Università degli Studi di Milano Bicocca

Physics Department

Bachelor's Degree

Comparative analysis of classical Deep Q-Networks and Parametrized Quantum Circuits for Reinforcement Learning Tasks

Supervisor: [illegible]

Prof. Andrea Giachero

Co-Supervisor:

Dr. Danilo Labranca

Candidate:

Francesco Farabegoli

ACADEMIC YEAR 2023/2024

A te, papà,

che mi hai insegnato ad osservare il mondo

con occhi pieni di cuoriosità e meraviglia.

Contents

Abstract

Learning from experience is essential for solving complex challenges: it enables us to recognize patterns, adapt to new situations, and make informed decisions, guiding our actions in both immediate and long-term contexts. This thesis work investigates Reinforcement Learning (RL), a machine learning method where an agent learns by interacting with an environment, using a trial-and-error approach to optimize its actions over time through a reward-based system.

In addition to exploring classical RL, this research examines how quantum computing might enhance RL. Unlike classical computers that represent data in binary bits (0 or 1), quantum computers use qubits, which can exist in a superposition of both 0 and 1 simultaneously. Quantum systems also leverage entanglement, a phenomenon allowing qubits to correlate deeply. These unique properties offer the potential for vastly increased computational capacity compared to classical systems. To evaluate whether quantum machine learning can outperform classical techniques, I implemented an RL algorithm that replaces the traditional neural network with a Parameterized Quantum Circuit (PQC), simulated on a classical computer. This hybrid approach combines classical and quantum processing, leveraging their respective strengths to work within the constraints of current quantum technology. PQC models use a combination of fixed gates (such as controlled-NOT gates) and adjustable gates (such as rotation gates) to encode and optimize complex problems. In this hybrid setup, intensive computational tasks are distributed between classical and quantum processors. For this study, all computations, including quantum simulations, were run on a classical computer: this choice enables to test the quantum algorithm on ideal and foolproof simulated qubits. While actual quantum devices from providers like IBM and Google are available, this study relied on simulation to bypass the limitations of small-scale quantum systems. Current quantum hardware introduces significant noise and operational limitations, including gate and readout errors, decoherence, and limited qubit connectivity, which impact the fidelity of quantum algorithms.

Experiments were conducted in Python using TensorFlow and TensorFlow Quantum libraries, and the Deep Q-Learning algorithm was implemented to estimate state-action values (Q-values) and learn the optimal policy. Both classical and quantum versions of the algorithm were tested

within the Acrobot-v1 environment from OpenAI Gym, a widely recognized benchmark for measuring RL performance in terms of efficiency and accuracy. In this environment, the agent controls a two-link robotic arm connected by a joint and aims to apply torque (positive, negative, or zero) to make the arm swing up and reach a target height. The reward system assigns a -1 penalty for each action that fails to reach the target, while each episode ends either when the arm swings above the target (earning a reward of 0) or after 500 time steps without achieving the goal. The primary goal of this research was to assess whether integrating quantum computing could enhance RL's efficiency and effectiveness compared to classical approaches. To allow this comparison, three parameters were varied: (1) batch size (the number of episodes sampled during training); (2) model expressivity (measured by the number of neurons in hidden layers for the classical model and the number of variational layers in the quantum model); and (3) the learning rate, which controls the speed at which parameters and Q-values are updated.

The results show that the hybrid quantum approach initially learns faster, achieving its peak performance with a reward of approximately -140 within 600 episodes. However, its training was limited by the heavy computational demands of simulating quantum circuits. In comparison, the classical model, trained over 20,000 episodes, converged to a more optimal reward of approximately -70. Over extended training, the classical approach consistently outperformed the quantum model, proving more efficient and producing a more stable policy. Traditional methods remain more effective for practical applications, though quantum computing holds promising future potential. The next phase of the research involves implementing and testing the algorithm on real quantum hardware, using IBM's quantum backend. Testing a quantum reinforcement learning (QRL) algorithm on actual quantum hardware could reduce training time and enable testing over a higher number of episodes and would also let us see how well it performs in real-world conditions. This experiment would help us understand how hardware limitations, like noise and gate errors, affect QRL: by seeing where the algorithm struggles or excels on real devices, we can improve both the algorithm and the hardware, moving us closer to practical and reliable quantum computing systems.

1. Introduction

Learning is the process by which individuals acquire the knowledge, skills, and attitudes needed to adapt and respond to life's challenges. When a child touches a burning candle, the immediate reaction is to withdraw their hand. The next time the child encounters something similar, they quickly learn to avoid it, demonstrating how experiences shape behavior. Over time, this process leads to a broader understanding, allowing the child to not only avoid the burning candle but also any other sources of harm. Our ability to learn through interaction with the environment, by observing cause and effect and understanding the consequences of our actions is fundamental to how we develop. By observing patterns, experimenting through trial and error, and receiving feedback from our surroundings, we continuously adapt and refine our skills. Memory and pattern recognition help us apply past experiences to new situations, enabling us to solve problems and make better decisions. Whether it's through achieving immediate goals or pursuing long-term objectives the goal-driven learning process is at the core of how we grow, both as individuals and as a species. Whether we're learning to drive, hold a conversation, or solve a complex problem, our actions are influenced by how the environment responds. This dynamic interaction with our surroundings forms the core of nearly all learning and intelligence theories.

In this thesis, I will explore a computational approach of learning from interaction, with a focus on a method called Reinforcement Learning (RL). Unlike other machine learning techniques, which may not always prioritize goal-oriented learning, RL is specifically designed to help systems learn through interactions with their environment in order to achieve specific objectives. The RL agent receives feedback from its actions and adapts its behavior to improve performance over time. This approach is particularly effective for tasks where step-by-step decision-making is required to reach a desired outcome, such as controlling robots, playing games, or navigating complex environments[1].

Building on this, I will explore the intersection of quantum computing and machine learning, with a particular focus on Quantum Machine Learning (QML). This emerging field leverages the unique properties of quantum systems (such as superposition, entanglement, and quantum parallelism) to enhance traditional machine learning models, including reinforcement learning (RL).

Quantum computing fundamentally differs from classical computing in how it processes information. While classical computers rely on bits that represent data as either 0 or 1, quantum computers use quantum bits, or qubits, which can exist in a superposition of both 0 and 1 simultaneously. This superposition allows quantum computers to process and store exponentially more information as the number of qubits increases. Moreover, quantum systems benefit from entanglement, a phenomenon that allows qubits to be interconnected in ways that classical bits cannot, further boosting the computational power. This unique capability makes quantum computers particularly well-suited for complex tasks, such as factoring large numbers (as demonstrated by Shor's algorithm), simulating quantum systems, and optimizing large-scale computations that are infeasible for classical systems. However, a major challenge in quantum computing is decoherence, where qubits lose their quantum state due to external interference from the environment. To perform meaningful computations, it is crucial to maintain quantum coherence long enough to complete the process, which requires advanced quantum error correction techniques to mitigate decoherence.

A promising development in quantum computing is the use of Parametrized Quantum Circuits (PQC). These circuits are a sequence of quantum gates (defined in chapter 4) that feature adjustable parameters, similar to the weights found in classical neural networks and can be optimized to perform tasks such as pattern recognition, classification, and more. Quantum Neural Networks build on this concept, aiming to replicate the functionality of classical neural networks but using quantum components.

In RL, quantum systems could significantly improve learning efficiency: QNNs allow agents to explore vast solution spaces with greater computational power, potentially speeding up the learning process and improving decision-making in situations where classical RL methods face limitations.

Given this exciting prospect, I will compare the performance of a classical Deep Q-Network (DQN) with that of a Parametrized Quantum Circuit (PQC) simulated throughout TensorFlow Quantum to assess whether the quantum approach can outperform its classical counterpart. This comparison will evaluate not only speed but also efficiency and accuracy in learning tasks, aiming to determine if quantum computing provides tangible advantages in reinforcement learning applications.

To conduct this analysis, I will implement two reinforcement learning algorithms: one based on the classical DQN framework and the other utilizing the quantum RL method. Both algorithms will be tested in a ready-to-use OpenAI Gym environment called ”Acrobot,” which provides a suitable setting for evaluating the learning performance and effectiveness of each approach. By systematically analyzing their outcomes, my goal is to investigate insights into the potential benefits of integrating quantum computing into reinforcement learning.

In chapter 2, I provide a brief introduction to reinforcement learning, highlighting its unique features and challenges compared to other machine learning techniques. I also discuss the implementation of quantum machine learning using Parameterized Quantum Circuits. The structure of a reinforcement learning task is defined in section 2.2. Methods are then described in section 2.3, where I emphasize distinctions between value-based, policy-based, and actor-critic methods. Chapter 3 outlines the key steps of Deep Q-Learning, the well established algorithm applied in both classical and quantum frameworks, with specific attention to the properties of the classical neural network used. In chapter 4, I introduce the fundamental properties of qubits and quantum gates. Section 4.3 then describes the primary features of parameterized quantum circuits. In chapter 5, I detail the environment, the state and action spaces, and the code implementations for both classical and quantum algorithms. Finally, chapter 6 presents the main results, showing how variations in batch size, network architecture, and learning rate affect performance.

2. Reinforcement Learning

Reinforcement learning (RL) is a method of learning through interaction with an environment, where an agent discovers the best actions for achieving a specific goal. Unlike other forms of machine learning, RL does not explicitly tell the agent which actions to take. Instead, the agent must explore, try different actions, and learn from the feedback (rewards or penalties) it receives. This process sets Reinforcement Learning apart from Supervised and Unsupervised Learning, two other machine learning problems. In Supervised Learning, the agent learns from a training set of labeled examples provided by an external supervisor. Each example includes a situation and the correct action or label, allowing the agent to generalize to new, unseen situations. For instance, a model may be trained to recognize objects in images based on labeled data. However, Supervised Learning is limited because it relies on predefined examples and does not provide the agent with the ability to learn from its own decisions in a dynamic environment. Unsupervised Learning, in contrast, focuses on uncovering hidden patterns or structures in data without the use of labeled examples. The goal is to organize or cluster the data meaningfully. While useful for identifying patterns, Unsupervised Learning does not involve decision-making or goal-directed actions. In Reinforcement Learning, the agent aims to maximize a reward signal through continuous interaction with its environment. The agent must learn which actions lead to higher rewards over time, without being explicitly told what to do. Unlike other approaches, RL considers the whole problem of real time decision-making, taking into account the complex, evolving nature of the environment.

One central challenge in RL is the exploration-exploitation trade-off: the agent must balance exploring new actions to improve future performance and exploiting known actions that yield immediate rewards. This trade-off is unique to RL and doesn't arise in supervised or unsupervised learning. This approach enables agents to adapt and improve performance over time, even in complex and unpredictable settings. One exciting aspect of modern reinforcement learning is its interaction with other fields, such as statistics, optimization, psychology, and neuroscience: RL is part of a long-term trend in artificial intelligence and machine learning that aims to integrate more with mathematics and other scientific areas. RL also has strong connections with psychology and neuroscience. Many RL algorithms were inspired by biological learning systems, in return, RL has contributed to psychology by providing models of animal learning that better match real-world data, and it has also helped explain parts of the brain's reward system [1].

Reinforcement Learning (RL) has found widespread application in various areas of physics, particularly in quantum systems. It has been successfully employed to determine ground states and model the unitary time evolution of complex interacting quantum systems [2], as well as optimize quantum-error-correction techniques to protect qubits from noise [3]. Moreover, RL-based control methods have demonstrated performance comparable to traditional optimal control approaches in many-body quantum systems [4]. These examples represent just a fraction of the research in this area, as RL continues to prove its versatility and effectiveness across different quantum domains. Beyond Quantum Mechanics, RL has also been applied to various classical physics problems. For example, it has been used to tackle fluid mechanics challenges [5], facilitate adaptive control in astronomy through optics [6], and optimize thermodynamic trajectories [7]. These applications underscore the broad utility of RL in both quantum and classical physics contexts.

2.1 Quantum Reinforcement Learning

Quantum Reinforcement Learning (QRL) is an area where the potential of Quantum Computing meets the principles of RL. In QRL, Parametrized Quantum Circuits replace the Classical Neural Networks used in Deep Q-Learning (DQL) (terated in chapter 3), a standard method in RL for approximating Q-values, which guide the agent's decision-making. It is anticipated that quantum phenomena may have the potential to speed up the learning process, supporting faster exploration and exploitation, and potentially helping the agent to find optimal solutions more quickly and efficiently. This could ultimately lead to improvements in RL tasks that involve large state or action spaces, where classical RL methods struggle due to the combinatorial complexity of the problem.

Despite their potential, Quantum Processing Units (QPUs) have not yet achieved the speed or computational efficiency needed to surpass classical Neural Processing Units (NPUs) in RL tasks. QPUs are still in the early stages of development, and their scalability remains an active area of research. However, platforms like PennyLane, Qiskit, and TensorFlow Quantum are making strides toward practical applications of Quantum Reinforcement Learning (QRL), enabling researchers to simulate quantum environments and train quantum models more easily.

The primary challenge is that QRL is still in its infancy, with many theoretical and experimental issues yet to be resolved. While formalizing QRL algorithms is a positive step, the full computational advantages over classical methods have not been realized. Researchers are exploring how quantum properties—such as quantum states, measurements, and gates—can enhance exploration-exploitation strategies, speed up convergence, and reduce learning times. QRL is therefore a cutting-edge area of Quantum Machine Learning that holds significant potential to transform the way intelligent agents learn and make decisions. By combining the powerful concepts of Quantum Computing (treated in chapter 4) with traditional Reinforcement Learning, QRL could eventually pave the way for solving some of the most complex decision-making problems across various domains, from autonomous systems to optimization tasks. As quantum hardware improves and more studies are conducted, the future of QRL appears promising, offering a glimpse into the next generation of machine learning algorithms [8, 9].

2.2 Structure of a Reinforcement Learning task

2.2.1 Markov Decison Process

To effectively apply Reinforcement Learning (RL), the problem must first be cast into the framework of a Markov Decision Process (MDP). This provides a structured way to model the agent-environment interaction, where decisions are made sequentially over time. By formalizing the decision-making problem in this way, the agent is able to evaluate which actions to take at any given point, based on the current state of the environment, to maximize future rewards. The defining feature of an MDP is the Markov property, which states that the future state of the system depends solely on the current state, without regard to the sequence of past states:

$$\mathbf{P}(S_{t+1}|S_t) = \mathbf{P}(S_{t+1}|S_1, S_2, S_3 ...., S_t) \tag{2.1}$$

This property ensures that the process is memoryless, as each state contains all relevant information needed to predict future states. The agent in an MDP is the decision-making entity tasked with interacting with the environment to achieve a specific goal. The agent's objective is to select actions that maximize cumulative rewards over time. As it interacts with its surroundings, the environment transitions between states, responding to the agent's actions by providing feedback in the form of rewards. These rewards guide the agent, serving as immediate indicators of how well its actions align with its goals [10].

2.2.2 State and Action Spaces

A key aspect of Markov Decision Process (MDP) formulation is defining the state space, which represents all possible configurations of the environment that the agent may encounter. The state must encapsulate all information relevant to decision-making and satisfy the Markov property, meaning that the future state depends only on the current state and action, not on the sequence of previous states. The action space, which defines the set of actions available to the agent in any given state, can involve either discrete or continuous decisions. In the case of discrete decisions, the action space is finite and countable, consisting of a well-defined list of possible choices. For example, in a simple grid-world environment, a robot might choose between moving up, down, left, or right, with a limited number of available actions. On the other hand, continuous decisions are selected from an infinite range of possible actions. In such cases, the action space is uncountable, allowing the agent to choose any value within a specified range. For instance, in a robotic arm control task, the agent might need to set the angle of rotation, which can take any value within a continuous interval, providing much finer control. The concept of discrete and continuous classifications applies similarly to the state space. In a discrete state space, the environment has a finite or countable set of distinct states, with each state representing a specific, well-defined situation. For example, in a grid-world environment, each square on the grid corresponds to a different discrete state. Conversely, a continuous state space consists of an infinite set of possible states, typically represented by continuous variables. The agent's state, in this case, can be described by real-valued parameters, allowing for a virtually unlimited number of potential configurations. An example of this would be the position and velocity of a robot in a control task, where the state is defined by continuously changing variables.

