby enabling error mitigation on a given quantum computer with a fixed number of qubits. To this end, we will create a model of a Denoising Autoencoder (DAE) and train it on a dataset comprising noisy simulations of real-world circuits, each labelled with the corresponding ideal simulation. In order to enhance the training of the autoencoder, we will apply a Gaussian form to the output of our circuits using an optimizer that can calculate the parameters of the gates to shape the output as desired. This process has been implemented with the objective of establishing a framework within which the autoencoder can be facilitated in discerning the impact of noise from the ideal output and to generate a distribution that is plausible in a real-world context.

The objective is to create a machine learning model able to mitigate the noisy output data of a generic quantum circuit. This model is intended to be a proof of concept. Indeed, due to resources constraints, we were not able to run the circuit on real quantum hardware but we used classical simulators including quantum noise models, which of course are quite inefficient as the size of the quantum system increases. With a real gate-based quantum computer, this process would be intrinsic to the device itself, enabling the same kind of study with larger circuits and in a fully realistic environment.

3.1 Data preparation

To train the DAE, it is necessary to create a sufficiently large dataset. This dataset was generated as pairs of ideal and noisy probability distributions, represented as histograms with a number of bins matching the 2^n computational basis states for an n-qubits quantum circuit. In our case we decided to set $n = 5$, since our final aim is just to validate the effectiveness of this QEM approach in a simple case. As previously discussed, the whole dataset should include all the instances required for training, validating, and testing the DAE model.

3.1.1 Circuits generation

We started choosing a few different Parameterized Quantum Circuit (PQC) architectures commonly used in quantum machine learning and quantum optimization problems. To make our study as much realistic as possible, these architectures are selected from a dataset of commonly used quantum circuits, made available online for benchmarking purposes [27]. All the circuits have 5 qubits and a varying number of trainable parameters, large enough to make the PQC sufficiently expressive to learn the given target probability distributions. For the generation of these target distributions, represented as 32-bins histograms, we used only data sampled from a normal $\mathcal{G}(\mu, \sigma)$ function, varying μ and σ in a fixed range of allowed values.

Circuit instances are then obtained by assigning different sets of specific values to the PQCs parameters. These values are computed as the result of a classical optimization process designed to prepare a particular quantum state $|\psi_i\rangle$ whose basis states probabilities match a given random histogram h_i previously generated. This process was performed using Pennylane, an open-source quantum framework specifically designed for quantum machine learning [28].

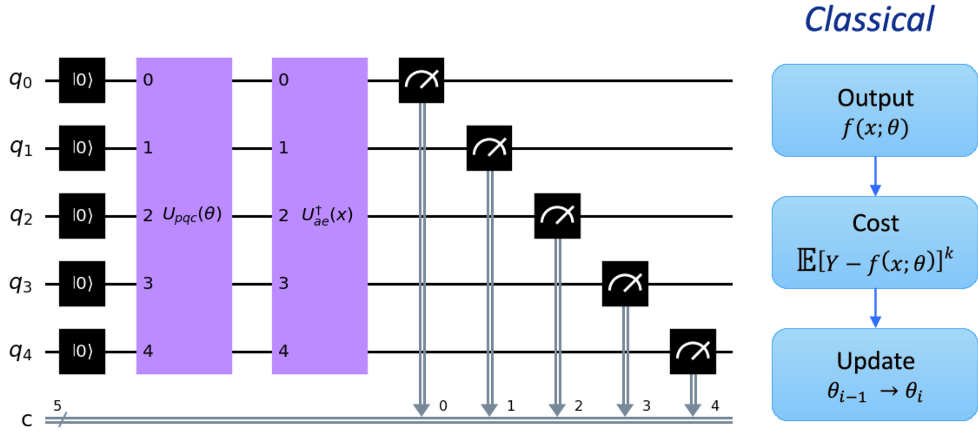


Figure 3.1: Overview of the hybrid quantum-classical workflow to train each Parameterized Quantum Circuit (PQC) and learn any given basis state probability distribution.

In the following, we give a more detailed explanation about the approach adopted to train each single quantum circuit in the dataset (see Figure 3.1 for an overview of the entire workflow).

Let's start from a circuit initialized in the state $|0\rangle^{\otimes n}$ (where n is the number of qubits). First, we apply the unitary $U_{pqc}(\theta)$, parameterized by the set of parameters θ and representing the quantum circuit we want to train. Then, we apply a second unitary $U_{ae}^{\dagger}(x)$ implementing the adjoint (or inverse) transformation of the amplitude embedding routine, used to prepare the state x that encodes the target probability distribution. The goal is to find the optimal parameters configuration θ^* in order to have:

$$U_{pqc}(\theta^*) = U_{ae}(x) \tag{3.1}$$

Notice that the problem now actually reduces to train the PQC so that the unitary transformation corresponding to the whole circuit is simply the identity, mapping the state $|0\rangle^{\otimes n}$ to itself. A quite robust way to do this in practice is by minimizing the following cost function:

$$\mathcal{C} = (1 - P^2(\{0\}^n))^{1/2} \tag{3.2}$$

where $P^2(\{0\}^n)$ is the square of the probability to measure the bitstring $\{0\}^n$, corresponding to the quantum state $|0\rangle^{\otimes n}$. The value of this cost function is bounded in $[0, 1]$ and, in particular, is equal to 0 only in the case where the probability of sampling $\{0\}^n$ is 1, meaning that we were able to find the optimal parameters θ^*. The actual code implementation of this optimization procedure is shown in Appendix B.

