Constrained quantum CNOT circuit re-synthesis using deep reinforcement learning

Author: Arianne van de GRIEND

Supervisors: Aleks KISSINGER
            Johan KWISTHOUT

A thesis carried out at the
Nijmegen Quantum Logic Group
of the Radboud University
for the Artificial Intelligence Master

August 27, 2019
“Quantum computing and consciousness are both weird and therefore equivalent.”

Scott Aaronson & Zach Weinersmith

\[1\] https://www.smbc-comics.com/comic/the-talk-3
Abstract

Nijmegen Quantum Logic Group

Artificial Intelligence Master

Constrained quantum CNOT circuit re-synthesis using deep reinforcement learning

by Arianne van de Griend

In this master thesis, we describe a novel approach to constrained CNOT circuit re-synthesis as a first step towards neural constrained quantum circuit re-synthesis. We train a neural network to do constrained Gaussian elimination from a parity matrix using deep reinforcement learning.

The CNOT circuit is transformed into a parity matrix from which an equivalent CNOT circuit is synthesized such that all CNOT gates adhere to the connectivity constraints provided by the quantum computer architecture.

For our n-step deep Q learning approach, we have used an asynchronous dueling neural network with three different action selection policies: $\epsilon$-greedy, softmax and a novel oracle selection policy. To train this neural network, we have proposed a novel phased training procedure that guides the training process from trivial problems to arbitrary ones while simulating.

Although we were only able to successfully train an agent for trivial quantum computer connectivity constraints, the 2 and 3 qubit coupling graphs. We did show that those agents were able to perform similar to the genetic Steiner baseline and could even improve on them. We also investigated the effect of coupling graph sizes and connectivity on network performance and training time. Lastly, we show that transfer learning can result in an improved network, but it takes longer to train.

This is a very promising start of a new research field that could result in a universal quantum circuit optimization and mapping algorithm that is robust to both expected and unexpected future changes in quantum computer architectures.
# Contents

Abstract ii  
List of Figures v  
List of Tables vi  
List of Abbreviations vii  

1 Introduction 1  
1.1 Outline of this thesis .................................. 4  

2 Quantum computing 6  
2.1 Quantum bits: an analogy ............................... 6  
2.2 Quantum gates and circuits ............................ 7  
2.3 NISQ devices ............................................. 9  
2.4 Quantum circuit mapping ............................... 10  
2.4.1 Qubit routing ......................................... 11  
2.4.2 CNOT circuit re-synthesis ............................ 12  

3 Deep reinforcement learning 15  
3.1 Artificial neural networks ............................... 15  
3.1.1 Neural network architecture .......................... 16  
3.1.2 Training procedure .................................. 17  
3.1.3 Transfer learning .................................... 18  
3.2 Reinforcement learning ................................. 18  
3.2.1 Q-learning ............................................ 19  
3.2.2 Deep Q-Networks .................................... 20  

4 Methods 23  
4.1 The reinforcement learning environment ............... 23  
4.1.1 The reward function ................................ 24  
4.1.2 Extraction simulation procedure ..................... 26  
4.2 Phased training procedure .............................. 27  
4.3 Neural network model .................................. 28  
4.4 Neural CNOT circuit re-synthesis procedure .......... 31  

5 Results 32  
5.1 Neural network performance ............................ 33  
5.2 Effects of the coupling graphs .......................... 35  
5.3 Quality of transfer learning ............................ 35
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The SWAP gate.</td>
<td>9</td>
</tr>
<tr>
<td>4.1</td>
<td>The 4 quantum computer coupling graphs.</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>The neural network architecture that was used.</td>
<td>29</td>
</tr>
<tr>
<td>A.1</td>
<td>Training and validation plots for the 2, 3, and 4 qubit line coupling graphs</td>
<td>56</td>
</tr>
<tr>
<td>A.2</td>
<td>Training and validation plots for the 2 and 3 qubit line coupling graphs</td>
<td>57</td>
</tr>
<tr>
<td>A.3</td>
<td>Training and validation plots for the 3 qubit line and circle coupling graphs.</td>
<td>57</td>
</tr>
<tr>
<td>A.4</td>
<td>Training and validation plots for the 2 qubit line coupling graph with and without transfer learning.</td>
<td>57</td>
</tr>
<tr>
<td>A.5</td>
<td>Training and validation plots for the 3 qubit line coupling graph with and without transfer learning.</td>
<td>58</td>
</tr>
<tr>
<td>A.6</td>
<td>Training and validation plots for the 3 qubit circle coupling graph with and without transfer learning.</td>
<td>58</td>
</tr>
<tr>
<td>A.7</td>
<td>Training and validation plots for the 4 qubit line coupling graph with and without transfer learning.</td>
<td>59</td>
</tr>
<tr>
<td>A.8</td>
<td>Training and validation plots for the 4 qubit line coupling graph without phased training.</td>
<td>59</td>
</tr>
</tbody>
</table>
List of Tables

