Universiteit Leiden
The Netherlands

Opleiding Informatica

Q-learning with Parameterized Quantum Circuits:
design, implementation and benchmarking

Koen Bouwman

Supervisors:
Vedran Dunjko & Casper Gyurik

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

05/08/2020

Abstract

Existing quantum computers are still very limited in scale, making it vital to utilize them to their fullest extent. One way of doing so is by using hybrid quantum-classical algorithms. In this context, so-called parameterized quantum circuits can be regarded as machine-learning models with the potential of remarkable expressive power. The use of such parameterized quantum circuit models has been studied extensively in both a supervised and unsupervised context. But, as it is still an emerging field, there has been little research on the use of parameterized quantum circuits in a reinforcement learning context. In this thesis, we will investigate the use of parameterized quantum circuits as Q-value approximators and attempt to train them on the FrozenLake environment. To be more precise, motivated by unstable performance of initial experiments using standard Q-learning methods, we thoroughly investigate the expressivity and optimal configurations of newly introduced quantum model hyperparameters (i.e., parameters that only appear due to the quantum model). We do so by performing supervised regression to investigate whether the parameterized quantum circuits can encode optimal Q-values and what configuration of quantum model hyperparameters works best. Our experiments found that our quantum model is able to very closely represent the optimal Q-values. Moreover, we found how to best configure the quantum model hyperparameters. As an extra experiment, we investigate a training technique that is an intermediary between supervised regression and true Q-learning, namely, Bellman updates with a fixed target network using the optimal parameters.

Contents

1 Introduction

Reinforcement learning has seen some significant advancements in recent years. Deep Q-learning algorithms in particular have had amazing results. AlphaGo [7] is a prime example of a deep Q-learning algorithm that has defied expectations.

Quantum computing is very promising because it can be used to significantly reduce the required computation time of certain problems. A universal quantum computer can also solve any problem that can be solved using classical computing. Quantum computing has an advantage for certain problems because an n-qubit system can be used to encode larger vectors than an n-bit system would, where these vectors are specified by the joint quantum state of the n qubits. Parameterized quantum circuits in particular have a lot of expressive power even with a very small amount of quantum gates.

In this thesis we would like to combine these two fields by using the efficiency and expressive power of quantum computing with the versatility of reinforcement learning. In previous works parameterized quantum circuits have been used to serve as models for both supervised and unsupervised learning (see [14] and [15] respectively), taking the role of neural networks, for example. Here we extend this idea to reinforcement learning. This transition is not as simple as one might think, as will follow from the results in the experiments section: chapter 6. In [10], a similar question is asked.

This thesis is structured as follows: in Chapter 2 give a more detailed overview of quantum computing. In Chapter 3 we outline reinforcement learning and specifically Q-learning in more detail. In Chapter 4 we state the research question in full, as well as the relevant sub-questions. In Chapter 5 we describe how we intend to answer these questions. We also explain the model and environment in this chapter. In Chapter 6 we discuss the results of our experiments. Lastly, in Chapter 7 we try to answer the research questions based on the results of our experiments.

2 Quantum Computing

This chapter will give a short introduction to Quantum Computing and the relevant concepts for this thesis. This chapter is based on [1] and [2]. For a more detailed overview of quantum computing please consult these books.

2.1 Quantum mechanics for quantum computing

In order to understand Quantum Computing we must first understand qubits and some basic concepts of quantum mechanics.

2.1.1 Qubits and Superposition

For classical computers, information is carried in bits. A bit can be in 2 states (usually called 0 ($|0\rangle$) and 1 ($|1\rangle$). For quantum computers, information is carried qubits (short for: quantum bits). Qubits can also be in states $|0\rangle$ and $|1\rangle$, but they can also be in any superposition of these two states.

More generally, a pure quantum state $|\phi\rangle$ of a system that can be in N distinct states is a superposition of all classical states in it's system. All states $|j\rangle \in \{|0\rangle \dots |N-1\rangle\}$ have their own complex amplitude α_j within the greater quantum state $|\phi\rangle$. The quantum state $|\phi\rangle$ is then the linear combination of all classical states multiplied by their amplitudes:

$$|\phi\rangle = \sum_{j=0}^{N-1} \alpha_j |j\rangle ,$$

with the restriction that $|\phi\rangle$ is normalized: $\sum_{j=0}^{N-1} |\alpha_j|^2 = 1$. For example, a single qubit (with $N = 2$ total states) can be in a superposition of states $|0\rangle$ and $|1\rangle$, i.e., $|\phi\rangle = \alpha |0\rangle + \beta |1\rangle$, with $|\alpha|^2 + |\beta|^2 = 1$. A specific example of such a state would be the state: $\frac{1}{\sqrt{2}} |0\rangle + \frac{1}{\sqrt{2}} |1\rangle$.

The set of all classical states $|0\rangle$, $\dots$, $|N-1\rangle$ of a quantum system forms an orthonormal basis of an N-dimensional vector space. $|0\rangle$ would correspond to e_0, $|1\rangle$ to e_1, $|N-1\rangle$ to e_{N-1}, etc. Hence, each quantum state $|\phi\rangle$ is an N-dimensional unit vector. Which can be expressed with respect to the basis of classical states as the vector:

$$|\phi\rangle = \begin{pmatrix} \alpha_0 \\ \vdots \\ \alpha_{N-1} \end{pmatrix}$$

It is also possible to combine the quantum states of two objects into one combined quantum state. The resulting state is the tensor product (denoted as $\otimes$) of the two original states. Likewise the combined state space is spanned by the tensor products of the basis states of the two objects. Let A and B be two quantum states, that can be written as a superposition over: $|0\rangle, \dots, |N-1\rangle$ and $|0\rangle, \dots, |M-1\rangle$ respectively. The set $\{|j\rangle \otimes |k\rangle : j \in \{|0\rangle, \dots, |N-1\rangle\}, k \in \{|0\rangle, \dots, |M-1\rangle\}\}$ is then an orthonormal bases for the combined state space of A and B.

A qubit is an example of a quantum system with a 2 dimensional state space, so $N = 2$. It has basis states: $|0\rangle$ and $|1\rangle$. Therefore a system with two qubits has the basis states $\{|0\rangle \otimes |0\rangle$, $|0\rangle \otimes |1\rangle$, $|1\rangle \otimes |0\rangle$, $|1\rangle \otimes |1\rangle\}$. Sometimes abbreviated to $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ or $\{|0\rangle, |1\rangle, |2\rangle, |3\rangle\}$. In their vector representation they look like this: [1]

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, |00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, |10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, |11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

In general a system with n qubits has a set of basis states that forms the standard orthonormal basis for $\mathbb{C}^{2^n}$.

[1] $|0\rangle$ and $|1\rangle$ refer to the single qubit case

2.1.2 Measurements and Bases

A state-vector of a quantum state $|\phi\rangle$ cannot be directly observed. Instead, one of the classical states $|j\rangle$ will be observed when a quantum state is measured. Which one depends on the amplitudes of the superposition. Namely, the probability of observing a state $|j\rangle$ is given by $|\alpha_j|^2$. Because the quantum state always collapses to one of the classical states, the α_j have the restriction that:

$$\sum_{j=0}^{N-1} |\alpha_j|^2 = 1$$

In other words, the norm of the vector $|\phi\rangle$ must be 1.

When $|\phi\rangle$ is measured and $|j\rangle$ is observed, $|\phi\rangle$ collapses to $|j\rangle$. Meaning that later operations act on state $|j\rangle$, not $|\phi\rangle$ (until the state is changed again, of course).

In linear algebra it is possible have different bases for the same vector space. For example: the standard basis for $\mathbb{C}^2$ is $\{e_0, e_1\}$. However, it is possible to find a set of vectors $\{v_0, v_1\}$ that form a different orthonormal basis spanning the same vector space, for example:

$$e_0 = \begin{pmatrix}1\\0\end{pmatrix} = |0\rangle, e_1 = \begin{pmatrix}0\\1\end{pmatrix} = |1\rangle, v_0 = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}}, v_1 = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

Similarly, it is possible to consider superpositions in different bases. We define e_j as the vector with 1 at index j and zeroes elsewhere. Using this notation, we define $|j\rangle = e_j$. We call $\{|0\rangle, \ldots, |N-1\rangle\}$ the 'standard basis' or 'the computational basis'. Normally measurements are taken in the standard basis (the computational basis). But it is also possible to measure qubits in different orthonormal bases. This has the added side effect of collapsing the superposition to one of the base-vectors from the basis used for measurement.

2.1.3 Entanglement

It is also possible for two qubits in quantum states $|\phi_1\rangle, |\phi_2\rangle \in H_1, H_2$ respectively to be entangled with each other. The combined state $|\phi\rangle \in H_1 \otimes H_2$ is entangled if you cannot write $|\phi\rangle$ as the tensor product $|\phi\rangle = |\phi_1\rangle \otimes |\phi_2\rangle$. What that means is that measuring the state of the first subsystem will not only collapse the superposition of that subsystem, but the superposition of the whole system, thus the superposition of the second subsystem as well, and vice versa. Note that this is different from combining the states as discussed in 2.1.1.

2.1.4 Unitary evolution

A quantum state can also be changed without measuring (and thereby collapsing) the superposition. If superpositions can be thought of as vectors, operations on quantum states can be thought of as matrices. Performing the operation on the quantum states then becomes multiplying the matrix and the vector (with the matrix on the left). Suppose we have a quantum state $|\phi_1\rangle$ and an operator U. Then $|\phi_2\rangle = U \cdot |\phi_1\rangle$ would also have to be a quantum state. But superpositions can only be changed in ways that preserve their constraints mentioned in 2.1.2. This means only unitary operations can be performed on superpositions. This is because unitary matrices preserve the norm of the vectors

after multiplication, since the determinant of a unitary matrix is always 1. Meaning $|\phi_1\rangle$ and $|\phi_2\rangle$ have the same norm, namely 1.