3.1.2 Clifford ideal simulation

The Clifford ideal simulation represents a method for accelerating the computation of ideal distributions. Clifford gates constitute a specific class of gates belonging to the Clifford group, as described in Section 1.2.2. Thanks to their particular mathematical properties, they are efficient to simulate classically. The basis of the Clifford group is composed of gates H, S, and CNOT and does not constitute a universal basis. In order to represent a general circuit, it is necessary to add the gates $T = \sqrt{S}$ and $T^\dagger$ to form a universal gate set. The utilisation of this extended set of gates enables the application of the Solovay-Kitaev.

The extended Clifford group was employed to facilitate the efficient computation of the ideal distribution of the approximation obtained with the Qiskit `SolovayKitaev` transpiler, which identifies the optimal approximation of the passed circuit for the basis gates set.

Finally, we used the Qiskit `Statevector` object to exactly compute the final output state vector according to the unitary evolution defined by the transpiled quantum circuit. This method allows to get the exact probabilities of each computational basis state, constituting the target noiseless distributions in our dataset.

3.1.3 Noisy simulation

The initial point of departure for the noisy simulation is the approximated circuit employed in the ideal simulation. In the absence of access to a real quantum computer, it is imperative to simulate a realistic noise model. The overall choice of these custom noise model parameters is summarized in Table 3.1 and its implementation shown in Appendix A.

Qiskit is a comprehensive library for quantum computing that provides the so-called `AerSimulator`, which encompasses a variety of quantum circuit simulation methods. Specifically, we selected the specific IBMQ `Sherbrooke` device to access the corresponding coupling map, thereby accurately reflecting the topology of a real machine in the noisy circuit simulation.

Parameter	**Value**
Relaxation time T1	2 ms
Dephasing time T2	1.5 ms
Single-qubit gate time	75 ns
Two-qubits gate time	100 ns
Depolarization probability	1×10^{-5}
Readout error	$\begin{bmatrix} 0.98 & 0.02 \\ 0.02 & 0.98 \end{bmatrix}$
Coupling map	Sherbrooke IBMQ backend

Table 3.1: Summary of the parameters values defining the circuit noise model.

Table 3.1 outlines the parameters of a custom noise model used for the noisy quantum circuit simulation.

- **Relaxation time T1**: it measures how long a qubit can maintain its state (typically the ground state) before losing coherence due to relaxation processes. A longer T1 time indicates better coherence and lower state decay.
- **Dephasing time T2**: it measures how long a qubit can retain its phase coherence without external perturbations causing phase errors. A longer T2 time indicates longer coherence and better resistance to dephasing.
- **Gate times**: they represent the time required to perform single-qubit and two-qubit unitary gate operations. Shorter gate times minimize exposure to environmental noise and reduce the likelihood of errors during quantum gate operations.
- **Depolarization probability**: it indicates the likelihood of a qubit becoming a mixed state due to interactions with the environment. A lower depolarization probability corresponds to higher qubit fidelity and stability against noise-induced errors.
- **Readout error matrix**: it models the error probabilities of different measurement outcomes for each qubit. Higher diagonal values indicate higher readout accuracy (lower readout errors), while off-diagonal elements represent misread outcomes.
- **Coupling map**: The coupling map defines the allowed connections (couplings) between qubits in a quantum processor. It outlines which qubits can interact directly with each other via two-qubit gates. The coupling map's configuration impacts gate implementation efficiency and error rates, as operations involving non-neighboring qubits may require additional resources and introduce higher error probabilities.

