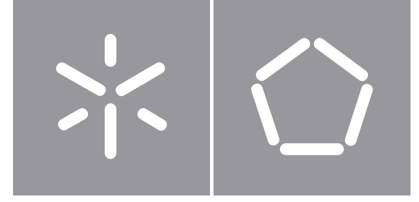
University of Minho
School of Engineering

Rodrigo da Silva Gomes Peres Coelho

Tradeoff between moving targets, gradient magnitude and performance in quantum variational Q-Learning

september 2023



University of Minho
School of Engineering

Rodrigo da Silva Gomes Peres Coelho

Tradeoff between moving targets, gradient magnitude and performance in quantum variational Q-Learning

Masters Dissertation
Master's in Engineering Physics

Dissertation supervised by
Luís Paulo Santos
André Sequeira

september 2023

Copyright and Terms of Use for Third Party Work

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

License granted to users of this work:

CC BY

https://creativecommons.org/licenses/by/4.0/

Acknowledgements

Writing this thesis has been a significant endeavor, and its completion would not have been possible without the assistance and support of many individuals. I'd like to take a moment to express my gratitude to those who have helped me throughout this process.

I would first like to thank my supervisor, Professor Luís Paulo Santos, for providing guidance, feedback and support throughout this year. I am also extremely grateful to my co-supervisor André Sequeira, who was always available to discuss my research and to guide me in the correct direction. This work could not have been complete without your help. On this note, thank you for taking me under your wing on my first conference, which is always a scary experience.

I would also like to thank my family, my parents, my brothers, my girlfriend and my friends. Thank you for the laughs and the good memories, but also for the continued support, especially during the hardships. I am sorry for sometimes pestering you with quantum computing and machine learning. Needless to say, I would not be here without each and every single one of you.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020. I also thank the support of the Foundation for Science and Technology (FCT, Portugal) under grant 10053/BII-E_B4/2023.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, september 2023

Rodrigo da Silva Gomes Peres Coelho

Abstract

Reinforcement Learning (RL) consists of designing agents that make intelligent decisions without human supervision. When used alongside function approximators such as Neural Networks (NNs), RL is capable of solving extremely complex problems. Deep Q-Learning, a RL algorithm that uses Deep NNs, even achieved super-human performance in some specific tasks. Nonetheless, it is also possible to use Variational Quantum Circuits (VQCs) as function approximators in RL algorithms. This work empirically studies the performance and trainability of such VQC-based Deep Q-Learning models in OpenAI's gym CartPole-v0 and Acrobot-v1 environments. More specifically, we research how data re-uploading affects both these metrics. We show that the magnitude and the variance of the gradients of these models remain substantial throughout training due to the moving targets of Deep Q-Learning. Moreover, we show that increasing the number of qubits does not lead to a decrease in the magnitude and variance of the gradients, unlike what was expected due to the Barren Plateau Phenomenon. This hints at the possibility of VQCs being specially adequate for being used as function approximators in such a context. We also use the Universal Quantum Classifier as a function approximator in VQC-based Deep Q-Learning and implement VQC-based models capable of achieving considerable performance in the Acrobot-v1 environment, a previously untapped environment for VQCs.

Keywords Reinforcement Learning, Quantum Computing, Variational Quantum Circuits, Neural Networks

Resumo

Reinforcement Learning (RL) consiste em projetar agentes que tomam decisões inteligentes sem supervisão humana. Quando usado em conjunto com aproximadores de funções, como Redes Neuronais (RNs), RL é capaz de resolver problemas extremamente complexos. Deep Q-Learning é um algoritmo de RL que usa RNs profundas e que alcançou um desempenho super-humano em algumas tarefas específicas. No entanto, também é possível utilizar Circuitos Variacionais Quânticos (VQCs) como aproximadores de funções em algoritmos de RL. Este trabalho estuda empiricamente o desempenho e a treinabilidade de tais modelos de Deep Q-Learning baseados em VQC nos ambientes CartPole-v0 e Acrobot-v1 do OpenAI gym. Mais especificamente, investigamos como o data re-uploading afeta ambas estas métricas. Demonstramos que a magnitude e a variância dos gradientes destes modelos permanecem substanciais ao longo do treino devido aos alvos móveis do Deep Q-Learning. Além disso, mostramos que aumentar o número de qubits não leva a uma diminuição na magnitude e variância dos gradientes, contrariamente ao que era esperado devido ao Barren Plateau Phenomenon. Isto sugere a possibilidade dos VQCs serem especialmente adequados para serem usados como aproximadores de funções neste contexto. Também utilizamos o Universal Quantum Classifier como um aproximador de funções em Deep Q-Learning e implementamos modelos baseados em VQC capazes de alcançar um desempenho considerável no ambiente Acrobot-v1, um ambiente anteriormente inexplorado para VQCs.

Palavras-chave Reinforcement Learning, Computação Quântica, Circuitos Variacionais Quânticos, Redes Neuronais

Contents

I Introductory material 1

1 Introduction 2

1.1 Context 2
1.2 Motivation 4
1.3 Contributions 5
1.4 Outline 6

2 Reinforcement Learning 8

2.1 The Reinforcement Learning Problem Statement 8
2.2 State and Value Functions and the Notion of Optimality 12
2.3 Bellman Equations and Dynamic Programming 14
2.4 Q-Learning 16
2.5 Deep Reinforcement Learning 19
2.6 Deep Q-Learning 21
2.7 Policy-Gradient Algorithms and Taxonomy of RL 24
2.8 Summary 25

3 Quantum Computing 27

3.1 Context 27
3.2 Quantum Circuits 28
3.2.1 Single-Qubit and Multi-Qubit Systems 28
3.2.2 Single-Qubit and Multi-Qubit Quantum Gates 30
3.2.3 Measurements 32
3.3 Variational Quantum Circuits 32
3.3.1 Data Encoding 33

3.3.2 Ansatz 34
3.3.3 Cost Function 35
3.3.4 Barren Plateau Phenomenon 36
3.4 Data Re-Uploading 36
3.5 VQC-Based Reinforcement Learning 37
3.5.1 VQC-Based Q-Learning 39
3.5.2 VQC-Based Policy-Gradients 43
3.6 Summary 44

II Core of the Dissertation **45**

4 The Problem and Methodology **46**
4.1 The Problem 46
4.2 The Algorithm Implementation Choices 47
4.3 Benchmark Environments 48
4.3.1 CartPole-v0 48
4.3.2 Acrobot-v1 49
4.4 Methodologies for Analysing Model's Performance and Trainability 51
4.4.1 Performance of a Model 51
4.4.2 Trainability of a Model 53
4.5 Summary 53

5 Results and Discussion **55**
5.1 Replication of Skolik et al. [2022]'s results 55
5.2 Performance of the Universal Quantum Classifier 59
5.2.1 The Single-Qubit Universal Quantum Classifier 60
5.2.2 The Multi-qubit Universal Quantum Classifier 63
5.3 Trainability Analysis of Skolik et al. [2022]'s models 67
5.4 Trainability Analysis of the UQC 70
5.5 Tradeoff between Moving Targets and Gradient Magnitude 72
5.6 Gradient Behavior for Increasing System Sizes 77
5.7 The Acrobot Environment 79
5.8 Summary 82

6 Conclusions and future work 84
6.1 Conclusions 84
6.2 Prospect for future work 85

III Appendices 93

A Models' Hyperparameters 94
A.1 Hyperparameters' explanation 94
A.2 Hyperparameters used throughout this work 94

List of Figures

1 The Agent-Environment Interface: The agent interacts with the environment at time step t by taking action A_t. The environment then changes to state S_{t+1} and produces reward R_{t+1}, which are both passed back to the agent so that it can decide the next action. The dotted lines indicate that this process repeats itself. Inspired by Sutton and Barto [2018]. 9

2 An example of an MDP - a grid world in which the goal of the agent is to find the shortest path to the apple while not bumping into the bombs. 11

3 A neural network is composed of neurons, which are grouped into layers. Each neuron receives m inputs $\{x_1, ..., x_m\}$, which are multiplied by the m respective weights $\{w_1, ..., w_m\}$ and summed. A bias b is also added. Finally, the result is passed through an activation function φ, which produces the output. 20

4 A comparison between Q-Learning and Deep Q-Learning (DQN). Q-Learning is a tabular approach, where the values of all the visited state-action pairs are stored in a lookup table. DQN, on the other hand, uses a deep neural network as the Q-function approximator. The Figure shows one of the possible architectures for the NN, where the input layer receives the components of the state and the output layer outputs the Q-values for all possible actions. See Morales [2020] for other possible architectures. 22

5 A non-exhaustive taxonomy of the RL algorithms mentioned in this chapter. 25

6 Representation of a Bloch Sphere 29

7 The building blocks of a VQC and how it is trained. In a typical VQC, data is encoded, then processed by a parameterized unitary and, finally, the expectation value of some observable is measured. Then, a cost function that depends on the expectation value is calculated and a classical optimizer updates the parameters $\boldsymbol{\theta}$. 33

8 Example of a hardware-efficient VQC. $|x\rangle$ is processed by layers of unitaries, each of them composed of parameterized single-qubit gates followed by a cascade of entangling gates. . . . 34

9 The Data Re-Uploading technique in action. The same data encoding block $S(x)$ is repeated several times throughout the quantum circuit to increase the expressivity of the quantum model. . . . 37

10 Deep RL versus VQC-based RL. The image on the left represents Deep RL, where Deep Neural Networks are used as function approximators for either value functions or policies. Similarly, the image on the right represents VQC-based RL, where VQCs are used for the same effect. The algorithms themselves are identical, with the exception of the model used as a function approximator. . . . 38

11 Architectures used in Chen et al. [2020] (Subfigure 11a), Lockwood and Si [2020] (Subfigure 11b) and Skolik et al. [2022] (Subfigure 11c) for $n = 4$ qubits, L layers, parameters θ and input $x = [x_0, x_1, x_2, x_3]$. The data encoding gates are green-coloured and compose the blocks $S(x)$. The variational gates are blue-coloured and compose the blocks $W(\theta^l)$, where l is the layer, along with the entangling gates. When Data Re-Uploading is used, as in Subfigure 11c, a layer is composed of the data encoding and variational blocks $U(x, \theta^l)$. . . . 40

12 The CartPole Environment . . . 48

13 The Acrobot Environment . . . 49

14 Both figures represent the performance of two random models on the CartPole-v0 environment. The thick lines are the mean returns for each model (in this case, 5 agents were initialized per model) and the shaded areas the standard deviation. This is how the performance of all models will be plotted. . . . 52

15 The functioning of the VQC-based Deep Q-Learning algorithm used by Skolik et al. [2022] to solve the CartPole environment. When Data Re-Uploading is used, $U(s, \lambda, \theta)$ is repeated several times. Otherwise, just $W(\theta)$ is repeated. . . . 57

16 Comparison of baseline (Subfigure 16a) and data re-uploading (Subfigure 16b) models with and without trainable input and output scaling in the CartPole-v0 environment. The optimal set of hyperparameters from Skolik et al. [2022] was used, see Table 6. The thick lines are the average return over all the 10 agents for each model, while the shaded areas indicate the standard deviation of the return over all agents. If an agent solves the environment (average reward over the last 100 episodes ≥ 195), training is stopped. . . 58

17 Analysis of the performance of the single-qubit UQC and Skolik data re-uploading models in the CartPole-v0 environment following the methodology defined in Section 4.4.1. 10 agents were initialized from each model. The full set of hyperparameters can be seen in Table 7. 62

18 Performance analysis of the two-qubit (see Subfigure 18a) and four-qubit (see Subfigure 18b) UQCs using the Partial and Full encoding techniques without entanglement. 10 agents were initialized from each model. The full set of hyperparameters can be seen in Table 8. 64

19 Performance analysis of the two-qubit (see Subfigure 18a) and four-qubit (see Subfigure 18b) UQCs using the Partial and Full encoding techniques with and without entanglement. 10 agents were initialized from each model. The full set of hyperparameters can be seen in Table 8. 66

20 Performance Analysis of the best-performing models in the CartPole-v0 environment. In concrete, the Skolik Data Re-Uploading model, the Single-Qubit UQC and the Full Encoding Multi-Qubit UQC without entanglement. 10 agents were initialized from each model. 67

21 Comparison of the gradients of the baseline and data re-uploading models with and without trainable input scaling in the CartPole environment. Subfigure 21a shows the mean norm of the vector gradient throughout training and Subfigure 21b the variace in the norms of the vector gradients throughout training. The optimal set of hyperparameters from Skolik et al. [2022] is used, see Table 6. Moreover, 10 agents are initialized from each model. If an agent solves the environment, training is stopped. These metrics are stopped after the first agent solves the environment, hence why some curves are shorter than others. 68

22 Return and Norm of the gradients for 3 random Baseline (see Subfigure 22a) and Data Re-Uploading (see Subfigure 22b) agents. Both the return an the norm of the agents are presented as moving averages to mitigate noise due to the unstable nature of DQN. . . . 69
23 Trainability analysis of the single-qubit UQC and Skolik data re-uploading models in the CartPole-v0 environment from Figure 17 following the methodology defined in Section 4.4.2. Subfigure 23a shows the mean norm of the vector gradient throughout training and Subfigure 23b the variance in the norms of the vector gradients throughout training. 10 agents were initialized from each model. 70
24 Trainability analysis of the two-qubit (see Subfigure 24a) and four-qubit (see Subfigure 24b) UQCs using the Partial and Full encoding techniques with and without entanglement. 10 agents were initialized from each model. The full set of hyperparameters can be seen in Table 8. 71
25 The mean loss across the 10 agents of the data re-uploading model. The loss is only shown until the first agent solves the environment. 73
26 Analysis of the performance and the loss of the data re-uploading model with different values of C. The other hyperparameters are constant. To analyse the performance and the loss function, 5 agents were initialized from each model. Then, the performance was measured following the methodology from 4.4.1 and the loss function was analyzed by computing the mean loss function over all the agents. The full set of hyperparameters is shown in Table 9. 75
27 Analysis of the gradients of the data re-uploading model with different values of C. The other hyperparameters are constant and 5 agents were initialized from each model. Then, the gradient analysis was performed following the methodology from 4.4.2. 76
28 Performance and Trainability analysis of the multi-qubit UQCs as the number of qubits increase. 10 agents were initialized from each model. The full set of hyperparameters can be seen in Table 8. 78
29 Performance and Trainability analysis of the Skolik Data Re-Uploading and three-qubit UQC models on the Acrobot environment. 10 agents were initialized from each model. The full set of hyperparameters can be seen in Table 10. 81

List of Tables

1 Cartpole's state space . . . 49

2 Acrobot's action-space . . . 50

3 Acrobot's state-space . . . 50

4 Chosen observables for the two models tested in the Acrobot-v1 environment. . . . 80

5 An explanation of VQC-Based Deep Q-Learning's hyperparameters . . . 94

6 Models' hyperpameters from Figure 16 . . . 95

7 Models' hyperpameters from Figure 17 . . . 95

8 Models' hyperpameters from Figures 19 and 28. . . . 96

9 Models' hyperpameters from Figure 26 . . . 96

10 Models' hyperpameters from Figure 29. . . . 97

Part I

Introductory material

Chapter 1

Introduction

1.1 Context

Intelligence has played a pivotal role in evolution. Loosely defined, intelligence is the ability to process information and make decisions based on that information. Humans, in particular, possess unique cognitive abilities that set them apart from all other animals. These abilities enable technological evolution, complex communication, environmental manipulation, and music, among many others. Consequently, a question arose - Can machines replicate these capabilities if humans build them? Alan Turing posed this question in the 1940s in a public conference and, in 1950, published the seminal work Turing [1950], which started the development of the research area known as *Artificial Intelligence (AI)*. AI is the research of machines performing tasks typically associated with human intelligence. Since then, the trajectory of AI has featured waves of optimistic predictions, followed by disillusionment, and then a resurgence of hope. However, the past decade has seen remarkable advancements in AI applications across diverse domains, such as medicine Hamet and Tremblay [2017], education Holmes et al. [2023], finance Bahrammirzaee [2010], and engineering Salehi and Burgueño [2018]. We currently find ourselves amidst a phase of palpable optimism, where investment in AI has reached unprecedented levels Mou [2019]. Nonetheless, some skepticism persists, with many anticipating a new wave of disillusionment after encountering some unforeseen barrier. However, widely recognized that AI offers numerous practical applications. Although the full extent of its impact remains uncertain, AI undoubtedly influences and will keep influencing the world in significant ways.

At the core of AI lies the pursuit of designing agents that learn how to make informed decisions and self-correct in possibly ever-changing environments without human supervision Russell and Norvig [2010]. Reinforcement Learning (RL) represents a significant stride toward realizing this vision. A RL agent learns from interaction with its surrounding environment in a feedback loop, relying solely on a reward signal to guide its actions. The concept of learning from interaction forms the bedrock of nearly all theories of

learning and intelligence Sutton and Barto [2018], with some even conjecturing that rewards alone can pave the path to achieving general AI agents Silver et al. [2021]. However, RL agents face a problem unparalleled in other AI frameworks: acting in unknown environments resorts to discovering the optimal balance between exploration and exploitation, also known as the exploration-exploitation tradeoff Kaelbling et al. [1996]. RL techniques, successful in the past, found their use in low-dimensional problems because of scalability limitations. This restriction meant RL couldn't address real-world challenges, which often involve complex and high-dimensional domains. Yet, recent advancements in RL, powered by Deep Neural Networks (DNNs) as potent function approximators LeCun et al. [2015], have changed the landscape. These algorithms now match or even exceed the performance levels of human experts in tasks like Chess Silver et al. [2017], Go Silver et al. [2016], video-games Mnih et al. [2015], and communications and networking Luong et al. [2019]. Nevertheless, state-of-the-art algorithms are extremely data-hungry and often not efficiently learnable in the language of statistical learning theory Agarwal et al. [2019].

Given the significant potential impact of RL and AI on the future, a continuous effort exists to improve and refine AI algorithms and systems. This momentum stems from the realization that, despite recent advancements, AI has yet to achieve its full potential. Therefore, besides just enhancing and optimizing existing algorithms, exploring new methods and emerging technologies remains essential. One such technology is quantum computing.

Richard Feynman introduced the concept of Quantum Computers Feynman [1982] that operate based on quantum physics principles, leveraging quantum phenomena like superposition and entanglement to manipulate quantum data. Nonetheless, it is also possible to use quantum computers to process classical data as long as it is encoded into a quantum state. Remarkably, quantum computers, in theory, offer a means to efficiently solve some problems that lack efficient classical algorithms. One such example is Shor's Algorithm for finding the prime factors of an integer, which is exponentially faster than the best-known classical algorithm and holds the promise of breaking public-key cryptography schemes given a quantum computer with a sufficient number of qubits Shor [1999]. Therefore, one might speculate that quantum computers may outperform classical computers in machine learning tasks, even though the immediate advantages remain somewhat uncertain Biamonte et al. [2017]. Nevertheless, ongoing research actively explores the potential applications of near-term quantum computers in machine learning tasks, primarily focused on seeking advantages in practical, real-world problems.

Having listed both the potential of Artificial Intelligence and Quantum Computing, and considering how both technologies can be combined, the importance of exploring their intersection becomes obvious. That is what this dissertation seeks to do. Specifically, the main goal is to research the field of variational

quantum circuits and their application to reinforcement learning problems.

1.2 Motivation

Quantum computers yield complexity advantages over classical computers in some problems, assuming enough qubits to implement error-correction codes to mitigate noise and errors due to decoherence Preskill [1998]. Nonetheless, in near-term devices or Noisy Intermediate-Scale Quantum (NISQ) devices, the algorithms that yield these advantages can not be implemented successfully to solve real-world problems Preskill [2018]. The challenge thus revolves around devising quantum algorithms that are resource-friendly and resilient to errors to harness near-term quantum computation's potential.

Variational Quantum Circuits (VQCs) are especially suitable for NISQ devices, given their lower demands in the number of qubits and circuit depth, which mitigates the effect of noise Cerezo et al. [2021a]. Consequently, integrating VQCs into machine learning algorithms has become an intriguing field of study. As delineated by Schuld et al. [2021], VQCs can be seen as Partial Fourier Series in the data, where the data encoding gates define the frequency spectrum. The authors also introduce a technique known as Data Re-Uploading, which consists of repeating the data encoding gates several times throughout the circuit, effectively increasing the frequency spectrum accessible to the quantum model. This subsequently broadens the scope of functions the model can approximate, rendering it more expressive. However, such an increase in circuit depth is not without its trade-offs. On the one hand, noise escalation in actual quantum hardware due to circuit depth could nullify the effectiveness of algorithms employing this technique in NISQ devices. On the other hand, the Barren Plateau Phenomenon states that the partial derivatives of a hardware-efficient VQC vanish exponentially with the number of qubits and layers McClean et al. [2018]. Thus, a balance between expressivity and trainability becomes essential. Given that Data Re-Uploading enhances the expressivity of the quantum model through increased circuit depth, it could potentially make the model more challenging to train.

Finally, VQCs as function approximators in RL problems have already been researched and proven to work for two different main algorithms: *REINFORCE* Sequeira et al. [2022], Jerbi et al. [2021] and *Q-learning* Chen et al. [2020], Lockwood and Si [2020], Skolik et al. [2022]. In all of these articles, VQCs were used as function approximators and managed to solve benchmark environments. However, none of them research the effect of Data Re-Uploading on the trainability of the RL models. Hence, that is the goal of this work. The focus is solely on Q-learning for several reasons. First, Q-Learning stands out in the RL domain because of recent accomplishments in addressing complex problems using this method Mnih et al.

[2015]. Furthermore, while the trainability of quantum models based on the REINFORCE algorithm has been researched, such as in Sequeira et al. [2022] and Jerbi et al. [2021], literature on models grounded on Q-learning is nonexistent. This gap underscores the need for the proposed research.

1.3 Contributions

The first contribution is the re-utilization of the methodology developed in Skolik et al. [2022] to analyze VQC-based Deep Q-Learning models' performance in benchmark environments. More specifically, using this methodology, the effect of data re-uploading and trainable input/output scaling on models' performance in the CartPole-V0 environment is empirically verified, reaffirming the importance of such techniques.

Furthermore, a methodology for analysing models' trainability throughout training in benchmark environments is developed. It consists of observing the gradients' norm and its variance throughout training of agents initialized from said model in the benchmark environment. It is verified that the gradients' norm and its variance remain substantial throughout training. Moreover, more expressive models appear to have higher gradients, which is surprising given the expressivity-trainability tradeoff demonstrated in Holmes et al. [2022].

Moreover, the tradeoff between the moving targets inherent to the Deep Q-Learning algorithm and the magnitude of the gradients is empirically studied in the CartPole-V0 environment.

The single-qubit Universal Quantum Classifier (UQC) from Pérez-Salinas et al. [2020] is adapted to be used as a function approximator in Deep Q-Learning and its performance and trainability are tested on the CartPole-v0 environment. Then, the multi-qubit version of the architecture is also tested using the same methods.

Finally, we developed VQC-based Deep Q-Learning models capable of achieving a considerable performance (according to our methodology) on the more complex Acrobot-v1 environment, a previously untouched environment for such models, to the best of our knowledge.

Some of the contributions of this work were accepted as posters at two different prestigious international conferences:

- Theory of Quantum Computation, Communication and Cryptography (TQC) - Data Re-Uploading in Quantum Variational Q-Learning, July 2023
- Quantum Techniques in Machine Learning (QTML) - Tradeoff between moving targets, gradient magnitude and performance in quantum variational Q-Learning, November 2023

1.4 Outline

As previously mentioned, this work's goal is to research the effect of Data Re-Uploading on the performance and trainability of VQC-based Deep Q-Learning models. To accomplish that goal, this dissertation is organized as follows.

Chapter 2 introduces Reinforcement Learning (RL), a type of machine learning in which an agent learns to make intelligent decisions by interacting with the surrounding environment in a feedback loop, a process that resembles trial-and-error learning. Moreover, this chapter also introduces the mathematical framework behind RL problems and some common methods for solving such problems. Furthermore, it introduces function approximation as a means to solve more complex RL problems, with a special emphasis on Deep Neural Networks (DNNs) and Deep RL. Finally, this chapter also introduces the Deep Q-Learning algorithm, which is the most important algorithm for this work.

Chapter 3 explains the basic concepts of the circuit model of quantum computation, before delving into Variational Quantum Circuits (VQCs) and some related concepts, such as the Barren Plateau Phenomenon and the Data Re-Uploading technique, which will be relevant throughout this work. It then introduces VQC-based RL, where VQCs are used as function approximators in Deep RL problems instead of the typical DNNs. Finally, this chapter reviews some recent papers on VQC-based RL, with a special emphasis on VQC-based Deep Q-Learning.

Afterward, Chapter 4 defines the problem this work attempts to research, using the knowledge acquired in the previous chapters. Then, it introduces the benchmark environments that will be used to test the VQC-based models and the methodologies that will be used throughout this work to analyze the models' performance and trainability.

Chapter 5 shows and discusses the empirical results obtained throughout the work. It first starts by analyzing the performance of the models from Skolik et al. [2022] in the CartPole-v0 environment with and without data re-uploading and trainable input/output scaling and discussing the importance of each of these techniques in the models' performance. Then, the single-qubit and multi-qubit Universal Quantum Classifiers are introduced and their performance is also analyzed in the same environment. Furthermore, the effect of entanglement on performance is discussed. After that, we shift our focus to the trainability of the models. We first start with the models from Skolik et al. [2022] and then repeat the procedure for the Universal Quantum Classifier. Subsequently, we show the effect that the moving targets of Deep Q-Learning have on the trainability of these models. Then, we study how the models' trainability behaves as the number of qubits increases. Finally, the performance and trainability of these models are analyzed in

the Acrobot-v1 environment, a previously untapped environment for VQC-based Deep Q-Learning models.

Chapter 2

Reinforcement Learning

The main goal of AI involves designing agents that make intelligent decisions without human supervision. Reinforcement Learning (RL) agents achieve this by interacting with their surrounding environment in a feedback loop. This learning method resembles the trial-and-error process seen in humans and other animals, allowing RL agents to address various real-world problems and identify practical solutions. This chapter explains RL based on Sutton and Barto [2018].

2.1 The Reinforcement Learning Problem Statement

Any RL problem has two primary constituents: the Agent and the Environment. The agent is the decision-maker and the representation of that which learns. The environment is everything that surrounds the agent and is affected by its decisions. The agent must learn which actions to take by interacting with the environment and receiving a reward that indicates the goodness of that particular action concerning the goal. This process is called trial and error search and is one of the defining characteristics of RL. In addition, important to note that actions are taken in a sequence and can have an impact not only on immediate rewards but also on all future rewards. This is commonly referred to as *delayed reward*.

Going into further detail, the agent interacts with the environment by taking *actions* at each moment t. The environment receives these actions and produces a *reward* and a new *state* at the next moment in time $t+1$. The agent then receives the new state and the reward to determine the following action. This iterative process forms what is termed *The Agent-Environment Interface* Sutton and Barto [2018], illustrated in Figure 1.

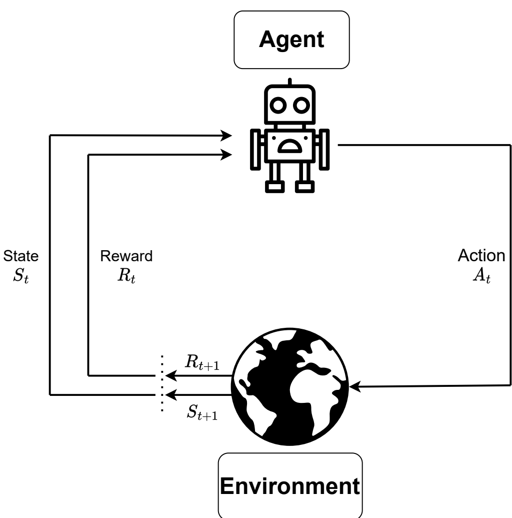


Figure 1: The Agent-Environment Interface: The agent interacts with the environment at time step t by taking action A_t. The environment then changes to state S_{t+1} and produces reward R_{t+1}, which are both passed back to the agent so that it can decide the next action. The dotted lines indicate that this process repeats itself. Inspired by Sutton and Barto [2018].

Time is discrete, which means it can be indexed $t \in \{0, 1, 2, ...\}$. Before going any further, it is important to note that uppercase letters indicate random variables and lowercase letters indicate values these variables can take. The states, actions and rewards may take several forms, depending on the problem one is trying to solve.