2.2.3 Reward Function

Another fundamental component of an MDP is the reward function, which provides feedback from the environment after each action taken by the agent. The reward function is critical, as it must be carefully designed to guide the agent toward achieving the desired long-term objectives. While the reward is given immediately after an action, the agent's ultimate goal is not merely to maximize immediate rewards but to maximize the cumulative reward over the course of an entire episode, which may contain many time steps. This distinction is important because short-term actions that yield high immediate rewards may not necessarily lead to optimal long-term outcomes and the agent must learn to consider the long-term consequences of its actions. This is achieved by introducing a discount factor, which assigns less weight to future rewards while still ensuring they influence the decision-making process. The discount factor ensures that the agent values future rewards, but not as highly as immediate ones, striking a balance between short-term and long-term gains. The implementation of this concept will be treated in section 2.3 where we will mathematically define this function. Thus, the agent's task is to develop a strategy that prioritizes actions contributing to overall success, even when they might involve sacrificing immediate rewards for larger cumulative returns in the future.

2.2.4 Agent-Environment Interaction

The interaction between the agent and the environment is cyclical and unfolds over discrete time steps. At each time step, the agent observes the current state, chooses an action based on this observation and the environment transitions to a new state, simultaneously providing an immediate reward. This creates a trajectory or path through the state space, as the agent continually refines its actions to improve its performance. The agent's ultimate aim is to discover an optimal policy, which is a mapping from states to actions. This policy serves as a guide, instructing the agent on which actions to take in each state. Policies can be deterministic, where the same action is always chosen for a given state, or stochastic, where actions are selected according to a probability distribution. The latter introduces an element of randomness that can be beneficial, especially in environments with significant uncertainty, as it allows for more exploratory behavior, potentially preventing the agent from settling into suboptimal strategies [11].

2.3 Reinforcement Learning Methods

As seen earlier, in Reinforcement Learning (RL), the agent does not learn from a fixed dataset. Instead, it interacts with the environment and generates its own samples through this interaction. This brings a unique challenge: the data is not pre-labeled, and the agent only receives feedback in the form of rewards. As the agent interacts with the environment, the data keeps changing, making RL dynamic and adaptive.

An environment in RL is defined by a set of possible states $\mathbf{S}$, and a set of actions $\mathbf{A}$ the agent can take to change the state. The agent performs an action a_t at time step t in state s_t, which results in a reward r_{t+1} and a transition to the next state s_{t+1}. This process forms a tuple $(s_t, a_t, r_{t+1}, s_{t+1})$, called a transition. Transition probabilities are modeled by the transition function $\mathbf{P}$, which represents the likelihood of moving from state s_t to s_{t+1} after performing action a_t:

$$\mathbf{P}^{a_t}_{s_t,s_{t+1}} = \mathbf{P}(s_{t+1}|s_t, a_t)$$

The reward function evaluates the effectiveness of the agent's actions within the given task and environment. The primary objective of the agent is to maximize the cumulative reward over time, known as the return G_t, which represents the total future rewards from time step t onwards. This return is calculated as:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.2}$$

Here, γ, the discount factor, ranges between 0 and 1 and expresses the agent's preference for immediate rewards over future ones. It ensures that the sum of rewards remains finite, effectively balancing the trade-off between short-term and long-term rewards. As emphasized in chapter 2.2, considering future rewards is crucial because the optimal action may not be the one that maximizes the immediate reward but rather one that positions the agent for higher rewards in the future. In many scenarios, the sum in equation 2.2 is finite, as tasks are divided into episodes with a fixed number of time steps, known as the horizon $\mathbf{H}$. Each episode runs for a limited duration, making the return calculation finite and bounded over the specified time horizon.

RL algorithms are typically divided into value-based and policy-based methods. Both aim to maximize the return, but they use different approaches. In both methods, the agent's behavior is defined by a policy $\pi(a|s)$, which describes the probability of selecting action a in state s. The main difference between the two methods is how the policy is modeled and optimized.

The performance of an RL agent is evaluated using a state-value function $\mathbf{V}_\pi(s)$, which gives the expected return when the agent follows a policy π starting from state s at time t:

$$\mathbf{V}_\pi(s) = \mathbb{E}_\pi[G_t | s_t = s]$$

Whereas rewards provide immediate feedback, value functions estimate long-term reward. This helps guide optimal actions not just for the next step but multiple steps into the future.

The recursive nature of these value functions is captured by the Bellman equation, which expresses the value of a state as the immediate reward plus the discounted value of the next state:

$$\mathbf{V}_\pi(s) = \mathbb{E}_\pi\left[r_t + \gamma \mathbf{V}_\pi(s_{t+1})\right]$$

Techniques like Temporal-Difference Learning and Deep Q-Networks use value functions and the Bellman equation to estimate future rewards and inform policies [12].

2.3.1 Policy-Based Methods

Policy-Based algorithms aim to directly learn the optimal policy. In this approach, the policy is represented as a parameterized conditional probability distribution $\pi(a|s;\theta)$, where a is an action, s is a state, and θ represents the parameters of the policy. The objective of the algorithm is to find the best parameters θ that lead to the optimal policy. The performance of the policy is measured using a function $\mathbf{J}(\theta)$, which quantifies how good the policy is. The algorithm's task is to maximize this performance measure. To do this, these algorithms use gradient ascent to adjust the policy: they compute an estimate of the gradient $\nabla \mathbf{J}(\theta)$ usually through Monte Carlo samples: these methods are known as Policy Gradient Methods.

One advantage of Policy Gradient Methods is that they produce smooth updates to the policy. This contrasts with value-based methods, where small changes in the value function can cause large changes in the policy. As a result, Policy Gradient Methods often converge more reliably to a locally optimal policy. However, they can suffer from high variance because updates rely on Monte Carlo samples, which can be noisy and vary a lot. To reduce this variance, several techniques have been developed, including the Actor-Critic approach. This method adds a value-based component (the critic) to the policy-based approach (the actor), helping to stabilize learning and reduce variance.

2.3.2 Value-Based Methods

In Value-Based algorithms, the focus is on learning a value function (as shown in equation 2.3), rather than learning the policy directly. The policy is implicitly determined by the value function: the agent chooses the action that maximizes the expected return, represented by $\mathbf{V}_\pi(s)$. In these methods, the agent estimates the value of each state-action pair and selects actions based on the maximum expected return. A key example of Value-Based learning is Deep Q-learning, discussed in detail in section 3. While value-based algorithms avoid the high variance seen in policy gradient methods, they often require more episodes to converge. Additionally, these methods typically result in deterministic policies, as the agent always chooses the action with the highest expected reward. This can be problematic when the optimal policy is inherently stochastic: post-training action selection is done using an argmax policy without considering the benefits of exploration. Value-Based algorithms can also be sensitive to small changes in the value function. Even a small update in the value function can lead to a significant change in the policy, potentially causing the agent to choose a different action. This sensitivity can lead to instability in learning, especially in scenarios where stability is important.

2.3.3 Actor-Critic Method

Both value-based and policy-based methods have their advantages and disadvantages. While one method may be more popular than the other at different times, there is no clear answer to which one is superior. The Actor-Critic method combines both Policy-Based and Value-Based learning to take advantage of the strengths of each approach while mitigating their drawbacks. This method is considered one of the most effective approaches in classical RL: it can be more efficient, depending on the environment and the problem at hand, as it allows for learning both the policy and the value function in parallel. The actor updates the policy based on the actions taken, while the critic evaluates these actions using the value function. By combining these methods, the Actor-Critic approach improves learning stability and reduces variance compared to pure policy gradient methods [12].

3. Deep Q-Learning

In Deep Q-Learning (DQN), the central focus is on learning the action-value function, $Q_\pi(s,a)$, rather than the value function alone. The action-value function provides the expected return for taking a specific action a in state s, assuming the agent follows a given policy π afterward. This function is a crucial component of Q-learning because it directly ties actions to future rewards, enabling the agent to learn which actions maximize cumulative rewards over time.

3.1 Action-Value Function and Optimal Policy

The Q-function under a policy π is defined as:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a]$$

where G_t is the return function defined in 2.2. This formulation estimates the expected return based on both the state and the action, providing more detailed information than the state-value function alone. While the state-value function $V_\pi(s)$ gives the expected return from being in state s and following policy π thereafter, the Q-function provides the expected return for taking a specific action a. This distinction is crucial because it allows the agent to assess not only how good a state is, but also to determine which actions in that state will maximize future rewards. Q-function provides a more action-specific view of the expected rewards, helping the agent make decisions based on the immediate choice of action, rather than just the state.

The optimal Q-function, $Q^*(s,a)$, represents the maximum expected return for taking action a in state s, assuming the agent follows an optimal policy moving forward. It provides the best possible outcome considering both the immediate reward and the best possible future rewards that can be obtained:

$$Q^*(s,a) = \max_\pi Q_\pi(s,a)$$

This equation states that the optimal Q-value for a state-action pair (s,a) is the highest possible Q-value across all policies. In practice, this means that the agent will follow the policy that maximizes expected returns.

Once the agent has learned the optimal Q-function, it can derive the optimal policy, $\pi^*(s)$, by selecting the action that maximizes the Q-value for the current state s:

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

By following the optimal policy, the agent is able to effectively solve the Reinforcement Learning environment, maximizing cumulative rewards and achieving the best possible outcomes over time.

3.2 Tabular Q-Learning

In the original Q-learning algorithm, a Q-table is used to store Q-values for each possible state-action pair.

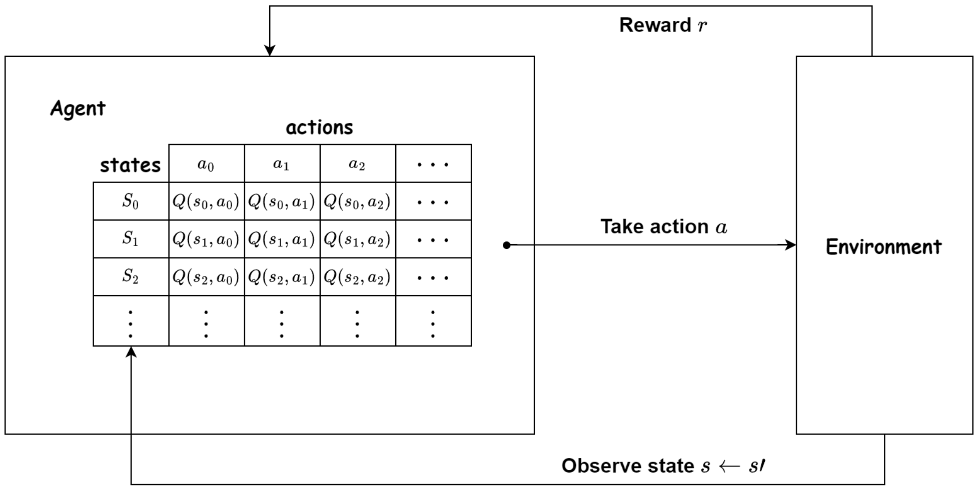


FIGURE 3.1: Simple schematic of Q-Learning implemented with a Q-Table (source: [13]).

The agent interacts with the environment, updating the Q-values iteratively using the following update rule:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right]$$

where α is the learning rate, controlling how much new information overrides old estimates, γ is the discount factor, determining the importance of future rewards, r_{t+1} is the reward at time $t+1$, $Q(s_t,a_t)$ is the current Q-value associated to taking action a_t in state s_t and $\max_a Q(s_{t+1},a)$ estimates the maximum future reward from the next state assuming that the agent follows the optimal policy [12].

The learning rate α is a critical hyperparameter in Reinforcement Learning. It must be carefully tuned to ensure the environment is solved efficiently and optimally .

While Q-tables work well for simple problems, they become impractical in environments with large state and action spaces, where storing and updating all possible state-action pairs is computationally expensive. Real-world problems often have vast or continuous state spaces, making the tabular approach intractable.

3.3 Deep Q-Networks

Traditional tabular Q-learning methods encounter significant challenges: a Q-table becomes impractically large in such scenarios, making tabular methods impractical. To address this limitation, Mnih et al. [14] introduced Deep Q-Networks (DQN), which combine Q-learning with deep Neural Networks (DNNs). DQNs utilize Neural Networks as function approximators to estimate the Q-values, eliminating the need for a Q-table and enabling the agent to scale to large state and action spaces.

3.3.1 Artificial Neural Networks (ANNs)

An Artificial Neural Network (ANN) is a machine learning model inspired by the structure and function of Biological Neural Networks. It is composed of nodes, or artificial neurons, organized into three types of layers: an input layer, one or more hidden layers, and an output layer.

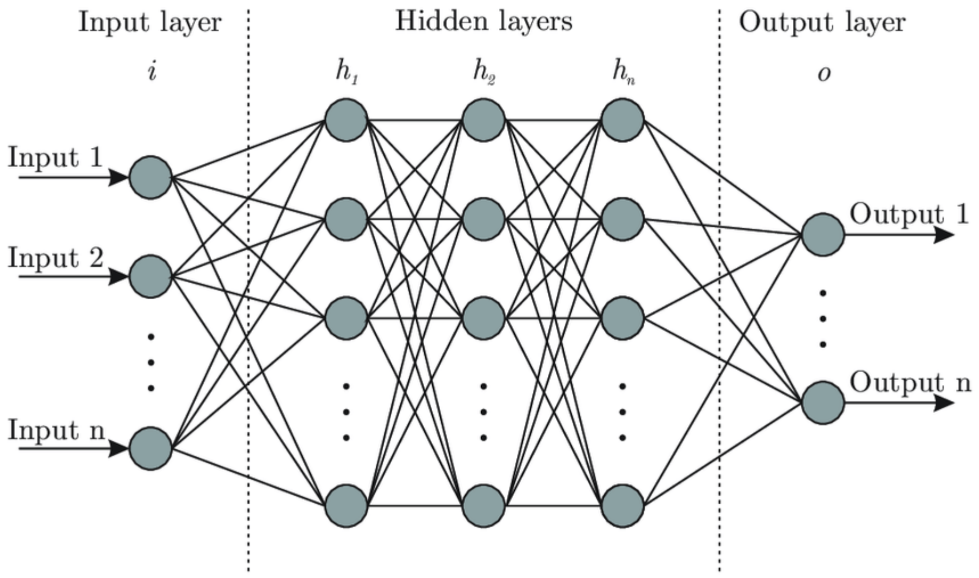


FIGURE 3.2: Simple schematic of a Deep Neural Network (source: [15]).

The neural network architecture is a tunable hyperparameter in my algorithm. Its variations are treated in section 6.2 and its shown how a low number of neurons can impact the learning. Each connection from neuron i in one layer to neuron j in the next has a weight θ_{ij}: these weights adjust during training based on how well the node's output contributes to the desired result, mimicking the strengthening of synapses in biological networks, which occurs based on correlated activity. In the case of Deep Q-Networks, this output represents the Q-value for each possible action. During training, the network adjusts these weights so that it can better predict the expected rewards for taking a particular action a in a given state s [16].