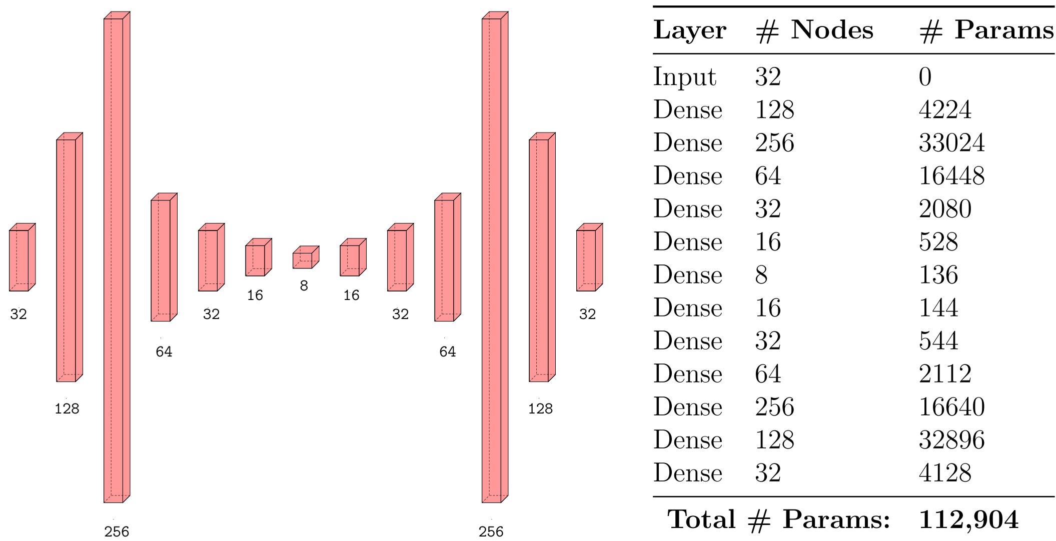
These parameters have been tuned to resemble a real IBMQ superconducting device but at the same to introduce a significant amount of noise to test the effectiveness of our mitigation technique.

To run both the ideal and noisy quantum circuit simulations, we employed

the Qiskit *AerSimulator* setting a fixed number of shots = 10^5. This number of samples (for each quantum circuit run) should be large enough number to properly approximate the final state probability distribution.

3.2 Model definition

The final component of our error mitigation technique is the autoencoder. The autoencoder denoiser was trained using the **TensorFlow** library and **Keras**, a high-level API within TensorFlow, was used to define the architecture of the autoencoder, including the layers, nodes, activation functions, and parameters for optimal learning.



Layer	# Nodes	# Params
Input	32	0
Dense	128	4224
Dense	256	33024
Dense	64	16448
Dense	32	2080
Dense	16	528
Dense	8	136
Dense	16	144
Dense	32	544
Dense	64	2112
Dense	256	16640
Dense	128	32896
Dense	32	4128
Total # Params:		**112,904**

Figure 3.2: Schematic representation of the autoencoder (left) and detailed summary of its fully-connected architecture (right).

Specifically, we implemented an autoencoder framework based on the number of qubits, the fact that we were working on a binned PDF distribution and the general form of the introduced noise. The model employs a layered architecture with `LeakyReLU` activations, which introduces non-linearity while mitigating vanishing gradient issues during training. The only parameter needed to define the `LeakyReLU`, which controls the slope of the straight line for $\vec{x} < 0$, has been set to 0.2, a value that is commonly employed in similar implementations. As previously stated in section 2.2.2, the structure is symmetrical with respect to the latent space layer, which has a dimension of 8, as can be seen in Figure 3.2. Finally, on the final layer, the outputs are defined with a `SoftMax` activation function, which constrains the data to be contained within the interval $[0, 1]$ and ensures that the total sum is equal to 1.

The data include a total number of 5000 instances as pairs of noisy-ideal output probability distributions. The sample was then split into training, validation, and testing set in proportions of 0.7, 0.2, and 0.1, respectively. The autoencoder was

trained over 150 epochs, using a batch size of 64 examples and the `ADAM` optimizer [29]. Furthermore, the selection of the loss function is of critical importance in guiding the learning process towards the effective denoising of the input data. The model was trained multiple times by using different losses commonly used in ML: Mean Absolute Error (MAE), Mean Squared Error (MSE), Kullback-Leibler divergence (KL), and Cross Entropy (CE). For a better robustness in the training convergence rate, we decided to use the MSE loss function, defined as:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2 \tag{3.3}$$

where N is the number of bins, $\hat{y}_i$ are the ideal probabilities, and y_i are the corresponding denoised quasi-probability computed by the model.

3.3 Data analysis

The proposed model has been able, with a certain precision margin, to restore and clean the data from the noise generated by the quantum processor as expected. We are now going to show the data obtained and the respective analysis, giving the graph for the metric used to analyze the quality of the framework. Lastly we are going to draw some conclusions and show how this kind of approach can be implemented.

A first preview of how the autoencoder will perform can be suggested tracking the train loss history and the validation loss history shown in Figure 3.3.

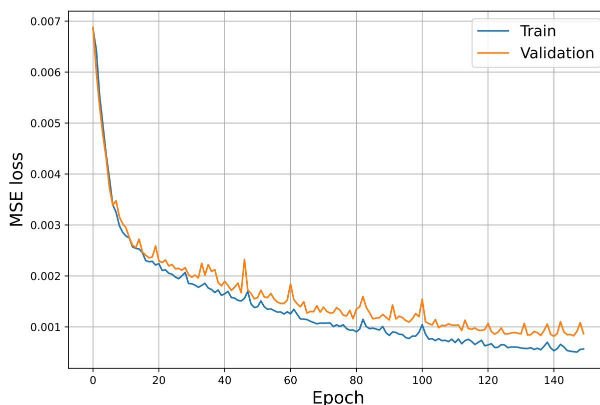


Figure 3.3: Train and validation loss as a function of the training epochs.

