POLITECNICO DI TORINO

Master degree course in Data Science and Engineering

Master Degree Thesis

Improving Financial Fraud Detection with Quantum Reinforcement Learning

Politecnico di Torino

Supervisors
Prof. Giovanna Turvani
PhD Deborah Volpe
Prof. Maurizio Zamboni

Candidate
Emanuela Piga

December 2025

Contents

Abstract

The growing use of online banking and digital payment systems has increased the risk of fraudulent activities. For this reason, identifying suspicious transactions is an essential task that should continuously evolve with technological advances and new forms of criminal activity. In recent years, fraud detection has been commonly implemented by exploiting Machine Learning (ML) models, which are able to detect irregular patterns and identify fraudulent transactions. However, classical ML performance is not always satisfactory, especially in strongly unbalanced classification tasks like fraud identification. To overcome these limits, Quantum Machine Learning (QML) has recently gained researchers' interest. Indeed, these systems, working intrinsically in a higher-dimensional computation space thanks to superposition and entanglement, can implement more complex models than the classical counterparts, potentially providing a significant advantage in complex pattern recognition. Among the various QML paradigms, Quantum Reinforcement Learning (QRL) is an emerging field that combines the ability of classical Reinforcement Learning (RL) to solve sequential decision-making problems with principles of Quantum Computing (QC) to improve speed and efficiency. This thesis specifically focuses on the application of QRL to a fraud detection task based on bank transaction data. The aim of this work is to provide a performance comparison between classical and quantum-based reinforcement learning. The study first shows how RL achieves higher classification performance than traditional supervised learning methods, thanks to its sequential decision-making nature and the use of a reward function. Additionally, by implementing two distinct algorithms, Q-learning and REINFORCE, this work compares two different approaches to RL. Then, the thesis studies how QC principles can further improve these capabilities. For each RL algorithm, a quantum version was created by substituting the classical neural network with its quantum counterpart implemented through a Variational Quantum Circuit (VQC). The reinforcement learning environment was implemented using Gymnasium library, which allows to create the agent–environment interaction loop and the reward function. Quantum components were implemented using PennyLane library, which enables integrating VQCs within the PyTorch models to create hybrid quantum–classical architectures. Experimental results show that QRL achieves comparable or better

performance than classical RL approaches with a limited number of trainable parameters (84 in Q-learning and 150 in REINFORCE, instead of the 5938 required by both classical architectures), with F1 score improvements of 0.39% in Q-learning and 36.7 % in REINFORCE. Due to the limitations of employing ideal quantum simulators, which are both computationally expensive and time-consuming, it has not been feasible to perform large-scale experiments. Future work should focus on optimizing training efficiency and exploring more scalable quantum-classical integration strategies.

Part I

Background

Chapter 1

Introduction and Motivation

In the current age of online banking and e-commerce, fraudulent activities have grown exponentially as fraudsters exploit the weaknesses of these systems. Hacking, phishing, money laundering, and other illegal activities have led to the spread of many different types of fraud. The most common is payment fraud, which refers to any unauthorized transaction executed after payment information is stolen, such as a credit card number and other security codes. Related to payment fraud is account takeovers, which involve the illegal appropriation of personal information, often through hacking. Furthermore, the use of money laundering to disguise illegally obtained funds as legitimate makes it harder to detect unauthorized activities. Due to the risks associated with fraud, financial institutions and any platform that provides digital payment options must adapt their security systems to protect customers from losing money and personal data. Traditional fraud detection systems were rule-based, relying on predefined thresholds and personal experience to identify suspicious transactions. Not only were these methods inaccurate and inefficient, but they also could not keep pace with evolving trends in data and technology. Nowadays, fraud detection is almost real-time, enabling immediate action to prevent or reduce the impact of illegal activities. This is possible thanks to Artificial (AI) Intelligence and Machine Learning (ML) systems. These models can be trained to recognize fraudulent transactions by analyzing large volumes of financial data. By learning patterns and other insights that may indicate a transaction's fraudulent nature, these systems can decide whether a transaction they have never seen before is fraudulent or not [1].

In high-stakes scenarios involving money and personal information, training models to achieve high classification performance poses two fundamental problems. The first concerns the nature of data, since large, high-quality datasets are a necessary prerequisite for any well-functioning ML system. The second problem relates to the computational resources required to train these systems extensively. In fact, with advancements in Deep Learning and the frequent use of Neural Networks (NN), models employed contain thousands of parameters, which contribute to inefficiency

and long training times.
In the context of the employed dataset, almost all fraud detection datasets pose a significant challenge for training ML models, since fraudulent transactions typically constitute less than 1% of the total number of transactions in the dataset [2]. While this reflects the reality of modern banking systems, the highly imbalanced nature of these datasets often leads to poor generalization and biases towards the majority class (non-fraud). For this reason, sampling techniques such as SMOTE are often employed to either under-sample or over-sample the majority class to ensure the correctness of the results [3]. However, this solution also presents some problems, as using synthetic data can lead to poor model performance and limited generalization.
These limitations have increased research in more adaptive learning frameworks, such as Reinforcement Learning (RL). In the context of fraud detection, RL was proven to offer a significant advantage over static models, thanks to its inherent ability to adapt to changes as soon as new information becomes available [4, 5]. RL is the third learning paradigm in the field of ML models. The first one, Supervised Learning, consists of training a model to make classifications or predictions based on the information learned from a series of labeled samples. The second paradigm, unsupervised learning, involves finding hidden patterns and relationships in data without any guidance. Both paradigms share the fact that the model learns only by analyzing a fixed set of samples. In contrast, in RL, the model, also called *agent*, learns from feedback received while interacting with an external system, called *environment*. At each step, the agent, based on the information received from the environment, takes an action (for instance, approving a transaction). The environment responds by providing the agent new information from the dataset and a reward, which is a score that will be used by the agent to understand whether the previous action was correct or not. Modeling the learning task sequentially allows the agent to receive immediate feedback after each decision; this enables the models to quickly adapt to changing patterns in the dataset. Moreover, the use of a reward function allows the model to learn about the dataset's imbalance by providing higher scores when a rare sample is correctly classified (a fraudulent transaction in this case).
However, there is a significant computational burden associated with updating the model's parameters every time a new pattern appears. This particularly affects RL models employing NN, since they have large numbers of parameters, and updating all of them dramatically increases training time. This could affect the deployment of these systems in resource-constrained environments such as edge devices.
In general, one of the main limiting factors in training ML systems is the hardware on which these algorithms run. The rise of graphical processing units (GPUs), compared to traditional central processing units (CPUs), enabled the current AI revolution by providing the computational power to train NNs [6].
Despite these advances, conventional architectures are approaching the limits of

Moore's principle [7]. Moore's Law states that the number of transistors on a chip doubles approximately every two years, leading to continuous improvements in computational power and reductions in cost and device size. However, as transistor dimensions approach the atomic scale, further reductions become physically challenging due to quantum effects, heat dissipation, and manufacturing limitations. Therefore, the exponential growth in transistor count predicted by Moore's Law is slowing [8]. On the other hand, demand for computational power is growing, as AI models become increasingly complex and require greater accuracy.
One possible option to overcome this issue is quantum computing, which leverages quantum-mechanical properties such as superposition, entanglement, and interference to solve computational tasks more efficiently than traditional computing devices. One of the first practical advantages of quantum computers was introduced in 1994 by mathematician Peter Shor [9], who demonstrated that an ideal quantum computer could perform integer factorization in polynomial time, rather than the super-polynomial time required by a classical computer. However, constructing an ideal quantum computer remains a significant challenge. Current quantum technology relies on Noisy Intermediate-Scale Quantum (NISQ) devices, which consist of tens to hundreds of noisy qubits. While the operations performed on these devices are imperfect, NISQ computers still offer the potential to solve challenging problems by exploiting quantum effects as superposition and entanglement [10]. Moreover, modern quantum simulators allow researchers to design, test, and analyze quantum algorithms with increasing realism. Therefore, research on quantum computing is progressing rapidly and is considered promising for future applications.
A field that has attracted particular interest is Quantum Machine Learning (QML), which explores how the computational power of quantum machines can be applied to traditional ML to outperform classical algorithms. QML has the potential to revolutionize AI, data analytics, and the efficiency of models by exploiting techniques such as superposition and entanglement. A notable example is Quantum Support Vector Machines, which uses quantum kernels to create more complex and effective classification boundaries than classical Support Vector Machines. Another promising example is Quantum Neural Networks (QNN), which exploit quantum parallelism to process multiple inputs simultaneously, thereby reducing training and inference times [11]. The leading technology in QML is Variational Quantum Circuits (VQCs). Their main distinguishing feature is the presence of trainable parameters that can be optimized using classical methods. VQCs operate in a hybrid quantum–classical framework: the quantum component processes input features exploiting quantum effects, while the classical component handles parameter optimization using techniques such as gradient-based methods. This hybrid approach allows VQCs to be implemented with relatively shallow circuits, making them more resilient to noise despite the limitations of NISQ hardware [12].
Among QML subfields, Quantum Reinforcement Learning (QRL) combines the adaptive learning strategies of RL with the computational advantages of QC [13].

In particular, this thesis investigates the application of QRL to a fraud detection task. The goal is to leverage the adaptive properties of RL to overcome challenges such as class imbalance and to respond quickly to evolving fraud patterns, while simultaneously exploiting quantum computational features to enhance model expressivity, efficiency, and learning performance.
The contribution of this work consists in the implementation of two QRL architectures for fraud detection: quantum Q-learning and quantum REINFORCE. The quantum version for both architectures was created by replacing the NN inside the agent with a VQC. The environment's reward system was adapted to the fraud detection task and the employed dataset in order to give higher rewards to rarer transactions. Furthermore, in this work, a study was conducted to determine the optimal number of transactions per episode, balancing performance and training time. A systematic hyperparameter optimization was performed for both classical and quantum models using OPTUNA. Extending its usual application from classic ML hyperparameters to quantum-specific ones (such as encoding type and ansatz), in the context of fraud detection.
Moreover, the thesis focuses on an extensive performance comparison between QRL architectures and their classical counterparts, using both classical evaluation metrics (accuracy, precision, recall, f1, reward) and combined statistics (average values, combined moving averages, and cumulative probability distribution). Finally, the Fisher Information Matrix was employed to compare the two implemented architectures' capabilities to adapt to parameter updates.
From the experimental results, it emerged that the combination of RL with Quantum Computing provides an advantage over the classical architectures. In particular, REINFORCE demonstrated that, with a sufficiently large number of episodes and with relatively low training time, it can achieve high performance in terms of F1-score and reward. The Q-learning algorithm demonstrated high responsiveness to parameter updates and the ability to adapt its decision-making mechanism with few episodes.

Chapter 2

Introduction to Quantum Computing

Quantum computing leverages principles of quantum mechanics and quantum physics to solve problems much faster than classical computers. Two of the most famous quantum algorithms, which have demonstrated the potential of quantum computing to revolutionize various fields, are Shor's and Grover's algorithms [11].
Shor's algorithm [9] demonstrated that an ideal quantum computer could perform integer factorization in polynomial time, rather than in super-polynomial time, as required by a classical computer. Therefore, if a large-scale quantum computer were available, it could break most cryptographic systems, posing significant security risks.
Grover's algorithm [14] has the ability to search for an item in an unsorted database four times faster than a classical algorithm. This could be useful in data analytics and optimization tasks.
In the following chapter, the fundamental principles that allow the realization of these algorithms are illustrated.

2.1 Qubits

The minimal unit of information in classical computers is called bit, and it can only assume state 0 and state 1. Its equivalent in the quantum world is is called *quantum bit* or *qubit* for short. The state of a qubit is represented with the symbol '$|\rangle$', known as "Dirac notation". For example the quantum equivalent of states 0 and 1 are states $|0\rangle$ and $|1\rangle$.

2.1.1 Single qubit and superposition

Qubits can be represented in the column vector form as shown in equation 2.1.1

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{2.1.1}$$

The difference between bits and qubits is that qubits can also be in other states. For example, it is possible to form linear combinations of states called *superpositions* [15]:

$$|\psi\rangle = c_0 |0\rangle + c_1 |1\rangle = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} \tag{2.1.2}$$

where c_0, c_1 are two complex numbers, called *amplitudes*, such that $|c_0|^2 + |c_1|^2 = 1$. The quantities $|c_0|^2|$ and $c_1|^2$ are interpreted as the probabilities of finding the qubit respectively in state $|0\rangle$ and in state $|1\rangle$, after it is measured.
In general, the concept of **superposition** refers to the ability of quantum systems to exist in multiple states simultaneously. This ability allows quantum computers to process multiple data points simultaneously, potentially speeding up computation [11].

2.1.2 Multiple qubits

Qubits can be merged together, in order to form multiple combinations. For example, qubits in states $|0\rangle$ and $|1\rangle$ can lead to four possibilities, known as *computational basis*:

$$|0\rangle \otimes |0\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \qquad |0\rangle \otimes |1\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \qquad |1\rangle \otimes |0\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \qquad |1\rangle \otimes |1\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

(2.1.3)

A qubit combination follows the tensor product rule, denoted by $\otimes$. A quantum state is said to be a *product state* if it can be written as the tensor product of two other states, both corresponding to the state of one or more qubits. The symbol of the tensor product can be omitted, and the general expression for a two-qubit system can be written as follows:

$$|\psi\rangle = a_{00} |00\rangle + a_{01} |01\rangle + a_{10} |10\rangle + a_{11} |11\rangle \tag{2.1.4}$$

where $a_{00}, a_{01}, a_{10}, a_{11}$ are complex numbers such that $\sum_{x,y=0}^{1} |a_{x,y}|^2 = 1$.
The two-qubit case can be generalized to a $N-$qubit system with $N > 2$:

$$|\psi\rangle = \sum_{i=1}^{2^N} c_i |i\rangle \tag{2.1.5}$$

where ψ is a 2^N-dimensional vector, $i \in 0{,}1$, and the coefficients satisfy $\sum_{i=1}^{2^N} |c_i|^2 = 1$ [6].

2.2 Entanglement

A quantum state is said to be **entangled**, it cannot be written as a tensor product of two other states. Entanglement is a principle in which two elements become so deeply connected that measuring the state of one qubit determines the outcome of measuring the other.
Consider the sate $|\psi\rangle$:

$$|\psi\rangle = |00\rangle + |11\rangle \tag{2.2.1}$$

This can be expanded, considering all the elements of the computational basis, by rewriting it in the following way:

$$|\psi\rangle = |00\rangle + |11\rangle = 1\,|00\rangle + 1\,|11\rangle + 0\,|01\rangle + 0\,|10\rangle \tag{2.2.2}$$

In order to determine whether $|\psi\rangle$ comes from the product of two tensors, it can be assumed that this product has the following form:

$$|\psi\rangle = (a_0\,|0\rangle + a_1\,|1\rangle) \otimes (b_0\,|0\rangle + b_1\,|1\rangle) \quad a_0, a_1, b_0, b_1 \in \mathbb{C} \tag{2.2.3}$$

This equation can be expanded as follows:

$$|\psi\rangle = a_0 b_0\,|00\rangle + a_1 b_1\,|11\rangle + a_0 b_1\,|01\rangle + a_1 b_0\,|10\rangle \tag{2.2.4}$$

The coefficients from the tensor product expansion $(a_0b_0, a_0b_1, a_1b_0, a_1b_1)$ should match the coefficients of the original equation 2.2.2. In particular, it should hold that $a_0b_1 = a_1b_0 = 0$ and also that $a_0b_0 = a_1b_1 = 1$. However, these two affirmations contradict each other; it follows that $|\psi\rangle$ can not be written as a tensor product.

2.3 Bloch Sphere

All the possible states of a qubit can be represented as points on the surface of a sphere [16]. Consider a qubit in state $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$, the amplitudes a and b, can be written a and b using polar coordinates as:

$$a = r_1 e^{i\alpha_1}, \quad b = r_2 e^{i\alpha_2}$$

where $\alpha_1, \alpha_2 \in [0, 2\pi], r_1 = |a|, r_2 = |b|$ and $r_1^2 + r_2^2 = 1$.
Given the following equivalence:

$$r_1 = cos(\theta/2) \quad \text{and} \quad r_2 = sin(\theta/2) \tag{2.3.1}$$

since it holds that $r_1^2 + r_2^2 = 1$, there must be an angle θ that satisfies the Pythagorean trigonometric identity:

$$cos^2(\theta/2) + sin^2(\theta/2) = 1 \tag{2.3.2}$$

In this way the the state vector $|\psi\rangle$ can be rewritten as follows:

$$|\psi\rangle = cos(\theta/2)e^{i\alpha_1} |0\rangle + sin(\theta/2)e^{i\alpha_2} |1\rangle \tag{2.3.3}$$

The vector $|\psi\rangle$, can be multiplied by $e^{-i\alpha_1}$, with $\varphi = \alpha_2 - \alpha_1$, in order to obtain an equivalent representation:

$$e^{-i\alpha_1} |\psi\rangle = cos(\theta/2)e^{i(\alpha_1 - \alpha_1)} |0\rangle + sin(\theta/2)e^{i(\alpha_2 - \alpha 1)} |1\rangle \tag{2.3.4}$$

by setting $\varphi = \alpha_2 - \alpha_1$, the equation becomes:

$$|\psi\rangle = cos(\theta/2) |0\rangle + sin(\theta/2)e^{i\varphi} |1\rangle \tag{2.3.5}$$

When all the elements of a quantum state are multiplied by complex factor $e^{i\alpha}$, is called *global phase*. It does not affect any physical measurement because only relative phases between components of a superposition can produce observable effects. Therefore, multiplying $|\psi\rangle$ by a global phase leaves the physical state unchanged. Using $\theta/2$ for the qubit representation ensures that, when the angles range from 0 to π, the cosine and sine correctly cover the full range of amplitudes.
The state of any qubit can be described with just two numbers: $\theta \in [0, \pi], \varphi \in [0, 2\pi]$. This gives a three-dimensional point with the following spherical coordinates:

$$(sin(\theta)cos(\varphi), sin(\theta)sin(\varphi), cos(\theta))$$

These coordinates locate the state of the qubit on the surface of a sphere, known as *Bloch sphere*, shown in figure 2.1

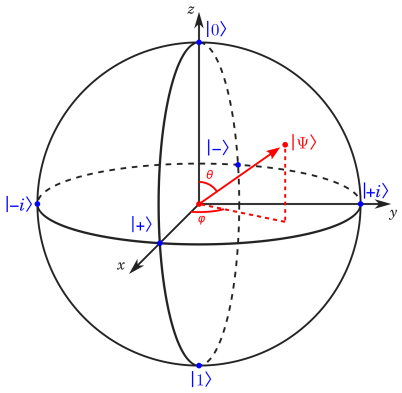

Figure 2.1: Qubit states on the bloch sphere

2.4 Quantum gates

A qubit can be considered as a quantum system whose evolution (from initial state to final state) is governed by the Schrödinger equation [17]:

$$i\hbar\frac{\partial}{\partial t}\left|\psi(t)\right\rangle = H\left|\psi(t)\right\rangle \tag{2.4.1}$$

where i is the imaginary unit, $\hbar$ is the reduced Planck constant, $|\psi(t)\rangle$ is the state vector of the system at time t, and H is the Hamiltonian of the system (representing its total energy). If the Hamiltonian operator is known, the Schrödinger equation can be solved easily, given the initial state $|\psi(0)\rangle$ and the following linear mapping [18]:

$$\left|\psi(t)\right\rangle = U(t)\left|\psi(0)\right\rangle \tag{2.4.2}$$

U(t) is a **unitary matrix**, which is a type of squared matrix such that:

$$U^{\dagger}U = UU^{\dagger} = I \tag{2.4.3}$$

where I is the identity matrix and $U^{\dagger}$ is the adjoint matrix, obtained by transposing U and replacing each element by its complex conjugate. In the context of quantum computers, the operations represented by these matrices are called quantum gates. This section present the most important quantum gates.