By definition, unitary matrices are invertible, with the inverse of a unitary matrix also being a unitary matrix. This means that for any operation on a set of qubits (that doesn't measure any of them) there exists an operation that reverses it's effects.

2.2 Quantum Gates

Similarly to classical computers the flow of information in a quantum computer is controlled by gates, quantum gates in this case. Unlike classical circuits, it is impossible to have two qubit fan out of one qubit because it is impossible to copy the superposition of one qubit onto another qubit. There are 3 categories of quantum gates:

- Single qubit gates: These unitary operations, represented by a unitary matrix, are applied to a single qubit. They function as explained in 2.1.4. These can also be extended to act on multiple qubits, but they apply the same unitary operation on all of them independently, therefore it would be the same applying the single qubit gate to all of the affected qubits.
- Controlled gates: These perform a unitary operation to one of the qubits (the target qubit) depending on the state of the other qubit (the control qubit). It does not measure the control qubit and cannot be represented by the tensor product of two unitary matrices. Therefore it entangles the two qubits, see 2.1.3. Two qubit gates are the only way to introduce entanglement between two qubits. This is important because entanglement is a powerful resource in quantum information.
- Measurements: These measure a qubit, thereby collapsing the superposition to one of the basis states (note that measurements can be in different bases 2.1.2). Because they always collapse superpositions this operation is not unitary[2].

2.2.1 Quantum Circuit Diagrams

Because quantum gates have different restrictions from normal gates, it makes sense that the diagrams used to represent classical circuits are also different from those used to represent quantum circuits. An example of a quantum circuit diagram is shown below in Figure 1.

In the circuit displayed in Figure 1 each qubit is assigned a horizontal line. Quantum gates are often represented by boxes on the line of the qubit(s) they act on. The exception are gates that require control qubits, for them a vertical line is drawn from a dot on the control qubit's line to the box representing the gate. The $\oplus$ in Figure 1 represents the CNOT gate, more on that in 2.2.2. The circuit should be read from left to right in the sense that chronologically the gates on the left are applied to the qubits before the gates on the right.

[2]Measurements are not technically quantum gates because they collapse a superposition and are therefore not reversible. Which means they can not be represented by unitary matrices. But they do affect qubits and are drawn in diagrams of circuits like gates, so I will include them in this category.

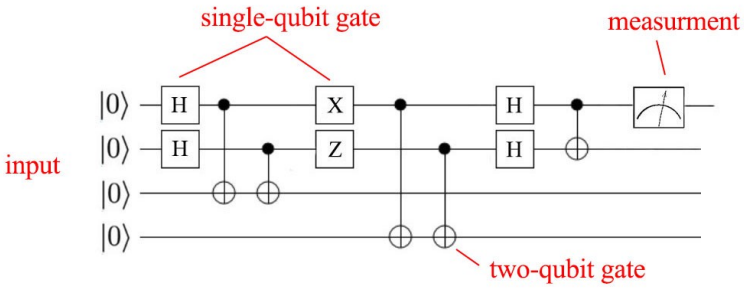


Figure 1: An example of a 4-qubit quantum circuit diagram. Source: [3]

2.2.2 Examples of quantum gates

In theory, since there is an infinite set of (2x2)-unitary matrices, there is also an infinite set of (1-qubit)-quantum gates. This section will outline some of the most commonly used quantum gates. Firstly, the Pauli-X, -Y and -Z gates:

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

The Pauli-X gate is the quantum equivalent of the classical not gate in the sense that it maps $|0\rangle$ to $|1\rangle$ and vice versa.

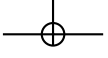
Another common quantum gate is the Hadamard gate:

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

It maps $|0\rangle$ and $|1\rangle$ to $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$, respectively. Both of these superposition states have a 50% chance of collapsing to $|0\rangle$ and a 50% chance of collapsing to $|1\rangle$ when measured. In essence turning classical states into uniformly distributed superpositions. The Hadamard Gate also has the interesting property that it is it's own inverse:

$$\boxed{H^2} = \frac{1}{\sqrt{2}} \cdot \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \boxed{I}$$

The last quantum gate we discuss in this section is the controlled Pauli-X gate, or controlled not gate, or CNOT for short:



$$= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

The CNOT gate applies the Pauli-X gate to the second qubit if and only if the first qubit is in state $|1\rangle$. More generally it maps 2 qubits in states $|a\rangle$ and $|b\rangle$ to states $|a\rangle$ and $|a \oplus b\rangle$.[3] This is why the part of the CNOT that effects a qubit is often denoted by $\oplus$ instead of $\boxed{X}$.
Because the state of the second qubit depends on the state of the first qubit after the CNOT gate is applied, the CNOT gate entangles the 2 qubits. Therefore, the matrix for CNOT cannot be written as the tensor product of two other quantum gates (i.e. unitary matrices).

2.3 Parameterized Quantum Circuits

The quantum gates can also be parameterized, meaning that the matrix representation of the gate has a tunable parameter θ. Or alternatively the matrix is raised to the power θ. For many of the physical implementations of quantum gates the qubit is exposed to some type of external field, depending on the exact implementation of the qubit. The strength and duration of the field can then be used to specify θ.

Here are three examples of parameterized rotations over different axes, that we will use in this thesis. They are the $R_x(\theta)$,$R_y(\theta)$ and $R_z(\theta)$, with their respective matrix forms:

$$R_x(\theta) = \begin{pmatrix} \cos(\frac{\theta}{2}) & -i\sin(\frac{\theta}{2}) \\ -i\sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{pmatrix}, R_y(\theta) = \begin{pmatrix} \cos(\frac{\theta}{2}) & -\sin(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{pmatrix}, R_z(\theta) = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$$

A parameterized quantum circuit is a quantum circuit that uses a combination of parameterized quantum gates (at least one) and regular quantum gates. The architecture of this circuit is fixed. The parameters of the gates specify a particular circuit, obtained by filling in the parameters into the fixed architecture. See Figure 7 and Figure 9 for examples of parameterized quantum circuits. An example of a particular circuit obtained by filling in the parameters into one of these parameterized quantum circuits can be found in Figure 8.

3 Reinforcement Learning

This chapter will give a brief introduction to reinforcement learning and the relevant concepts. it will also outline: agents, environments, Q-learning, models and learning methods in reinforcement learning. This chapter is based on [4]. For a more in depth look at reinforcement learning please consult this book.

3.1 Agents and environments

Reinforcement learning is about an agent learning to fulfill an objective and maximizing a reward function. Agents can take many forms. A character in a video game trying to complete a level or an autonomous vehicle trying to travel safely from one location to another, to name a few.

[3] $\oplus$ represents summation modulo 2 (also known as binary xor) in this context

An agent is always situated in an environment with a certain set of rules and constraints. The self-driving car for example is bound by the laws of physics as well as the traffic laws applicable at the road it is on. In some environments there are also other agents to interact with. These could be other vehicles or pedestrians for the self-driving car. The agent does not always know everything about the environment. It can use its sensors to get information about the environment. For the autonomous vehicle, for example, the sensors could be cameras, radar or even data about other autonomous vehicles. The agent can perform actions, which potentially change the state of the environment. For the autonomous vehicle these actions would be things like a breaking or acceleration.

The abstract specification of what action the agent takes in which situation (state) is called the policy It can be as simple as always move right or as complicated as a process modelled by a deep neural network (c.f. 3.4.1). Policies are often initialized randomly or by a predetermined default value. The agent then interacts with its environment by performing an action. The environment then returns a reward back to the agent, based on the state of the environment and the action played by the agent. The agent then uses the reward to adapt it's policy such that it can maximize the discounted reward it gets over time. Discounted, in this sense, means that rewards in the distant future are valued less then more immediate rewards. This is implemented by using a discount factor γ, which is multiplied by the sum of expected rewards for every timestep it would take to reach that reward. This would make the expected discounted reward G_t at timestep t the sum of all futute rewards R_k at later timesteps k, multiplied by the discount factor γ raised to the power m, where m is the number of timesteps to reach R_k:

$$G_t = \sum_{k=t+1}^{\infty} \gamma^{k-t-1} \cdot R_k$$

Note that this sum is computed up to infinity. However, most useful policies have the agent terminating the game after a finite amount of time (steps) T_f. Or at the very least most training methods terminate the play through after a predetermined number of steps T_t, meaning that, in practice, this is often a finite sum up to T_f or T_t, respectively.

3.2 Value functions and Q-values

Q-learning is a type of reinforcement learning that attempts to assign a value to every action in every state. This score is called the Q-value and it represents the total discounted expected reward over time, assuming the agent plays by a given policy π from that point on.
The value $v_\pi(s)$ of a state s when playing according to policy π would then be the expected value of G_t given that the agent plays by π and the agent is in state s at time step t:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^{t+k} \cdot R_{k+t+1}|S_t = s]$$

To get the Q-value $Q_\pi(s, a)$ we can simply take $v_\pi(s)$ and add the constraint that action a is played

at time step t:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^{t+k} \cdot R_{k+t+1}|S_t = s, A_t = a]$$

This mapping of all state-action pairs to their Q-value is called the Q-function. There are two main ways of implementing Q-functions, the first way is to store the Q-values in a (Q-)table. This is often preferred for smaller discrete environments, as the size of the table scales with the amount of different actions and states. The second way is to compute Q-values directly using a some mathematical function(s) or model (as discussed in Section 3.4).