The data indicates that the learning curve of the model, both during training and validation, is declining. This suggests that the model is improving. Additionally, the nearly flat shape of the final portion of the curve indicates that the model is converging to a minimum, and so, to an optimal set of parameters.

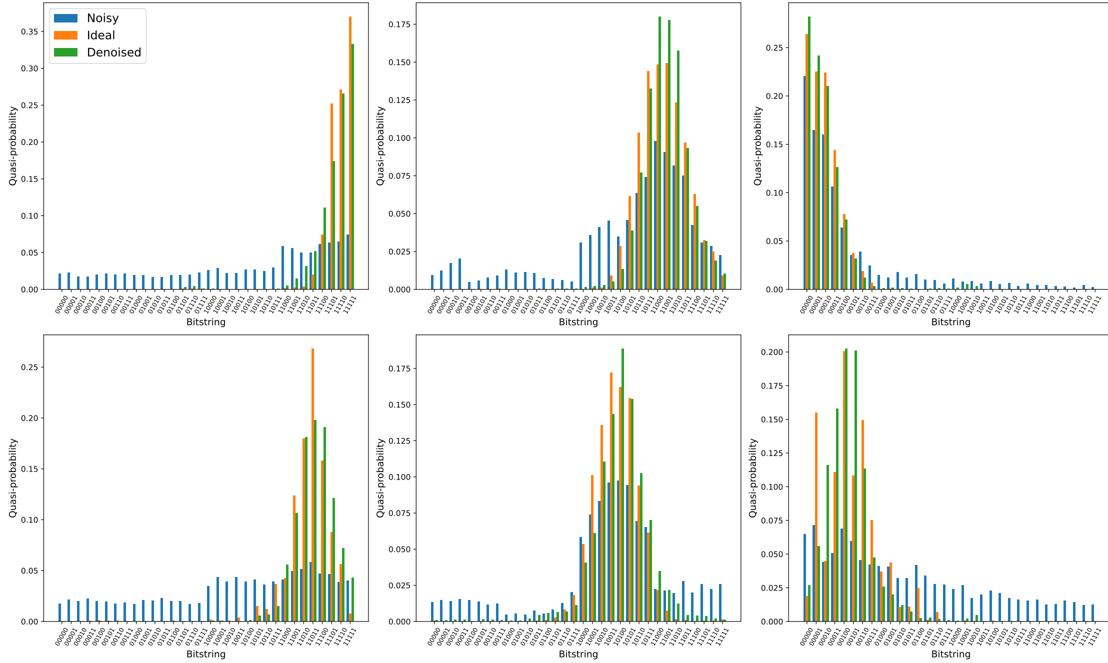


Figure 3.4: Example histograms showing a comparison between the noisy, ideal, and denoised final state probability distributions.

We can now show the output of the model compared to the ideal and the noisy simulations in Figure 3.4 and start to analyze if and how well the autoencoder matches our expectations.

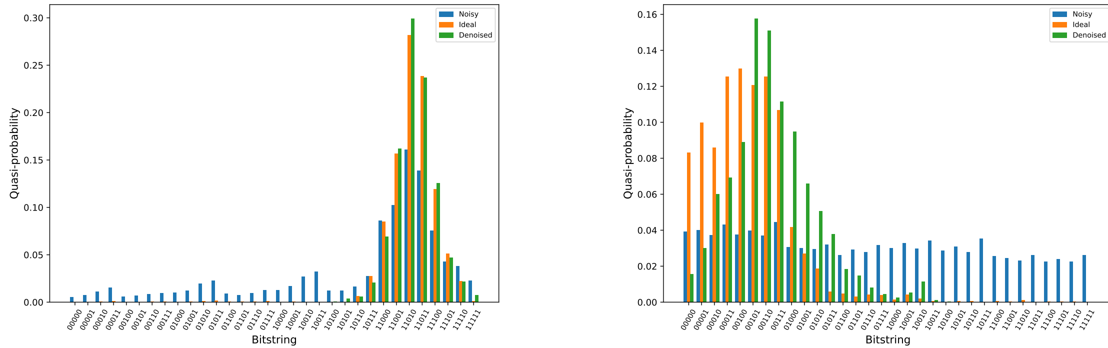


Figure 3.5: Example of optimal (left) and sub-optimal (right) result for two specific histograms drawn from the data sample.

It is evident that the denoised data is a better representation of the ideal distribution than the noisy distribution. While there are minor discrepancies in the height of the peak centre, the majority of the bin heights are close to the ideal. Significantly, in most cases the model is able to effectively eliminate the bit-strings that have non-zero values solely due to the introduced noise. However, in a small percentage of cases, the approximation of the shape of the denoised distribution does not align closely with the ideal one. Some representatives of these examples are shown in Figure 3.5.