- The states can be numbers, words, colors, shapes and images, for instance. In short, they have to encode all the possible states of the environment. This set will be represented by $\mathcal{S}$ and so $s \in \mathcal{S}$.
- In a given environment, the set of actions an agent can take may depend on the state it is in. If it does, then $a \in \mathcal{A}(s)$. However, in general, $\mathcal{A}(s) = \mathcal{A}, \forall s \in \mathcal{S}$. For example, in the game of chess, the actions the agent can take depend on the pieces on the board and their position. This set of actions may be continuous or discrete, depending on the problem at hand. For instance, if the problem consists of driving a car autonomously from point A to B, then the set of actions may include the steering angle, which is continuous. On the other hand, Chess has a discrete set of actions, since there are only some possible actions from a certain position.
- The rewards are usually encoded as $r \in \mathbb{R}$ and indicate the goodness of the agent's decisions concerning the goal. Going back to the chess example, the rewards could be $\{-1, 0, 1\}$ depending on whether the agent loses, draws, or wins a game, respectively. The expected rewards are calculated using the reward function $R_s^a : S \times A \mapsto r$, in which $r \in \mathcal{R}$.

However, the environment dynamics still need to be defined. How do the actions taken by the agent change the state of the environment? The answer is given by the *state transition probability function*,

$$p^a_{s's} = Prob(S_{t+1} = s'|S_t = s, A_t = a) \tag{2.1}$$

Put in words, this is the probability that taking action a in state s at time t will lead to state s' at time $t+1$. The *state transition probability matrix* $\mathcal{P}$ defines a multi-dimensional array encoding the probabilities of transitioning between states s and s' once the agent takes action a.

One last important property is that states in a RL setting are *Markov*, which means that evolution only depends on the current state, neglecting the past. A state is Markov if and only if $Prob(S_{t+1}|S_t) = Prob(S_{t+1}|S_1, ..., S_t)$. Put simply, by knowing the present state of the environment, the agent has all the necessary information to make a decision. Chess, for example, is a Markov game, since all the states satisfy the Markov property. A player only needs to look at the board to decide what the best play is, there is no advantage in knowing what plays led to that moment.

Now, one has all the ingredients needed to define the mathematical framework in which RL problems are set. That is the *Markov Decision Process (MDP)*. An MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, R \rangle$ Sutton and Barto [2018], where:

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{A}$ is a finite set of actions.
- $\mathcal{P}$ is a state transition probability matrix.
- R is the reward function.

It is important to distinguish between two different types of MDPs - *Partially Observable MDPs (POMDPs)* and *Fully Observable MDPs* Wiering and Van Otterlo [2012]. When the environment sends the new state s_{t+1} to the agent, then the environment is a fully observable MDP, since the agent always knows the state it is in. For instance, Chess is an MDP, since both players have access to the full information about the state they are in. However, if the environment sends an observation containing only partial information about the new state s_{t+1}, then it is a POMDP. For instance, consider the game of Poker. Each player only has access to his own hand of cards and to the "community" cards (the ones shown at the center of the table). However, the cards of the other players are not visible. Hence, Poker is a POMDP, since an agent only has partial information about the state it is in and has to take into account the different possible cards the other players may have and the cards yet to be revealed at the center of the table. In this work, only fully observable MDPs will be considered.

The behavior of the agent is given by the *policy*, which is a probability distribution of actions over states $\pi(a|s)$. It determines the probability the agent will take each action a from a particular state s. Then, the agent samples randomly from this distribution to decide which action to take. However, policies can also be deterministic, meaning there is no randomness involved, in which case only one action can be taken from a certain state: $a = \pi(s)$.

As an example of an MDP, consider Figure 2. This environment has 16 states, each represented by a real number that indicates the agent's position on the grid such that $\mathcal{S} \in \{1, 2, ..., 16\}$. The agent has four possible actions $\mathcal{A} \in \{\text{left, right, up, down}\}$ as long as they respect the limits of the grid world. For example, if the agent is in state 1, then it can only move up or right. This shows how the actions can be conditioned by the states. In this scenario, we want the agent to find the apple in the least number of steps possible, so the environment will produce a -1 reward per time-step and a $+10$ reward for finding the apple. If the agent moves into a state with a bomb, then it receives a -5 reward.

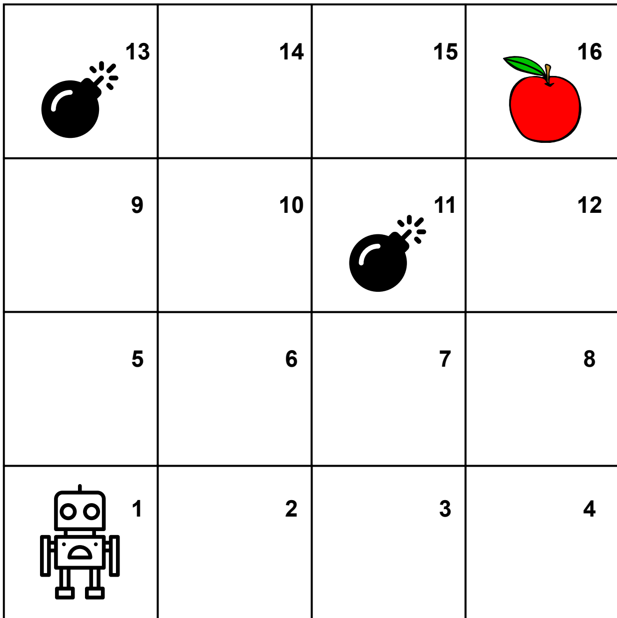


Figure 2: An example of an MDP - a grid world in which the goal of the agent is to find the shortest path to the apple while not bumping into the bombs.

Now, the question is: what makes for a good policy? Before answering that question, let's introduce one last concept - the *Return*. The return is the total *discounted reward* from time-step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.2}$$

where $\gamma \in [0, 1]$ is the *discount factor*, which determines how much long-term rewards are valued over short-term rewards. If $\gamma = 0$, then the agent only cares about the immediate reward. On the

other extreme, if $\gamma = 1$, the agent values all future rewards equally. This means that γ needs to be chosen carefully depending on the problem at hand and the desired behaviour of the agent. However, it is important to note that, for the convergence of series, γ may not take the value 1, such that $\gamma \in [0, 1)$.

The objective of a RL agent becomes clear: to identify the policy that maximizes the expected return $\max_\pi \mathbb{E}_\pi[G_t]$. In essence, since rewards gauge the quality of the agent's actions relative to its objective, a policy accumulating substantial rewards indicates proficient and intelligent decision-making within the environment. It is important to understand that, as a consequence of the defined framework, every RL agent needs to have a well-defined goal and a reward function that reflects that goal in its actions. For example, if the problem is a game of chess, the goal of the agent could be to win the game, and the rewards $\{-1, 0, 1\}$ depending on whether the agent loses, draws, or wins. In a race-like scenario, the goal of the agent is to take as little time as possible to complete the task, so the reward function could be -1 reward per time-step.

In this section, the mathematical framework in which RL problems are set was defined. Furthermore, it was stated that the goal of a RL agent is to find the policy that maximizes the expected return. Now, the important question is how to find such an optimal policy.

2.2 State and Value Functions and the Notion of Optimality

Let's start by introducing two important functions: the *State Value Function* and the *Action-Value Function*. The *State Value Function* $v_\pi(s)$ defines how good it is for an agent to be in a certain state s. It is the expected return starting from state s, and then following policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] \tag{2.3}$$

This function can be broken down by conditioning on actions, creating the *Action-Value Function* $q_\pi(s, a)$, which defines how good taking a certain action from a particular state is. It consists of the expected return starting from state s, taking action a, and then following policy π thereafter:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \tag{2.4}$$

Furthermore, in any MDP, there exists an optimal state-value function and the optimal action-value function Sutton and Barto [2018].

The optimal state-value function $v_*(s)$ is the maximum value function over all policies

$$v_*(s) = \max_{\pi} v_\pi(s) \tag{2.5}$$

Similarly, the optimal action-value function $q_*(s, a)$ is the maximum action-value function over all policies

$$q_*(s, a) = \max_{\pi} q_\pi(s, a) \tag{2.6}$$

The optimal value function specifies the best possible performance in the MDP. In other words, an MDP is considered solved once the optimal value function is known. But how so? The goal is to find a policy and not a value function. Even if these functions specify the best policy somehow, how can it be extracted? First, let's define a partial ordering over policies so that policies can be compared qualitatively. A policy π is better than or equal to a policy π' if, for all states, the value function generated by π is equal or greater than the value function generated by π':

$$\pi \geq \pi' \text{ if } v_\pi(s) \geq v_{\pi'}(s), \forall s \tag{2.7}$$

Then, there is a theorem that states that, for any MDP Sutton and Barto [2018]:

- There exists an optimal policy π_* that is better than or equal to all other policies, $\pi_* \geq \pi, \forall \pi$
- All optimal policies achieve the optimal value function, $v_{\pi_*}(s) = v_*(s)$
- All optimal policies achieve the optimal action-value function, $q_{\pi_*}(s, a) = q_*(s, a)$

Keeping this in mind, it becomes clear that, once one has the optimal action-value function $q_*(s, a)$, the optimal policy can be found by simply maximizing over it,

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in \mathcal{A}} q_*(s, a) \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

This is obvious if one considers a very simple scenario. Imagine there is a state s in which the agent can take two possible actions: a_1 and a_2. Now, imagine that $q_*(s, a_1) = 10$ and $q_*(s, a_2) = 3$. Put in words, this means that, by taking action a_1, the agent is expecting a return of 10 from then on, and similarly, by taking action a_2, it expects a return of 3. The best decision in this scenario would be to take action a_1. By applying this way of thinking to all possible states, one obtains the optimal policy. It is also possible to obtain the optimal policy once the optimal state-value function is known, although harder. In that case, the action chosen should always be the one that leads to the state with the higher value. This

involves an extra step since the agent has to know the model of the environment to see the successor states in order to choose the optimal action.

It is also important to note that, for any MDP, there always exists a deterministic optimal policy Sutton and Barto [2018]. This means that, even if the policy is a distribution of actions over states, the goal is for it to eventually converge into the deterministic optimal policy. However, this is not the case for POMDPs, where the optimal policy may be stochastic.

Now the question is: How is the optimal action-value function found? There are many ways to solve this problem, and some of them will be discussed in the next section.

2.3 Bellman Equations and Dynamic Programming

The *Bellman Equation* relates the value of a state to the value of its successor states:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{2.9}$$

Similarly, there is also a *Bellman Equation* for the action-value function:

$$q_\pi(s, a) = \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a) \left[r + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a') \right] \tag{2.10}$$

Both of these equations are valid for any policy π, all $s \in \mathcal{S}$ and all $a \in \mathcal{A}$.

Additionally, Sutton and Barto [2018] states that the so-called *Bellman Optimality Equations* hold true solely under an optimal policy, asserting that the value of any specific state, under such a policy, has to align with the expected return for the best action available in that state.

$$v_*(s) = \max_a \sum_{s', r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{2.11}$$

$$q_*(s, a) = \sum_{s', r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{2.12}$$

One way of finding the optimal value functions and, consequently, the optimal policy, is by solving a system of $|\mathcal{S}|$ equations, one Bellman Optimality Equation for each state $s \in \mathcal{S}$, using any method for solving non-linear equations. However, this solution is very rarely useful, since it is equivalent to an

exhaustive search (making it unfeasible for complex problems with high dimensional state and action spaces) and relies on the agent knowing the dynamics of the environment, a property known as *Complete Knowledge*, which is an assumption that very rarely holds in real-world problems Sutton and Barto [2018].

This is a good opportunity to introduce the two major types of RL methods: *Model-Based* and *Model-Free* Kaelbling et al. [1996]:

- In model-based methods, the agent has access to (or learns an approximated version of) the model and can use it to plan. Since the agent knows the dynamics of the environment, it can use that knowledge to simulate the different possible paths and their consequences, before taking an actual action in the real environment. For example, in Chess, because the rules and the structure of the game are known and the transitions are deterministic, the agent can search through the different possible paths and, in a sense, think a couple of steps ahead. Then, using that knowledge, it can choose the path that seems most favorable. An example of this is Silver et al. [2017].

- In model-free methods, the agent doesn't have access to a model. These methods are simpler and easier to implement, but also usually suffer from a higher sample complexity, since the agent has to deal with the extra burden of not knowing the dynamics of the environment, and thus needs extra interactions to reach a considerable level of performance.

Model-free methods are more popular and more researched than model-based methods for a couple of reasons, with the main one being that, most of the time, the real model of the environment is not known. Consequently, model-based algorithms have to learn an approximation of the model, which by itself may be a complicated process, but this also leads to another problem. Since the model the agent is using to plan isn't a perfect model of the environment, the best performance in the model doesn't have to translate to the best performance in the actual environment. In other words, the policy of the agent will only be as good as the model it built from the environment. In this work, we will only research a particular model-free method known as *Q-Learning* Watkins and Dayan [1992].

Nonetheless, before getting into Q-Learning, it is important to explain what *Dynamic Programming (DP)* is. DP is a collection of model-based algorithms that require access to the dynamics of the environment $p(s', r|s, a)$ and that use value-functions as guides in the search for optimal policies Sutton and Barto [2018]. These algorithms are of limited utility because they are computationally costly and require a perfect model. However, they are extremely important from a theoretical standpoint and serve as the basis for Q-Learning, as well as many other RL algorithms. In short, they turn the Bellman Equations into assignments (update rules) that, every time they are applied, improve the approximations of the value functions, with convergence guarantees as the number of updates tends to infinity.

They work by intertwining two different processes: *Policy Evaluation* and *Policy Improvement*. In Policy Evaluation, the goal is to approximate the value function given by the policy π. This is done by sweeping over the entire state space using the Bellman Equation as an update rule. Then, as this process is repeated time and time again, the approximation of the value function becomes better and better. In practice, a stopping criterion has to be used. A common one is to define a small threshold $\theta > 0$ and, if the difference between the new values and the previous values is smaller than θ, the process is stopped. In Policy Improvement, a new policy is created, one which is greedy with respect to the original value function. This new policy is guaranteed to always be at least as good as the original policy Sutton and Barto [2018]. The alternation of any variation of these two processes is known as *Generalized Policy Iteration (GPI)* and, as long as all the states continue to be updated, converges to the optimal value function and the optimal policy.

In this section, the Bellman Equations and Dynamic Programming, which offer a means to solve the RL problem, are introduced. In the next section, Q-Learning is explained.

2.4 Q-Learning

In model-free RL, the agent does not have access to the environment dynamics. In other words, a model is neither known nor learned. Consequently, agents learn from experience - finite sequences of states, actions, and rewards, in the format $\{s, a, r, s'\}$, which are called *trajectories* Sutton and Barto [2018]. However, learning from experience introduces some new problems.

The first problem has to do with the policy. The agent is learning from experience that has to be generated by some policy π. Now, how is this policy chosen? On the one hand, the agent has to explore the state space to find high-value states. On the other hand, the agent has to exploit the acquired knowledge, taking actions that it knows yield high rewards and lead to good states. This problem is known as the exploration-exploitation tradeoff Kaelbling et al. [1996] and has some practical solutions. Maybe the simplest one, which will be used in this work, is an *ϵ-greedy policy* Sutton and Barto [2018], which chooses a random action with probability ϵ and the greedy action with probability $1 - \epsilon$. It is typical to decay ϵ over time. This idea comes from the understanding that, in the beginning, as the agent has very little to no knowledge, exploration is crucial. Then, as the agent learns, exploitation becomes more important. Still, a minimum value for ϵ is defined to ensure continuous exploration (a requirement for convergence guarantees) Sutton and Barto [2018]. Nonetheless, the choice of the decaying scheduling of ϵ depends on the environment and must be chosen carefully taking the aforementioned tradeoff into consideration.

In addition, model-free RL algorithms can be classified into two types: On-Policy and Off-Policy. On-policy algorithms learn a policy π from experience generated from the same policy π, while off-policy algorithms learn a policy ν from experience generated from a different policy π. The latter are noteworthy as they enable agents to learn from humans or other agents and reuse experience generated from previous policies. Notably, Q-Learning is an off-policy algorithm.

The second problem has to do with the value function the agent learns. In model-free methods, if one wants to find the optimal policy (or a near-optimal one), the learned value function has to be the action-value function $Q(s, a)$. Learning the state-value function $V(s)$ requires a model of the MDP for greedy policy improvement, as the agent must identify the action leading to the highest valued state. To do so, the agent must know the resulting state and reward for each action. Greedy policy improvement over $Q(s, a)$ is model-free and involves maximizing over actions, as mentioned in the previous section.

To estimate the action-value function and determine the optimal policy, two effective methods exist: Monte-Carlo (MC) and Temporal Difference Learning (TD-Learning). Monte-Carlo techniques use averages to approximate expected returns and GPI to derive a policy π that approximates the optimal policy π_*. Specifically, these techniques use averages of returns for estimating the action-value functions in the Policy Evaluation step and an ϵ-greedy policy in the Policy Improvement step. However, these methods apply only to episodic environments where all trajectories terminate. In this scenario, each trajectory is often called an episode. TD-Learning overcomes some MC limitations, by combining it with DP. Like MC methods, it learns from raw experience but updates estimates with other learning estimates without waiting for an outcome - *bootstrapping*. For instance, consider a robot that has to guess how many steps it needs to take before reaching a goal using TD-Learning. Let's say each step yields a -1 reward. In its original position, the agent might think that 15 steps are necessary, and thus expects a return of -15 (assuming $\gamma = 1$). Then, the agent takes one step and receives a reward of -1. However, now being closer to the goal, the agent notices that it overshot its first prediction and now thinks only 10 steps are necessary to reach the goal. Consequently, the agent updates its estimation of the original state toward the improved estimation of the successor state. That is the idea behind TD-Learning - updating estimates with other estimates. TD-Learning offers lower variance, updates state values at every time step, and functions in non-episodic environments. For a more detailed explanation of MC and TD methods and their differences, refer to Sutton and Barto [2018].

Finally, one has all the information needed to understand Q-Learning. In summary, it is an off-policy algorithm that attempts to approximate the optimal action-value function $q_*(s, a)$ using TD-learning. The pseudo-code for the algorithm follows:

Algorithm 1 Q-Learning (Sutton and Barto [2018])

1: Algorithm parameters: step size $\alpha \in (0, 1]$, small $\epsilon \in [0, 1]$
2: Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal_state, \cdot) = 0$
3: **loop** for each episode:
4: Initialize S
5: **repeat** for each step of the episode:
6: Choose A from $\mathcal{A}(S)$ using policy derived from Q (e.g., ϵ-greedy)
7: Take action A, observe R, S'
8: $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
9: $S \leftarrow S'$
10: **until** S is terminal
11: **end loop**

As previously explained, uppercase letters indicate random variables and lowercase letters indicate values these variables can take. Hence, all the variables in the pseudo-code are uppercase letters. As one can see from lines 6 and 7, the experience is generated from a behavior policy, which has to balance the exploration-exploitation tradeoff. As previously mentioned, this work will use an ϵ-greedy policy for simplicity. Then, line 8 establishes the update rule for the Q-Values. Put in words, the Q-values are updated a little (depending on step size α) in the direction of the immediate reward plus the discounted maximum Q-values of the successor states. There is much to unpack there, so let's start one step at a time.

- α is the step size and must be chosen depending on the problem. A larger α leads to more pronounced updates. The larger the α, the more the agent considers recent information and ignores prior knowledge.

- $R + \gamma \max_a Q(S', a)$ is the *TD-target* and consists of the immediate reward plus the discounted maximum Q-values of the successor states. TD-Learning bootstraps by updating estimates using other estimates. It is also possible to verify why Q-Learning is an off-policy algorithm. Even though the experience is generated from a behavior policy π (e.g. ϵ-greedy), the agent always considers the maximum Q-values of the successor states. In other words, the target policy (the one the agent is learning) is greedy with respect to the Q-values.

- The difference $\delta = R + \gamma \max_a Q(S', a) - Q(S, A)$ is called *TD-Error* and determines the update to be done to the current estimates of the Q-values. It consists of the difference between the TD-

target and the current estimates of the Q-values. These estimates are the updated in the direction of the TD-error, scaled by the step-size α.

This section introduced methods to determine the (near) optimal policy using value functions as guides, such as Q-Learning. The next section will explain the scalability problem that undermines this algorithm and introduce one of the possible solutions to solve it.

2.5 Deep Reinforcement Learning

The previous section outlines *tabular* approaches Sutton and Barto [2018], which use lookup tables of (at most) dimensions $\mathcal{S} \times \mathcal{A}$ to assign values to state-action pairs encountered by the agent. However, these methods suffer from the *curse of dimensionality*, making them inadequate for handling large problems. For instance, the game of Go has 10^{170} states, making it unfeasible to construct a lookup table with at least 10^{170} entries and determine values for each entry with current technology. Additionally, some problems have continuous state and/or action spaces. To apply RL to real-world issues, the methods must possess the capability to solve intricate, complex problems. Can model-free methods be expanded to tackle these challenges?

The answer to the question is positive. In fact, there are several possible solutions, such as using state-abstraction Andre and Russell [2002] and/or dimensionality reduction techniques Sorzano et al. [2014]. This work focuses on the most popular solution - using *Function Approximation*. The idea is to generalize from seen states to unseen states by approximating a target function. Consequently, a new function is defined:

$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a) \tag{2.13}$$

where **w** is a vector of parameters that must be updated using TD-Learning or MC. It is important to refer that function approximation is only useful if $dim(\mathbf{w}) \ll |\mathcal{S} \times \mathcal{A}|$, otherwise it would be the same as the tabular regime.

There are many function approximators, each with pros and cons, such as linear combinations of features and Kernel-based approximators Busoniu et al. [2017]. Nonetheless, the most used function approximators, not only in RL but in machine learning in general, are Neural Networks (NNs). NNs are popular because they are differentiable, accurately approximate complex functions, have strong generalization capabilities, scale up to handle larger datasets, and benefit from GPU computing and groundbreaking results in practice Morales [2020]. Neural networks (NNs) are made up of neurons that receive inputs,

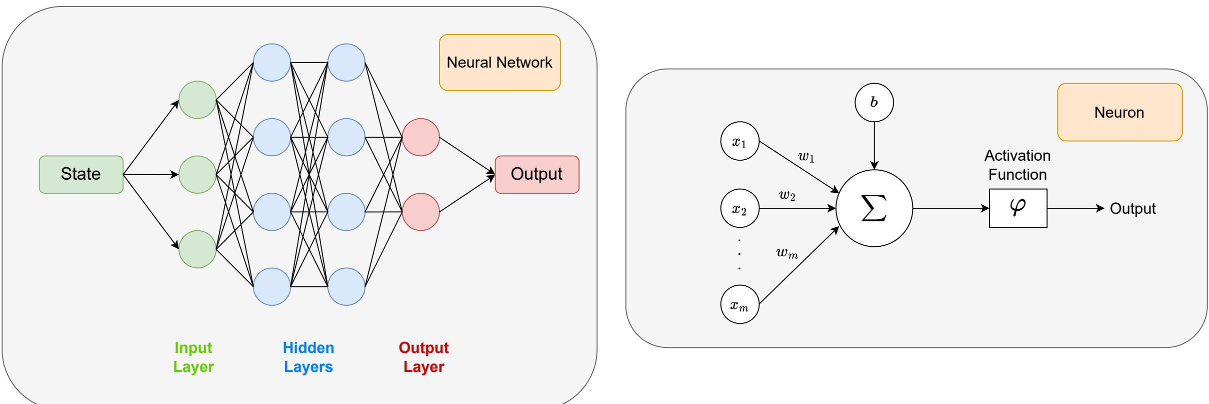


Figure 3: A neural network is composed of neurons, which are grouped into layers. Each neuron receives m inputs $\{x_1, \ldots, x_m\}$, which are multiplied by the m respective weights $\{w_1, \ldots, w_m\}$ and summed. A bias b is also added. Finally, the result is passed through an activation function φ, which produces the output.

each with a weight that is adjusted during training. The neuron multiplies each input by its weight and sums all weighted inputs, adds a trainable weight called the bias, and passes the result through a non-linear activation function to determine the neuron's output. These neurons are organized into layers, with the input layer receiving the data, and the output layer yielding the processed result. Any layer in between is called a hidden layer. If a neural network has two or more hidden layers, it is known as a Deep Neural Network (DNN). See Figure 3 for a visualization of a NN and a neuron. The more layers, the more complex functions the NN can approximate. To train a neural network (NN), an objective function, commonly known as a loss function, is defined. This function quantifies the discrepancy between the NN's predicted outputs and the desired outputs. The goal during training is to minimize this error. Typically, the learning algorithm adjusts the network's weights based on the gradient of this loss function. The gradient, for each weight, indicates how much the error would change if that weight were perturbed by a small amount. The general idea is to adjust the weights in the direction that reduces the error. This concept was extensively discussed by LeCun et al. LeCun et al. [2015]. In practice, an optimization algorithm, often a variant of Stochastic Gradient Descent (SGD), is employed. Instead of calculating the gradient using the entire dataset — which can be computationally intensive — SGD approximates the gradient using a subset of the data. This subset, called a mini-batch, is randomly sampled from the dataset. This approach not only reduces computational requirements but can also introduce beneficial noise, potentially aiding in generalization and avoiding local minima LeCun et al. [2015].

Any RL algorithm that uses DNNs as function approximators is considered Deep Reinforcement Learning (DRL). DRL has been applied to increasingly complex problems, with high-dimensional state-spaces

and/or action-spaces, achieving comparable or better performance than humans in a variety of benchmark environments. In 2016, Deepmind's Alpha-Go, a Go-playing computer program that used DRL and Monte-Carlo Tree Search, defeated the European Go Champion by 5-0 Silver et al. [2016]. This result was a breakthrough for DRL and AI in general since Go was considered the most challenging classic game for artificial intelligence due to its vast search space. In 2017, a subsequent version of Alpha-Go, based solely on RL, without human data, achieved super-human performance, beating the previous Alpha-Go version by 100-0 Silver et al. [2017].

Even outside of the typical benchmark environments of board games and video games, DRL has been applied with great success in real-world problems. Nvidia, a multinational technology company that specializes in designing Graphics Processing Units (GPUs), has recently announced that they have used DRL for designing more efficient arithmetic circuits Roy et al. [2021]. Deepmind has also shown that DRL can be used to assist the operations of nuclear fusion Degrave et al. [2022].In quantum physics, it has been applied to the many-body problem that consists of predicting the properties of systems made of many interacting quantum particles. Since the numerical solution to this problem is intractable for systems of moderate size, different methods that use approximations have been used to solve it. Recently, NNs have been used to approximate the quantum states and DRL techniques have been employed to find the ground-state and describe the unitary time-evolution of complex interacting quantum systems Carleo and Troyer [2017]. DRL has been applied successfully in diverse fields, such as robotics Nguyen and La [2019], finance Hu and Lin [2019], healthcare Yu et al. [2021] and communications/networking Luong et al. [2019].

In this section, the need for function approximators in RL was made clear, considering the scalability limitations of tabular approaches. In particular, DNN and, consequently, DRL were introduced. The following section will explain how DNN can be used in Q-Learning.

2.6 Deep Q-Learning

To solve the dimensionality problem, function approximators can be used to approximate the Q-function in Q-Learning Melo and Ribeiro [2007]. In the seminal work Mnih et al. [2015], a DNN is used as the Q-function approximator and the resulting algorithm is named Deep Q-Network (DQN). A comparison between Q-Learning and DQN can be seen in Figure 4.

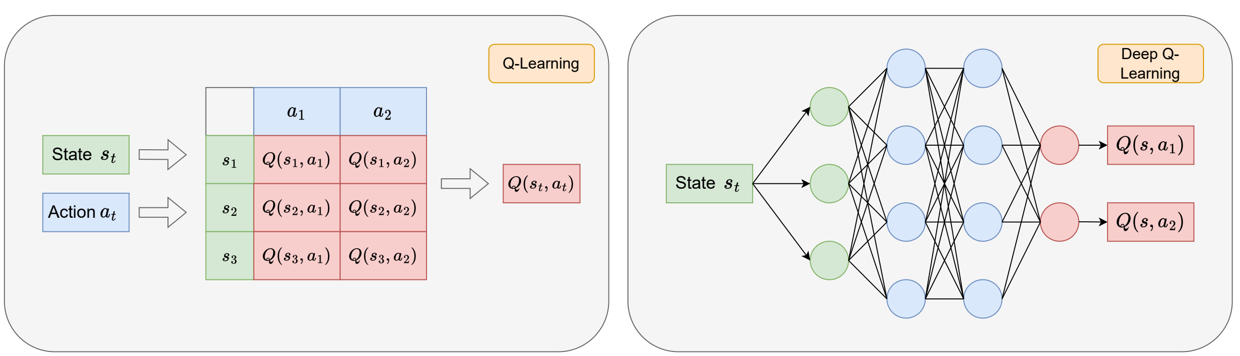


Figure 4: A comparison between Q-Learning and Deep Q-Learning (DQN). Q-Learning is a tabular approach, where the values of all the visited state-action pairs are stored in a lookup table. DQN, on the other hand, uses a deep neural network as the Q-function approximator. The Figure shows one of the possible architectures for the NN, where the input layer receives the components of the state and the output layer outputs the Q-values for all possible actions. See Morales [2020] for other possible architectures.