2.4.1 NOT gate

The NOT gate transforms a qubit in state $|0\rangle$ into a qubit in state $|1\rangle$ (or vice versa). Since a single qubit has dimension 2, the NOT gate is represented by the following 2x2 matrix:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{2.4.4}$$

The NOT gate functioning is illustrated below:

$$X\left|0\right\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \left|1\right\rangle \qquad X\left|1\right\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \left|0\right\rangle \tag{2.4.5}$$

2.4.2 Hadamard gate

This gate is commonly used to create superposition states; its matrix is given in equation 2.4.6.

$$H = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{2.4.6}$$

When applying the H gate on a qubit in state $|0\rangle$, the **plus** state $|+\rangle$ is obtained. In a similar way, the **minus** state $|-\rangle$ is obtained, if the H gate is applied on a qubit in state $|1\rangle$:

$$H|0\rangle = |+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \qquad H|1\rangle = |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{2.4.7}$$

Several gates can be applied to the same qubit in sequence. For example, in the circuit shown in 2.2, the H gate is first applied, then an X gate, and finally another H gate.

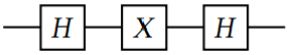

Figure 2.2: One bit quantum circuit

Suppose that the initial state of the qubit is $|1\rangle$. Applying the H gate means performing the second operation shown in equation 2.4.7, which leads to state minus $|-\rangle$. When the NOT gate is applied, the result becomes $-|-\rangle$. Finally, after using the second H gate, the final result is $-|1\rangle$ as shown in equation 2.4.8

$$H(-|-1\rangle) = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\frac{1}{\sqrt{2}}\begin{pmatrix}-1\\ 1\end{pmatrix} = \frac{1}{2}\begin{pmatrix}-1+1\\ -1-1\end{pmatrix} = \begin{pmatrix}0\\ -1\end{pmatrix} = -|1\rangle \tag{2.4.8}$$

2.4.3 Pauli matrices and Rotation Gates

Other important quantum gates are the Y-gate and the Z-gate, which, along with the NOT gate, are called the **Pauli** matrices 2.4.3:

$$X = \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} \quad Y = \begin{pmatrix}0 & -i\\ i & 0\end{pmatrix} \quad Z = \begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$$

Considering the Bloch sphere in picture 2.1, it can be noticed that the X gate acts like a rotation of π around the X axis of the Bloch sphere. Similarly, Z and Y gates are rotations of π radians around their respective axis. Therefore, for X, Y, and Z, the following **rotation gates** can be defined:

$$R_X(\theta) = e^{-i\frac{\theta}{2}X} = \begin{pmatrix}cos(\theta/2) & -isin(\theta/2)\\ -isin(\theta/2) & cos(\theta/2)\end{pmatrix} \tag{2.4.9}$$

$$R_Y(\theta) = e^{-i\frac{\theta}{2}Y} = \begin{pmatrix}cos(\theta/2) & -sin(\theta/2)\\ isin(\theta/2) & cos(\theta/2)\end{pmatrix} \tag{2.4.10}$$

$$R_Z(\theta) = e^{-i\frac{\theta}{2}Z} = \begin{pmatrix}e^{-i\frac{\theta}{2}} & 0\\ 0 & e^{-i\frac{\theta}{2}}\end{pmatrix} \tag{2.4.11}$$

2.5 Multiple-qubit gates

Two-qubit quantum gates are 4x4 unitary matrices that act on 4-dimensional column vectors. Given two one-qubit gates U_1 and U_2 and two one-qubit states $|\psi_1\rangle, |\psi_2\rangle$, a two-qubit gate $U_1 \otimes U_2$ can be constructed in the following way:

$$(U_1 \otimes U_2)(|\psi_1\rangle \otimes |\psi_2\rangle) = (U_1 |\psi_1\rangle) \otimes (U_2 |\psi_2\rangle) \tag{2.5.1}$$

The tensor product operation can also be applied between gates. For example, in the two-qubit circuit of figure 2.3, the gate $X \otimes X$ acts on the two qubits and is followed by the operation $H \otimes I$, where I is the identity matrix.

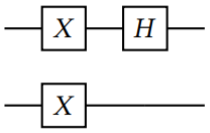

Figure 2.3: Two-qubit circuit

There are many gates that cannot be written as tensor products of other matrices, for example, the **CNOT (controlled-NOT) gate**,whose matrix is shown in 2.5.2, flips the value of the second qubit if and only if the value of the first is 1

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.5.2}$$

Applying the CNOT on a two-qubit computational basis leads to the following results:

$$\text{CNOT}|00\rangle = |00\rangle,\ \text{CNOT}|01\rangle = |01\rangle,\ \text{CNOT}|10\rangle = |11\rangle,\ \text{CNOT}|11\rangle = |10\rangle,$$

Figure 2.4 represents the CNOT circuit symbol, where the control qubit is in the bottom line and the target qubit is the top one.

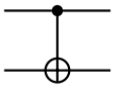

Figure 2.4: CNOT circuit symbol

Another gate is the SWAP gate, whose role is to exchange the states of two qubits, as reported below:

$$SWAP\,|\psi\rangle \otimes |\phi\rangle = |\phi\rangle \otimes |\psi\rangle \tag{2.5.3}$$

The SWAP gate can be implemented using a sequence of three CNOTs with the target and control qubits alternating, as it can be seen in picture 2.5.

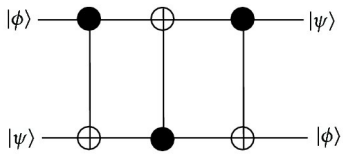

Figure 2.5: SWAP circuital symbol

There are also **controlled-Y**, **controlled-Z**, **controlled-H** gates, that work in a similar way to the CNOT gate. For any quantum gate U, its controlled version, *controlled-U*, can be defined. Its action on the computational basis is the following:

$$\mathrm{CU}|00\rangle = |00\rangle,\ \mathrm{CU}|01\rangle = |01\rangle,\ \mathrm{CU}|10\rangle = |1\rangle\, U\, |0\rangle,\ \mathrm{CU}|11\rangle = |1\rangle\, U\, |1\rangle,$$

Figure 2.6 displays the symbol for a controlled-U gate.

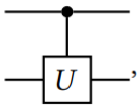

Figure 2.6: Controlled quantum gate symbol

Extending the concept of two-qubit gates, to n-qubit gates, these can also be realized by performing the tensor product between other gates.
However, this is not true for any gate, for example, the **Toffoli** or **CCNOT** gate is a three-qubit gate that applies a doubly controlled NOT gate to the third qubit of the computational basis, it acts as follows:

$$CCNOT\, |x\rangle\, |y\rangle\, |z\rangle = |x\rangle\, |y\rangle\, |z \oplus (x \wedge y)\rangle \tag{2.5.4}$$

where $\oplus$ and $\wedge$ are respectively the XOR and AND boolean functions. The Toffoli gate is important because, with the help of auxiliary qubits, it can be used to construct any classical Boolean operator.

2.6 Measurements

Measurement operations are performed at the end of a quantum computation; they are necessary to extract classical information from the final quantum state of the implemented task.
According to Born's rule, given a state $|\psi\rangle = \sum_i c_i\, |i\rangle$, the probability that when measured in the computational basis it reduces to state $|\psi\rangle$, is given by:

$$p_k = |\langle k|\psi\rangle|^2 = |c_k|^2 \tag{2.6.1}$$

Most algorithms use the default computational basis $|0\rangle, |1\rangle$. In this case the probabilities would be:

- probability of observing $|0\rangle$: $p_0 = |\langle 0|\psi\rangle|^2 = |c_0|^2$
- probability of observing $|1\rangle$: $p_1 = |\langle 1|\psi\rangle|^2 = |c_1|^2$

others are admissible such as the Hadamard basis $|+\rangle, |-\rangle$.
Beyond producing probabilistic outcomes, quantum measurements allow to infer physical properties of the quantum system. These properties are called *observables*, and mathematically they are represented by Hermitian operators O, which are linear operators for which it holds $O\dagger = O$. The expected value of an observable O, on a quantum system described by $|\psi\rangle$ is:

$$\langle O\rangle = \langle\psi|O|\psi\rangle \tag{2.6.2}$$

In real experiments, the estimation of the expected value of O is a multi-step process that consists of repeatedly measuring the quantum state and combining the measurements with classical post-processing [12].
Another common measure, is quantum state *fidelity*,which for two quantum states $|\psi\rangle$ and $|\phi\rangle$, measures their closeness:

$$F = |\langle\psi|\phi\rangle|^2 \tag{2.6.3}$$

where $|\psi\rangle$ is the actual state and $|\phi\rangle$ is the target one [19]. It is used to evaluate the effectiveness of quantum gates and circuits. As the fidelity values approach zero, it shows that the quantum computer is more affected by hardware noise and errors, and therefore its results are not reliable.

Chapter 3

Reinforcement Learning

Machine Learning (ML) is a subfield of Artificial Intelligence (AI) that studies how computer algorithms can improve automatically through experience [20]. According to how the algorithm learns new information, three ML paradigms can be distinguished:

- **Supervised Learning**: in which the model learns from labeled data.
- **Unsupervised Learning**: where the model recognizes patterns without any guidance from labeled samples.
- **Reinforcement Learning**: that involves an agent learning how to interact with an environment and receive feedback from it.

The first section of this chapter will introduce fundamental concepts of ML, in particular related to supervised learning methods. The rest of the chapter focuses on reinforcement learning and its most common implementations.

3.1 Fundamental concepts of Machine Learning and Deep Learning

To understand how Reinforcement Learning works, some basic principles common to most ML methods must be introduced.

3.1.1 Supervised and unsupervised learning algorithms

Supervised learning algorithms

Supervised learning consists in making classifications or predictions on new unseen data, based on a dataset of labeled instances. Two tasks can be distinguished:

- Classification: given a set of possible classless, it consists in assigning a new unlabeled sample to one of them. For instance, given a picture of animal, the model should be able to recognize whether it is a dog or a cat.
- Regression: it consists in predicting a real number. For example, given the set of features of a car, the model should predict the car's price.

Common supervised learning algorithms include:
K Nearest Neighbors. This method assigns a new samples to the same class that the majority of its K neighbors have. It is based on the concept that similar data has similar features, and therefore it should have the same class. The learning outcome is influenced by the number of K neighbors and the distance metric used to define closeness. The most common distance metric used is Euclidean Distance. Given two points $A = (x_1, y_1)$ and $B = (x_2, y_2)$, their distance D_e is given by:

$$D_e = \sqrt{(x_2^2 - x_1^2) + (y_2^2 - y_1^2)} \tag{3.1.1}$$

Support Vector Machine. The goal of this algorithm is finding the optimal hyperplane that best separates points belonging to different classes. This is possible by maximizing the distance (margin) between the closest points of each class (called support vectors). The employment of Kernel Functions, that map data into higher dimensional spaces, allow to perform classification even in cases where linear separation is not possible.
Tree-based methods Decision trees work by recursively splitting data into subsets based on features' values. The result is a tree-based structures, where the terminal nodes represent the final class label or regression values. There are many tree-based algorithms, one of them is XGBoost. This algorithm build many small decision tree one after the other and each new tree focuses on correcting the errors of the previous ones. In this way, the accuracy is improved by focusing on the features that are the hardest to predict [21].

Unsupervised learning

Unsupervised learning is a far more complex task than supervised learning, since the algorithm learns on its own how to identify specific patterns, without any guidance or expected output. Common unsupervised learning techniques are:
Clustering: it consists in grouping similar data points together. Close points are identified using distance metrics as it happens in KNN.
Principal Component Analysis: Principal Component Analysis (PCA) is a statistical technique used to reduce the dimensionality of the dataset, while retaining as much information as possible. It transforms the original data into a new coordinate system where the first axis captures the direction of the greatest variance in the data and the second axis is orthogonal to the first and represents the direction of the second highest variance in the data [22].

3.1.2 Deep Learning and Neural Networks

Another commonly employed technique is **Deep Learning**, an extension of ML that uses Deep Neural Networks. These models are proven to be universal function approximators, meaning that any function can be approximated up to a certain accuracy by a large-enough neural network [16]. They are composed by thousands of neurons organized in layers, where the output of the neurons in one layer, is the input of the neurons in the next. This structure can be observed in picture 3.1. Each neuron takes N numerical inputs and returns a single output y, which depends on a series of weights w and on a bias b:

$$y = \sum_{i=1}^{N} w_i x_i + b \tag{3.1.2}$$

Moreover, between each layer, there are activation functions that are used to model non-linear relationships.

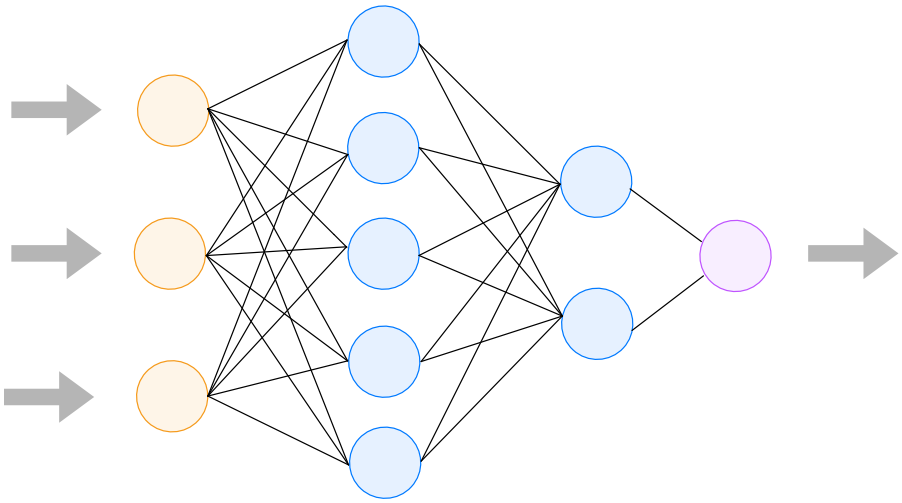

Figure 3.1: Example of Neural Network architecture

3.1.3 Training and optimization

The goal for a ML model is learning a function $f_\theta(x)$ that, given an input x, is able to generate a prediction y that is as close as possible to the actual value $\hat{y}$. The loss function measures this difference (also called residual), and the objective during training is finding the optimal parameters that are able to minimize it. Some commonly employed loss functions are:

Mean Squared Error (MSE) In this common loss function the objective is

minimizing the mean of the squared residuals

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y})^2 \tag{3.1.3}$$

Mean Absolute Error (MAE) This loss is similar to the MSE, but replaces the squared value with the absolute value of the residuals.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}|^2 \tag{3.1.4}$$

Cross-Entropy Loss (CEL) It is useful for classification tasks, as it can measure how well the predicted probabilities match the actual labels.

$$CEL = -\frac{1}{N}\sum_{i=1}^{N}[(y_i log(\hat{y_i}) + (1 - y_i) log(1 - \hat{y_i}))] \tag{3.1.5}$$

Regularization

The common procedure for training a model is splitting the dataset in training and validation set. In this way the model can learn on the first dataset and its ability to generalize is evaluated on the second.
A common risk when training a ML model is **overfitting**. It happens when the model learns noise and other patterns specific of the dataset employed for training it, and is unable to generalize on new data. The opposite problem is **underfitting**, this happens when the model is not able to grasp any relationship in the training data and it does not learn anything.
In order to avoid overfitting, a regularization term is often added to the loss. Its job is avoiding that during training the model becomes too complex and therefore unable to generalize. There are two main types of regularization:

- *Lasso*: is able to set some weights to zero, helping in selecting only the important features.

- *Ridge*: it shrinks the value of some weights, reducing the correlation between features.

By adding the regularization term to the loss function, the *cost function* is obtained. The term λ controls how strong the regularization effect is: larger value may lead to underfitting, while smaller values may contribute to overfitting. The objective function is:

$$Cost = Loss + \lambda Regularization \tag{3.1.6}$$

Optimization

To minimize the cost function and improve model performance, a widely used optimization technique is gradient descent. This iterative algorithm updates the model's parameters by computing the partial derivative of the loss function with respect to each parameter. The gradient indicates the direction in which the function's value increases the faster (steepest ascent), so the algorithm moves in the opposite direction, towards the minimum. When the gradient is large, the parameter updates are larger; as the algorithm approaches a minimum, the gradient diminishes, leading to smaller updates. This process continues until the gradient is close to zero. The update rule at each step is the following for a parameter θ:

$$\theta^{t+1} = \theta^{(t)} - \alpha \frac{\partial C}{\partial \theta} \tag{3.1.7}$$

where α is the learning rate, that determines how large the parameter's update will be [23].

3.1.4 Evaluation Metrics

A common way to evaluate a model's performance, is by looking at the number of its True Positives (TP), False Negatives (FN), False Positives (FP) and True Negatives (TN). These quantities constitute the confusion matrix of a classifier, shown in picture 3.2 metrics that compare real values and predictions are used.
In a classification problem, the most commonly employed metrics, derive from the confusion matrix:

- **Accuracy**: it measures the number of correct predictions for the positive(negative) class with respect to the total number of predictions.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.1.8}$$

- **Precision:** it measures the number of correct predictions for the positive(negative) class, with respect to the total number of instances predicted as belonging to the positive(negative) class.

$$Precision = \frac{TP}{TP + FP} \tag{3.1.9}$$

- **Recall:** it measures the number of correct predictions for the positive(negative) class, with respect to the total number of instances actually belonging of the positive(negative) class.

$$Recall = \frac{TP}{TP + FN} \tag{3.1.10}$$

- **F1-score** is the harmonic mean of precision and recall. It is useful in order to balance the two metrics

$$F1 = \frac{2Precision \cdot Recall}{Precision + Recall} \tag{3.1.11}$$

In an imbalanced dataset, the F1-score is a more reliable metric than accuracy. For instance, in a dataset presenting 95% of instances belonging to the positive class, if the model predicted all of them correctly, it would have 95% of accuracy. However, this means that none of the 5% of negative-class samples were recognized. In these types of scenarios, using F1-score, precision, and recall provides a better evaluation.

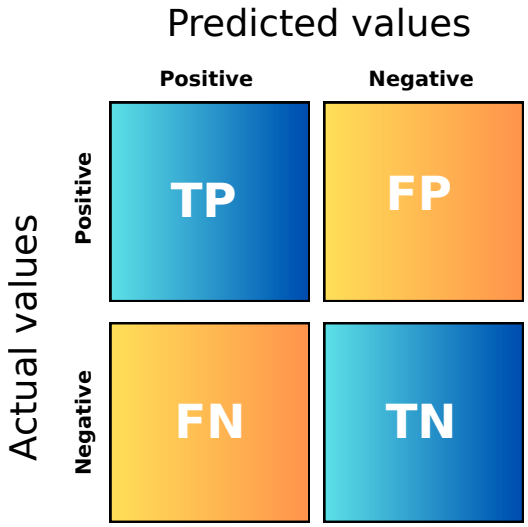

Figure 3.2: Confusion Matrix

3.2 Basic concepts of Reinforcement Learning

Reinforcement learning (RL) is a class of methods for solving sequential decision-making problems [24]. A RL task usually involves the following actors:

- an **agent**, that observes the state s of the environment,
- an **environment** that interacts with the agent,
- a **policy** π, according to which the agent acts,
- a **reward** that the agent tries to maximize over the long run,

At each time step t, the agent receives information s_t (state) from the environment. Based on this feedback, the agent selects an action a_t guided by its policy $a_t = \pi(s_t)$. The policy π is a function mapping from the current state, observed from the

environment, to the action performed by the agent. The policy can be stochastic, meaning that for a state s_t, the action a_t is determined by a probability distribution $\pi(a_t|s_t)$. After performing an action, the agent receives information about the subsequent state s_{t+1} and a reward $r_t \in \mathbb{R}$ [25]. The reward is usually a real number that is positive if the agent's action is correct, and negative otherwise. The interaction loop between the agent and environment is displayed in picture 3.3 The goal of the agent is to implement a policy such that the sum of rewards is maximized.
The following subsections introduce fundamental concepts in order to understand RL.