Finally, for the evaluation of the results, we computed the Wasserstein distance metric, which is the best choice when comparing one-dimensional probabilities distributions [30] The Wasserstein metric $W_1(P, Q)$ between two probability distributions P and Q on a metric space $(\mathcal{X}, d)$, in the case of discrete distributions, is defined as:

$$W_1(P, Q) = \min_{\gamma \in \Gamma(P,Q)} \sum_{i,j} d(x_i, x_j)\gamma_{i,j}, \tag{3.4}$$

Here, $\Gamma(P, Q)$ denotes the set of all transportation plans γ where $\gamma_{i,j} \geq 0$ for all i, j. A transportation plan γ specifies how probability mass is transferred from points x_i to x_j, with $\gamma_{i,j}$ indicating the amount transported. The constraints $\sum_j \gamma_{i,j} = P(x_i)$ ensure that the total mass transported from x_i equals $P(x_i)$, and $\sum_i \gamma_{i,j} = Q(x_j)$ ensures that the total mass received at x_j equals $Q(x_j)$, thereby adhering to the probabilities P and Q. The term $d(x_i, x_j)$ represents the distance between points x_i and x_j.The total mass transported is to the overall quantity of probability mass moved between sets of points as defined by a transportation plan.

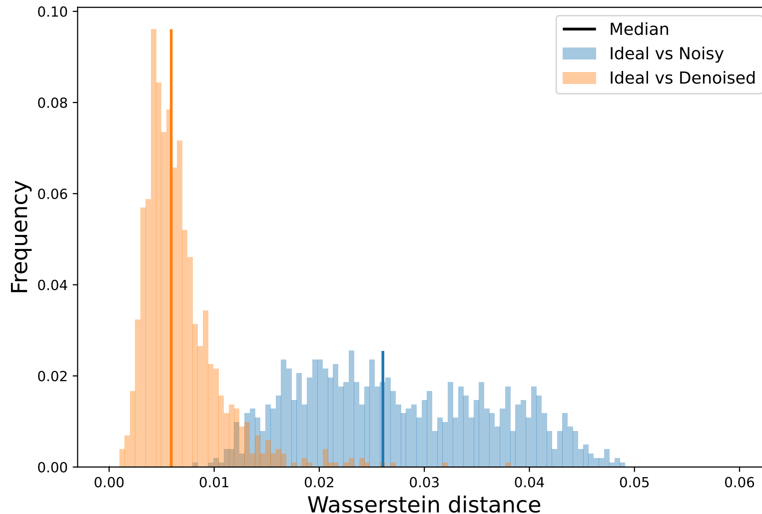


Figure 3.6: Comparison of the distributions of the Wasserstein distance between pairs of ideal-noisy histograms (in blue) and ideal-denoised histograms (in orange).

To further explain, a Cumulative Distribution Function (CDF) is a function that describes the probability that a random variable takes on a value less than or equal to a certain value. In our case, we have an empirical histogram instead of an analytical function. Therefore, the weight W is calculated by binning the data and summing over all the bins, rather than integrating over the entire range of possible values. This means that we approximate the integral by considering discrete intervals (bins) and summing the contributions from each bin instead of integrating in the x axis.

Therefore, $W_1(P, Q)$ measures the minimum average distance required to transport the distribution P to Q, where the average distance is computed based on the distances $d(x, y)$ and weighted by the transportation plan γ.

Figure 3.6 shows the distribution of this metric computed between all the pairs of ideal-noisy histograms and all the pairs of ideal-denoised histograms in the dataset.

In the case of a perfect matching in a pair of histograms, we would get a Wasserstein distance equal to 0. Even if this is never the case (our aim is just to reduce the error mitigating its impact but we can't have a complete removal as for QEC), the comparison between the two distributions shows the effectiveness of our mitigation method, since the peak related to the denoised results is significantly shifted towards 0 compared to the almost flat distribution of the original noisy data. In particular, the median value of the two distributions and their corresponding Confidence Interval (CI) for a 99% Confidence Level (CL) are the following:

$$\begin{aligned} \hat{W}_{\text{denoised}} &= 0.0056 \qquad \text{with } \text{CI}(\hat{W}_{\text{denoised}}) = [0.0053, 0.0060] \\ \hat{W}_{\text{noisy}} &= 0.0261 \qquad \text{with } \text{CI}(\hat{W}_{\text{noisy}}) = [0.0251, 0.0271] \end{aligned} \tag{3.5}$$

This result shows that our mitigation approach was able to reduce the median of the Wasserstein distance metric by almost an order of magnitude. The confidence interval on the median has been computed using the statistical bootstrapping technique [31].

3.4 Possible applications

Implementing an error mitigation approach that adapts to quasi-probability distributions, as opposed to relying solely on expected value error correction, represents a forward-looking strategy with potential for advancing quantum computing. In our implementation, we first trained our model with a fixed number of qubits, which allowed us to establish a basic framework.