3.3 Learning algorithms

The first thing we should discuss when talking about learning algorithms is exploration versus exploitation. The agent can explore the environment, in which case the agent plays a random action with the intention of visiting states it has not yet visited to better understand the potential rewards of the state from which that action was taken. The agent can also exploit its current knowledge of the environment and greedily chose the action with the highest Q-value. There is always a trade-off between exploration and exploitation. On the one hand we would like to explore the environment as much as possible, but on the other hand we want the agent to take the most promising actions to gain a higher reward. This is why the agent usually takes more exploratory actions in the early stages of the training, as the policy is still not very optimized for the environment. Later it will exploit its knowledge more as the policy should be better optimized for the environment (at least in theory it should be better optimized).

When computing Q-values as defined in 3.2, there is a shortcut: instead of considering every possible future state from an action. It is possible to define Q-values recursively, since the agent will always play according to its policy:

$$Q_\pi(s_t, a_t) = r_t + \gamma \cdot Q_\pi(s_{t+1}, a_{t+1}) \mid a_{t+1} = \pi(s_{t+1})$$
$$Q_\pi(s_t, a_t) = r_t + \gamma \cdot Q_\pi(s_{t+1}, \pi(s_{t+1}))$$

This equation is called the Bellman equation. Here, s_t and a_t are respectively the state and action played at time t. Also, r_t is the reward the agent gets immediately after playing a_t from s_t. Finally, γ is the discount factor and it penalized future rewards for every step it takes to get to that reward.

A policy π is called greedy, if it always selects the option with the highest expected reward. In other words $\pi(s) = \max_a(Q_\pi(s,a))$. For a greedy policy the Bellman equation simplifies to:

$$Q_\pi(s_t, a_t) = r_t + \gamma \cdot \max_{a_{t+1}}(Q_\pi(s_{t+1}, a_{t+1}))$$

All Q-values in an optimal policy have to satisfy this Bellman equation. Moreover, as the policy gets better optimized for the environment, the Q-values converge to those that fulfill the Bellman equation. In other words the Bellman equation represents the final state of the Q-values; after training. To get to that stage we can use temporal difference learning:

$$Q^{new}(s_t, a_t) = Q^{old}(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_{a_{t+1}}(Q^{old}(s_{t+1}, a_{t+1})) - Q^{old}(s_t, a_t))$$

The α in this equation is the learning rate. It is hyperparameter smaller than 1. It ensures that the learning process happens smoothly and that the policy doesn't make too big jumps in one update. The rest of the variables have the same meaning as they do in the Bellman equation.

When the environment is finite and small enough the entire entire Q-function can be stored in one table. For each possible state the table has a row and for each possible action the table has a column. The Q-value $Q_\pi(s, a)$ is then the entry of the table in the s^{th} row on the a^{th} column. The table is usually initialized randomly, but depending on the environment it can also be initialized to some default value (like zero everywhere). In this case the agent can simply episodically run through the environment and update the Q-values using temporal difference learning:

Algorithm 1: Temporal Difference Learning for tabular Q-values

```
initialize_policy();
ϵ ← 1 ;
for episode in range(max_episodes) do
    environment.reset();
    for step in range(max_steps) do
        #exploration/exploitation tradeoff:
        exp_tradoff ← random(0,1);
        if exp_tradoff < ϵ then
            action ← argmax(Qtable[state]);
        else
            action ← env.action_space.sample();
        end
        #perform action and update the state, reward, done variables
        new_state, reward, done, info ← env.step(action) ;
        Qtable[state, action] ←Qtable[state, action] + α · (reward + γ·
         max_action(Qtable[new_state, ·]) − Qtable[state,action])
        state ← new_state ;
        ϵ← reduce_epsilon(ϵ)
        if done then
            break ;
        end
    end
end
```

When the environment is more complicated and a model is used to estimate Q-values this approach has to be altered somewhat because Q-values can no longer be altered directly. Instead the parameters of the model have to be changed such that the model converges to the optimal model. This is usually achieved by performing a gradient descent step on the model. This finds a model in a local minimum near the current model. The parameters of the old model are then moved by α in the direction of the parameters of the target model.

3.4 Function Approximation in Reinforcement Learning

Sometimes it is not practical, or even possible, to store all Q-values of a policy directly. Because the state space of the problem is either continuous or just to large. In this case Q-learning is still possible, except all Q-values now have to be computed directly, when they are needed. This paragraph will outline different models used to compute Q-values, why these models are relevant and how quantum computing can be used as a model to compute Q-values.

3.4.1 From Linear functions to Deep Neural Networks

Perhaps the simplest type of model with which to compute Q-values is a linear model. A simple line in a 2-dimensional space that can easily be specified by two parameters, the slope of the line and the vertical shift. There are two main uses for linear models in machine learning; classification and regression. In regression the model tries to fit a line to the data by trying to reduce the mean squared error between the line and the data points available. In classification the model will try to classify the different data points into a few different categories, where the line acts as a boundary between the (two) different groups. A visual representation of the difference between classification and regression can be seen in Figure 2.

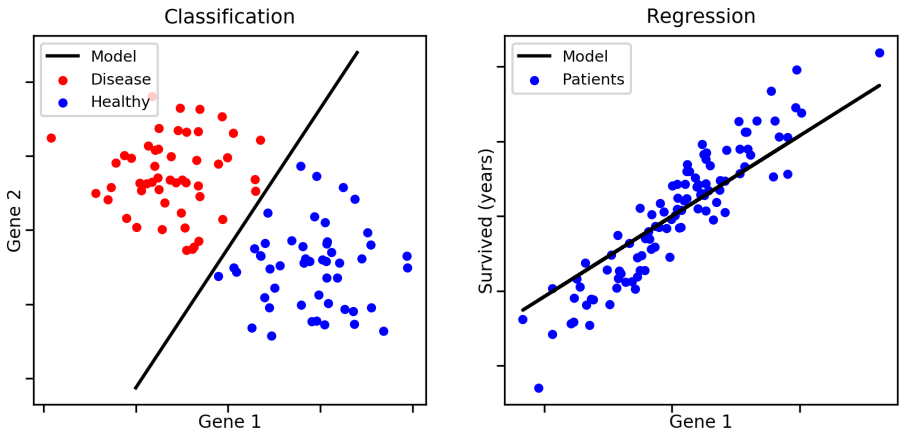


Figure 2: A linear model used for classification (left) or regression (right). Source: [5]

Since we are trying to teach an agent to chose the best action in a given state, that could be considered classifying state-action pairs as either good or bad actions (with the restriction that the agent has to chose exactly one action in every state). In fact if you give the agent access to the Q-value of each state-action pair this would be exactly what you are doing. But that would also be supervised learning and not reinforcement learning. So in Q-learning the agent should not know what the Q-values are beforehand and should only find out what works trough trial and error.

Linear models can also be used to directly compute Q-values. In equation form this usually looks as follows: $Q(s,a) \approx w^T \cdot \phi(s,a) + b$. In which $\phi(s,a)$ is a vector representation of the state-action pair (s,a) and w and b are the parameters that need to be trained.

Sometimes the classification of the different states is too complicated to be divided or estimated by a straight line. In that case, the model needs to assign a Q-value to each state-action pair and select the one with the highest value. At that point it could be very impractical to classify anything using "simple" mathematical functions. Therefore, more advanced models are necessary.

An artificial neural network is inspired by the structure of the the human brain; like the brain a neural network passes information between neurons (nodes or cells) which pass on information to other neurons based on the strength of the connections (weights on vertices or thickness of the axons). In general, a neural network is a weighted directed graph with a subset of input nodes, a (few) subset(s) of hidden nodes and a subset of output nodes. The different subsets of nodes form layers. There are no connections between nodes in the same layer and generally only connections from nodes in one layer to nodes in the layer directly behind that layer. There are exceptions to this rule, for example when there is a connection between nodes that skips a (few) layer(s). It is typically also the case that every node in a given layer is connected by a directed vertex to every node in the layer before it (these are the inputs for that node) and every node in the layer behind it (for which it acts as an input). Below in Figure 3 is an example of a neural network:

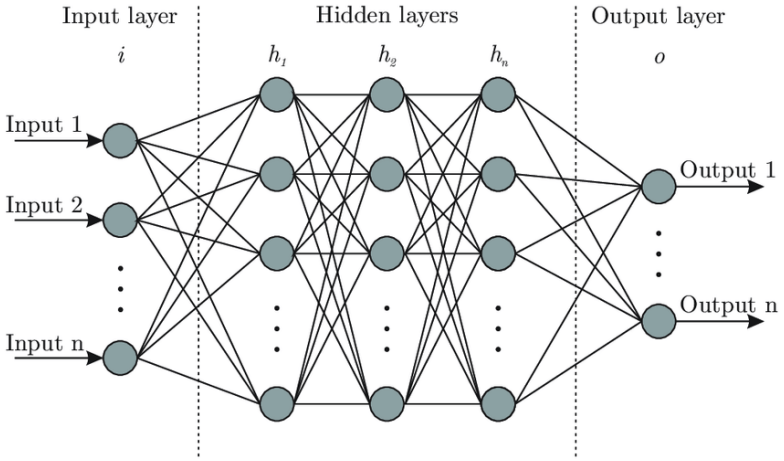


Figure 3: An example of a (feed-forward) neural network with three hidden layers. Source: [6]

The value of each hidden or output node in the neural network is computed by taking the sum of the value of its input nodes multiplied by the weight on the vertex connecting the respective input node to the current node. After this computation the values of the nodes are normalized by a so-called activation function. The values are usually normalized to be between -1 and 1 or 0 and 1, but this may vary.

3.4.2 Importance of models

Tabular Q-learning based directly on using the Bellman equation can be very useful. But the requirement to store all Q-values limits this method to small discrete environments as the space-complexity tends to explode for larger environments.

Deep neural network based Q-learning, also called deep Q-learning, has made some significant advances in recent years. Consider AlphaGo (see [7]), which in 2016 had beaten Lee Sedol, the second highest ranking Go player at the time, without a handicap[4]. But deep Q-learning has also made significant advances in other fields like image recognition.