The pseudo-code for the algorithm follows:

Algorithm 2 Deep Q-Learning

1: Algorithm parameters: step size $\alpha \in [0,1]$, small $\epsilon \in [0,1]$
2: Initialize replay buffer $\mathcal{D}$ to capacity N
3: Initialize action-value function Q with random weights θ
4: Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
5: **for** episode$= 1, M$ **do**
6: Initialize s_1
7: **for** $t = 1, T$ **do**
8: Choose a_t from s_t using ϵ-greedy policy
9: Take action a_t, observe r_{t+1}, s_{t+1}
10: Store experience $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
11: Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
12: Compute the TD-targets using target network (with old, fixed parameters θ^-)
13: Perform a gradient descent step on $\left[r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right]^2$ w.r.t parameters θ
14: Every C steps reset $\hat{Q} = Q$
15: Initialize s_t
16: **end for**
17: **end for**

In the algorithm, several control constants warrant additional explanation. Firstly, the Experience Replay Buffer is initialized with a capacity of N. This capacity determines the maximum number of experiences that the buffer can hold. It is a hyperparameter that requires fine-tuning, since its optimal value varies across different environments. In line 10, for clarity, only the storage of experiences in the replay buffer was described. However, when the replay buffer reaches its capacity, older experiences are discarded to accommodate newer ones. The parameter M denotes the number of episodes, T the steps per episode, and C stands for the target network update frequency, which will be further elaborated in the subsequent paragraph.

The authors demonstrated that this algorithm achieves super-human performance on a number of arcade games. They also introduce two novel improvements over previous approaches: *experience replay* and *target network*. The former consists of using a *Replay Buffer*, where all experiences - interactions with the environment in the format $(s_t, a_t, r_{t+1}, s_{t+1})$ - are stored. Then, batches of these experiences are randomly sampled to compute the parameter updates. This way, temporal correlations between the samples are broken, making the data appear more *Independent and Identically Distributed (IID)*, which stabilizes training. The *Target Network* consists of a second NN that is used to compute the TD-targets. The target network has "frozen" weights that are sporadically (every C steps) updated with a copy of the parameters of the online network. The idea behind this is to make the targets more stationary, which also stabilizes training. However, there is a tradeoff here between speed of convergence and stability which must be accounted for when choosing C. For a more nuanced discussion on the necessity of these techniques, the reader is referred to the original paper Mnih et al. [2015] and the book Morales [2020]. It is also important to note that there are algorithms that build on DQN, such as the Double DQN Hasselt [2010] (DDQN) and the Dueling DDQN Wang et al. [2016]. Nonetheless, the concept is essentially the same.

DQN has been applied successfully to various problems. Starting with the aforementioned research paper that introduced DQN, which created a DQN agent that, receiving only raw visual input, achieved a level of performance comparable to (and sometimes better than) that of a professional human games tester in 49 games from the Atari 2600 games Mnih et al. [2015], which is one of the most exciting results in DRL. It has also been used in stock market forecasting, proving to develop strategies better than a typical benchmark strategy Carta et al. [2021]. Additionally, it has been employed in the path planning of coastal ships, improving the safety, economy, and autonomous decision-making ability of ship navigation Guo et al. [2021].

Nonetheless, there is an important caveat to DQN that must be mentioned - it is highly unstable and

has no convergence guarantees Tsitsiklis and Van Roy [1996]. In fact, Sutton and Barto [2018] even go as far as to define what they call the *deadly triad* - function approximation, bootstrapping, and off-policy data. Every algorithm that uses these three elements, such as DQN, is unstable and likely to diverge. The target network and experience replay techniques were introduced to DQN to ameliorate this problem.

2.7 Policy-Gradient Algorithms and Taxonomy of RL

So far, many different RL algorithms have been discussed. The first distinction made was between model-free and model-based algorithms. Then, some model-free methods that use value functions as guides in the search for the optimal policy were introduced and explained. These methods are usually referred to as *Value-Based*. However, those are not the only model-free methods for solving RL problems. Some try to learn the policy directly - *Policy-Based* algorithms. They typically work by parameterizing the policy $\pi_\theta(a|s)$, defining an objective function that serves as some performance measure $J(\pi_\theta)$ and optimizing the parameters θ by performing gradient ascent on an approximation of the gradient $\nabla J(\pi_\theta)$, which is obtained using Monte-Carlo samples. Consequently, they are commonly referred to as *policy gradient* methods. Some notable policy-gradient algorithms include *REINFORCE* Williams [1992], REINFORCE with baseline Sutton and Barto [2018] and *Proximal Policy Optimization (PPO)* Schulman et al. [2017].

Policy-based methods have their pros and cons when compared with value-based methods. On the one hand, unlike value-based methods, they can learn stochastic policies, which makes them more suitable for problems with incomplete information (POMDPs). Additionally, the action probabilities change smoothly with small changes to the parameters, whereas in value-based methods a small change to the estimated action values may lead to a dramatic change in the action probabilities, meaning that the former have better convergence guarantees Sutton and Barto [2018]. Perhaps the simplest advantage is that, in some scenarios, it may be easier to learn the policy than a value function. On the other hand, since updates are based on Monte-Carlo samples, they suffer from high variance, which may slow down training Greensmith et al. [2004]. The important conclusion from this comparison is that both methods are useful and their applicability vastly depends on the problem itself. Consequently, both are actively researched and equally important. There is also another type of RL algorithm that combines both value-based and policy-based methods, leveraging their strengths while undermining their weaknesses, called *actor-critic* algorithms Konda and Tsitsiklis [1999]. In these, both the parameterized policy and the value function are learned, where the former can be seen as the actor and the latter the critic. Actor-critic algorithms often yield state-of-the-art performance in many DRL benchmark environments Morales [2020], Mnih et al. [2016].

For a more detailed discussion on policy-gradient and actor-critic algorithms, along with their advantages and disadvantages, the reader is referred to Sutton and Barto [2018].

A simple tree diagram to visualize the taxonomy of the RL methods learned so far can be seen in Figure 5.

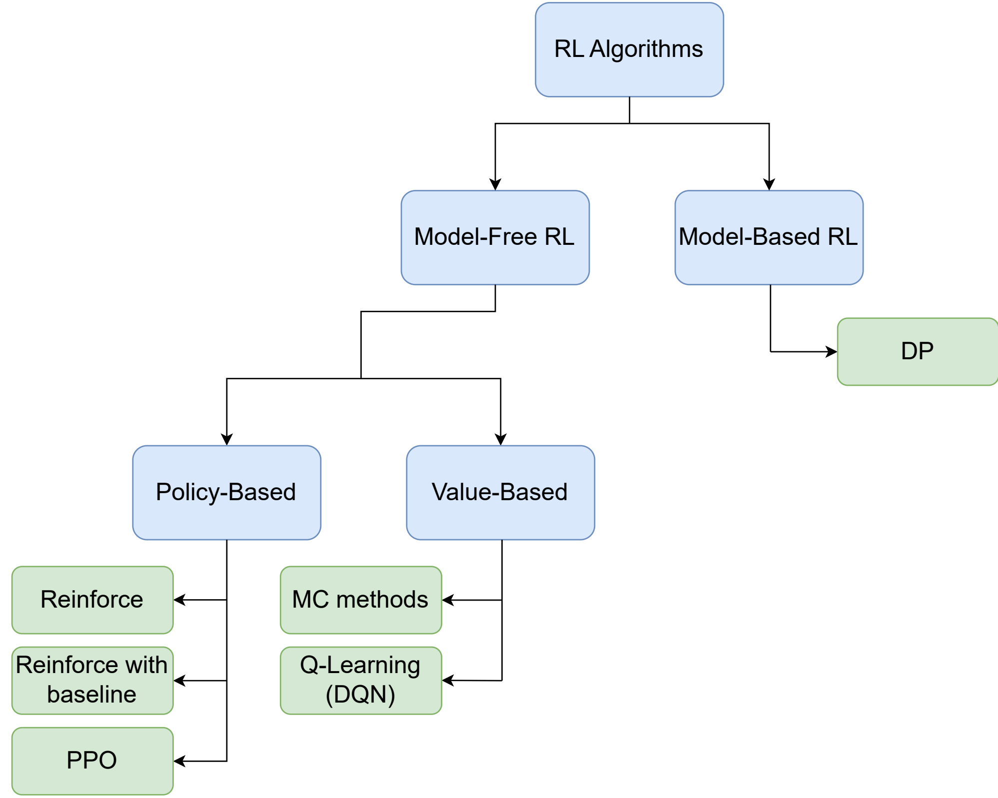


Figure 5: A non-exhaustive taxonomy of the RL algorithms mentioned in this chapter.

2.8 Summary

Reinforcement Learning (RL) is a subset of machine learning where an agent learns to make decisions by taking actions in an environment to maximize cumulative rewards over time.

The goal is to find the optimal policy, a distribution of actions over states, which can be thought of as the optimal behavior in the environment. To search for the optimal policy, value functions can be used as guides. These functions estimate the expected cumulative reward an agent can obtain from a given state or state-action pair. Every single state is related to every successor state by the Bellman Equations. In particular, under an optimal policy π_*, the Bellman Optimality Equations apply.

One way to find the optimal policy is to solve several systems of non-linear Bellman Optimality Equations, one for each state. Nonetheless, other methods reduce the computational cost. One such example is Dynamic Programming (DP), which turns the Bellman Equations into update rules and is guaranteed to

converge to the optimal policy as the number of updates tends to infinity. However, these methods require access to a model of the environment and are hence called model-based. A notable algorithm that doesn't require access to a model is Q-Learning, which works by having the agent interact with the environment and using that experience to approximate the state-action values.

Nonetheless, it still suffers from scalability problems, as all the values have to be stored in a lookup table. A solution to this problem is to use function approximators. In particular, Deep Neural Networks (DNNs) are commonly used in RL algorithms, giving rise to Deep Reinforcement Learning (DRL). Deep Q-Network (DQN) is an algorithm that uses DNNs to approximate the Q-values. Still, some methods attempt to learn the policy directly, without using value functions. These algorithms are known as Policy-based and have their advantages and disadvantages. Consequently, all methods and algorithms are actively researched and used across diverse fields.

Chapter 3

Quantum Computing

Quantum computers process data and solve problems by leveraging the principles of quantum physics. By making use of such phenomena, they hold the promise of solving specific problems, like factoring large numbers or simulating quantum systems, exponentially faster than the most advanced classical computers. Nonetheless, building scalable and error-resistant quantum machines remains a significant challenge. Consequently, the benefits of near-term quantum computing remain somewhat unclear.

This chapter begins by contextualizing and introducing the fundamentals of quantum computing. It then delves into a specific subset of quantum circuits designed for near-term quantum computing, known as variational quantum circuits. The latter sections introduce quantum reinforcement learning, with a spotlight on strategies that utilize variational quantum circuits.

3.1 Context

During a conference in May 1981, Richard Feynman introduced the idea of a quantum computer. He argued that classical computers could not efficiently simulate quantum systems, and suggested that computers that could harness the principles of quantum physics would be better suited for the task Feynman [1982]. Then, in 1985, David Deutsch formalized quantum computers Deutsch [1985] and raised an important question: Can quantum computers solve problems that have nothing to do with quantum physics more efficiently than classical computers? Surely enough, in 1997, David Simon developed a quantum algorithm that is exponentially faster than the best-known classical algorithm at finding the period of a function Simon [1997]. Then, in 1999, Peter Shor found a quantum algorithm with a similar complexity advantage over the best-known classical algorithm at factoring large numbers Shor [1999]. However, unlike Simon's algorithm, Shor's has a very important practical application - the potential to break public-key cryptography schemes which are widely used to protect digital information all over the globe. This discovery generated considerable interest in quantum computing Preskill [2023].

One hurdle remained - the implementation of these algorithms in quantum hardware. For the quantum algorithm to work, the involved qubits (the quantum version of bits) have to remain in a superposition during the whole process. However, these superpositions are extremely fragile and can suffer decoherence, leading to computational errors Haroche and Raimond [1996]. Still, error-correcting codes were developed Steane [1996] and Shor introduced fault-tolerant quantum computing, which allows for implementations in noisy quantum hardware Shor [1996].

Nonetheless, present-day quantum computers are devices with up to a few hundred qubits without error correction. These devices were coined *Noisy-Intermediate Scale Quantum (NISQ)* devices by John Preskill Preskill [2018]. In 2019, the Google AI Quantum Group claimed they achieved "quantum supremacy" by getting a superconducting quantum computer to perform a task exponentially faster than the best classical supercomputer at the time Arute et al. [2019]. However, while this task may be of practical use in the future, that is not the case at the present time. Hence, most researchers in the NISQ era have been trying to find practical problems where quantum computers outperform classical ones. A particularly promising approach, which will be explained in more detail in the following sections, consists of using hybrid classical/quantum algorithms to find approximate solutions to optimization problems Farhi et al. [2014].

3.2 Quantum Circuits

3.2.1 Single-Qubit and Multi-Qubit Systems

There are different models of quantum computing. However, this work focuses solely on the circuit model of quantum computing Nielsen and Chuang [2010].

A bit is the most basic unit of information in classical computing. It can only have one of two values: 0 and 1. The *quantum bit (qubit)* is the quantum analog of the classical bit. The main difference between these two is that qubits make use of a quantum phenomenon known as *superposition*, meaning that they can be in a linear combination of 0 and 1. Thus, the general state of a qubit is given by:

$$|\Psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle \tag{3.1}$$

where α_0 and α_1 are complex coefficients that represent probability amplitudes $\alpha_0, \alpha_1 \in \mathbb{C}$ and $|\alpha_0|^2 + |\alpha_1|^2 = 1$. Furthermore, the states $|0\rangle = (1,0)^T$ and $|1\rangle = (0,1)^T$ make up the *computational basis*.

The complex conjugate of the general one-qubit state $|\Psi\rangle$ defined in equation 3.1 is given by:

$$\langle\Psi| = \alpha_o^* \langle 0| + \alpha_1^* \langle 1| \tag{3.2}$$

where α_0^*, α_1^* are the complex conjugates of α_0, α_1, respectively, and $\langle 0| = (1, 0)$, $\langle 1| = (0, 1)$.

Another difference between qubits and bits is that qubits cannot be examined to determine their state. Instead, a measurement has to be performed. An *observable* (which is a quantum operator) is measured and the state collapses to one of the *eigenvectors* of the observable - measurements are *destructive*. Typically, qubits are measured in the *computational basis* $\{|0\rangle, |1\rangle\}$. A single measurement yields only a single bit of information about the state of the qubit.

If the qubit from equation 3.1 is measured in the computational basis:

- With probability $|\alpha_0|^2$ - The result is 0 and the state of the qubit after measurement is $|0\rangle$.
- With probability $|\alpha_1|^2$ - The result is 1 and the state of the qubit after measurement is $|1\rangle$.

A qubit can thus be in an arbitrary state parameterized by three angles γ, θ and φ:

$$|\Psi\rangle = e^{i\gamma}\left(\cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle\right) \tag{3.3}$$

The factor $e^{i\gamma}$ can be ignored, since it has no observable effects - it is a *global phase*. The values θ and φ define a point on the unit three-dimensional sphere, which can be thought of as the geometric representation of a single-qubit state. This sphere is commonly known as the *Bloch Sphere*, see Figure 6. If the qubit is in a pure state and not in a mixture of several states, then it is represented by a point on the surface of the Bloch sphere Nielsen and Chuang [2010].

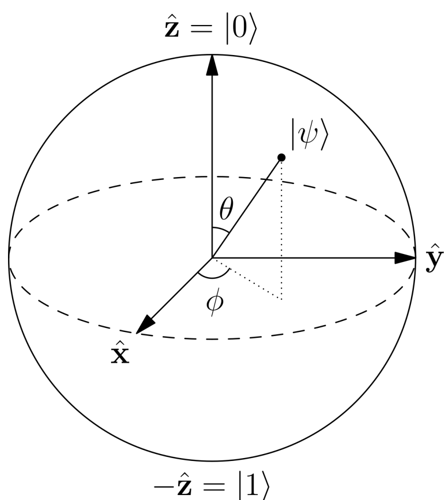


Figure 6: Representation of a Bloch Sphere

It is also possible to use several qubits to construct multi-qubit systems. A general N qubit quantum state is given by:

$$|\Psi\rangle = \sum_{x\in\{0,1\}^N} \alpha_x |x\rangle \tag{3.4}$$

Where

$$\sum_{x \in \{0,1\}^N} |\alpha_x|^2 = 1 \tag{3.5}$$

For example, consider $N = 2$. If one has two classical bits, then there are four possible combinations, or states, 00, 01, 10 and 11. Similarly, a system with two qubits has four possible computational basis states $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$. Thus, a general state for the 2-qubit system is given by:

$$|\Psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle \tag{3.6}$$

Some multi-qubit systems can be built from single-qubit systems using the *tensor product* ($\otimes$):

$$|\Psi\rangle = |\Psi_1\rangle \otimes |\Psi_2\rangle \otimes ... \otimes |\Psi_N\rangle = |\Psi_1\Psi_2...\Psi_N\rangle = \bigotimes_{i=1}^{N} |\Psi_i\rangle \tag{3.7}$$

However, not all multi-qubit states can be decomposed into tensor products of single-qubit states. These states are said to be *entangled*. For example, consider the well-known *Bell State*, which is a very important two-qubit state:

$$|\Psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{3.8}$$

This state is entangled because there is no way to express $|\Psi\rangle$ as a product of two separate qubit states. This property, known as *entanglement*, is essential in quantum computing and is what allows for information processing beyond what is possible in the classical world Nielsen and Chuang [2010].

3.2.2 Single-Qubit and Multi-Qubit Quantum Gates

To perform meaningful computations and solve tasks, one has to be able to manipulate the quantum states. That is the goal of the elementary quantum operations, the *quantum gates*. These gates are represented by matrices with a single constraint - they have to be *unitary*. That is, given a matrix U and its *adjoint* $U^\dagger$, U is unitary if $U^\dagger U = I$, where I is the identity matrix. Let's start with some well-known single-qubit gates, such as the *Hadamard* gate H, which is used to create uniform superpositions:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{3.9}$$

Applying it to the computational basis yields:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle \tag{3.10}$$

$$H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle \tag{3.11}$$

There are some other important single-qubit gates, such as the *Pauli* X, Y and Z gates and the Identity gate I:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \; Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \; Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \; \mathbb{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{3.12}$$

Even if one is not familiar with quantum computing, some of these gates are pretty intuitive. For example, the identity gate I does nothing to a quantum state, such that $I|\Psi\rangle = |\Psi\rangle$. The Pauli X gate is the quantum analog of the classical NOT gate. It turns $|0\rangle$ into $|1\rangle$ and vice-versa. More importantly, it turns a superposition $\alpha|0\rangle + \beta|1\rangle$ into $\alpha|1\rangle + \beta|0\rangle$. In other words, it acts *linearly*. This is a characteristic of quantum computing - quantum gates act on quantum states via linear transformations. The Pauli Z gate, also known as a *phase-flip* gate, flips the phase of the $|1\rangle$ state but leaves the $|0\rangle$ state unchanged. The Pauli-Y gate transforms $|0\rangle$ to $i|1\rangle$ and $|1\rangle$ to $-i|0\rangle$. In other words, it applies both Pauli-X and Pauli-Z operations simultaneously but with a complex coefficient.

From the Pauli matrices, one can also define the R_X, R_Y and R_Z rotation gates, which perform rotations on a quantum state over the x, y and z axis of the Bloch Sphere, respectively:

$$R_X = e^{-i\theta X/2} = \begin{bmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ -i\sin(\theta/2) & \cos(\theta/2) \end{bmatrix} \tag{3.13}$$

$$R_Y = e^{-i\theta Y/2} = \begin{bmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{bmatrix} \tag{3.14}$$

$$R_Z = e^{-i\theta Z/2} = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix} \tag{3.15}$$

An arbitrary single-qubit gate can be decomposed as a product of these rotation gates:

$$U(\theta, \varphi, \lambda) = R_Z(\theta)R_Y(\varphi)R_Z(\lambda) = \begin{bmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\varphi}\sin(\theta/2) & e^{i\varphi+i\lambda}\cos(\theta/2) \end{bmatrix} \tag{3.16}$$

Some of the main single-qubit quantum gates were introduced. However, to create entanglement between multiple qubits and extract the full power of quantum computing, multi-qubit gates are needed. The typical multi-qubit quantum gate is the *Controlled-NOT* or *CNOT* gate. It is given by the following unitary matrix:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{3.17}$$

It receives two inputs, a *control* qubit and a *target* qubit, and its action is very simple. If the control qubit is set to 1, then a X gate is applied to the target qubit (in other words, the target qubit is flipped). If the control qubit is set to 0, then nothing changes in the target qubit.

Using a CNOT and the arbitrary single-qubit gate defined in equation 3.17, it is possible to implement any arbitrary n-qubit operation Nielsen and Chuang [2010]. Thus, the CNOT, R_X, R_Y and R_Z gates form a universal set of gates. However, it is possible to build arbitrarily good approximations of any n-qubit operations using fixed values of θ, φ and λ in the arbitrary single-qubit gate along with the CNOT gate. In other words, it is possible to approximate any single-qubit gate using a finite set of quantum gates, which are also considered universal gate sets.

Keeping all of this in mind, quantum computations are done through *quantum circuits*. These involve a set of qubits, which are acted on by a circuit of quantum gates (which can be conditioned on classical computations) and measured at the output (and possibly at intermediate steps) Nielsen and Chuang [2010].

3.2.3 Measurements

An arbitrary state $|\Psi\rangle \in \mathbb{C}^{2^n}$ is represented by:

$$|\Psi\rangle = \sum_{i=0}^{2^{n-1}} c_i |\Psi_i\rangle \tag{3.18}$$

where $|\Psi_i\rangle$ are the basis states and c_i the probability amplitudes associated with each of these states. Measuring this state in the computational basis will collapse it into one of the basis states $|\Psi_i\rangle$ with probability $|c_i|^2$ Nielsen and Chuang [2010]. Furthermore, the expectation value of some observable $\hat{O}$ is given by:

$$\left\langle \hat{O} \right\rangle = \langle\Psi| \hat{O} |\Psi\rangle = \sum_{i=0}^{2^{n-1}} \lambda_i p_i \tag{3.19}$$

where λ_i is the eigenvalue associated with the eigenvector of $\hat{O}$ and $p_i = |c_i|^2$ its respective probability. This notion of expectation values will be critical in Variational Quantum Circuits.

3.3 Variational Quantum Circuits

As was mentioned in section 3.1, fault-tolerant quantum computers are still not available and may remain that way for years, maybe decades Cerezo et al. [2021a]. Instead, one has to make do with NISQ devices, which have a limited number of qubits and circuit depth due to noise. Consequently, it is crucial to find

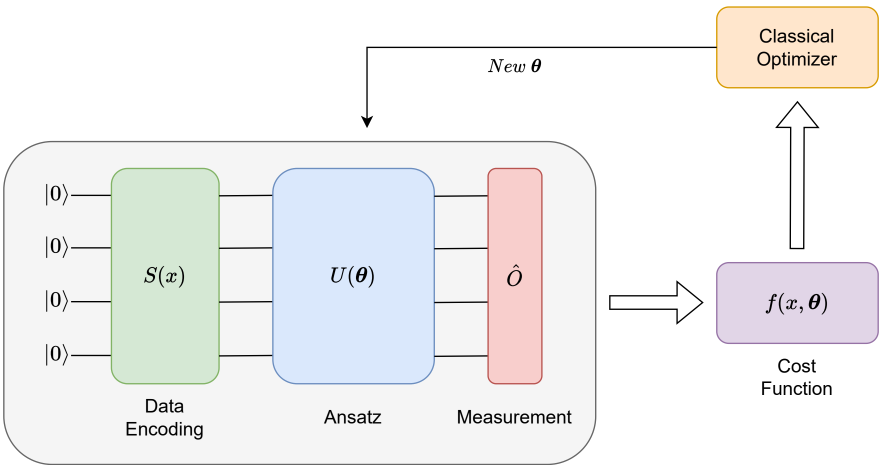


Figure 7: The building blocks of a VQC and how it is trained. In a typical VQC, data is encoded, then processed by a parameterized unitary and, finally, the expectation value of some observable is measured. Then, a cost function that depends on the expectation value is calculated and a classical optimizer updates the parameters $\boldsymbol{\theta}$.

practical applications for such devices. A promising approach consists of using hybrid classical/quantum algorithms to find approximate solutions to optimization problems Farhi et al. [2014].

Variational Quantum Circuits (VQCs) are quantum circuits that depend on free parameters, which are iteratively updated by a classical optimizer to minimize an objective function estimated from measurements Ostaszewski et al. [2021]. Typically, VQCs can be divided into three main components: *Data Encoding*, *Ansatz*, and *Measurement and Cost Function*, which will be explained in the next three subsections. In Figure 7, one can see the general structure of a VQC.

3.3.1 Data Encoding

When dealing with classical data (as will be the case in this work), the data has to be encoded into a quantum state so that it can be processed by a quantum computer. There are several techniques to encode classical data into quantum states Schuld and Petruccione [2021]. However, this work will focus solely on *Angle Encoding*. Using this technique, each component (also known as feature) of the input data x is encoded by a single qubit using arbitrary Pauli rotations R_X, R_Y, R_Z, where the angle is usually the feature itself after some classical pre-processing, e.g. normalization. Thus, given the input data $x = \{x_0, x_1, ..., x_{n-1}\}$, the resulting quantum state is given by:

$$|x\rangle = \bigotimes_{i=0}^{n-1} R_\alpha(\phi(x_i)) \, |0_i\rangle \tag{3.20}$$

where $R_\alpha \in \{R_X, R_Y, R_Z\}$, ϕ is some classical pre-processing function and $|0_i\rangle$ is the ith qubit

initialized in state $|0\rangle$.

This technique has pros and cons. On the one hand, it allows the encoding of a given input vector using circuits of depth 1 Sequeira et al. [2022]. On the other hand, the number of qubits grows linearly with the number of features of the input vector, which limits the dimensionality of the inputs one can encode due to the reduced number of qubits in NISQ devices.

3.3.2 Ansatz

An architecture (sometimes referred to as *ansatz*) needs to be chosen for the VQC. What specific sequence of gates applied to which qubits will be used to solve the task? There are several choices, some of them are *problem-inspired*, meaning that knowledge about the problem is used to build them. These types of ansatzes are commonly used in quantum chemistry Cao et al. [2019], as well as other tasks where the physics behind the problem might help tailor the ansatz. However, in RL, to the best of our knowledge, there aren't any problem-inspired anstazes. Thus, a particular subset of problem-agnostic ones are typically used - *Hardware-Efficient Ansatzes*. These ansatzes allow for implementations with reduced circuit depth by bringing correlated qubits together for depth-reduction Cerezo et al. [2021a]. Furthermore, they usually consist of a layered architecture composed of single-qubit parameterized gates followed by a cascade of entangling gates Sequeira et al. [2022].

Usually, the ansatz $U(\theta)$ can be divided into *layers*, such that:

$$U(\boldsymbol{\theta}) = U_L(\theta_L)...U_2(\theta_2)U_1(\theta_1) \tag{3.21}$$

An example of a Hardware-Efficient VQC may be seen in Figure 8.

