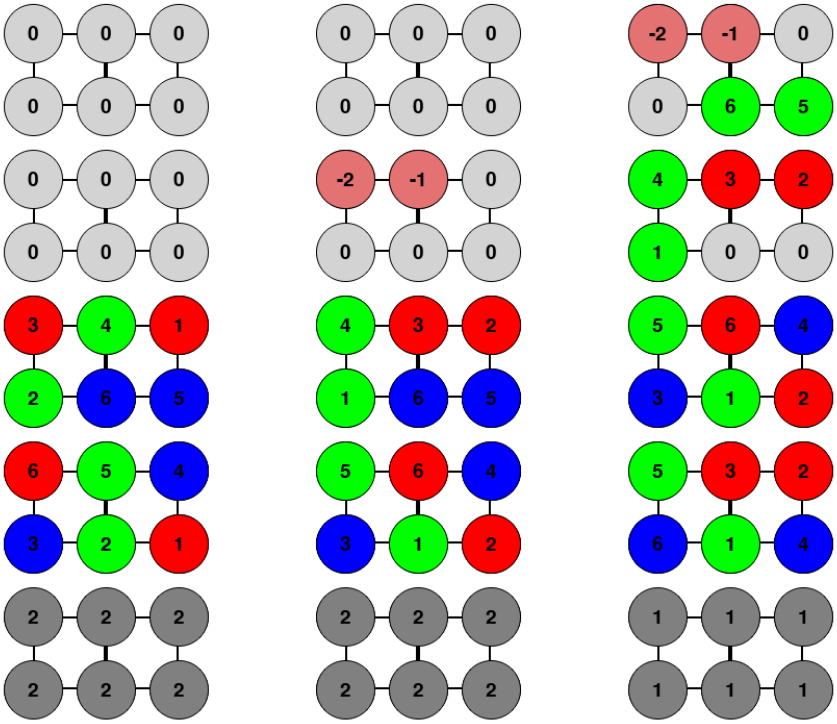
CHALMERS
UNIVERSITY OF TECHNOLOGY



Quantum Routing using Value-Based Reinforcement Learning

Master's thesis in Physics

MIKKEL OPPERUD

Department of Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023

Master's thesis 2023:06

Quantum Routing using Value-Based Reinforcement Learning

MIKKEL OPPERUD

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2023

Quantum Routing using Value-Based Reinforcement Learning
MIKKEL OPPERUD

Supervisor: Mats Granath, Physics
Examiner: Mats Granath, Physics

Master's Thesis 2023:06
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2023

Quantum Routing using Value-Based Reinforcement Learning
MIKKEL OPPERUD
Department of Physics
Chalmers University of Technology

Abstract

This thesis addresses the Quantum routing problem through the implementation of a reinforcement learning algorithm. Quantum routing is the problem of making quantum circuits executable on a quantum computer with limits of connectivity which requires requires swapping information between qubits. A value-based variant of the Q-learning algorithm, coupled with deep convolutional neural networks, was employed to optimize the routing process in a grid topology environment. The environment allowed the agent to place and remove swaps and to "pull back" any immediately executable qubits. The reward scheme was designed to optimize for a shortened circuit depth with the first layers of swaps not counted, thus solving the Quantum routing and placement problem concurrently. The study focused on smaller grid sizes of 3x2, 3x3, and 3x4. Due to time constraints we were not fully able to adequately access the performance of the model and were only able to achieve solutions for smaller models, while the results for the larger ones (3x3 and 4x3) were lackluster. For larger grid sizes our analysis on multiple hyper-parameters revealed a better understanding for the reasons for this, suggesting possible remedies. In conclusion, while the algorithm encountered issues during the experiment, these obstacles present opportunities for future improvement and refinement. This research provides a foundation for future studies in the realm of Quantum routing, highlighting potential avenues for enhanced algorithm performance.

Keywords: Quantum Routing, Q-Learning, Reinforcement Learning, Quantum Placement, Deep Convolutional Neural Networks, Grid Topology Environment, Qubits, Agent, Concurrency, Quantum Circuit Depth.

Acknowledgements

I would like to thank my masters thesis supervisor, Prof. Mats Granath, for his contributions, interests and support.

I owe special thanks to my family for their unending support and faith in me. I would not be able to achieve going through this process without my father, my mother and my brother. I have always felt the support of my grandparents and my cousins, even though we are hundreds of kilometers apart.

Lastly, I also owe special thanks to the computational resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC) at Chalmers Centre for Computational Science and Engineering (C3SE) partially funded by the Swedish Research Council through grant agreements no. 2022-06725 and no. 2018-05973. Without these resources this project would have been exceedingly difficult.

Mikkel Opperud, Gothenburg, June 2023

Contents

List of Figures

1
Introduction

In this chapter we start by providing a concise history and introduction to the concept of quantum computing. Next, we delve into the necessity for quantum routing, followed by a brief summary of the existing work in this field and an explanation of our proposed approach. Finally, we offer a more in-depth introduction to the quantum routing problem.

1.1 An Introduction: On Quantum Computing and The Need for Compiler

Quantum computation emerged from the pioneering idea proposed by physicist Paul Benioff, which combined the foundational elements of a Turing machine with quantum mechanical principles [5]. This innovative model aimed to simulate quantum mechanical systems, a potential that was later underscored by Feynman [8]. David Deutsch substantiated the superiority of this computing model in his 1985 study [1]. However, it was Peter Shor's groundbreaking article introducing a quantum algorithm for prime factorization – an algorithm exponentially swifter than classical counterparts – that truly catalyzed the field [23].
As quantum technology progressed, we entered the 'NISQ era' (Noisy Intermediate-Scale Quantum) as coined by John Preskill in 2018 [19]. This era underscores the inherent 'noisiness' of quantum computers, emphasizing that computations must be constrained in duration to prevent them from being disrupted by noise. The longer a quantum computation persists, the more susceptible it becomes to errors introduced by this noise. Consequently, there is an inherent limit to how extensive these computations can be, making optimization paramount. Major players, such as Google and IBM, have been working diligently to create quantum devices that navigate the challenges of this era. One pivotal element in this journey is the development of optimal compilers. Prominent compilers, like Cambridge's Tket [27] and IBM's qiskit compilers, streamline quantum computations into two primary stages: initial circuit optimization and the subsequent routing problem. This thesis zeroes in on the intricacies and solutions to the latter.
Over the recent past, diverse strategies have been advanced to address the qubit routing problem in quantum computing. IBM's qiskit compiler, for instance, employs the A* algorithm as elaborated in the work by A. Zuhlener [31]. On the other hand, Cambridge's t-ket compiler relies on a non-optimal brute force algorithm based on an approximate distance metric, a concept detailed in A. Cowtan et al.'s paper [7]. Moving away from conventional methods, G. Nannicini et al. [16] introduced an

integer programming technique, a foundation built upon by F. Wagner et al. [29] who explored token swapping, while A. Bapat et al. [4] ventured into fast reversals. In a unique departure from traditional algorithms, there have been strides in applying reinforcement learning to this issue. Reinfocement learning has seen a lot of success in various tasks from playing Atari Games from only the pixel information and score [15], mastering the Game of Go [24], and for robotics such as training a bipedal robot to perform various soccer skills [9]. Lastly it has also been used with some success for Quantum error correction [2]. Recently this method has been used with some sucess on qubit routing such as "Using Reinforcement Learning to Perform Qubit Routing in Quantum Compiler" by M. Pozzi et al. [18], and a study by A. Sinha et al.[26], have brought forward approaches leveraging graph neural networks in conjunction with Monte Carlo treesearch.
This thesis leans predominantly on the aforementioned reinforcement learning approach, building on insights from a preceding master's study. We endeavor to adopt a value-based strategy, a divergence from the prevalent q-learning method. The overarching objective is to mitigate the complexity intrinsic to neural networks, aspiring to achieve convergence for architectures accommodating a larger qubit count than what current methodologies permit.

1.2 Overview of the Problem

The job of the Quantum Compiler can be brought into four steps, that effectively are only two. Fistly the gates of the quantum code needs to be converted into the gates for the quantum hardware, after that is done comes the quantum circuit optimization problem, where these gates are reduced by use of different operation relations. Then the 'logical' quantum bits of the 'code' circuit must be placed on the physical circuit. This circuit does not have full connectivety, meaning that not all qubits can interact with each other. Circuits are usually placed in a grid like pattern, where they can interact with their neighbours. Thus after the placement these qubits must then be transported in such a way as to be able to interact with the correct qubits by means of swapgates. This is known as the routing problem.

2

Literature Review

This chapter presents the previous work conducted in the literature on approaches to the Quantum Routing Problem and Enhancements to Reinforcement Learning Approaches.

2.1 Overview

The routing problem in quantum computing has been an area of active research for the past few years. The task involves finding the optimal sequence of qubit operations, given the constraints imposed by the quantum hardware architecture.

2.2 Early Approaches to Quantum Routing:

Quantum routing problem started by introducing the quantum logic to network architectures to decrease latency in data transfer. A routing algorithm was first introduced by Shamsa et al. [22], on a hypercube network topology based on Quantum Dot Cells to overcome the latency issue in data transfer. They tried perform parallel computation using Quantum Dot Architecture to decrease latency and find shortest path.
Another trend in early approaches to Quantum Rounting was to identify the problems which classically took exponential time but from a quantum perspective took less time, i.e. polynomalial time. Motivated by this trend, Kempe [11] discusses continuous and discrete quantum walks and proposes a packet routing algorithm.

2.3 Classical approaches to the quantum routing problem

The t|ket> compiler, as described by Cowtan et al. [7], utilizes a hardware-agnostic, four-step routing procedure for quantum circuits. The process begins by decomposing the circuit into timesteps of simultaneously executable gates. An initial mapping of logical to physical qubits is then established, creating a graph to dictate placement based on interactions. The routing algorithm then iteratively builds a new circuit compatible with the target architecture, using a minimal SWAP strategy to manage qubits required for each gate. Finally, any non-native SWAP operations are replaced with hardware-specific gates, and a clean-up phase removes extraneous

gates introduced during the synthesis. In addition, there have been multiple studies [14]-[17] on the application of switching algorithms on quantum systems to propose a congestion-free self-routing algorithm for quantum packet transfer.

2.4 Deep Reinforcement Learning for Quantum Routing

One of the early works by Herbert et al. [10] introduced the concept of using reinforcement learning to solve the routing problem. The authors proposed an algorithm that traverses the circuit layer by layer, learning a state-value function that determines the quality of SWAP insertions before the next layer. The optimal placement is then calculated using simulated annealing. This work provided a proof of principle for using reinforcement learning in qubit routing, paving the way for subsequent research in this area.
Following the initial success of reinforcement learning in Quantum Routing, van de Griend [13] considered the compilation of hardware incompatible CNOT gates into a minimal sequence of compatible CNOT gates based on restricted Gaussian Elimination. The optimal sequence of steps in the elimination procedure was found by a reinforcement-learning agent, demonstrating the potential for deep reinforcement learning in quantum circuit synthesis.
In a more recent study, Pozzi et al. [18] sought to improve on the work of Herbert et al. by inserting SWAP gates within a layer rather than before a layer. They also introduced a more compact state representation and learned a (state, next state) value function with double Q-learning. The authors reported an improvement on state-of-the-art benchmarks, showcasing the potential for continual enhancement of reinforcement learning methods for Quantum Routing.
In the next section, we discuss the theoretical background that underpins these research efforts, including an introduction to quantum computing, reinforcement learning, and their applications to the routing problem.