3.3.2 Stabilizing Training

To stabilize the learning process, DQN employs two neural networks:

- A Main Network that estimates the Q-values for the current state s and action a, represented by $Q(s,a;\theta)$. The parameters of this network are denoted by θ, and the network is responsible for learning and updating Q-values.
- A Target Network that is an identical copy of the main network parametrized with θ' that remains frozen for several iterations, meaning its parameters θ' are periodically updated to match the main network's weights. The target network is used to compute the target Q-values q_i^δ for the next state s_i' and action a_i'.

The key advantage of using a target network is to avoid the moving target problem. As the network updates its Q-values, the Q-value of the next state also changes, which can cause the learning process to oscillate or become unstable. By periodically copying the weights of the main network to the target network, the Q-values in the target network remain stable over a period, allowing the Q-values in the main network to converge more effectively.

3.3.3 Experience Replay

Another key feature of DQN is Experience Replay, which enhances training stability. Transitions from past episodes are stored in a memory buffer and sampled randomly during training. This breaks temporal correlations between consecutive experiences, leading to more stable learning and better generalization. By learning from a diverse set of experiences, the agent can avoid overfitting to recent observations and improve its ability to generalize across different states and actions.

3.4 DQN Training

Training a DQN involves minimizing the difference between the predicted Q-values and the target Q-values. At each training step, a batch B of past transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ is sampled from the replay memory. The target Q-value is calculated as:

$$q_i^\delta = r_{t+1} + \gamma \cdot \max_a Q_{\theta'}(s_{t+1}, a)$$

Here, θ' represents the parameters of the target network, and γ is the discount factor. The loss function used to update the network is the mean squared error (MSE) between the predicted Q-values $Q_\theta(s_t, a_t)$ and the target Q-values q_i^δ:

$$L(\theta) = \frac{1}{|B|} \sum_{b \in B} (Q_b - q_b^\delta)^2$$

The DQN algorithm optimizes the neural network parameters θ using gradient descent methods such as stochastic gradient descent or more advanced methods like Adam. This allows the network to update the Q-values for the actions taken, improving the agent's decision-making over time. The updated Q-value for a given state-action pair (s_t, a_t) is computed as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[q_i^\delta - Q_\theta(s_t, a_t) \right] = Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q_{\theta'}(s_{t+1}, a) - Q_\theta(s_t, a_t) \right]$$

The term $r_{t+1} + \gamma \max_a Q_{\theta'}(s_{t+1}, a) - Q(s_t, a_t)$ is the difference between the target Q-value and the current Q-value, and it shows how far off the current Q-value is from the ideal or target Q-value. This difference is also called the temporal difference (TD) error.

3.4.1 Exploration-Exploitation trade-off

To ensure the agent explores enough of the state-action space, Deep Q-Learning uses an ε-greedy policy. With probability ε, the agent selects a random action, and with probability $1-\varepsilon$, it chooses the action that maximizes the Q-value for the current state:

$$a_t = \arg\max_a Q(s_t, a)$$

The exploration-exploitation trade-off is essential for effective learning, as it prevents the agent from converging prematurely to suboptimal strategies. Initially, a higher exploration rate ε encourages the agent to sample a wide variety of actions, gathering sufficient information about the environment. Over time, as the agent learns more about the state-action space, ε is gradually reduced, allowing the agent to increasingly rely on the learned policy and exploit the knowledge gained. This controlled reduction ensures that, after ample exploration, the agent prioritizes actions that maximize long-term rewards, while avoiding unnecessary randomness [17, 18].

4. Quantum Information and Computation

4.1 Quantum Bits

Similarly to the bit for the classical, quantum information and computation theory relies on an analogous concept: the quantum bit, or qubit. In the realm of quantum mechanics, a qubit is defined as any system that can exist in two states or levels. Just as a classical bit has states identified as 0 or 1, a qubit is also represented by two physical states, typically denoted $|0\rangle$ and $|1\rangle$. The fundamental difference between bits and qubits lies in the ability of quantum states to exist in configurations beyond just $|0\rangle$ and $|1\rangle$: in fact, qubits can form linear combinations or superpositions of states, expressed mathematically as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{4.1}$$

where α and β are complex numbers. A quantum state $|\psi\rangle$ can be seen as a vector in a specific vector space, which can be decomposed into elements of the orthonormal basis $\{|0\rangle, |1\rangle\}$, commonly referred to as the computational basis. In quantum mechanics, measurement is inherently probabilistic: when a qubit is in a specific state like $|0\rangle$ or $|1\rangle$, we can be certain of obtaining that result upon measurement, however, when the qubit is in a superposition, the measurement outcome depends on the coefficients α and β. For instance, if the qubit is in the superposition $\alpha|0\rangle + \beta|1\rangle$, the probability of measuring $|0\rangle$ or $|1\rangle$ is given by:

$$P(|0\rangle) = |\alpha|^2 \quad \text{and} \quad P(|1\rangle) = |\beta|^2 \tag{4.2}$$

To ensure the state is valid and the probability sums up to 1, the coefficients must satisfy the normalization condition:

$$|\alpha|^2 + |\beta|^2 = 1 \tag{4.3}$$

It is important to note that any global phase in the state $|\psi\rangle$ is irrelevant, as it cancels out in all physical calculations. The only method to extract information from a qubit is through measurement. It is impossible to retrieve both α and β from a single measurement since the state collapses: for example, if the system is in the state $\alpha|0\rangle + \beta|1\rangle$ and a measurement gives $|0\rangle$, the qubit collapses into $|0\rangle$, and subsequent measurements (if the state is not perturbed) will consistently return $|0\rangle$ with probability 1. To uniquely determine both α and β, an infinite

number of measurements on identically prepared qubits would be necessary. However, this setup is impractical due to the No-Cloning theorem. This theorem states that:
Given a normalized quantum state $|\psi\rangle$, there does not exist any unitary operator U such that

$$|\psi\rangle \otimes |\phi\rangle \to U(|\psi\rangle \otimes |\phi\rangle) = |\psi\rangle \otimes |\psi\rangle \quad \forall |\psi\rangle \textit{ normalized.}$$

This means that it is impossible to create an identical copy of an arbitrary unknown quantum state. Despite this limitation, qubits remain valuable for computation, as certain quantum operations can extract useful information without the need to fully determine both α and β.

Observables

In quantum mechanics, observables are Hermitian operators and are the quantities that we can actually measure in a quantum system. In the case of a two-level quantum system, operators are represented by 2×2 matrices. Just as we can decompose a generic state $|\psi\rangle$ as a linear combination of eigenstates $|n\rangle$, we can also decompose any Hermitian operator in $\mathbb{C}^2$ as a linear combination of the Pauli matrices, defined as:

$$\sigma_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \sigma_2 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad \sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{4.4}$$

These Pauli matrices form a basis for the space of Hermitian 2×2 matrices and are essential in quantum mechanics for describing spin and other two-level systems.

Bloch Sphere Representation

A useful tool to visualize a qubit is the Bloch sphere. In this representation, the state of the qubit is depicted as a point on the surface of a sphere. The poles of the Bloch sphere correspond to the classical states $|0\rangle$ and $|1\rangle$, while the position of a arbitrary qubit state can be defined by the angles θ and ϕ using:

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

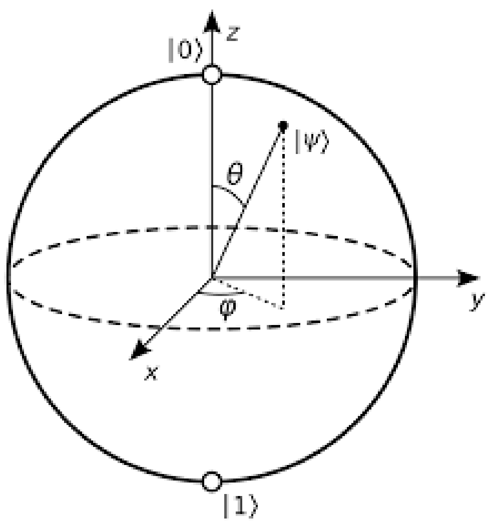


FIGURE 4.1: Representation of the state $|\psi\rangle$ on the Bloch Sphere

It is essential to emphasize that the Bloch sphere functions primarily as a graphical tool. Specifically, it can be viewed as a mapping $\mathbb{C}^2 \to \mathbb{R}^3$, which is not an isomorphism. This discrepancy results in the non-conservation of angles and, consequently, the inner product. In $\mathbb{C}^2$, the states $|0\rangle$ and $|1\rangle$ are orthogonal, however, on the Bloch sphere, we observe that

$$\langle 0|1\rangle = -1 \tag{4.5}$$

Multi-Qubits Systems and Entanglement

Quantum systems become particularly interesting when composed of multiple qubits, as they allow access to exponentially larger Hilbert spaces. Consider a quantum system composed of two subsystems, A and B, with their respective Hilbert spaces $\mathscr{H}_A$ and $\mathscr{H}_B$. The total Hilbert space for the composite system is the tensor product of these individual spaces:

$$\mathscr{H} = \mathscr{H}_A \otimes \mathscr{H}_B$$

One of the key advantages of quantum computing over classical computing arises from this exponential scaling of information. While a classical system can process $O(n)$ bits of information for n-bit registers, a quantum system with n qubits can encode and process information that scales as $O(2^n)$.

A particularly fascinating aspect of multi-qubit quantum systems is the phenomenon of entanglement. Entanglement occurs when the quantum state of a system of two or more qubits cannot be factored into a simple tensor product of individual qubit states. For instance, consider the entangled state:

$$|\Psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}.$$

This state cannot be written as a product state of the form $|\Psi\rangle \neq |a\rangle \otimes |b\rangle$ for any single-qubit states $|a\rangle$ and $|b\rangle$. The inability to separate the state into individual components implies that the qubits are entangled; this property is crucial for quantum computing, as entanglement allows for correlations between qubits that are unattainable in classical systems, providing a substantial computational advantage [19] .

4.2 Quantum Gates

Quantum gates are the quantum analog of classical logic operations. One key difference between classical and quantum gates is that we cannot directly implement simple classical operations like AND, OR, or XOR on qubits, as we do with bits, because qubits are implemented troughout a physical systems. The physical phenomena used to manipulate quantum states vary: for example, when qubits are encoded in particles with quantum mechanical spin, logic gates are applied by manipulating the spin via a magnetic field with varying orientations, alternatively, if the qubit is encoded in the internal excitation state of an ion, gate operations are performed by changing the duration of laser irradiation or adjusting the wavelength of the laser. These manipulations are described by unitary, Hermitian operators acting on the Hilbert space. Since the operators governing the time evolution of a closed quantum system are reversible, they can be represented as unitary matrices. This property also ensures that quantum operations preserve the length of vectors. The Pauli matrices, which are unitary and Hermitian, allow us to construct quantum gates. These are defined as follows:

$$\sigma_1 \equiv X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad \sigma_2 \equiv Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad \sigma_3 \equiv Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Hadamard Gate

A particularly important gate is the Hadamard gate, represented by the Hadamard matrix[19]:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{4.6}$$

The Hadamard matrix is unitary since $H^{\dagger}H = I$, and it is frequently used in quantum circuits. The Hadamard gate allows for a change of basis from $\{|0\rangle, |1\rangle\}$ to $\{|+\rangle, |-\rangle\}$. The $|+\rangle$ and $|-\rangle$ states, known as the *Hadamard basis*, are defined as:

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

Thus, under the Hadamard transformation:

$$|0\rangle \xrightarrow{H} |+\rangle, \quad |+\rangle \xrightarrow{H} |0\rangle, \quad |1\rangle \xrightarrow{H} |-\rangle, \quad |-\rangle \xrightarrow{H} |1\rangle$$

Rotation Gates

Rotation gates perform rotations around the Bloch sphere's axes, parameterized by a continuous angle θ. The matrix representations of the rotation gates are:

- **X-Rotation Gate**:

$$R_X(\theta) = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix}$$

- **Y-Rotation Gate**:

$$R_Y(\theta) = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix}$$

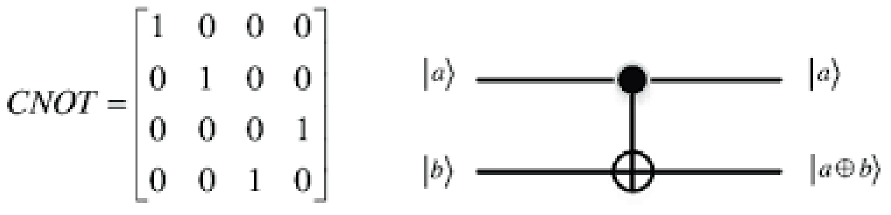
- **Z-Rotation Gate**:

$$R_Z(\theta) = \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix}$$

They play a critical role in parameterized quantum circuits, where the angles are adjusted during optimization [19].

Two-Qubit Gates

Two-qubit gates introduce entanglement, which is a key feature in quantum computing. A common example is the Controlled-NOT (CNOT) gate, its matrix representation is:



$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
$|a\rangle$
$|b\rangle$
$|a\rangle$
$|a \oplus b\rangle$

FIGURE 4.2: CNOT matix and quantum circuit representation (source: [20]).

The CNOT gate flips the state of the target qubit if the control qubit is in the $|1\rangle$ state [19].

4.3 Parametrized Quantum Circuits

As discussed earlier in Section 2.1, Quantum Machine Learning (QML) seeks to leverage quantum principles to enhance traditional machine learning methods. A key tool in this approach is the use of Parameterized Quantum Circuits, also known as Variational Quantum Circuits (VQCs), which serve to approximate policy functions much like neural networks do in classical ML. A quantum circuit is a model of quantum computation that applies a sequence of quantum gates to a set of qubits. At the end of the circuit, the qubits are measured, yielding classical data that reflects the outcome of the computation. In the context of Q-learning, VQCs have been utilized as function approximators. Here, instead of using Deep Neural Networks (DNNs), VQCs predict the expected rewards for actions taken in different states, allowing a quantum-classical hybrid approach to reinforcement learning. VQCs work as function approximators by adjusting free parameters in the quantum circuit to optimize the model's performance iteratively. This hybrid method divides the computational work between classical and quantum systems, saving quantum resources for parts that are challenging for classical computers, such as high-dimensional optimizations. As a result, the need for extensive quantum coherence time, deep circuits, and a large number of qubits is minimized.

4.3.1 Steps of the PQC-based Q-Learning Model

To apply PQCs in environments with continuous states (such as Acrobot), the VQ-DQL model follows these steps [21]:

1. **Data Encoding (State Preparation)**: Classical data has to be encoded into quantum states to be processed by a quantum computer. In this model, each component of the continuous input vector is encoded as a rotation angle using *Angle Encoding*. Specifically, each input component x is transformed through a classical preprocessing function, normalized, and mapped to rotation gates (RX, RY, RZ) on qubits, representing the values as rotation angles. For added flexibility, a technique called *trainable input scaling* is applied to each feature, allowing the VQC to adaptively scale each input angle, which helps the model match the target function better.

   Given an input vector $x = \{x_0, x_1, \dots, x_{n-1}\}$, the resulting encoded quantum state is:

   $$|x\rangle = \bigotimes_{i=0}^{n-1} R_\alpha(\phi(x_i \cdot \lambda_i))|0_i\rangle$$

   where:

   - $R_\alpha \in \{RX, RY, RZ\}$ are Pauli rotation gates,
   - ϕ is a preprocessing function (for instance, normalization)
   - λ_i are the trainable scaling parameters.

   This encoding method efficiently represents each input feature as a rotation, but requires a number of qubits that scales with the number of input features, which can be limiting on current Noisy Intermediate-Scale Quantum (NISQ) devices.

2. **Variational Processing**: The circuit introduces entanglement between qubits with controlled gates, and rotation gates are adjusted by parameters to enhance the model's expressivity. For complex problems, additional layers of data encoding and processing (known as *data re-uploading*) may be used, where input data is fed into multiple layers of the circuit to increase expressivity. The number of encoding layers is a tunable hyperparameter that needs to be optimized carefully to ensure the reinforcement learning model converges to an optimal policy.