3.4.3 Quantum models in reinforcement learning

Quantum computing has the potential to significantly reduce the complexity of certain computations. An example of which would be prime factorization using Shor's Algorithm (see [8]). Currently large-scale, fault-tolerant quantum computers don't exists, nor will they likely exist in the very near future. So the number of qubits that can be used is very limited. The number of operations applied on those qubits before they succumb to noise is also very limited. Despite that, there are still uses for these smaller scale quantum computers.

One of the ways to still utilize noisy intermediate scale quantum computers is by using parameterized quantum circuits (PQC) (see 2.3). Because they can still have a lot of expressive power, even with a relatively low amount of qubits and quantum gates. This could make them suitable models for computing Q-values. And in fact, they have been used for both supervised and unsupervised problems before (see [14] and [15], respectively). The parameters of the model have to be optimized using some sort of classical training loop, just like any other model. Figure 4 shows the general idea behind a parameterized quantum circuit as model for Q-learning:

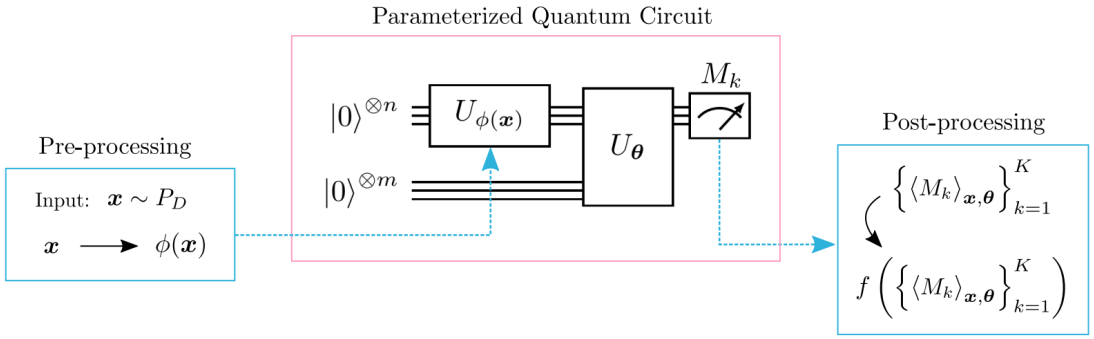


Figure 4: Machine learning models, using parameterized quantum circuits. Source: [9]

This model can be split up into three different phases:

- A pre-processing phase, in which the state x of the environment is used to generate a list of parameters $\phi(x)$ that are used as parameters of a circuit that is applied to the all-zeroes state. Effectively encoding x into a quantum state.
- A parameterized quantum circuit phase, in which a quantum circuit U_θ is applied to the qubits. Here, θ represents the list of all tuneable parameters in this context. Some of these gates of U_θ can be regular quantum gates, but there have to be some parameterized quantum gates.

[4]No handicap meaning that the AI did not receive extra stones at the beginning of the match

- A post-processing phase, in which the final state of all qubits is measured and the result is analyzed further using classical algorithms.

Next, one repeats the steps above, while tuning the parameters θ using a classical optimisation algorithm along the way. Thereby training the quantum model using a classical computer.

4 Problem Statement

In this chapter we will go over the main research questions of this thesis. In section 3.4.3 the general structure of a quantum model for Q-learning is outlined. An immediately evident question is whether we can train a parameterized quantum circuit using reinforcement learning techniques?

From previous works (see [**?**]) we know that for classification regression problems, we can use a parameterized quantum circuits as a substitute for a neural network and train it using similar techniques. We have tried training the parameterized quantum circuit for many different hyper-parameter configurations. But none of them were very successful. All of the pure reinforcement learning runs we tested had a progression similar to the run in Figure 5, the metrics by which we judge the model will be explained in section: 5.4.
It is evident from this figure that the model does not improve over time. This immediately

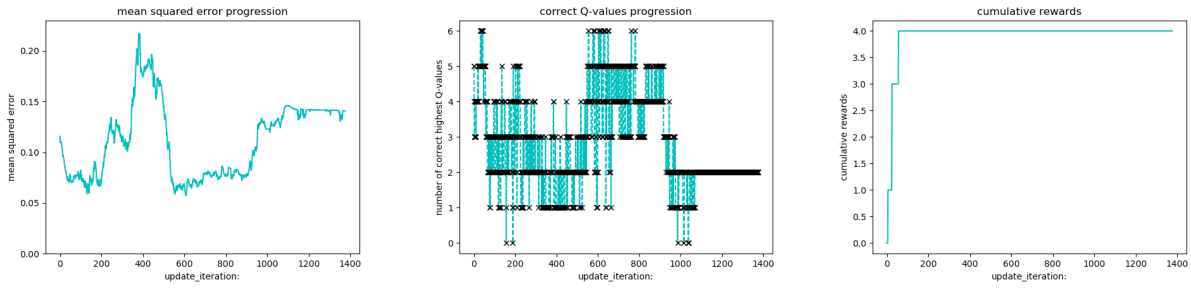


Figure 5: On the left: the progression mean squared error between the theoretical Q-table and the Q-table if it were computed by the model, after each episode. In the center: the number of states for which the highest Q-values in the theoretical and computed Q-tables match. On the right: the cumulative reward over all episodes. All three are using the model parameters: amplitudes/action = 2, correction factor = 3, $\gamma = 0.8$

raises the question of whether or not our parameterized quantum circuit is expressive enough to accurately represent all Q-values for FrozenLake. Moreover, as the use of parameterized quantum circuits introduces new quantum-specific hyperparameters, we would also like to find out how to best configure these. To answer these questions we will use regression to train the parameterized quantum circuit so that its output is a close approximation of the theoretically computed Q-values for FrozenLake. Furthermore, this allows us to sweep the new hyperparameters introduced by this quantum model and find out what configuration works best. It is easier to investigate the best configuration of these new hyperparameters using regression because than we don't have to deal with the many extra hyperparameters and challenges that come with reinforcement learning.

As an extra experiment we take one more step towards reinforcement learning (as opposed to supervised regression), Namely, knowing the optimal configuration of the parameterized quantum circuit, we can use the optimal parameters[5] as a target network to apply Bellman updates with. Doing this will allow us to investigate the stability of our model in setting that is slightly closer to full-blown reinforcement learning.

To sum up, motivated by the shortcoming of standard Q-learning techniques to train our parameterized quantum circuit model, we investigate the following three main research questions:

- Is our parameterized quantum circuit expressive enough to accurately represent the optimal Q-values?
- What is the optimal hyperparameter configuration for the newly introduced quantum hyperparameters (i.e., number of amplitudes per action and correction factor)?
- Is our model stable enough to stay near an optimal configuration when we apply Bellman updates with a fixed target?

5 Methods

This chapter will outline the methods used to get an answer to the main research questions of this thesis.

5.1 The environment: FrozenLake

Because we expect that the training of our model might be difficult, we will focus on a simple environment for this thesis. The environment we will use to train the model is called FrozenLake, it is played on a 4x4 grid. The agent can be in one square at a time and can move up, down, left or right to a new square. The grid does not loop around and the agent cannot move off the grid. The agent starts the game in the top left of the grid on the square labeled S (for start). The game ends when the agent moves onto one of the squares labeled H or G (for hole or goal respectively). If the game ends with the agent on a square labeled H the reward is 0 (the agent should avoid the holes). If the game ends with the agent on the square labeled G the reward will be 1 (the agent has won the game). Figure 6 shows the environment for FrozenLake:

[5] parameter in this context refers to the quantum gate parameters not the models hyperparameters

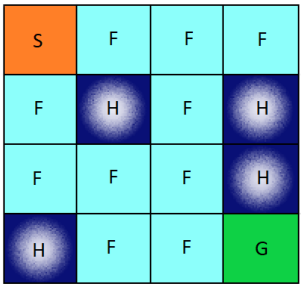


Figure 6: The environment for FrozenLake: Source: [11]

5.2 The Quantum Model

We will use 4-qubit quantum circuit, inspired by the so-called "hardware efficient Ansatz" from [14]. We will be simulating the quantum circuit and FrozenLake using the python libraries cirq [12] and openAI_gym [13]. We will not run the algorithms on a real quantum computer, when computing Q-values. Instead we will simulate the circuit using vector and matrix multiplications and directly use the amplitudes of the different basis states.

5.2.1 Pre-processing

The Pre-processing phase of this model is rather straightforward. Since FrozenLake is played on a 4x4 grid and the state is solely dependant on the position of the agent on that grid. There are 16 possible states in FrozenLake. We can simply number these states 0-15 and then encode each state to the binary representation of its number. Since there are 16 states we would need 4 qubits to encode them all into basis states.

This would be implemented as follows: given a state-number n, let N be the list of 0's and 1's that represents the binary notation of n. Start with combined state $|0\rangle$ (every qubit in state $|0\rangle$) then simply apply the NOT-gate to every qubit that corresponds to a 1 in N. In other words, apply the NOT-gate to the power $N[i]$ to the i^{th} qubit, for every qubit. Figure 7 shows the parameterized quantum circuit we use for this "basis state encoding" for any game-state n. Figure 8 shows this circuit when we fill in the parameters for game-state 5.

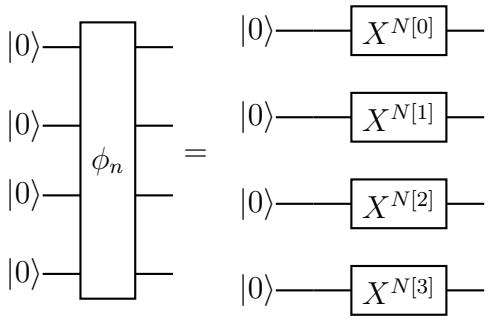


Figure 7: This is a general overview of the basis-state encoding we used, for any game-state n, with binary representation N.