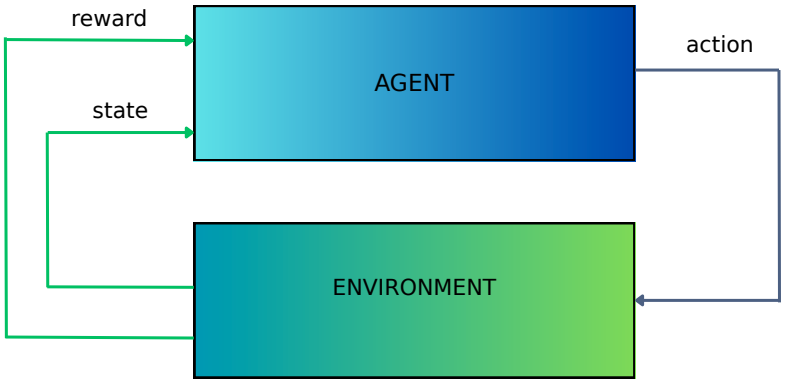

Figure 3.3: Interaction loop between the agent and environment in a RL model

3.2.1 Episodic vs continual task

An **episode** is defined as a sequence of interactions between the agent and the environment. It starts from an initial state and usually ends when a stopping criterion is met. An episode usually has a random length, but if it is fixed and known, the problem is called a *finite-horizon problem*. Based on the duration of interactions between agent and environment, it can be distinguished between:

- **continual task** if the interaction can last forever
- **episodic task** if the interaction is finite.

3.2.2 Markov Decision Process

A generic way to model sequential decision-making processes is using a Markov Process, in which the probability of an event occurring at time t depends only on

events at time $t-1$. The previously introduced RL setup can therefore be described as a finite Markov Decision Process, which is defined by the following components [13]:

- A set of states $\mathcal{S}$ the agent can observe from the environment
- A set of actions $\mathcal{A}$ the agent can execute in the environment
- A set of rewards $\mathcal{R} \in \mathbb{R}$ the agent receives from the environment.
- the probability distribution p, where the value $p(s', r|s, a)$ gives the probability that the environment transitions to the next state s' and the agent receives a reward r_t, if the agent takes action a_t in state s_t.
- the discount factor $0 \leq \gamma \leq 1$

As previously introduced, the goal for an agent is maximizing **the expected return**, which for a state at time t is defined as the sum of expected rewards, each of which is multiplied by a discount factor:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{T-1} r_{T-1} \tag{3.2.1}$$

For episodic tasks that terminate at time T, the expected return is set to zero for actions taken after T. The term γ is the discount factor; if it is too high, the agent will only focus on improving the long-term reward. On the other hand, if it is too small, the agent will only try to maximize the immediate reward.

3.2.3 Exploration-exploitation trade-off

The agent needs to explore the environment to understand which actions to take. The **exploration-exploitation tradeoff** consists in choosing between actions that the agent knows will lead to high reward, or in choosing actions that might not yield high immediate reward, but that are informative about potential future gains. If the agent simply exploits its current knowledge, it uses a **greedy policy**, meaning it always chooses the best action given the information it currently has. However, with this policy, there is no way to know whether different actions would perform well; for this reason, a ϵ-**greedy policy** is often used. With this type of policy, the greedy action (what is best known) is chosen with probability $1 - \epsilon$, and with probability ϵ, a new random action is picked. Increasing the value of ϵ leads to a more explorative policy, otherwise the policy will mostly stay exploitative.

3.2.4 Model-based and model-free RL

Model-based RL methods attempt to learn an explicit model of the environment's dynamics, how states evolve in response to agent actions, and what rewards are delivered. Once the MDP of the model is learned, the policy can be computed using

various methods, like approximate dynamic programming.
On the other hand, **model-free** methods learn directly by interacting with the environment, without a model of the environment dynamics. The two main categories of model-free methods are *policy-based* and *value-based* models.
The main disadvantage of model-free methods is that they can be very sample-inefficient, requiring many interactions with the environment before finding a good policy. However, they can learn directly from the environment without constructing a model of it.

3.3 Approaches to Reinforcement Learning

In this section, the two main sub-categories of model-free RL algorithms are introduced: policy-based methods and value-based methods.

3.3.1 Value-based RL

As previously mentioned, the goal of the agent is to find a policy π that maximizes the reward. A tool used to define optimality is the *state value* function for a state s under the current policy π:

$$V_\pi(s) = \mathbb{E}[G_t|s_t = s] \tag{3.3.1}$$

it can be interpreted as a quality measure of how good it is to be in a certain state, where quality is measured relatively to the expected return G_t. **Value-based** RL methods focus on estimating value functions. The value function for the optimal policy $\pi*$ satisfies the following Bellman equation:

$$V^*(s) = V^*(s) = \max_a \Big[R(s,a) + \gamma\, \mathbb{E}_{pS(s'|s,a)}\Big[V^*(s')\Big]\Big] \tag{3.3.2}$$

The Bellman equation is a recursive formula used in dynamic programming, in which a solution to a complex problem is found by dividing it into smaller subproblems and combining their solutions. In this case, the value function is decomposed, expressing it as a sum of the immediate reward plus the sum of the discounted expected rewards.

Q-learning

The goal of Q-learning is teaching the agent to choose actions that lead to the highest expected reward, which is expressed in terms of Q-values. The **action-value function (Q-value)** is defined as the expected sum of the discounted rewards:

$$Q_\pi(s,a) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t r_t|s_0 = s, a_0 = a] \tag{3.3.3}$$

The Q-value $Q(s, a)$ can also be interpreted as the agent's guess of how good an action is in a given state s. The policy consists of choosing the actions that maximize the Q-values:

$$Q*(s, a) = \max_{\pi} Q(s, a) \tag{3.3.4}$$

The algorithm builds a table (Q-table), where for each state and action, a Q-value is estimated. Initially, the Q-values are set randomly, then every time that the agent selects an action a_t and observes a reward r_t, the Q-values are updated according to the following Bellman equation:

$$Q(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_{a} Q(s_{t+1}, a) - Q(s, a)] \tag{3.3.5}$$

Q-values are updated by comparing the old guess ($Q(s_t, a_t)$) with the new evidence, which is given by the sum of the current reward and the maximum reward for the next state over all possible actions: $\max_a Q(s_{t+1}, a)$. The learning rate α determines how much the agent updates its old Q-value in response to new evidence. The discount factor γ determines how much current rewards count relative to long-term rewards. The quantity $r + \gamma \max_a Q(s_{t+1}, a) - Q(s, a)$ is also known as *Temporal difference*. Q-learning functioning is illustrated in Algorithm 1.

Algorithm 1 Q-Learning

1: Initialize Q-table $Q(s, a)$ arbitrarily for all states s and actions a
2: Set learning rate $\alpha \in [0,1]$ and discount factor $\gamma \in [0,1]$
3: **for** each episode **do**
4: Initialize starting state s_0
5: **for** each step of the episode **do**
6: Select action a_t from state s_t using a policy derived from Q
7: Execute action a_t, observe reward r_t and next state s_{t+1}
8: Update Q-value:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[r_t + \gamma \max_{a} Q(s_{t+1}, a) - Q(s_t, a_t)\Big]$$

9: $s_t \leftarrow s_{t+1}$
10: **end for**
11: **end for**
12: **Policy derivation:** $\pi(s) = \arg\max_a Q(s, a)$

Deep Q-learning

In larger state-action spaces, a simple Q-table is not efficient; for this reason, it is replaced by a Neural Network (NN), called Deep Q-Network, and the algorithm is named **Deep Q-learning**. The loss is a parametric function, with respect to

parameters $\mathbf{w}$, given by the squared Temporal Difference:

$$\mathcal{L}(\mathbf{w}|\mathbf{s}, a, r, \mathbf{s}') = ((r + \gamma \max_a Q_w(\mathbf{s}', a) - Q_w(\mathbf{s}, a)))^2 \tag{3.3.6}$$

The Loss Function can be updated using Stochastic Gradient Descent. The difference between the predicted Q-value and the target optimal value is continuously backpropagated through the network to update the parameters [26]. However, the issue with this loss function, is that the target value $Q *_w (s, a)$ uses the same parameters of the function that is being updated. In non-linear systems such as NNs, this can cause instability; for this reason, a proposed solution is to use a **target network** [27]. Specifically, an extra copy of the original network with the same structure is computed earlier in training and is responsible for computing the target value. Periodically (usually after a few episodes), the weights of this second network are updated as follows:

$$\mathbf{w}_{target} \leftarrow \mathbf{w} \tag{3.3.7}$$

In this way, when updating the loss, the phenomenon of the network "chasing its own tail" is avoided by keeping the target value constant for a few episodes.
Another employed technique in Deep Q-learning is the **experience replay**, first introduced by [28]. Instead of updating the network immediately after every interaction with the environment, the agent stores its experiences as tuples s_t, a_t, r_t, s_{t+1} in a replay memory (or buffer). During training, mini-batches of experiences are randomly sampled from this buffer to update the network. This approach has two main benefits. First, it breaks the correlation between consecutive samples, with the goal of increasing variability in data in order to avoid overfitting. Second, it allows the agent to reuse past experiences multiple times, improving data efficiency and stability.

Benefits and drawbacks of value-based methods

Value based methods have the benefit of learning optimal actions through trial and errors, making them suitable for applications such as robotics. Their deterministic behavior, where for each combination of state and action a specific value is returned, make them less prone to high variance and instability during the learning phase.
However, computing the value for each state and action results in long training time per episode, and limits their application to larger action-state spaces.

3.3.2 Policy-based RL

Policy-based RL methods learn a parameterized policy that directly maps states to actions, without explicitly computing value functions. Similarly to Q-learning, the goal is finding the optimal parameters such that the policy will maximize the

excepted cumulative reward. Usually, a NN is employed as a function approximator for the parametrized policy $\pi_\theta(a|s)$.
In order to find the optimal parameters, most policy-based models use the **policy gradient method**, which consists in using gradient ascent to find the optimal policy. The objective function for policy gradients is defined as the expected cumulative reward:

$$J(\theta) = \mathbb{E}_{\imath_\theta(\tau)}[R(\tau)] = \int p_\theta(\tau)R(\tau)d\tau \tag{3.3.8}$$

where $R(\tau) = \gamma^0 r_0 + \gamma^1 r_1 + ...$ is the return along a trajectory τ, which is a sequence of states, actions, and rewards. While $p_\theta(\tau)$ is the probability distribution over possible trajectories that could happen under the policy $\pi_\theta(a|s)$ and the environment dynamics. In order to maximize it, the gradient is computed as follows:

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau)R(\tau)d\tau = \int p_\theta(\tau)\frac{\nabla_\theta p_\theta(\tau)}{\theta p_\theta(\tau)}R(\tau)d\tau = \mathbb{E}_\tau[\frac{\nabla_\theta p_\theta(\tau)}{\theta p_\theta}R(\tau)] \tag{3.3.9}$$

This function can be maximized by taking the partial derivative of the objective with respect to the policy parameter θ. Considering the log derivative trick $\nabla log x = \frac{\nabla x}{x}$, the above gradient can be rewritten as follows [29, 24]:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta log \pi_\theta(a|s)R(\tau)] \tag{3.3.10}$$

The parameters can be updated using gradient ascent in the following way:

$$\theta \leftarrow \theta + \alpha \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, R(\tau) \tag{3.3.11}$$

REINFORCE

REINFORCE [30] is a policy gradient method based on Monte Carlo sampling, which means that it collects trajectories to compute the gradient of the expected reward. This avoids the need to compute and update value functions. REINFORCE functioning is illustrated in Algorithm 2.

Algorithm 2 REINFORCE algorithm

Initialize policy parameters θ
Set learning rate $\alpha \in [0,1]$ and discount factor $\gamma \in [0,1]$
for each episode **do**
    Generate a trajectory $(s_0, a_0, r_0, \ldots, s_T)$ using policy $\pi_\theta(a \mid s)$
    for $t = 0$ to T **do**
        Compute return: $G_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$
    end for
    Estimate policy gradient: $\hat{g} = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, G_t$
    Update policy parameters: $\theta \leftarrow \theta + \alpha \hat{g}$
end for

For each episode, the REINFORCE algorithm uses the current policy to sample a trajectory of length T. Each step of the trajectory corresponds to a state, action and reward $(s_0, a_0, r_0, ...s_T)$. Once the trajectory ends, the expected return is computed by summing the return for each step of the trajectory. The return is used to estimate the policy gradient using equation 3.3.10, and the parameters are updated with gradient ascent using equation 3.3.11. If the return corresponding to a certain state-action combination is high, the resulting probability distribution will favor those action, otherwise if will shift away from them.

Benefits and Drawbacks of policy based methods

Policy-based methods are able to learn directly a stochastic policy while methods based on value functions are not. The first immediate benefit of this, is that training time per episode is faster, since there is no need to compute value-functions at each step. This also makes it more adapt in high-dimensional action spaces, where value-based method would requiring outputting a Q-value for each action. The second advantage is that the exploration/exploitation trade-off mechanism does not need to be implemented by hand (for example with a $\mathcal{E}-$greedy policy), but it is inherent to the method, since the output is a probability distribution.
Since in stochastic environments taking the same action in the same state does not always give the same result, the same initial state could lead to different trajectories of states, actions and rewards. This high variability in trajectories leads to large differences in total reward across episodes, and high variability is introduced in the return estimates. This high variance is problematic, because it can lead to noisy gradients updates, requiring more episodes to actually improve the policy [31].

Chapter 4

Quantum Reinforcement Learning

4.1 Quantum Machine Learning

Quantum Machine learning (QML) consists in applying quantum computing to Machine Learning (ML) techniques. The ability of qubits to be in superposition states, and therefore to execute multiple operations at the same time, leads to faster problem-solving. The goal is to exploit these properties by adapting classical algorithms to run on a quantum computer or, more likely, in a simulator. QML techniques can be classified in the following way [32]:

- **Classical Algorithm-Classical Data**: classical ML approaches.
- **Classical Algorithm-Quantum Data**: it consists in training classical ML models with quantum data.
- **Quantum Algorithm-Quantum Data**: it involves a quantum computer processing quantum data. It is the least explored approach due to current hardware limitations.
- **Classical Data-Quantum Algorithm**: it consists in processing classical data with the employment of a quantum computer in the model or during training. This approach will be applied in the following work.

Common QML approaches include Quantum K-Nearest-Neighbors (Quantum KNN), Quantum Support Vector Machines (Quantum SVM), and Quantum Neural Networks (Quantum NN).

4.1.1 Quantum KNN

KNN is a popular classification algorithm. It assumes that features close to each other encode similar samples. Authors in [33] introduce a method to measure how similar two quantum states are using a **swap test**. In particular, given two quantum states $|a\rangle, |b\rangle$ and an extra qubit (ancilla), the SWAP operation is used to create an interference pattern, from which the probability of the ancilla being in state $|0\rangle$ is indicative of the similarity between the two states.

4.1.2 Quantum Support Vector Machines

Support Vector Machines (SVM) is a classification algorithm that, in its simplest form, finds the hyperplane that best separates two classes. This works if data is linearly separable; when this is not possible, SVM employs the **kernel trick**, which consists in using a function that maps data points into a new space where they can be separated. This is possible by computing the dot product between the feature vector and the kernel function's weight vector. Authors in [34] showed that the evaluation of inner products between classical vectors can be performed efficiently on a quantum computer by using a quantum kernel. The quantum kernel maps data into high-dimensional spaces more efficiently, resulting in better performance.

4.2 Quantum Neural Networks

In the field of QML, a QNN is a type of model inspired by the behavior of a classical Neural Network (NN) . Similar to classical NNs, there are three main steps in QNN: data preparation, data processing, and data output [16].

- **Data preparation**: In order to be processed by a quantum circuit, classical data should be embedded into a quantum state using a **feature map**. Depending on the chosen feature map, the input data may be further normalized or scaled.
- **Data processing**: The processing stage of a QNN consists in the application of a circuit that depends on some optimizable parameters.
- **Data Output**: a measurement operation is applied in order to return a classical output.

4.2.1 Encoding

Classical data cannot be processed by a quantum circuit as it is, but it must be encoded into quantum states. The following section illustrates common types of encoding (also called *feature maps*)

Basis Encoding

In basis encoding, every bit with value 1 gets replaced with a qubit in state $|1\rangle$, and every bit of value 0 gets replaced with a qubit in state $|0\rangle$. In this way, a n-bit string is associated with a n-qubit system. To encode a vector of real values, it first needs to be translated into a binary sequence, afterwards it can represented in a quantum state. For instance the vector $x = (0.1, -0.6, 1.0)$ is first transformed into $x = (00001, 11001, 01111)$, and then is encoded into $|0000 \quad 11000 \quad 01111\rangle$ [32]. This type of encoding is straightforward, but is often considered too simplistic, since it may lose other important properties of the data.

Amplitude encoding

Amplitude encoding maps a classical input vector $\vec{x}$ into the **amplitudes** of a quantum state. Using n qubits, one can represent a vector of size 2^n, with each basis state $|k\rangle$ corresponding to one entry of the vector. The encoded quantum state is

$$|\phi(\vec{x})\rangle = \frac{1}{\|\vec{x}\|} \sum_{k=0}^{2^n-1} x_k |k\rangle , \tag{4.2.1}$$

where $\|\vec{x}\| = \sqrt{\sum_k x_k^2}$ ensures that the resulting quantum state is normalized. This encoding is valid for any non-zero input vector, since the zero vector cannot be normalized [16].

Consider a 2-qubit system ($n = 2$), which can represent a vector of size $2^2 = 4$. Let the input vector be $\vec{x}$ = (2,1,2,3), then its norm is $|\vec{x}\| = \sqrt{2^2 + 1^2 + 2^2 + 3^2} = \sqrt{18}$. The amplitude-encoded quantum state is:

$$|\phi(\vec{x})\rangle = \frac{1}{\sqrt{18}} \left(2 |00\rangle + 1 |01\rangle + 2 |10\rangle + 3 |11\rangle\right) . \tag{4.2.2}$$

Angle encoding

Angle encoding converts classical data into a quantum state through a rotation angle. For instance, in order to convert the value $x = 0.5$ using a R_Y on the initial qubit state $|0\rangle$, the value must be plugged in the following equation:

$$|\psi\rangle = R_Y(x) = cos(\frac{x}{2}) |0\rangle + \sin\left(\frac{x}{2}\right) |1\rangle \tag{4.2.3}$$

after computing the probability of measuring $|0\rangle$ and $|1\rangle$, by replacing $x = 0.5$, the resulting state becomes:

$$|\psi\rangle = 0.968 |0\rangle + 0.247 |1\rangle \tag{4.2.4}$$

In this type of encoding, a qubit is required for each input value to be encoded, making it ideal for shallow architectures. Scaling input to [-1,1] or [0, π] ensures that input features are mapped to a meaningful portion of the rotation angle.

ZZ encoding

In the ZZ feature encoding, the Hadamard gate is firstly applied to each qubit, putting them in superposition. Every pair of qubits is connected through a controlled-NOT gate, followed by a Z-rotation on the target qubit that depends on both inputs. Then the CNOT is repeated to complete the entangling operation. This sequence creates an entangled feature space that can separate complex patterns in the data.

4.2.2 Variational Quantum Circuits

In QML, particularly in QNN, the main technology used for classification tasks is Variational Quantum Circuits (VQCs). A VQC is composed of three main parts: an input encoding, a parametric circuit optimized by a classical algorithm, and a final measurement. This structure is illustrated in figure 4.1. The types of encoding were already introduced in section 4.2.1, while measurement in quantum systems was discussed in section 2.6.