3. **Measurement and Output Decoding**: The processed quantum states are measured, producing Q-values for different possible actions, which represent the expected rewards for each action in a given state. In Variational Quantum DQL, the Q-value for a specific action a is derived from the expectation value of an observable O_a associated with that action, calculated as:

   $$Q(s,a) = \langle 0^{\otimes n} | U_\theta(s)^\dagger O_a U_\theta(s) | 0^{\otimes n} \rangle$$

   where:

   - n is the number of qubits,
   - $U_\theta(s)$ represents the parameterized quantum circuit operation based on the state s
   - O_a is the observable specific to action a, typically a Pauli operator or a tensor product of Pauli operators like σ_z.

   This vector of Q-values serves as the basis for action recommendations based on the current state.

4. **Parameter Optimization**: To optimize the VQC, a cost function is defined, which represents the error between the Q-value predictions and the target values. The cost function often used in Q-learning is the Mean Squared Error (MSE) as for the classical case, computed over a batch of samples from the environment. For a dataset $D = \{(x_i, y_i)\}_{i=0}^{M-1}$, the MSE cost function is given by:

   $$L(\theta) = \frac{1}{M} \sum_{i=0}^{M-1} (f_\theta(x_i) - y_i)^2$$

   where $f_\theta(x)$ is the VQC output for input x with parameters θ, and y_i represents the target Q-value.

   In practice, the value of $f_\theta(x)$ for a specific input x is obtained by running the VQC multiple times and averaging the measurement results to estimate the expectation value. This approach, known as sampling, reduces noise in the output. To minimize $L(\theta)$, gradient-based optimization methods (e.g., stochastic gradient descent or Adam) are typically used to iteratively adjust the VQC parameters.

In the approach used for my work, the observables for evaluating the Q-values corresponding to different actions are defined as the tensor product of σ_z operators applied to pairs of qubits. Each action is evaluated with a distinct product of these σ_z operators acting on different qubit pairs. For example, the observable for each action is formed by applying the σ_z operator to different pairs of qubits, such as qubits 0 and 1, 2 and 3, and 4 and 5. The PQC model can be thought of as a Fourier series, where the frequency range is set by the eigenvalues of the encoding gates, and the series coefficients are influenced by variational gates and measurements. By tuning the parameters of the VQC, the model approximates the Q-function needed to solve reinforcement learning tasks. In this study, we implement PQC-based Q-learning in the Acrobot environment, a continuous control task. The PQC's structure i used in my implementation can be visualized in the following image:

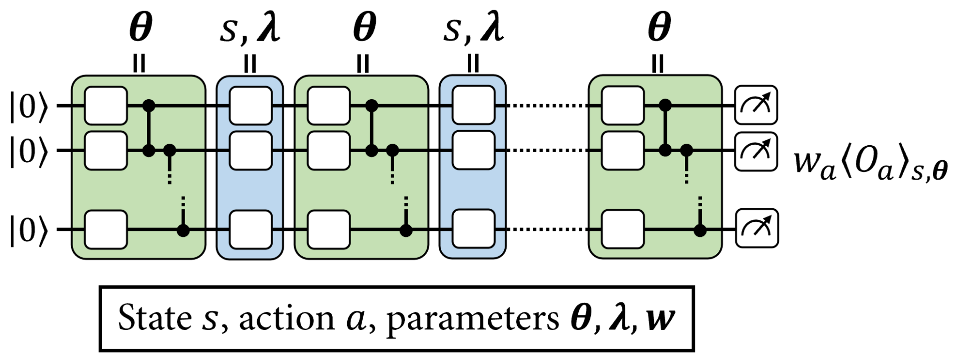

$\boldsymbol{\theta}$
$s, \boldsymbol{\lambda}$
$\boldsymbol{\theta}$
$s, \boldsymbol{\lambda}$
$\boldsymbol{\theta}$
$|0\rangle$
$|0\rangle$
$|0\rangle$
$w_a \langle O_a \rangle_{s,\boldsymbol{\theta}}$
State s, action a, parameters $\boldsymbol{\theta}, \boldsymbol{\lambda}, \boldsymbol{w}$

FIGURE 4.3: Simple schematic of the PQC used in my algorithm (source: [22]).

5. Reinforcement Learning problem implementation

In this chapter, I will outline the environment and code implementation used for data collection, which will be analyzed in the chapters that follow. As previously mentioned, my reinforcement learning problem was approached using Python, with core implementations based on TensorFlow and TensorFlow Quantum, supported by a variety of supplementary libraries.

5.1 The Acrobot-v1 Environment

The Open AI Gym Acrobot environment provides a valuable setup for training reinforcement learning agents by simulating a two-link robotic arm that operates against gravity in a vertical plane. It is described by a continuous state space and discrete action space and it is straightforward to integrate with TensorFlow, making it suitable for various reinforcement learning techniques. The Acrobot consists of two connected links, or segments, arranged in a chain, with one end anchored at a fixed pivot point. The system's only actuator is located at the joint between the two links (analogous to an elbow or waist), while the first joint (shoulder) is passive. This configuration allows torque application only at the actuated joint: by carefully coordinating the motion, the agent learns to swing the free end of the chain upward to reach a specified height. The primary control objective is the swing-up task. This task is complex due to the gravitational forces acting on the system and the limited actuation at the single joint, challenging the agent to master pendulum-like swinging and precise timing to reach the desired configuration. The control challenges are significant and involve energy-efficient movement (the agent must optimize torque application to build momentum without direct actuation at both joints), timing and coordination. The Acrobot system's dynamics mirror those of simpler models of bipedal or walking robots, where coordinated actuation at limited joints (e.g., the hip or knee) is required for effective motion.

Its resemblance to such systems makes it an invaluable training ground for algorithms aimed at walking or hopping robots. Additionally, its setup is a common benchmark for exploring reinforcement learning algorithms, control theory, and robotics principles. Studying control strategies for the Acrobot helps researchers gain insights into energy-efficient control, timing-based actuation, and stability maintenance, essential components for robotic systems operating in complex environments [23, 24].

5.1.1 State Space description

As noted earlier, the state space in this environment is continuous, defined by six observables that provide essential information about the system's two rotational joint angles and their respective angular velocities. These observables are detailed in Table 5.1 below [25].

Num	Observable	Min	Max
1	Cosine of θ_1	-1	1
2	Sine of θ_1	-1	1
3	Cosine of θ_2	-1	1
4	Sine of θ_2	-1	1
5	Angular velocity of θ_1 ($\frac{rad}{s}$)	-4π	$+4\pi$
6	Angular velocity of θ_2 ($\frac{rad}{s}$)	-9π	$+9\pi$

Table 5.1: Acrobot continuous state space description.

By analyzing the evolution of these six parameters, the agent must determine the action that will guide it toward the most advantageous subsequent state, ultimately helping it achieve the desired goal.

5.1.2 Action Space description

The action space is discrete. It contains all the possible actions that the agent can perform and represents the torque applied on the actuated joint between the two links. These actions are described in the table below [25].

Num	Action	Unit
0	Apply -1 torque to the actuated joint	torque (Nm)
1	Apply 0 torque to the actuated joint	torque (Nm)
2	Apply 1 torque to the actuated joint	torque (Nm)

Table 5.2: Acrobot discrete action space description

The actions can be visualized in the following figure.

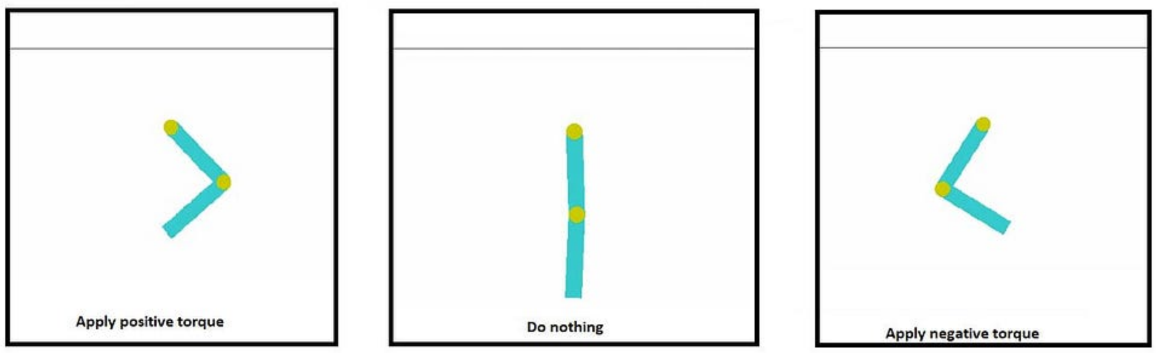


FIGURE 5.1: Vialization of Agents actions on acrobot environment

5.1.3 Episode description

An Acrobot episode consists of 500 time steps, where the agent selects actions based on either its learned policy or an exploratory random policy. Initial state parameters, θ_1, θ_2, and the two angular velocities, are randomly initialized within $[-0.1, 0.1]$, positioning both links downward with slight variability. The goal is to reach a target height in as few steps as possible. Each unsuccessful step results in a reward of -1, totaling -500 if the agent fails to reach the target within 500 steps. A successful attempt ends the episode with a reward of 0.

5.2 Algorithms implementation

5.2.1 Deep Q-Learning C51/Rainbow implementation

In section 3, I introduced the Deep Q-Learning algorithm, outlining the primary functions and steps that an agent follows to learn an optimal policy. The classical reinforcement learning framework I used builds upon Deep Q-Learning, incorporating a variation known as C51/Rainbow. The code can be fully consulted on the TensorFlow official web page [26]. The following sections will provide an overview of the code's main structures. C51 is an extension of the Q-learning algorithm based on DQN, applicable to environments with a discrete action space. Unlike DQN, which estimates a single Q-value for each state-action pair, C51 models a probability distribution over possible Q-values for each state-action pair. By approximating a distribution rather than an expected value, C51 achieves greater training stability and improved performance. To learn over these probability distributions rather than scalar values, C51 employs specialized distributional computations within its loss function.

CategoricalQNetwork initialization

To create a C51 Agent, we begin by defining a CategoricalQNetwork, which incorporates a crucial parameter called num_atoms. This parameter determines the number of discrete support points in the probability distribution used to estimate future rewards. Support points are specific values within this distribution that represent potential outcomes for Q-values associated with state-action pairs. Instead of providing a single estimated Q-value, the network evaluates the probability of landing on each support point, thus modeling the entire distribution of possible future rewards. These support points enable the agent to model a distribution of Q-values, rather than a single expected value, which is central to the C51 approach.

```
categorical_q_net = categorical_q_network.CategoricalQNetwork(
    train_env.observation_spec(),
    # Environment observation space
    train_env.action_spec(),
    # Environment action space
    num_atoms=num_atoms,
    # Number of atoms for probability distribution
    fc_layer_params=fc_layer_params)
    # Network architecture (fully connected layers)
```

CategoricalDqnAgent initialization

The CategoricalDqnAgent is initialized using the previously defined categorical_q_net, with key modifications specific to the categorical approach. Unlike the standard DqnAgent, this version requires the specification of min_q_value and max_q_value, which set the lower and upper bounds for the agent's Q-Value distribution. Here, these values are set to -500 and 0, representing the worst and best possible game scores, respectively. Furthermore, the code specifies the discount factor γ and the epsilon-greedy exploration policy, which define how the agent learns and interacts with the environment.

```
agent = categorical_dqn_agent.CategoricalDqnAgent(
# Define the Categorical DQN agent
    train_env.time_step_spec(),
    # Environment time step specification
    train_env.action_spec(),
    # Environment action specification
    categorical_q_network=categorical_q_net,
    # Categorical Q-network
    optimizer=optimizer,
    # Optimizer
    min_q_value=min_q_value,
    # Minimum Q-value
    max_q_value=max_q_value,
    # Maximum Q-value
    n_step_update=n_step_update,
    # n-step update for return
    td_errors_loss_fn=common.element_wise_squared_loss,
    # Loss function for TD errors
    gamma=gamma,
    # Discount factor for future rewards
    train_step_counter=train_step_counter,
    # Step counter
    epsilon_greedy=lambda: epsilon_fn(train_step_counter))
    # Epsilon-greedy policy for exploration
agent.initialize()
# Initialize the agent
```

Data collection

The focus of this section of the code is on gathering initial experience data by employing a random policy and storing the results in a replay buffer. This collection of data is essential for training the agent, as it ensures a diverse array of experiences that can be revisited during the training process to enhance learning efficiency.

```python
def collect_step(environment, policy):
    time_step = environment.current_time_step()
    # Get the current time step
    action_step = policy.action(time_step)
    # Choose an action using the given policy
    next_time_step = environment.step(action_step.action)
    # Take a step in the environment
    traj = trajectory.from_transition(time_step, action_step,
   next_time_step)
    # Create a trajectory from the transition
    replay_buffer.add_batch(traj)
    # Add the experience to the replay buffer
# Collect initial random steps to populate the replay buffer
for _ in range(initial_collect_steps):
    collect_step(train_env, random_policy)
# Dataset for sampling from replay buffer
dataset = replay_buffer.as_dataset(
# Create a dataset from the replay buffer
    num_parallel_calls=3, sample_batch_size=batch_size,
    # Use parallel calls and batch size for sampling
    num_steps=n_step_update + 1).prefetch(3)
    # Prefetch batches for efficiency
iterator = iter(dataset)
# Create an iterator for the dataset
```

Agent training

The training loop begins in the following section of the code where the agent collects data and then samples experiences from the replay buffer to optimize its performance. Periodically, the agent is evaluated by running episodes to measure its performance, which is logged and saved for analysis. This loop continues for a specified number of iterations, with regular logging.

```python
for iteration in range(num_iterations):
    # Collect steps using the agent's collect policy
    for _ in range(collect_steps_per_iteration):
        collect_step(train_env, agent.collect_policy)
        # Collect data using the agent's collect policy
    # Sample a batch of data from the replay buffer and update the agent
    experience, unused_info = next(iterator)
    # Sample a batch of experiences
    train_loss = agent.train(experience)
    # Train the agent using the sampled batch
    step = agent.train_step_counter.numpy()
    # Get the current training step
    if step % log_interval == 0:
    # Log the training loss at regular intervals
        print(f'Episode = {step}: loss = {train_loss.loss}')
    # Evaluate the agent every eval_interval steps
    if step % eval_interval == 0:
        avg_return = compute_avg_return(eval_env, agent.policy,
   num_eval_episodes)
        # Evaluate the agent's performance
        print(f'Episode = {step}: Average Return = {avg_return:.2f}')
        returns.append(avg_return)
        # Store the evaluation results
        # Run an evaluation episode and store the total reward
        total_reward = run_episode(eval_env, agent.policy)
        # Run a single episode and get the total reward
        episode_rewards.append(total_reward)
        # Append the reward to the list of episode rewards
```

The code was executed over 20,000 episodes, logging the average reward every 100 steps (with eval_interval = 100).

Each logged value represents the mean reward achieved over the preceding 100 steps. In the next chapter will compare the agent's performance across variations in three key parameters:

- Batch Size: Adjusted to identify a suitable size that maintains stability without excessively impacting runtime.
- Neurons in Hidden Layers: Different neural network architectures were tested to find the optimal number of neurons that enhances policy learning depth without compromising runtime efficiency.
- Learning Rate: Tuned to assess which rate yields the most effective agent behavior.

5.2.2 Deep Q-Learning with PQC approximators implementation

The quantum framework used for my reinforcement learning problem involves a TensorFlow implementation of Parametrized Quantum Circuits (PQC) for approximating Q-functions. It is important to note that the circuits are simulated on a classical computer and are not subject to noise and decoherence that qubits experience in actual quantum computers. To accurately assess the true potential of quantum reinforcement learning based on PQC, I should have executed the code on real quantum computers. This could have been accomplished using IBM's quantum systems; however, for the sake of simplicity and to expedite the process, I opted to rely on simulated circuits. The code can be accessed and consulted on the TensorFlow official page [27].