2.5 Monte Carlo Tree Search for Quantum Routing:

In a novel approach, Zhou et al. [30] utilized Monte Carlo Tree Search (MCTS) for qubit routing. At each layer, the search over the tree corresponds to the search for suitable SWAP gates. The simulation step is based on a heuristic that brings the consecutive interacting qubits closer together. This approach, leveraging the strengths of MCTS, presented a new direction for solving the routing problem.
Building on this idea, Sinha et al. [26] also employed MCTS for qubit routing. In their approach, the tree corresponds to establishing a set of SWAP gates for the next layer. Traversing the tree corresponds to either committing the set or extending it by further SWAPs. The researchers integrated Q-learning on the tree and a graph neural network to estimate the value of SWAP-gate sets in the simulation step, combining reinforcement learning and MCTS in a novel way.

3
Theory

In the following sections, theoretical background on convolutional neural networks, value based Q-learning and quantum computing are introduced. We start by giving a brief overview of quantum computing. Here not everything is needed to understand the problem, but the main points to take away is understanding of the structure of quantum circuits and our approximate way of calculating the computational time of the circuits.
Next we move on to introducing the problem in the section "The Qubit Routing Problem". Moving on we talk about the techniques utilized to solve the problem, talking about reinforcement learning, Q-learning, value-based Q-learning and deep Q-learning. Then talking in detail about how neural networks work, both fully connected and convolutional neural nets.

3.1 Quantum Computing

In this section we briefly go over the fundamentals of quantum information before describing quantum circuits, and then the quantum routing problem.
We begin by describing how quantum information works, how it is built up by qubits, how these qubits can change state by applying gates, and interact through two qubit gates. Then more importantly how these gates come together to form circuits, which then leads us to discuss physical gate connectivity and how that leads to the quantum routing problem, which then is explained in detail.

3.1.1 Quantum Information

In quantum computing, the fundamental unit of information is the quantum bit, or qubit. Unlike classical bits that can hold a value of either 0 or 1, a qubit can hold a superposition of states. The state of a qubit is typically represented as a two dimensional complex vector of length 1, which in dirac notation becomes:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{3.1}$$

Here, $|0\rangle$ and $|1\rangle$ are the basis states and $\alpha, \beta \in \mathbb{C}$ are numbers that represent the probabilities of measuring each of the states through the Born rule. According to the Born rule, when measuring this state, the probability of obtaining the result $|0\rangle$ is $|\alpha|^2$ and obtaining the result $|1\rangle$ is $|\beta|^2$. After measuring the qubit collapses to the state that has been measured. The total probability of each of the states

must be one which explains why the length of the state (or vector) must be one ($\langle\psi|\psi\rangle = \alpha^2 + \beta^2 = 1$).
This can be extended to measurements in different basis states. If we consider the basis $\{|+\rangle, |-\rangle\}$, where $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, measuring a qubit in this basis will collapse the qubit state to either $|+\rangle$ or $|-\rangle$. The probability is determined by projecting $|\psi\rangle$ onto the chosen basis states.

$$P_{\pm} = |\langle\pm|\psi\rangle|^2 \tag{3.2}$$

For multi qubit states the same thing applies, but with more basis states, where each added qubit doubles the number of states. Mathematically these states are described by the tensor product of their individual qubit states states. Explicitly for two qubits $|\psi_1\rangle$ and $|\psi_2\rangle$, the combined state $|\psi\rangle$ is represented as:

$$|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \tag{3.3}$$

This is known as a product state where the individual states are not connected. In this way we can produce twice as many basis states for each added qubit. By combining these product states we can get states that are entangled, meaning that their states are intertwined in such a way that the state of one qubit cannot be described independently of the other.
An example of an entangled state is the Bell state, given by:

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{3.4}$$

In this state, if the first qubit is measured and found to be in state $|0\rangle$, the second qubit is instantaneously found to be in the same state, regardless of the distance between them.

3.1.2 Quantum Gates and Circuits

Now that we understand how the underlying information is represented we move on to the actual operations on the quantum bits, namely the quantum gates, which is what is interesting for this study. These gates are the building blocks of quantum circuits and perform operations on qubits. Quantum gates are represented as unitary matrices. For a single qubit, common quantum gates include the Pauli-X, Pauli-Y, Pauli-Z (the spin matricies) and Hadamard gates. Each of these gates performs a specific operation on a qubit, changing its state. These maintain the length or 'probability' of the qubit meaning that they are Unitary, which is a requirement for any quantum operation/gate. The operations are shown below:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \qquad H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Quantum gates can also operate on multiple qubits. Notably, the CNOT (Controlled-NOT) gate operates on two qubits: a control qubit and a target qubit. If the control qubit is in state $|1\rangle$, it applies a NOT operation (or a Pauli-X gate) on the target qubit. Another important two-qubit gate is the SWAP gate, which exchanges the states of the two qubits, which is shown in the figure below 3.1. As we will see later this two bit gate will be the most important for this study.

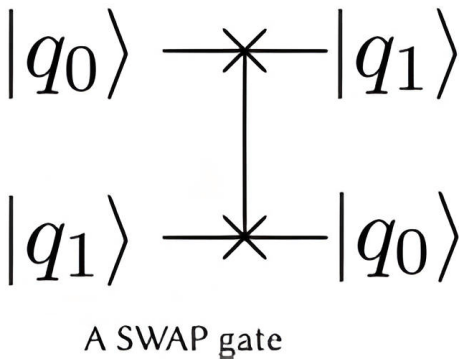


Figure 3.1: Shows a swap gate and how it switches the information of the qubits q_0 and q_1

These quantum gates can in turn be used together to build circuits that form more complex quantum operations. These complex quantum operations can always be broken down into one and two qubit gates, given that one has the right set of these gates (universal quantum gate set). Together with the fact that quantum computers of today nearly exclusively deal with one and two qubit gates, means that we will only look at those gates.
The circuit can then in turn be divided into layers of the two qubit gates that can be performed in parallel. This gives a good estimate of how long it will take to execute the gate, given that the two qubit gates take significantly longer to perform than one qubit gates. These layers are what will be used later to estimate the performance of the routing, and is what in this project will be optimized for. This (execution time) along with the fact that one qubit gates do not have any connectivity issues (see later), means that the one qubit gates can be effectively ignored.

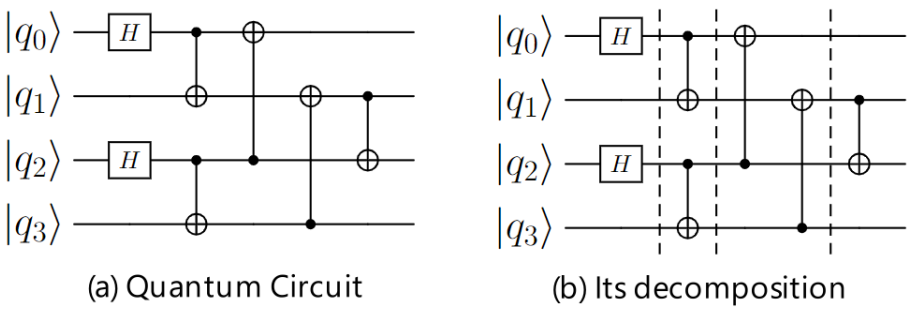


Figure 3.2: Shows a 4 qubit circuit with two hardamand gates and how it is decomposed into layers based on the two qubit gates that can be computed in parallell

A simple example of a four qubit quantum circuit can be seen in fig:3.2, which uses hardamard gates and CNOT gates. In figure (b) we can also see how it is is decomposed into layers based on the two qubit gates that can be performed in parallell. In this case we have three layers of two qubit gates.

These quantum circuits do not directly correspond to quantum hardware in two respects. Firstly, the set of gates available on the hardware will usually not correspond to the gates of the quantum code. These will thus be translated into sequences of gates available on the actual hardware. This process is known as quantum gate synthesis. Then these gates will be compressed, where series of gates will be translated either into the identity or into fewer gates, in a process called quantum circuit optimization.

3.1.3 The Qubit Routing Problem

Moving on we look closer at the qubit routing problem. As stated previously the main issue is with the qubit connectivity of the quantum Hardware. Specifically, for regular qubit gates any qubit n can interact with any qubit m. In today's Quantum Computers, this is not possible. Here there are physical constraints imposed on the circuit that only allows adjacent qubits to interact with each other. There are different connectivities that the quantum computer can have.
Quantum computers today predominantly utilize a grid-based layout for their qubits, where two-qubit operations can only occur between neighboring qubits, such as those in IBM systems, Google's quantum computer, and local developments at Chalmers. A visual representation of this topology can be seen in fig 3.3.

Here the qubits are represented by dots, with the connecting lines symbolizing potential interactions. These connections primarily span horizontally and vertically (black lines), though diagonal interactions (gray lines) may also occur. The focus of our study will be on the standard grid-like structures (only black lines) currently in development at Chalmers. The figure itself shows one qubit gate being composed of two qubits (marked by red) being routed by means of swap gates in two ways. The first, solution (a) has two overlapping swap-gates placed between qubit $q2$ and $q4$ and then between $q4$ and $q6$, while solution (b) performs two swaps that can be done in parallel. Swapping $q2$ and $q4$ and simultaneously $q6$ and $q8$. This illustrates in a simple way the routing problem, and the way different solutions lead to different results. For the full problem one not only have to take into account multiple gates at once for each layer, but also one has to think about swapping the qubits in the layers behind as well. These added complexities is what leads to finding an optimal solution to the problem being NP-complete as proven in [10], in turn meaning that to solve this problem practically we need good approximate solutions.