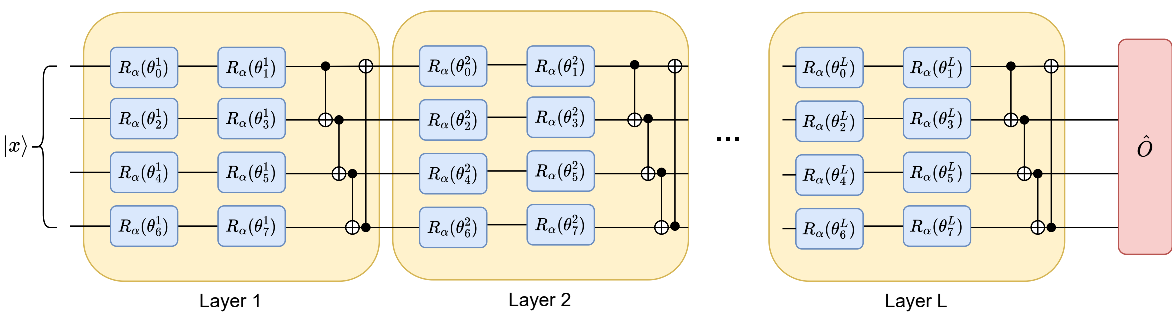


Figure 8: Example of a hardware-efficient VQC. $|x\rangle$ is processed by layers of unitaries, each of them composed of parameterized single-qubit gates followed by a cascade of entangling gates.

3.3.3 Cost Function

To use a VQC to solve a problem, one has to first encode the problem into a cost function, such that solving the problem corresponds to finding the global optima of said function. Let $f_{\boldsymbol{\theta}(x)}$ be the expectation value of some observable $\hat{O}$ as follows:

$$f_{\boldsymbol{\theta}}(x) = \langle 0| U^{\dagger}(x, \boldsymbol{\theta}) \hat{O} U(x, \boldsymbol{\theta}) |0\rangle \tag{3.22}$$

where $U(x, \boldsymbol{\theta})$ is a quantum circuit that depends on the input x and a set of free parameters $\boldsymbol{\theta}$. In short, it is a VQC.

Then, the cost function $L(\boldsymbol{\theta})$ is a function of the expectation value itself. In many scenarios, $L(\boldsymbol{\theta})$ reduces to $f_{\boldsymbol{\theta}}(x)$. However, in the context of machine learning, classical post-processing is usually added to the expectation value. For instance, consider the typical supervised learning scenario. Given a labeled dataset $D = \{(x_i, y_i)\}^M$, the typical cost function is the Mean Squared Error, which can be expressed as follows:

$$L(\boldsymbol{\theta}) = \frac{1}{M} \sum_{i=0}^{M-1} (f_{\boldsymbol{\theta}}(x_i) - y_i)^2 \tag{3.23}$$

Such a cost function will be important in the context of this dissertation applied to the RL problem, as treated in Subsection 3.5.1.

In practice, the value of $f_{\boldsymbol{\theta}}$ for a specific input x is given by running the VQC multiple times and averaging over the results. Then, typically, gradient-based methods are used to update the parameters of the VQC. However, gradient-free methods also exist and have their advantages and disadvantages, see Cerezo et al. [2021a].

When using gradient-based methods, as will be the case of this work, the gradient of the cost function with respect to the parameters is calculated using the Parameter-Shift Rules Schuld et al. [2019]. These rules state that the partial derivative of $f_{\boldsymbol{\theta}}(x)$ w.r.t a single variational parameter (assuming this parameter is the angle of a Pauli-rotation) is given by:

$$\frac{\partial f_{\boldsymbol{\theta}}(x)}{\partial \theta_i} = \frac{1}{2} \left[f_{\boldsymbol{\theta}}(x; \theta_i + \pi/2) - f_{\boldsymbol{\theta}}(x; \theta_i - \pi/2) \right] \tag{3.24}$$

To compute the partial derivative of the cost function w.r.t a single parameter, two circuit executions are needed. Thus, to compute the gradient of the cost function, which has p parameters, $2 \times p$ executions are necessary.

However, there is one problem that pertains to the gradients of hardware-efficient ansatzes, which will be explained in the following section.

3.3.4 Barren Plateau Phenomenon

There is a tradeoff one needs to take into account when using hardware-efficient VQCs. Since these ansatzes take no inspiration from the structure of the problem itself, they need to be highly expressive so they can be applied to any generic task Bilkis et al. [2021]. In this context, high expressivity means these quantum circuits are able to nearly cover the full Hilbert space associated with its quantum register space.

Let $L(\boldsymbol{\theta})$ be a hardware-efficient VQC cost-function. The gradient of this function is given by:

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \left\{ \frac{\partial L}{\partial \theta_1}, \frac{\partial L}{\partial \theta_2}, ..., \frac{\partial L}{\partial \theta_k} \right\}, \ \boldsymbol{\theta} \in \mathbb{R}^k \tag{3.25}$$

where $\boldsymbol{\theta}$ are the cost function's parameters and k the number of parameters.

The Barren Plateau Phenomenon asserts that

$$Var[||\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})||] \in \mathcal{O}\left(\frac{1}{\alpha^n}\right), \ \alpha > 1 \tag{3.26}$$

where $||\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})||$ is the gradient's L2-norm.

Put in words, the variance of the gradient of hardware-efficient VQC-based cost functions decays exponentially with system size McClean et al. [2018]. Furthermore, Holmes et al. [2022] demonstrated that the more expressive the ansatz, the lower the variance in the cost gradient. Consequently, the cost landscape is flatter, making these circuits extremely hard to train. Nonetheless, it is important to emphasize that Cerezo et al. [2021b] observed that the Barren Plateau Phenomenon highly depends on the locality of the observable employed. To solve this issue, several strategies have been proposed, such as initialization strategies Grant et al. [2019]. Nonetheless, the problem severely limits the applicability of hybrid classical/quantum algorithms to solve practical problems, warranting further research.

3.4 Data Re-Uploading

In the previous section, it was explained that classical data needs to be encoded into a quantum state to be processed by the VQC. Furthermore, from Figure 7, it seems as though this encoding is done just once at the beginning of the circuit to prepare the quantum state. However, that doesn't need to be the case. Schuld et al. [2021] determined that the data encoding strategy used in the quantum circuit influences the expressive power of the quantum model. Specifically, they show that a VQC may be written as a Partial Fourier Series, where the accessible data frequencies are determined by the eigenvalues of the data-encoding Hamiltonians. Furthermore, quantum models can access increasingly rich frequency spectra by repeating simple data encoding gates multiple times either in series or in parallel, a technique called *Data*

Re-uploading Pérez-Salinas et al. [2020]. To demonstrate this, the authors considered a standard quantum model that consists of multiple layers, each made up of a data encoding block $S(x)$ and a trainable block $W(\theta)$ (see Figure 9), and assume that the input data is encoded using angle encoding, such that:

$$U(x, \boldsymbol{\theta}) = W^{L+1}(\theta_{L+1})S(x)W^{L}(\theta_L)...W^{2}(\theta_2)S(x)W^{1}(\theta_1) \tag{3.27}$$

Where $W^i(\theta_i)$ is the i-th parameterized trainable block, $S(x)$ is a data encoding block that utilizes angle encoding, and L is the number of layers.

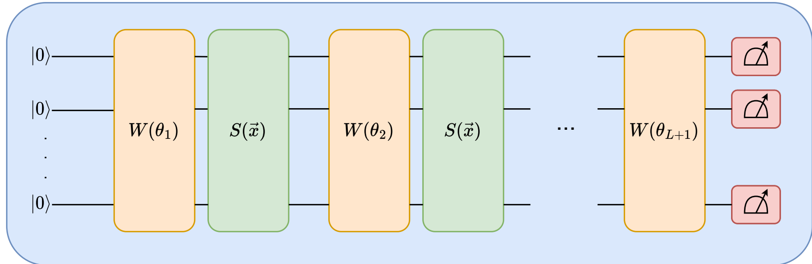


Figure 9: The Data Re-Uploading technique in action. The same data encoding block $S(x)$ is repeated several times throughout the quantum circuit to increase the expressivity of the quantum model.

To exemplify the power of data re-uploading, the authors showed that a quantum model employing a single Pauli-rotation encoding can only learn a sine function. In other words, if the encoding is done just once, the VQC can only approximate functions with a single non-zero frequency in the frequency spectrum. However, repeating Pauli encoding linearly extends the frequency spectrum, such that, if the encoding is repeated n times, then the quantum model can approximate functions with up to n different non-zero frequencies. The authors also verified that *input scaling*, a technique that consists of multiplying each of the input features by a scaling factor, before encoding it as the angle of some Pauli rotation, scales the frequency spectrum itself. Moreover, Pérez-Salinas et al. [2020] proposed making this scaling weight trainable to allow for an "adaptive frequency matching".

3.5 VQC-Based Reinforcement Learning

VQCs have been extensively researched in supervised Mitarai et al. [2018], Schuld et al. [2020], Schuld and Killoran [2019], Farhi et al. [2014] and unsupervised Coyle et al. [2020], Zoufal et al. [2021] machine learning. However, research on Quantum RL remains scarce, and only recently have variational approaches emerged Chen et al. [2020],Lockwood and Si [2020],Skolik et al. [2022], Jerbi et al. [2021], Sequeira et al. [2022]. All of these references apply VQCs to RL in the same intuitive way. Just like NNs can be used

to approximate either a value-function or a parameterized policy, so can VQCs, see Figure 10. The result is a hybrid classical-quantum algorithm that generally works as follows. The agent observes some state s_t and applies some classical pre-processing ϕ. Then, the result $\phi(s)$ is encoded into a VQC $U_{\boldsymbol{\theta}}(\phi(s))$ using some data encoding technique. The VQC, with the current parameters $\boldsymbol{\theta}_t$, prepares a quantum state. An observable O_a is measured for each possible action. The expectation values of these observables $\langle O_a \rangle_{s,\boldsymbol{\theta}}$ are post-processed and the result represents either the Q-values $Q_{\boldsymbol{\theta}}(s,a)$ in value-based methods or the policy $\pi_{\boldsymbol{\theta}}(a|s)$ in policy-based methods. Then, the agent chooses an action a_t using these predictions and executes it in the environment. The reward r_t and the consecutive state s_{t+1} are observed by the classical optimizer. Using the parameter-shift rule, the gradients of the VQC w.r.t the parameters $\boldsymbol{\theta}_t$ are calculated. Finally, the classical optimizer determines the new parameters $\boldsymbol{\theta}_{t+1}$ Meyer et al. [2022].

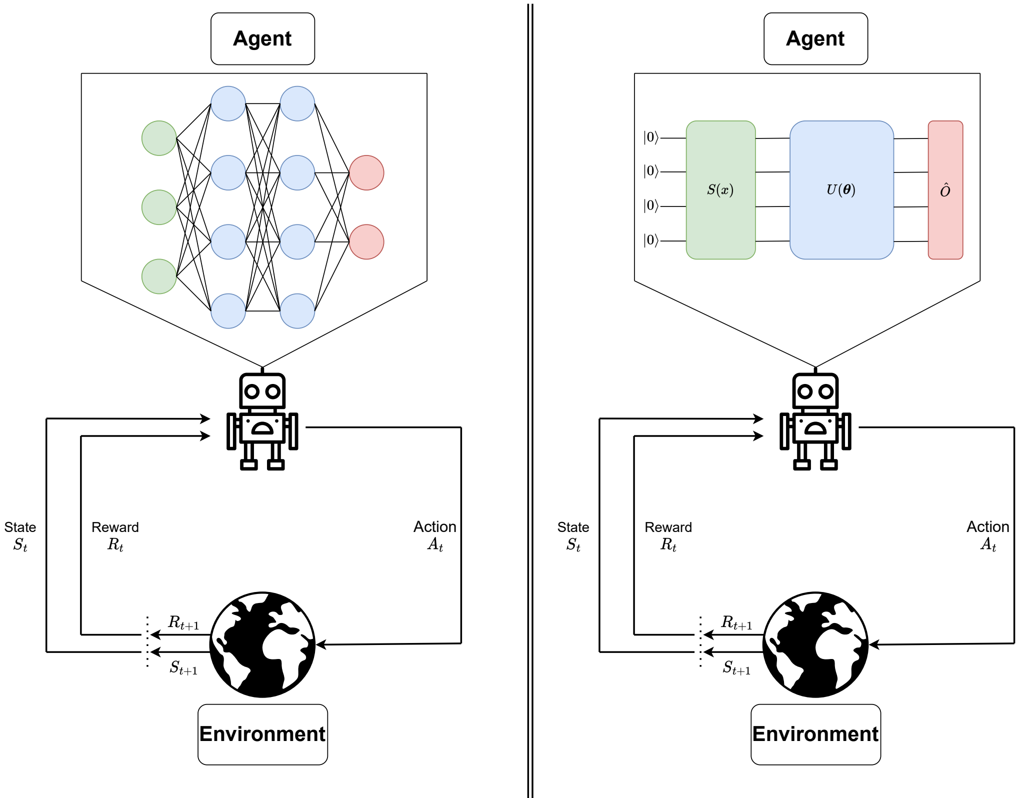


Figure 10: Deep RL versus VQC-based RL. The image on the left represents Deep RL, where Deep Neural Networks are used as function approximators for either value functions or policies. Similarly, the image on the right represents VQC-based RL, where VQCs are used for the same effect. The algorithms themselves are identical, with the exception of the model used as a function approximator.

It is important to note that this work will consider only VQCs being used as function approximators in classical environments, even though they can also be used in quantum environments Jerbi et al. [2021]. For a survey that encompasses all of the quantum approaches to RL, the reader is referred to Meyer

et al. [2022]. Some of the aforementioned references apply VQCs to the policy-based algorithm named REINFORCE with baseline. These references will be reviewed in subsection 3.5.2. Others apply VQCs to the value-based algorithm Q-Learning (DQN), which will be reviewed in subsection 3.5.1. Even though this work is particularly concerned with Q-Learning, it is important to understand the methodologies used in the policy-based algorithm as well as the results obtained, since some details translate from one algorithm to the other.

3.5.1 VQC-Based Q-Learning

To the best of our knowledge, Chen et al. [2020] was the first paper to use VQCs as function approximators in RL algorithms. Specifically, the authors used VQCs to approximate the Q-function in a Deep Q-Learning algorithm. The pseudo-code of the algorithm follows:

Algorithm 3 Variational Quantum Deep Q-Learning

1: Algorithm parameters: step size $\alpha \in [0, 1]$, small $\epsilon \in [0, 1]$
2: Initialize replay buffer $\mathcal{D}$ to capacity N
3: Initialize action-value function Q (quantum circuit) with random parameters θ
4: Initialize target action-value function $\hat{Q}$ (target quantum circuit) with parameters $\theta^- = \theta$
5: **for** episode$= 1, M$ **do**
6: Initialize s_1 and encode into the quantum state
7: **for** $t = 1, T$ **do**
8: With probability ϵ select a random action a_t, otherwise select $a_t = \max_a Q^*(s_t, a; \theta)$ from the output of the quantum circuit
9: Execute action a_t in emulator and observe r_{t+1}, s_{t+1}
10: Store experience $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathcal{D}$
11: Sample random mini-batch of transitions (s, a, r, s') from $\mathcal{D}$
12: Compute the TD-targets using target quantum circuit (with old, fixed parameters θ^-)
13: Perform a gradient descent step on $\left[r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\right]^2$ w.r.t parameters θ
14: Every C steps reset $\hat{Q} = Q$
15: Initialize s_t and encode into the quantum state
16: **end for**
17: **end for**

The authors used computational basis encoding Schuld and Petruccione [2021] to encode the states into the VQC. Then, they used an ansatz composed of a ladder of CNOTs that connects the nearest neighbors followed by general-single qubit gates with three parameters per qubit, see Figure 11a.

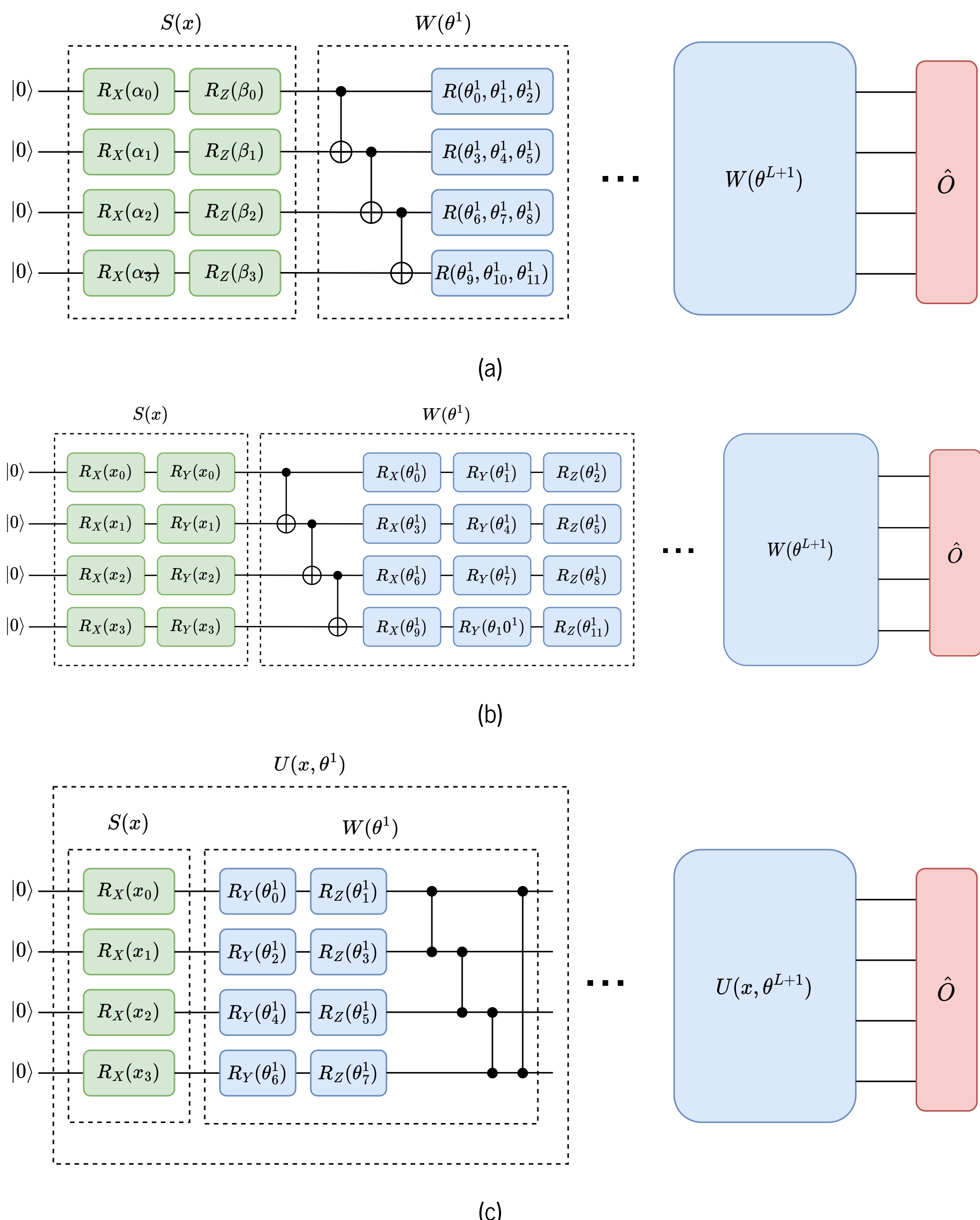


Figure 11: Architectures used in Chen et al. [2020] (Subfigure 11a), Lockwood and Si [2020] (Subfigure 11b) and Skolik et al. [2022] (Subfigure 11c) for $n = 4$ qubits, L layers, parameters θ and input $x = [x_0, x_1, x_2, x_3]$. The data encoding gates are green-coloured and compose the blocks $S(x)$. The variational gates are blue-coloured and compose the blocks $W(\theta^l)$, where l is the layer, along with the entangling gates. When Data Re-Uploading is used, as in Subfigure 11c, a layer is composed of the data encoding and variational blocks $U(x, \theta^l)$.

Finally, the authors measured Pauli-Z observables on a number of qubits equal to the number of actions in the environment. The authors tested the algorithm in two very simple benchmark environments, with low-dimensionality state and action-spaces, and claimed that the VQC performed as well as a NN. Furthermore, they claimed a ”memory consumption advantage”, since VQCs required one order of magnitude less parameters than NNs to achieve a comparable performance.

In the context of this work, the term ”performance” holds significant relevance and will be frequently mentioned. In RL, performance typically refers to the return achieved by a model in a specific environment. Essentially, it is an evaluation of the policy generated by the model. While the highest possible return is the ultimate goal, when multiple models achieve it, the one requiring fewer samples is deemed superior. However, interpretations of performance can vary. Some papers average the returns of N agents derived from a specific model. Others employ a single agent, using a moving average of its return as the performance metric. Given this subjectivity and variations in definitions, the reader is referred to the original papers for a detailed description of the methodology used.

Then, the paper by Lockwood and Si [2020] changed some aspects of the algorithm. Notably, they introduced a new data encoding method named *scaled encoding*, capable of encoding continuous states. In short, it scales each feature s_i of the input state s to the range $[0, 2\pi]$, which is then encoded using two single-qubit Pauli rotations - angle encoding. Consequently, each feature of the input state has to be encoded into a different qubit. In particular, they used R_X and R_Y rotations to encode the inputs. The ansatz of the VQC then consisted of an entangling block composed of nearest-neighbor CNOTs followed by parameterized R_X, R_Y and R_Z rotations per qubit, see Figure 11b. They also used two different methods to obtain the Q-values from the VQCs. The first method feeds the measurement of the VQC into a classical single dense layer with a number of nodes equal to the number of possible actions. The second one uses a technique known as quantum pooling Cong et al. [2019]. Then, the algorithm was tested in slightly more complex benchmark environments than Chen et al. [2020] and similar results were verified: the VQC-based agents achieved a level of performance comparable to or better than the NN-based agents, with a reduced parameter count.

Finally, the most important paper for this work is the one by Skolik et al. [2022], which also uses VQCs in a Deep Q-Learning algorithm. The authors studied the effect of the data encoding techniques on the performance of the agents. Furthermore, they also investigated the role of the observables and different post-processing techniques. The data encoding technique was very similar to the one used by Lockwood and Si [2020]. Features are scaled to the finite interval $[-\pi/2, \pi/2]$ by applying the arctan function, and the results are encoded as Pauli-rotations. Moreover, *input scaling* is used, meaning that each feature

is scaled by a classical trainable weight before being fed to the arctan function and encoded into the quantum circuit, which allows for adaptive frequency matching between the target function and the output of the quantum model Schuld et al. [2021]. The encoding block is repeated several times throughout the quantum circuit, following the data re-uploading technique Pérez-Salinas et al. [2020], Schuld et al. [2021]. The ansatz used is a hardware-efficient one composed of an entangling block of circular CZ rotations, followed by two parameterized Pauli-rotations per qubit, see Figure 11c. Then, to decode the action values from the VQC, the agents measure the expectation value of a number of observables equal to the number of actions allowed, such that:

$$Q(s,a) = \left\langle 0^{\otimes n} \right| U_{\boldsymbol{\theta}}^{\dagger}(s) O_a U_{\boldsymbol{\theta}}(s) \left| 0^{\otimes n} \right\rangle \tag{3.28}$$

where s is the state encoded into the VQC, $U_{\boldsymbol{\theta}}(s)$ is the VQC and n is the number of qubits.

Furthermore, the authors also multiply the expectation values by classical trainable weights, such that the Q-values that the quantum model outputs are given by:

$$Q(s,a) = \left\langle 0^{\otimes n} \right| U_{\boldsymbol{\theta}}^{\dagger}(s) O_a U_{\boldsymbol{\theta}}(s) \left| 0^{\otimes n} \right\rangle \times w_i \tag{3.29}$$

The reason for using such a strategy is actually quite simple. Q-values $Q_\pi(s,a)$ are the expected return after taking action a from state s and then following the policy π thereafter. Thus, the range of the optimal Q-values varies from environment to environment, depending on its structure and reward function. Moreover, for an agent to achieve optimal or near-optimal performance in an environment, it needs to be able to approximate the optimal Q-values with good precision. This is not a problem for NNs, since the range of the output values can change arbitrarily during training. However, the same is not true for VQCs. Expectation values of observables are bounded, e.g. the expectation value of the Pauli-Z observable is bounded by $[-1,1]$. Consequently, a VQC can't approximate the optimal value functions of some environments, which will significantly impact performance. Hence the need to multiply the expectation values by classical trainable weights. Then, the resulting Q-values can also have any arbitrary range, which means they can approximate the optimal Q-functions in all environments.

The authors tested their algorithms in benchmark environments and verified that the VQC-based agents were able to solve them. Moreover, the performance in one of the environments is compared to the performance of a NN-based agent with the same number of parameters, with the VQC-based agents achieving a higher average return, hence performing better in the environment. Furthermore, the effect of the input scaling and output scaling was empirically tested. As for the input scaling, it was noticed that it improves the performance of the agent, likely due to the adaptive frequency matching and consequent

increased expressivity. Furthermore, having a trainable output weight instead of multiplying the expectation values by a fixed scalar also seems to improve the performance of the agent.

It is important to note that the three mentioned articles Chen et al. [2020], Lockwood and Si [2020] and Skolik et al. [2022] used the Mean Squared Error cost function seen in algorithm 3. Moreover, although any observable may be used to extract the information from the VQCs, all of the papers used Pauli-Z observables. This choice was not due to some insight into the correct choice of observables, but rather because measurements in the computational basis are common and because it works.

3.5.2 VQC-Based Policy-Gradients

Jerbi et al. [2021] and Sequeira et al. [2022] adapted the VQC-based approach to Q-Learning from Chen et al. [2020] to policy-based methods. More precisely, they used VQCs as function approximators for the policy in the REINFORCE with baseline algorithm. However, there are some dissimilarities between the two references, particularly the architecture of the VQCs. While Jerbi et al. [2021] used a more expressive architecture with Data Re-Uploading and trainable input scaling Schuld et al. [2021], Sequeira et al. [2022] chose a simpler variational architecture with a single encoding layer. Both proposals used observables encoding the numerical preference of a given action and a softmax function to process these values and yield probabilities for each action, such that:

$$\pi(a|s,\theta) = \frac{e^{\beta\langle a\rangle_\theta}}{\sum_b e^{\beta\langle b\rangle_\theta}} \tag{3.30}$$

where $\langle a\rangle_\theta$ is the respective expectation value and β is an hyperparameter that controls the policies' greediness.

Furthermore, both references tested the algorithms in some of the same benchmark environments and the VQCs managed to solve them with a level of performance similar to or even better than DNNs. Furthermore, Sequeira et al. [2022] noted that even simpler circuits with decreased depth and fewer parameters could be used to solve the environments, claiming that the loss in expressivity compared to the VQCs used by Jerbi et al. [2021] could be compensated by better generalization properties and reduced parameter count. Although not as relevant to this work in particular, Jerbi et al. [2021] built RL environments based on the discrete logarithm problem and empirically verified a separation between classical and quantum agents, which was to be expected, since Shor's algorithm solves the discrete logarithm problem with an exponential speedup over the best known classical algorithm Shor [1999]. Furthermore, they empirically verified that input scaling improves the performance of the agent and explained that this is probably due to the increase in the expressivity of the VQC Schuld et al. [2021]. Moreover, Sequeira

et al. [2022] empirically verified that the VQCs were less prone than Neural Networks to plateaus in the optimization landscape, using the empirical Fisher Information matrix Ly et al. [2017].

3.6 Summary

This chapter started by contextualizing the reader about the history of quantum computing and its current state. Then, it introduced the circuit model of quantum computing, covering the essential concepts for the understanding of this work. Furthermore, it introduced Variational Quantum Circuits (VQCs), explaining their key aspects and why they are such a promising approach in the NISQ era. Finally, it tied everything together with the previous chapter on Reinforcement Learning by explaining and reviewing proposals of VQCs being used as function approximators in both policy and value-based RL algorithms.

Part II

Core of the Dissertation

Chapter 4

The Problem and Methodology

In the preceding chapters, we laid the groundwork by introducing the core concepts central to this work. We began by explaining the RL problem, discussing its mathematical framework and highlighting notable algorithms. The need for using function approximators to solve complex RL problems was emphasized. Of significant relevance to this work, the Deep Q-Learning algorithm was introduced. Afterward, the fundamentals of Quantum Computing were explored and VQCs were introduced as promising approaches in the NISQ era. Finally, we explained the intersection between RL and Quantum Computing, in particular using VQCs as function approximators in RL algorithms.

In this chapter, we aim to delve deeper into the specific problem this thesis addresses and the methodology employed.

4.1 The Problem

In Section 3.5, we discussed the main VQC-based RL algorithms. For the VQC-based Reinforce algorithms detailed in Section 3.5.2, Jerbi et al. [2021] employed a VQC with data re-uploading and assessed the performance of models in benchmark environments. Conversely, Sequeira et al. [2022] used a simpler, hardware-efficient ansatz without data re-uploading, comparing its performance and trainability against classical DNNs. Intriguingly, the impact of data re-uploading on model trainability remains unexplored. Given that data re-uploading increases the VQC's circuit depth and expressivity, it could have negative effects in the trainability, as suggested by McClean et al. [2018] and Holmes et al. [2022].