Circuit Generation

The first step in implementing reinforcement learning with Parametrized Quantum Circuits (PQC) is to generate the quantum circuit itself. To achieve this, I utilized various libraries from TensorFlow Quantum. The following function defines the structure of the circuit, which includes the input qubits, as well as the encoding and variational layers. In this implementation, the function generate_circuit prepares a data re-uploading circuit by following a structured approach: it initializes the circuit with the specified number of qubits and layers, defines the variational parameters and input angles, and constructs the circuit layer by layer. Each layer consists of variational operations, entangling operations, and encoding operations that collectively contribute to the circuit's overall functionality.

```python
def generate_circuit(qubits, n_layers):
 # Number of qubits being used in the circuit
 n_qubits = len(qubits)
 # Creation of the variational angles (parameters) used in the circuit
 params = sympy.symbols(f'theta(0:{3*(n_layers+1)*n_qubits})')
 params = np.asarray(params).reshape((n_layers + 1, n_qubits, 3))
 # Creation of th input angles that will be used to encode data
 inputs = sympy.symbols(f'x(0:{n_layers})'+f'_(0:{n_qubits})')
 inputs = np.asarray(inputs).reshape((n_layers, n_qubits))
 circuit = cirq.Circuit()
 # Loop over the number of layers to build the circuit
 for l in range(n_layers):
     # Variational layer: applies one-qubit rotations for each qubit
     circuit += cirq.Circuit(one_qubit_rotation(q, params[l, i]) for i, q in enumerate(qubits))
     # Entangling layer
     circuit += entangling_layer(qubits)
     # Encoding layer
     circuit += cirq.Circuit(cirq.rx(inputs[l, i])(q) for i, q in enumerate(qubits))
 # Last variational layer
 circuit += cirq.Circuit(one_qubit_rotation(q, params[n_layers, i]) for i, q in enumerate(qubits))
 # Return the constructed circuit
 return circuit, list(params.flat), list(inputs.flat)
```

ReUploadingPQC class definition

The ReUploadingPQC class implements a re-uploading Parametrized Quantum Circuit (PQC). In its setup, the class first calls the parent class constructor and stores the number of layers and qubits. It then generates the quantum circuit and the corresponding parameters for variational angles and input scaling.

```
class ReUploadingPQC(tf.keras.layers.Layer):
    def __init__(self, qubits, n_layers, observables, activation="linear
   ", name="re-uploading_PQC")
    """Initializes the ReUploading Parametrized Quantum Circuit (PQC).
        Args:
            qubits: Number of qubits used in the circuit.
            n_layers: The number of layers in the PQC.
            observables: The quantum observables to measure.
            activation: The activation function to apply to the inputs.
            name: The name of the layer """
```

The class initializes two key variables: self.theta for the variational angles and self.lmbd for the input-scaling parameters. It also prepares an empty quantum circuit and sets up a computation layer. In the call method, the class handles input data by first determining the batch size and then replicating the quantum circuit to match this size. It also tiles the variational angles and input data, scales the inputs using the input-scaling parameters, and applies an activation function. Finally, it combines the variational angles and scaled inputs, reorders them as needed, and returns the output from the computation layer. Overall, this class efficiently integrates quantum circuits into the TensorFlow framework for use in neural networks.

Agent Training

In the final code section, the model is trained on batched samples of episodes collected through interactions with the environment. For each batch, the states and actions are aggregated, while the returns (discounted future rewards) are calculated based on the observed rewards. These returns, paired with the state-action data, guide the model's parameter updates to improve its policy. After each episode, the cumulative reward is recorded in episode_reward_history, and an average reward over the last 10 episodes is computed to monitor the agent's learning progress and performance trends. This rolling average serves as a metric for evaluating improvement over time.

```python
for batch in range(n_episodes // batch_size):
# Iterate over batches of episodes
    # Gather episodes: collect experiences in the environment
    episodes = gather_episodes(state_bounds, n_actions, model,
   batch_size, env_name)
    # Group states, actions, and returns into numpy arrays
    states = np.concatenate([ep['states'] for ep in episodes])
    # Combine all states from episodes into a single array
    actions = np.concatenate([ep['actions'] for ep in episodes])
    # Combine all actions from episodes into a single array
    rewards = [ep['rewards'] for ep in episodes]
    # Extract rewards for each episode
    returns = np.concatenate([compute_returns(ep_rwds, gamma) for
   ep_rwds in rewards])
    # Compute returns for each episode's rewards
    returns = np.array(returns, dtype=np.float32)
    # Create an array of (index, action) pairs to identify actions taken
    in the states
    id_action_pairs = np.array([[i, a] for i, a in enumerate(actions)])
    # Update model parameters using the REINFORCE algorithm
    reinforce_update(states, id_action_pairs, returns, model)
    # Store the total reward collected from each episode
    for ep_rwds in rewards:
        episode_reward_history.append(np.sum(ep_rwds))
    # Calculate the average reward from the last 10 episodes
    avg_rewards = np.mean(episode_reward_history[-10:])
    # Average rewards over the last 10 episodes
```

The Quantum DQN code was executed for 600 episodes, with average rewards evaluated every 10 steps. To compare the quantum agent's sensitivity to parameter variations with that of a classical agent, the adjusted parameters included batch size, learning rate, and the number of PQC hidden layers. As detailed in section 6.3, the learning rate was subdivided into three distinct values, each managed by a specific optimizer for the input weights, variational parameters, and output weights, enabling more precise control over the learning process.

6. Data Collection and Analysis

In this chapter, I will assess the performance of both classical and quantum agents by adjusting the parameters discussed in previous chapters to observe each agent's behavior. I will then compare the best-performing runs for each agent to identify the conditions under which each agent excelled.

6.1 Batch size variations

Batch size refers to the number of transitions (state-action-reward-next state tuples) sampled from the replay buffer for a single update step of the model during training. It plays a critical role in influencing the learning dynamics of the agent. Larger batch sizes typically lead to more stable estimates of Q-values by averaging them over more samples, which can enhance convergence and improve learning efficiency. This allows the agent to learn from a wider range of experiences in each update, potentially aiding generalization in complex environments. However, larger batch sizes require more memory and computational resources, leading to longer training times. Thus, it is essential to strike a balance between stability and computational efficiency.

Classic DQN run

In my data collection, I carefully investigated which batch size—32, 64, 128, or 256—would be the optimal choice for achieving the best performance in the specific problem or environment at hand. Single runs rewards are shown in the four pictures below **??**. The points of the graph were plotted using a moving average with a window size of 2 points. All data collection was conducted using a learning rate of $5 \cdot 10^{-5}$ and a neural network architecture with three hidden layers of 256 units each. This configuration was identified as optimal based on prior exploratory runs around the TensorFlow suggested value. As shown in the following figures, a larger batch size enables the agent to learn more quickly. For instance, if we set a performance threshold at -100, the run with a batch size of 32 surpasses this threshold after approximately 5,000 episodes, while a batch size of 64 allows the agent to reach the threshold slightly faster, around 4,800 episodes. A batch size of 128 achieves the threshold at about 4,600 episodes, though with increased instability. The 256 batch size demonstrates the fastest learning rate, crossing the threshold in only 2,300 episodes. A larger batch sizes should stabilize training by

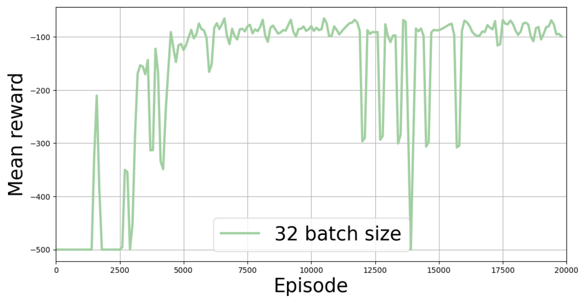


FIGURE 6.1: Average rewards over training with 32 batch size

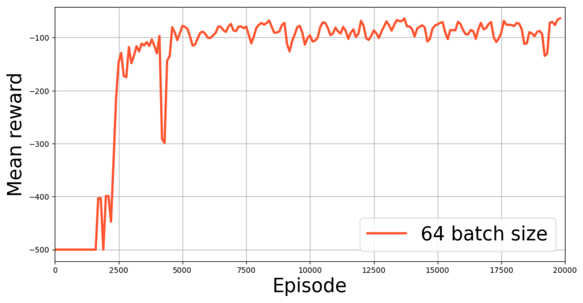


FIGURE 6.2: Average rewards over training with 64 batch size

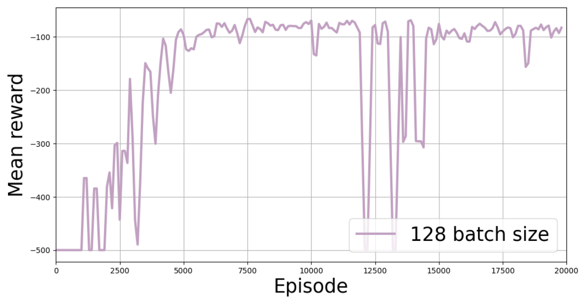


FIGURE 6.3: Average rewards over training with 128 batch size

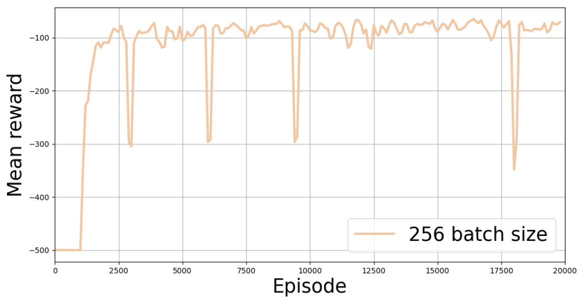


FIGURE 6.4: Average rewards over training with 256 batch size

using a broader sample of experiences for each update, which generally produces smoother and more accurate learning. This pattern holds when comparing 32 and 64 batch sizes, where the larger size improves stability. However, the 128 batch size unexpectedly introduces the most instability giving high oscillations between 11000 and 1500 episode range. A batch size of 128 is large enough to reduce immediate responsiveness to data changes but too small to fully stabilize gradient noise as the 256 batch does, causing inconsistent corrections in the model's updates. Categorical DQN's unique structure, which learns a probability distribution across reward bins, may further contribute to instability. This method is sensitive to rare events which shift probability distributions unevenly. The 128 batch size may capture these events inconsistently, leading to overreaction to high rewards or over-penalization of low ones. Random initialization of the environment can further amplify this effect, as the batch size of 128 lacks the adaptability of smaller batches and the stability of larger ones, resulting in a volatile learning process. A batch size of 256 leads to a slight more stable policy with random single peaks of bad rewards. The runs are illustrated in the figure below, where the highlighted red line represents the batch size of 64. This size was selected as the optimal choice due to its balance of stability and efficient run time.

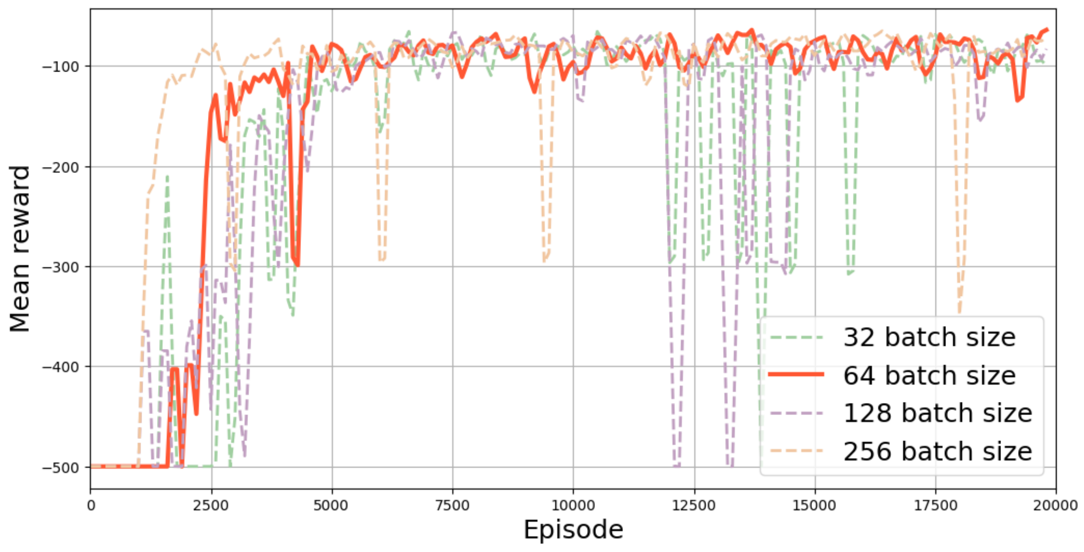


FIGURE 6.5: Batch size variation runs comparison

Quantum DQN run

For the following runs i fixed the learning rate to $1 \cdot 10^{-4}$, the standard value used by TensorFlow, and the number of variational layers to 10. To better visualize the reward trend the point in the graph were drawn throughout a moving average with a window size of 10 points. From the graphs below it is possible to observe that the number of episodes sampled from the replay batch does not influence the learning stability that much. What is important to note is the fact that the computational time of the runs increases more than the classical one because the algorithm has to simulate an exponential increasing of quantum circuits. It important to note also that the maximal score is achieved by the maximal batch size, decreasing with smaller ones. The 64 bath seems to be the most stable, but the computational time went up to ~ 6 hours, my choice fell on a batch size of 32, with a run time of ~ 3 hours. This leads to a not so stable run, but i observed that the stability is then reached choosing a number of variational layers equal to eight as it is possible to see in figure 6.17 in section 6.2.