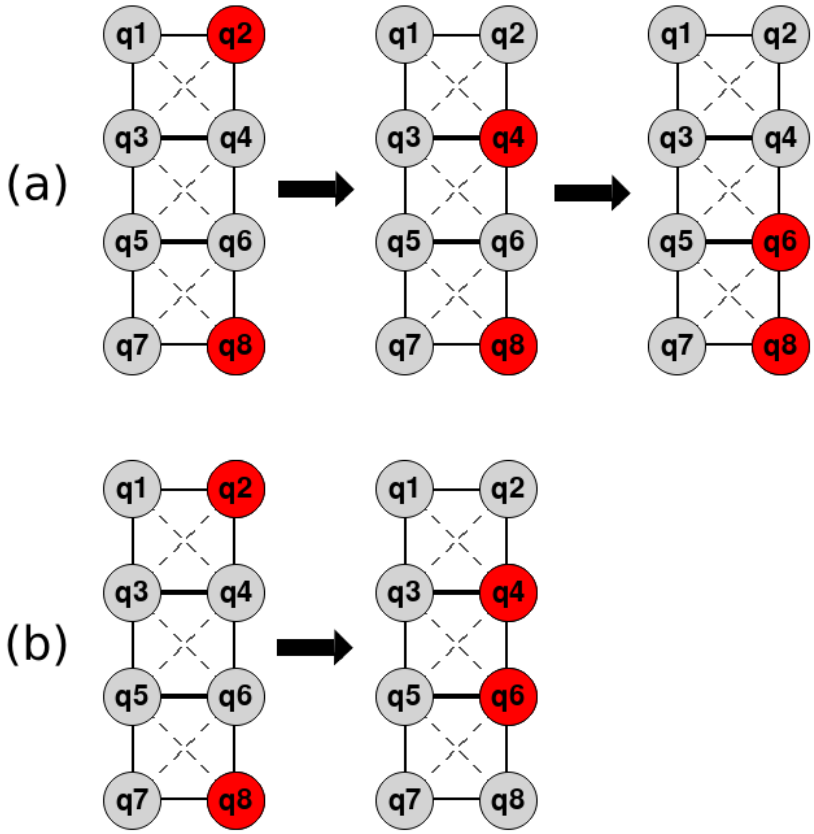


Figure 3.3: Qubit positioning in a 4x2 quantum computer. Highlighted are qubits at positions (1,2) (q_2) and (4,2) (q_8) needing interaction via a gate, necessitating relocation through swap operations. Two approaches are illustrated. The first, shown by the gray arrow, swaps q_2 with q_4 and subsequently q_4 with q_6, enabling interaction. The second, optimal approach concurrently swaps q_2 with q_4 and q_6 with q_8, illustrated by the black arrows.

3.2 Reinforcement Learning

Moving on we describe reinforcement learning. Starting off with the basic theory coming from Markov Descicion Processes (MDP's) and how they are solved with different algorithms. Continuing by describing the Q-learning algorithm followed by the value based version of this. Finally ending with how to use these in conjunction with neural networks, which gets us the deep Q-learning algorithm (DQN).

3.2.1 Introduction to Reinforcement Learning as a Markov Decision Process

Reinforcement learning (RL) is a paradigm of machine learning where an agent learns to make decisions by interacting with an environment. At the heart of RL is an agent that learns to make decisions based on interactions with its environment.

In the context of the Qubit Routing problem, our agent will be executing swaps in the quantum circuit, which serves as its environment. RL is particularly useful in situations where there is no correct answer readily available but instead, the optimal strategy is discovered through the process of exploration and exploitation over time. It has proven itself useful in a wide variety of aplications, from mastering strategic games such as chess and Go [24, 25], autonomous vehicles [3], to robotics [9] and quantum error correction [2].

More formally, the RL problem is an instance of the Markov Decision Process (MDP), a mathematical framework typically denoted by a tuple (S, A, P, R, γ), that provides a precise language for stating and solving problems of stochastic control. The components of this tuple are:

S: The state space, which represents the set of all possible states the environment can be in. In our case, each state would correspond to a specific configuration of the quantum circuit. A: The action space, encompassing all the possible actions the agent can take. For the Qubit Routing problem, an action is a swap operation on the quantum circuit. P: The state transition probability matrix. It characterizes the dynamics of the environment by defining the probability of transitioning to each possible next state, given the current state and action, often denoted as $P(s_{t+1}|s_t, a)$. R: The reward function, denoted as $R(s_{t+1}, a, s_t)$, signifies the expected reward the agent will receive after performing a specific action in a particular state and transitioning to the next state. γ: The discount factor. A number between 0 and 1, it determines the present value of future rewards.
It is important to note that both the transition probability and reward function only depend on the current and next state and has no memory of previous states or actions.
The agent's objective in an MDP is to find a policy, which is a mapping from states to actions, that maximizes the expected cumulative discounted reward over time. This is often written as:

$$\max_{\pi} E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \middle| \pi\right] \tag{3.5}$$

Reinforcement learning is a specific method for solving this problem that uses machine learning, and that learns the policy by trial and error through exploring the environment in a 'action-reward feedback loop'. This loop is illustrated in figure 3.4. Here agent-environment interaction occurs in successive steps, providing the cyclical nature of reinforcement learning. At each time-step t, the agent receives the current state s_t of the environment and chooses an action a_t based on its current policy. After the agent takes the action, the environment transitions to a new state s_{t+1} and provides a reward r_{t+1} to the agent. The agent's policy then gets updated based on the received reward, and the next iteration of the interaction begins. This iterative process of the agent interacting with the environment, learning from the reward, and improving its policy forms the core of reinforcement learning.

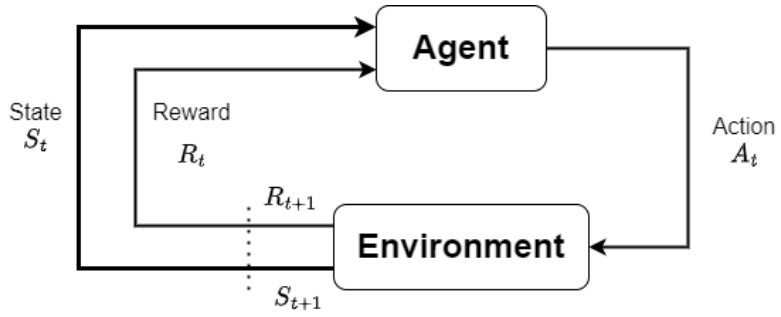


Figure 3.4: The agent-environment interaction in a Markov Decision Process, known as the Action-Reward Feedback Loop in Reinforcement Learning. At each time step t, the agent selects an action a_t based on its current knowledge and the received state s_t. This action is then passed to the environment. In response, the environment transitions to a new state s_{t+1} and provides the agent with a reward R_{t+1}, which the agent then uses to learn from the environment thus creating the "Action-Reward Feedback Loop".

Among the variety of RL algorithms, here are a few prominent ones as talked about in [28] and presented in [21]:

Value Iteration and **Policy Iteration** are classical methods that solve for the optimal policy in a tabular setting, where states and actions are discrete and the entire model of the environment is known.

Q-Learning is an off-policy algorithm that learns an action-value function, which provides the expected return for each action in each state. It can handle environments with discrete states and actions.

Deep Q-Networks (DQN) extend Q-learning to environments with high-dimensional state spaces by using deep neural networks as function approximators.

Policy Gradient Methods directly optimize the policy in the direction of increasing return. **Actor-Critic Methods** combine the benefits of value-based methods and policy-based methods.

Proximal Policy Optimization (PPO) is a policy optimization method that achieves good performance in a wide range of tasks and is relatively easy to implement.

Each of these algorithms uses different techniques and ideas to learn the optimal policy, but they all rely on the core principle of maximizing the cumulative reward.

Despite its potential, RL is known to be sample inefficient and sensitive to hyperparameters, but ongoing research is tackling these issues. The beauty of this framework however lies in its generality, making it a potent tool for a wide range of sequential decision-making problems, including the Qubit Routing problem. It allows us to not

only systematically explore the solution space but also to exploit learned knowledge for more efficient problem-solving.

3.2.2 Q-Learning

Q-learning is an off policy algorithm, off policy meaning that it uses an other policy to learn then it uses to solve the problem. The central component in Q-Learning is the Q-function, denoted as $Q(s, a)$. This function gives the expected return or the cumulative discounted future reward, for taking action a in state s and following policy π thereafter. It is defined more precisely as:

$$Q(s, a) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \middle| s_0 = s, a_0 = a \right] \tag{3.6}$$

The policy here is the optimal policy defined in the subsection about Markov Decision Processes.
The goal of Q-Learning is to find the optimal Q-function, $Q^*(s, a)$, which yields the maximum expected return achievable by following any strategy, starting from state s, taking action a, and thereafter following the optimal policy.
Writing out this expected reward given the policy and the transition probability we get the Bellman equation for the Q-function:

$$Q_\pi(s, a) = \sum_{s'} p(s'|s, a)[R(s, a, s') + \gamma Q_\pi(s')] \tag{3.7}$$

If we set $\pi(a|s) = \max_a Q(s, a)$ and have a deterministic transition matrix $p(s'|s, a) = \delta_{s_a s'}$, we get a simpler expression for the Q-function:

$$Q^*(s, a) = R(s, a, s') + \gamma \max_{a'} Q^*_(s', a') \tag{3.8}$$

In order to obtain the optimal Q-function we then iteratively update the Q-values using the Temporal Difference (TD) learning. Where we update the Q-value from reward received r and the q-values from the next state s'. We have the one step Q-learning algorithm:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right], \tag{3.9}$$

This algorithm is has been proven to be guaranteed to converge to the optimal Q-value $Q \to Q^*$,where α is the update rate, a tunable parameter often set between 0 and 1

In the equation above, $r + \gamma \max_{a'} Q(s', a')$ is the learned value or updated Q-value. The difference between this learned value and the old value, $Q(s, a)$, is the Temporal Difference (TD) error, denoted as δ. The Q-value is updated by moving a fraction α of the way towards the learned value.

In classical Q-learning, a table known as the Q-table is used to store the Q-values for each state-action pair. This table keeps track of the learned values and is updated

through iterative exploration of the environment and application of the Bellman equation.

In this exploration phase, an ε-greedy strategy is typically utilized to balance exploration and exploitation. This involves a trade-off: with probability ε, the algorithm explores by selecting an action randomly, and with probability $1 - \varepsilon$, it exploits by choosing the action with the highest Q-value in the current state. This policy encourages the agent to try various actions while still mostly choosing what it currently thinks is the best action.

However, for larger environments, where the number of state-action pairs can be prohibitively large, using a Q-table may not be feasible due to memory constraints. In these scenarios, an approximation of the Q-function becomes necessary. One popular approach is the use of neural networks to approximate the Q-function, a method known as Deep Q-Learning. The neural network essentially learns to predict the Q-values given a state-action pair, thus providing a scalable and efficient alternative for large problem spaces. We will delve deeper into this technique in an upcoming section on "Deep Q-Learning".

3.2.3 Value-Based Learning

Value-Based Learning forms a group of reinforcement learning strategies focused on determining the optimal value function. Derived from this, the optimal policy can be identified. Contrasting with policy-based methods, which directly seek to learn the policy, value-based methods indirectly encode the policy within the value function. Consequently, the policy is expressed by always opting for the action that maximizes the current estimate of the value function.
Comparing this to Q-learning we can express this value function given the state $V(s)$ in terms of the Q-value state action pair $Q(s, a)$ as:

$$V(s) = \max_a Q(s, a), \tag{3.10}$$

Here, $V(s)$ represents the value of state s under an optimal policy, and $Q(s, a)$ is the maximum action-value function.
When applying the concept of temporal difference learning to value-based methods, the update rule changes slightly. It takes the form of:

$$V(s) \leftarrow V(s) + \alpha \left[r + \gamma \max_{s'} V(s') - V(s) \right], \tag{3.11}$$

where:
s is the current state, r is the reward received after transitioning to the new state, s' is the new state, α is the step-size parameter, and γ is the discount factor. This approach to learning can often provide more stability, as it doesn't require the maintenance of a separate policy which might be subject to oscillations or instability. Instead, it optimizes the value function, allowing the policy to emerge implicitly.