The picture is even less clear when it comes to VQC-based Deep Q-Learning algorithms, detailed in Section 3.5.1. Both Chen et al. [2020] and Lockwood and Si [2020] used simple VQCs without data re-uploading, centering their studies on performance in benchmark environments and comparisons with DNNs. On the other hand, Skolik et al. [2022] integrated data re-uploading and analyzed its impact on performance. Yet, to the best of our knowledge, no research has been done on the trainability of VQC-based

Deep Q-Learning models.

This work aims to directly address these questions. Specifically, we seek to investigate the impact of data re-uploading on both the performance and trainability of VQC-based Deep Q-Learning models. While data re-uploading can enhance model expressivity Schuld et al. [2021], increasing performance in VQC-based RL algorithms, it may also reduce the trainability. If this reduction is considerable, the practical use of data re-uploading becomes questionable. Conversely, if the reduction is minimal, then the increased expressivity more than compensates for it. We plan to empirically study this tradeoff, testing the algorithms in benchmark environments and evaluating both performance and trainability metrics. In the following sections, the implementation of the algorithm, the benchmark environments and the methodologies used during this research will be explained.

4.2 The Algorithm Implementation Choices

The first step toward completing the aforementioned goal is to implement VQC-based Deep Q-Learning in a quantum differentiable programming language. The algorithm to be implemented is the one from 3. So, all that's missing is choosing the programming language. There are several possibilities, the main ones being IBM's *Qiskit* Qiskit contributors [2023], Xanadu's *Pennylane* Bergholm et al. [2018] and TensorFlow-Quantum Broughton et al. [2020]. All of them have their pros and cons. Qiskit is more general and has direct integration with IBM's quantum machines. However, it is less abstract than the other languages and suffers from some performance problems, which would rather complicate this task. Pennylane, on the other hand, is designed to integrate seamlessly with machine learning libraries, which would be useful for this work. Nonetheless, the chosen language was tensorflow-quantum for a few reasons. It has integration with TensorFlow, a popular machine learning library, offers high-level abstraction in the design and training of hybrid quantum-classical algorithms and is compatible with high-performance quantum simulators Broughton et al. [2020]. Moreover, being built on TensorFlow, it is highly optimized for machine learning. Finally, Jerbi et al. [2021] and Skolik et al. [2022] implemented their VQC-based algorithms on TensorFlow-Quantum and even created a tutorial, which facilitates the beginning of the implementation.

Building upon the insights from the tutorial mentioned earlier, code was written using predominantly Python objects. This design choice promotes high customizability and adaptability, especially in components like the policy, the VQC architecture, the classical pre-processing technique, input scaling, output scaling, and more. A fitting analogy is to compare the code structure to Lego. It is designed in a way that allows for easy swapping of individual "pieces" or components. In addition to this, the training function is

parallelized, enabling simultaneous training of multiple agents. For those keen to delve deeper,the code is available in Coelho [2023].

4.3 Benchmark Environments

This study will focus on numerical analysis of the developed algorithms. In particular, they need to be tested in benchmark environments to analyse and compare them.

OpenAI Gym, an open-source Python library, offers a standardized API that bridges learning algorithms with a standard set of environments, providing a platform to analyse and compare RL algorithms. Two specific environments were chosen: CartPole-v0 and Acrobot-V1. The preference for these is twofold: firstly, they are established benchmark environments that have been frequently adopted in academic research for algorithm testing and comparison. Secondly, these environments strike a balance in complexity; they present sufficient challenges to test the robustness of VQC-based algorithms, yet remain tractable for experimental purposes. While CartPole has already been the subject of investigation in several studies, Acrobot presents a heightened degree of complexity and intriguingly, hasn't yet been solved using VQC-based Q-Learning.

4.3.1 CartPole-v0

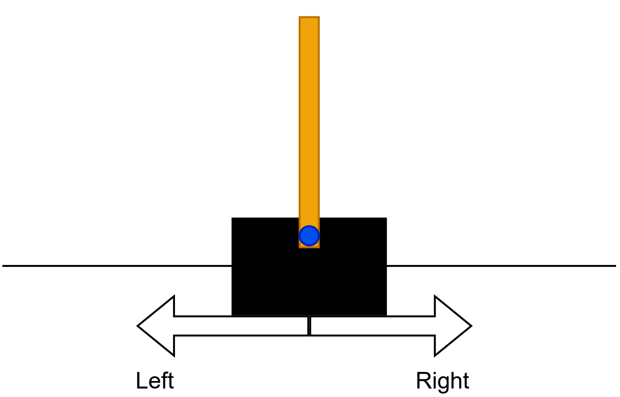


Figure 12: The CartPole Environment

A cart moves on a frictionless track. There is a pendulum placed upright on the cart attached by a joint. The goal is to balance the pole by applying forces in the left and right direction on the cart, see Figure 12. Consequently, the action can take the values {0,1} indicating whether the cart is pushed with a fixed force to the left or right, respectively. The state-space, on the other hand, is composed of 4 features:

Feature	**Range**
Cart Position	$[-4.8, 4.8]$
Cart Velocity	$]-\infty, \infty[$
Pole Angle	$[-0.418, 0.418]\ rad$
Pole Angular Velocity	$]-\infty, \infty[$

Table 1: Cartpole's state space

For the starting state, all the features are assigned a value between -0.05 and 0.05. The episode ends if one of three conditions is verified:

- The Pole Angle is greater than $\pm 0.214\ rad$
- The Cart Position is greater than ∓ 2.4
- Episode length is greater than 200

The reward function is very simple: the agent receives a $+1$ reward per time-step. Consequently, the third condition asserts that the maximum return possible in an episode is 200 if one considers $\gamma = 1$. This environment is considered solved if the average return over the previous 100 episodes is greater than 195.

4.3.2 Acrobot-v1

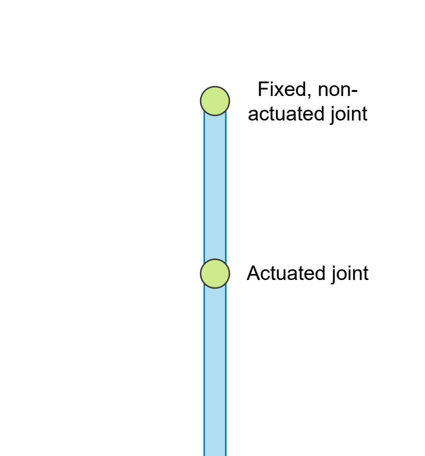


Figure 13: The Acrobot Environment

Two links are linearly connected to form a chain, with one end of the chain fixed. The joint between the two links is the one we can act upon and the goal is to apply torques to this joint to swing the free end of the chain above a given height while starting from the initial position of hanging downwards, see Figure 13. There are three possible actions:

Action	**Unit**
Apply -1 torque to the actuated joint	Torque (Nm)
Apply 0 torque to the actuated joint (do nothing)	No unit
Apply 1 torque to the actuated joint	Torque (Nm)

Table 2: Acrobot's action-space

There are two important angles for the definition of the state-space. First, θ_1 is the angle of the first joint, where an angle of zero indicates the first link is pointing directly downwards. Moreover, θ_2 is relative to the angle of the first link, where an angle of zero corresponds to having the same angle between the two links. Then, the state-space can be defined:

Action	**Range**
Cossine of θ_1	$[-1, 1]$
Sine of θ_1	$[-1, 1]$
Cossine of θ_2	$[-1, 1]$
Sine of θ_2	$[-1, 1]$
Angular Velocity of θ_1	$[-4\pi, 4\pi]$
Angular Velocity of θ_2	$[-9\pi, 9\pi]$

Table 3: Acrobot's state-space

The starting state is defined as follows. Each feature of the state is initialized uniformly between -0.1 and 0.1, such that the links are pointing downwards with some initial stochasticity. The goal is for the free end to reach a given height in as few steps as possible. Consequently, every step that does not reach the goal receives a reward of -1 and achieving it results in termination with a reward of 0. Moreover, the maximum number of steps in an episode is 500, after which the episode also terminates.

4.4 Methodologies for Analysing Model's Performance and Trainability

As outlined earlier, this study seeks to empirically assess the impact of data re-uploading on the performance and trainability of VQC-based Q-Learning models. To achieve this, we must establish methodologies for analyzing both metrics. The subsequent sections detail these methodologies.

4.4.1 Performance of a Model

In any RL problem, the goal is for the agent to find the optimal or a near-optimal policy, which translates to the behaviour that accumulates the most rewards. Consequently, performance is typically measured by the return models achieve in benchmark environments. However, there are several methodologies to measure this performance.

We are particularly concerned with keeping the randomness inherent to Deep Q-Learning to a minimum. The algorithm has several stochastic parts. For example, since the initialization of the parameters is random, some agents may be initialized in a good spot in the optimization landscape, such as near a good local minimum, while others may be initialized in poor spots. To evaluate models fairly and make results reproducible, the methodology for testing the performance of a model on a particular environment is the following: N agents are randomly initialized according to the model. These agents are then trained in the environment over M episodes. Should an agent solve the environment before completing the M episodes, its training ceases (i.e., no further updates to the model's parameters), but the agent continues interacting with the environment using its learned knowledge until all episodes conclude. The returns from each episode are collected for all agents. Ultimately, we average the returns across all N agents for each episode, compute the standard deviation, and plot the results. An advantage of this methodology is that the randomness inherent to Q-Learning is mitigated by using mean returns of N agents.

However, how is performance compared between different models? Imagine we apply this methodology to two different models in a particular environment. How can we claim one of the models is superior to the other when it comes to performance? Let's use Figure 14 as an example.

Let's begin by analysing the two models from Figure 14a. In this scenario, it is easy to decide which is the best performing model. The blue model achieved an average return much higher than the red model. In other words, the agents generated from the former were able to derive a policy much better than the agents derived from the latter. Consequently, the blue model is the best performing model out of the two.

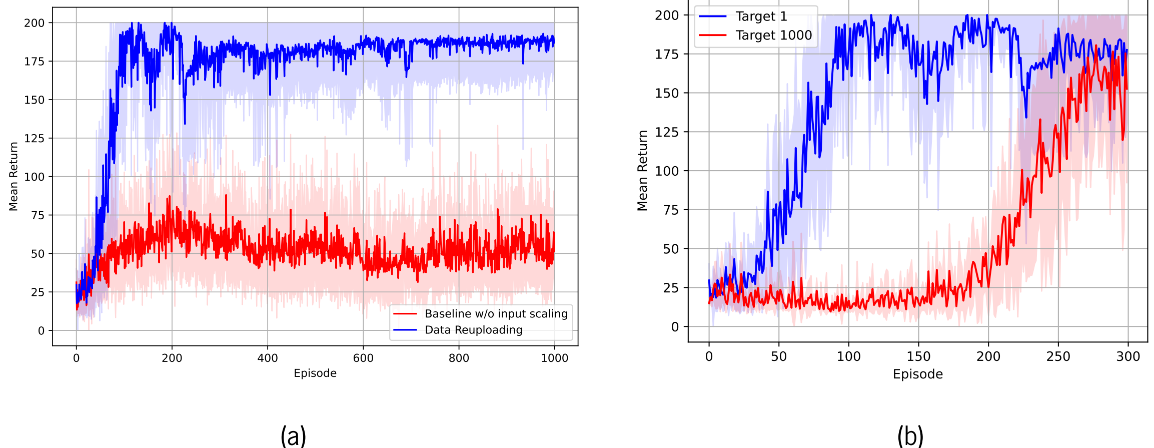


Figure 14: Both figures represent the performance of two random models on the CartPole-v0 environment. The thick lines are the mean returns for each model (in this case, 5 agents were initialized per model) and the shaded areas the standard deviation. This is how the performance of all models will be plotted.

Figure 14b represents a different scenario. In this case, both models were able to achieve a similar average return. Nonetheless, the blue model was able to achieve that average return much sooner than the red model. This reveals an advantage in sample complexity, since it requires less samples to learn a good policy. Consequently, the blue model is also the best performing model out of the two in this scenario.

However, it is important to note that some cases are not as easy to analyse. For example, some models may achieve a very high return with fewer samples but then keep unlearning, with the average return decreasing as training goes on. Nonetheless, we are not particularly interested in being able to perfectly distinguish the performances of all models. We are simply concerned with being able to gauge considerable differences in performance between models.

The standard deviation is particularly interesting for measuring the instability of the models. If it is substantial, then it means different agents initialized from the same model performed very differently on the environment. Some may have achieved high returns during training, while others might have gotten stuck in low returns. Conversely, if it is small, then it means that the agents achieved similar returns throughout training. Consequently, the best model would be one that achieves and maintains the highest possible return in the fewest amount of episodes and exhibits minimal standard deviation.

4.4.2 Trainability of a Model

There are several methods for analyzing a model's trainability. For instance, Sequeira et al. [2022] used the empirical Fisher Information matrix Ly et al. [2017]. Another prevalent and straightforward approach is examining the gradient of the cost function, defined in Equation 3.25.

In the training of machine learning models, including Neural Networks (NNs) and Variational Quantum Circuits (VQCs), parameters are updated iteratively using variations of Gradient Descent. Essentially, an approximation of the gradient is calculated, and parameters are updated in an inverse proportion, aiming to minimize the cost function. If these partial derivatives are large, the updates to parameters are significant, potentially causing unstable training. Conversely, if these derivatives are small, then the parameter updates are also small, which leads to the opposite problem, in which the model gets stuck on the optimization landscape. While both extremes are problematical, we are particularly concerned with the latter. Let $L(\boldsymbol{\theta})$ be a hardware-efficient VQC-based cost function as defined in Subsection 3.3.3. The Barren Plateau Phenomenon, detailed in Subsection 3.3.4, asserts that

$$Var[||\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})||] \in \mathcal{O}\left(\frac{1}{\alpha^n}\right), \ \alpha > 1 \tag{4.1}$$

where $||\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})||$ is the gradient norm.

As a result, we will analyse the gradients of the cost function for each model to draw conclusions about their trainability using the following methodology: When using the methodology for measuring model's performance described in the previous section, we also store the gradient of the cost function with respect to the parameters at each training step, which is every step where the parameters are updated. Subsequently, we compute the norm of these gradients for every training step. To mitigate the effects of stochasticity, we calculate the average norm of the gradients across the N agents and the variance of these norms at each training step. However, since agents cease training once the environment is solved, different agents have a different number of training steps. As a result, we limited our procedure to only encompass the steps up until the first agent successfully solved the environment. To facilitate the visualization of the results, a rolling average of the last 100 training steps is used for both the norm of the gradients and the variance of the norms.

4.5 Summary

In this chapter, we started by defining the precise issue addressed in this research. The primary focus is centered on investigating the performance and trainability of VQC-based Deep Q-Learning models, with

a special emphasis on understanding the potential implications of data re-uploading on those metrics. Furthermore, we explained the reasoning behind the selection of Tensorflow-Quantum as the preferred language for implementing the VQC-based Deep Q-Learning models. Subsequently, we introduced the benchmark environments chosen for testing the models and outlined the methodologies that will be employed to analyse their performance and trainability.

Chapter 5

Results and Discussion

The previous section explained the benchmark environments that will be used to test the Deep Q-Learning models throughout this work. Moreover, the methodologies for analyzing a model's performance and trainability were explained. In this chapter, we aim to test different VQC-based Deep Q-Learning models on the aforementioned environments and analyze their performance and trainability.

5.1 Replication of Skolik et al. [2022]'s results

Having implemented the algorithm in TensorFlow-Quantum, it was imperative to test it. Since this work is going to build upon the paper by Skolik et al. [2022], it is only natural that the first step should be to replicate their methods and compare the results. If the algorithm is well implemented, then the results should be equivalent.

Skolik et al. [2022]'s VQC solved the CartPole environment using a VQC-based algorithm with the defining characteristics (see Figure 15):

- **Classical Pre-Processing and Data Encoding method** - To allow for the encoding of continuous features, each feature was passed through an arctan function which normalized it to the interval $[-\pi/2, \pi/2]$. Then, a Pauli-X rotation with the result as the angle was used to encode the data into the VQC. The authors named this technique *Continuous Encoding*. Consequently, each feature s_i of the input state vector $\boldsymbol{s}$ was encoded as $R_X(arctan(s_i))$. Since CartPole state-space has four features which have to be encoded by different qubits, the VQC has four qubits.

- **Trainable Input Scaling** - When applicable, each feature s_i of the state-space s is multiplied by a classical trainable weight Pérez-Salinas et al. [2020], such that the encoding is given by $R_X(arctan(s_i \times \lambda_i))$, where λ_i is the input scaling weight.

- **Ansatz** - The ansatz is a hardware efficient one, composed of two parametrized rotations over the y

and z axis per qubit, followed by a cascade of CZ gates in a circular setup (all nearest-neighbours are connected and then the last qubit is connected to the first one).

- **Data Re-Uploading** - When applicable, to increase the expressivity of the quantum model, the data encoding block is repeated several times throughout the circuit. Specifically, it is repeated once every layer, such that the VQC is expressed as:

$$U(s, \boldsymbol{\lambda}, \boldsymbol{\theta}) = W^{L+1}(\theta^{L+1})S(s, \lambda^{L})W^{L}(\theta^{L})...W^{2}(\theta^{2})S(s, \lambda^{1})W^{1}(\theta^{1}) \tag{5.1}$$

  where s is the input state, $S(s, \lambda^i)$ is the data encoding block, $W^i(\theta^i)$ is the i-th parametrized trainable block and L the number of layers.

- **Measurement and Trainable Output Scaling** - The expectation value of one observable has to be measured for each possible action. Since CartPole's action-space has two possible actions, two observables need to be measured. In particular, Skolik et al. [2022] used $\langle Z_0 Z_1 \rangle$ for the "left" action and $\langle Z_2 Z_3 \rangle$ for the "right" action, where Z is the Pauli-Z observable and the numbers 0 to 3 indicate the qubit in which the observable is measured. Moreover, *output scaling* is used, meaning each of the two expectation values is multiplied by a classical trainable weight to allow for arbitrary Q-values, as explained in section 3.5. Consequently, the two Q-values are given by:

$$Q(s, left) = \langle 0^{\otimes n} | U_{\boldsymbol{\theta}}^{\dagger}(s) Z_0 Z_1 U_{\boldsymbol{\theta}}(s) | 0^{\otimes n} \rangle \times w_0 \tag{5.2}$$

$$Q(s, right) = \langle 0^{\otimes n} | U_{\boldsymbol{\theta}}^{\dagger}(s) Z_2 Z_3 U_{\boldsymbol{\theta}}(s) | 0^{\otimes n} \rangle \times w_1 \tag{5.3}$$

- **Parameters Initialization** - The rotational parameters are randomly initialized by sampling from a uniform distribution between 0 and π. Moreover, the input and output scaling weights are initialized as 1s.

- **Cost Function** - The Mean Squared Error (MSE) cost function is used, according to algorithm 3.

Skolik et al. [2022] tested the effect of data re-uploading and trainable input scaling in the performance of the agents in the CartPole environment. In particular, they verified that agents which used just one of the techniques couldn't solve the environment in up to 5000 episodes. However, when both techniques were used together, the models were able to solve the environment using a set of optimal hyperparameters and sub-optimal hyperparameters (as defined by the authors after a search over some of the hyperparemeters). According to the authors, this result demonstrates the importance of both these techniques in increasing the expressivity of the VQC, as described in Schuld et al. [2021]. However, we are also concerned with the

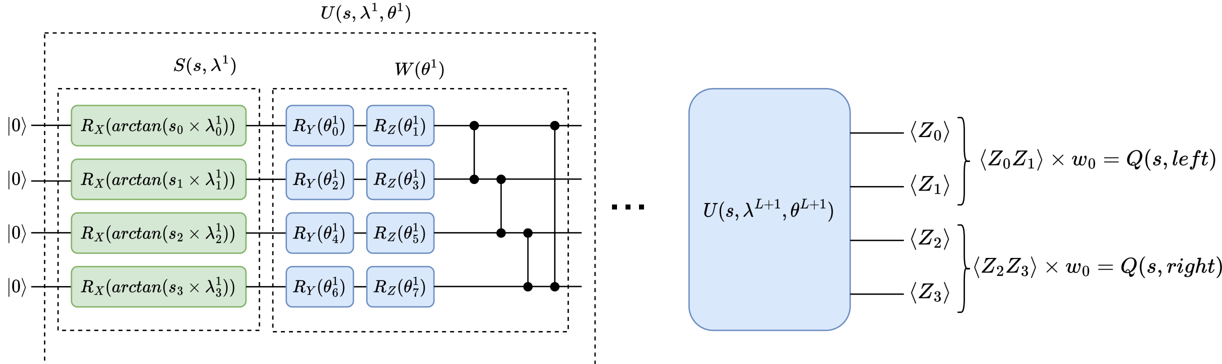


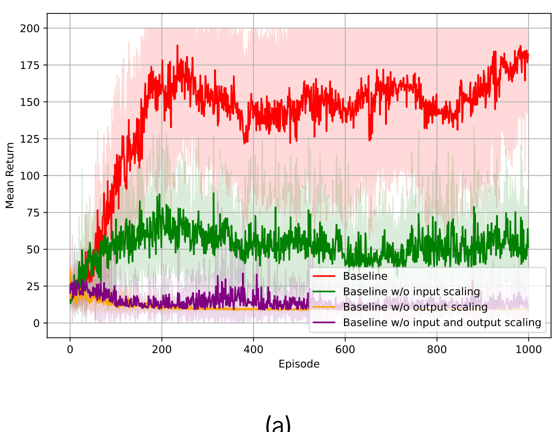
Figure 15: The functioning of the VQC-based Deep Q-Learning algorithm used by Skolik et al. [2022] to solve the CartPole environment. When Data Re-Uploading is used, $U(s, \lambda, \theta)$ is repeated several times. Otherwise, just $W(\theta)$ is repeated.

effect of output scaling on the performance of the agents. Consequently, we implemented the VQC-based Deep Q-Learning algorithm with the 8 different combinations of:

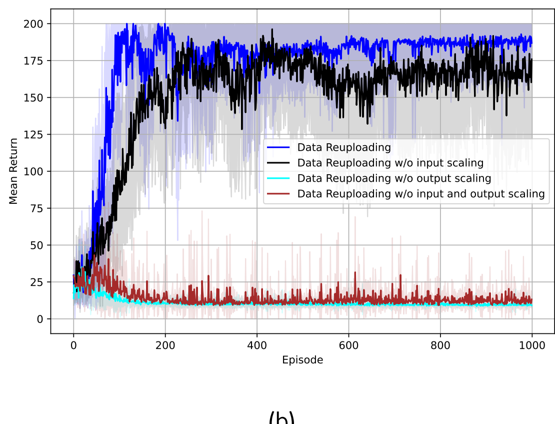
- Using Data Re-Uploading or not (a model that doesn't make use of data re-uploading will be referred to as baseline)
- Using trainable input scaling or not
- Using trainable output scaling or not

Since the goal was to replicate the work by Skolik et al. [2022], all the models were trained using their set of optimal hyperparameters, which can be seen in Table 6, in the CartPole-v0 environment. Furthermore, each agent was composed of 5 layers of the circuit architecture from Figure 15. Figure 16 shows the results after using the methodology described in section 4.4.1 to analyse the performance of the models, which were trained using a noiseless simulator.

Models without trainable output scaling (represented by the purple and orange lines in Subfigure 16a and by brown and cyan lines in Subfigure 16b) perform very poorly, maintaining an average return just above 0, while models that use the technique can perform well and even solve the environment. This result underscores the significance of matching the outputs of the VQC to the range of the optimal Q-values. However, there is also a clear problem with using trainable output weights: it becomes difficult to differentiate the contributions of the quantum model from the classical trainable weights. There are some ways to overcome this problem, such as multiplying the expectation values by scalar weights. Nonetheless, Skolik et al. [2022] show that, while it is possible to achieve a good level of performance using that technique, agents that use trainable output scaling still perform better.



(a)



(b)

Figure 16: Comparison of baseline (Subfigure 16a) and data re-uploading (Subfigure 16b) models with and without trainable input and output scaling in the CartPole-v0 environment. The optimal set of hyper-parameters from Skolik et al. [2022] was used, see Table 6. The thick lines are the average return over all the 10 agents for each model, while the shaded areas indicate the standard deviation of the return over all agents. If an agent solves the environment (average reward over the last 100 episodes ≥ 195), training is stopped.

From Subfigure 16a, it is also possible to see that the baseline model without input scaling doesn't perform well, achieving an average return close to 50 throughout training (green line). However, when trainable input scaling is used (red line), the performance increases drastically. To understand this difference in performance, let's review the effect of trainable input scaling on a model. The baseline model consists of a single encoding block followed by L parametrized blocks, where L is the number of layers. Consequently, according to Schuld et al. [2021], this VQC can only fit a Fourier series with a single non-zero frequency (a sine function). Furthermore, trainable input scaling simply scales the frequency spectrum the VQC has access to, which allows for an adaptive frequency matching between the function that the VQC outputs and the target function. In other words, the baseline model with trainable input scaling can still only fit Fourier series with a single non-zero frequency, except this frequency is scaled. Thus, one can conclude that the optimal Q-function of this environment is relatively simple, since the baseline model with input scaling achieved a considerable level of performance, meaning it is possible to approximate this Q-function with a Fourier series with a single non-zero frequency. Nonetheless, since the baseline model without trainable input scaling performs poorly, there is probably a mismatch in the non-zero frequency of the function the VQC outputs and the optimal Q-function.

Moreover, Figure 16 clearly shows that data re-uploading models perform better than baseline models. While both data re-uploading models that did not use trainable output scaling performed very poorly (brown

and cyan lines in subfigure 16b), the two which used (black and blue lines) performed better than the correspondent baseline models. First, the model which did not use trainable input scaling also performed well, unlike the baseline model without trainable input scaling, which only managed to obtain an average return of approximately 50 throughout training. This shows the importance of data re-uploading. Since it increases the frequencies accessible to the VQC, it can represent a wider range of functions Schuld et al. [2021]. However, even though this VQC has access to a frequency spectrum with more non-zero frequencies, it seems like there is still a frequency mismatch between the optimal Q-function and the function the VQC outputs, since the data re-uploading model with trainable input scaling performs better. Actually, this model achieves the best performance of all the tested models as expected, since it has access to more frequencies and is also capable of adaptive frequency matching, thus being the most expressive VQC and the one capable of best approximating the optimal Q-function.

Finally, it is important to note that the algorithm is very unstable, as one can see by the considerable standard deviations for each of the tested models. This result is due to a couple of reasons. As explained in section 2.6, DQN is inherently unstable, since it uses the *deadly triad*, as Sutton and Barto [2018] called it: Bootstrapping, TD-Learning and off-policy data. Moreover, as also explained in section 2.6, the targets in DQN are obtained from the target network, which is a network with frozen parameters that are updated every some steps to match the parameters of the online network. This technique stabilizes training by making the targets more stationary, thus facilitating the optimizer's job. However, the optimal set of hyperparameters found by Skolik et al. [2022] used in this replication study updates the parameters of the target VQC to the parameters of the online VQC at every interaction with the environment (every single step). Consequently, in practice, there is no target VQC since both VQCs always have the same parameters, which makes the algorithm even more unstable. As we will see in section 5.5, this instability has several consequences.

In conclusion, these results match the results obtained by Skolik et al. [2022], proving the significance of the data re-uploading, trainable input scaling and trainable output scaling techniques.

5.2 Performance of the Universal Quantum Classifier

In the previous section, we observed that data re-uploading and trainable input/output scaling indeed increase the performance of the tested VQC-based Deep Q-Learning models in the CartPole-v0 environment. However, these results might be VQC-dependent. In other words, these results may be true only for the models employed by Skolik et al. [2022]. Thus, it is imperative to test other VQCs with different ansatzes.

Hence, in this section, we introduce and test a different type of data re-uploading VQC that also uses trainable input/output scaling - the *Universal Quantum Classifier (UQC)* Pérez-Salinas et al. [2020].

The UQC is particularly interesting because it allows the encoding of a given input vector using an arbitrary number of qubits. In other words, it allows us to control the width of the circuit, while the Skolik et al. [2022]'s models always have a number of qubits that grows linearly with the number of features of the input vector. We will start by delving into the single-qubit UQC and analyzing its performance in the CartPole-v0 environment.