5.1  Extracted CNOT gate counts for all parity matrices.          34
B.1  Constrained CNOT circuit re-synthesis overhead (%) for 10000 random CNOT circuits.  61
### List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>Artificial Intelligence.</td>
<td>1, 4</td>
</tr>
<tr>
<td>CNN</td>
<td>Convolutional Neural Network.</td>
<td>16, 49</td>
</tr>
<tr>
<td>DDQN</td>
<td>Double Deep Q-Network.</td>
<td>12, 21</td>
</tr>
<tr>
<td>DQN</td>
<td>Deep Q-Network.</td>
<td>2–4, 20, 23, 39, 49</td>
</tr>
<tr>
<td>FC</td>
<td>Fully Connected layer.</td>
<td>16</td>
</tr>
<tr>
<td>GRU</td>
<td>Gated Recurrent Unit.</td>
<td>16</td>
</tr>
<tr>
<td>h-DQN</td>
<td>hierarchical Deep Q-Network.</td>
<td>21</td>
</tr>
<tr>
<td>LSTM</td>
<td>Long Short Term Memory.</td>
<td>16</td>
</tr>
<tr>
<td>MDP</td>
<td>Markov Decision Process.</td>
<td>19</td>
</tr>
<tr>
<td>MLP</td>
<td>Multi-Layer Perceptron.</td>
<td>16</td>
</tr>
<tr>
<td>NISQ</td>
<td>Noisy Intermediate-Scale Quantum.</td>
<td>9</td>
</tr>
<tr>
<td>PSO</td>
<td>Particle Swarm Optimization.</td>
<td>46</td>
</tr>
<tr>
<td>ReLU</td>
<td>Rectified Linear Unit.</td>
<td>17</td>
</tr>
<tr>
<td>RL</td>
<td>Reinforcement Learning.</td>
<td>4, 18, 23, 37</td>
</tr>
<tr>
<td>RMSProp</td>
<td>Root Mean Square Propagation.</td>
<td>18</td>
</tr>
<tr>
<td>RNN</td>
<td>Recurrent Neural Network.</td>
<td>16, 49</td>
</tr>
<tr>
<td>SGD</td>
<td>Stochastic Gradient Descent.</td>
<td>17</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Quantum algorithms are known to be faster than their classical counterparts which can be beneficial for various domains. One such domain is text mining, the automatic processing of large amounts of text that is often used for processing data from the web. For example, quantum classifiers can learn quicker than classical classifiers (Yoo et al., 2014). Similarly, texts can be searched faster with quantum algorithms (Ambainis and Montanaro, 2012) even when the search query contains wildcards (Ramesh and Vinay, 2003). This is extremely useful for techniques such as sentiment analysis and information retrieval. Furthermore, quantum logic can better represent the semantics of natural language as opposed to classical logic (Zeng and Coecke, 2016). Moreover, large quantum computers will be invaluable for processing large amounts of data because they are more efficient in storing it (Schumacher, 1995).

Unfortunately, these techniques can only be directly applied once quantum computers become large enough to do large-scale computing. But that does not make quantum machine learning algorithms less relevant. Because quantum technology requires researchers to think about the existing AI technologies in a new way, it can inspire new classical algorithms in their own right (Preskill, 2018). In fact, this has already happened with the new quantum-inspired algorithm for recommendation systems that is faster than the previously existing ones (Tang, 2018). Thus, quantum computing is a useful for AI and we will show in our research that AI can also be a useful tool for quantum computing too.

In this master thesis, we will use established Artificial Intelligence (AI) techniques to solve a recently introduced interpretation of the quantum circuit mapping problem: The constrained CNOT circuit re-synthesis problem (see section 2.4 and section 2.4.2 for an explanation). Solving this problem more efficiently creates circuits with smaller errors, which makes smaller quantum computers more usable such that the field of AI can benefit from the new quantum techniques described above.

AI techniques have a proven track record of solving complex problems (see e.g. Milan et al., 2017), but in practice, neural networks are mostly used for text and image processing. Theoretically, neural networks can approximate any function (Csáji, 2001), thus we expect that they can also learn to find suitable heuristics for any complex optimization problem.