3.2.4 Deep Q-Learning

Deep Q-Learning, or Deep Value-Based Learning, is an extension of the traditional Q-Learning algorithm. The term "Deep" in Deep Q-Learning refers to the use of deep neural networks as function approximates to estimate the Q-function, $Q(s, a)$. As previously noted, in standard Q-Learning we maintain a table of Q-values for each state-action pair. However, in environments with large state-action spaces, this becomes computationally infeasible. Deep Q-Learning addresses this problem by using a neural network, the Q-network, to approximate the Q-function.
The inputs to the Q-network are the state representations, and the network outputs a Q-value for each possible action in the state. During training, we aim to minimize the difference between the predicted Q-values and the "target" Q-values by updating the networks weights. The weights of the neural network are typically updated with input-target pairs (see next section). The network will receive an input, guess an output and then be trained to match the target. In our case this target is specified by the updated value from the temporal difference equation (3.11).
The target values here are calculated by a target network, which is used to update the weights of the policy network that we are using. The target network is then updated to the policy network's weights from time to time. The reason for this is to ensure stability in the learning process, where the network learns to just continually increase the Q-values.

In Deep Q-learning, the algorithm begins by exploring the environment and collecting experiences, guided by the epsilon-greedy policy. These experiences, recorded as tuples in an experience replay buffer, contain crucial information: the current state, the action taken, the received reward, the subsequent state, and whether it's the final state, often denoted as (s, a, r, s'). In our value-based approach, we store the state alongside all corresponding actions, rewards, and next states: (s, a_i, r_i, s'_i).
The replay buffer serves two crucial roles. First, by storing and reusing past experiences in the learning process, the replay buffer helps break the correlation between consecutive samples, stabilizing learning. This is significant as neural networks, widely employed in contemporary reinforcement learning algorithms, assume the data samples are independently and identically distributed (i.i.d), an assumption violated when consecutive samples from the environment are considered.
Second, a replay buffer allows the agent to learn from rare but important experiences multiple times, which can greatly speed up the learning process. In practice, the replay buffer operates as a cyclic array: as new state transition tuples are added, the oldest tuples are purged once the buffer reaches capacity and the samples are chosen based on a uniform distribution.

3.3 Neural Networks

In this last section in the theory we describe neural networks, and how they function from quite a mathematically precise standpoint. Describing how they are built up, how the training process works, along with all of the different terms that are later used. We start by describing fully connected neural networks, how they are built

up and trained, before moving on to describing how convolutional neural nets are built up.

3.3.1 Fully Connected Neural Networks

An artificial neural network is a non linear function $\mathbf{NN}(\mathbf{x}|\theta)$ with tunable parameters θ, which is used for pattern-recognition tasks such as image classification, face recognition and in natural language processing among other things.
A base element of a neural net are the so called neurons $\mathbf{n}$ that output a numerical value. These neurons are typically arranged in interconnected layers, where the values of the neurons of one layer $\mathbf{n}^L$ determine the value of the neurons in the next layer $\mathbf{n}^{L+1}$ through some non-linear function $\mathbf{F}^L(\mathbf{x})$. For a fully connected layer neural network this function is given by:

$$\mathbf{n}^{L+1} = \mathbf{F}^L(\mathbf{n}^L) = f\left(W^L \cdot \mathbf{n}^L + \mathbf{b}^L\right). \tag{3.12}$$

Where W^L is a matrix whose elements are called weights, and $\mathbf{b}^L$ are called the biases of the layer L. These are the tunable parameters of the neural network. The function $f : \mathbb{R} \to D \subset \mathbb{R}$ is called activation and the range D reflects the possible output of the neuron. Thus a neural network is a function that maps an an input $\mathbf{x}$ to an output $\mathbf{y}$ through a layered set of functions:

$$\mathbf{NN}(\mathbf{x}|\theta) = \mathbf{F}^L(\mathbf{F}^{L-1}(... \mathbf{F}^1(\mathbf{x}) ...)). \tag{3.13}$$

This can be visualized by the figure 3.5. All the weights $\mathbf{w}$ and biases $\mathbf{b}$ that describe the neural net should be chosen in a way to produce a meaningful output. The goal of the neural net is to tune the parameters θ in such a way as to given a set of inputs $\mathbf{x}$ reproduce a desired target outputs $\mathbf{y} \to \mathbf{t}$. The parameters in question here is the set of all weights and biases of the network $\theta = \{W^L, b^L \mid \forall L\}$. These input target pairs are organized in a dataset $\mathcal{D} = (\mathbf{x}, \mathbf{t})$. In our case this dataset is composed of the observation of the environment $\mathbf{x}$ for the input and the rewards r_i and next observations (state) s_{i+1} used to calculate the target (the V-values), through the temporal difference equation from the value function section 3.11.

In order to tune the parameters θ one uses a so called loss-function $L(y, t)$ that measures how far away one the output is from the target. Then what one wants to optimize the total loss of the data-set with respect to the parameters θ. This total loss is known as the cost function:

$$C(\theta|\mathcal{D}) = \sum_{(x,t)\in\mathcal{D}} L(\mathbf{NN}(\mathbf{x}|\theta), t) \tag{3.14}$$

This allows one to alter the parameters $\theta = (\mathbf{w}, \mathbf{b})$ in order to minimise the value of C. This parameter optimization is performed via *gradient-descent* by updating according to:

$$\theta_{i+1} = \theta_i - \eta \nabla_\theta C \tag{3.15}$$

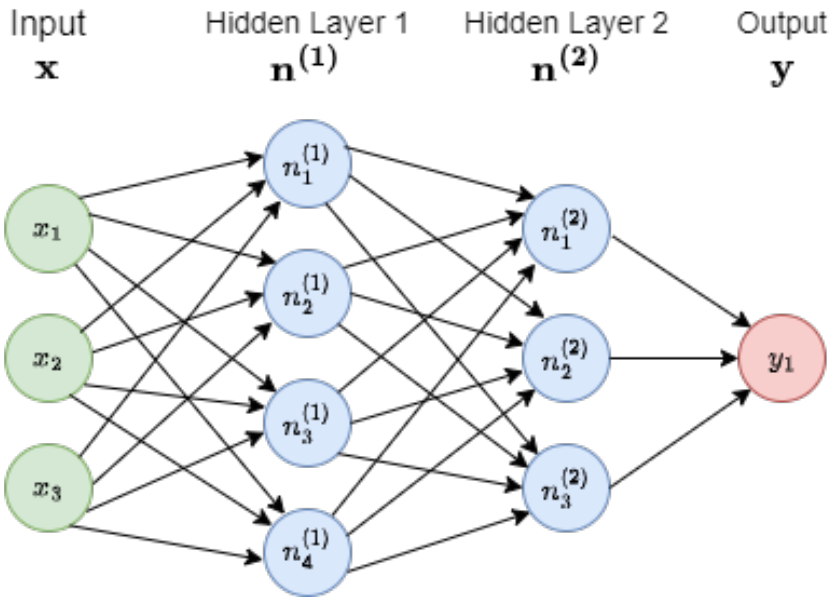


Figure 3.5: The schematic structure of an ANN with three input and one output neuron and two hidden layers. Here the arrows between the neurons of the different layers represent the individual weights of the weight matrices W_{ij}^{L}.

where η is a hyperparameter, called learning rate. This optimization can be done by using the target value pair from the dataset $\mathcal{D}$, which is referred to as *training*.

The underlying algorithm utilized for computing these gradients is known as backpropagation. Fundamentally rooted in the chain rule of calculus, backpropagation is a procedure that efficiently computes the gradient of the cost function with respect to the weights and biases in the network. The essential feature of backpropagation is that it operates iteratively, updating the network's parameters from the last layers to the first. This recursive application of the chain rule allows for efficient computation of gradients, even in networks with a large number of layers. Consequently, backpropagation forms the backbone of training procedures in deep learning.

In the interest of computational efficiency, the complete dataset $\mathcal{D}$ is typically divided into subsets known as mini-batches, $\mathcal{B}_i \subset \mathcal{D}$. Each of these mini-batches usually contain around 100 instances. Rather than calculating the exact gradient of the cost function over the entire dataset, which can be computationally expensive and time-consuming, these mini-batches are used to estimate the gradient. This approach enables more frequent updates to the model parameters, promoting faster convergence during training. However, as each update is now based on a random subset of the total data rather than the full dataset, the gradient calculations can vary significantly from one mini-batch to another. This can introduce a degree of volatility in the training process, potentially leading to erratic movements in the parameter space and difficulties in achieving stable convergence. While this sounds bad, this actually helps the training process as a whole in that it prevents one from getting stuck in local minima.

To combat these instabilities, optimization algorithms such as the Adam algorithm are typically employed [12]. Adam algorithm incorporates adaptive learning rates and momentum, essentially performing an averaging of previous gradients with exponentially decaying importance. This effectively smooths the gradient descent path, mitigating the volatility caused by mini-batch learning, and assists in stable and efficient convergence towards the optimal solution. This averaged out gradient is computed with:

$$g_{\text{exp-avg}} = \frac{1-\beta_1}{1-\beta_i^t} \sum_{i=0}^{t} g_{t-i}\beta_1^i \tag{3.16}$$

Where $\beta_1 \in [0,1]$ is the parameter of exponential decay. And the factor in front of the sum makes sure that this becomes an average. E.g. $g_{\text{exp-avg}} = 1$ if $g_i = 1 \forall i$.
This is then divided by the length of the gradient averaged in the same way:

$$|g|_{\text{exp-avg}} = \sqrt{\frac{1-\beta_2}{1-\beta_2^t} \sum_i g_{t-i}^2 \beta_2^i} \tag{3.17}$$

Here with $\beta_2 \in [0,1]$ as the exponential decay parameter. The squaring of the gradient is an elementwise Then the parameters θ will be updated according to:

$$\theta_t \leftarrow \theta_{t-1} + \alpha \frac{g_{\text{exp-avg}}}{|g|_{\text{exp-avg}} + \epsilon} \tag{3.18}$$

Here α is the step-size, and ϵ is a small parameter to prevent any infinities. In [12] this the exponentially decaying averages of the gradients are called the first moment vector $m_t = g_{\text{exp-avg}} \cdot (1-\beta_1^t)$ and second moment vector $v_t = [|g|_{\text{exp-avg}}]^2 \cdot (1-\beta_2^t)$, (in-spite of being a scalar) respectively. These are updated each iteration.

3.3.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a special type of neural network, optimized for pattern detection in grid-like topology data, such as images. This is achieved by using a mathematical operation called convolution in place of general matrix multiplication in at least one layer, thus the name convolutional neural network.
The convolution operation involves applying a filter, also known as a kernel, across the input data to produce a feature map. This filter is applied by sliding over the input data in a grid-like manner, such that the output (a 'filtered' version of the input) represents local regions of the input.
If we denote $\mathbf{F}^L$ as the Lth layer of the CNN, and let $\mathbf{K}$ be the kernel, the operation of a convolutional layer can be expressed mathematically as:

$$\mathbf{n}^{L+1} = \mathbf{F}^L(\mathbf{n}^L) = f\left(\mathbf{K} * \mathbf{n}^L + \mathbf{b}^L\right), \tag{3.19}$$

where "$*$" represents the convolution operation, $\mathbf{n}^L$ represents the input to layer L, $\mathbf{b}^L$ is the bias, and f is a non-linear activation function.