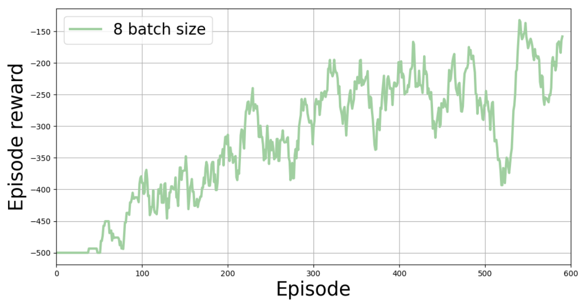


FIGURE 6.6: Average rewards over training with 8 batch size

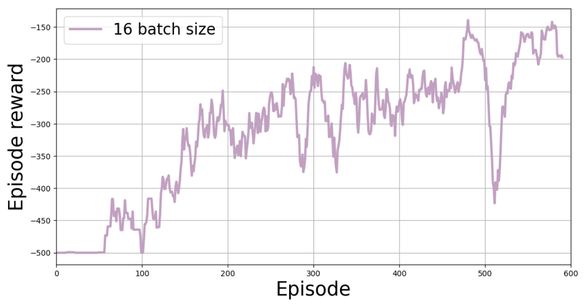


FIGURE 6.7: Average rewards over training with 16 batch size

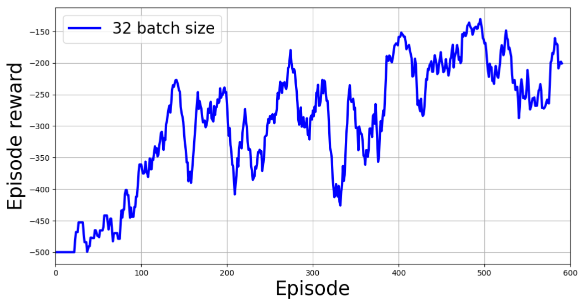


FIGURE 6.8: Average rewards over training with 32 batch size

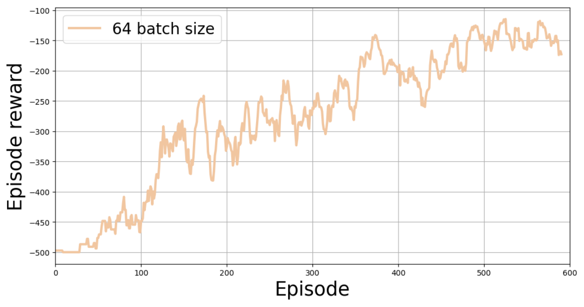


FIGURE 6.9: Average rewards over training with 64 batch size

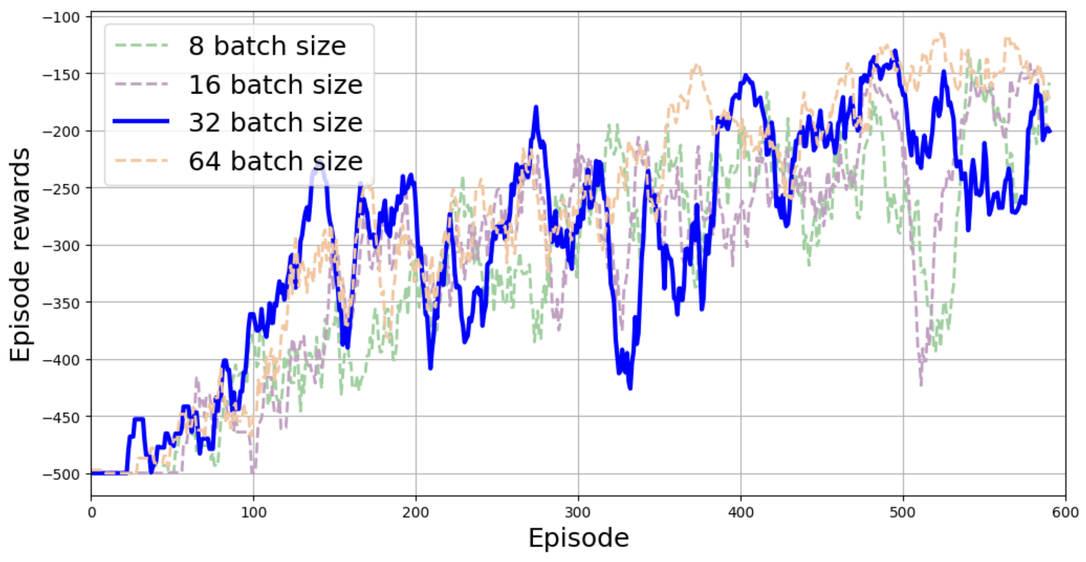


FIGURE 6.10: Batch size variation runs comparison

6.2 Neural Network and PQC architecture variations

The architecture of the neural network is a crucial hyperparameter that significantly impacts convergence in reinforcement learning algorithms. An insufficient number of neurons or hidden layers can hinder the network's ability to effectively understand and adapt to the environment. The number of neurons directly correlates with the number of learnable weights that guide the agent in its decision-making process. If the number of learnable parameters is too low, the agent may struggle to select the most appropriate actions, leading to suboptimal performance. This limitation arises because a neural network with a reduced number of neurons lacks the capacity to represent the complexity of the environment adequately. As a result, it may fail to capture essential patterns and relationships within the state space, which are critical for effective decision-making. The same principle applies to the variational layers of a PQC: an insufficient number of layers results in a poor understanding of the environment and a suboptimal policy. PQCs are affected by the "barren plateau" phenomenon: this occurs when, beyond a certain threshold of variational layers, adding more layers no longer enhances the model's expressivity, resulting in no further performance improvements. In this regime, the optimization landscape becomes nearly flat, and gradients approach zero, making training practically ineffective. Consequently, adding more layers does not improve the model's performance, as optimization becomes increasingly difficult. A similar effect can occur in classical neural networks, where an excessive number of neurons can lead to overfitting. When a model overfits, it learns to fit the training data too precisely, capturing noise rather than generalizable patterns. This results in poor generalization to new data and inefficient learning, as the model struggles to find stable policies that perform well outside the training environment. In both quantum and classical models, then, over-parameterization can hinder effective learning, highlighting the importance of carefully tuning the model's complexity to balance expressivity and efficiency.

Classic DQN run

For the upcoming runs, I fixed the batch size at 64 and set the learning rate to $5 \cdot 10^{-5}$. I then varied the number of neurons in the hidden layers, testing configurations of 32, 64, 128, and 256 neurons for each of the three layers. The points were plotted throughout a moving average with a window size of 2, to make the graph less noisy.

This approach enables an evaluation of how different network sizes affect the agent's learning abilities and overall performance in the given environment. As it is possible to see in plots 6.11 and 6.12, the 32 and 64 neurons configurations are not sufficient to deeply understand the environment evolution and the learning process is very unstable. The performance of these neural networks also stabilizes at lower reward levels, with scores oscillating between -100 and -250 for the configuration with 32 neurons and between -100 and -150 for the configuration. However, even with this increase, the model does not fully escape the limitations of its expressivity and optimization dynamics, ultimately reaching a plateau in performance within these reward ranges. The 6.13 shows a unstable learning in the beginning (from 7500 to 12500) that slowly stabilizes with a reward oscillating around -100, and the 6.14 run is stable from almost the start settling on a reward oscillating around -90. Due to the fact that i could observe that the run time is not influenced so much by the neurons in the hidden layers, i decided to opt for 256 neurons, sacrificing the run time for a better stability and ultimate reward.

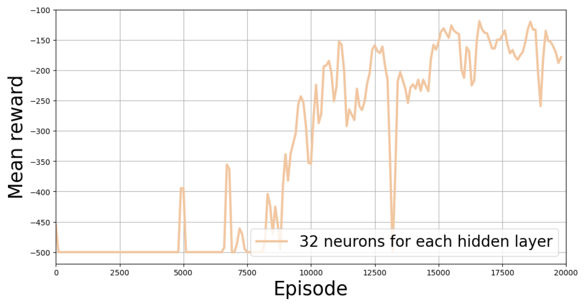


FIGURE 6.11: Average rewards over training with 32 neurons for each hidden layer

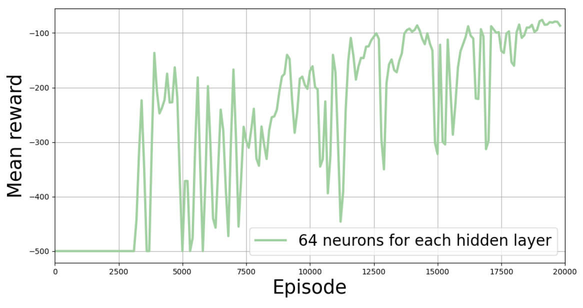


FIGURE 6.12: Average rewards over training with 64 neurons for each hidden layer

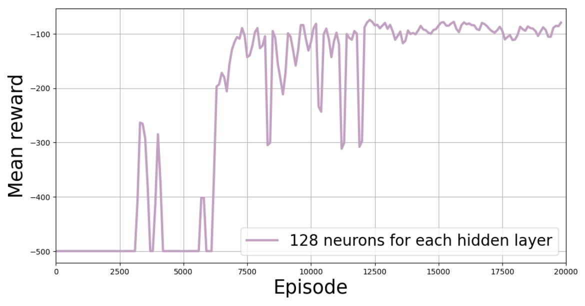


FIGURE 6.13: Average rewards over training with 128 neurons for each hidden layer

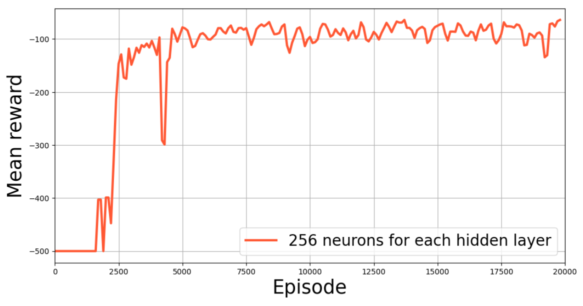


FIGURE 6.14: Average rewards over training with 256 neurons for each hidden layer

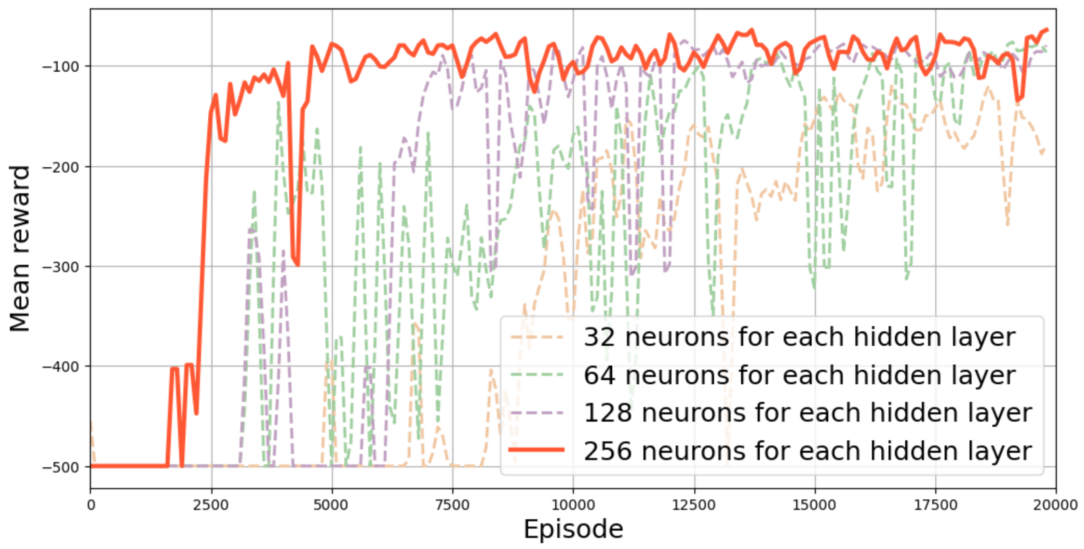


FIGURE 6.15: Hidden neurons variation runs comparison

Quantum DQN run

In the following graphs, we analyze the agent's behavior as the number of encoding layers varies, with the batch size fixed at 32 (as chosen in Section 6.1) and the learning rate set to $1 \cdot 10^{-4}$. The results show that increasing the number of encoding layers does not necessarily enhance learning stability or performance. For example, the learning curve for the 8-layer configuration, shown in 6.17, exhibits better stability and accuracy compared to the 14-layer setup in 6.19. The latter demonstrates oscillations between -140 and -400 in the episode range from 400 to 500, highlighting a lack of consistency. Adding more layers can lead to increased sensitivity to noise, particularly in quantum environments where noise can disrupt the learning process: this sensitivity introduces instability, making it harder for the model to converge smoothly. The instability observed in 6.19 suggests that adding more layers can add unnecessary complexity without substantial improvements in learning performance. In fact, the 14-layer model may begin to track random fluctuations in the environment, rather than learning a stable, generalizable policy. Moreover, PQC are inherently prone to overparameterization, which further amplify this issue. The 10-layer configuration also shows considerable variability, particularly between 200 and 400 episodes, where rewards oscillate between -150 and -420. Based on these observations, I selected the 8-layer configuration as it strikes an optimal balance between speed and stability, while also achieving higher performance than the 6-layer setup. Specifically, the rewards for the 8-layer model oscillate between -140 and -200 in the final 100 episodes, whereas the 6-layer configuration shows a wider oscillation range, between -140 and

-250, in the same episode range. Thus, the 8-layer model offers more consistent learning and a better overall performance than both the 6-layer and 10-layer configurations.

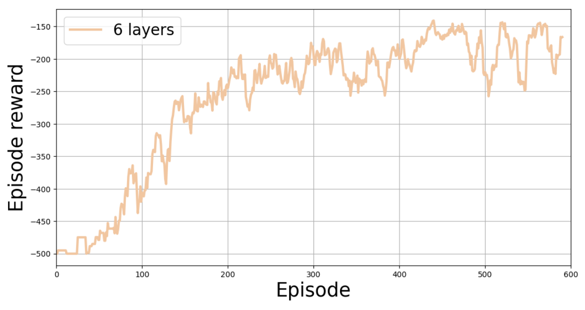


FIGURE 6.16: Average rewards over training with 6 variational layers

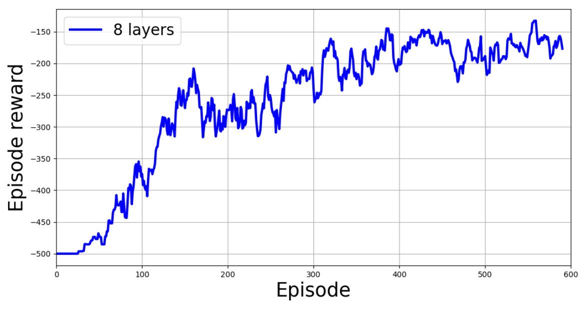


FIGURE 6.17: Average rewards over training with 8 variational layers

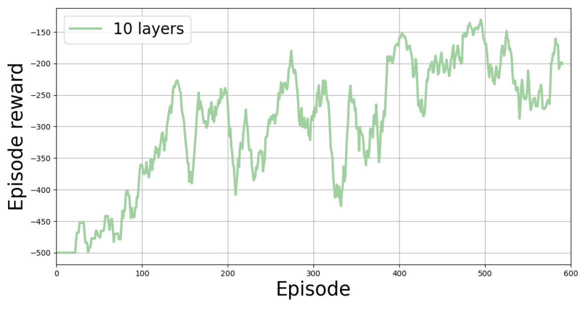


FIGURE 6.18: Average rewards over training with 10 variational layers

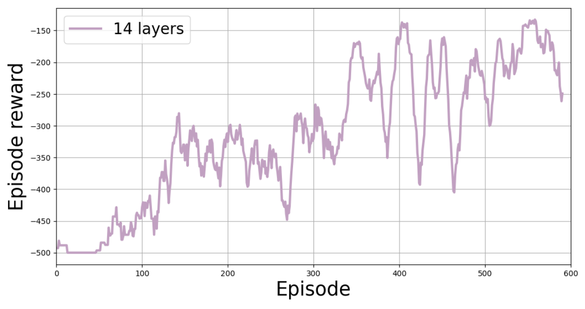


FIGURE 6.19: Average rewards over training with 14 variational layers

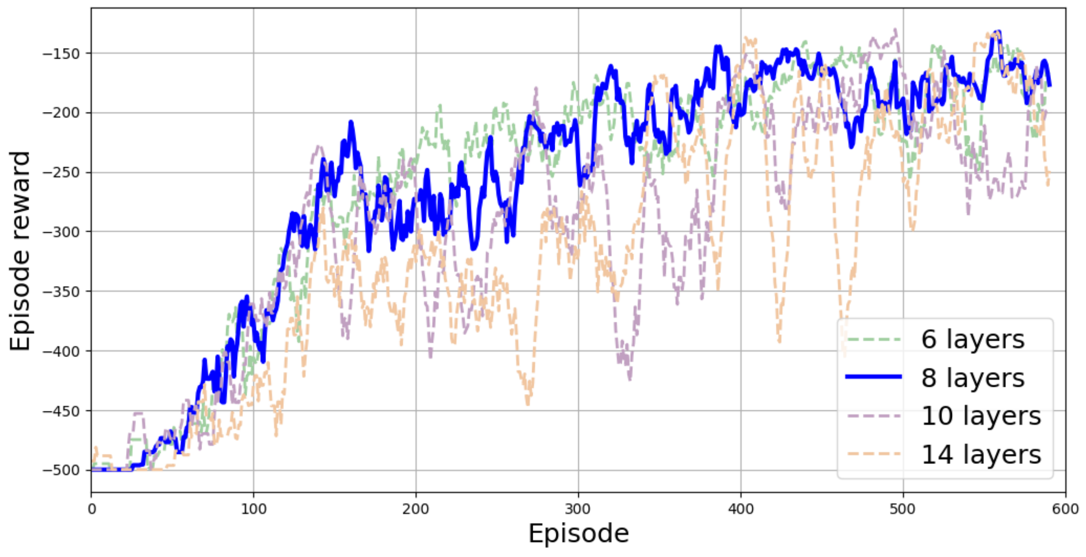


FIGURE 6.20: Number of parametrized layers variation runs comparison

6.3 Learning Rate variations

The learning rate, a positive scalar value, is crucial in reinforcement learning as it controls the pace at which the algorithm updates model parameters, as shown in 3.2. This hyperparameter directly affects how fast or slow the model learns from data. A low learning rate allows for smaller, more precise updates to the model's parameters, which can make training more stable and help the model gradually converge to a solution. However, it may require more iterations to reach an optimal policy. On the other hand, a high learning rate speeds up training, helping the model learn faster. If the learning rate is set too high, it can cause the model to overshoot optimal parameter values, leading to oscillations and instability. This can prevent the model from converging and may reduce performance. The learning rate choice strongly impacts both training efficiency and success, and it often requires careful tuning or dynamic adjustment methods, such as learning rate scheduling or adaptive algorithms, to achieve the best results. In my case, I adjusted the learning rate based on a small exploration around the optimal values suggested by the original TensorFlow code. For both the classic and quantum DQN, I tested values slightly higher and lower than these to find the best setting.