It is important to note that scalability is a crucial factor in the mitigation of errors, as mentioned previously. In order for these error mitigation strategies to be applicable to larger quantum systems, which are becoming increasingly complex and computationally demanding, scalability is necessary. This allows the techniques to be extended beyond the limits of the original system, providing a robust and adaptable approach. The approach described above employs a fixed number of qubits, which precludes the autoencoder from being scaled. However, this does not imply that the model is not scalable, as the requisite step to increase the number of qubits is simply to select new scaled circuits. Furthermore, the data set must be recreated using code. The availability of an actual quantum computer would greatly simplify training by providing direct access to noisy quantum data, as opposed to relying on simulated noisy datasets, reducing the required amount of traditional resources.

The ability to train on real quantum noise data would also further improve the fidelity of our error mitigation strategies, as real quantum systems exhibit nuances and characteristics that are difficult to simulate accurately. Therefore, although our current implementation makes effective use of simulated noisy data, the potential integration of real quantum data represents a promising avenue for advancing

the accuracy and applicability of error mitigation techniques in quasi-probability distributions.

Important future applications, that derive from the quantum supremacy regime and the application of quantum computing in real world problems, can be numerous:

- Quantum algorithm: with quantum algorithms increasing in complexity and depth, they get more affected by noise and need to be validated to ensure their accuracy and helping in calibration and validation process;
- Quantum Machine Learning: QML process can improve its reliability with the help of distribution based error mitigation because of its probabilistic approach in various domains, such as pattern recognition, optimization, and classification;
- Quantum chemistry simulation: quantum chemistry simulation highly relies on precision since it is based on accuracy simulation of quantum scale objects.

An advantage of this implementation could finally be the need for only a train for a single quantum machine. We could then create a dataset for a new computer, train the autoencoder and then use it to clean all the data generated from the machine. This approach centered on the computer and not the data can be totally general instead of having to train a new model for each study made with the machine.

Conclusion

In this thesis, we introduce a new approach to error mitigation in quantum computing using machine learning techniques. Our focus is on reducing noise directly within the quasi-probability distribution, rather than solely correcting expected values. Quantum Error Mitigation (QEM) offers significant advantages in hardware efficiency compared to traditional Quantum Error Correction (QEC), and provides a variety of techniques suited to different application scenarios, making it integral to recent experimental demonstrations of quantum hardware.

Initially, our dataset consisted of real-world circuits involving five qubits designed for benchmarking and simulations. Subsequently, we generated circuits from ideal Gaussian distributions, which served as targets for our optimization algorithm applied to circuit parameters.

We implemented an autoencoder with 13 layers and used mean squared error (MSE) as the loss function, with the goal of transforming noisy distributions into distributions closer to their ideal counterparts. Training the autoencoder involved 4500 entries of noisy distributions paired with their corresponding ideal distributions, generated using a custom noise model defined in Qiskit.

Our results show promising outcomes, effectively reducing errors and producing distributions that closely approximate the ideal targets. The Wasserstein metric was used to confirm that the denoised data aligns well with the expected ideal distribution, demonstrating a clear improvement in quantum computing data quality.

These encouraging findings pave the way for the implementation of quantum error mitigation in more sophisticated experimental settings. The demonstrated reduction of errors and agreement with the ideal distributions may indicate that QEM techniques could prove instrumental in enhancing the precision and reliability of quantum computations. It may facilitate the advancement and maturation of quantum computing technology, enhancing its robustness and applicability in diverse practical and experimental scenarios. This progress is of critical importance for the continued development of quantum computing, as it addresses significant challenges in this field and brings the realisation of its potential closer.

Bibliography

[1] Mark M. Wilde. *Quantum Information Theory*. Cambridge University Press, Cambridge, UK, 2013.

[2] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[3] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC)*, pages 212–219. ACM, 1996.

[4] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, Cambridge, UK, 10th anniversary edition edition, 2010.

[5] David Deutsch. *Lectures on Quantum Computation*. University of New South Wales, Sydney, Australia, 1997.

[6] John Preskill. *Lecture Notes for Physics 219: Quantum Computation*. California Institute of Technology, 1998.

[7] Daniel Grier and Luke Schaeffer. The classification of clifford gates over qubits. *arXiv preprint arXiv:1603.03999*, 2016.

[8] Daniel Gottesman. The heisenberg representation of quantum computers. *arXiv preprint quant-ph/9807006*, 1998.

[9] Maarten Van den Nest. Classical simulation of quantum computation, the gottesman-knill theorem, and slightly beyond. *Quantum Information & Computation*, 10:258–271, 2010.

[10] A. Yu Kitaev. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys*, 52(6):1191–1249, 1997.

[11] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.

[12] Qiskit Development Team. Qiskit: An open-source quantum computing software development framework, 2024. Accessed: 2024-07-09.

[13] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[14] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, Inc., 2nd edition, 2019.

[15] Tom M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[16] Alexander Jung. Machine learning: The basics, 2018.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[18] Xiaojin (Jerry) Zhu. Semi-supervised learning literature survey. Technical Report TR1530, University of Wisconsin-Madison Department of Computer Sciences, 2005.