Note that unlike in fully connected layers, where each neuron's output is influenced by every neuron in the previous layer, in convolutional layers, neurons are only affected by a local region of neurons in the previous layer. This region is defined by the size of the kernel.
CNNs share the weights of the kernel among all neurons in the same feature map. This makes CNNs translation invariant, meaning they can detect patterns regardless of where they occur in the input space. This structured approach and local connectivity make CNNs particularly suited to tasks such as image and video recognition, where local pixel patterns e.g. edges and textures are significant features.

If we look closer at the convolution operation for a 2D input I of size $m \times n$ and a 2D kernel K of size $a \times b$, then the convolution operation is defined as:

$$(I * K)[i, j] = \sum_{u=0}^{a} \sum_{v=0}^{b} I[i - u, j - v]K[u, v], \tag{3.20}$$

where $I * K$ is the output after applying the kernel, and $[i, j]$ denotes the indices in the output matrix. The kernel is applied on every valid $a \times b$ sub-region of the input. This operation is repeated for each channel in the case of multi-channel inputs.
The input to a CNN is typically a 3D tensor. Where one has multiple channels c of images $m \times n$. This can be the three channels for the three RGB-values or in our case for each layer in the observation (see later).
The output of a convolutional layer is an other set of channels c' of new images $m' \times n'$, where n' and m' depend on three variables. These are kernel size $(a \times b)$ as discussed before, stride s, which is the distance taken between each kernel operation, and padding p which are zeros that are added around the image. c' is the number of kernels $K_i[u, v]$ (or filters) used. For every output channel there exists multiple filter that each convolve with the separate channels of the input and are then added together: $O_{c'}[i, j] = \sum_c (I_c * K^c_{c'})$, where c is the channel index for the input and c' is the channel index for the output. Rewriting the kernel operation (3.20) with this information we get the exact formula for the output $O_{c'}$ given the input, kernels, stride and padding:

$$O_{c'}[i, j] = \sum_{c} \sum_{u,v=0}^{a,b} I'_c[si - u, sj - v]K^c_{c'}[u, v] \tag{3.21}$$

For instance, in image processing tasks, the input is an image with width n, height m, and a number of channels c corresponding to the color space used (e.g., 3 for RGB images). These neurons represent pixel intensity values. A convolutional layer transforms this input volume to an output volume of size $n' \times m' \times c'$

Another key characteristic of CNNs is the use of pooling layers. These layers are used to reduce the spatial size (width and height) of the input volume and to decrease the computational complexity. A commonly used pooling operation is max pooling, where the maximum value within a certain window is passed to the next layer.

4
Methods

Here we mainly describe the reinforcement learning environment focusing on how its observations, the possible actions and what they do in addition to how the reward scheme works. We then move on to describing implementation details around, which libriaries were utilized and how the environment along with the algorithm was implemented. Next we talk about the experimental setup, describing which metrics we used to assess the agents performance and different hyperparameters we tested and how they work.

4.1 The Environment

The environment in short is a quantum circuit into which one can place swaps in order to rout each layer of the circuit and to "pull back" the routed gates of that layer once one (in this case the agent) is satisfied. Each layer is represented as a $m \times n$ grid for the agent reflecting the connectivity of the architecture (see the observation subsection).
The underlying structure that stores the circuit is a list containing each layer, which is in turn a list containing each gate detailed by the index of the qubits that compose them and a separate list detailing if these gates are swap gates or not. In addition to this there is a counter for the array to indicate which layer one should add new swap gates, where the preceding layers have already been routed (solved layers), and the next layers are to be routed (Unsolved layers).

4.1.1 The Observation

The agent interprets the environment through multi-layered observations of the quantum circuit, where the layout of each layer mirrors the physical grid layout of the quantum computer. Figure 4.1 shows three such observations, where each successive observation (left to right) shows a change in the observation due to the two types of actions: A swap and a 'pullback' action which is explained in detail in the next subsection on actions. Each layer is represented as a matrix where each entry signifies a qubit, with its value indicating the index of the qubit it is meant to interact with. If a qubit doesn't have an interaction in a specific layer, its matrix entry will be zero. As interactions are bidirectional, the partner qubit's entry will reflect the first qubit's index, creating a compact representation of pairwise qubit interactions per layer.
The observation then segregates into specific groupings of layers: solved layers,

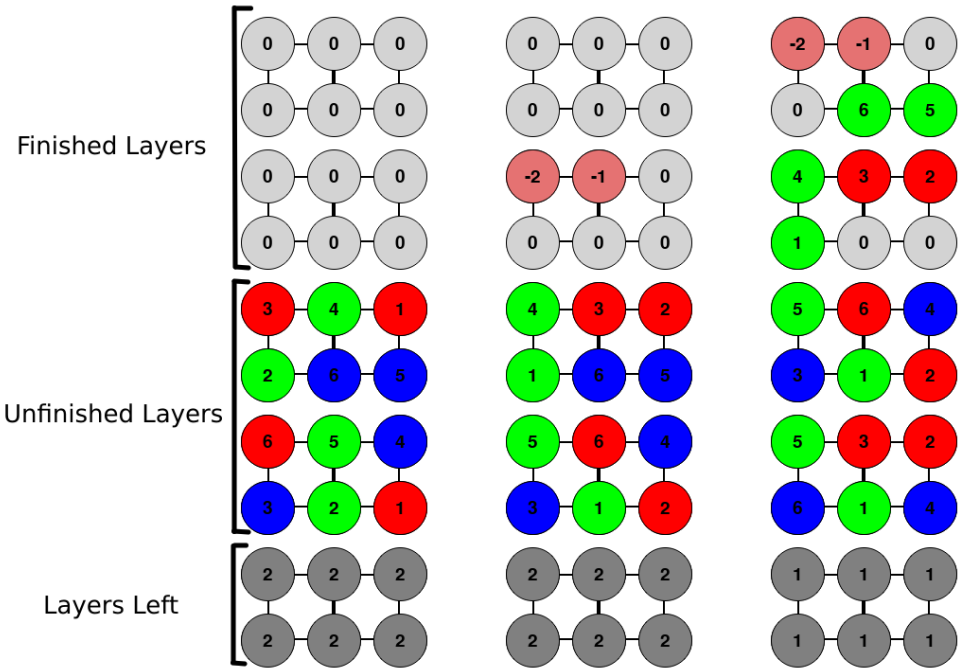


Figure 4.1: The observation for three time-steps. Each observation is composed of multiple layers, where the first layers represent a finished routed circuit. The subsequent layers are layers that are to be routed, and the last layers gives the number of layers beyond that. Each layer is arranged in a grid, a matrix with each entry representing a qubit, and where each qubit has the index of the other qubit it is supposed to interact with (same colors). A swap gate is represented with negative indexing (same indices but negative). The figure shows the two types of action. The first, between the first and second time-step is a swap, which is placed on the first and second qubit. The second action is a 'pullback', which pulls back all solved qubit pairs from the first unfinished layer.

unsolved layers, and a 'layers left' layer.
The solved layers, typically composed of two or more layers, contain completed or routed layers. This is where the swaps are placed, more specifically in the last of these layers. These swaps are marked with negative indices, a differentiation crucial as swaps can be removed by placing additional swaps at the same location. The purpose of displaying this layer is to provide the agent with a comprehensive understanding of the circuit, enabling more strategic placement of swaps to prevent the addition of extra layers.
Following are the unsolved layers, which are the immediate layers that are supposed to be routed and thus usually form the most significant portion of the observations. Like the layers preceding it, it is composed of quantum gates indexed in the same way, but with no swap gates since they all are added in the solved layers.
Finally, the 'layers left' layer consists of entries representing the number of layers remaining beyond the unsolved layers. This provides a measure of the routing task still required which is vital for the value based network to be able to give an accurate estimation, given (as we shall see later) that the reward scheme gives points for solving layers.

This observation format allows the agent to easily process the quantum circuit's state and qubit interactions, facilitating strategic quantum gate manipulations.

4.1.2 The Actions

For an environment there are two types of actions. These are the swap actions and the 'pullback' action. There is one swap action for each of the adjacent qubit pairs in the circuit. Meaning that for a circuit with a grid-size of $m \times n$ we have $m(n-1)+n(m-1)$ swap actions. For a swap action a swap will be placed as far back is possible until it overlaps with qubits of gates in the preceding layer. However, if it fully overlaps with a swap, that swap will be removed, and if there are overlapping qubits in the first finished layer a new layer will be created.
Next there is the pullback action which does two things. First it takes all of the solved qubits in the first unsolved layer of the environment and places them behind in a new layer, the last of the finished layers and thus the new current layer. After that is done the circuit is compressed, in order to ensure that there are as few layers as possible. Here the back and the front layers are done separately in order for the qubits of the solved and unsolved to be separated. In the compression step one iterates through each layer and each gate in that layer, where they are pulled back as far as possible according to the rules established when placing the swap gates. If a layer is empty at the end of this process, then that layer is removed.

4.1.3 The Reward function

The reward function is based how many added vs how many expected added layers one has, and how many layers there are in total. The idea is that if all of the rewards from one completed episode are added up (without a discount factor) $R_{\text{tot}} = \sum_i r_i$

the result should be

$$R_{\text{tot}} = d_{\text{start}} - \beta(d_{\text{end}} - d_{\text{start}}) \tag{4.1}$$

Here d_{start} and d_{end} is the depth of the circuit at the start and end respectively. $s_{\max}$ is the maximum number of steps needed to transport two qubits from opposite ends of the circuit. Since the steps are overlapping it also means the amount of layers that is needed to be added to transport these layers together.
β is a tunable parameter that together with $s_{\max}$ should give an estimate of the maximum number of added layers per layer solved $l_{\max} = \beta s_{\max}$.
This total reward function decreases linearly with the increase in the number of added rewards.
This total reward can then easily be reformulated to a reward for each step:

$$r_i = p_i - p_{i-1} - \beta(d_i - d_{i-1}) \tag{4.2}$$

Where p_i is the number of remaining (pending) layers for the agent to solve for time-step $t = i$, and d_i is the total depth of the circuit for time-step $t = i$.
An alternative is to instead use the number of solved qubit gates for each timestep instead of counting each solved layer. In that case the reward instead becomes:

$$r_i = \frac{g_i - g_{i-1}}{G} - \beta(d_i - d_{i-1}) \tag{4.3}$$

where g_i is the number of remaining gates for the agent to solve for time-step $t = i$ and G is the average number of gates per layer, thus it still satisfies (4.1). Here the total reward is the same, but the agent receives a reward more often, reducing the problem of sparse rewards.
In addition to this the environment receives a fix negative reward σ if it was not able to finish the routing within the allotted maximum steps $S_{\max}$. Finally, we also have a parameter $\eta \in [0, 1]$, which allows one to continually switch between the reward scheme based on layers (4.2) or based on gates (4.3). Effectively we only ever have either $\eta = 1$ or $\eta = 0$. Taken together the full reward is given by:

$$r_i = \eta(\Delta p_i) + (1 - \eta)\frac{\Delta g_i}{G} - \beta(\Delta d_i) - \sigma\delta_{iS_{\max}} \tag{4.4}$$

Here the 'Δ' denotes the difference between the current (i) and previous ($i - 1$) time-step of the given variable (p_i, g_i or d_i), and the δ_{ij} is the Kronecker's delta.