Classic DQN run

In the classic DQN setup, I initially experimented with a learning rate lower than the optimal value, tuning it to $3 \cdot 10^{-5}$, which led to instability during training, as illustrated in Figure 6.21. Setting a reward threshold of -100, the model successfully converged after approximately 8500 episodes. However, after a period of stability between episodes 10,000 and 12,500, the learning process began to oscillate again. This suggests that while the model initially achieved stable performance, it encountered some form of instability or sensitivity to environmental fluctuations as it continued to learn. To address this issue, I tested higher learning rates and ultimately settled on a value of $5 \cdot 10^{-5}$. This learning rate provided an effective balance, enabling the model to converge more rapidly while maintaining stability throughout the training process. The main difference between the chosen learning rate and the other two ($8 \cdot 10^{-5}$, $1 \cdot 10^{-4}$) is the stability obtained in the first 5000 episodes: the $5 \cdot 10^{-5}$ learning rate settles really smoothly from the starting -500 reward to the -100 reward in approximately 4000 episodes. The final choice of this learning rate significantly enhanced both the efficiency and reliability of the model's learning.

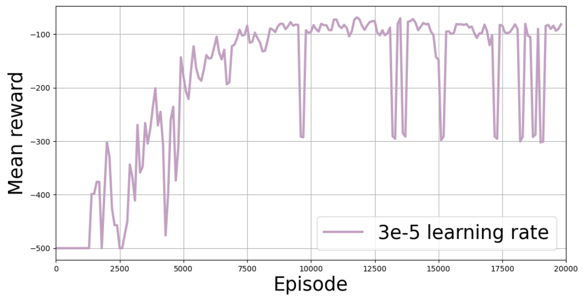


FIGURE 6.21: Average rewards over training with lr = 3e-5

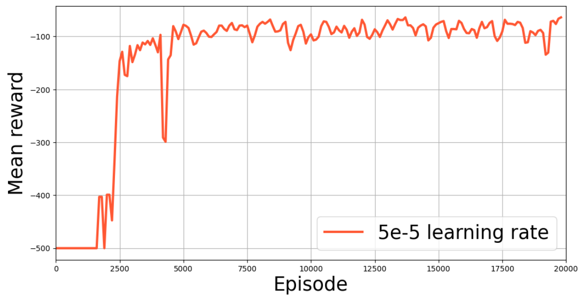


FIGURE 6.22: Average rewards over training with lr = 5e-5

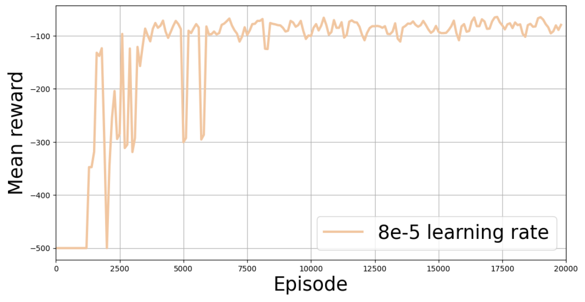


FIGURE 6.23: Average rewards over training with lr = 8e-5

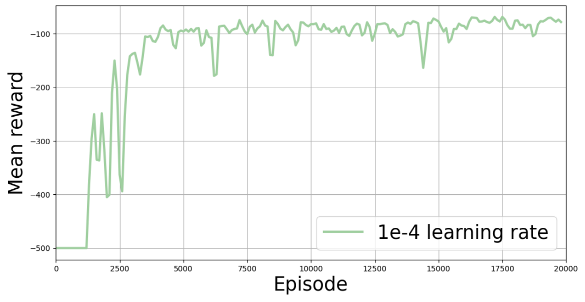


FIGURE 6.24: Average rewards over training with lr = 1e-4

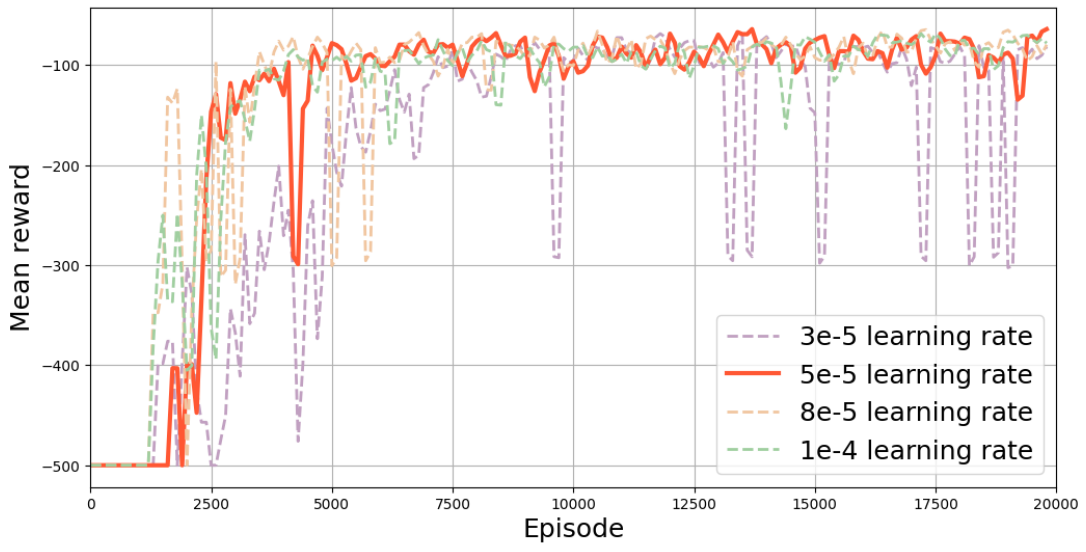


FIGURE 6.25: Learning rate variation runs comparison

Quantum DQN Run

In the quantum DQN, the PQC utilizes three distinct learning rates, as discussed in Section 5.2.2. These learning rates correspond to the input, variational, and output layers, each playing a distinct role in the learning process. Among them, the variational learning rate is the most sensitive and crucial for effective training. This is because the variational layer contains the parameters that directly affect the model's ability to fine-tune its performance. Therefore, the variational learning rate has the greatest impact on the quantum DQN's overall performance. Given its importance, I focused primarily on the variational learning rate; if this learning rate is too high, the model adjusts its parameters too aggressively, leading to instability and making it difficult to reach an optimal solution. Conversely, if the learning rate is too low, the model adapts too slowly, requiring more episodes to converge or possibly settling on suboptimal solutions. Therefore, it is crucial to carefully tune this rate to strike a balance between efficient learning and model stability. By default, the variational learning rate is set to $1 \cdot 10^{-4}$, which, as shown in the following images, is highly sensitive to small changes. When reduced to $0.95 \cdot 10^{-4}$, the model's learning process becomes less stable. Convergence time increases, meaning it takes significantly more episodes to reach a solution. Additionally, the policy developed under this reduced learning rate is less effective, with performance oscillating between -250 and -180. These oscillations suggest that the model struggles to settle into an optimal policy, and its learning trajectory becomes less predictable and consistent. On the other hand, increasing the variational learning rate causes even more instability. Larger learning rates result in erratic updates to the model's parameters, causing large fluctuations in the learning process. These fluctuations can lead the model to overshoot optimal solutions, hindering convergence. For instance, when the learning rate is increased to $1.02 \cdot 10^{-4}$ or $1.05 \cdot 10^{-4}$, the performance drops further, with the model's performance peaking at -250 and -200, respectively, both of which are worse than the baseline performance achieved with the original learning rate. These unstable runs were discarded not only because they produced suboptimal results but also because they made the training process unpredictable and unreliable. The instability observed with both lower and higher learning rates highlights the sensitivity of the quantum DQN to this hyperparameter. A balance must be found where the learning rate is neither too large, causing erratic behavior, nor too small, causing slow convergence. In my case, the optimal learning rate was found to be $1 \cdot 10^{-4}$, as it provided the best balance between stability and efficient learning, allowing the model to converge reliably and perform optimally.

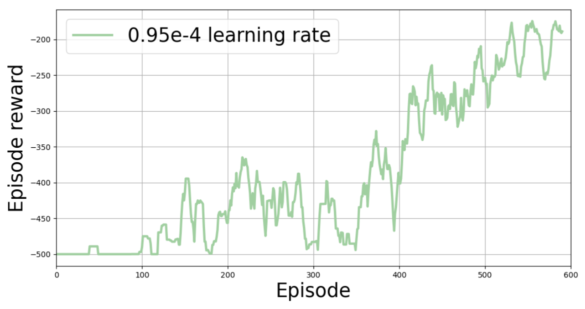


FIGURE 6.26: Episode rewards over training with lr = 0.95e-4

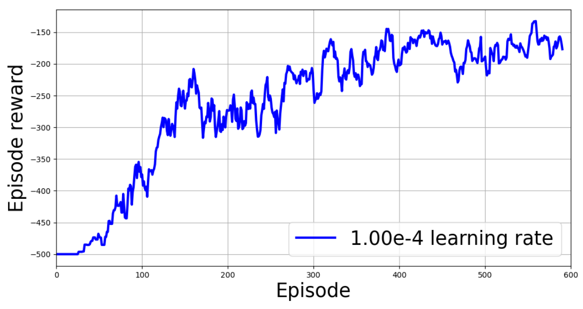


FIGURE 6.27: Episode rewards over training with lr = 1e-4

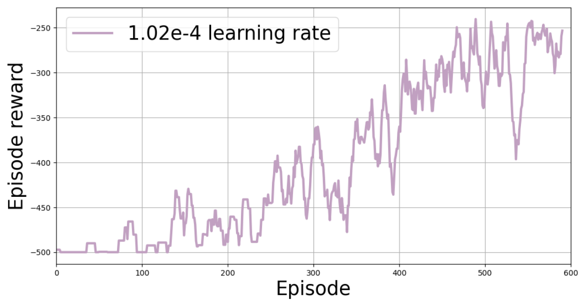


FIGURE 6.28: Episode rewards over training with lr = 1.02e-4

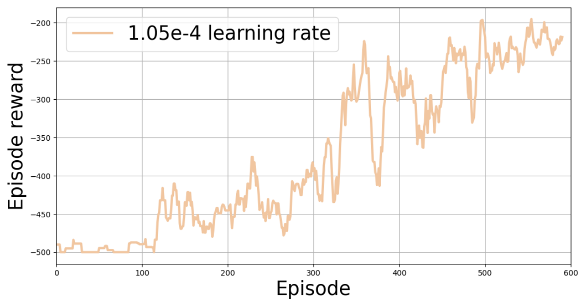


FIGURE 6.29: Episode rewards over training with lr = 1.05e-4

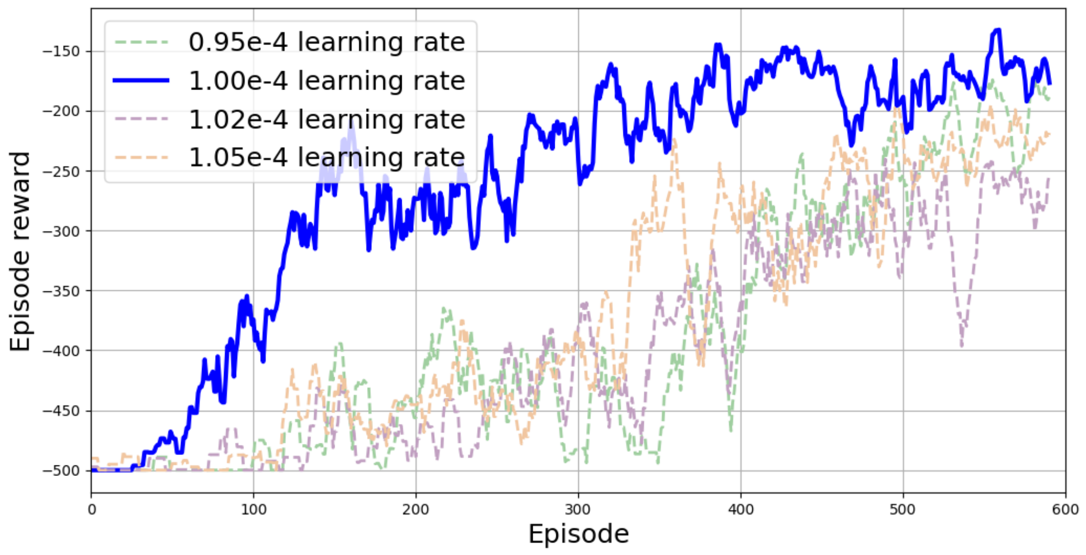


FIGURE 6.30: Learning rate variation runs comparison

6.4 Comparison between Classic and Quantum Runs

To enable a direct comparison between the classical and quantum approaches, I standardized the number of episodes to 600 for both methods. With standard hyperparameters fixed in sections 6.1, 6.2 and 6.3, the classical approach initially plateaued at a reward of -500 for the limited number of episodes, making adjustments necessary. The classical neural network required significant tuning to yield meaningful results. To address this, I increased the network's capacity by setting the number of neurons to 600, which proved optimal for enhancing model expressivity while avoiding overfitting. Higher neuron counts led to poorer performance, likely due to instability in the learning process. I also set a larger batch size of 1000, allowing the model to leverage more data per update and improve its gradient estimates. The learning rate was set to the optimal value identified in Section 6.3. For the quantum approach, hyperparameters were fixed to the best configuration identified in previous sections, using 8 variational layers, a batch size of 32, and a learning rate of $1 \cdot 10^{-4}$. The figure below shows the reward comparison for the classical and quantum models over the 600 episodes, with rewards smoothed using a moving average window of 10 points for clarity.