[19] Ahmet Iscen, Giorgos Tolias, Yannis Avrithis, and Ondrej Chum. Label propagation for deep semi-supervised learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5070–5079. IEEE, 2019.

[20] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[21] Wikipedia contributors. File:3d-gradient-cos.svg. `https://en.m.wikipedia.org/wiki/File:3d-gradient-cos.svg`, 2024. Accessed: 2024-06-25.

[22] B. Yegnanarayana. *Artificial Neural Networks.* Prentice-Hall of India, 1999.

[23] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11:625–660, 2010.

[24] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11:3371–3408, 2010.

[25] The TensorFlow Team. What's coming in tensorflow 2.0. `https://blog.tensorflow.org/2019/01/whats-coming-in-tensorflow-2-0.html`, 2019. Accessed: 2024-06-25.

[26] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, and Greg S. Corrado et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[27] Nils Quetschlich, Lukas Burgholzer, and Robert Wille. MQT Bench: Benchmarking software and design automation tools for quantum computing. *Quantum*, 2023. MQT Bench is available at `https://www.cda.cit.tum.de/mqtbench/`.

[28] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Carsten Blank, Keri McKiernan, and Nathan Killoran. Pennylane: Automatic differentiation of hybrid quantum-classical computations, 2018. Version 0.15.0.

[29] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[30] L. V. Kantorovich. *Mathematical Methods of Organizing and Planning Production.* Management Science Press, Chicago, 1960.

[31] SciPy. scipy.stats.bootstrap. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.bootstrap.html`, n.d. Accessed: July 11, 2024.

Appendix A

Noisy circuit simulation in Qiskit

The following code sets up a custom noise model for quantum simulations using the Qiskit framework. It defines various noise parameters including T1 and T2 times, gate operation times for single and two-qubit gates, and the probability of depolarizing errors. The depolarizing error probability is set to a very low value, indicating infrequent errors. Readout error probabilities are specified in a matrix, detailing the probabilities of incorrect measurement outcomes. The code then configures a simulator backend to mimic the properties of the IBMQ Sherbrooke quantum computer. The coupling map, defining the connectivity between qubits in the backend, is also retrieved to ensure the noise model accurately reflects the qubit interactions. This setup allows for realistic simulation of quantum circuits under specified noise conditions.

```
from qiskit.synthesis import generate_basic_approximations
from qiskit_aer import AerSimulator
from qiskit_ibm_runtime.fake_provider import FakeSherbrooke
from qiskit_aer.noise import NoiseModel, depolarizing_error,
    thermal_relaxation_error, ReadoutError
```

```
# Noise model parameters
t1 = 200e4  # T1 relaxation time in nanoseconds
t2 = 150e4  # T2 relaxation time in nanoseconds
gate_time_single = 75  # Single-qubit gate time in nanoseconds
gate_time_two = 100  # Two-qubit gate time in nanoseconds

# Depolarizing error probability
depolarizing_prob = 0.000001

# Readout error probabilities
readout_error_prob = [[0.98, 0.02], [0.02, 0.98]]

# Create a noise model
noise_model = NoiseModel()

# Define the gates for which to add errors
basis_gates = ["x", "y", "z", "s", "sdg", "h", "t", "tdg"]

```

```
19 # Add depolarizing noise to single qubit gates
20 error_1 = depolarizing_error(depolarizing_prob, 1)
21 noise_model.add_all_qubit_quantum_error(error_1, basis_gates)
22
23 # Add thermal relaxation noise to single qubit gates
24 thermal_error_single = thermal_relaxation_error(t1, t2,
       gate_time_single)
25 noise_model.add_all_qubit_quantum_error(thermal_error_single,
       basis_gates)
26
27 # Add depolarizing noise to two qubit gates
28 error_2 = depolarizing_error(depolarizing_prob, 2)
29 noise_model.add_all_qubit_quantum_error(error_2, ['cx'])
30
31 # Add thermal relaxation noise to two qubit gates
32 thermal_error_two = thermal_relaxation_error(t1, t2, gate_time_two
       ).tensor(thermal_relaxation_error(t1, t2, gate_time_two))
33 noise_model.add_all_qubit_quantum_error(thermal_error_two, ['cx'])
34
35 # Add readout error
36 readout_error = ReadoutError(readout_error_prob)
37 noise_model.add_all_qubit_readout_error(readout_error)
38
39 # Set up the backend
40 backend = AerSimulator.from_backend(FakeSherbrooke())
41 coupling_map = backend.configuration().coupling_map
```

Listing A.1: Setting custom noise model parameters and running noisy quantum circuit simulation using the Qiskit *AerSimulator*.

Appendix B

Circuit optimization in Pennylane

The `pqc_workflow` function implements the main workflow for the optimization of a parameterized quantum circuit using PennyLane software. The circuit is converted from Qiskit to PennyLane and decorated using `qml.qnode`. The optimization process involves calculating gradients using finite differences and updating the parameters through iterative steps.