One such complex optimization problem is quantum circuit mapping. This is
the problem of preparing a quantum circuit, to be executed on a quantum computer. A quantum circuit is a type of computer program for a quantum computer that describes the operations on the level of logic gates over qubits. Arbitrary quantum circuits can be expressed by compositions of a set of single qubit gates and the CNOT gate (Barenco et al., 1995). Most currently existing quantum computers have their qubits laid out in a fixed structure, i.e. the topology, where multi-qubit gates may only be applied between neighboring qubits (Cowtan et al., 2019). This restricts which gates can be used in the quantum circuit, but the quantum circuits to be executed may contain gates that do not abide by those restrictions. Quantum circuit mapping solves this problem by adjusting the quantum circuit such that it can be executed on a quantum computer with the given set of connectivity constraints.

Aside from being mapped to the quantum computer, a quantum circuit also needs to be as small as possible with respect to the number of gates. This is necessary because the calculations that the current quantum computers can do are not flawless: They introduce a small amount of noise to the result each time a gate is applied. If a quantum circuit requires too many gates, the noise added to the result can become so large that it is impossible to read the result, making the computations pointless. Theoretical solutions for programs that remove these introduced error, error correction codes, require more qubits than the current quantum computers have (Paler, 2018; Preskill, 2018).

Secondly, the current quantum computers can only maintain the state of a qubit for a small amount of time, the decoherence time. This means that if computations take too long, even if it introduces just a small amount of error, the qubit can lose their quantum state before the calculations are finished (Li et al., 2018).

The quantum circuit mapping problem is usually solved by swapping qubits across the quantum computer architecture, such that all quantum computer connectivity restrictions are adhered to (see section 2.4.1 for an overview). But in our research, we will address this problem from a constrained CNOT circuit re-synthesis perspective (see section 2.4.2 for more details).

In this thesis, we describe a general approach to re-synthesize CNOT circuits using AI. Machine learning techniques have been successfully applied to classical compiling (Wang and O’Boyle, 2018), so we expect that quantum compiling techniques can also benefit from machine learning. In fact, simple optimization algorithms, such as A* and temporal planning, have already been used in the past for finding optimal swaps (Zulehner et al., 2018; Venturelli et al., 2019, respectively). Recently, a Deep Q-Network (DQN) was used to find better swapping heuristics (Herbert and Sengupta, 2018). However, solving quantum circuit mapping from a swapping perspective will always result in adding more gates (Herbert and Sengupta, 2018).

Nevertheless, recent work by Kissinger and Meijer-van de Griend (2019) and Nash et al. (2019) has shown the quantum circuit mapping problem can also be solved from another perspective where the resulting circuit could even have less gates than the original circuit. However, this was only shown for circuits consisting of solely CNOT gates. The authors focused on mapping the sequences of CNOTs between the single qubit gates in the given circuit since the connectivity constraints of the current quantum computers only affect CNOT gates. These sequences of
CNOTs describe a parity that can be represented in a square matrix called a parity matrix. Semantically equivalent sequences of CNOTs can be extracted from such a parity matrix with Gaussian elimination (see section 2.4.2 for a detailed description). However, this Gaussian elimination procedure needs to be constrained with respect to the connectivity restrictions posed by the quantum computer. Kissinger and Meijer-van de Griend (2019) and Nash et al. (2019) have proposed different algorithms to achieve this. In this research, we will propose a method to teach a neural network to find such an algorithm automatically.

To summarize, these new algorithms do not adjust the original circuit by adding SWAP gates, but they re-synthesize parts of the circuit from a higher level representation instead. Re-synthesis allows for more flexibility when placing gates, therefore such an approach can result in smaller mapped circuits for pure CNOT circuits as was shown by Kissinger and Meijer-van de Griend (2019) when compared to the best current quantum compilers.

In our research, we will make the first step towards investigating whether it is possible to train a Deep Q-Network (DQN) for scalable end-to-end automatic quantum circuit re-synthesis as a compiler for quantum computers. We do this by investigating the use of a DQN for constrained CNOT circuit re-synthesis on small, fictional quantum computers and their connectivity constraints. This allows us to determine whether our proposed approach to quantum circuit mapping is desirable and whether its performance is promising. Although this is a significant reduction of the full problem, this problem is still NP-hard (Amy et al., 2018).

The main contribution of this research is the novel approach to quantum circuit mapping using AI techniques. However, our research can also be used to study the limitations of the currently established techniques when using them in a new domain such as quantum computing.