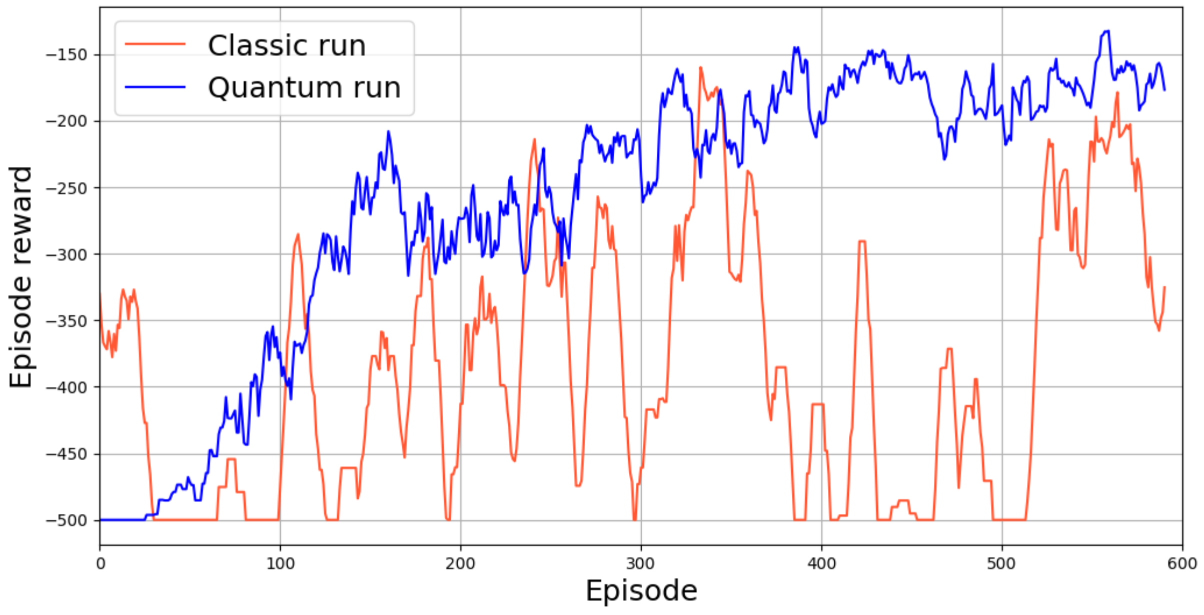


FIGURE 6.31: Comparison between classical and quantum runs rewards obtained over 600 episodes.

The comparison between the classical and quantum approaches over 600 episodes reveals several key differences in performance and stability. The quantum approach, shown by the blue line, demonstrates a steady improvement in rewards throughout the episodes. Around episode 300, the quantum model's performance stabilizes, with rewards consistently falling between -150 and -200, with peaks reaching -140, suggesting that the model has effectively learned and reached a stable performance level. In contrast, the classical approach, represented by the red line, shows a more unstable trajectory. While there are periods of improvement, particularly between episodes 200 and 400, the classical model frequently drops back to around -500, indicating it struggles with stable learning and maintaining higher rewards. In terms of stability, the quantum model is considerably more consistent. After an initial learning phase, its reward values fluctuate within a narrow range, implying that it has converged to a stable solution within the 600-episode limit. Meanwhile, the classical model experiences sharp and frequent oscillations, often reverting to lower rewards. This volatility suggests that the classical approach has not yet achieved stable learning and may still be struggling to generalize effectively within the current configuration. By the end of the 600 episodes, the quantum approach maintains significantly higher rewards than the classical approach. While the quantum model plateaus at a favorable reward level, the classical model remains inconsistent, with lower average rewards and notable drops, indicating a less reliable learning process. Overall, this comparison highlights that the quantum model is better suited for this task within a 600-episode training limit, likely due to the unique properties of its architecture, which may facilitate higher exploration within a constrained range. Quantum models inherently leverage quantum superposition and entanglement, allowing them to explore a broader solution space more effectively. This increased exploration potential enables the quantum model to find and stabilize around higher reward values more efficiently than the classical model within the same episode limit. Despite attempts to enhance the classical model with increased batch size and network complexity, it does not reach comparable performance within the same episode count. The quantum model therefore demonstrates a more efficient and stable learning process under the given conditions. It is also important to note that, even with enhanced parameters, the classical algorithm requires significantly less computational time compared to the quantum approach, with a time difference of approximately one hour. This suggests that while the quantum model achieves superior reward stability and exploration efficiency within the 600-episode limit, it comes at a cost of increased computational time—a limitation that could be addressed by running the code on actual quantum hardware.

7. Conclusions

The primary objective of this thesis work was to explore whether quantum computing could enhance the performance of a reinforcement learning algorithm by leveraging the unique properties of qubits to accelerate the agent's learning and understanding of its environment. To achieve this, a comparison was conducted between a classical RL algorithm using a neural network and a quantum RL algorithm employing Parametrized Quantum Circuits (PQCs). The latter approach used a hybrid system, where computation was split between classical and quantum processors. Although IBM's quantum hardware is accessible through the IBM Quantum backend, this study used simulated quantum circuits to avoid the limitations of small-scale quantum systems that could impact the fidelity of quantum algorithms. The implementations for both algorithms were written in Python, with support from TensorFlow and TensorFlow-Quantum libraries, allowing flexible adjustments of key hyperparameters for a direct comparison. The hyperparameters adjusted were those common to both the classical and quantum models: batch size, learning rate, and the architectures of the learning component (Neural Network and PQC).

The evaluation was conducted in the OpenAI Gym environment known as "Acrobot-v1," a control problem requiring optimized torque application to generate sufficient momentum for achieving a specific configuration. Performance was measured by the rewards obtained during training and the stability of the training process. Due to the significant difference in runtime for a single episode between the two algorithms, the classical RL was run over 20,000 episodes, while the quantum RL was evaluated over 600 episodes.

For the classical RL algorithm, the optimal parameters were found to be a batch size of 64, which balanced stability and runtime, and a learning rate of $1 \cdot 10^{-5}$, which facilitated stable convergence. The neural network architecture that achieved the best results consisted of three hidden layers, each with 256 neurons. This configuration proved to be the only one capable of yielding both convergence and stability. With these settings, the classical agent's policy converged after approximately 6,500 episodes, achieving an average reward of about -80 ± 10.

For the quantum RL approach, the batch size was set to 32 to manage the higher computational demands of quantum circuit simulations. The learning rate was set to the TensorFlow-recommended value of $1 \cdot 10^{-4}$, as this value offered the best balance of variability and stability, while higher or lower values led to instability. The PQC architecture that achieved the most stable learning policy comprised 8 variational layers, as increasing the layer count to 10 or 14 introduced overparameterization issues, causing the agent to overfit minor policy variations, resulting in unstable learning. With these parameters, the quantum agent reached convergence after approximately 500 episodes, achieving an average reward of around -175 ± 35. This policy was less stable than the classical one.

The classical algorithm demonstrated superior performance in terms of stability and optimal policy. However, due to the extended runtime required for quantum simulations, the quantum RL was run over fewer episodes. To facilitate a more direct comparison, the classical neural network was configured to attempt convergence within the same 600 episodes allocated to the quantum RL. This required a batch size of 1,000 and an increase to 600 neurons in each of the three hidden layers, the largest configuration found that did not cause overparameterization. Under this constraint, the classical model's performance declined, with the quantum model outperforming it over the limited episodes. These findings suggest that a quantum RL model, when executed on an actual quantum processor (thereby reducing simulation overhead), might outperform its classical counterpart given a larger number of episodes. Doing this, however, introduces new problems such as decoherence, noise and gate errors that were neglected in the simulations.

The next phase of this research will involve implementing the algorithm on IBM's quantum hardware, enabling a more extensive evaluation over a greater number of episodes. Running the quantum RL algorithm on real quantum devices will allow an examination of the effects of the hardware limitations listed above on algorithm performance. Observing the challenges and strengths of the quantum model under real-world conditions will help improve both quantum RL algorithms and quantum hardware, advancing the field toward practical and reliable quantum computing solutions.

Bibliography

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2020. URL: `http://incompleteideas.net/book/RLbook2020.pdf`.

[2] G. Carleo and M. Troyer. “Solving the quantum many-body problem with artificial neural networks”. In: *Science* (2017). URL: `https://www.science.org/doi/full/10.1126/science.aag2302?casa_token=xOOONq03kPQAAAAA%3AfvDFmC6djU_4LKkdEECAyB4x-PQ_DrvJWgAnUtrg8DEVZHhCBeTTGQiB-vDmxwgWa1FCa9FOXu6e3w`.

[3] T. Fösel et al. “Reinforcement Learning with Neural Networks for Quantum Feedback”. In: (2018). URL: `https://doi.org/10.1103/PhysRevX.8.031084`.

[4] M. Bukov et al. “Reinforcement Learning in Different Phases of Quantum Control”. In: *Phys. Rev. X* 8.3 (2018), p. 031086. URL: `https://doi.org/10.1103/PhysRevX.8.031086`.

[5] P. Garnier et al. “A review on deep reinforcement learning for fluid mechanics”. In: *Comput. Fluids* 225 (2021), p. 104973. URL: `https://doi.org/10.1016/j.compfluid.2020.104973`.

[6] J. Nousiainen et al. “Adaptive optics control using model-based reinforcement learning”. In: *Opt. Express* 29 (2021), pp. 15327–15344. URL: `https://doi.org/10.1364/OE.415116`.

[7] C. Beeler et al. “Optimizing thermodynamic trajectories using evolutionary and gradient-based reinforcement learning”. In: *arXiv* (2019). URL: `https://arxiv.org/abs/1903.08453`.

[8] Yunseok Kwak et al. “Introduction to quantum reinforcement learning: Theory and pennylane-based implementation”. In: *2021 international conference on information and communication technology convergence (ICTC)*. IEEE. 2021, pp. 416–420.

[9] Daoyi Dong et al. “Quantum Reinforcement Learning”. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 38.5 (2008), pp. 1207–1220. DOI: `10.1109/TSMCB.2008.925743`.

[10] Rohan Jagtap. *Understanding the Markov Decision Process (MDP)*. Accessed: 2024-11-07. 2024. URL: `https://builtin.com/machine-learning/markov-decision-process`.

[11] Ketan Doshi. *Reinforcement Learning Made Simple (Part 2) Solution Approaches*. 2020. URL: `https://towardsdatascience.com/reinforcement-learning-made-simple-part-2-solution-approaches-7e37cbf2334e`.

[12] Andrea Skolik, Sofiene Jerbi, and Vedran Dunjko. "Quantum agents in the Gym: a variational quantum algorithm for deep Q-learning". In: *Quantum* 6 (May 2022), p. 720. ISSN: 2521-327X.

[13] Unknown. *Diagram of DQN Architecture*. 2024. URL: `https://wikidocs.net/174536`.

[14] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[15] Facundo Bre, Juan Gimenez, and Víctor Fachinotti. "Prediction of wind pressure coefficients on building surfaces using Artificial Neural Networks". In: *Energy and Buildings* 158 (Nov. 2017). DOI: `10.1016/j.enbuild.2017.11.045`.

[16] Anders Krogh. "What are artificial neural networks?" In: *Nature biotechnology* 26.2 (2008), pp. 195–197.

[17] Markel Sanz. *Introduction to Reinforcement Learning: Part 3 - Q-learning with Neural Networks (Algorithm: DQN)*. Accessed: 2023-10-17. 2021. URL: `https://markelsanz14.medium.com/introduction-to-reinforcement-learning-part-3-q-learning-with-neural-networks-algorithm-dqn-1e22ee928ecd`.

[18] Towards Data Science. *Reinforcement Learning Explained Visually — Part 5: Deep Q-Networks Step by Step*. Accessed: 2023-10-17. 2021. URL: `https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197f4b`.

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

[20] Wenjie Liu et al. "A Novel Quantum Visual Secret Sharing Scheme". In: *IEEE Access* PP (July 2019), pp. 1–1. DOI: `10.1109/ACCESS.2019.2931073`.

[21] Rodrigo Coelho, André Sequeira, and Luís Paulo Santos. “VQC-based reinforcement learning with data re-uploading: performance and trainability”. In: *Quantum Machine Intelligence* 6.2 (Aug. 2024), p. 53. ISSN: 2524-4914. DOI: 10.1007/s42484-024-00190-z. URL: https://doi.org/10.1007/s42484-024-00190-z.

[22] TensorFlow Team. *Quantum Reinforcement Learning*. Accessed: 2024-10-27. 2021. URL: https://www.tensorflow.org/quantum/tutorials/quantum_reinforcement_learning.

[23] Analytics Vidhya. *Acrobot with Deep Q-Learning*. Accessed: 2024-11-07. 2021. URL: https://www.analyticsvidhya.com/blog/2021/06/acrobot-with-deep-q-learning/.

[24] Frank C. Enendu. *Implement a Deep Reinforcement Learning Method for an Acrobot Game*. Accessed: 2024-11-07. 2021. URL: https://medium.com/@enendufrankc/implement-a-deep-reinforcement-learning-method-for-an-acrobot-game-f1f71b1aead2.

[25] OpenAI. *Acrobot Environment*. 2022. URL: https://www.gymlibrary.dev/environments/classic_control/acrobot/.

[26] TensorFlow. *TensorFlow Agents: C51 Tutorial*. Accessed: 2024-10-31. 2023. URL: https://www.tensorflow.org/agents/tutorials/9_c51_tutorial.

[27] TensorFlow Quantum. *Quantum Reinforcement Learning*. Accessed: 2024-10-31. 2024. URL: https://www.tensorflow.org/quantum/tutorials/quantum_reinforcement_learning.

Ringraziamenti

Desidero ringraziare prima di tutto i miei amici, inseparabili compagni di viaggio che mi hanno affiancato e reso leggeri i momenti più difficili. Un ringraziamento speciale è d'obbligo per te, Gianne. Grazie per esserci sempre stato da quando ho ricordi, per aver vissuto con me momenti di serenità e per non avermi abbandonato nei momenti difficili. Ringrazio profondamente i miei compagni di avventura. Vi ringrazio, matematici Luche e Anto, per avermi fatto passare analisi e per tutti gli aiuti e le risate che mi avete donato. Ti ringazio, Tommi, per tutti quei momenti di spensieratezza che mi hai sempre concesso e per avermi sempre accolto in auletta quando avevo bisogo di sdraiarmi (scusa per il divano eheh). Vi ringrazio, Tommy e Rocco, per aver preso parte e reso possibile il trio, per avermi accompagnato negli alti e nei bassi. Tommy, con la tua profondità mi hai fatto crescere molto, e con la tua ignoranza mi hai fatto divertire altrettanto. Rocco, sei quella persona che tutti dovrebbero avere nella propria vita, quella persona che, quando gli dici che non hai ancora iniziato a studiare, ti guarda negli occhi con una serenità pietrificante e ti dice: "neanche io, chill, partitina a Clash?". Grazie, Ale, per essere come sei, per aver reso questo mio percorso più leggero ma allo stesso tempo per avermi dato una svegliata nei momenti in cui più mi siedevo sugli allori. Non pensavo di trovare una persona con una energia così tanto compatibile con la mia. Infine, ultima ma non per importanza (forse), ti ringrazio Emma, per tutto quello che hai fatto per me, tutti gli appunti passati, tutti i momenti, le ore di studio e le risate condivise. Ti ringrazio per le ubriacate, il divano rotto, le partite a biliardo (alle quali ho sempre vinto io), i passaggi e le conversazioni profonde nel Toyoto. Sei stata la cosa più bella che mi sia capitata in questi 3 anni. Ringrazio la mia famiglia, per avermi sempre supportato in ogni scelta, per avermi sempre spianato la strada ed accompagnato mano nella mano. Giazie Zio Walter, porto sicuro a cui attraccare quando più ne avevo bisogno. Grazie, Emma, per essere sempre stata un punto di rifermento e per avermi insegnato ad avere coraggio. Grazie Mamma, per avermi dato le ali, per avermi lasciato percorrere la mia strada e vivere le mie passioni anche quando non ne le capivi pienamente, per tutti i sacrifici che fai da una vita intera. E infine, grazie papà, per avermi instillato la passione per la fisica, la curiosità di conoscere il mondo, la volontà di non soffermarmi ai dettagli superficiali delle cose. Ti ringrazio per la pazienza che hai sempre avuto e per la fiducia che mi hai sempre dato. Spero tu sia fiero di me.