5.2.1 The Single-Qubit Universal Quantum Classifier

The authors of Pérez-Salinas et al. [2020] introduce the single-qubit UQC and compare it with a NN with a single hidden layer. To understand the architecture of the UQC, let's consider some input vector $\vec{s} = (s_0, s_1, ..., s_{n-1})$ and some vector $\vec{\theta}$ that contains all the parameters to be updated by the classical optimizer. The single-qubit UQC is defined as:

$$U(\vec{s}, \vec{\theta}) = \prod_{i=0}^{L-1} L_i(\vec{s}, \vec{\theta}_i) \tag{5.4}$$

where L is the number of layers and L_i the fundamental gate defined as:

$$L_i(\vec{s}, \vec{\theta}_i = (\vec{w}_i, \alpha_i, \varphi_i)) = R_y(2\varphi_i)R_z(2\vec{w}_i \cdot \vec{s} + 2\alpha_i) \tag{5.5}$$

where

$$\vec{w}_i \cdot \vec{s} = \sum_{j=0}^{n-1} w_{ij} s_j \tag{5.6}$$

This model is trained over the parameters $\vec{w}, \vec{\alpha}$ and $\vec{\varphi}$.

One can see from Equations 5.4 and 5.5 that this architecture uses data re-uploading, since the input vector $\vec{s}$ is encoded in every single gate L_i. Moreover, since the weights $\vec{w}$ and $\vec{\alpha}$ are trainable, this architecture also makes use of trainable input scaling.

The authors proved that such a quantum circuit can approximate any classification function up to arbitrary precision, using the Universal Approximation Theorem (UAT) of NNs Hornik [1991]. Moreover, this circuit can encode an input vector independently of the number of features. For instance, a CartPole's state could be encoded into a single qubit, since the R_z rotation angle is given by the dot product between the state and some weight vector plus a bias. One simply has to ensure that the weight vector has a dimension that matches that of the input vector. It is also important to note that, even though the R_y and R_z rotations are used, the only requirement is that two orthogonal rotation axis are used. Hence, it also works with R_z and R_x, for example.

To the best of our knowledge, the single-qubit UQC was never used as a function approximator in a RL algorithm. Thus, we decided to test its performance in the CartPole-v0 environment to see if this universal approximant is capable of solving the environment using a single-qubit. However, some choices had to be made for the architecture to be suitable for the task:

- **Classical Pre-Processing**: We experimented with different classical pre-processing techniques. Whether we used continuous encoding from Skolik et al. [2022] or just directly fed the data to the UQC, the performance of the models was remarkably similar. This could be because the UQC may adjust its weights and biases that best process the data. The UQC is very similar to NNs, as Pérez-Salinas et al. [2020] reference several times throughout the paper, and in particular $\vec{w}$ and $\vec{\alpha}$ are the analogs of the weights and biases of NNs. In NNs trained in the CartPole-v0 environment, while pre-processing the data may have some benefits, it isn't strictly required. Hence, since we saw no difference in performance, we opted for the most simple method of feeding the UQC the raw data.

- **Observables**: The single-qubit UQC, being a single-qubit model, posed a unique challenge. Our approach to VQC-based Deep Q-Learning demands measuring the expectation value of one observable per action. However, given that the UQC has one single qubit, it is only possible to measure one observable per circuit execution. To solve this issue in the CartPole environment, we used $\langle Z \rangle$ for the "left" action and $\langle X \rangle$ for the "right". In other words, two different Pauli observables were measured on the same qubit. However, the drawback is the need to execute the quantum circuit twice: once for each observable.

- **Trainable Output Scaling**: For the same reasons mentioned for the Skolik et al. [2022]'s architectures, in particular, being able to match the range of the output values of the quantum model to the range of the optimal Q-values for the environment, trainable output scaling is used.

- **Parameter Initialization**: There are several ways to initialize the parameters of the UQC. Taking into consideration the similarities between the UQC and classical NNs, as Pérez-Salinas et al. [2020] mentioned throughout the paper, we opted for an initialization strategy that is typically used in NNs. More precisely, we initialize the weights $\vec{w}$ by sampling from a normal distribution with mean 0 and with a low standard deviation of 0.01 and the biases $\vec{\alpha}$ as zeros. Finally, taking inspiration from Skolik et al. [2022], the rotational parameters $\vec{\varphi}$ were initialized uniformly between 0 and π. The output scaling weights were initialized as ones. Nonetheless, an informal search was performed on other typical initialization strategies, but none achieved superior performance in the environment.

- **Set of Hyperparameters**: For an accurate comparison between our UQC model and the other tested quantum models from Skolik et al. [2022], we decided to use the same set of hyperparameters, see Table 7, ensuring that any performance differences can be attributed more to the models themselves than to the hyperparameters.

Figure 17 compares the performance of the single-qubit UQC and the best performing model from Skolik et al. [2022] (the blue model with data re-uploading and trainable input and output scaling from Figure 16b), which from now on will be named Skolik Data Re-Uploading model.

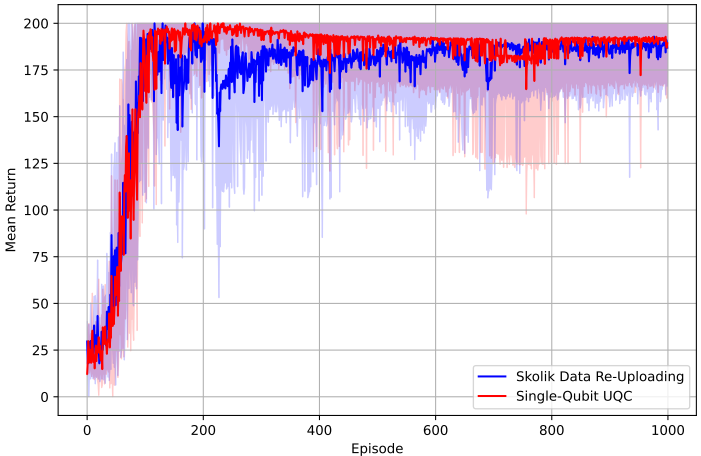


Figure 17: Analysis of the performance of the single-qubit UQC and Skolik data re-uploading models in the CartPole-v0 environment following the methodology defined in Section 4.4.1. 10 agents were initialized from each model. The full set of hyperparameters can be seen in Table 7.

From Figure 17, the single-qubit UQC appears to reach the highest possible average return faster than the data re-uploading model and with a lower standard deviation. This suggests superior performance. However, we should be cautious in our interpretation. The observed differences might be influenced by the statistical variance tied to our performance testing method. Given that we initialize only 10 agents for each model (due to limitations in computational resources), even one poorly initialized agent can skew the average return and increase the standard deviation. Nonetheless, it is fair to say that the single-qubit UQC is able to generate near-optimal policies in the CartPole environment.

This result has an intriguing consequence. A product state is a quantum state that can be factorized as the product of states of its individual parts. A multi-qubit system with no entanglement, for instance, is a product state. A single-qubit quantum circuit is also, by definition, a product state. Product states are classically simulable, since their description does not grow exponentially with the number of qubits. Hence, if the single-qubit UQC managed to generate near-optimal policies in the CartPole environment,

then it doesn't showcase any quantum advantage. After all, we can replicate its behaviour with classical means. However, this observation carries a caveat: the CartPole environment is very simple. More intricate environments, specially those inspired by complex physical phenomena, might demand quantum resources such as entanglement to generate good policies.

5.2.2 The Multi-qubit Universal Quantum Classifier

It is possible to generalize from the single-qubit UQC to a multi-qubit UQC. The main reason for doing so is that the UQC allows for the choice of an arbitrary number of qubits. Thus, it would be interesting to study if increasing the number of qubits also leads to an increase in performance. That is the goal of this section.

Nonetheless, a data encoding technique has to be defined. We experiment with two different types of encoding:

- **Full Encoding**: Using full encoding, the whole input vector is encoded into all the qubits. For instance, consider the CartPole environment which has four features per state $\vec{s} = [s_1, s_2, s_3, s_4]$ and a full encoding multi-qubit UQC with five qubits. Then, the whole state vector $\vec{s}$ would be encoded into the five different qubits. Consequently, the number of parameters grows linearly with the number of qubits.

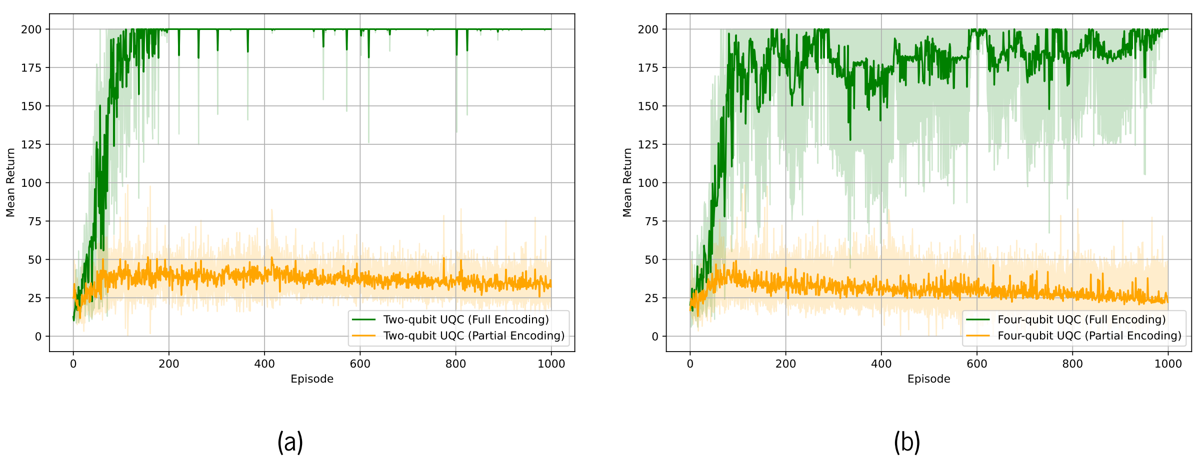
- **Partial Encoding**: In this data encoding technique, we divide the number of the input vector features by the number of qubits used and encode a different subvector in each qubit. Let's consider the CartPole environment once again. Then, if a partial encoding multi-qubit UQC with two qubits is used, the first two features are encoded into the first qubit and the last two into the second qubit. A limitation of this method is that the number of qubits has to be smaller than or equal to the number of features of the input vector.

The reason for using these two types of encoding is two-fold. On the one hand, these two techniques allow us to study the impact of introducing entanglement on the performance of the models in the CartPole-v0 environment. We will see how throughout this section. On the other hand, the Full Encoding technique allows us to encode a given input vector in an arbitrary number of qubits that may even be greater than the number of features of the input vector. Thus, we may study how the performance and trainability behave as the number of qubits increases, see Section 5.6.

We also modified the observables used in the model. In the single-qubit version of the UQC, the expectation values of the Z and X observables were measured on the same qubit, serving as Q-values that

encode the two potential actions in the CartPole environment (after being multiplied by a classical trainable weight - output scaling). However, in the multi-qubit UQC, the necessity to measure the observables on the same qubit across different circuit executions is eliminated. Consequently, in the two-qubit UQC, we designate the expectation value of the Pauli-Z observable on the first qubit, denoted as $\langle Z_0 \rangle$, to represent the "left" action, and $\langle Z_1 \rangle$ the "right" action, where 0 and 1 are the indices of the two qubits. Extending this approach to the four-qubit UQC, we employ $\langle Z_0 Z_1 \rangle$ as the observable indicating the "left" action, and $\langle Z_2 Z_3 \rangle$ for the "right" action, a strategy in line with the methodology outlined in the work of Skolik et al. [2022].

In the previous section, we observed that the single-qubit UQC, which is a product state, is capable of solving the CartPole-v0 environment. Expanding on those results, we would like to see if multi-qubit UQCs, without entanglement, are also capable of solving the environment. Thus, we first start by testing how the two-qubit UQC and the four-qubit UQC perform without entanglement with the two different data encoding techniques, see Figure 18.



(a) (b)

Figure 18: Performance analysis of the two-qubit (see Subfigure 18a) and four-qubit (see Subfigure 18b) UQCs using the Partial and Full encoding techniques without entanglement. 10 agents were initialized from each model. The full set of hyperparameters can be seen in Table 8.

Analyzing Figure 18, it is clear that the different encoding techniques have a significant impact on the performance of the model when entanglement is not used. This result has a simple but interesting explanation. Let's consider the two-qubit UQC.

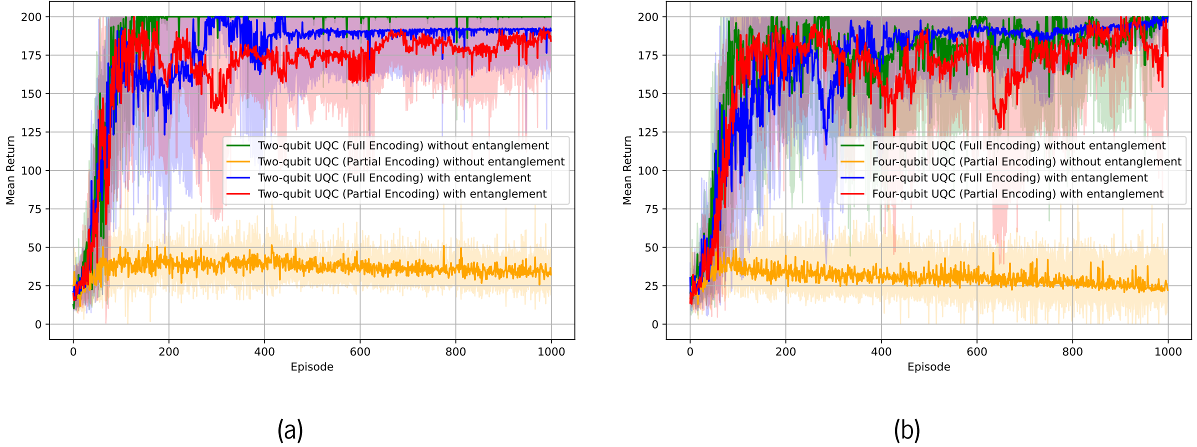
When Partial Encoding is used, see the orange line of Subfigure 18a, the first two features of the CartPole's state-space are encoded into the first qubit, and the last two are encoded into the second qubit (remembering that CartPole's state-space has four features). Moreover, the expectation value of the Z observable is measured on the first qubit as the Q-value for the "left" action and the expectation value of

the Z observable is measured on the second qubit as the Q-value for the "right" action. Thus, it is clear to see why this model is not capable of generating optimal or near-optimal policies. The Q-values for each of the two actions are approximated using just a subset of the features of the state-space. For instance, the Q-values for the "left" action are being approximated using only the cart's position and the cart's velocity, without any information about the pole angle and the pole angular velocity, see Table 1. Thus, the models are capable of learning a little bit and achieving an average return of 30 but are not capable of learning near-optimal policies, since some information is missing. Nonetheless, it is important to note that these models could still be able to achieve near-optimal performance in an environment where certain features are not as relevant as others. That does not seem to be the case with CartPole-v0.

On the other hand, when Full Encoding is used, models without entanglement are capable of solving the environment, similar to what was observed for the single-qubit UQC. This result is expected, since each qubit, which is a universal function approximant, has full information about the CartPole's state-space, thus being capable of approximating with good precision the optimal Q-value of the corresponding action. Consequently, the models, which are composed of two of these qubits each approximating the Q-value of one of the actions with good precision, generate a good approximation of the optimal Q-function and, hence, near-optimal policies. Surprisingly though, this model is the most stable of all the models tested so far, with small deviations from the maximum average return of 200 achieved around episode 200.

Analyzing Subfigure 18b, the results seem to be similar for the four-qubit models. The Partial Encoding model is not capable of learning near-optimal policies, since the Q-values for the two actions are approximated using partial information about the state-space. When Full Encoding is used, the model achieves the highest possible average return, although this model appears to have a worse performance than the Full Encoding two-qubit UQC since it takes more episodes to achieve the same average return and also suffers from a higher standard deviation. Thus, interestingly, it appears that increasing the number of qubits and, consequently, the number of parameters, does not necessarily lead to an increase in performance.

To see if the performance changes when entanglement is used, we added a circular layer of CZ gates after every processing layer and tested the models in the CartPole-v0 environment, see Figure 19. For a comprehensive analysis, we have also included comparisons with the multi-qubit UQC models without entanglement.



(a) (b)

Figure 19: Performance analysis of the two-qubit (see Subfigure 18a) and four-qubit (see Subfigure 18b) UQCs using the Partial and Full encoding techniques with and without entanglement. 10 agents were initialized from each model. The full set of hyperparameters can be seen in Table 8.

The first notable observation from Figure 19 is that the models that use the Partial Encoding technique and entanglement can generate near-optimal policies, unlike the models that do not use entanglement. Although interesting, it is not surprising. Even though each qubit only encodes a subset of the features of the state-space, entanglement introduces strong correlations between the qubits, such that the output state depends on all the features. Thus, the Q-values for each action are approximated using all the available information about the state-space. In fact, this is similar to what the models from Skolik et al. [2022] do. They encode each feature into a different qubit and then use entanglement to correlate all of the subsystems.

Another interesting observation is that, for the Full Encoding models, it seems that introducing entanglement does not lead to a considerable increase in performance. In fact, for the two-qubit UQC, see Subfigure 19a, introducing entanglement decreased the model's performance. For the four-qubit UQC, see Subfigure 19b, it is not so clear which model performs the best at the end of the training, but the model without entanglement learns much faster than the model with entanglement.

Finally, having tested all of the different models, Figure 20 compares the performance of the best-performing models obtained in the CartPole-v0 environment. Consequently, we included the Skolik Data Re-Uploading Model, the Single-Qubit UQC, and the Multi-Qubit UQC without entanglement.

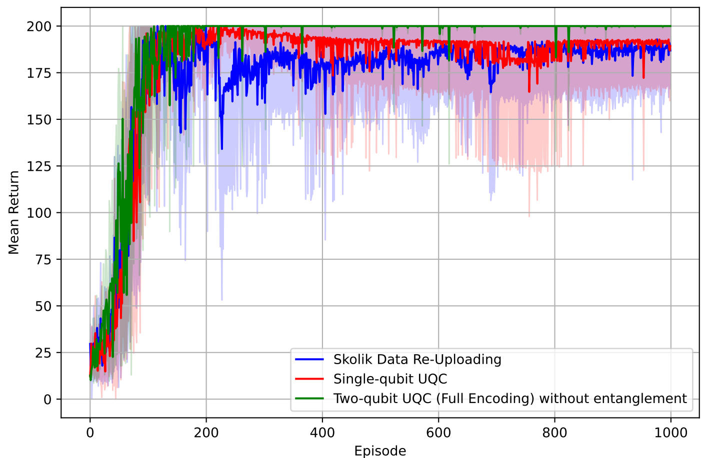


Figure 20: Performance Analysis of the best-performing models in the CartPole-v0 environment. In concrete, the Skolik Data Re-Uploading model, the Single-Qubit UQC and the Full Encoding Multi-Qubit UQC without entanglement. 10 agents were initialized from each model.

Interestingly, Figure 20 shows that the best-performing model is the Full Encoding Two-Qubit UQC without entanglement. Thus, it seems that increasing the number of qubits of the UQC from one to two increased the average return obtained in the environment and improved the model's stability, as one can see by the small deviations from the average return. However, increasing it beyond two qubits negatively impacted the performance of the models.

Moreover, out of the three different models, the Skolik Data Re-Uploading appears to be the worst-performing model both in terms of the average return obtained as well as in stability.

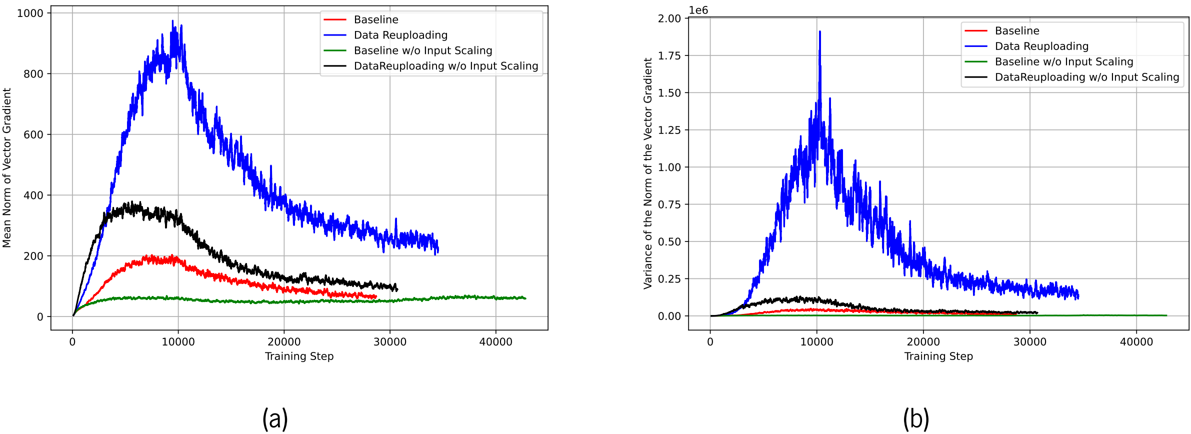
5.3 Trainability Analysis of Skolik et al. [2022]'s models

In the previous sections, we analyzed the performance of the models from Skolik et al. [2022] and the single/multi-qubit UQC. Nonetheless, our goal is to study the trainability of these models by observing the norm and the variance of the gradients throughout training. That will be done from this section onwards.

We start by applying the procedure described in 4.4.2 to the four models from Figure 16 which used output scaling (the red, green, blue and black lines), since the others achieved a level of performance so poor that further study is not relevant. The results can be seen in Figure 21.

From Figure 21, it is clear that both the norm of the gradients and the variance of the norm appear to be closely tied to the model's performance in the environment. The model with the poorest performance among the four (as shown in Figure 16) also exhibits the lowest average gradient norm and variance of that norm, and vice versa. Moreover, the difference in the gradients' norm and its variance is substantial

between the different models, but particularly between the data re-uploading model with input scaling and all the other models.



(a)

(b)

Figure 21: Comparison of the gradients of the baseline and data re-uploading models with and without trainable input scaling in the CartPole environment. Subfigure 21a shows the mean norm of the vector gradient throughout training and Subfigure 21b the variace in the norms of the vector gradients throughout training. The optimal set of hyperparameters from Skolik et al. [2022] is used, see Table 6. Moreover, 10 agents are initialized from each model. If an agent solves the environment, training is stopped. These metrics are stopped after the first agent solves the environment, hence why some curves are shorter than others.

The data re-uploading model with input scaling, depicted as the top-performing model in Figure 16, is noteworthy in several ways. Despite its greater circuit depth and enhanced expressivity, it exhibits the highest variance in the gradient norm among all models. This promising finding is in stark contrast to our initial expectations derived from Holmes et al. [2022], which suggests a trade-off between expressivity and trainability. In particular, Barren Plateaus are to be expected when using hardware-efficient ansatzes with high expressivities. However, in this case, the most expressive model is also the one with the highest gradients' magnitude and variance. In fact, at a certain point during training, this model achieved a gradient norm of 1000 and a variance in this norm of $2,000,000$! This result is the exact opposite of the Barren Plateau Phenomenon. The norm of the gradients and its variance are so high that parameter updates should be very large and lead to unstable training. Nonetheless, this model achieves the best performance. It is crucial, however, to set these observations in their proper context: they are specific to RL and the CartPole-v0 environment, and any broader generalizations would be premature. Still, these results warrant further investigation.

Finally, examining Figure 21 also reveals intriguing patterns in the training process. The norm of the

gradients and its variance display similar trajectories: both increase in the early stages of training, achieve a maximum value, and then start to decrease as training progresses. However, from Figures 16 and 21, determining the exact performance at which the model's gradients begin to decrease is challenging due to their distinct x-axes. While Figure 16 uses episodes for the x-axis, Figure 21 employs training steps. Thus, there isn't a direct correlation, as each episode can entail an arbitrary number of steps. To address this, we opted to analyze three randomly-selected agents from the Data Re-uploading and Baseline models, plotting both performance and gradient norm against training steps. This choice, over averaging returns and gradients across all 10 agents, offers a clearer visual representation of the potential performance disparities between agents within the same model, highlighting the inherent instability of the algorithm. Furthermore, the three agents displayed for each model represent the ten agents well, as similar results are observed across all of them. These results are illustrated in Figure 22.

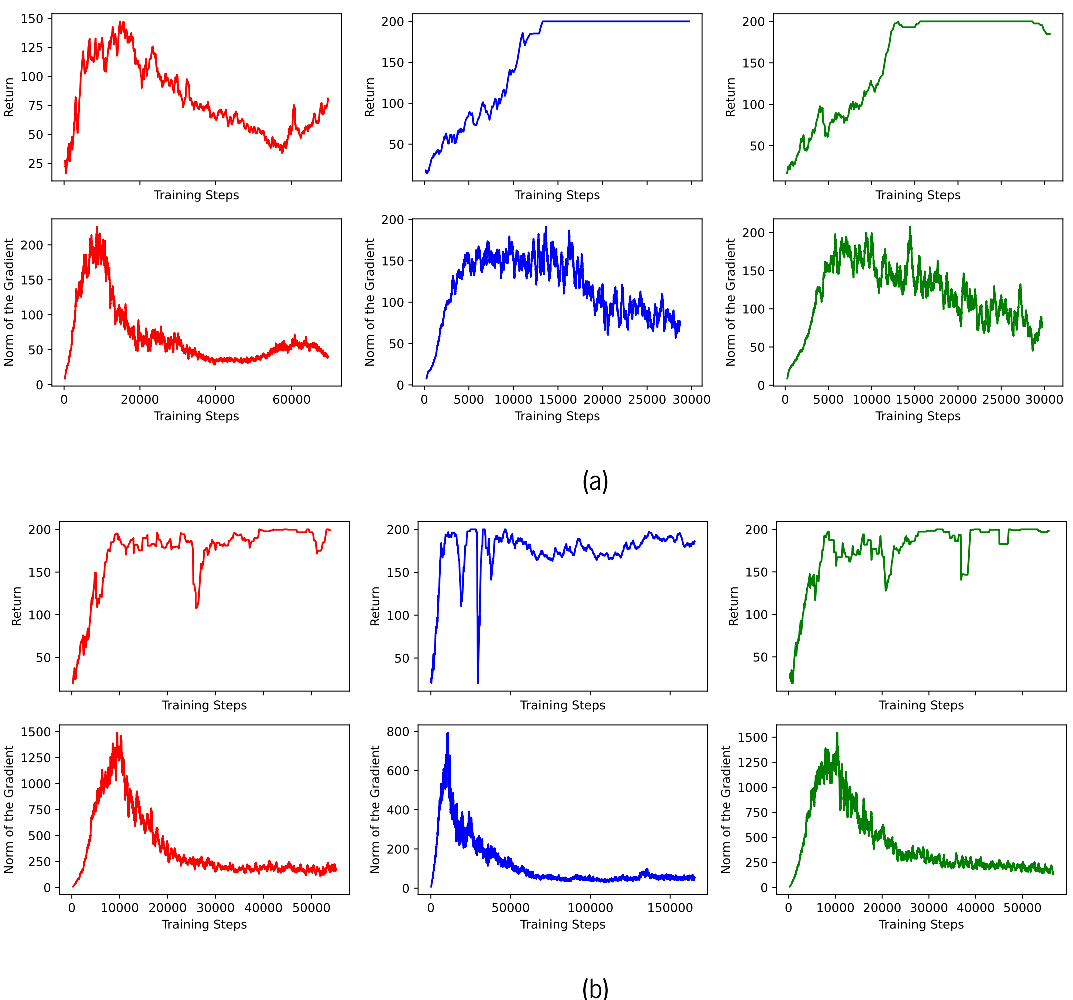
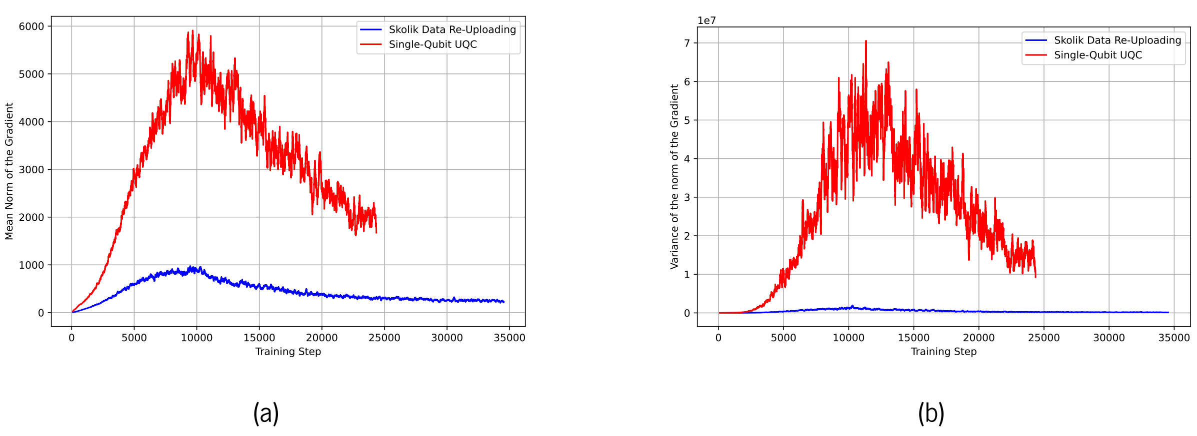


Figure 22: Return and Norm of the gradients for 3 random Baseline (see Subfigure 22a) and Data Re-Uploading (see Subfigure 22b) agents. Both the return an the norm of the agents are presented as moving averages to mitigate noise due to the unstable nature of DQN.