Since DQNs have not been used to solve this problem before, we have a limited theoretical framework for this research. Therefore, we start with trivial examples which are to be extended in future research. The quantum computer connectivity constraints can be represented in a graph, called the coupling graph. We used a fictional quantum computer with 3 qubits connected in a line as the smallest restricted coupling graph. We also train our neural network for 2 and 4 qubit line graphs and the fully connected 3 qubit circle graph to investigate the scalability of our approach over varying numbers of qubits and over varying qubit connectivity.

The neural network architecture that we used is inspired by the network from Herbert and Sengupta (2018) that was used for finding optimal SWAP gate placements. We extended it with an asynchronous n-step dueling structure, as opposed to the double DQN (see section 4.3 for more details).

A useful tool when working with neural networks is transfer learning. This is a technique that initializes the (subset of) weights of an untrained neural network with those of a trained neural network. The heuristics learned by the DQN should, in theory, be transferable to different coupling graphs. Moreover, constrained CNOT circuit re-synthesis is a complex problem where transferring heuristics from another network could improve training time. Thus, we will also investigate the quality of such transfer-learned neural networks.
Chapter 1. Introduction

Training a DQN to learn suitable heuristics for the quantum circuit mapping problem has the added benefit that it could be used as a universal approach to solving that problem. Quantum computers are still in active development, so they might change drastically in the future. Such changes could bring new quantum circuit mapping restrictions that make the existing swapping algorithms obsolete, thus requiring the development of new algorithms. A DQN, on the other hand, only requires a new description of the problem in the form of a Reinforcement Learning (RL) environment and the algorithm will automatically learn new heuristics from that description. Therefore, in theory, a RL approach to quantum circuit mapping would be more robust against changes in the quantum computer design than the current algorithms.

1.1 Outline of this thesis

This thesis will be very interesting for quantum computer researchers who are working on the quantum circuit mapping problem as well as artificial intelligence researchers who are interested in the generalizability of AI in new domains. Therefore, we expect our readers to be well versed in either quantum computing or artificial intelligence, but not both.

Because of this, we will give a brief introduction to both quantum computing and deep reinforcement learning in Chapter 2 and 3, respectively. The focus of these chapters is to get the reader familiar with important terms and techniques that will be used in the remainder of this thesis. Since this is an AI master thesis, we do expect the reader to have some basic understanding of AI or a strong mathematical background. However, we do suspect that Chapter 3 gives a suitable introduction for readers unfamiliar with AI techniques. Nevertheless, if new concepts appear at any point, we suggest to the reader to look them up on Wikipedia for a simple explanation. Readers who are interested in what we can realistically expect from quantum computing are referred to Preskill (2018), which gives a comprehensible introduction.

In the remainder of this thesis, we will first give the necessary background information that is needed to understand our research. We have split this into a quantum computing (Chapter 2) and an Artificial Intelligence (AI) (Chapter 3) part. Both chapters start with an overview of the fundamentals and end with a detailed explanation of the specific techniques used in our research.

The quantum computing chapter gives an introductory level description of qubits, quantum circuits and quantum computers (section 2.1, 2.2, and 2.3), followed by a description of the quantum circuit mapping problem and two current perspectives of solving this problem: Qubit routing (section 2.4.1) and CNOT circuit re-synthesis (section 2.4.2). The latter is the problem that we are trying to solve with machine learning.

The AI chapter starts with a simple explanation of neural networks (section 3.1) and Reinforcement Learning (RL) (section 3.2). The latter includes recent developments in DQNs in section 3.2.2 of which we have used a few techniques to build train our neural network.
Chapter 1. Introduction

This is followed by the description of our RL environment (section 4.1) and the methods (Chapter 4). We used a novel phased training procedure, that is described in section 4.2. The neural network architecture that we used is described in section 4.3. We also describe how the trained neural network and reinforcement learning environment can be used to find a solution to the constrained CNOT circuit re-synthesis problem in section 4.4. We evaluate the performance of our trained RL agent in Chapter 5. In particular, we discuss the quality of the re-synthesized circuits with respect to the (genetic) Steiner-Gauss baselines (section 5.1), we compare the performance of trained agents for different quantum computer architectures (section 5.2), and we compare the quality of the transfer-learned neural networks with the neural networks trained from scratch for the same quantum computer connectivity constraints (section 5.3).

In Chapter 6, we discuss our design decisions and the limitations of our approach. In particular, our environment design (section 6.1), the phased training approach (section 6.2), the use of neural networks and their design (section 6.3), the usability of transfer learning in our approach (section 6.4), and other limitations (section 6.5). This is followed by our conclusions in Chapter 7.