4.2 Implementation

In this project, we leveraged several existing libraries to create an efficient implementation. For the enviromental setup we utilized the Gym library [6], which was chosen to enable seamless integration with stable baselines [20]. The use of stable baselines offered us the benefit of quick implementation and the robustness of the Deep Q-Network (DQN) algorithm, which we modified to suit our value-based approach.
The primary modification was adjusting the structure of the experience replay buffer. In standard Q-learning, the buffer stores tuples in the format (s, a, r, s'). However,

we altered this approach to store the state alongside all corresponding actions, rewards, and next states: (s, a_i, r_i, s'_i).
A second significant change involved the computation of Q-values. Instead of relying on conventional calculations, we derived Q-values from the values of all the next states, i.e., $Q(s, a) = V(s'_a)$. This modification not only impacted the training of weights but also had ramifications on the policy.
For the training process, we utilized the 'rl-baselines3-zoo' repository, an extensive collection that houses a predefined training function. This function accepts a range of parameters, including the choice of algorithm, the number of environments, and the location of hyperparameters. This extensive customization capability greatly expedited the training process and enhanced the overall efficacy of our model.

4.3 Experimental setup

Next we move on to an overview of our testing parameters and the training procedure employed for our 'experimental setup' for reinforcement learning agent. Our tests were conducted extensively on the 3x2 environment using various hyperparameters. Further, primary hyperparameters, which we postulated would exert a significant effect, were rigorously tested on the 4x3 environment. The computations for this research were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC) at Chalmers Centre for Computational Science and Engineering (C3SE), partially funded by the Swedish Research Council through grant agreements no. 2022-06725 and no. 2018-05973. We utilized an A40 GPU for the experiments, conducting tests on multiple nodes. The algorithm took approximately two to three hours to run 500,000 episodes - a timeframe that could likely be improved substantially with more efficient memory management. The circuit configurations for these tests were entirely random, two layers deep, and each layer was fully populated, implying maximum utilization of gates per layer.
We tested various hyperparameters, including the β reward parameter as detailed in equation (4.2), the network type (3D convolutions versus 2D convolutions), different network sizes, and distinct 'exploration fractions'. The exploration fraction, a critical hyperparameter, pertains to our utilization of an epsilon-greedy policy for agent exploration.
As per this policy, a fraction of the agent's actions, determined by ϵ, are randomly selected, thereby ensuring exploration of the environment. Over time, ϵ decays in a linear fashion, starting from $\epsilon = 1$ (indicating completely random action selection) and tapering off to a lower limit $\epsilon = \epsilon_{\text{min}}$, reached at a specific point during the training period. This point in training is denoted by the 'exploration fraction'. Additionally, there is a period during the run where the algorithm only collects experiene without training the agent, and thus only chooses actions randomly, effectively turning $\varepsilon = 1$.
Figure 4.2 shows this by illustrating how ε changes as a function of steps for various exploration fractions: 0.1, 0.3, 0.5, 0.7, and 0.9. Showing the collection period of $s \in [0, 50000]$ where $\varepsilon = 1$, and the training period where it goes over to the decaying ε. This diagram demonstrates the different exploration fractions tested during one

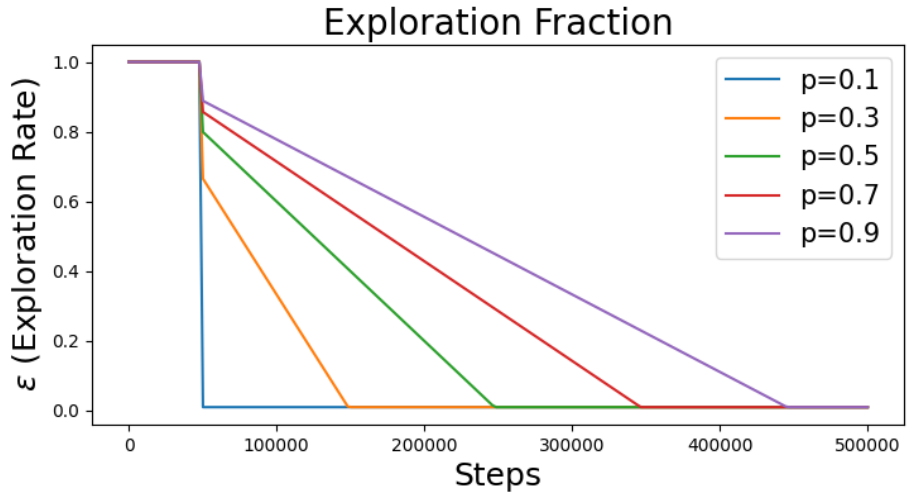


Figure 4.2: Graph of the exploration rate ε as a function of steps s for different exploration fractions ($p = 0.1, 0.3, 0.5, 0.7$ and 0.9), for a run that starts training at $s = 50\,000$ and ends at $s = 500\,000$. Here the minimum exploration rate is fixed at $\varepsilon_{\text{min}} = 0$.

of the experiments detailed in the 'Results' section.

Lastly, it is important to note that we performed hyperparameter tuning not just to optimize the agent's performance but also to enhance our understanding of the underlying issues affecting performance. The purpose of testing various hyperparameters was twofold - firstly, to evaluate the algorithm's potential for optimization, and secondly, to discern whether the characteristics of the environment were contributing to any stagnation in progress. As such, the following results offer insights not just into the effectiveness of the reinforcement learning agent, but also a broader understanding of the impact of environmental and algorithmic factors on its performance.

We primarily relied on two performance metrics, with the first being the number of steps per episode across environments of different complexity - 3x2, 3x3, and 4x3. For the tests focusing exclusively on 3x2 and those where environments of different sizes (3x2, 3x3, and 3x4) were utilized, we established a maximum limit of 100 steps. Meanwhile, for the tests with the larger, more challenging 4x3 environment, we permitted a more generous maximum of 250 steps.

The rationale for adopting this metric was multifaceted. Foremost, it served as an indicator of learning efficiency. A reduction in the average episode length over time signifies that the model is finding optimal solutions more promptly. Secondly, this metric revealed valuable insights about the exploration-exploitation trade-off. Longer initial episode lengths might be observed as the agent explores the environment, which ideally decrease over time as the agent starts exploiting its learned knowledge.

Furthermore, the average episode length allowed us to understand the problem's complexity, with shorter episodes suggesting simpler problems or efficient learning algorithms, and vice versa. It also provided an estimate of the inference time,

a crucial factor in real-world applications where decision-making time is critical. Therefore, while the success rate is an essential measure, the number of steps per episode provided a more comprehensive view of the learning dynamics.

Complementing the primary metric of the number of steps per episode, we also considered the average reward per episode as a secondary performance indicator. This measure gave us insights into the agent's decision-making efficacy during the training process. Specifically, we quantified this by calculating the mean of cumulative rewards garnered by the agent across each episode. Given that the environment was two layers deep, the theoretical maximum possible reward was set to be 2. However, the effective maximum reward varied based on the selected value of β, which represented the penalty for each added layer.
The decision to use the average reward per episode as a performance metric was motivated by several key considerations. Firstly, it allowed us to understand how well our agent was maximizing its rewards, a core principle of reinforcement learning. By evaluating the ability of the agent to optimize its cumulative rewards over time, we could assess its capacity to learn the optimal policy.
Moreover, this metric provided a means to examine the agent's strategy as it navigated through the environment. Specifically, the balance between immediate versus future rewards, a critical component of reinforcement learning algorithms, could be observed through the lens of the average reward per episode.
Finally, the variation in the average reward per episode, depending on the β value, allowed us to study the impact of the penalty associated with adding layers. This served as an effective way to gauge the sensitivity of the agent's learning process to changes in the reward structure, further enriching our understanding of its performance.
Therefore, the average reward per episode, coupled with the number of steps per episode, painted a holistic picture of the learning process, offering deep insights into the performance and decision-making patterns of the reinforcement learning agent.

5
Results

Results include the aforementioned metrics of the average reward, and average number of steps for each episode. In addition to this we look at some examples of how the environment went about solving the problem for the cases where it got stuck, to illustrate the issues that occurred when it didn't go to plan. All of the environments looked at has two layers, reward parameter β of $\beta = 0.1$ or $\beta = 0.2$. Meaning that one has a maximum obtainable reward of $r_{\text{tot}} = 2$. Meaning that any reward episode with a reward over $r_{\text{tot}} = 1$ will have routed the problem correctly. The η and σ parameters were set to $\eta = 1$ and $\sigma = 5$ for all except the 4x3 environment, which had $\eta = 0$ and $\sigma = 10$ respectively

5.1 Environment size 3x2

This section shows four main hyperparameters that were tested, β, the exploration fraction (explained in the method), different types of convolutional neural networks and finally different enviornment sizes.

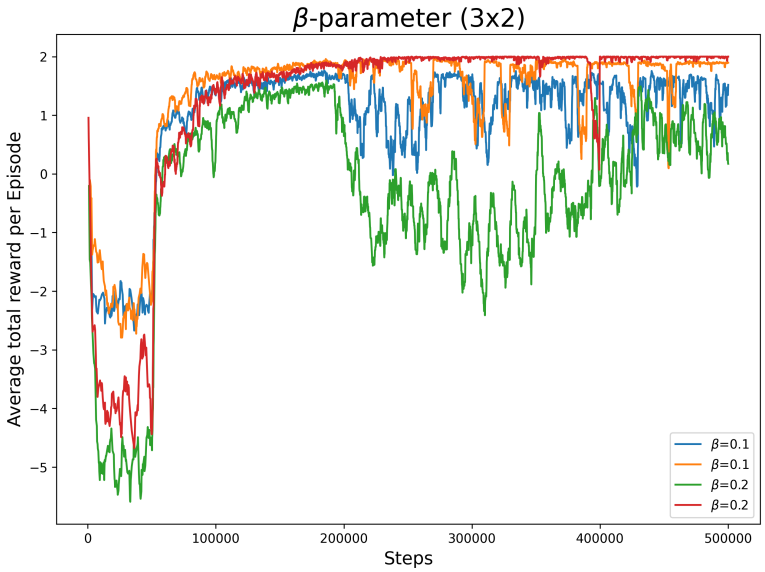


Figure 5.1: Shows average total reward per episode for two runs each of two beta reward parameters, that are responsible for giving negative reward $\beta = 0.1$ and $\beta = 0.2$.