In Subfigure 22b, representing the three data re-uploading agents, there is a common trend: as the return increases, so does the norm of the gradients. Upon reaching a stabilization point between 150 and 200 of return, the gradient norm peaks before gradually descending to a lower value. Two of the baseline agents follow a similar pattern. However, one agent peaks at return 150, concurrent with the gradient norm's maximum, after which both metrics decline. Section 5.5 delves deeper into the potential reasons behind this behavior.

5.4 Trainability Analysis of the UQC

After analyzing the trainability of the models used by Skolik et al. [2022], we applied the same methodology to study the trainability of the single-qubit and multi-qubit UQCs. Figure 23 shows the trainability analysis of the single-qubit model, using the Skolik Data Re-Uploading model for reference (the analysis of the performance of these two models was done in Figure 17 and the color scheme is the same).



(a) (b)

Figure 23: Trainability analysis of the single-qubit UQC and Skolik data re-uploading models in the CartPole-v0 environment from Figure 17 following the methodology defined in Section 4.4.2. Subfigure 23a shows the mean norm of the vector gradient throughout training and Subfigure 23b the variance in the norms of the vector gradients throughout training. 10 agents were initialized from each model.

The gradient norms and their variance were already notably large for the data re-uploading model. Yet, they're even more pronounced for the single-qubit UQC. This reaffirms the idea that Deep Q-Learning's inherent instability can result in models with significant gradient magnitudes, yet these models can still excel in the given environment. Notably, the magnitude of the gradients and its variance seem to be influenced by the choice of model. The reason why the single-qubit UQC shows much larger gradient magnitudes than the Data Re-uploading model is still a topic for further investigation.

Now, let's study the trainability of the multi-qubit UQC models. Figures 24a and 24b show the trainability analysis of the two-qubit UQC and the four-qubit UQC with and without entanglement in the CartPole-v0 environment. Once again, the same set of hyperparameters is used and all quantum circuits have 5 layers.

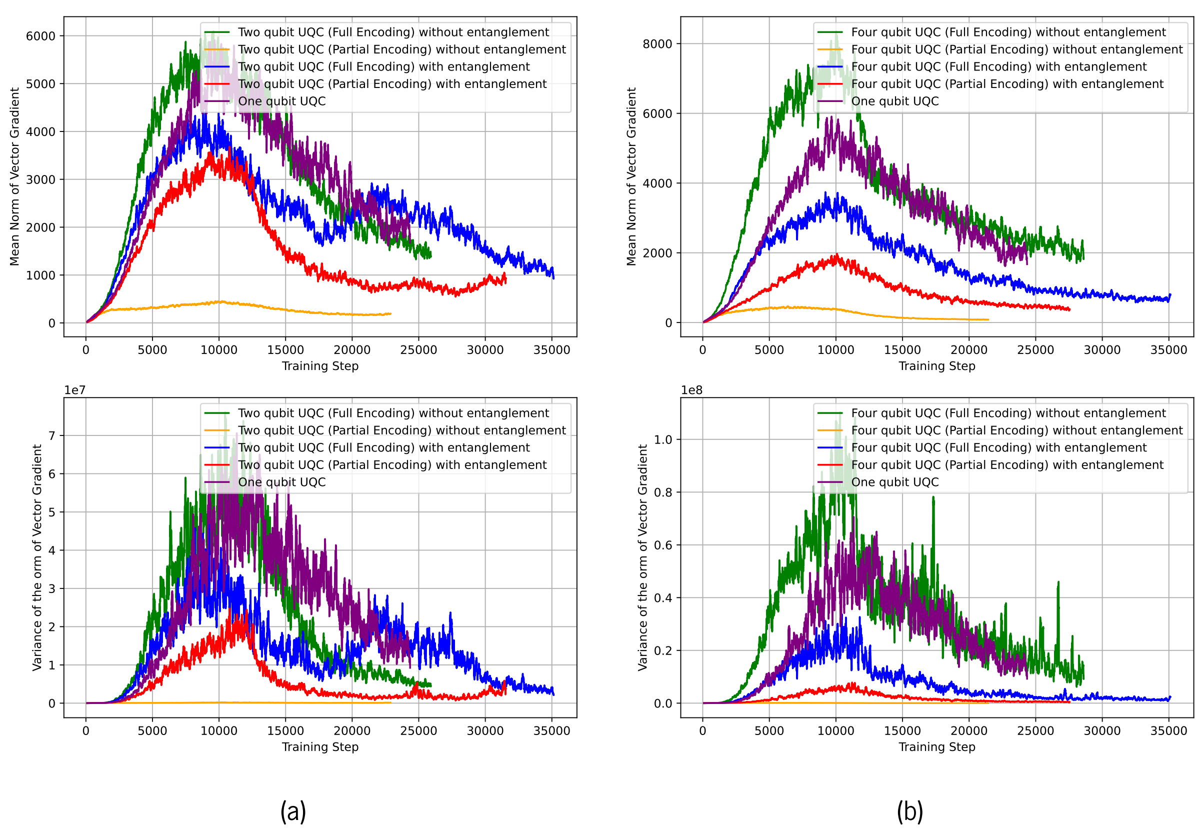


Figure 24: Trainability analysis of the two-qubit (see Subfigure 24a) and four-qubit (see Subfigure 24b) UQCs using the Partial and Full encoding techniques with and without entanglement. 10 agents were initialized from each model. The full set of hyperparameters can be seen in Table 8.

The trainability analysis reveals insights into the behavior of the gradients as we increase the number of qubits in the UQC and also the effect that entanglement has on this behavior. The first observation is that the Partial Encoding models without entanglement have a much lower gradient norm and variance, which is to be expected since these models perform poorly (see Figure 19) and we have already seen that performance and gradient magnitude are deeply correlated.

Moreover, interestingly, the Full Encoding models without entanglement achieve a higher gradient norm and variance than the models with entanglement. In fact, these models have the highest gradient norms and variances out of all the tested models. One possible explanation for this result is that the models with entanglement have a worse performance than the models without entanglement (see Figure

19) which, according to the connection between performance and gradients, might lead to this decrease in the gradients' norm and variance.

Finally, the decrease in the variance does not seem to be exponential as the number of qubits increases. In fact, the four-qubit UQC without entanglement has the highest variance out of all the tested models. These observations reinforce the possibility of using the inherent instability of Deep Q-Learning to mitigate the Barren Plateau Phenomenon. The following section explains some of the main causes behind this instability.

5.5 Tradeoff between Moving Targets and Gradient Magnitude

As we have seen in the previous section, the magnitude of the gradients and their variance throughout training behave differently from expected. Concretely, they continuously increase as the agent learns, reach substantial values when the agent achieves the maximum return, and then start decreasing until the end of training. Moreover, the performance of the agents and their gradients seem to be deeply correlated, which might be useful for VQC-based Deep Q-Learning. If the more expressive models, which typically perform better on the environment since they can better approximate the optimal Q-function, also have higher gradients due to this observed behavior, then the Barren Plateau Phenomenon might be mitigated. However, this is only a possibility for now, since the results pertain only to the CartPole environment, specific models, and the MSE cost function. Consequently, further research is required.

In this section, we attempt to comprehend the reasons behind this behavior. Let's start by looking at the mean loss function across the 10 different agents initialized from the Data Re-Uploading model (the blue lines from Figures 16b, 21a and 21b), see Figure 25.

Immediately after the start of training, the loss increases in a manner that is similar to the norm and variance of the norm of the gradients. It reaches a maximum when the agent achieves the highest return, then decreases throughout the rest of training. This loss curve reveals why the behavior of the gradients is different from the expected. The gradient is a vector containing the partial derivatives of the loss function w.r.t all the parameters. If the loss function increases sharply at the beginning of training, the partial derivatives should also increase, leading to a higher norm of the gradients. But why does the loss function behave in such a way?

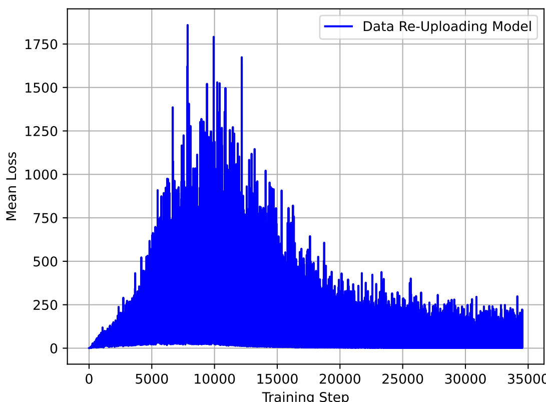


Figure 25: The mean loss across the 10 agents of the data re-uploading model. The loss is only shown until the first agent solves the environment.

To contextualize this anomaly, let's examine how loss functions typically behave in more conventional machine learning tasks. Consider a supervised learning task, such as a classifier. Given a dataset of input-output pairs, the goal is for the model to learn to predict the output for a new input by minimizing some loss function, such as the mean squared error, between the predicted outputs and the actual outputs during training. Since the training dataset is static, meaning it doesn't change throughout training, as the agent learns and gets better at predicting outputs, the loss function decreases. Consequently, the typical behaviour of the loss function in such tasks is to steadily decrease throughout training, while the accuracy of the model (its performance in the context of a classifier) increases.

Contrasting this with our problem, the behavior of the loss function is clearly atypical: it first increases as the agent learns, peaks when the maximum return is achieved, and then decreases throughout training, while still converging to a somewhat considerable value. Thus, now the question is: Why is the loss function behaving in such a way? The discrepancy arises from the fundamental differences between Deep Q-Learning and supervised learning. While there are many reasons that could influence this behaviour, one of the primary reasons is the fact that Deep Q-Learning targets are *non-stationary* - *moving targets*.

Since the targets keep changing during training due to the agent's evolving knowledge, predicting Q-values becomes increasingly challenging. This is specially pronounced in the beginning of training, when the agent is focused on exploring the state-space. The more states explored, the higher the variance in the return and, consequently, the higher the loss. Take, for instance, the CartPole environment. An agent in the beginning of training chooses mostly random actions and, thus, achieves modest returns in the early episodes - let's say, for illustration, in the range $[5, 10]$. However, as the agent begins to learn, it discovers sequences of states and actions that yield returns in the range of, say, $[50, 100]$. This increased range can

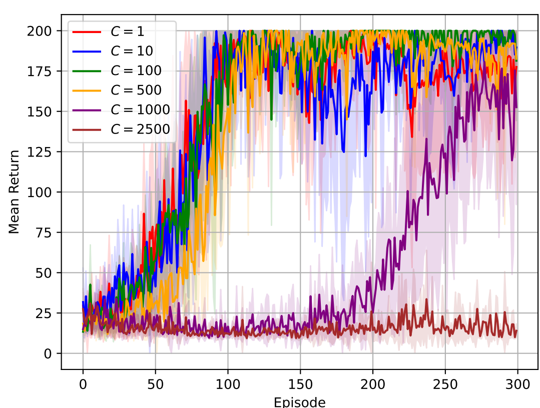
make the task of accurately predicting Q-values more challenging, leading to an increase in the TD-error. When employing the mean squared error loss function, which squares the TD-errors, large errors have a disproportionately large impact on the loss. This explains the sharp increase in the loss function in the early stages of training, when the agent is exploring the state-space. Eventually, as the agent becomes more knowledgeable, the loss is expected to steadily decrease since the agent improves at predicting the Q-values for the most visited states. Nonetheless, it is important to note that, while this is the typical behavior observed so far in the CartPole environment, this decrease in the loss isn't guaranteed, specially in complex environments or if the agent maintains a substantial level of exploration throughout training.

Due to the instability that arose from the moving targets, the original paper Mnih et al. [2015] introduced the concept of a target network. The targets are calculated using this network that has frozen weights, which are updated every C steps to match the weights of the online network. Hence, for C steps at a time, the targets appear stationary. Here, C is an hyperparameter that should be chosen taking into account the tradeoff between speed of convergence and trainability. If C is set too high, the targets move slowly and the model takes longer to train. If C is too low, then the targets change frequently and the algorithm becomes unstable. As previously mentioned, Skolik et al. [2022] used $C = 1$, which means no target network was used. We also used the same hyperparameters, which explains the observed behavior. Figure 26a shows how the data re-uploading model performs on the CartPole environment for different values of C. Figure 27b shows the behavior of the loss function for each of those models.

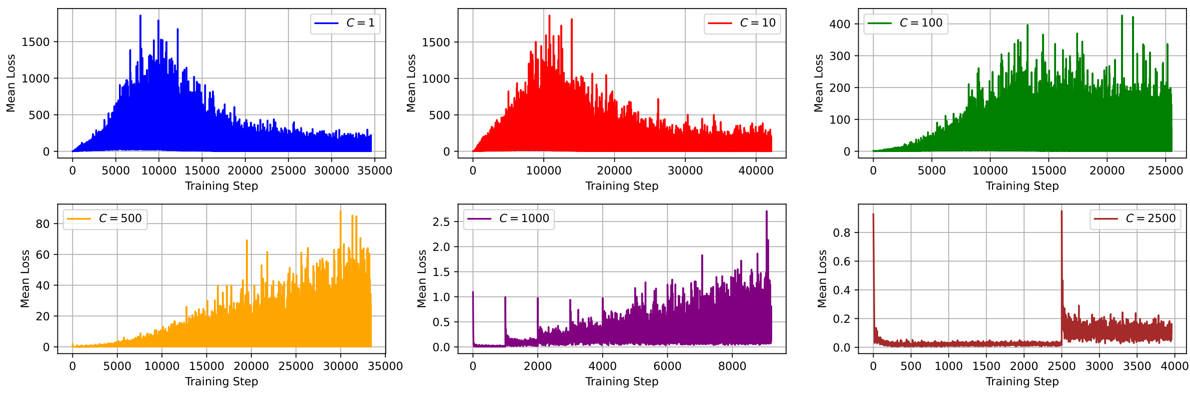
Starting from Figure 26a, it is possible to see how the speed of convergence is affected by the different values of C. From $C \in [1, 100]$, there doesn't appear to be a significant difference in the models' performance. In other words, when the target network is updated every 1 to 100 steps, the models perform similarly. However, when $C = 500$, the model takes noticeably more episodes to learn, although the difference is still not substantial. Then, when $C = 1000$, it is possible to see the full extent of the the negative impact the choice of C has on the speed of convergence. This model takes significantly more episodes to learn than all the aforementioned models, although it still eventually reaches a similar average return. When $C = 2500$, the model isn't even capable of learning and the performance remains mediocre throughout training.

Turning to Figure 27b, it is possible to see the other side of the tradeoff. In other words, it is possible to see how the loss function is affected by the choice of C. The first observation is that, as C increases, the loss function becomes more stable and the maximum values it achieves start decreasing. For $C = 1$ and $C = 10$, it achieves maximum values over 2500. However, when $C = 1000$, it achieves a maximum value of just over 2.5, even though this model is still capable of learning. This shows that increasing C

stabilizes training. Moreover, as C increases, the dynamics of the moving targets become more evident. For example, when $C = 2500$, the loss peaks in the first training step. However, since $C = 2500$, for the next 2499 steps, the loss decreases and converges close to 0. Then, in the training step number 2500, we can see the targets changed, since there was a sudden increase in the loss.



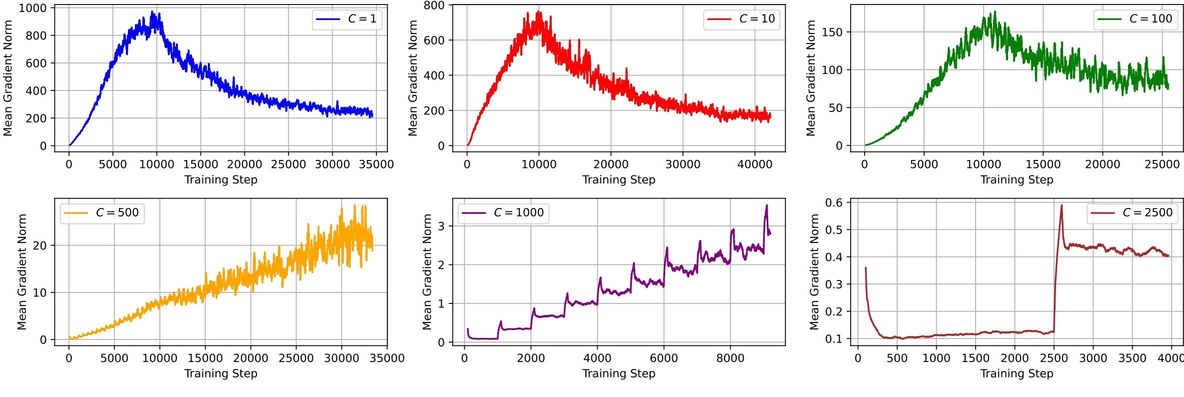
(a) Performance of the data re-uploading model for different values of C.



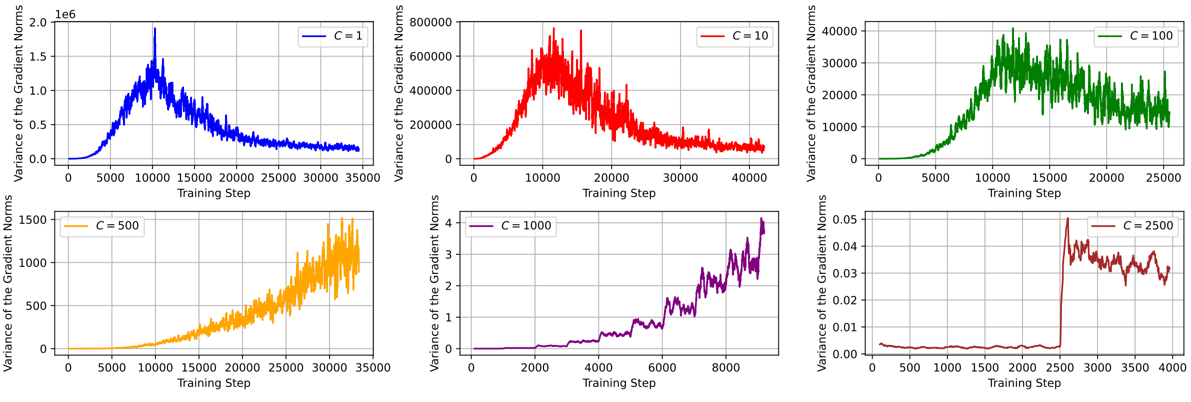
(b) Mean loss of the agents initialized from the data re-uploading model with different values of C.

Figure 26: Analysis of the performance and the loss of the data re-uploading model with different values of C. The other hyperparameters are constant. To analyse the performance and the loss function, 5 agents were initialized from each model. Then, the performance was measured following the methodology from 4.4.1 and the loss function was analyzed by computing the mean loss function over all the agents. The full set of hyperparameters is shown in Table 9.

To see how the choice of C affects the gradients of the models, see Figure 27.



(a) Norm of the Gradients of the data re-uploading model for different values of C.



(b) Variance of the norm of the gradients of the data re-uploading model with different values of C.

Figure 27: Analysis of the gradients of the data re-uploading model with different values of C. The other hyperparameters are constant and 5 agents were initialized from each model. Then, the gradient analysis was performed following the methodology from 4.4.2.

From Figure 27, it is interesting to verify that both the norm of the gradients and the variance of the norm behave similarly to the loss function for all values of C. In particular, we can see how the maximum values of both metrics decay and how the dynamics of the moving targets become more evident as C increases.

This analysis is of utmost importance for VQC-based Deep Q-Learning. In the previous sections, we observed that well-performing models exhibit substantial gradient magnitudes and variances throughout training. This section has empirically demonstrated the influential role of moving targets and the target network update frequency C on the behavior of both the loss function and, consequently, the gradients. In particular, we saw that increasing C stabilizes the loss function, which contributes to more controlled magnitudes of gradients and their variance. However, it is important to highlight that even with relatively low C values — where the gradients' magnitudes and variances were pronounced — the models showcased

impressive performance. They were capable of achieving maximum returns in as few episodes as other more "stable" models with higher values of C.

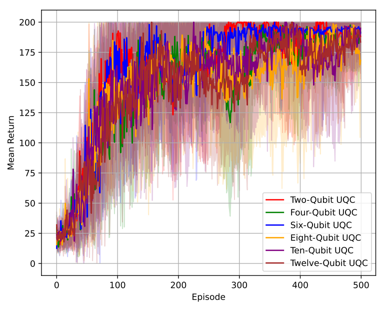
This raises an interesting question. It is known that hardware-efficient VQCs suffer from the Barren Plateau Phenomenon McClean et al. [2018], such that the variance of the gradients decays exponentially with system size. However, it appears that Deep Q-Learning is an inherently unstable algorithm. Intriguingly, certain hyperparameter configurations seem to allow Deep Q-Learning models to effectively learn good policies while sustaining considerable gradient magnitudes and variances. This hints at a potential advantage: might this inherent instability help counteract the Barren Plateau Phenomenon? In other words, could the utilization of highly expressive VQCs (such as data re-uploading ones) in such a context be somewhat resistant to Barren Plateaus? It is crucial to underscore, however, that our empirical observations are specifically tied to the CartPole environment and the MSE loss function. While these initial findings are promising, they underscore the need for broader and more comprehensive research.

There are a few key factors to consider. First, while the moving targets contribute to the substantial gradients observed throughout training, the impact of the loss function must not be undervalued. The observed behavior in these results is so pronounced because the loss function is the MSE, which squares the TD-errors, leading to the massive loss and gradient values. Nonetheless, there are some techniques and some other loss functions that might counteract this effect. For example, gradient-clipping, which consists of clipping the norm of the gradients at a certain pre-defined value, could be used Zhang et al. [2019]. Moreover, it would also be possible to use other loss functions that might be less sensitive to outliers, such as the Huber loss function Huber [1964], which employs a similar technique to gradient clipping. This is to say we are not claiming that there are no solutions for solving this instability and controlling the gradients. Even choosing a more stable set of hyperparameters with a larger target network update frequency could achieve that. We are simply stating that this instability might help counteract the Barren Plateau Phenomenon and thus could be desirable (up to a certain degree) in VQC-based Deep Q-Learning.

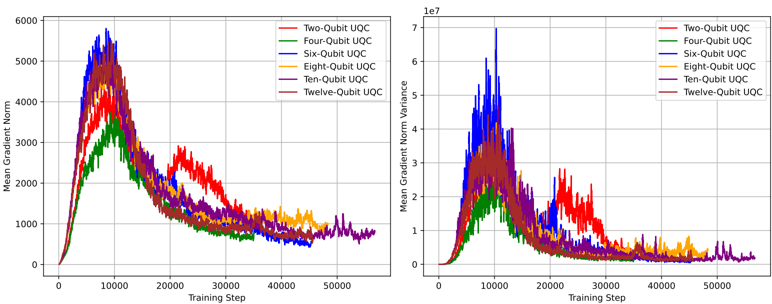
5.6 Gradient Behavior for Increasing System Sizes

One of the advantages of the multi-qubit UQC architecture is that, using the full encoding method, one may encode any input vector into an arbitrary number of qubits. In particular, it is possible to increase the number of qubits even further than the number of features of the input vector. To test how the norm of the gradients and its variance change as the number of qubits increases, we trained the full encoding

multi-qubit UQC model with the set of even numbers of qubits from 2 to 12 in the CartPole-v0 environment. Using a similar technique to Skolik et al. [2022], for a given number of qubits n, each action's observable considers half the qubits, such that the observables are $[Z_0 ... Z_{\frac{n}{2}-1}, Z_{\frac{n}{2}} ... Z_{n-1}]$. For instance, the four-qubit UQC uses $[Z_0Z_1, Z_2Z_3]$, while the six-qubit model uses $[Z_0Z_1Z_2, Z_3Z_4Z_5]$. The performance and trainability analysis are represented in Figures 28a and 28b, respectively.



(a) Performance of the full encoding multi-qubit UQC for a varying number of qubits.



(b) Trainability of the full encoding multi-qubit UQC for a varying number of qubits.

Figure 28: Performance and Trainability analysis of the multi-qubit UQCs as the number of qubits increase. 10 agents were initialized from each model. The full set of hyperparameters can be seen in Table 8.

First, from the performance analysis shown in Figure 28a, it seems that the performance slightly decreases as the number of qubits increases in the range $[2, 12]$. One can see that by the lower average return at the beginning of training and by the higher standard deviation exhibited by models that use a higher number of qubits.

Nonetheless, the most relevant observation comes from Figure 28b. Both the magnitude of the gradient and its variance behave similarly during training for all models, while also achieving a similar range of values. This is particularly interesting because it confirms the suspicions that arose throughout this work. The Barren Plateau Phenomenon states that the gradients decay exponentially with system size. However, for the VQC-based Deep Q-Learning algorithm, using the multi-qubit UQC model in the CartPole-v0 environment with the MSE loss function, that doesn't seem to be the case. This hints at the possibility of hardware-efficient VQCs being especially suitable to be used as function approximators in Deep Q-learning since they might be somewhat resistant to the Barren Plateau Phenomenon due to the moving targets' effects on the gradients of this algorithm.

However, it is important to note that these results are preliminary and insufficient to draw general conclusions. Several factors require a more in-depth analysis. For instance, we only considered cost

functions that use half the qubits. However, the locality of the observables used influences the norm and the variance of the gradients Cerezo et al. [2021b], such that different cost functions that consider other observables should be tested. Moreover, how these gradients change as the depth of the circuit increases should also be researched. Unfortunately, due to a lack of time and computational resources, we could not afford to do such a study. Thus, these results should serve as the basis for more nuanced studies.

5.7 The Acrobot Environment

In previous sections, we focused our discussion on the CartPole-v0 environment. However, as highlighted in section 4.3, the CartPole is relatively simple. The Acrobot-v1 environment, on the other hand, presents a more complex scenario. In fact, to the best of our knowledge, no VQC-based Deep Q-Learning model has yet been tested in this environment. Moreover, we have noted substantial gradient magnitudes and variances in the CartPole environment. We are keen to investigate whether similar behavior can be observed in the Acrobot environment.

Consequently, in this section, we will delve into the performance and trainability of VQC-based Deep Q-Learning models in the Acrobot environment. We plan to test two models that have achieved some of the best performances so far: the Skolik Data Re-Uploading model with four qubits and the Full Encoding Multi-Qubit UQC with three qubits and entanglement. In fact, the multi-qubit UQC models without entanglement performed better than the models with entanglement, but due to the computational demands of the Acrobot environment, we decided to test just the model with entanglement.

Nevertheless, it is necessary to make a few adjustments before proceeding with these tests:

- **Hyperparameter Search** - In our previous experiments, we utilized a consistent set of hyperparameters across all models to facilitate fair comparisons and isolate the influence of hyperparameter variations. However, given the shift to a different and more complex environment, it became necessary to conduct a hyperparameter search to optimize the learning potential of the models. We used optuna Akiba et al. [2019] to perform a smart search method that usually reduces the number of combinations tested before finding a near-optimal set of hyperparameters when compared with a grid-search method. Details of the final hyperparameter set can be found in Table 10. One particularly important hyperparameter is the target network update frequency, which we set to $C = 250$.

- **Classical Pre-Processing** - The state-space of the Acrobot environment encompasses six features: $[\cos(\theta_1), \sin(\theta_1), \cos(\theta_2), \sin(\theta_2), \omega(\theta_1), \omega(\theta_2)]$ where ω is the angular velocity, as detailed in section 4.3.2. We reduced the dimensionality of the state-space to include only $[\theta_1, \theta_2,$

$\omega(\theta_1), \omega(\theta_2)]$, and normalized all features to fall within the range of $[-2\pi, 2\pi]$.