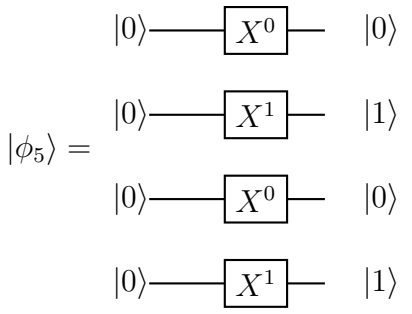


Figure 8: This is an example of the basis-state encoding for game-state 5, with binary representation 0101.

5.2.2 Quantum Circuit

This part of the model in particular is heavily based on the "hardware efficient Ansatz" from [14]. The parameterized quantum circuit is built up from several layers. Alternating between entanglement layers and a rotation layers. The entanglement layer connects every qubit to the next qubit using a CNOT-gate (see 2.2.2). Whilst the rotation layer consists of two parameterized rotations (see 2.3), one around the Y-axis and one around the Z-axis. One layer of this circuit is broken down in Figure 9.

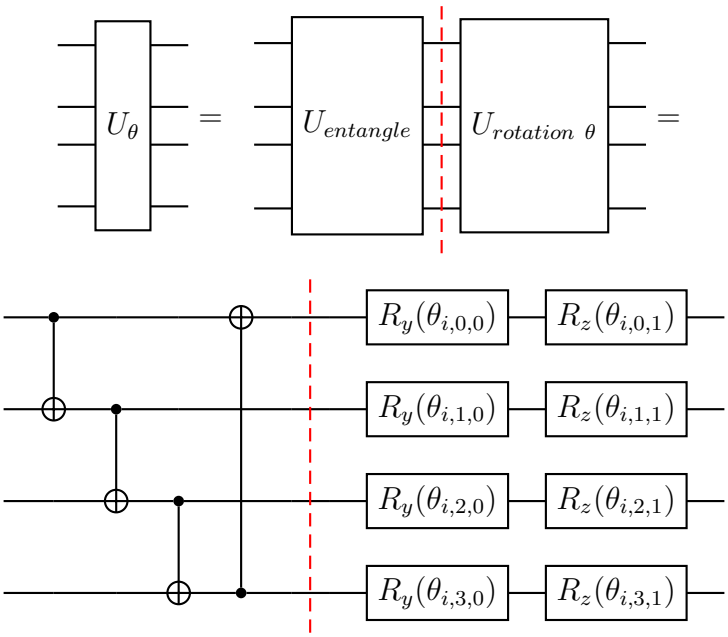


Figure 9: This is the breakdown of one layer of the parameterized quantum circuit. This part of the circuit is repeated as much as the depth of the circuit.

This layer can be repeated any number of times. The depth of the circuit is defined as the number of repetitions of this layer. The total circuit is then set up as follows: start with the pre-processing circuit ϕ_n defined in 5.2.1. Follow that up by one rotation layer $U_{rotation\ \theta}$. Then repeat an entanglement layer $U_{entangle}$ followed by $U_{rotation\ \theta}$ for the depth of the circuit[6]. Figure 10 shows the complete overview of the parameterized quantum circuit we used.

This kind of circuit has a few notable analogies to neural networks, namely:

- Intuitively, the parameters θ_i correspond to the weights of a neural network; these θ_i are ultimately what changes during the training phase.
- The depth of the circuit is variable like the depth of a neural network.
- The qubits are very interconnected between the layers due to the entanglement layers. This is comparable to the many connections between the nodes in a neural network.

[6]Please note that for the purpose of this theses we will only simulate the states of the qubits and not simulate (multiple) measurements.

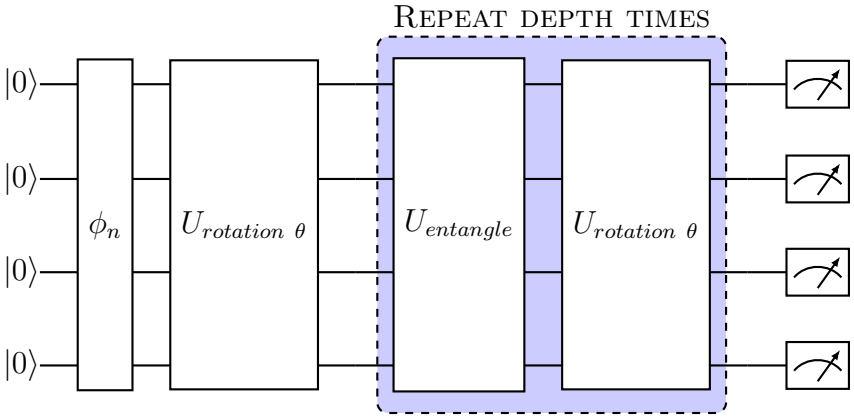


Figure 10: This is a full diagram of the circuit we used. The ϕ_n sub-circuit is broken down in Figure 7. The $U_{rotation\ \theta}$ and $U_{entangle}$ sub-circuits are broken down in Figure 9

5.2.3 Post-processing

There are 4 qubits and thus 16 amplitudes to consider. Despite that there are only 4 actions that can be taken in FrozenLake: Up, Down, Left, Right. This means that we can assign up to 4 amplitudes to each action; discarding all amplitudes that aren't assigned to an action.
To find out which amplitudes to assign to which action we will first define 4 different sets of numbers with the property that: for every number n, in its binary representation, we would need all 4 (qu)bits to determine in which set n belongs. This is because we want our outcome to depend on all qubits, because single qubit measurement outcomes can be efficiently simulated classically. These are the sets we have chosen:

- left: {0, 5, 10, 15},
- down: {1, 6, 11, 12},
- right:{2, 7, 8, 13},
- up: {3, 4, 9, 14}.

The choice of these sets is somewhat arbitrary except for the property mentioned above.

As mentioned before there are four different configurations we will test. For the first one, the Q-value is simply the square of the amplitude of the first state in the list. For the second one it is the sum of the squares of the first two amplitudes in the list. This pattern continues until the forth one, which is the sum of the squares of all four amplitudes in the list.

Each of these Q-values is then multiplied by a constant correction factor; a hyperparameter to be set beforehand. This parameter is used to compensate for the fact that the squares of all amplitudes always sum up to 1 and thus the sum of the Q-values can never exceed 1.

To give a more concrete example: let $|\phi\rangle$ be the final quantum state after the paremeterized quantum circuit has been applied, and let ϕ_j be the $j-$th entry of $|\phi\rangle$ in the standard computational basis.

Let c be the correction factor, and assume that we measure in the standard computational basis. If we use 2 amplitudes per action the Q-values would be computed as follows:

- $Q_{left} = c \cdot (|\phi_0|^2 + |\phi_5|^2)$
- $Q_{down} = c \cdot (|\phi_1|^2 + |\phi_6|^2)$
- $Q_{right} = c \cdot (|\phi_2|^2 + |\phi_7|^2)$
- $Q_{up} = c \cdot (|\phi_3|^2 + |\phi_4|^2)$

The amplitudes $\phi_8 \dots \phi_{15}$ will be ignored in this computation

5.3 Training the Quantum Model

This section will discuss the specific training methods used for the quantum model; a supervised learning algorithm and a reinforcement learning algorithm.

5.3.1 Supervised training

Before taking on the more daunting task of training a reinforcement learning agent it might be necessary to try to find out whether or not the circuit is expressive enough to solve the problem. In short is there a configuration of parameters of the circuit such that: given the state of the system, the circuit can accurately produce the Q-values for each action the agent can take in that state?

The easiest way to answer that question is to theoretically compute the Q-values for every (relevant) state-action pair and using supervised learning techniques to tune the parameters of the circuit such that the Q-values the circuit produces are as close to the theoretical Q-values as possible. The mean squared difference (also called mean squared error) will be used as the value to be minimized.

5.3.2 Reinforcement learning training

The algorithm used to train the model using reinforcement learning is a variation on the one explained in 3.3. The three most significant changes are: one, the fact that the argmax over the Q-table has been replaced by an argmax over the results of the Quantum model (see 5.2). Two the update rule for θ (and by extension the Policy) has been changed to a gradient decent step using scipy.minimize. And three the fact that we now use experience replay, which simply means that we store the last X amount of experiences in a memory, then take a random batch of experience from this memory and apply our gradient decent step on all these experiences. The pseudo code for this algorithm can be seen below in Algorithm 2:

Algorithm 2: Temporal Difference Learning for Quantum Model

initialize_policy();
$\epsilon \leftarrow 1$;
Experiences $\leftarrow$ [] ;
$\theta \leftarrow$ RandomInitialPolicy ;
for *episode* ***in*** range(*max_episodes*) **do**
  environment.reset();
  for *step* **in** range(*max_steps*) **do**
    #exploration/exploitation tradeoff:
    exp_tradoff $\leftarrow$ random(0,1);
    if *exp_tradoff* $< \epsilon$ **then**
      action $\leftarrow$ argmax(evaluate_PQC(θ,*state*));
    else
      action $\leftarrow$ env.action_space.sample();
    end
    #perform action and update the state, reward, done variables
    new_state, reward, done, info $\leftarrow$ env.step(*action*) ;
    Batch $\leftarrow$ randomsample(Experiences) ;
    $last\_step = [reward + \gamma \cdot$ max(evaluate_PQC(θ,*new_state*)), *state*] ;
    #[y_t, s_t] Batch.append(*last_step*);
    Experiences.append(*last_step*);
    #gradient decent step
    Objective $\leftarrow \sum_{i \in Batch}$ (Batch[i,0] - max(evaluate_PQC(θ,B[i,1])))^2
    minimize(Objective, θ) ;
    state $\leftarrow$ *new_state* ;
    $\epsilon \leftarrow$ reduce_epsilon(ϵ)
    if *done* **then**
      break ;
    end
  end
end