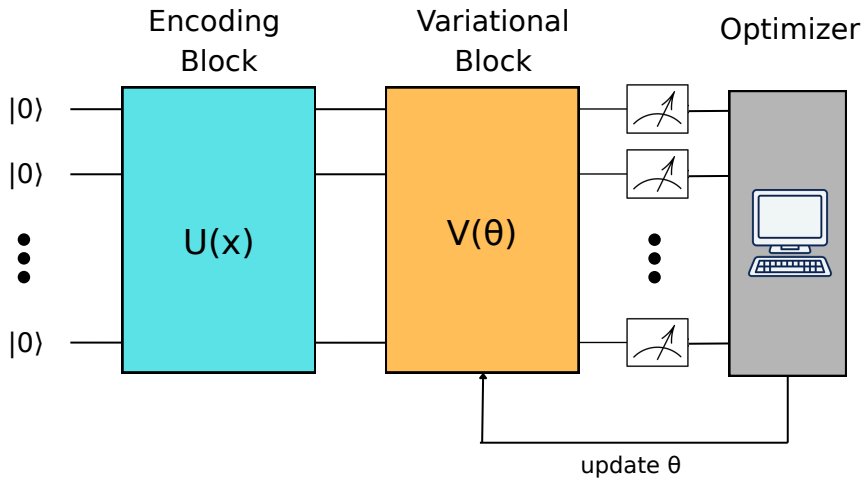

Figure 4.1: Building blocks of a variational quantum circuits

Once features are encoded in a quantum state, they are fed into a parametric quantum circuit for processing.
The parametrized circuit is also called **variational form** or **ansatz**. The structure of the ansatz is chosen based on the problem to solve. It must be sufficiently expressive to capture the essential features of the data (strongly entangled ansatz), and at the same time it should remain sufficiently simple in terms of qubit count

and circuit depth, to allow practical implementation on a quantum device (basic). Data is processed by applying a series of parametrized rotational gates, which are optimized using gradient-based methods. The loss function used to optimize the model is typically computed from the qubit measurement outcomes [35]. In this way, a classical optimizer running on traditional hardware can be used for the optimization, leveraging mature algorithms and architectures. At the same time, the properties of quantum computers can be exploited by running the processing part on an NISQ device, without requiring large numbers of qubits for implementing the the optimization part.

4.2.3 Gradient computation

As it happens in classical NNs, the parameters of the variational circuits used to process data within the QNN need to be periodically updated. A commonly employed optimization algorithm is gradient descent. In order to make this algorithm work there are multiple differentiation methods that can be employed for hybrid classical-quantum architectures:

- **Numerical approximation**: it is useful when the gradients cannot be directly computed, and therefore need to be estimated. Given a function f taking n real-valued inputs and a sufficiently small value h, the sensitivity of f to the parameter x_j can be estimated as shown in equation 4.2.5

$$\frac{\partial f}{\partial x_j} \approx \frac{f(x_1, \ldots, x_j + h, \ldots, x_n) - f(x_1, \ldots, x_j, \ldots, x_n)}{h} \tag{4.2.5}$$

  The difference between f with a small increment and f with the normal input values is computed. The small change is divided by h. The quantum circuit is run multiple times, changing parameters and estimating gradients.

- **Parameter shift rule:** suppose there is a circuit consisting of a single rotation gate $R_X(\theta)$, and that the measurement of its expectation $E(\theta)$ is known. The derivative can be computed with respect to θ by running the circuit twice with that parameter slightly shifted in both directions as shown in equation 4.2.6

$$\frac{\partial E}{\partial \theta} = \frac{1}{2}(E(\theta + \pi/2) - E(\theta - \pi/2)) \tag{4.2.6}$$

4.3 Quantum Reinforcement Learning with Variational Quantum Circuits

Research on QML mostly focuses on supervised and unsupervised learning. However, in recent years, there has been interest in Quantum Reinforcement Learning (QRL), particularly in methods that employ VQC as a function approximator.

In the typical pipeline, the state information s is preprocessed and encoded in a quantum state via a feature map. Using the current variational parameters θ_t, a quantum state is obtained and an observable O_a associated with action a is measured. Its expected value is post-processed to represent a value function $Q_\theta(s, a)$ or the policy $\pi_\theta(a|s)$. This function will be employed to sample an action a_t and execute it in the environment. A classical optimizer updates the VQC parameters. The gradients are often computed using a parameter-shift module, with respect to the variational parameters. This process goes on for several episodes [13].
There are two main sub-categories of QRL techniques employing variational quantum circuits:

- Value-Function Approximation
- Policy Approximation

In the following subsections, the most widely studied implementations of these techniques are presented to highlight their benefits and drawbacks.

4.3.1 Value-Function Approximation

The work by [36] was the first to propose implementing the Q-learning algorithm by replacing the Deep Q-network with a VQC. Similar to the classic Deep Q-learning, techniques such as a target network and experience replay are used (both introduced in previous sections). The inputs were first transformed into quantum states using basis encoding. The training was performed by minimizing a loss function using gradient descent to update the parameters. Specifically, the loss function used is the Mean Squared Error:

$$\mathcal{L} = E[(r_t + \gamma \cdot max_{a'} Q_{\theta'}(s_{t+1}, a') - Q_\theta(s_t, a_t))]^2 \tag{4.3.1}$$

The proposed algorithm was evaluated in two distinct environments. The first, FrozenLake, consists of 16 states, 4 possible actions, and 4 qubits. While the second, CognitiveRatio, was used with a VQC containing 2-5 qubits. The authors declare that their VQC-based algorithm achieves performance comparable to that of a classical NN while using fewer parameters. However, the environments used were small-scale; therefore, this claim should be supported by further experiments. Moreover, the encoding scheme could be overly simplistic.
There are numerous expansions of this study; for example, authors in [37] use a so-called "directional" encoding that encodes the sign of the value. They use a CartPole environment and a Blackjack environment with 2 actions, and VQCs with 4 and 3 qubits, respectively. Also in this second case, the authors report a reduction in the number of parameters compared to the classical model. A downside of this experiment could be the directional encoding employed, since most models also benefit from value information rather than just the sign.

A more complex parameter setting is presented in [38], where the authors propose a hybrid model composed of a classical NN to encode the state, which is then fed to the VQC. They use Pong-v0 and Breakout-v0 as environments, varying the number of qubits from 5 to 15. However, these models perform poorly compared to a classical CNN.

4.3.2 Policy Approximation

These techniques use a VQC as a function approximator for the stochastic policy. The work proposed in [39] is the first to implement this type of architecture. They implement two types of policies:

1. **RAW-VQC policy**: where the probability of measuring a certain action is given by the squared magnitude of that action the quantum state (Born's rule presented in 2.6):
$$P(a) = |\langle a|\phi\rangle|^2 \tag{4.3.2}$$
where the quantum state $|\phi\rangle$ contains all possible outcomes mixed together
2. **SOFTMAX-VQC**: in which the quantum circuit produces an expectation value for each action, describing how strongly each possible action is favored, and a softmax layer turns these values into probabilities.

The architecture uses single and two-qubit gates, and input data is encoded with angle encoding. The authors introduce learnable state-scaling parameters that multiply the input state values before encoding. In the work, multiple environments were tested, including CartPole and Acrobot. A series of ablation studies revealed that the SOFTMAX-VQC policy outperformed RAW-VQC, and that increasing the circuit depth improved performance. Moreover, incorporating learnable state-scaling parameters improves accuracy. The authors state that their results provide both theoretical and empirical indications of a potential quantum advantage in RL using a VQC. However, they also acknowledge that the practical realization of this advantage remains beyond the capabilities of current NISQ devices.

Using a similar architecture, authors in [40] propose a quantum version of RE-INFORCE with a VQC-based function approximator. The authors state that it is possible to estimate the log-policy gradient with only a logarithmic number of samples (in terms of the number of variational parameters). While this is true in simulation, it has not been verified on quantum hardware [13].

Part II

Implementation

Chapter 5

Dataset

This chapter presents the dataset used to train and evaluate both the classical and the quantum models. The main characteristics are reported, followed by the pre-processing steps needed to embed the data into the quantum devices

5.1 Dataset Analysis

The dataset employed in this work is the **Fraud Detection Transaction Dataset** [41]. It is a synthetic dataset that contains 50000 records and 21 features organized as follows:

- 8 numericals
- 6 categoricals
- 4 booleans
- 3 non-predictive: UserID, TransactionID and Timestamp

A preliminary analysis was conducted to verify its properties and understand what type of pre-processing techniques apply. Since the dataset is synthetic, it presents some unusual characteristics with respect to other financial transaction datasets. For example, it does not contain duplicates or missing values and the majority of variables does not present outliers. These peculiarities are illustrated in the following subsections.

5.1.1 Missing data and duplicates

Most financial transaction datasets have an high number of missing or duplicate values. This could be due to human errors, machine errors or it could be a signal of fraudulent activities. However, this dataset does not present this type of irregularities, as the number of both missing values and duplicates is zero.

5.1.2 Class Imbalance

As already mentioned, real class imbalance datasets have less than 1% of transactions labeled as fraud. In this dataset, the percentage of transactions labeled as non-fraudulent is 68%, while fraudulent transactions are 32%, as shown in Figure 5.1. Having more instances of the fraudulent class helps the model to better learn the patterns associated with that class. Despite this, the dataset remains moderately imbalanced. In this way the model will not learn to expect frauds too frequently, reflecting the rarity of fraudulent patterns of real-life datasets, and improving the generalization capabilities of the model.

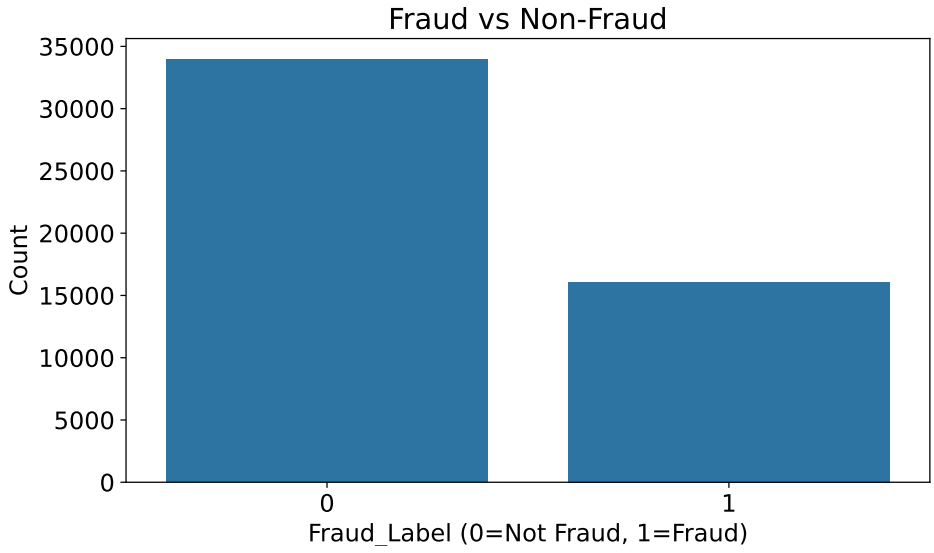

Figure 5.1: Distribution of target class

5.1.3 Absence of Correlation

Another peculiarity is the absence of correlation between features. Correlation between two numerical variables X and Y can be measured using the Pearson's Correlation Coefficient shown in the following equation:

$$r = \frac{\sum(x - \bar{x})(y - \bar{y})}{\sqrt{\sum(x - \bar{x})^2 \sum(y - \bar{y})^2}} \tag{5.1.1}$$

where x and y are the elements of X and Y, while $\bar{x}$ and $\bar{y}$ are their mean. The value of the Pearson's coefficient is in the range [-1,1], where -1 indicates full negative correlation, 0 indicates no correlation and 1 indicates full positive correlation. An higher magnitude of the coefficient, implies stronger correlation between two variables [42]. If there are strongly correlated features, one of them is usually removed to avoid redundancy in the dataset and reduce its dimensionality.

These coefficients can be used to build the Feature Correlation Matrix shown in Figure 5.2, where each entry shows the correlation between two variables. It can be observed that features are not correlated, since all correlation coefficients have value zero or around zero.

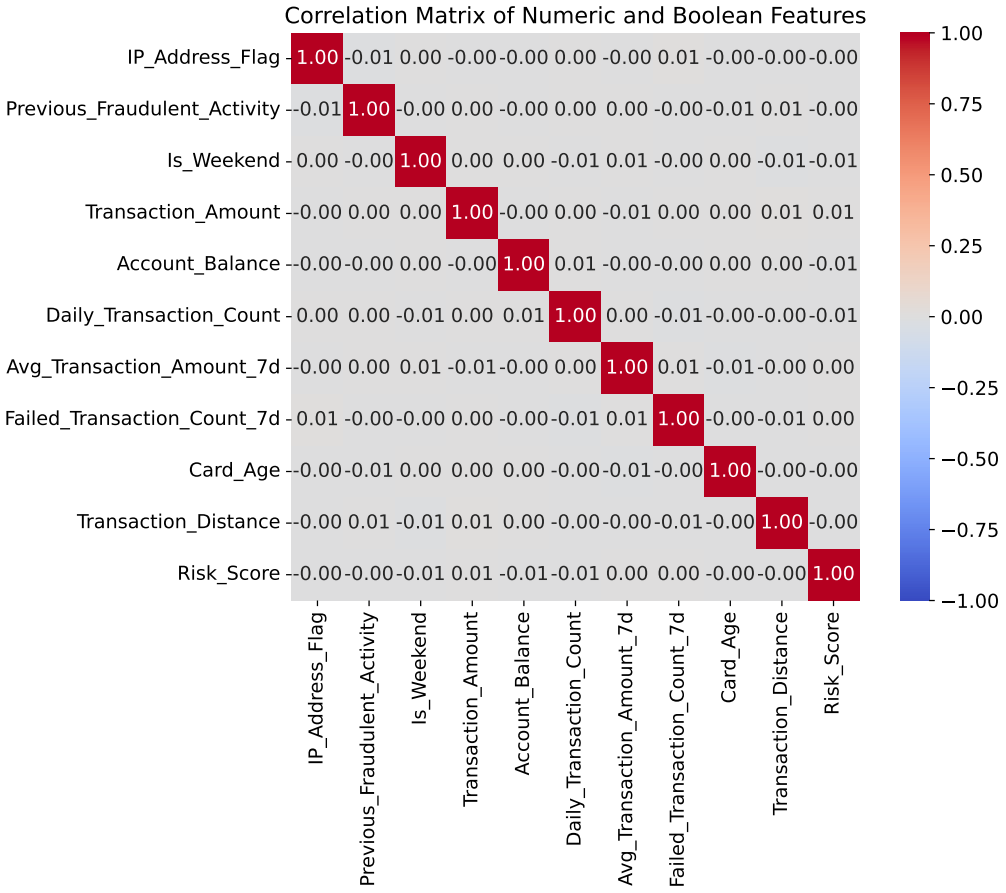

Figure 5.2: Correlation Matrix of Numerical Features

5.1.4 Skewness and class distribution

Feature skewness is a measure of how symmetric the feature distribution is around its mean value. Negative skewness means that the tail of the distribution is on the left side, and positive skewness means that the tail of the distribution is on the right side [43]. For example, the feature “Transaction_Amount” shown in Figure 5.3 has positive skewness.

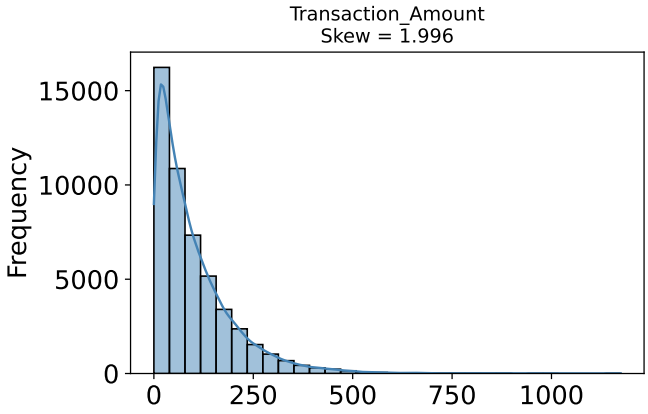

Figure 5.3: Skewness of Transaction Amount

The skewness of a feature can be quantified using Fisher-Pearson standardized moment coefficient, that can be computed as:

$$skewness = \frac{E[(X - \mu)^3]}{\sigma^3} \tag{5.1.2}$$

for a feature X, with mean μ and sample standard deviation σ. The standard deviation is computed as the sample standard deviation in the following way:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} \tag{5.1.3}$$

where x_i it is each data point, $\bar{x}$ it is the sample mean and n is the number of samples.
If the Fisher-Pearson coefficient is less than zero the distribution is left skewed, otherwise if it is greater than zero it is right skewed. If the coefficient has value zero or around zero the distribution is roughly symmetric. This dataset presents only a few features with skewness above zero, and no feature with negative skewness. In fact, as shown in Figure 5.4, the majority of variables have skewness around zero and therefore symmetric distribution. This is unusual because in a real financial transaction dataset there would be much more features with asymmetric distribution. For instance, the feature "Account_Balance" has a skewness equal to zero, while normally there are few accounts with a very high balance and the majority with a low/medium balance, resulting in an asymmetric distribution of the feature values.

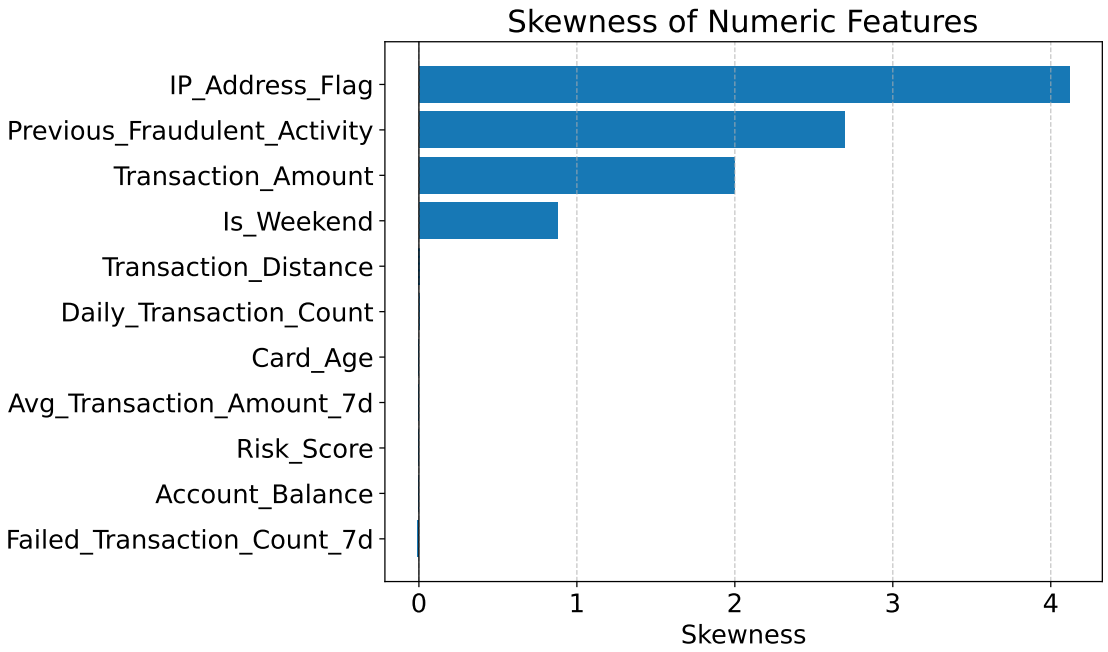

Figure 5.4: Skewness of Numerical Feature

Categorical features are also affected by an unusually balanced distribution. Figure 5.5 shows the distribution of "Transaction_Type", where it can be observed that samples are almost perfectly balanced across all categories. All categorical features in the dataset exhibit a balanced distribution of elements within their categories.