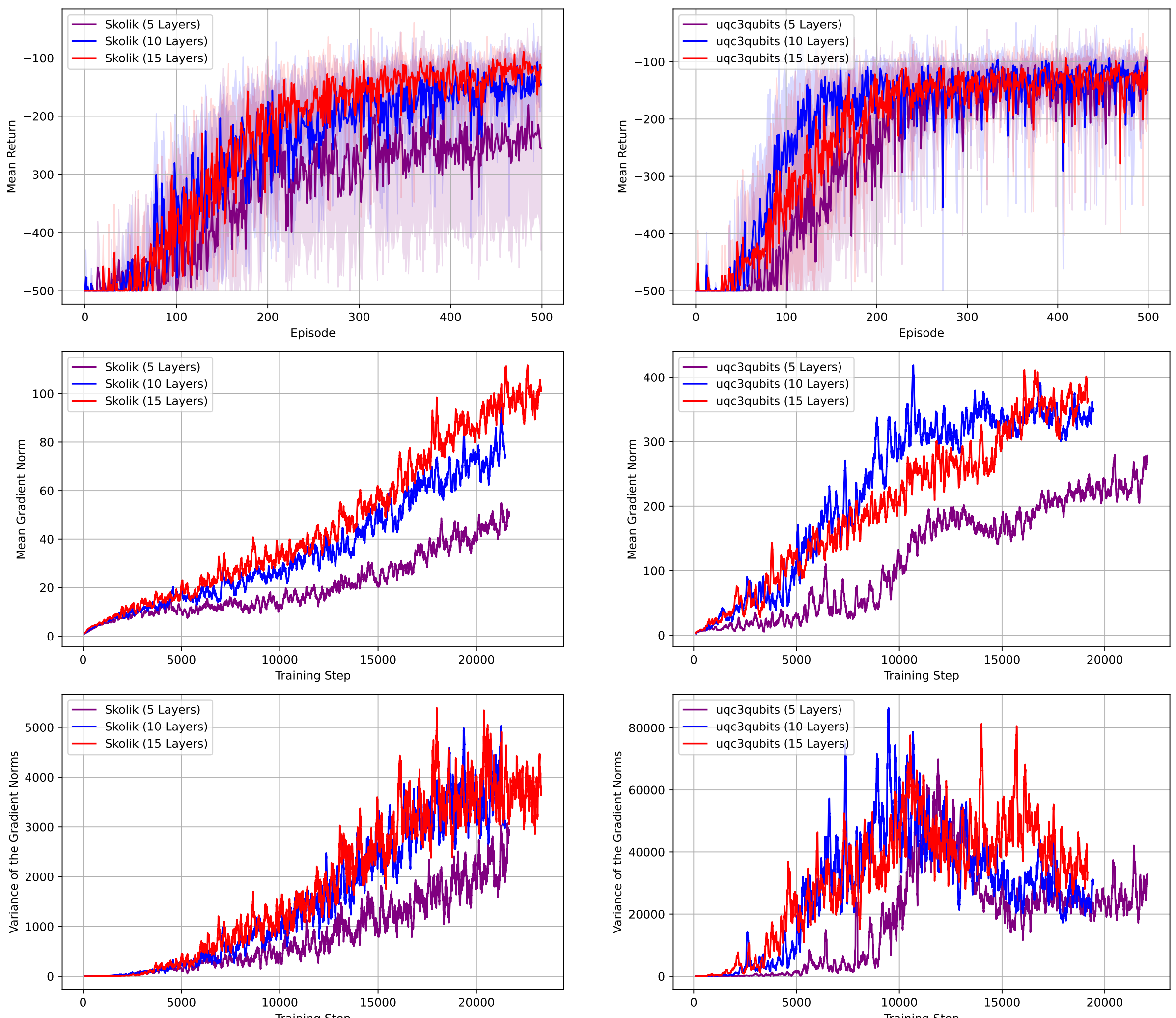
- **Observables** - In the Acrobot environment, the set of actions is given by $\mathcal{A} = \{-1, 0, 1\}$, with each of the actions corresponding to applying the respective torque to the actuated joint. This requires the measurement of three different observables to accurately represent these actions. The observables used for the two tested models are seen in Table 4.

	Expectation Values	
Action	**Skolik Data Re-Uploading**	**3-Qubit UQC**
-1	$\langle Z_0 \rangle$	$\langle Z_0 \rangle$
0	$\langle Z_1 Z_2 \rangle$	$\langle Z_1 \rangle$
1	$\langle Z_3 \rangle$	$\langle Z_2 \rangle$

Table 4: Chosen observables for the two models tested in the Acrobot-v1 environment.

There is also a significant difference that must be mentioned. In the CartPole environment, if an agent gets a return greater or equal to 195 over the past 100 episodes, the environment is considered solved and training is halted. However, the Acrobot environment has no such condition, so the goal is to obtain the maximum return possible in a finite number of episodes. The results of the performance and trainability analysis of these models with a varying number of layers can be seen in Figure 29

Let's start by analysing the performance of the models. It is important to note that fairly good performing NN-based Deep Q-Learning agents on the Acrobot environment typically achieve an average return close to or less than -100 OpenAI [2023]. From Figure 29a, one can observe that the Skolik Data Re-Uploading model is able to perform considerably well in this environment. Nonetheless, there is a clear improvement in the performance when the number of layers is increased from five to ten. From Figure 29b, one can see that the three-qubit UQC is also capable of achieving a high average return. However, increasing the number of layers of this model doesn't seem to considerably impact performance. Moreover, the UQC appears to achieve the mark of -100 average return sooner than the Skolik Data Re-uploading models. Once again, we emphasize that, to the best of our knowledge, this is the first implementation of VQC-based Deep Q-Learning models in the Acrobot environment.



(a) Performance and trainability analysis of the Skolik Data Re-Uploading model on the Acrobot environment.

(b) Performance and trainability analysis of the three-Qubit UQC model on the Acrobot environment.

Figure 29: Performance and Trainability analysis of the Skolik Data Re-Uploading and three-qubit UQC models on the Acrobot environment. 10 agents were initialized from each model. The full set of hyperparameters can be seen in Table 10.

Next, we proceed to analyze the trainability of the models, focusing particularly on the gradient behavior observed during the training process. We verify a similar gradient behavior to the observed in the Cartpole environment, where both the norm of the gradients and its variance escalate to substantial values. However, unlike what was verified in the CartPole environment, the gradients don't peak and then decrease sustainably throughout training. Instead, they continuously increase (with the exception of the variance of the gradient norm of the three-qubit UQC). These results suggest that the observed behavior might not be intrinsically linked to the characteristics of the environment, but possibly influenced by

another variable or parameter.

Based on the analysis conducted thus far, it is reasonable to assume that the two most important factors for this gradient behavior are the target network update frequency and the cost function, as seen in section 5.5. The former due to the tradeoff between moving targets and gradient magnitude. The latter because the MSE cost function squares the (already large due to the moving targets) TD-errors, exacerbating the gradients.

Although replicating all experiments conducted in the CartPole environment would be interesting, the associated computational expenses are prohibitively high. Nonetheless, we were still able to create VQC-based models capable of achieving a considerable performance in the previously untapped Acrobot-V1 environment and seeing a substantial gradients' norm and variance, which hints at the possibility that the results obtained in the Acrobot environment would be similar to those obtained in the CartPole environment. Thus, further research is needed to confirm or deny these claims.

5.8 Summary

In this chapter, we examined the performance and trainability of various VQC-based models within the CartPole-v0 and Acrobot-v1 environments.

Initially, we replicated the study conducted by Skolik et al. [2022] in the CartPole environment, reaffirming the significance of data re-uploading as well as trainable input and output scaling in enhancing the models' performance. After that, we used the single-qubit UQC as a function approximator in VQC-based Deep Q-Learning which, to the best of our knowledge, had never been done. Moreover, the multi-qubit UQC was also introduced and tested in the CartPole environment.

Subsequently, we analyzed the trainability of these models and found that the gradient norms and variances reached significant values, especially in highly expressive models that use data re-uploading. This result is in stark contrast with what is expected since models with an increased circuit depth and expressivity should lead to smaller gradients Holmes et al. [2022].

Afterward, we empirically verified that the gradients' behavior is affected by a characteristic inherent to Deep Q-Learning - moving targets. This instability is further exacerbated by the mean squared error loss function. These results suggest that this inherent instability to Deep Q-Learning might counterbalance the effects of the Barren Plateau Phenomenon. If confirmed, this would be a promising result for VQC-Based Deep Q-Learning, since highly expressive VQCs could be used as function approximators without the negative effects of the vanishing gradients. However, it is important to note that these findings are confined

to the models explored within the CartPole environment and the mean squared error loss function.

Afterward, we verified that increasing the number of qubits does not seem to considerably affect the gradients' magnitudes nor their variance in the CartPole environment.

Finally, we used VQC-based Deep Q-Learning models to achieve a considerable performance in the slightly more complex Acrobot-v1 environment which, to the best of our knowledge, has not been previously done. It was also verified that the gradients' magnitudes and their variance remain significant even in this more complex environment.

Chapter 6

Conclusions and future work

6.1 Conclusions

This work researched the intersection between Quantum Computing and Deep Reinforcement Learning. More concretely, the main goal was to study the effects of data re-uploading on the performance and trainability of VQC-based Deep Q-Learning models. It was already empirically shown by Skolik et al. [2022] that using data re-uploading increases the performance of these models in the CartPole environment, possibly due to its increased expressivity, which allows the approximation of more intricate functions Schuld et al. [2021]. Throughout this work, we implemented and tested different data re-uploading models on the CartPole environment and the results match those by Skolik et al. [2022] and Schuld et al. [2021]. Moreover, we adapted the Universal Quantum Classifier (UQC) Pérez-Salinas et al. [2020], which also uses data re-uploading, to be used as a function approximator in this setting and achieved the best performance of all VQC models tested thus far on the CartPole environment. Furthermore, we tested different data re-uploading models on the more complex Acrobot-v1 environment and achieved a considerable level of performance. To the best of our knowledge, this was the first implementation of a VQC-based Deep Q-Learning model in the Acrobot environment. From these results, we conclude that data re-uploading indeed increases the performance of the Deep Q-Learning models.

However, a concern would be that the increase in expressivity and circuit depth that arises from the use of data re-uploading could decrease the trainability of these models due to the Barren Plateau Phenomenon McClean et al. [2018]. To investigate such a concern, we analyzed the norm of the gradients and the variance of this norm. From this analysis, we verified that the gradients achieve substantial values and actually increase when data re-uploading is used versus when it is not used. Furthermore, we empirically showed that this increase is due to an instability that is inherent to Deep Q-Learning, which is the fact that the targets are non-stationary. Alongside the Mean Squared Error loss function, this leads to a very unstable algorithm with substantial gradients. Moreover, we also verified that increasing the number of

qubits of the multi-qubit UQC in the CartPole environment does not lead to a decrease in the gradients' magnitude or variance.

Nonetheless, it is important to note that these results pertain only to the CartPole and Acrobot environments, specific data re-uploading models and sets of hyperparameters, and the mean squared error loss function. Thus, while promising, it is still too preliminary to generalize.

6.2 Prospect for future work

For future work, it would be very interesting to develop a different methodology to analyze the trainability of VQC-based models. The magnitude and variance of the gradients already give some insights into the trainability of a model, but a more complex and intricate methodology that could add more information to this analysis would be interesting. For instance, it may be possible to study the eigenvalues of the Hessian Matrix.

Another interesting possibility would be to analyze these models from a Fourier Analysis perspective. Schuld et al. [2021] show that VQCs may be seen as Partial Fourier Series in the data and that using data re-uploading increases the frequencies the VQC "has access to". Thus, it may be possible to analyze the optimal Q-functions of certain environments and the Fourier Series that the VQCs are approximating. This might reveal some insights into what is happening behind the scenes.

Finally, while the results of this work hint at the possibility of the inherent instability of Deep Q-Learning counteracting the effects of the Barren Plateau Phenomenon, they are still insufficient for making such a claim. However, if this were true, then VQCs would be especially adequate for being used as function approximators in such a setting. Thus, it would be interesting to build upon this work and analyze the trainability of VQC-based Deep Q-Learning models on different, more complex environments, with different loss functions and sets of hyperparameters. It might even be possible to theoretically derive bounds for the gradients of these models, which could reinforce the empirical analysis.

Bibliography

Alekh Agarwal, Nan Jiang, Sham M Kakade, and Wen Sun. Reinforcement learning: Theory and algorithms. *CS Dept., UW Seattle, Seattle, WA, USA, Tech. Rep*, pages 10–4, 2019.

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

David Andre and Stuart J Russell. State abstraction for programmable reinforcement learning agents. In *Aaai/iaai*, pages 119–125, 2002.

Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

Arash Bahrammirzaee. A comparative survey of artificial intelligence applications in finance: artificial neural networks, expert system and hybrid intelligent systems. *Neural Computing and Applications*, 19 (8):1165–1195, 2010.

Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M Sohaib Alam, Guillermo Alonso-Linaje, B AkashNarayanan, Ali Asadi, et al. Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*, 2018.

Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, 2017.

M Bilkis, María Cerezo, Guillaume Verdon, Patrick J Coles, and Lukasz Cincio. A semi-agnostic ansatz with variable structure for quantum machine learning. *arXiv preprint arXiv:2103.06712*, 2021.

Michael Broughton, Guillaume Verdon, Trevor McCourt, Antonio J Martinez, Jae Hyeon Yoo, Sergei V Isakov, Philip Massey, Ramin Halavati, Murphy Yuezhen Niu, Alexander Zlokapa, et al. Tensorflow quantum: A software framework for quantum machine learning. *arXiv preprint arXiv:2003.02989*, 2020.

Lucian Busoniu, Robert Babuska, Bart De Schutter, and Damien Ernst. *Reinforcement learning and dynamic programming using function approximators*. CRC press, 2017.

Yudong Cao, Jonathan Romero, Jonathan P Olson, Matthias Degroote, Peter D Johnson, Mária Kieferová, Ian D Kivlichan, Tim Menke, Borja Peropadre, Nicolas PD Sawaya, et al. Quantum chemistry in the age of quantum computing. *Chemical reviews*, 119(19):10856–10915, 2019.

Giuseppe Carleo and Matthias Troyer. Solving the quantum many-body problem with artificial neural networks. *Science*, 355(6325):602–606, 2017.

Salvatore Carta, Anselmo Ferreira, Alessandro Sebastian Podda, Diego Reforgiato Recupero, and Antonio Sanna. Multi-dqn: An ensemble of deep q-learning agents for stock market forecasting. *Expert systems with applications*, 164:113820, 2021.

Marco Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, et al. Variational quantum algorithms. *Nature Reviews Physics*, 3(9):625–644, 2021a.

Marco Cerezo, Akira Sone, Tyler Volkoff, Lukasz Cincio, and Patrick J Coles. Cost function dependent barren plateaus in shallow parametrized quantum circuits. *Nature communications*, 12(1):1791, 2021b.

Samuel Yen-Chi Chen, Chao-Han Huck Yang, Jun Qi, Pin-Yu Chen, Xiaoli Ma, and Hsi-Sheng Goan. Variational quantum circuits for deep reinforcement learning. *IEEE Access*, 8:141007–141024, 2020.

Rodrigo Coelho. Vqc_qlearning: A github repository. https://github.com/RodrigoCoelho7/VQC_Qlearning, 2023. GitHub repository.

Iris Cong, Soonwon Choi, and Mikhail D Lukin. Quantum convolutional neural networks. *Nature Physics*, 15(12):1273–1278, 2019.

Brian Coyle, Daniel Mills, Vincent Danos, and Elham Kashefi. The born supremacy: quantum advantage and training of an ising born machine. *npj Quantum Information*, 6(1):60, 2020.

Jonas Degrave, Federico Felici, Jonas Buchli, Michael Neunert, Brendan Tracey, Francesco Carpanese, Timo Ewalds, Roland Hafner, Abbas Abdolmaleki, Diego de Las Casas, et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602(7897):414–419, 2022.

David Deutsch. Quantum theory, the church–turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 400(1818):97–117, 1985.

Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028*, 2014.

Richard P Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21 (6/7), 1982.

Edward Grant, Leonard Wossnig, Mateusz Ostaszewski, and Marcello Benedetti. An initialization strategy for addressing barren plateaus in parametrized quantum circuits. *Quantum*, 3:214, 2019.

Evan Greensmith, Peter L Bartlett, and Jonathan Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5(9), 2004.

Siyu Guo, Xiuguo Zhang, Yiquan Du, Yisong Zheng, and Zhiying Cao. Path planning of coastal ships based on optimized dqn reward function. *Journal of Marine Science and Engineering*, 9(2):210, 2021.

Pavel Hamet and Johanne Tremblay. Artificial intelligence in medicine. *Metabolism*, 69:S36–S40, 2017.

Serge Haroche and Jean-Michel Raimond. Quantum computing: dream or nightmare? *Physics Today*, 49 (8):51–52, 1996.

Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

Wayne Holmes, Maya Bialik, and Charles Fadel. Artificial intelligence in education. Globethics Publications, 2023.

Zoë Holmes, Kunal Sharma, Marco Cerezo, and Patrick J Coles. Connecting ansatz expressibility to gradient magnitudes and barren plateaus. *PRX Quantum*, 3(1):010313, 2022.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

Yuh-Jong Hu and Shang-Jen Lin. Deep reinforcement learning for optimizing finance portfolio management. In *2019 amity international conference on artificial intelligence (AICAI)*, pages 14–20. IEEE, 2019.

Peter J Huber. Robust estimation of a location parameter. *The Annals of Mathematical Statistics*, pages 73–101, 1964.

Sofiene Jerbi, Casper Gyurik, Simon Marshall, Hans J Briegel, and Vedran Dunjko. Variational quantum policies for reinforcement learning. *arXiv preprint arXiv:2103.05577*, 2021.

Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

Owen Lockwood and Mei Si. Reinforcement learning with quantum variational circuit. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, pages 245–251, 2020.

Nguyen Cong Luong, Dinh Thai Hoang, Shimin Gong, Dusit Niyato, Ping Wang, Ying-Chang Liang, and Dong In Kim. Applications of deep reinforcement learning in communications and networking: A survey. *IEEE Communications Surveys & Tutorials*, 21(4):3133–3174, 2019.

Alexander Ly, Maarten Marsman, Josine Verhagen, Raoul PPP Grasman, and Eric-Jan Wagenmakers. A tutorial on fisher information. *Journal of Mathematical Psychology*, 80:40–55, 2017.

Jarrod R McClean, Sergio Boixo, Vadim N Smelyanskiy, Ryan Babbush, and Hartmut Neven. Barren plateaus in quantum neural network training landscapes. *Nature communications*, 9(1):1–6, 2018.

Francisco S Melo and M Isabel Ribeiro. Q-learning with linear function approximation. In *International Conference on Computational Learning Theory*, pages 308–322. Springer, 2007.

Nico Meyer, Christian Ufrecht, Maniraman Periyasamy, Daniel D Scherer, Axel Plinge, and Christopher Mutschler. A survey on quantum reinforcement learning. *arXiv preprint arXiv:2211.03464*, 2022.

Kosuke Mitarai, Makoto Negoro, Masahiro Kitagawa, and Keisuke Fujii. Quantum circuit learning. *Physical Review A*, 98(3):032309, 2018.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

Miguel Morales. *Grokking deep reinforcement learning*. Manning Publications, 2020.

Xiaomin Mou. Artificial intelligence: Investment trends and selected industry uses. *International Finance Corporation*, 8, 2019.

Hai Nguyen and Hung La. Review of deep reinforcement learning for robot manipulation. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 590–595. IEEE, 2019.

Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

OpenAI. Gym leaderboard. GitHub, 2023. URL https://github.com/openai/gym/wiki/Leaderboard.

Mateusz Ostaszewski, Edward Grant, and Marcello Benedetti. Structure optimization for parameterized quantum circuits. *Quantum*, 5:391, 2021.

Adrián Pérez-Salinas, Alba Cervera-Lierta, Elies Gil-Fuster, and José I Latorre. Data re-uploading for a universal quantum classifier. *Quantum*, 4:226, 2020.

John Preskill. Reliable quantum computers. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 454(1969):385–410, 1998.

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

John Preskill. Quantum computing 40 years later. In *Feynman Lectures on Computation*, pages 193–244. CRC Press, 2023.

Qiskit contributors. Qiskit: An open-source framework for quantum computing, 2023.

Rajarshi Roy, Jonathan Raiman, Neel Kant, Ilyas Elkin, Robert Kirby, Michael Siu, Stuart Oberman, Saad Godil, and Bryan Catanzaro. Prefixrl: Optimization of parallel prefix circuits using deep reinforcement learning. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 853–858. IEEE, 2021.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

Hadi Salehi and Rigoberto Burgueño. Emerging artificial intelligence methods in structural engineering. *Engineering structures*, 171:170–189, 2018.

Maria Schuld and Nathan Killoran. Quantum machine learning in feature hilbert spaces. *Physical review letters*, 122(4):040504, 2019.

Maria Schuld and Francesco Petruccione. *Machine learning with quantum computers*. Springer, 2021.

Maria Schuld, Ville Bergholm, Christian Gogolin, Josh Izaac, and Nathan Killoran. Evaluating analytic gradients on quantum hardware. *Physical Review A*, 99(3):032331, 2019.

Maria Schuld, Alex Bocharov, Krysta M Svore, and Nathan Wiebe. Circuit-centric quantum classifiers. *Physical Review A*, 101(3):032308, 2020.

Maria Schuld, Ryan Sweke, and Johannes Jakob Meyer. Effect of data encoding on the expressive power of variational quantum-machine-learning models. *Physical Review A*, 103(3):032430, 2021.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

André Sequeira, Luis Paulo Santos, and Luís Soares Barbosa. Variational quantum policy gradients with an application to quantum control. *arXiv preprint arXiv:2203.10591*, 2022.

Peter W Shor. Fault-tolerant quantum computation. In *Proceedings of 37th conference on foundations of computer science*, pages 56–65. IEEE, 1996.

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

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

David Silver, Satinder Singh, Doina Precup, and Richard S Sutton. Reward is enough. *Artificial Intelligence*, 299:103535, 2021.

Daniel R Simon. On the power of quantum computation. *SIAM journal on computing*, 26(5):1474–1483, 1997.

Andrea Skolik, Sofiene Jerbi, and Vedran Dunjko. Quantum agents in the gym: a variational quantum algorithm for deep q-learning. *Quantum*, 6:720, 2022.

Carlos Oscar Sánchez Sorzano, Javier Vargas, and A Pascual Montano. A survey of dimensionality reduction techniques. *arXiv preprint arXiv:1403.2877*, 2014.

Andrew M Steane. Error correcting codes in quantum theory. *Physical Review Letters*, 77(5):793, 1996.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

JN Tsitsiklis and B Van Roy. An analysis of temporal-difference learning with function approximationtechnical. *Rep. LIDS-P-2322). Lab. Inf. Decis. Syst. Massachusetts Inst. Technol. Tech. Rep*, 1996.

AM Turing. Computing machinery and intelligence. 1950.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

Marco Wiering and Martijn Van Otterlo. *Reinforcement Learning: State of the Art*. Springer, 2012. ISBN 978-3-642-27644-6. doi: 10.1007/978-3-642-27645-3.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

Chao Yu, Jiming Liu, Shamim Nemati, and Guosheng Yin. Reinforcement learning in healthcare: A survey. *ACM Computing Surveys (CSUR)*, 55(1):1–36, 2021.

Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. Why gradient clipping accelerates training: A theoretical justification for adaptivity. *arXiv preprint arXiv:1905.11881*, 2019.

Christa Zoufal, Aurélien Lucchi, and Stefan Woerner. Variational quantum boltzmann machines. *Quantum Machine Intelligence*, 3:1–15, 2021.

Part III
Appendices

Appendix A
Models' Hyperparameters

A.1 Hyperparameters' explanation

Some of the hyperparameters for the VQC-based Deep Q-Learning models are explained in the following table:

Hyperparameter	**Explanation**
qubits	The quantum circuit's number of qubits
layers	The quantum circuit's number of layers
γ	The return's discount factor
trainable input scaling	whether trainable input scaling is used or not
trainable output scaling	whether trainable output scaling is used or not
learning rate of parameters θ	the learning rate of parameters θ
learning rate of input scaling parameters	the learning rate of input scaling parameters
Learning rate of output scaling parameters	the learning rate of output scaling parameters
batch size	the batch size
decaying schedule of ϵ-greedy policy	defines the decaying schedule of the ϵ-greedy policy (e.g exponential)
ϵ_{init}	the initial value of ϵ
ϵ_{dec}	the decay rate of $epsilon$ per episode
ϵ_{min}	the minimum value of ϵ
update model	the update model's frequency
update target model	the target model's update frequency
data re-uploading	whether data re-uploading is used or not

Table 5: An explanation of VQC-Based Deep Q-Learning's hyperparameters

A.2 Hyperparameters used throughout this work

The set of hyperparameters used for the VQCs in Figures 16, 21 and 22, can be seen in table 6.

	CartPole-v0
qubits	4
layers	5
γ	0.99
trainable input scaling	yes, no
trainable output scaling	yes, no
learning rate of parameters θ	0.001
learning rate of input scaling parameters	0.001
learning rate of output scaling parameters	0.1
batch size	16
decaying schedule of ϵ-greedy policy	Exponential
ϵ_{init}	1
ϵ_{dec}	0.99
ϵ_{min}	0.01
update model	1
update target model	1
size of replay buffer	10000
data re-uploading	yes, no

Table 6: Models' hyperpameters from Figure 16

The set of hyperparameters used for the single-qubit UQC in Figure 17 can be seen in table 7.

	CartPole-v0
qubits	1
layers	5
γ	0.99
trainable input scaling	yes
trainable output scaling	yes
learning rate of parameters θ	0.001
learning rate of input scaling parameters	0.001
learning rate of output scaling parameters	0.1
batch size	16
decaying schedule of ϵ-greedy policy	Exponential
ϵ_{init}	1
ϵ_{dec}	0.99
ϵ_{min}	0.01
update model	1
update target model	1
size of replay buffer	10000
data re-uploading	yes

Table 7: Models' hyperpameters from Figure 17

The set of hyperparameters used for the multi-qubit UQCs in Figures 19 and 28 can be seen in table 7.

	CartPole-v0
qubits	$2, 4, 6, 8, 10, 12$
layers	5
γ	0.99
trainable input scaling	yes
trainable output scaling	yes
learning rate of parameters θ	0.001
learning rate of input scaling parameters	0.001
learning rate of output scaling parameters	0.1
batch size	16
decaying schedule of ϵ-greedy policy	Exponential
ϵ_{init}	1
ϵ_{dec}	0.99
ϵ_{min}	0.01
update model	1
update target model	1
size of replay buffer	10000
data re-uploading	yes

Table 8: Models' hyperparameters from Figures 19 and 28.

The set of hyperparameters used for the models of Figure 26 can be seen in table 9.

	CartPole-v0
qubits	4
layers	5
γ	0.99
trainable input scaling	yes
trainable output scaling	yes
learning rate of parameters θ	0.001
learning rate of input scaling parameters	0.001
learning rate of output scaling parameters	0.1
batch size	16
decaying schedule of ϵ-greedy policy	Exponential
ϵ_{init}	1
ϵ_{dec}	0.99
ϵ_{min}	0.01
update model	1
update target model	1,10,100,500,1000,2500
size of replay buffer	10000
data re-uploading	yes

Table 9: Models' hyperparameters from Figure 26

The set of hyperparameters used for the multi-qubit UQC and Skolik Data Re-uploading models in Figure 29 can be seen in table 10.

	Skolik Data-Reup Acrobot-v1	**Multiqubit UQC Acrobot-V1**
qubits	4	3
layers	$5, 10, 15$	$5, 10, 15$
γ	0.99	0.99
trainable input scaling	yes	yes
trainable output scaling	yes	yes
learning rate of parameters θ	0.001	0.001
learning rate of input scaling parameters	0.001	0.001
learning rate of output scaling parameters	0.1	0.1
batch size	32	32
decaying schedule of ϵ-greedy policy	Exponential	Exponential
ϵ_{init}	1	1
ϵ_{dec}	0.99	0.99
ϵ_{min}	0.01	0.01
update model	5	5
update target model	250	250
size of replay buffer	50000	50000
data re-uploading	yes	yes

Table 10: Models' hyperpameters from Figure 29.