5.4 Testing and Benchmarking

The agent will be tested on the game of FrozenLake. The states of FrozenLake will be numbered as follows:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Using this naming of states we can compute the Q-value for each state action pair for a given γ. This can be done by computing the length l of the shortest between the new state (the one the

state-action pair would lead to) and the goal state and taking γ^l as the Q-value for that state-action pair. Unless the next state is a hole (denoted by H), in which case the Q-value will be 0. Or alternatively if the current state is a hole or the goal state, in which case the Q-value is undefined and we will not use it to rate the agent's performance.[7] The resulting Q-table is shown below:

state	left	down	right	up
0	γ^6	γ^5	γ^5	γ^6
1	γ^6	0	γ^4	γ^5
2	γ^5	γ^3	γ^5	γ^4
3	γ^5	0	γ^6	γ^6
4	γ^5	γ^4	0	γ^6
5	X	X	X	X
6	0	γ^2	0	γ^4
7	X	X	X	X
8	γ^4	0	γ^3	γ^5
9	γ^4	γ^2	γ^2	0
10	γ^3	γ	0	γ^3
11	X	X	X	X
12	X	X	X	X
13	0	γ^2	γ	γ^3
14	γ^2	γ	1	γ^2
15	X	X	X	X

There will be two different metrics by which we will judge the agent's performance.[8] Firstly, we can use the model to compute a Q-table for FrozenLake, then compare the mean squared error between the entries of that Q-table and the theoretical Q-table. This would give us a good indication of how different the current model is from a theoretical optimal model. But this is not the only metric that matters. In the end the agent will always select the action with the highest Q-value. Therefore, in order to win the game, it is important that, for each state the action(s) with the highest theoretical Q-value also has the highest Q-value produced by the model. That is why we will also count the number of states for which the highest Q-value produced by the model is matched to the same action as the action with the highest theoretical Q-value in that state.[9]

6 Results

This chapter contains the results of the experiments. We will discuss which configurations of the model and circuit work and which do not, as well as which training methods work and which do not. All parameterized quantum circuits we used for these experiments have a depth of 5.

[7] One could argue that the Q-value in a hole state is always 0 and the Q-value in the goal state is always 1. But since the game terminates after reaching these states it does not matter to the agent which values they have

[8] These are only judgements after the agent has been trained and will not be used for training purposes (except for the supervised learning experiment)

[9] Of course not counting the Hole and Goal states

For the first few experiments we performed, we treated the parameterized quantum circuit as a substitute for a neural network and attempted to train it using the same reinforcement learning techniques. These experiments all had very similar results. The progression of one of them is shown in Figure 11. The configuration we use for this particular experiment is as follows: to compute the Q-values we used 2 amplitudes per action, a correction factor of 3 and a γ of 0.8. For the computation of the gradient we used the scipy function 'minimize' with the method 'Nelder-Mead'. We used an ϵ-greedy strategy with an exponentially decaying ϵ. We also used experience replay, with a batch size of 32. Finally, we updated the policy every 20 steps.

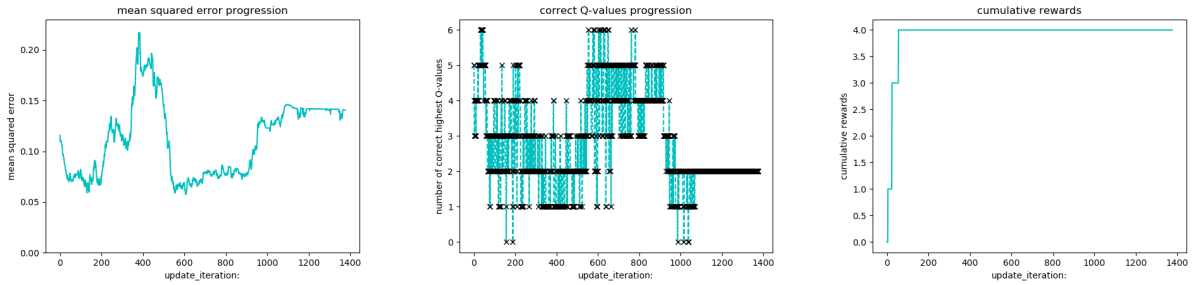


Figure 11: On the left: the progression of the mean squared error between the theoretical Q-table and the Q-table computed by the parameterized quantum circuit, after each parameter update. In the center: the number of states for which the highest Q-values computed by our model and the highest theoretical Q-values correspond to the same action(s). On the right: the cumulative reward over all episodes. All three use these model parameters: amplitudes per action = 2, correction factor = 3, $\gamma = 0.8$

We have tested many more hyperparameter configurations, with the hyperparameters[10] in the following ranges:

- All possible number of amplitudes per action; 1 to 4.
- Correction factors ranging from 1 to 12.
- Several γ's, ranging from 0.8 to 0.99
- Updating the parameters every step versus updating them every 20^{th} step.
- Different batch sizes ranging from 1 to 32.
- Various scipy minimize methods to compute the gradient descent step: Nelder-Mead, COBYLA, Conjugate Gradient (CG), SLSQP and BFGS.

None of these hyperparameter settings yielded models for which the mean squared error between Q-values of the model and the theoretical Q-values converged to zero. Nor did any of the resulting models score particularly well when judging them based on the amount of states for which the correct action is played or their total cumulative rewards. All of the hyperparameter configurations

[10]Note that we have not tested every possible combination of these 7 parameters.

we have tested had results similar to the results in Figure 11

So far we have identified 3 potential problems with our model:

- The parameterized quantum circuit might not be expressive enough to fully represent the environment of FrozenLake.
- The configuration of the newly introduced quantum hyperparameters might still be wrong.
- The parameterized quantum circuit might not be stable enough, meaning a small change in the parameters to improve one Q-value might reduce other Q-values to much.[11]

6.1 Supervised Regression Experiments

This section will show the results of the supervised learning training method. To recap, in section 5.2.3 we have discussed 4 different ways of post-processing the output vector: we can use 1, 2, 3 or 4 amplitudes per action. This gives us 4 slightly different models based on this choice[12].

6.1.1 How does regression progress?

In this section we will try to determine whether or not the parameterized quantum circuit is expressive enough to compute all the Q-values for FrozenLake. We will do this by computing the theoretical Q-values, as discussed in section 5.4, and reducing the mean squared error between those Q-values and the Q-values the model produces as much as possible, using regression.

One of our best results using regression is shown in Figure 12. This Figure shows 20 iteration-steps of the regression model (300 optimization iterations using scipy 'COBYLA' for each of these steps). The exact model we use for this experiment uses 2 amplitudes per action in the post-processing, as well as a correction factor of 3. The discount factor for the rewards was set at $\gamma = 0.8$. This makes it really one data point in the next section, Section 6.1.2, Figure 15.

From Figure 12 We can see that the mean square error converges to a value close to 0 (optimal model) and the number of states for which the correct action has the highset Q-value converges to 11 (all relevant states). From this we can conclude that the model is expressive enough to model the Q-function for FrozenLake.

The final stage of this model has a mean squared error from the theoretical Q-values of approximately 0.0155. And for each of the 11 relevant entries in the Q-table the highest Q-value is the one corresponding to the best action. It plays the following actions: down $\rightarrow$ down $\rightarrow$ right $\rightarrow$ down $\rightarrow$ right $\rightarrow$ right. Following the path shown in Figure: 13.

[11] Changes to Q-values are implemented by changing the parameters and are thus implemented indirectly.
[12] Many more when we consider the possible correction factors and γ's

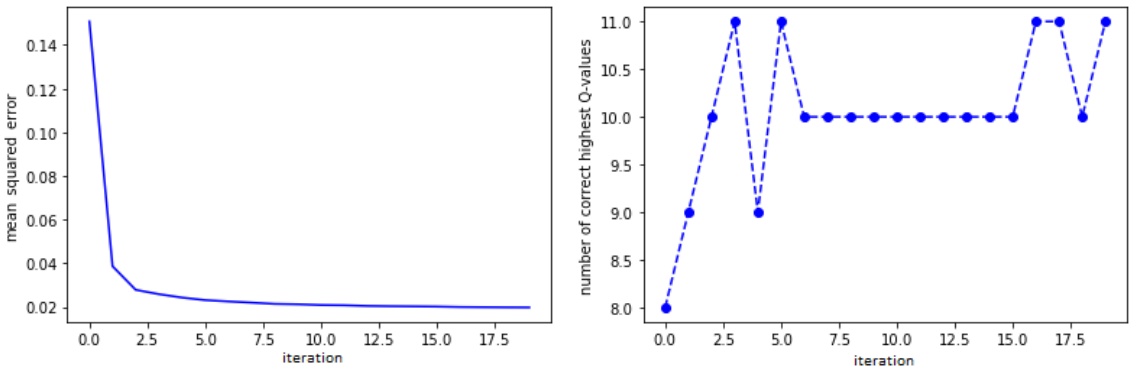


Figure 12: On the left: the progression of the mean squared error between the theoretical Q-table and the Q-table computed by the parameterized quantum circuit, after each parameter update. On the right: the number of states for which the highest Q-value computed by our model and the highest theoretical Q-value correspond to the same action(s). Using these model parameters: amplitudes per action = 2, correction factor = 3, $\gamma = 0.8$

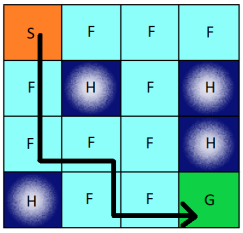


Figure 13: The path our supervised learning agent took. Source: [11] (modified image)

6.1.2 Which configurations work?

In this section we will try to figure out how to best configure the newly introduced quantum hyperparameters. To do this we will test all possible combinations of these configurations: we test all options for the amplitudes per action, 4 different γ's and correction factors ranging from 1 to 9 with steps of 0.5. The next Figures: Figure 14, Figure 15, Figure 16 and Figure 17, show this mean squared error, as well as the number of correctly taken steps in each state, as a function of the constant correction factor. For each Figure the number of amplitudes that is considered increments from the previous Figure. Starting with 1 amplitude per action in Figure 14 and ending with 4 in Figure 17. The 4 different lines in each of these Figures correspond to a γ, as indicated by the legends of the figures.