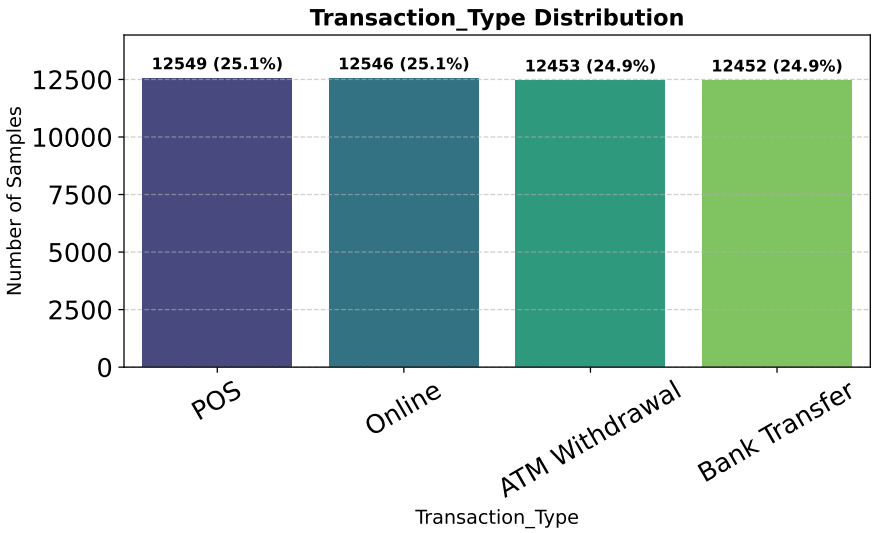

Figure 5.5: Distribution of categorical feature "Transaction_Type"

5.2 Preprocessing

Since the dataset does not present errors, cleaning procedures such as filling missing values and removing duplicates are not necessary. Additionally, class imbalance is present, but is not as excessive as it would happen in a real transaction dataset. Therefore, no procedure was employed to balance the two classes, otherwise the model would risk to overfit. Moreover, only the features "TransactionsID" and "UserID" were removed, since they do not have any predictive capability. Finally, since there are not any highly correlated features, the dataset does not present redundancy, therefore all the other features are kept in order to enhance predictive capabilities.
The main focus in this section is about other preprocessing steps, such as data split, feature encoding, standardization and Principal Component Analysis (PCA).

5.2.1 Data split

Splitting data into training, evaluation and test set using the right proportion is essential to ensure that the model has a significant amount of data to learn, but also to be evaluated.
Transactions were organized in chronological order according to the feature "Timestamp". Starting from the oldest transaction, the dataset was split in the following way:

- Training set: first 70% of transactions
- Evaluation set: next 20% of transactions
- Test set: remaining 10% of transactions

In this way, during the training phase, the agent learns from past transactions and it is evaluated and tested on future transaction, that contain new unseen patterns.

5.2.2 Encoding

In order to work with most of Machine Learning (ML) models, categorical variables need to be turned into numerical features. The chosen technique for converting categorical features into numerical was One Hot Encoding. It consists in representing the categorical feature as a binary vector, in which '1' indicates that a category is present and '0' its absence. Using One-Hot encoding increases the dimensionality of the dataset, because a separate column is created for each category of the original variable. This could lead to slower training time and slower convergence of the model. However, since the employed dataset does not have many categorical variables, and since PCA will be applied, using One-Hot Encoding does not affect the performance.

5.2.3 Standardization

Feature scaling consists in transforming the data to fall within a smaller range of values (for example [-1,1] or [0,1]) or such that it follows a certain distribution (for example the Gaussian distribution). Feature scaling techniques are useful for many ML algorithms, in particular distance-based classifiers, but also in Neural Networks backpropagation to speed up the learning phase [44].
Features are initially scaled using **Z-score Normalization** or **Standardization**, which transform data in order to have unitary variance and zero mean (so, to follow the Gaussian Distribution). The transformation consists in applying the following equation:

$$X_{std} = \frac{X - \mu}{\sigma} \tag{5.2.1}$$

where X is the original feature, μ is its mean and σ the standard deviation. This technique ensures that features with larger scales will not dominate those with smaller values. This is useful since the Neural Networks used in Q-learning and REINFORCE, might be sensitive to outliers in the input data. Although Standardization is not necessary for XGBoost, since it is a tree-based method that does not rely on computing distances, it was applied to maintain consistency when comparing it to other classical methods.
Further scaling was performed for the quantum architectures, by scaling data within the range [-1,1]. While this is particularly useful for certain quantum encoding strategies, such as angle encoding, it also improves numerical stability throughout the entire training process.

5.2.4 PCA

As introduced in section 3.1, PCA is a statistical technique used to reduce the dimensionality of the dataset, while retaining as much information as possible. In this work, the number of principal components used is 6. Although this affects the models classification capabilities (as it will be shown in Chapter 7), is a necessary step to work with quantum circuits in simulated environments. For instance, if a circuit employs angle encoding in a dataset with 18 features, it would require 18 qubits to encode each feature into a rotation angle. Working with a quantum circuit that uses 18 qubits is impossible in simulators due to computational constraints, therefore using PCA to reduce the number of qubits/components to 6, is necessary to conduct the experiment.

Chapter 6

Architecture and parameters

The goal of this chapter is to illustrate the implemented Quantum Reinforcement Learning (QRL) architectures. As previously mentioned, two architectures were built based on their classical counterparts:

- Quantum Q-learning
- Quantum REINFORCE

Both classical and quantum models use the same Environment and the same optimizer; the main difference lies in the Agent. In fact, while the classical agent uses a Neural Network (NN), in the quantum architectures this is replaced by a Quantum Neural Network (QNN) based on a Variational Quantum Circuit (VQC).

6.1 Optimizer and Environment

In this section, the optimizer and environment used for both QRL architectures are described.

6.1.1 Optimizer: ADAM

The majority of optimizers are based on Gradient Descent algorithms, which aim to minimize the loss function of ML models. This algorithms works by iteratively adjusting the model's parameters in the direction that minimizes the loss function the most, which is the direction of the negative gradient. A well-known variant of this algorithm is Stochastic Gradient Descent, which, instead of using the entire dataset at once to update the parameters, uses batches of data (subsets of the dataset) to speed up optimization.
In this work, the employed optimizer is ADAM [45], which combines the advantages of two stochastic gradient descent extensions:

- **Momentum**: it keeps track of the direction and magnitude of past gradient updates and uses them to update the current gradient, reducing oscillations.
- **RMSprop**: is an optimization algorithm designed to adapt the learning rate for each parameter individually, based on the history of past squared gradients.

By combining these two methods, ADAM can guarantee convergence and efficient parameter updates.

6.1.2 Environment

The Environment is a custom OpenAI Gym environment [46], created to model fraud detection as a sequential-decision making task. In each episode, the environment provides the agent with a fixed-length sequence of transactions. The agent must decide whether to label them as fraudulent or not. The environment returns a reward based on whether the agent's prediction matches the true label. The Environment is composed of the following modules: Input Data, Action Space, Observation Space, Reward Function, and Episode Progression mechanism.

Input Data

The environment receives transactions and their features from the dataset, the number of transactions processed per episode, and a binary label vector. In each episode, transactions are processed sequentially. By analyzing non-overlapping blocks of transactions, the agent learns temporal sequences corresponding to separated subsets of the dataset.

Action Space

It is the set of all possible actions that the agent can perform. In this case, just two actions are possible:

- **Action 0:** the transaction is approved, it is not fraudulent
- **Action 1:** The transaction is not approved; it is fraudulent

Observation space

The observation space is the information that the agent receives from the environment; it is also known as *state*. For a transaction, the state is a vector with all its features. At the end of each episode, the environment provides the agent:

- the reward, computed for the action the agent has just taken.
- a new state, which is the feature vector for the new transaction that needs to be classified.

The agent must learn to classify this current transaction using its features (the state) and the reward received for the previous action. Features of past transactions that have already been classified should not be used.

Reward function

At each step, the environment retrieves the true labels of the current transactions and computes a reward based on the correctness of the agent's action. The reward function works as shown in Table 6.1. This reward feature is designed to encourage

Action	True Label	Reward
Approve (0)	0	1
Approve (0)	1	-3
Flag (1)	0	-2
Flag (1)	1	4

Table 6.1: Reward function based on action and true label.

the model to correctly identify fraudulent actions and heavily penalize failure to detect them.

Episode Progression

Episodes are created by slicing the dataset into non-overlapping chunks. Thanks to a reset function, when a new episode begins, the environment advances to the next block of transactions and sets the first transaction of the episode as the initial state. The environment then provides to the agent the following information:

- the new initial state, corresponding to the first transaction of the new episode
- a termination flag, to communicate the end of the previous episode and the start of a new one.
- the reward for the agent's action on the previous transaction (the last transaction of the previous episode)

6.2 Agent

The Agent is responsible for learning a policy that maximizes the total expected reward. In both cases, Q-learning and REINFORCE, the function representing the Agent is realized by replacing a classical NN with a QNN based on a VQC, whose structure was introduced in section 4.2.2. However, the two architectures differ in how the VQC output is interpreted and in how gradients are used during optimization. Picture 6.1 shows the schema for the quantum Q-learning architecture, while Picture 6.2 displays the quantum REINFORCE architecture.

6.2.1 Shared architecture: VQC-based function approximator

For both agents, the same hybrid classical-quantum QNN was implemented, which combines:

1. A classical prepossessing stage, which maps the state from the environment into a vector suitable for the VQC.
2. A VQC that outputs one Q-value per action for the Q-learning architecture, and one logit per action for REINFORCE.

Classical Encoder

The classical encoder applies a linear transformation to the input data. Specifically, the linear layer is a type of fully connected layer that, given the dimension of the input vector x and the output vector y, applies the following operation:

$$y = Wx + b \tag{6.2.1}$$

where W and b are the weight matrix and bias vector, and are both trainable. The linear transformation applies a trainable linear operator to the input, improving the representation fed to the quantum circuit. In the current implementation, the dimension of the input vector is the number of features in the dataset, while the dimension of the output vector is the number of qubits in the circuit. In this case, since PCA was applied, the input vector has dimension 6, which matches the number of qubits. If the input and output dimensions were different, the linear layer would additionally perform dimensionality reduction or expansion to match the output vector dimension.
Since the linear transformation applies arbitrary weights, it may push values outside the range [-1, 1], which is necessary for some types of quantum encoding, such as angle encoding. For this reason a Tanh transformation, shown in equation 6.2.2, is applied to ensure data still fall within [-1,1]:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{6.2.2}$$

VQC

The encoded state is fed into the VQC. The VQC used 6 qubits and 6 layers for both the Q-learning and REINFORCE algorithms. The ansatz type, rotation gate, and encoding type were considered trainable hyperparameters. As reported in section 6.3.2, hyperparameter tuning was performed separately for quantum Q-learning and quantum REINFORCE. As a result, the VQC has the following configurations:

- Q-learning: encoding=ZZ, rotation gate=Z, ansatz=basic
- REINFORCE: encoding=angle, ansatz= strongly

6.2.2 Q-learning agent

The quantum Q-learning follows the training framework introduced classical for Q-learning and Deep Q-learning in section 3.3.1.

Action selection

The agent uses an ϵ-greedy selection mechanism, in order to balance the exploration/exploitation trade-off introduced in section 3.2.3. As shown in Equation 6.2.3, the agent mostly follows an exploitative policy, choosing actions that lead to the maximum reward most of the time. While exploratory actions are chosen at random with probability ϵ.

$$\begin{cases} \text{random action} & \text{probability:}\epsilon \\ \max_{a'} Q(s', a') & \text{otherwise} \end{cases} \tag{6.2.3}$$

The term ϵ decays over time; as a result, at the beginning the agent explores more, and at the end it exploits the knowledge it has gathered.

Update rule

The VQC parameters are trained by minimizing the squared Temporal Difference error, already introduced in section 3.3.1:

$$\mathcal{L}(\theta) = (r + \gamma \max_{a} Q_\theta(s_{t+1}, a) - Q_\theta(s, a))^2 \tag{6.2.4}$$

The gradients of the loss function are computed using the parameter-shift rule and are optimized via ADAM.

Target Network and Experience Replay

As already introduced in section 3.3.1, an *experience replay* system is implemented. Multiple tuples of actions, states and rewards are stored in a replay memory (or buffer), and are randomly sampled to update the network, with the goal of breaking correlation between consecutive samples.
Furthermore, a *target network* is updated periodically to compute the target values. As previously explained in section 3.3.1, having a second network that computes the target value helps stabilize temporal-difference learning.

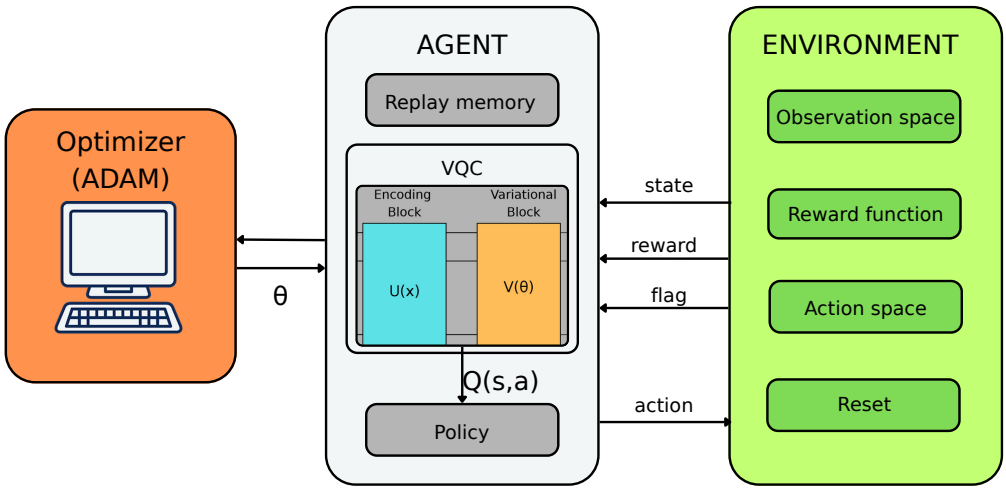

Figure 6.1: Quantum Q-learning architecture

6.2.3 REINFORCE

In REINFORCE, the output of the VQC is the probability of performing an action given the current state and parameters.

Action Selection mechanism

The encoded features are fed into a parametrized quantum circuit, which returns a vector of output logits for each possible action.

$$logists = [l_1, ..., l_n] \tag{6.2.5}$$

Logits are real numbers encoding the relative preference or score for each possible action. Using a softmax function, they are converted into a probability distribution over actions. The probability $\pi_\theta(a \mid s)$ of selecting action a given the state s and parameters θ, is computed using the softmax function:

$$\pi_\theta(a \mid s) = \frac{e^{l_i}}{\sum_{j=1}^{n} e^{l_j}} \tag{6.2.6}$$

Using the probability distribution outputted from the VQC, the agent **chooses an action** sampling from this distribution. This design naturally encourages exploration, without needing an ϵ-greedy policy.

Update rule

During an episode, a trajectory of length $T = 150$ (corresponding to the number of transactions per episode) is collected by storing the log-probabilities of sampled actions and the rewards r_t received at every step. At the end of an episode, the discounted return is computed as follows:

$$G_t = \sum_{k=0}^{T-t} \gamma^k r_{t+k} \tag{6.2.7}$$

Returns are then standardized to reduce variance and stabilize training:

$$G' = \frac{G - \mu(G)}{\sigma(G) + 10^{-8}} \tag{6.2.8}$$

where $\mu(G)$ is the mean value and $\sigma(G)$ is the standard deviation of the discounted return. Standardized returns are used to compute the policy gradient loss, which is backpropagated through the network and used to update parameters with ADAM optimizer.

$$\mathcal{L} = -\sum_t log\pi(a_t|s_t)G'_t \tag{6.2.9}$$

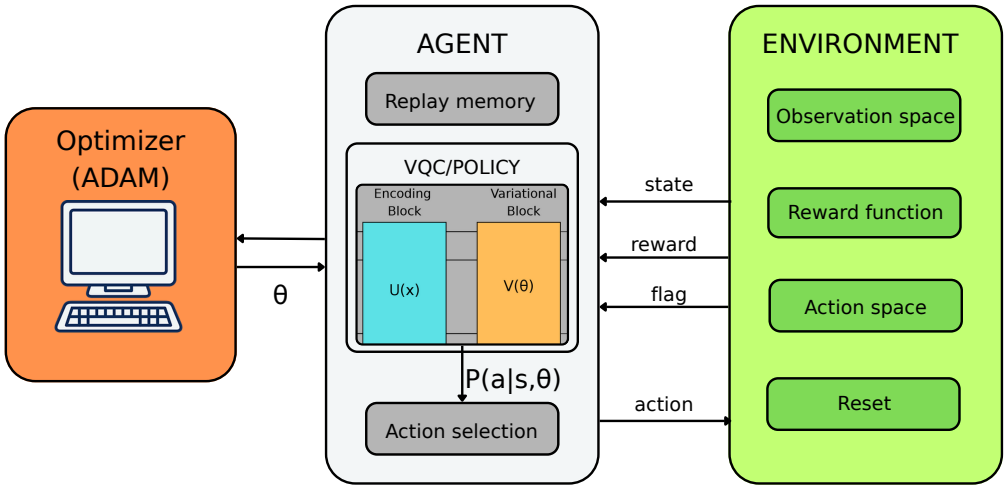

Figure 6.2: Quantum REINFORCE architecture

6.3 Parameters and Hyperparameters

In this section, the impact of the number of transactions per episode is analyzed to determine how it affects performance. Then, the chosen hyperparameter search

method is introduced, along with the best hyperparameter found for both classical and quantum architectures.

6.3.1 Number of transactions per episode

Before performing hyperparameter tuning, a study was conducted to evaluate the effect of varying the number of transactions per episode (TPE). Increasing the number of TPE improves the agent's classification performance.

However, this also involves longer training time for the agent, especially for the Q-learning architecture, which has the slowest training time in both its classical and quantum versions. For this reason, to find the best trade-off between performance and training time, an experiment was conducted on the quantum q-learning architecture. The results are reported in Table 6.2.

Method	Accuracy	Precision	Recall	F1	Reward	Time
100 TPE	0.3500	0.2537	0.5312	0.3434	-59.00	1h40'
150 TPE	0.3333	0.3289	1.000	0.4949	-3.00	2h32'
200 TPE	0.3556	0.3400	0.9789	0.505	2.00	7h16'

Table 6.2: Performance comparison when increasing the number of transactions per episode in the Q-learning architecture

The best results, in terms of F1-score and reward, were achieved with the quantum Q-learning architecture trained for 50 episodes with 200 TPE, achieving an F1-score of 0.505 and a reward of 2.00. However, training this model took more than 7 hours, while using 150 TPE took less than 3 hours, reaching an F1-score of 0.4949 and a reward of -3.00. For this reason, the number of TPE was set to 150 in both classical and quantum models, accepting a lower classification performance but reducing training time to 2.5 hours instead of more than 7.

6.3.2 Hyperparameter TUNING

The hyperparameters of the models influence the learning process and, therefore, their classification performance. A commonly used method for hyperparameter optimization is *Grid Search*, which checks all possible hyperparameter combinations. Since this technique is extremely time-consuming in large search spaces, another common approach is *Random Search*, which involves checking some randomly selected combinations of hyperparameters. However, if the search space grows too much, even Random Search can take a long time to find the optimal hyperparameters. Another hyperparameter search method is *Bayesian Search*, in which it is

build a probabilistic model that estimates the relationship between hyperparameters and their performance. In Bayesian Search, the model is progressively trained and evaluated multiple times, using the results of each trial to guide the selection of future hyperparameter configurations [47].