The workflow initializes the parameters randomly and runs the PQC optimization for a specified number of training iterations.

```
1 import pennylane as qml
2 import jax, optax, catalyst
```

```
1 # Configuration settings for the simulation
2 opt_iters = 200
3 num_qubits = 5
4 dev = qml.device("lightning.qubit", wires=num_qubits)
5 # Main workflow for parameterized quantum circuit
6 def pqc_workflow(qiskit_pqc, check_training_convergence=True):
7     pennylane_pqc = qml.from_qiskit(qiskit_pqc)
8     opt = optax.adam(learning_rate=0.2)
9 
10     @qml.qjit
11     @qml.qnode(device=dev)
12     def circuit(params, psi):
13         pennylane_pqc(params)
14         qml.adjoint(qml.AmplitudeEmbedding(psi, wires=range(
              num_qubits)))
15         return qml.probs()
16 
17     @qml.qjit
18     def objective_function(params, psi):
19         probs = circuit(params, psi)
20         return jax.numpy.sqrt(1 - probs[0]**2)
21 
22     @qml.qjit
23     def update_step(_, args):
24         params, opt_state, psi = args
```

```
25          grads = catalyst.grad(objective_function, method="fd")(
                params, psi)
26          updates, opt_state = opt.update(grads, opt_state)
27          params = optax.apply_updates(params, updates)
28          return params, opt_state, psi
29
30      @qml.qjit
31      def run_optimization(params, psi, num_iters):
32          opt_state = opt.init(params)
33          args = params, opt_state, psi
34          params, opt_state, _ = qml.for_loop(0, num_iters, 1)(
                update_step)(args)
35          return params
36
37      params = jax.numpy.array(np.random.rand(qiskit_pqc.
            num_parameters))
38      psi = jax.numpy.array(generate_random_psi())
39      run_optimization(params, psi, 1)
```

Listing B.1: Definition of the quantum circuits optimization workflow in Pennylane.

Appendix C

Denoising Autoencoder in Keras

This section defines a set of custom loss functions and implements a denoising autoencoder model using TensorFlow. The custom loss functions include Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Kullback-Leibler Divergence (KL), and Cross-Entropy (CE). These functions are stored in a dictionary, allowing them to be easily referenced by name. The denoising autoencoder model consists of an encoder and a decoder, both composed of dense layers with LeakyReLU activations. The encoder reduces the input dimensionality to a latent space, while the decoder reconstructs the input from this latent representation. The final layer of the autoencoder uses a softmax activation function to output the reconstructed data. The model is compiled with a specified optimizer and a custom loss function selected from the dictionary based on the provided loss name. This structure allows for flexible and efficient denoising of input data.

```
1 import tensorflow as tf
2 from tensorflow.keras.layers import Input, Dense, LeakyReLU
3 from tensorflow.keras.models import Model
4 from tensorflow.keras.optimizers import Adam
```

```
1 # Setting the optimizer and loss function name
2 optimizer = Adam(learning_rate=5e-4, amsgrad=True)
3 def mse_loss(y_true, y_pred):
4     return tf.reduce_mean(tf.square(y_true - y_pred))
5 
6 # Defining the autoencoder model for denoising
7 def denoising_autoencoder(in_out_shape, latent_units, optimizer):
8     # Encoder layers
9     inputs = Input(in_out_shape)
10    x = Dense(units=128, activation=LeakyReLU(alpha=0.2))(inputs)
11    x = Dense(units=256, activation=LeakyReLU(alpha=0.2))(x)
12    x = Dense(units=64, activation=LeakyReLU(alpha=0.2))(x)
13    x = Dense(units=32, activation=LeakyReLU(alpha=0.2))(x)
14    x = Dense(units=16, activation=LeakyReLU(alpha=0.2))(x)
15    x_latent = Dense(latent_units, activation=LeakyReLU(alpha=0.2)
          )(x)
16 
```

```
17      # Decoder layers
18      x = Dense(units=16, activation=LeakyReLU(alpha=0.2))(x_latent)
19      x = Dense(units=32, activation=LeakyReLU(alpha=0.2))(x)
20      x = Dense(units=64, activation=LeakyReLU(alpha=0.2))(x)
21      x = Dense(units=256, activation=LeakyReLU(alpha=0.2))(x)
22      x = Dense(units=128, activation=LeakyReLU(alpha=0.2))(x)
23      outputs = Dense(in_out_shape[0], activation="softmax")(x)
24
25      autoencoder = Model(inputs=inputs, outputs=outputs)
26      autoencoder.compile(optimizer=optimizer, loss=mse_loss)
27      return autoencoder
28
29  # Creating the autoencoder for denoising
30  denoiser = denoising_autoencoder(in_out_shape=(32,), latent_units
        =8, optimizer=optimizer)
```

Listing C.1: Definition of the autoencoder architecture and compilation of the Keras model using the ADAM optimizer and the MSE loss function.