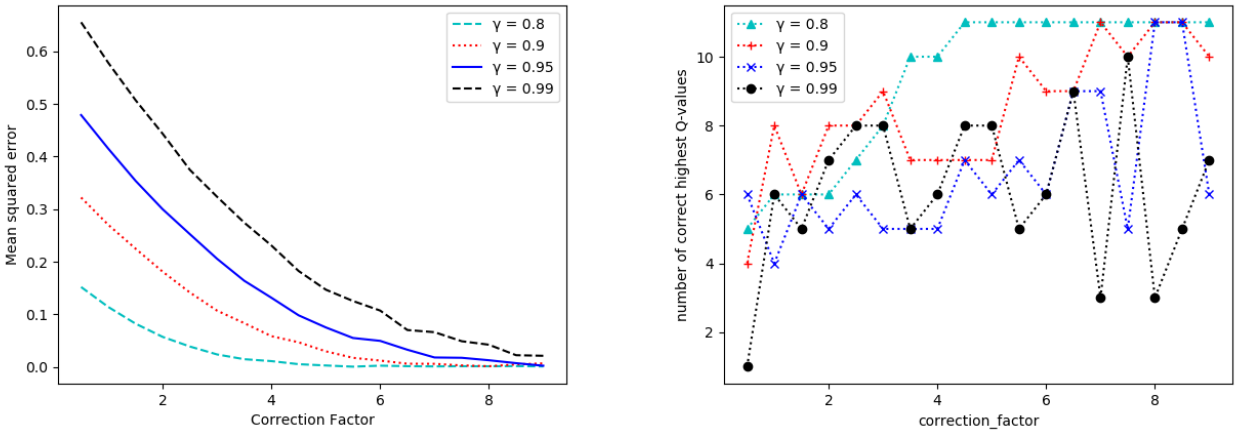


Figure 14: On the left: the mean squared error between the theoretical Q-table and the Q-table computed by the parameterized quantum circuit, as a function of the correction factor that was used. On the right: the number of states for which the highest Q-value computed by our model and the highest theoretical Q-value correspond to the same action, as a function of the correction factor that was used. These were computed when using 1 amplitude per action.

From these plots we can see that for each number of amplitudes we use there is a configuration that has a mean squared error that approaches zero whilst also having the correct highest Q-values for each of the eleven relevant states. This means that for each of these experiments, depending on the γ and the correction factor, there exists a set of angles θ for which our parameterized quantum circuit model can accurately represent the Q-values. Meaning that the agent can successfully navigate FrozenLake. The most optimal configuration that we have discovered uses 2 amplitudes per action, a correction factor of 3 and a γ of 0.8 (not coincidentally used in Figures: 11 and 12).

There are a few different patterns that emerge from these plots. Firstly, we notice that a higher γ shifts the mean squared error functions slightly to the right, as the Q-values themselves scale with γ. (see 5.4) Secondly, we notice that number of states for which the correct action has the highest Q-value tends to decrease as γ increases, it also seems to be a very noisy plot in general. Both can

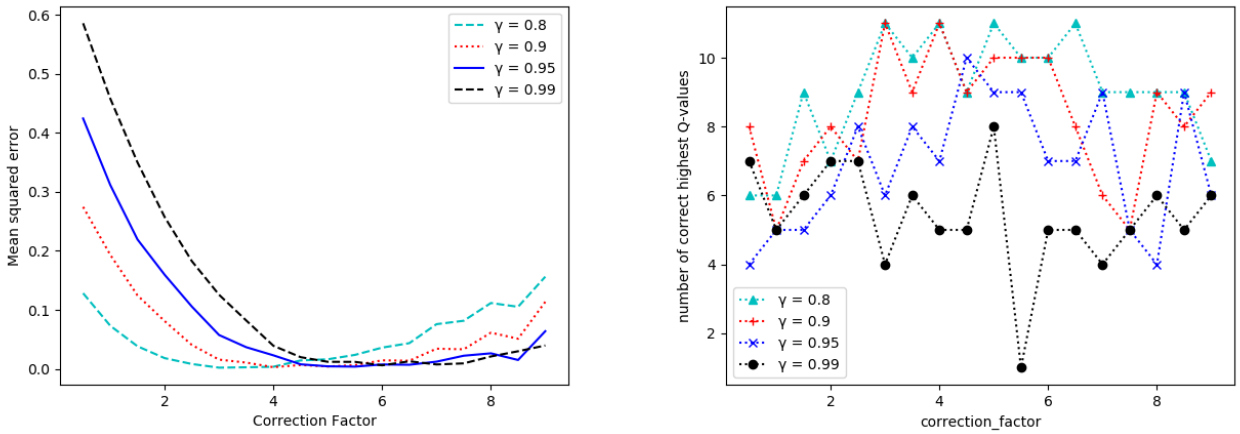


Figure 15: On the left: the mean squared error between the theoretical Q-table and the Q-table if it were computed by the model, as a function of the correction factor that was used. On the right: the number of states for which the highest Q-values in the theoretical and computed Q-tables match, as a function of the correction factor that was used. These were computed when using 2 amplitudes per action.

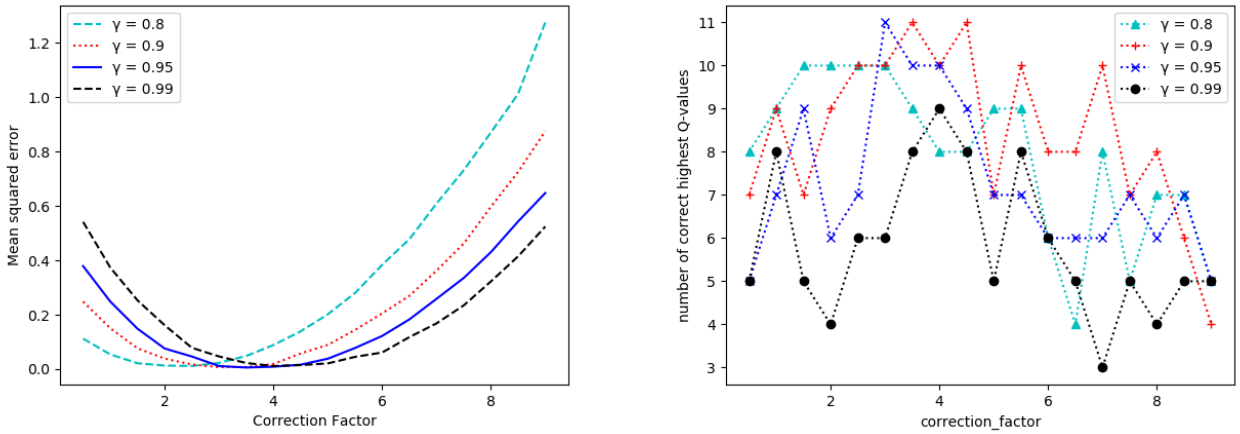


Figure 16: On the left: the mean squared error between the theoretical Q-table and the Q-table computed by the parameterized quantum circuit, as a function of the correction factor that was used. On the right: the number of states for which the highest Q-value computed by our model and the highest theoretical Q-value correspond to the same action, as a function of the correction factor that was used. These were computed when using 3 amplitudes per action.

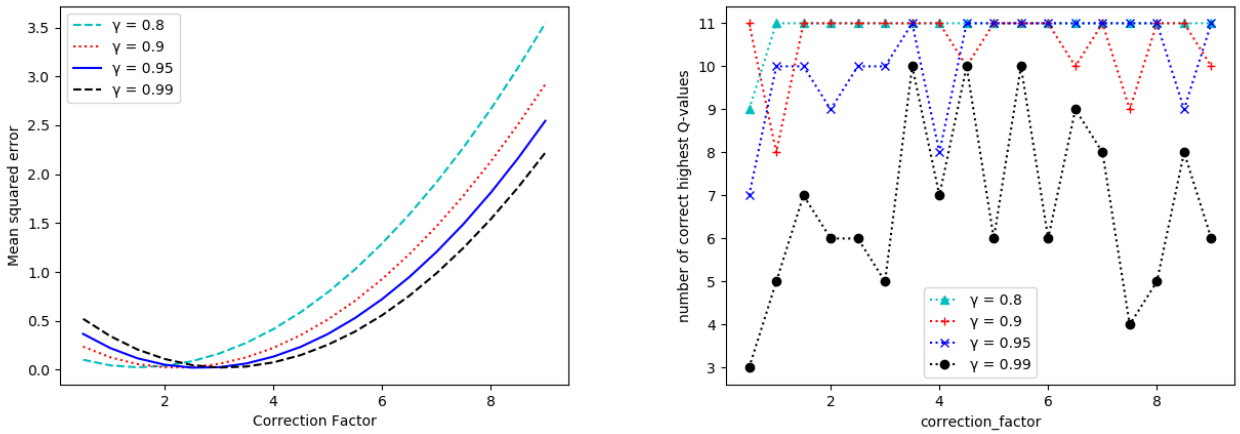


Figure 17: On the left: the mean squared error between the theoretical Q-table and the Q-table computed by the parameterized quantum circuit, as a function of the correction factor that was used. On the right: the number of states for which the highest Q-value computed by our model and the highest theoretical Q-value correspond to the same action, as a function of the correction factor that was used. These were computed when using 4 amplitudes per action.

be explained by the fact that the difference in Q-values for most actions in the same state tends to be small, this is amplified by larger γ as they are closer to 1. Smaller differences between Q-values are harder to train on, thus resulting in more errors when γ increases.. Lastly, we notice that all of the mean squared error plots have a (local) minimum somewhere in this range. We also notice that the right side of the curve flattens when fewer amplitudes are used. This is a side effect of the fact that we use the amplitudes of quantum states to compute Q-values. If we use all amplitudes to compute Q-values that would result in the sum of all Q-values always being constant and scaling with the correction factor. This means that lowering one Q-value inadvertently increases another and vice versa, when considering all amplitudes. The more amplitudes are ignored, the less this effect applies, as more 'excess Q-value' can be transferred to the unused amplitudes. Which is also why only the right part of the graph flattens, as we cannot increase the total Q-value increasing the correction factor.