In this work hyperparameter search was performed using OPTUNA [48], an open-source Python library based on Bayesian optimization that automatically searches for the best hyperparameters and models. OPTUNA has three main components: objective function, trial, and study [49]

- **Objective function:** defines the optimization goal and parameter search ranges.
- **Trial:** training and evaluation process of a specific combination of hyperparameters.
- **Study:** used to specify whether the objective should be maximized or minimized

At the end of each trial, OPTUNA uses the results, in terms of performance metrics, to refine the search in subsequent trials. In this way, it avoids trying random combinations, prunes less performative ones, and finds the best hyperparameters more efficiently.
In this work, the objective metric chosen for maximization was the F1-score. The reason for this choice lies in the dataset's imbalance and the need for a metric to compare all models (XGBoost and RL architectures).
The following subsections describe the best hyperparameters found using OPTUNA for all the models employed in this work.

XGBoost

Table 6.3 shows all the hyperparameters that were included in the objective function for XGBoost, their ranges, and final best values. The tuning process required 10 trials.

The selected hyperparameters show that the model benefits from regularization. For example, "max_depth" defines the maximum depth of a tree, and "min_child_weight" specifies the minimum sum of instance weights needed in a leaf node. The relatively low value of "max_depth" (3) and the higher value of "min_child_weight" (6.8) reduce the risk of over-splitting in sub-trees and therefore the risk of over-fitting by having the model learn too specific patterns. Additionally, "gamma", which controls the minimum loss reduction required for a split, is set to a non-zero value, meaning that only splits that result in a significant loss improvement are accepted. The presence of "alpha" and "lambda", which respectively represent Lasso and Ridge regularization on weights, further reduces the risk

parameter	candidates	value
Nestimators	100-600	447
l.r.	0.01-0.03	0.02589
max depth	3-10	3
min child weight	1-10	6.80573
gamma	0.0-5.0	1.79090
subsample	0.5-1.0	0.61520
colsample	0.5-1.0	0.59294
alpha	0.0-10.0	1.38718
lambda	0.0-10.0	3.65728

Table 6.3: Hyperparameters, search space, and best values for XGBoost

of overfitting. Since the learning rate is low, reducing the impact of parameters updates on each individual tree, a greater number of estimators is required to obtain good performance.

Q-learning

In Table 6.4 are shown the search space and best value for each hyperparameter of classical Q-learning, which was tuned for 10 trials of 200 episodes each. Just like in XGBoost, a low learning rate was chosen. In this way, gradually updating weights reduces the risk of surpassing the optimal solution. The discount factor "gamma" is set to a high value, which means the agent heavily values long-term rewards rather than focusing on the current ones, therefore reducing the risk of stopping at suboptimal solutions. The "epsilon start" and "epsilon end" parameters relate to the exploration/exploitation tradeoff. A high value of "epsilon start" ensures that in early training the model explores widely, while the lower "epsilon end" ensures that towards the end of training the model focuses on exploiting the learned knowledge. The use of a moderated "epsilon decay" makes sure that epsilon does not decrease too rapidly from its start to its end, but at the same time does not slow convergence. Finally, choosing a high batch size (128) allows for more stable gradient estimates and efficient use of computational resources

REINFORCE

The best hyperparameters found by tuning classical REINFORCE with OPTUNA for 10 trials of 500 episodes each are reported in Table 6.5. The number of episodes was increased compared to Q-learning, since REINFORCE needs more episodes to converge, while requiring lower per-episode training time. Just like for Q-learning, a low learning rate was chosen to allow gradual updates of the Neural Network's weights. A high value of the discount factor "gamma" means that the agent will

parameter	candidates	value
l.r.	1e-5 - 1e-3	5.61152e-05
gamma	0.90 - 0.999	0.99412
epsilon start	0.8-1.0	0.94640
epsilon end	0.01-0.2	0.12375
epsilon decay	2000-15000	4028
batch size	32, 64, 128	128

Table 6.4: Hyperparameters, search space, and best values for Q-learning

consider long-term consequences of its actions.

parameter	candidates	value
l.r.	1e-5 - 1e-3	5.61152e-05
gamma	0.90 - 0.999	0.99412

Table 6.5: Hyperparameters, search space, and best values for REINFORCE

Quantum

In QRL, just like classical RL, hyperparameters control how the model learns and how well it generalizes. Common hyperparameters, such as the learning rate and discount factor, were already identified using OPTUNA for the classical models and applied to the quantum architectures as well. Additionally, for the quantum models, OPTUNA was reused to tune quantum hyperparameters, including encoding, gate type, and ansatz type. Quantum Q-learning was tuned for 10 trials of 20 episodes each, while REINFORCE was tuned for 10 trials of 50 episodes each. The results are shown in Table 6.6. For the Q-learning architecture, the optimal setup

parameter	candidates	value q-learning	value REINFORCE
encoding	amplitude, angle, phase, ZZ, ZZ-qiskit	ZZ	amplitude
gate used	X, Y, Z	Z	-
ansatz	strongly, basic	basic	strongly

Table 6.6: Hyperparameters, search space, and best values for VQC

used ZZ encoding with Z gates and a basic ansatz. ZZ encoding entangles qubits to represent classical states, allowing the agent to capture correlations between features. Using the basic ansatz leads to a shallower, less expressive circuit, but one that is easier to train.
For the quantum REINFORCE model, the best hyperparameters found are amplitude encoding with a strongly entangling ansatz. Amplitude encoding efficiently

maps classical states into quantum amplitudes, enabling the agent to represent probability distributions. The strongly entangling ansatz provides high expressivity, allowing the circuit to capture feature correlations.

Part III

Results and Conclusions

Chapter 7

Results

The goal of this chapter is presenting the results of training classical and quantum models on the fraud detection dataset. First, a comparison between XGBoost and classical RL algorithms was conducted to prove the advantages of RL on the fraud detection task. Then, the following sections focus on the performance comparison of classical RL architectures with their quantum counterparts. Finally, the issue of Barren Plateaus is presented, verifying its impact on the models performances.

7.1 Classical Comparison: XGBoost vs Classical Reinforcement Learning

In this section, the performance of classical architectures is compared with XGBoost to demonstrate how RL can achieve better performance than supervised learning models on the fraud detection task.

7.1.1 Comparing XGBoost with Reinforcement Learning

After performing hyperparameter tuning for all classical models, the performance of XGBoost, Q-learning, and REINFORCE was compared in two experiments of different training lengths. Q-Learning and REINFORCE were trained for 50 episodes and compared with XGBoost trained with 50 estimators. The computed metrics were accuracy, precision, recall, and F1-score. Particular attention is paid to the F1-score, since the dataset employed exhibits class imbalance. The results displayed in Table 7.1 show the predominance of Q-learning, especially in terms of F1-score, which reaches a value of 0.4925, while for the two other classifiers, the F1-score is zero. These results demonstrate the ability of Q-learning to achieve high performance with a small number of episodes. At the same time, it also reflects the convergence issues of XGBoost and REINFORCE, when they are not trained extensively.

Method	Accuracy	Precision	Recall	F1
XGBOOST	0.6772	0.000	0.000	0.000
Q-learning	0.3267	0.3266	1.000	0.4925
REINFORCE	0.6733	0.000	0.000	0.000

Table 7.1: Performance comparison of classical architectures (faster training)

In a second experiment, 500 episodes were used for Q-learning and REINFORCE, while almost 450 estimators were used for XGBoost. The goal was to determine whether extending training would improve performance. REINFORCE increases its F1-score to 0.0816 , while XGBoost still maintains an F1 score of 0, as it can be observed in Table 7.2. Q-learning outperforms the other two models with F1-score of 0.8163. These results confirm the superiority of RL-based methods.

Method	Accuracy	Precision	Recall	F1
XGBOOST	0.6766	0.1250	0.0003	0.0006
Q-learning	0.4067	0.3333	0.8163	0.4734
REINFORCE	0.6733	0.5000	0.0816	0.1404

Table 7.2: Performance comparison of classical architectures (extensive training)

7.2 Metrics comparison: classical vs quantum

In this section, a metric comparison is made for Q-learning, REINFORCE, and their quantum counterparts. For both algorithms, multiple runs were performed, increasing the number of episodes to find the best model. Classical metrics such as accuracy, precision, and recall were computed, along with training time and number of parameters. Particular attention is dedicated to the F1-score and reward. As already mentioned the first is particularly useful for assessing classification performance given the dataset's imbalanced nature. The second provides information on the agent's behavior and whether it is learning to make correct decisions.

7.2.1 Q-learning: classical vs quantum

The best model for the quantum Q-learning architecture was found after training for 50 episodes. As shown in Table 7.3, both F1-score and reward increase with respect to the classical version. In fact, the F1-score reaches a value of 0.4462 (instead of 0.4072 of the classical version), and the reward reaches a value of 1.00 (instead of -60.00).

On the other hand, increasing the number of episodes decreases the performance of the quantum architecture relative to the classical architecture. Results are shown

Method	Accuracy	Precision	Recall	F1	Reward	Params	Time
Classic	0.3400	0.2881	0.6939	0.4072	-60.00	5938	25”
Quantum	0.5200	0.3580	0.5918	0.4462	1.00	84	2h30’

Table 7.3: Q-learning: classical vs quantum performance comparison (50 episodes)

in Table 7.4 and 7.5. For instance, in the model trained with 200 episodes, the reward decreases from -6.00 to -15.00 in the quantum version. Even the F1-score is worse, with a value of 0.4098 in the quantum model with respect to a value of 0.4804 in the classical model. Moreover, increasing the number of episodes, leads to longer training time, reaching almost 12 hours with 200 episodes.

Method	Accuracy	Precision	Recall	F1	Reward	Params	Time
Classic	0.3556	0.3353	0.9500	0.4957	0.00	5938	48”
Quantum	0.3278	0.3296	0.9833	0.4937	-7.00	84	4h51’

Table 7.4: Q-learning: classical vs quantum performance comparison (100 episodes)

Method	Accuracy	Precision	Recall	F1	Reward	Params	Time
Classic	0.3800	0.3309	0.8776	0.4804	-6.00	5938	122”
Quantum	0.5200	0.3425	0.5102	0.4098	-15.00	84	11h45’

Table 7.5: Q-learning: classical vs quantum performance comparison (200 episodes)

It can be finally noted that the quantum model uses only 84 trainable parameters instead of the 5938 required by its classical version, resulting in a much smaller architecture.

7.2.2 REINFORCE: classical vs quantum

The best model for the REINFORCE algorithm was found by training it for 500 episodes. With this configuration, it beats its classical counterpart with a significant increase in reward and F1-score, as reported in Table 7.6. In fact, it achieves a reward of 20.00 and an F1-score of 0.5065, compared to a reward of -30.00 and an F1-score of 0.1404 reached by its classical counterpart.

On the other hand, decreasing the number of episodes leads to poorer results, as shown in table 7.7 and 7.8. Not only is the performance worse than than the quantum model trained with 500 episodes, but it is also worse than its classical counterpart. For instance, training the quantum model with only 200 episodes yields an F1-score of 0.4154, compared to 0.4947 for the classical model. The reward also decreases to -28.00, while the classical model is able to reach 1.00

Method	Accuracy	Precision	Recall	F1	Reward	Params	Time
Classic	0.6733	0.5000	0.0816	0.1404	-30.00	5938	32”
Quantum	0.4933	0.4062	0.6724	0.5065	20.00	150	2h34’

Table 7.6: REINFORCE: classical vs quantum performance comparison (500 episodes)

Method	Accuracy	Precision	Recall	F1	Reward	Params	Time
Classic	0.3600	0.3333	0.9592	0.4947	1.00	5938	18”
Quantum	0.4933	0.3750	0.4655	0.4154	-28.00	150	1h16’

Table 7.7: REINFORCE: classical vs quantum performance comparison (200 episodes)

Method	Accuracy	Precision	Recall	F1	Reward	Params	Time
Classic	0.6733	0.000	0.000	0.000	-46.00	5938	7”
Quantum	0.4467	0.2526	0.667	0.3664	-39.00	150	29’

Table 7.8: REINFORCE: classical vs quantum performance comparison (50 episodes)

For the quantum REINFORCE algorithm, the architecture is much smaller, requiring only 150 trainable parameters instead of 5938 needed by its classical version. The reason both classical Q-learning and classical REINFORCE have the same number of parameters is that their architectures are mostly equal to each other, except that in REINFORCE a softmax layer is present to return a probability distribution. This layer does not increase the number of parameters, hence, classical Q-learning and classical REINFORCE have the same number of trainable parameters.

7.3 Performance Comparison: classical vs quantum

In this section, further comparisons between classical and quantum architectures are illustrated. In particular, since randomness factors affect the stability of results, this section analyzes average metrics to improve the evaluation. In particular, for both Q-learning and REINFORCE algorithms in their classical and quantum versions, the models were trained for 10 separate runs. For each run, accuracy, precision, recall, F1-score, and reward were collected to compute their averages and calculate various statistics.
As already mentioned, particular attention is given to the F1-score and reward. For

this reason, the following subsections present graphs illustrating various trends in these two metrics.

7.3.1 Average Reward and F1-score

The plots display the average F1-score and reward across runs as a function of the number of episodes for both Q-learning and REINFORCE. In each plot, the blue line represents the classical model, with its surrounding blue shaded region indicating the standard deviation. Similarly, the orange line corresponds to the quantum model, and the orange shaded area reflects its standard deviation. A wider shaded area implies more variability across episodes and less stable models. For the Q-learning algorithm, Figure 7.1 shows the average F1 score as a function of the number of training episodes. The classical model, despite starting with lower values and greater variability, achieves better performance than the quantum architecture. The quantum model performs better only in the first 10 episodes, suggesting a tendency for the quantum Q-learning architecture of improving faster at the beginning of training.

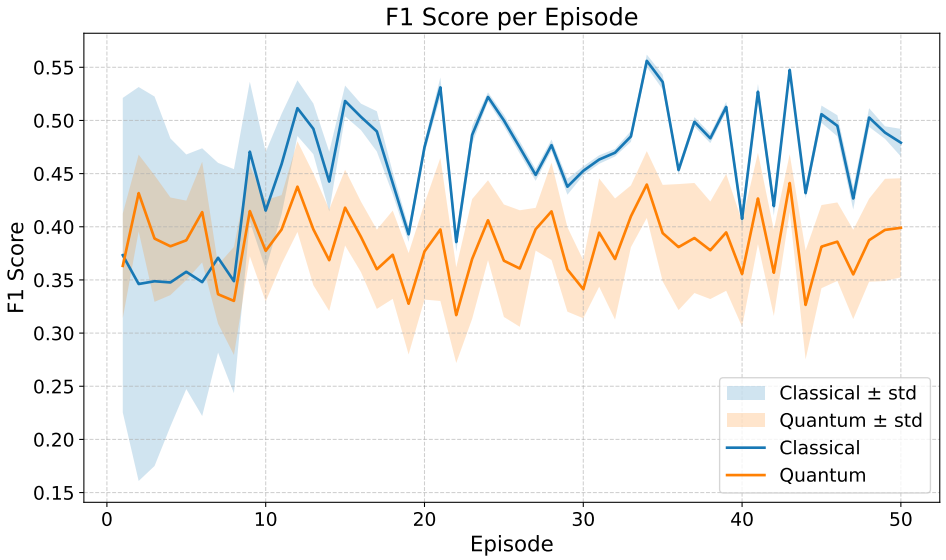

Figure 7.1: Average F1-score - Q-learning

The average reward for the Q-learning architecture displayed in Figure 7.2, shows a more comparable trend for the two models. Performance is similar, as both models achieve nearly the same average value, but the quantum model is more unstable, as shown by a greater standard deviation.

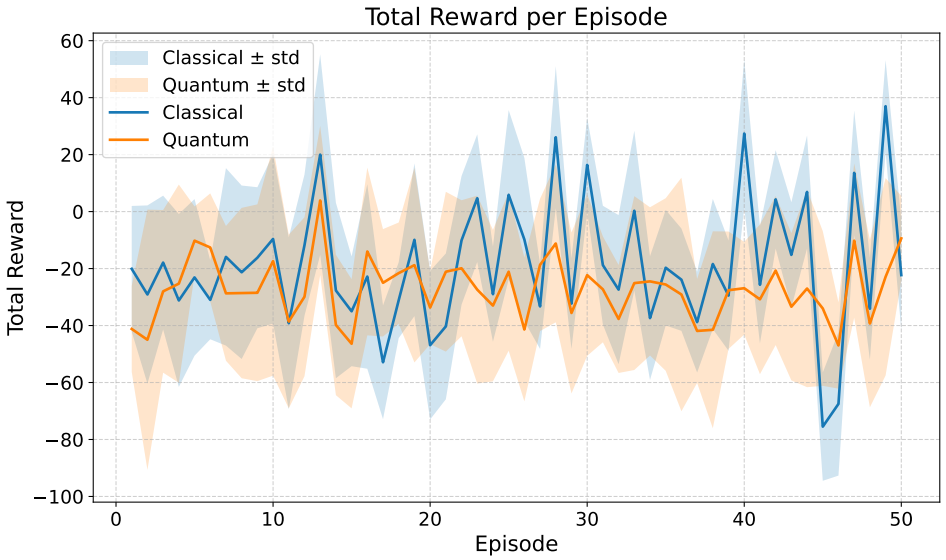

Figure 7.2: Average Reward - Q-learning

The quantum REINFORCE algorithm has a better performance in terms of F1-score, as shown in picture 7.3. In fact, the average F1-score reaches higher values than its classical counterpart. Moreover, the classical algorithm shows greater instability, as indicated by its higher standard deviation.

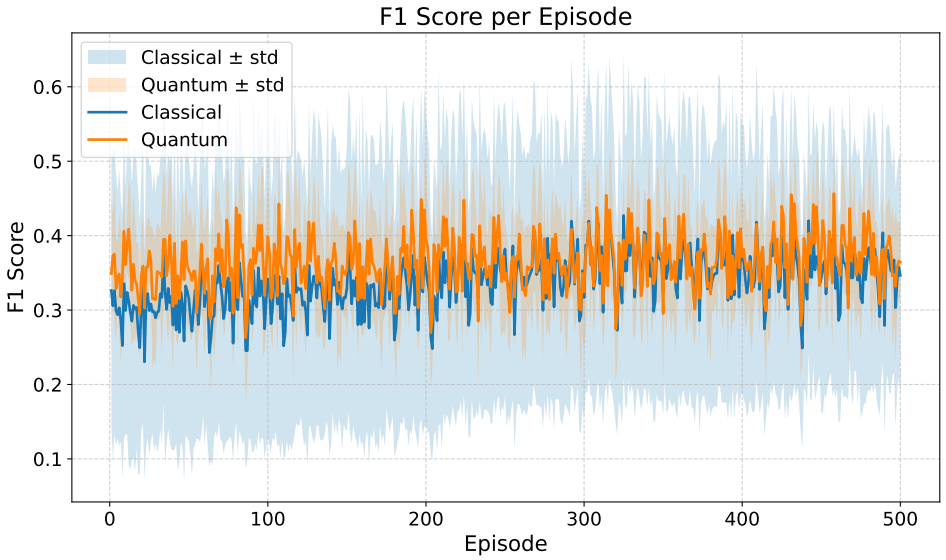

Figure 7.3: Average F1-score - REINFORCE

The performance in terms of reward is comparable between quantum REINFORCE and quantum Q-learning. Aa shown in Figure 7.4, both models reach the same average reward, with a stable performance.