Figure 5.1 average total reward per episode for two runs each of two beta parameters, $\beta = 0.1$ and $\beta = 0.2$.
In figure 5.2 we see the average steps per episode as a function of time-steps taken,

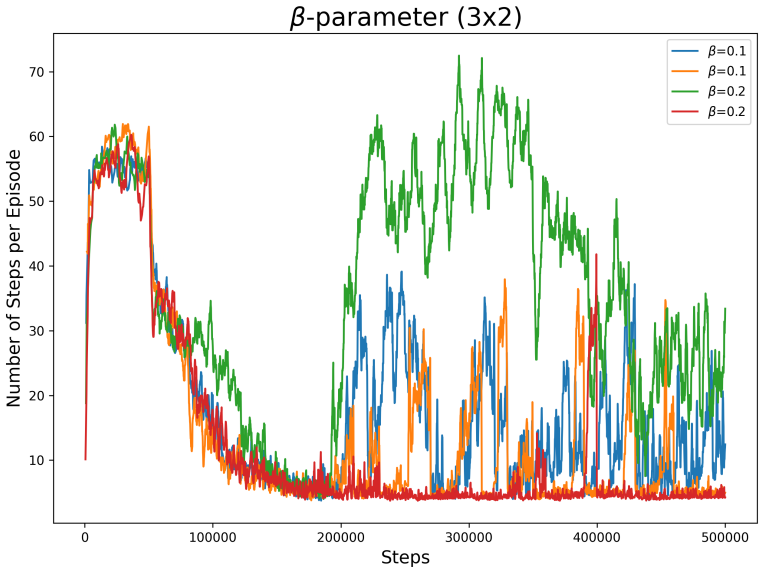


Figure 5.2: Average steps per episode as a function of time-steps taken, for two runs each of two beta parameters, $\beta = 0.1$ and $\beta = 0.2$.

for two runs each of two beta parameters, $\beta = 0.1$ and $\beta = 0.2$.
From these figures one can clearly see the stability caused by the lower β-value, yet that there is quite some variation in the runs.

In figure 5.3 we see the average reward per episode as a function of time-steps and in figure 5.4 we have the average steps for three runs each of both a 3D-convolutional network and a 2D-convolutional neural network, each with 8 convolutional and 4 linear layers respectively.
From these two figures we see a general trend of all heading towards being able to solve the problem, but with some instability some of them, and quite a bit of instability in one case. From the figure there is no clear distinction between the 2D and 3D convolutional networks performance or stability, due to the high variation in between the runs.

Then we have the average rewards 5.5 per episode as a function of steps, for different runs with different exploration fraction, which is the fraction of the training period where the agent explores the environment through the epsilon greedy policy $\epsilon > \epsilon_{\text{min}} = 0.05$. The graph of ε as a function of agent steps for this experiment can be seen in figure 4.2.
From these figures one sees firstly that there is a clear improvement from the increase in exploration, however the second thing to note is the instability in the environment once random actions have been turned off, which can most clearly be seen from for the $p = 0.3$ and $p = 0.5$. This that the agent still hasn't learned the environment

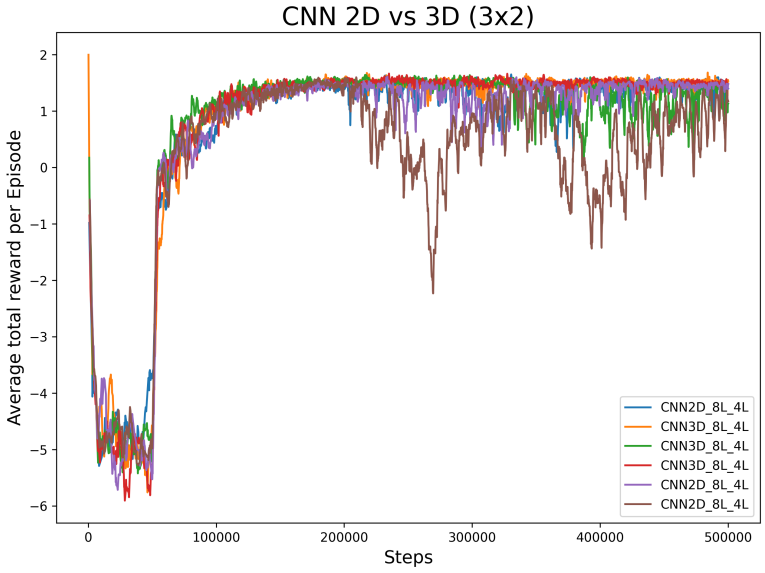


Figure 5.3: Average reward per episode as a function of time-steps taken, for three runs each of both a 3D-convolutional network and a 2D-convolutional neural network.

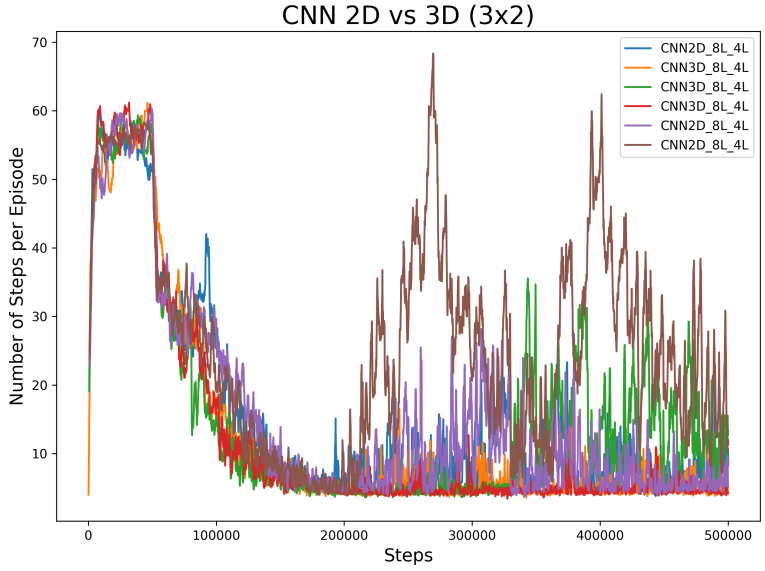


Figure 5.4: Average steps per episode as a function of time-steps taken, for three runs each of both a 3D-convolutional network and a 2D-convolutional neural network.

completely and that the randomness allows it to run into situations that it is more familiar with. Given the way the environment is set up with the ability to remove swaps one has placed previously.

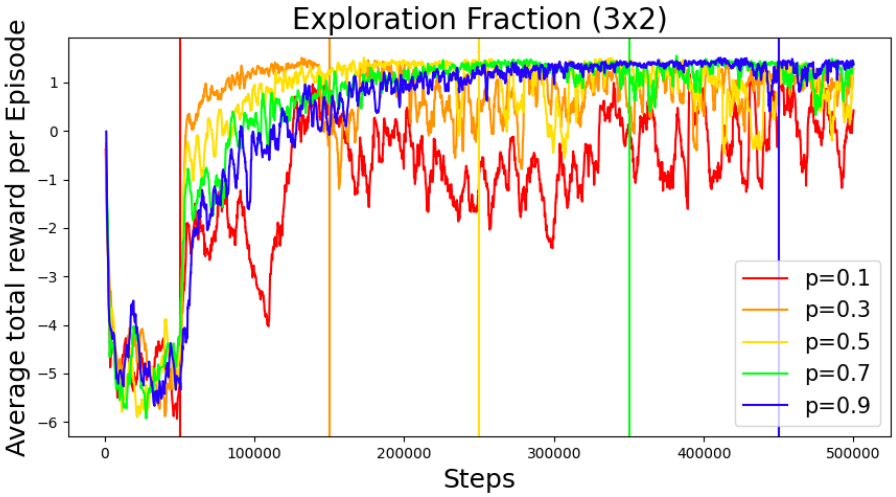


Figure 5.5: Average rewards per episode as a function of time-steps taken, for multiple different exploration fractions, which is the fraction of the training time that environments relies on random actions (epsilon greedy policy). The colored vertical lines represents when ε has reached ε_{min} as seen in 4.2

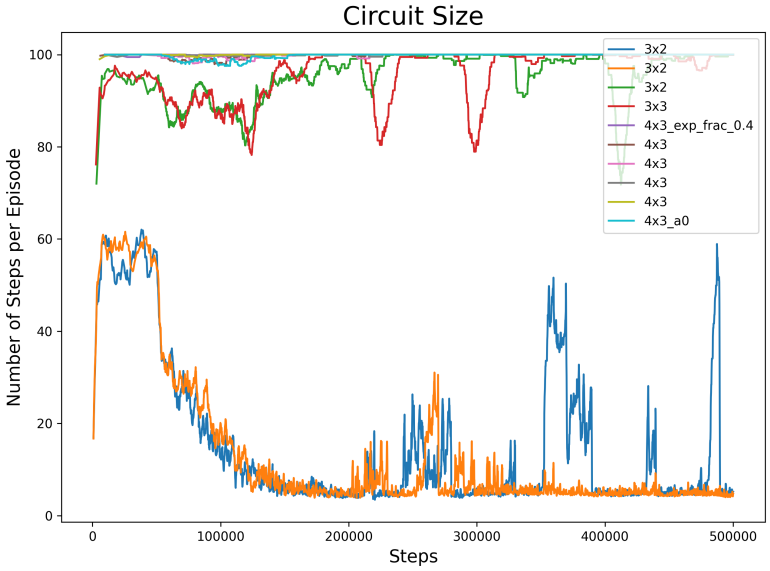


Figure 5.6: Average steps per episode as a function of time-steps taken, for multiple runs of both 3x2, and 4x3 and one 3x3 environment.

Lastly we move on to comparing environments of different sizes, from 3x2, to 3x3, to 4x3 in their average number of steps 5.6 per episode.
Here we have a maximum number of episodes of 100, an exploration fraction of $p = 0.4$ for the smaller circuits and one 4x3 circuit and $p = 0.7$ for the rest.
From the figure one can see the 3x2 runs stabilizing into being able to solve the problem, while it seems like the 3x3 is able to solve an individual layer. Whether or not that is just due to the randomness or the network had some impact is unclear.

Then lastly, we see that the runs for the 4x3, directly stabilize into states where they do some sort of loop to prevent layers from being added and thus end up getting the negative punishment from the $\sigma = 5$ parameter.

5.2 Environment of size 4x3

Next we move on to a circuit of size 4x3 and a depth of two. here we had $\beta = 0.1$ and $\eta = 0$, and the negative reward for not completing the environment after 250 steps was $\sigma = 10$. We just multiple neural networks of varying sizes, convolutions channels of 64 to 256 for each layer, and a corresponding number of neurons in the linear layers, with lengths of both layers being 4 to 8. This can be seen in figure 5.7. This shows the different Networks perform on this environment in terms of the average reward and the average episode length as a function of the time during the training.