6.2 From Supervised Learning to Reinforcement Learning

In this section we will try to determine whether or not the model is stable enough to stay in a (near)-optimal configuration once it is reached. To test this we will perform an experiment for which we let the agent start with the model that has the same parameters as the final model after the optimisation from Section 6.1.1, Figure 6.1. We will also use that model as a fixed target model when applying Bellman updates to the policy. The results of this experiment are shown in figure 18.

When we start our circuit in a near optimal configuration, and keep the corresponding Q-function as the fixed target, it seems like the circuit still diverges a little from the near optimal starting con-

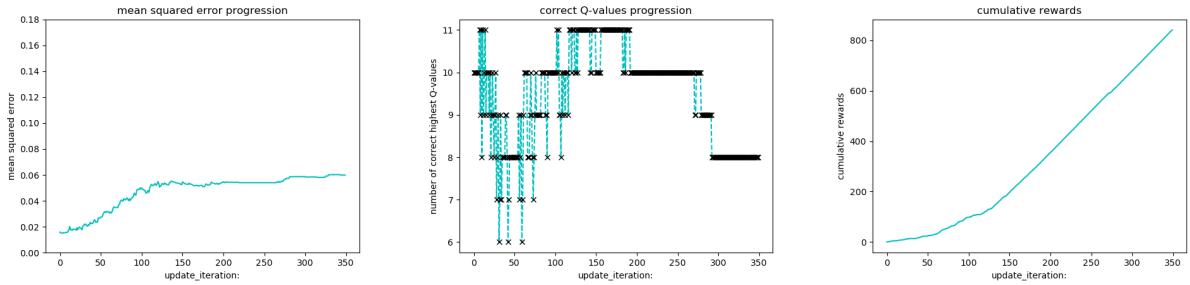


Figure 18: These results were obtained using a fixed target acquired from Figure 6.1.1 as well as starting in that target. On the left: the progression of the mean squared error between the theoretical Q-table and the Q-table computed by the parameterized quantum circuit, after each parameter update. In the center: the number of states for which the highest Q-values computed by our model and the highest theoretical Q-values correspond to the same action(s). On the right: the cumulative reward over all episodes. All three use these model parameters: amplitudes per action = 2, correction factor = 3, $\gamma = 0.8$

figuration. However, it does seem to stabilize more than a run without the fixed target (see Figure 11).

For the next experiment we still use Bellman updates with the same fixed target Q-function. However, we now initialize the parameters of the circuit randomly to see if the circuits Q-function converges to the near optimal one which use as the target Q-function. The results of this experiment are shown in Figure 19.

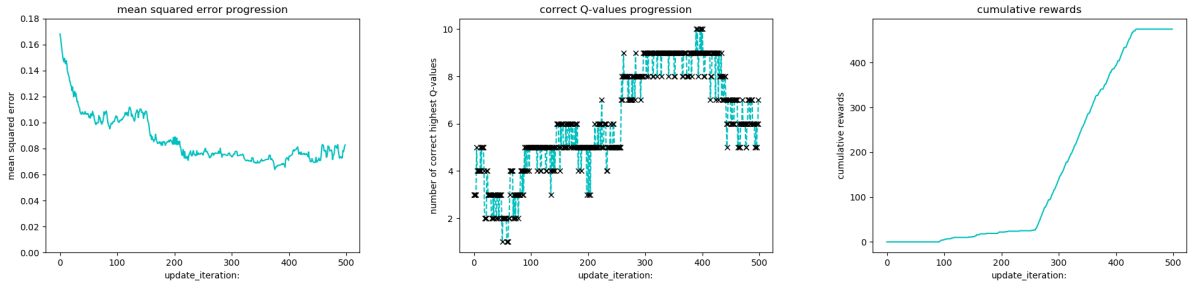


Figure 19: These results were obtained using a fixed target acquired from Figure 6.1.1, but starting with a random policy. On the left: the progression of the mean squared error between the theoretical Q-table and the Q-table computed by the parameterized quantum circuit, after each parameter update. In the center: the number of states for which the highest Q-values computed by our model and the highest theoretical Q-values correspond to the same action(s). On the right: the cumulative reward over all episodes. All three use these model parameters: amplitudes per action = 2, correction factor = 3, $\gamma = 0.8$

From Figure 19 we notice that the Q-function seems to be moving in the right direction. However, it seems to get stuck somewhere in the middle; perhaps there is a local minimum there that traps

our Q-function? Another thing to point out is that the mean squared error of this Q-function seems significantly more stable than the one in Figure 11, meaning that the fixed target does seem to have some stabilizing effect on the training process.

7 Conclusions and Further Research

Given the results of the supervised learning experiments in Section 6.1, we can conclude that the parameterized quantum circuit we used is expressive enough to accurately compute the Q-values for FrozenLake. We have also proven that there are parameter configurations for which it wins the game. Furthermore, we have found the optimal configuration for the newly introduced quantum hyperparameters, namely: correction factor = 3 and amplitudes per action = 2.

From the hybrid supervised-reinforcement learning experiments in Section 6.2, we conclude that fixing the target Q-function does stabilize the model to some extend. But it does not stabilize it enough to improve upon this Q-function found by regressing towards the optimal Q-values. It is also not stable enough to approach the (near) optimal target Q-function when initialized randomly. Therefore the stability of the Q-function and training methods still needs to be improved.

For future research we would want to find ways to stabilize this model further. If we can get the hybrid training techniques from Section 6.2 stabilized and if we can get the randomly initialized circuit to converge to the near optimal parameters using a fixed target, we can start reducing our dependence on this target network and eventually remove it completely. This would move us closer to full-blown reinforcement learning and further away from supervised learning.

References

[1] Ronald de Wolf,
QuSoft, CWI and University of Amsterdam,
Quantum Computing: Lecture Notes,
https://homepages.cwi.nl/~rdewolf/qcnotes.pdf,
19-06-2019.

[2] Michael A. Nielsen & Isaac L. Chuang,
Cambridge University Press,
Quantum Computation and Quantum Information,
http://mmrc.amss.cas.cn/tlb/201702/W020170224608149940643.pdf,
2010.

[3] *Jonathan Hui: QC — Programming with Quantum Gates (Single Qubits)*,
https://medium.com/@jonathan_hui/qc-programming-with-quantum-gates-8996b667d256,
last referenced at 07-07-2020

[4] Richard S. Sutton and Andrew G. Barto,
The MIT Press Cambridge, Massachusetts London, England,
Reinforcement Learning: An Introduction,

http://incompleteideas.net/book/bookdraft2017nov5.pdf,
05-11-2017.

[5] *Microbiome Summer School 2017: Introduction to machine learning*,
https://aldro61.github.io/microbiome-summer-school-2017/sections/basics/,
last referenced at 07-07-2020.

[6] *MC.AI: My notes on Neural Networks*,
https://mc.ai/my-notes-on-neural-networks-2/,
last referenced at 07-07-2020.

[7] *DeepMind: AlphaGo*,
https://deepmind.com/research/case-studies/alphago-the-story-so-far,
last referenced at 07-07-2020.

[8] P. W. Shor,
Proceedings 35th Annual Symposium on Foundations of Computer Science, Santa Fe, NM, USA,
Algorithms for quantum computation: discrete logarithms and factoring,
https://ieeexplore.ieee.org/document/365700,
20-11-1994

[9] Marcello Benedetti, Erika Lloyd, Stefan Sack, and Mattia Fiorentini,
Cambridge Quantum Computing Limited, CB2 1UB Cambridge, United Kingdom 2Department of Computer Science, University College London, WC1E 6BT London, United Kingdom,
Parameterized quantum circuits as machine learning models,
https://arxiv.org/pdf/1906.07682.pdf,
10-10-2019

[10] Samuel Yen-Chi Chen, Chao-Han Huck Yang, Jun Qi, Pin-Yu Chen, Xiaoli Ma and Hsi-Sheng Goan,
National Taiwan University, Georgia Institute of Technology and IBM Research,
VARIATIONAL QUANTUM CIRCUITS FOR DEEP REINFORCEMENT LEARNING,
https://arxiv.org/pdf/1907.00397.pdf,
17-08-2019

[11] *Adesh Gautam: Introduction to Reinforcement Learning (Coding Q-Learning) — Part 3*,
https://medium.com/swlh/introduction-to-reinforcement-learning-coding-q-learning-part-3-
last referenced 07-07-2020

[12] *cirq Python Library*,
https://github.com/quantumlib/Cirq,
last referenced 07-07-2020

[13] *OpenAI Gym Python Library*,
https://gym.openai.com/,
last referenced 07-07-2020

[14] Vojtěch Havlíček, Antonio D. Córcoles, Kristan Temme, Aram W. Harrow, Abhinav Kandala, Jerry M. Chow and Jay M. Gambetta ,
Nature,
supervised learning with quantum-enhanced feature spaces,
https://www.nature.com/articles/s41586-019-0980-2,
13-03-2019

[15] Jin-Guo Liu and Lei Wang,
Chinese Academy of Sciences, Beijing,
Differentiable Learning of Quantum Circuit Born Machine,
https://arxiv.org/pdf/1804.04168.pdf,
11-04-2018