Figure 7.4: Average Reward - REINFORCE

7.3.2 Cumulative Moving Average

The second statistical metric employed is the Cumulative Moving Average (CMA). It is typically used in settings where data arrives in an ordered stream and the average is computed every time a new sample arrives [50]. Considering the current time t, when a new data point x_t arrives, the current CMA is:

$$CMA(t) = \frac{x_0 + ... + x_t}{t + 1} \tag{7.3.1}$$

Computing the F1-score and Reward for each episode, showing the averaged metrics can result in noisy plots. The CMA can progressively smooth the curve, making long-term trends easier to interpret. The following line graphs show the CMA for the classical architectures (blue line) and quantum architectures (orange line), with the shaded region around each line representing the standard deviation around the average F1-score or Reward.
In picture 7.5, the CMA for the F1 score of the Q-learning algorithm is displayed. The image shows that the classical model achieves better final performance, but the shaded region representing the standard deviation is wider, indicating less stability, especially in the first episodes.

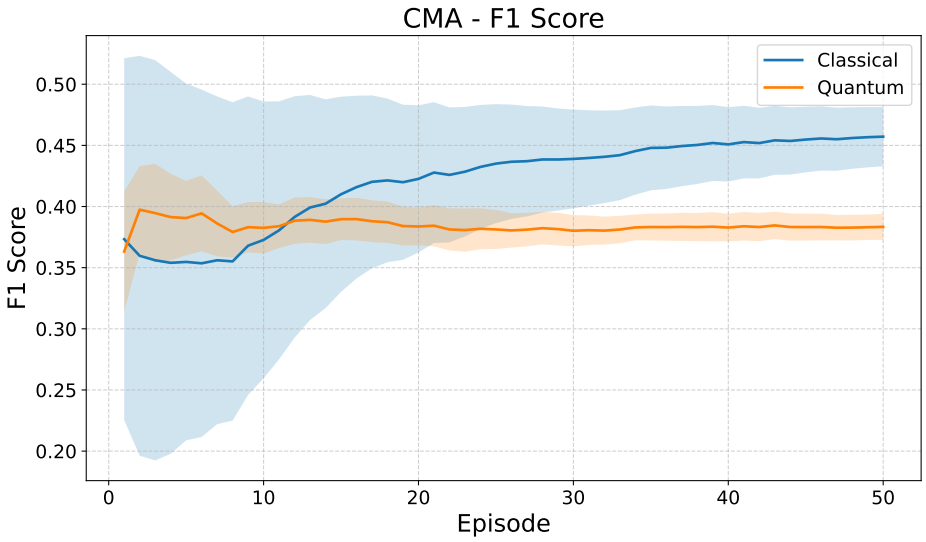

Figure 7.5: Cumulative Moving Average for the F1-score - Q-learning

Looking at the CMA for the reward, shown in picture 7.6, it can be observed that the classical model reaches a better performance. In this case, both models exhibit a high standard deviation in the first few episodes, which decreases as the episodes progress, indicating that both become more stable.

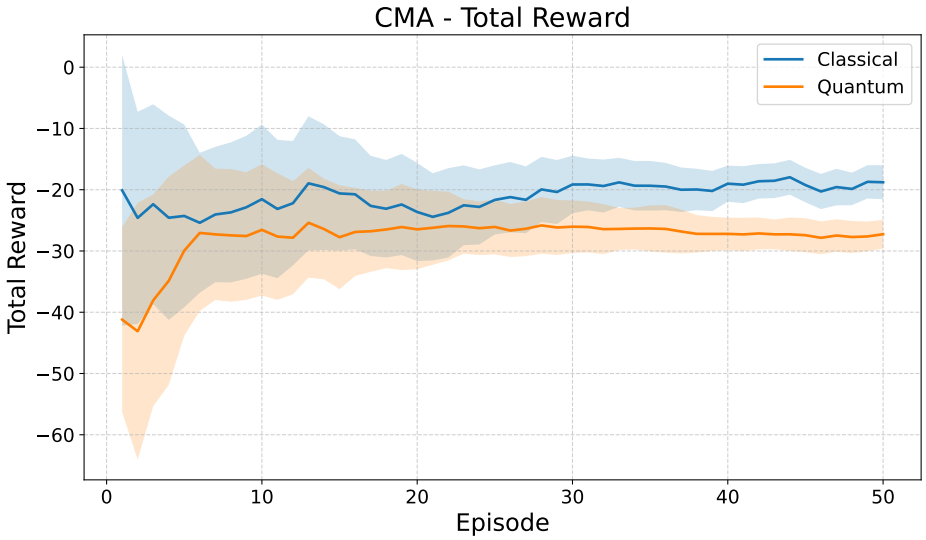

Figure 7.6: Cumulative Moving Average for the Reward - Q-learning

In the REINFORCE algorithm, the CMA for the F1-score represented in picture 7.7 shows that the quantum model outperforms its classical counterpart. It can be

observed that it reaches a higher CMA value and is more stable than the classical model, whose wide, shaded blue region indicates high variability.

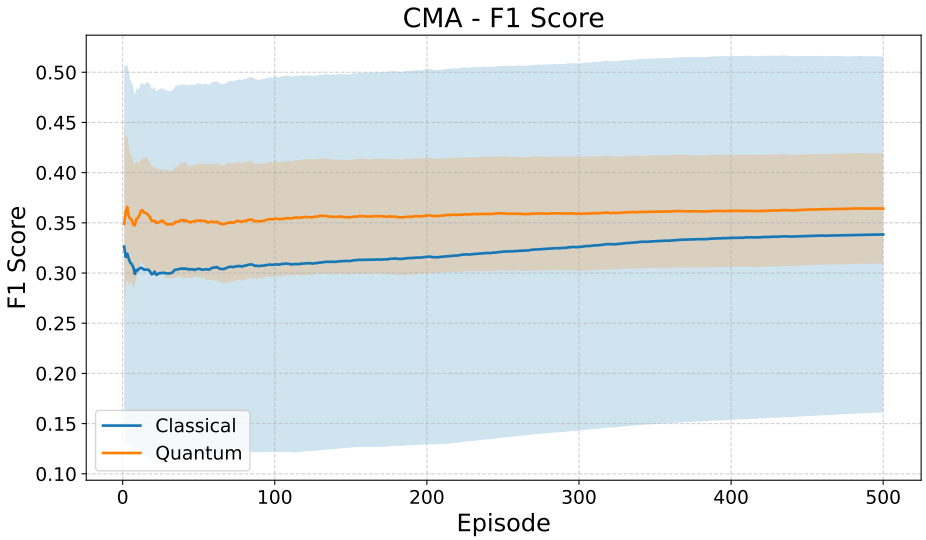

Figure 7.7: Cumulative Moving Average for the F1-score - REINFORCE

In picture 7.8, the CMA for the reward shows fluctuation and instability for the classical model. However, it is able to reach a slightly better performance than its quantum version.

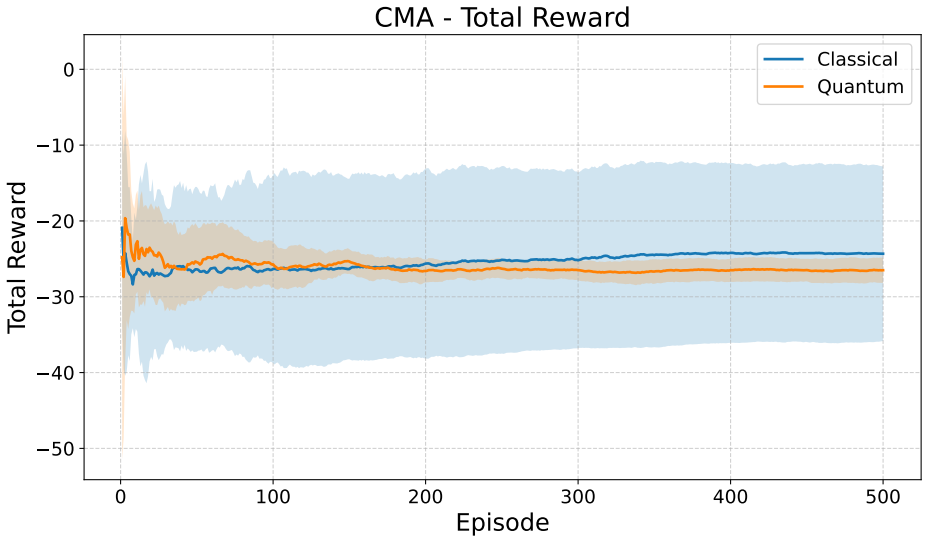

Figure 7.8: Cumulative Moving Average for the Reward - REINFORCE

7.3.3 Cumulative Probability distribution

Finally, the last statistic analyzed is the Cumulative Probability Distribution (CPD) of the chosen metric. As in the previous sections, the metrics considered are the average F1 score and the average total Reward across runs, shown with respect to the number of episodes. The CPD of a random variable $X \in \mathbb{R}$ is the function given in the following equation:

$$F_x(x) = P(X \leq x) \tag{7.3.2}$$

and it represents the probability that the random variable takes a value less than or equal to x. In this case, it is considered the probability that the average F1-score and average Reward are less than 90% of their maximal value, with respect to the episodes. If, within a certain number of episodes, the slope of the function for a model is steeper, it means that the probability of reaching the maximum value is higher for that model. Therefore, this metric can be interpreted as the speed at which a model learns. In the following plots, the classical model's CPD is blue, while the quantum model's CPD is orange.

Image 7.9 shows the CPD for the F1-score of the Q-learning algorithm. The quantum model has a steeper slope than the classical architecture in the first 5 episodes. However, the classical model reaches its maximum within the first 20 episodes, whereas the Quantum model takes more than 30 episodes to reach its maximum.

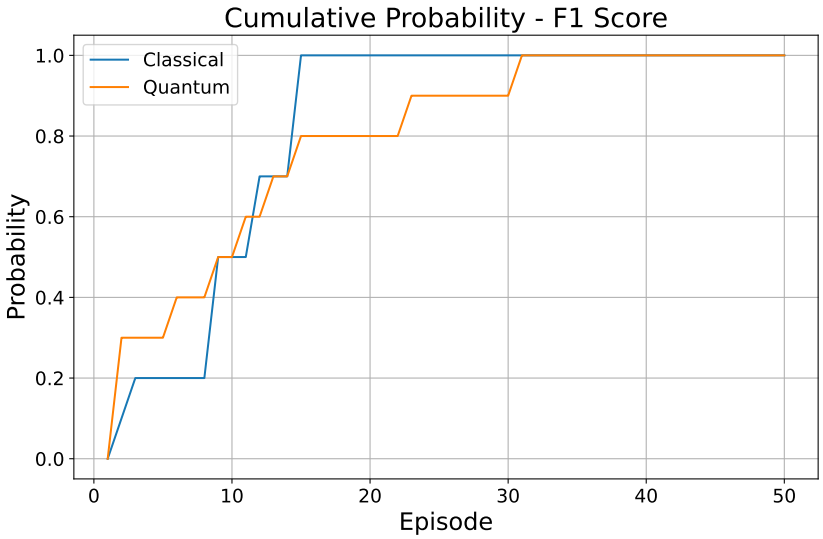

Figure 7.9: Cumulative Probability Distribution for the F1-score - Q-learning

The CPD for the Reward shown in Image 7.10, shows a slightly better trend for the quantum model, since the probability mostly grows faster than its classical version.

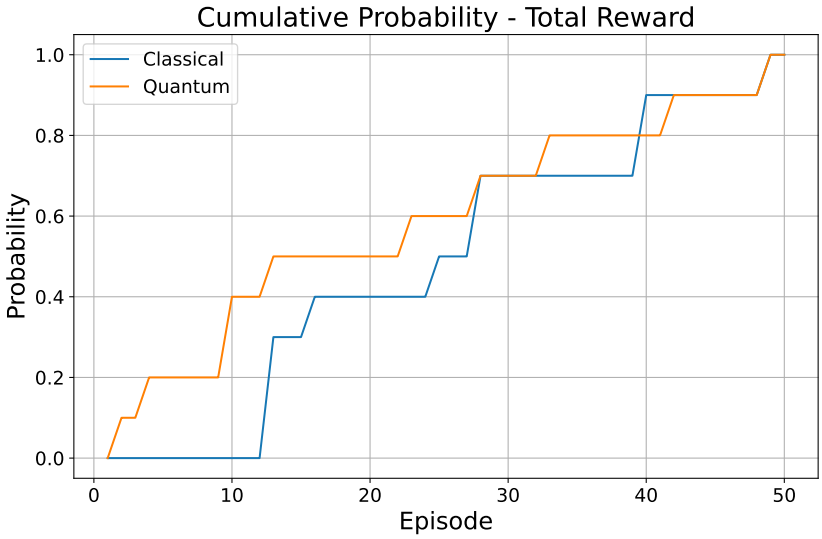

Figure 7.10: Cumulative Probability Distribution for the Reward - Q-learning

Image 7.11 shows the CPD for the F1-score of the REINFORCE model. The quantum model performs better, since the slope is extremely steep at the beginning and the maximum value is reached within the first 100 episodes, while it takes more than 400 episodes for the classical architecture to reach the maximum.

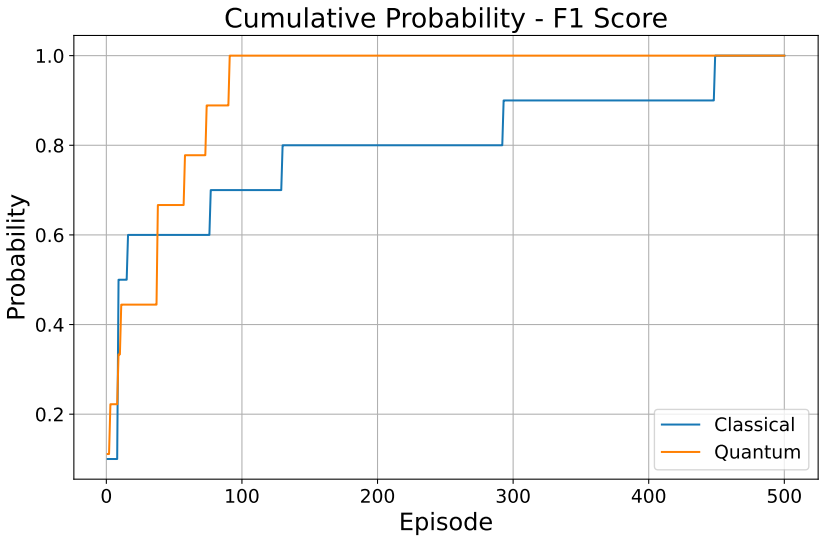

Figure 7.11: Cumulative Probability Distribution for the F1-score - REINFORCE

On the other hand, the trend for the CPD of the Reward, is for the most comparable, as it can be observed in Image 7.12.

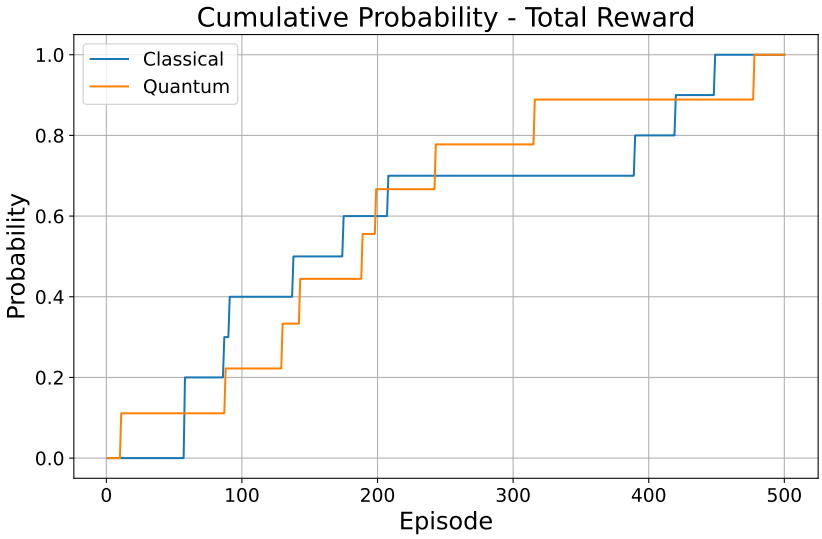

Figure 7.12: Cumulative Probability Distribution for the Reward - REINFORCE

7.4 Fisher Information Matrix

Quantum architectures are known to suffer from barren plateaus, where the gradient vanishes exponentially with the model's size. In this section, the behavior of the implemented algorithms during optimization is analyzed to understand whether barren plateaus affect their performance. In particular, the Fisher Information Matrix is used to study the responsiveness of the models to parameter updates.

7.4.1 The issue of Barren Plateaus

The term "barren plateaus" describes the phenomenon of vanishing gradients in VQCs and, more generally, in QNNs. This means that the gradients used to adjust the circuit's parameters during training approach zero as the circuit size increases. The optimizer loses information about the direction in which it should update the parameters, and the model cannot improve. In particular, it has been demonstrated that, given a VQC with cost function C and N qubits, the variance of the gradient of the cost function gets exponentially smaller as the number of qubits increases:

$$VAR[\partial C] \leq F(N) \tag{7.4.1}$$

where $F(N)$ is a quantity that decreases exponentially as N increases [51].

7.4.2 Fisher Information and Barren Plateaus

The Fisher Information can be used in order in order to study the effect of barren plateaus on the loss function [52].
In order to define the concept of Fisher Information, the definition of statistical model must be first given: A *statistical model* $f(x_i|\theta)$ for a random variable X, is a function that represents how a parameter θ is related to potential outcomes x_i. When θ is fixed, the statistical model becomes a function $p_\theta(x_i) = (x_i|\theta)$ that describes all the potential outcomes for that random variable. This function is known as:

- *probability density function* when X is continuous
- *probability mass function* when X is discrete

Often, the parameter's value is unknown, and the goal of statistical inference is to estimate it as accurately as possible from the observed data. To quantify how much information the data provide about the unknown parameter, the concept of Fisher Information can be introduced [53].
The Fisher Information $I_x(\theta)$ of a random variable X about a parameter θ is defined as:

$$I_X(\theta) = \begin{cases} \sum_{x\in X} \left(\frac{d}{d\theta} \log f(x \mid \theta)\right)^2 p_\theta(x), & \text{if } X \text{ is discrete,} \\ \int_X \left(\frac{d}{d\theta} \log f(x \mid \theta)\right)^2 p_\theta(x)\, dx, & \text{if } X \text{ is continuous.} \end{cases} \tag{7.4.2}$$

For a vector of parameters $\vec{\theta} = (\theta_1, ..., \theta_d)$, the Fisher Information Matrix (FIM) is a Variance-Covariance matrix where each entry i, j is given by:

$$\begin{cases} \sum_{x\in X} \left(\frac{\partial}{\partial\theta_i} \ell(x \mid \vec{\theta})\right) \left(\frac{\partial}{\partial\theta_j} \ell(x \mid \vec{\theta})\right) p_{\vec{\theta}}(x), & \text{if } X \text{ is discrete,} \\ \int_X \left(\frac{\partial}{\partial\theta_i} \ell(x \mid \vec{\theta})\right) \left(\frac{\partial}{\partial\theta_j} \ell(x \mid \vec{\theta})\right) p_{\vec{\theta}}(x)\, dx, & \text{if } X \text{ is continuous.} \end{cases} \tag{7.4.3}$$

where $l(x|\vec{\theta}) = logf(x|\vec{\theta})$ is the log-likelihood function and $\frac{\partial}{\partial\theta_i} l(x|\vec{\theta})$ is the partial derivative with respect to the ith component of the vector $\vec{\theta}$.
The FIM can be written in a more compact way as:

$$F(\theta) = \mathbb{E}_{x\sim p_\theta}\left[(\nabla_\theta \log p_\theta(x)) (\nabla_\theta \log p_\theta(x))^\top\right] \tag{7.4.4}$$

In statistics, the likelihood function (or, in this case, the log-likelihood function) evaluates the relative support that the observed data provide for each possible value of the parameter vector. The gradient of the log-likelihood measures the sensitivity of the log-likelihood to small changes in $\vec{\theta}$. Therefore, the Fisher Information provides insights into how strongly changes in parameters affect the likelihood. A

small value for a FIM entry indicates that the gradients are very small, corresponding to flat regions of the log-likelihood function where the model does not improve. Inspecting the FIM reveals information about the flatness of the loss function, and therefore the presence of vanishing gradients and barren plateaus in the case of VQCs.