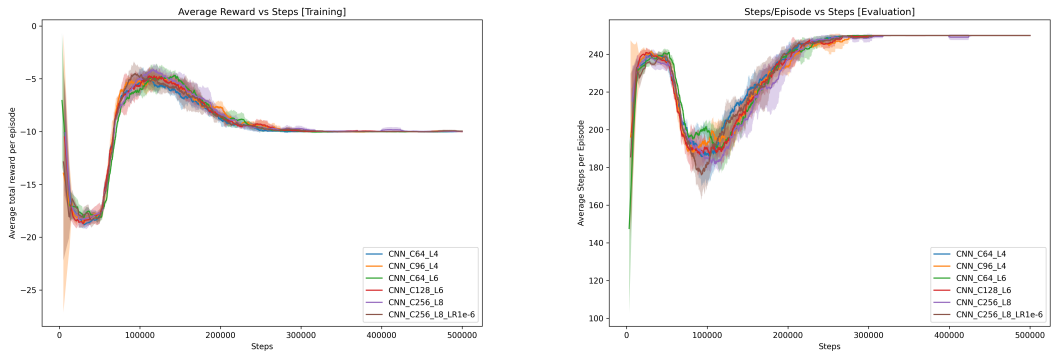


Figure 5.7: Average steps and average rewards per episode as a function of steps, for a 4x3 environment

First off we see no indication of an improvement given larger network size, where the variance shown in Similarly to the 3x3 run from the previous figure of differently sized environments 5.6, we see some indication of being able to solve some layers with the help of randomness. We can see an improvement over the previous figure's 4x3 runs, which is most likely caused by the increase in the number of total steps. We can clearly see from the previous run how it immediately shot up to 100 steps, and thus was not able to receive any positive rewards.

One first sees the great increase in performance from the time the agents starts to perform actions around the 50k step mark (as per usual). Before the performance starts to level off again, and stabilizes around the 250-300k step mark where it falls flat, using up the maximum 250 steps and gaining only the punishment for not routing the circuit ($\sigma = 10$) as punishment.
From these figures there is no indication of improvement due to network size, as one can see more variance between the individual for one network size as apposed

to the difference between runs of different sizes. Similarly to the 3x3 run from the previous figure (5.6), we see some indication of being able to solve some layers with the help of randomness. Given the improvement over the previous figure's 4x3 runs, it appears that being able to take more steps was vital to even allow the environment to be able to solve the environment in the first place.

6
Discussion

The main point of this study was to determine the efficacy of this value based method for solving the routing problem. However due to time constraints, the method was only able to be tested to a point. From this the main point became instead to determine, from what tests that were performed, whether or not the failure of the environment to solve the larger 4x3 case was due to some fault with the hyperparameters or whether or not one could blame the method itself.
From the different 3x2 runs one can clearly see a large difference in the stability of the runs even for runs with the same parameter is tested. Examples of this can clearly be seen in the run for the testing of the 2D in comparison to the 3D convolutional networks 5.4 and the testing of β-parameters 5.2. This large variation makes it difficult to say anything about the efficacy of different β-paramaters nor to compare the performance of either 2D or 3D convolutional neural networks.
Similarly, for the comparison of different sizes of networks for a 4x3 environment (figure 5.7) we see that all the tested networks appear to perform equally well, the range of variances exhibited by the other networks, indicating no distinguishable differences in their performance. Here a wider selection of networks with a large difference in size were tested. Which means that we can with more confidence say that the network size didn't matter much.

The main parameter of interest however was the exploration fraction, where one can see that the lower ones does not allow the agent to explore enough to be able to solve the environment. Interestingly one sees a decreased performance after below a certain ε threshold indicating that, randomness had a positive impact on the ability of the agent to solve the problem. However without any randomness it doesn't seem to perform well until it has learned enough as we see with those environments that have learned with randomness for longer. This drop-off in performance as $\varepsilon \to 0$ can be seen in all of the training runs of the 4x3 environment.
Looking closer at the environment one can see the importance of randomness, and how the path to a solution to the problem remains stable in spite of it. Randomly selecting actions presents no inherent issues, as demonstrated by the exploration of various actions leading to immediate states with consistent values. This observation emerges due to the commutative nature of numerous actions, stemming from the fact that a lot of the swaps are non-overlapping. Consequently, for a given state, the values associated the subsequent states tend to align closely, if not identically. Furthermore, the intrinsic reversibility of certain mistakes, like those involving swap gates, diminishes the drawbacks of random action selection. However this reversibility will in some cases cause the agent to get stuck in an infinite loop. This issue gets

rectified by having random actions, as it allows the agent to get to adjacent states outside of that loop.

In light of this, two strategies come to the forefront. The initial strategy involves extended training sessions infused with randomness. This will prevent the agent from getting stuck in these infinate loops. As a result, continuity in training supplemented by randomness emerges as a key pathway for further improvement.
The second strategy contemplates the adoption of an alternative policy, potentially a softmax policy in lieu of the epsilon-greedy approach. By introducing controlled randomness through a softmax policy, the agent will more naturally choose between different non-overlapping actions lead to similarly valued states.
Another potential avenue for further improvement lies in the adoption of Monte Carlo Tree Search (MCTS). The success of MCTS in various problem domains, as demonstrated in previous studies such as [30] and [26], highlights its potential applicability to the quantum routing compilation challenge. By integrating MCTS, the agent could navigate the solution space with a blend of guided exploration, influenced by both randomness and the insights provided by the value-based neural network. This approach allows the agent to look ahead further to see how different actions affect which paths can be chosen, and might give more insight than the similarly valued potential actions given directly by the network.

Furthermore, beyond the realm of reinforcement learning, the neural network's learned value estimates can potentially serve as a valuable metric for classical graph algorithms. One notable approach could involve leveraging the network's insights within a regular graph traversal algorithm like A*. By utilizing the network's value assessments as heuristic information, the algorithm could make more informed decisions about node prioritization during the routing process. This marriage of value-based insights with classical algorithmic techniques could potentially harness the strengths of both paradigms.

In summary, the application of value-based reinforcement learning to the quantum routing compilation problem has demonstrated promising potential, particularly in the context of enhancing decision-making through controlled randomness and extended training. However, looking ahead, the integration of Monte Carlo Tree Search stands out as a viable strategy to further guide exploration, building upon the convergence achieved by the value-based network. Moreover, the fusion of value-based estimates with classical graph algorithms offers an exciting avenue for future research, potentially leading to a powerful symbiosis of quantum-inspired learning and classical problem-solving techniques.

7
Conclusion

This study focuses on Quantum Routing Problem and attempts at finding a solution based on Value-based variant of Q-learning algorithm. Three deep convolutional neural networks have been proposed for three grid setup environments, with grid size 3x2, 3x3, and 4x3. For the 3x2 grid-size the agent was able to find solutions to the problem, with most hyper-parameters not affecting the training much. On the other hand, the two models for 3x3 and 4x3 grid sizes, were only able to reach a steady state, that indicated that it was stuck in an infinite loop. The potential of obtaining a similar success with the models proposed for 3x3 and 4x3 grid sizes were discussed. It is indicated that the proposed models would likely be able to solve the problem if they were trained for longer with randomness. In addition further improvements to the model were discussed, such as using a soft-max policy and using Monte Carlo Tree Search, due to the similar values of the resultant states and the important role randomness had in breaking any loops.
In addition it is essential to acknowledge the invaluable support and computational resources that facilitated the execution of this research. The computations for this study were made possible by the generous provision of resources from the National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC) at Chalmers Centre for Computational Science and Engineering (C3SE).

Bibliography

[1] 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, July 1985.

[2] Philip Andreasson, Joel Johansson, Simon Liljestrand, and Mats Granath. Quantum error correction for the toric code using deep reinforcement learning. *Quantum*, 3:183, 2019.

[3] Szilárd Aradi. Survey of deep reinforcement learning for motion planning of autonomous vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 23(2):740–759, 2022.

[4] Aniruddha Bapat, Andrew M. Childs, Alexey V. Gorshkov, Samuel King, Eddie Schoute, and Hrishee Shastri. Quantum routing with fast reversals. *Quantum*, 5:533, aug 2021.

[5] Paul Benioff. Quantum mechanical hamiltonian models of turing machines. *Journal of Statistical Physics*, 29(3):515–546, 1982.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[7] Alexander Cowtan, Silas Dilkes, Ross Duncan, Alexandre Krajenbrink, Will Simmons, and Seyon Sivarajah. On the qubit routing problem. 2019.

[8] Richard P. Feynman. Quantum mechanical computers. *Conference on Lasers and Electro-Optics*, 1984.

[9] Tuomas Haarnoja, Ben Moran, Guy Lever, Sandy H. Huang, Dhruva Tirumala, Markus Wulfmeier, Jan Humplik, Saran Tunyasuvunakool, Noah Y. Siegel, Roland Hafner, Michael Bloesch, Kristian Hartikainen, Arunkumar Byravan, Leonard Hasenclever, Yuval Tassa, Fereshteh Sadeghi, Nathan Batchelor, Federico Casarini, Stefano Saliceti, Charles Game, Neil Sreendra, Kushal Patel, Marlon Gwira, Andrea Huber, Nicole Hurley, Francesco Nori, Raia Hadsell, and Nicolas Heess. Learning agile soccer skills for a bipedal robot with deep reinforcement learning, 2023.

[10] Steven Herbert and Akash Sengupta. Using reinforcement learning to find efficient qubit routing policies for deployment in near-term quantum computers, 2019.

[11] Julia Kempe. Discrete quantum walks hit exponentially faster - probability theory and related fields, Feb 2005.

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

[13] Aleks Kissinger and Arianne Meijer van de Griend. Cnot circuit extraction for topologically-constrained quantum memories. *Quantum Information and Computation*, 20(7,8):581–596, 2020.

[14] Jung-Shian Li and Ching-Fang Yang. The design of a quantum benes switch. In *2007 IEEE Conference on Electron Devices and Solid-State Circuits*, pages 539–544, 2007.

[15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[16] Giacomo Nannicini, Lev S. Bishop, Oktay Günlük, and Petar Jurcevic. Optimal qubit assignment and routing via integer programming. *ACM Transactions on Quantum Computing*, 4(1), oct 2022.

[17] Peter J. Pemberton-Ross and Alastair Kay. Perfect quantum routing in regular spin networks. *Physical Review Letters*, 106(2), 2011.

[18] Matteo G. Pozzi, Steven J. Herbert, Akash Sengupta, and Robert D. Mullins. Using reinforcement learning to perform qubit routing in quantum compilers. *ACM Transactions on Quantum Computing*, 3(2):1–25, 2022.

[19] John Preskill. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, aug 2018.

[20] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[21] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[22] M. Shamsa and B. Ahluwalia. Qca-based routing mechanism for parallel computers and application in railways. *Proceedings of the 2001 1st IEEE Conference on Nanotechnology. IEEE-NANO 2001 (Cat. No.01EX516).*

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

[24] 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.

[25] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[26] Animesh Sinha, Utkarsh Azad, and Harjinder Singh. Qubit routing using graph neural network aided monte carlo tree search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(9):9935–9943, 2022.

[27] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t|ket⟩ : $aretargetablecompilerforNISQdevices.QuantumScienceandTechnology, 6(1) : 014003, nov2020.$

[28] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[29] Friedrich Wagner, Andreas Bärmann, Frauke Liers, and Markus Weissenbäck. Improving quantum computation by optimized qubit routing. *Journal of Optimization Theory and Applications*, 197(3):1161–1194, may 2023.

[30] Xiangzhen Zhou, Yuan Feng, and Sanjiang Li. A monte carlo tree search framework for quantum circuit transformation. *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020.

[31] Alwin Zulehner, Alexandru Paler, and Robert Wille. An efficient methodology for mapping quantum circuits to the ibm qx architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(7):1226–1236, 2019.