7.4.3 Relevance for Reinforcement Learning

In practice, the FIM is estimated empirically using samples obtained during training:

$$\hat{F}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(\nabla_\theta \log p_\theta(x_i)\right) \left(\nabla_\theta \log p_\theta(x_i)\right)^\top \tag{7.4.5}$$

The **trace** of the FIM, denoted as $\mathrm{Tr}(\hat{F}(\theta))$, measures the total amount of information captured across all parameters:

$$\mathrm{Tr}(\hat{F}(\theta)) = \sum_{j=1}^{d} \lambda_j \tag{7.4.6}$$

where λ_j are the eigenvalues of $\hat{F}(\theta)$. A larger trace typically indicates greater sensitivity of the model parameters to the training data, implying a more informative optimization landscape.
Comparing the normalized Fisher traces between Q-learning and REINFORCE allows the analysis of the relative smoothness of their optimization functions and their robustness to parameter perturbations.
For both algorithms, the log-probability of the selected action was computed at each step. Then, for each log-probability, the gradient with respect to the model's parameters was calculated. The gradients across episodes were stacked in order to compute the FIM, and its trace was computed. Since in Q-learning there is no probability distribution that allows the use of Equation 7.4.4, the Fisher Information Matrix is approximated using the squared gradient norm of the Q-values , as a proxy for how "sensitive" the Q-function is to parameter changes.
The trace values were normalized to fall in the range [0-1] to ensure comparable results. Since adding gradient computation increases training time per episode, both algorithms were only trained for 25 episodes. Moreover, the choice was justified by the observation that larger gradient variations occur in the first 20 episodes.
The Trace of FIM was firstly computed for classical Q-learning and classical REINFORCE, as reported in pictures 7.15 and 7.16. The Q-learning model shows signs of information loss early in training; the trace collapses to stable, low values, indicating a general lack of responsiveness to parameter updates. On the other hand, the trace for the REINFORCE model is highly oscillatory but consistently reaches high peaks, showing that the model retains more information during training.

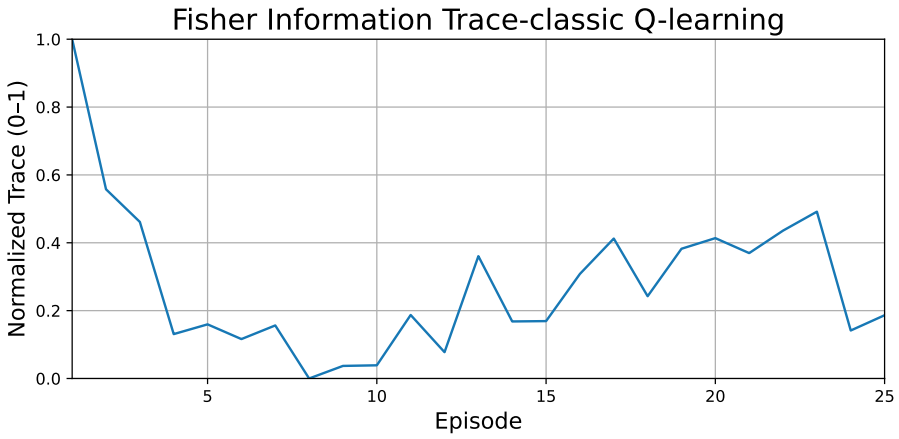

Figure 7.13: Trace across episodes - classic Q-Learning

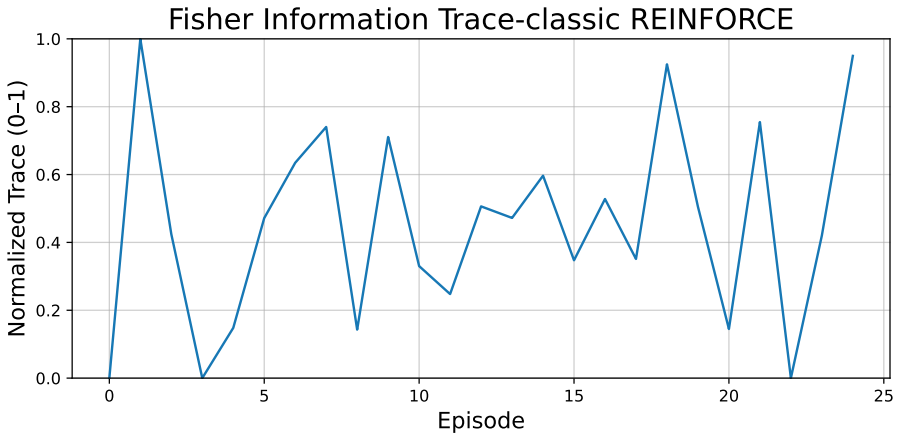

Figure 7.14: Trace across episodes - classic REINFORCE

In quantum architectures, the behavior for Q-learning is the opposite. The trace initially remains low and then stabilizes at higher values, showing that as training progresses, the model becomes more responsive to updates. The REINFORCE architecture also exhibits oscillatory behavior in its quantum version, but it consistently reaches lower values than the quantum Q-learning architecture. This is related to the emergence of barren plateaus, meaning that gradient updates in the quantum REINFORCE architecture become increasingly uninformative. In contrast, Q-learning is less susceptible to this phenomenon, showing greater responsiveness to changes in the data. The ability of the quantum Q-learning architecture to rapidly adapt to new patterns, aligns with the observation made in the previous sections. In fact, when comparing the performance of classical models with their quantum counterparts across multiple metrics, quantum Q-learning showed the ability of reaching high metric values within only a few episodes.

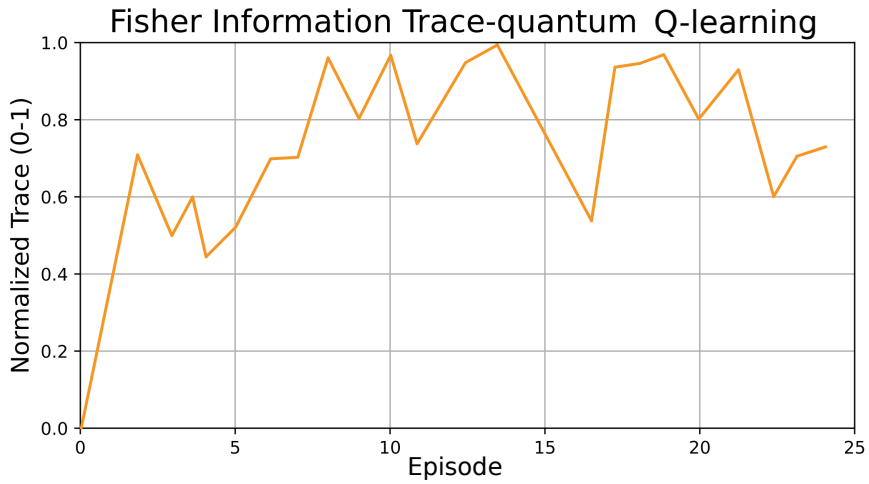

Figure 7.15: Trace across episodes - quantum Q-Learning

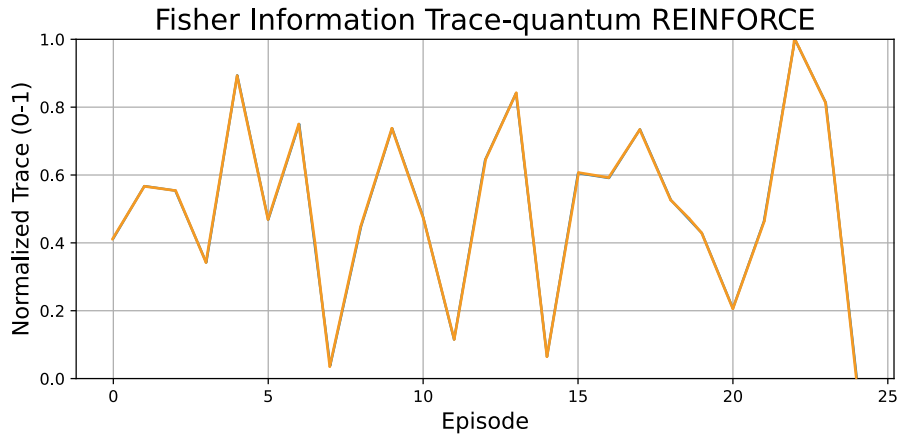

Figure 7.16: Trace across episodes - quantum REINFORCE

Across both classical and quantum settings, the FIM trace reveals consistent behavior associated with the functioning of the two RL algorithms. The Q-learning architecture tends to converge to stable values, while the REINFORCE architecture tends to exhibit highly oscillatory behavior. This difference comes from the way each algorithm computes gradient updates. Q-learning updates its parameters based on the current state, action, and reward, resulting in predictable changes. REINFORCE computes parameter updates at the end of an episode, using the entire trajectory of states, actions, and rewards. Since each element of the trajectory contributes to the gradient, the resulting updates are highly variable, leading to fluctuations in the FIM trace.

Chapter 8

Conclusions

The goal of this thesis was to merge the sequential decision-making capabilities of RL with the efficiency of Quantum Machine Learning techniques to improve performance on the fraud detection task. Specifically, two RL algorithms were implemented, Q-learning and REINFORCE, and their quantum counterparts were constructed by replacing the Neural Network with a Variational Quantum Circuit. All the architectures, both classical and quantum, were trained and tested on the Fraud Detection Dataset. To ensure that all algorithms achieve optimal performance, an automated hyperparameter search was conducted using OPTUNA. The search space included not only classical hyperparameters but also quantum-specific ones, such as the encoding type. Additionally, a study was conducted in order to determine the optimal number of transactions per episode to balance performance and training time. Finally, the Fisher Information Matrix trace was analyzed to assess the model's response to parameter changes.
The superiority of classical RL on the fraud detection task is demonstrated by benchmarking Q-learning and REINFORCE against XGBoost. Moreover, the results display the ability of quantum Q-learning and quantum REINFORCE to outperform their classical counterparts. Further insights can be drawn by considering the variability in model performance across different runs.

REINFORCE shows better overall performance than both its classical version and Q-learning. In fact, it proved to be more stable and achieved higher values of the evaluation metrics. On the other hand, Q-learning exhibited greater oscillations during training and, despite occasionally outperforming the classical model, generally showed lower results. Moreover, it also required much longer training time than REINFORCE.

Nonetheless, Q-learning can achieve relatively high results results in fewer episodes while maintaining high stability. This is true for both its classical counterpart and

quantum REINFORCE. It is attributed to Q-learning's ability to update its parameters after every transaction, therefore adapting more quickly to changes. Although REINFORCE achieves better performance than Q-learning and consistently beats its classical version, Q-learning could be more suitable for the fraud detection task thanks to this peculiarity.

Future expansions of this work should analyze the performance of the implemented architecture on a real dataset, instead of a synthetic one. This could help determine whether the superior performance of REINFORCE would still hold under higher class imbalance and more dynamic data patterns.

This thesis focused on value-based RL with Q-learning and on policy-based RL with REINFORCE. Another paradigm that could be studied in both its classical and quantum versions is the Actor-Critic approach, which combines the frequent updates of value-based methods with the direct policy updates of REINFORCE.

Finally, future studies should examine how the model's performances would benefit by using more powerful hardware to run the experiments proposed in this thesis. In particular, future work could extend training time, perform a broader parameter search, and conduct a more deep analysis of the FIM

In conclusion, this work demonstrates the potential of integrating quantum techniques into reinforcement learning for fraud detection, contributing to a rapidly-evolving field with new methods and applications.

Bibliography

[1] Amarnath Immadisetty. Real-time fraud detection using streaming data in financial transactions. *Journal of Recent Trends in Computer Science and Engineering (JRTCSE)*, 13(1):66–76, 2025.

[2] eba.europa.eu. `https://www.eba.europa.eu/sites/default/files/2024-08/465e3044-4773-4e9d-8ca8-b1cd031295fc/EBA_ECB%202024%20Report%20on%20Payment%20Fraud.pdf`. [Accessed 06-11-2025].

[3] Palak Gupta, Anmol Varshney, Mohammad Rafeek Khan, Rafeeq Ahmed, Mohammed Shuaib, and Shadab Alam. Unbalanced credit card fraud detection data: A machine learning-oriented comparative study of balancing techniques. *Procedia Computer Science*, 218:2575–2584, 2023. International Conference on Machine Learning and Data Engineering.

[4] Itunuoluwa Adegbola. Reinforcement learning for optimizing real-time fraud intervention strategies in digital banking systems. 2025.

[5] Sofia Patel, Isabella Cruz, Priya Singh, Aria Martinez, and Noah Kim. Leveraging reinforcement learning for real-time fraud detection in financial transactions, 01 2025.

[6] Yuxuan Du, Xinbiao Wang, Naixu Guo, Zhan Yu, Yang Qian, Kaining Zhang, Min-Hsiu Hsieh, Patrick Rebentrost, and Dacheng Tao. Quantum machine learning: A hands-on tutorial for machine learning practitioners and researchers, 2025.

[7] Gordon E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86:82–85, 1998.

[8] Fulai Zhu, Peiyu Xu, and Jiahao Zong. Moore's law: The potential, limits, and breakthroughs. *Applied and Computational Engineering*, 10:307–315, 09 2023.

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

[10] Kishor Bharti, Alba Cervera-Lierta, Thi Ha Kyaw, Tobias Haug, Sumner Alperin-Lea, Abhinav Anand, Matthias Degroote, Hermanni Heimonen, Jakob S. Kottmann, Tim Menke, Wai-Keong Mok, Sukin Sim, Leong-Chuan Kwek, and Alán Aspuru-Guzik. Noisy intermediate-scale quantum algorithms. *Reviews of Modern Physics*, 94(1), February 2022.

[11] Sonia Rani, Ravinder Kaur, Chitra Desai, and R P Ambilwade. Quantum machine learning: Leveraging quantum computing for enhanced learning algorithms. 6:1–15, 09 2024.

[12] Stefano Mangini. Variational quantum algorithms for machine learning: theory and applications, 2023.

[13] Nico Meyer, Christian Ufrecht, Maniraman Periyasamy, Daniel D. Scherer, Axel Plinge, and Christopher Mutschler. A survey on quantum reinforcement learning, 2024.

[14] Lov K. Grover. A fast quantum mechanical algorithm for database search, 1996.

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

[16] Elías F. Combarro, Alberto Di Meglio, and Samuel González-Castillo. *A Practical Guide to Quantum Machine Learning and Quantum Optimization*. Sciendo, 2023.

[17] David J. Griffiths. *Introduction to Quantum Mechanics*. Cambridge University Press, 3rd edition, 2018.

[18] Shubhayan Ghosal. Quantum reinforcement learning in non-abelian environments: Unveiling novel formulations and quantum advantage exploration, 2024.

[19] Jian Wang, Ding Zhong, Liangzhu Mu, and Heng Fan. Fidelity of measurement-based quantum computation in a bosonic environment. *Physical Review A*, 90(5), November 2014.

[20] Tom Mitchell. *Machine learning*. McGraw-hill New York, 1997.

[21] XGBoost Contributors. *XGBoost Documentation*, n.d. Accessed: 2025-09-20.

[22] Ian T Jolliffe. *Principal Component Analysis*. Springer Series in Statistics. Springer, New York, NY, 2 edition, October 2002.

[23] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[24] Kevin Murphy. Reinforcement learning: An overview, 2025.

[25] Samuel Yen-Chi Chen. An introduction to quantum reinforcement learning (qrl), 2024.

[26] Fadi AlMahamid and Katarina Grolinger. Reinforcement learning algorithms: An overview and classification. In *2021 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, page 1–7. IEEE, September 2021.

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Kirkeby Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[28] Longxin Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

[29] Matthias Lehmann. The definitive guide to policy gradients in deep reinforcement learning: Theory, algorithms and implementations, 2024.

[30] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 2004.

[31] Yuxi Li. Deep reinforcement learning, 2018.

[32] Maria Schuld and Francesco Petruccione. Machine learning with quantum computers. *Quantum Science and Technology*, 2021.

[33] Esma Aimeur, Djabeur Zekrifa, and Sébastien Gambs. Machine learning in a quantum world. pages 431–442, 05 2012.

[34] Patrick Rebentrost, Masoud Mohseni, and Seth Lloyd. Quantum support vector machine for big data classification. *Physical Review Letters*, 113(13), September 2014.

[35] Nicola Assolini, Luca Marzari, Isabella Mastroeni, and Alessandra di Pierro. Formal verification of variational quantum circuits, 2025.

[36] Samuel Yen-Chi Chen, Chao-Han Huck Yang, Jun Qi, Pin-Yu Chen, Xiaoli Ma, and Hsi-Sheng Goan. Variational quantum circuits for deep reinforcement learning, 2020.

[37] Owen Lockwood and Mei Si. Reinforcement learning with quantum variational circuit. *Proc. AAAI Artif. Intell. Interact. Digit. Enterain. Conf.*, 16(1):245–251, October 2020.

[38] Owen Lockwood and Mei Si. Playing atari with hybrid quantum-classical reinforcement learning. In Luca Bertinetto, João F. Henriques, Samuel Albanie, Michela Paganini, and Gül Varol, editors, *NeurIPS 2020 Workshop on Pre-registration in Machine Learning*, volume 148 of *Proceedings of Machine Learning Research*, pages 285–301. PMLR, 11 Dec 2021.

[39] Sofiene Jerbi, Casper Gyurik, Simon C Marshall, Hans J Briegel, and Vedran Dunjko. Parametrized quantum policies for reinforcement learning, 2021.

[40] André Sequeira, Luis Paulo Santos, and Luis Soares Barbosa. Policy gradients using variational quantum circuits. *Quantum Mach. Intell.*, 5(1), June 2023.

[41] Samay Deepak Ashar. Fraud detection transactions dataset, 2025.

[42] Ahmad Alshammari. Implementation of feature selection using correlation matrix in python. *International Journal of Computer Applications*, 186:29–34, 12 2024.

[43] Skewness - Measures and Interpretation - GeeksforGeeks — geeksforgeeks.org. `https://www.geeksforgeeks.org/data-science/skewness-measures-and-interpretation/`. [Accessed 26-11-2025].

[44] Mohammed Z Al-Faiz, Ali A Ibrahim, and Sarmad M Hadi. The effect of Z-Score standardization (normalization) on binary input due the speed of learning in back-propagation neural network. *Iraqi Journal of ICT*, 1(3):42–48, February 2019.

[45] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[46] Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2025.

[47] Shashank Shekhar, Adesh Bansode, and Asif Salim. A comparative study of hyper-parameter optimization tools, 2022.

[48] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[49] Yuxin Chen, Mohammad Hossein Kadkhodaei, and Jian Zhou. Development of the optuna-ngboost-shap model for estimating ground settlement during tunnel excavation. *Underground Space*, 24:60–78, 2025.

[50] Chhavi Bajpai. Comparative analysis of simple moving average and cumulative moving average in financial time series forecasting. 03 2024.

[51] Jarrod R McClean, Sergio Boixo, Vadim N Smelyanskiy, Ryan Babbush, and Hartmut Neven. Barren plateaus in quantum neural network training landscapes. *Nat. Commun.*, 9(1):4812, November 2018.

[52] André Sequeira, Luis Paulo Santos, and Luís Soares Barbosa. Policy gradients using variational quantum circuits, 2023.

[53] Alexander Ly, Maarten Marsman, Josine Verhagen, Raoul Grasman, and Eric-Jan Wagenmakers. A tutorial on fisher information, 2017.