Lastly, we give an extensive overview of possible future research in Chapter 8. Where we first describe possible ways to use partially trained neural networks for quantum circuit mapping (section 8.1). Then, in section 8.2, we describe how our approach can be extended to include the initial qubit placement (section 8.2.1), parallel gates (section 8.2.2), and better error estimation (section 8.2.3). And finally, we give a few suggestions on how our approach can be used for full quantum circuit re-synthesis by either integrating it in other existing algorithms (section 8.3.1) or by extending the neural network to extract a full quantum circuit (section 8.3.2) in section 8.3.
Chapter 2

Quantum computing

As this is an artificial intelligence master thesis, the reader is not expected to know anything about quantum computing. Therefore, this chapter will first give an introduction to quantum computing, before going into the details required to understand the problem that this research is trying to solve and to place it in the context of previous developments.

Analogous to classical computing, quantum computations are described by a set of operations, called gates, and which quantum bits (qubits) they act on. To give a feel for the quantum nature of qubits, section 2.1 describes their logical properties using a cat-friendly analogy. Then an overview of quantum gates is given in section 2.2 together with a means of combining gates into programs: quantum circuits and programming languages. Afterwards, a brief introduction into the limitations of current quantum computers is given in section 2.3. Lastly, the quantum circuit mapping problem is discussed (section 2.4) with an overview of previous SWAP-based (section 2.4.1) and constrained CNOT re-synthesis (section 2.4.2) approaches.

2.1 Quantum bits: an analogy

To get a feel for quantum computing, we will describe classical bits and quantum bits (qubits) using Schrödinger’s cat. Suppose we model a classical bit as a box with a cat in it or not$^1$. If the cat is in the box, the value of the bit is 1 and if the cat is not in the box, it is 0. The gates of this system are rules that describe what happens to the cat if the gate is applied to the bit. For example, the NOT-gate adds a cat to the box if it was empty and otherwise it removes the cat. If you have enough cats, boxes, friends and catnip, this is essentially all you need to build a classical computer.

Qubits are not that much different from these cats in boxes, except that this time, the boxes are closed. This may not seem like a big problem, but if the boxes are closed, how do you know if there is a cat inside? For sake of the analogy, let’s assume that if we do not know if a cat is in the box, it is both a little bit full and a

$^1$In the original version, the cat was either dead or alive, but we found it a bit too morbid when applying a NOT-gate.
little bit empty. So the box has two states (*super position*): the cat is inside the box and the box is empty.

This can be described as a linear combination of the two states. If the constants in the linear combination are positive real numbers smaller than or equal to zero, the linear combination would describe the probability of the qubit being in one state or another. However, this is not enough to describe the quantum state. The probability of observing the state of the box depends on how you look at the box. Therefore, complex numbers are used to describe these conditional probabilities. Resulting in a generalization of ordinary probability.

A neat side-effect of using complex *probability amplitudes* that they can be positive, negative or complex. Thus, they can cancel each other out. This is called interference and it is the core reason behind the low computational complexity of quantum algorithms. The trick is to design an algorithm where undesired outcomes interfere with each other and cancel out, leaving only the desired outcomes. However, designing such an algorithm is not straightforward which explains why only a few quantum algorithms exist.

Since these algorithms are not needed to understand our research, we will not go into more detail, but the interested reader is referred to the surveys by Mosca (2009) and Montanaro (2016). A good low level explanation of quantum interference is also given in the web comic “The talk” by Scott Aaronson and Zach Weinersmith.

Although quantum interference can be very powerful, as we will show later, qubits have one major caveat: if they are observed, they lose their quantum *super position*, they *collapse*. From our cats in boxes analogy, this can also make sense. The super position exist as long as you do not know if the cat is inside the box. Once you looked, you do know whether the cat is in the box and thus the other state cannot exist, you lose the quantum state. Unfortunately, qubits can only keep their super position under extreme conditions, making them quite unstable.

Another drawback of qubits is that they cannot be copied either. This makes sense, intuitively, because in order to make a copy, you need to observe the state of the qubit that you want to copy. As a direct result, all calculations on qubits with a quantum computer need to be performed locally. Which calculations can be performed on qubits are described in the next section.

### 2.2 Quantum gates and circuits

In practice, qubits are described by a position on a sphere, the *Bloch sphere*. The operations that can be done on a qubit are equivalent to rotating such a sphere. In quantum computing, we distinguish 6 different rotation gates:

1. The *Pauli-X gate*, a rotation of \( \pi \) around the X-axis, this is equal to the classical *NOT gate*,

2. the *Pauli-Y gate*, a rotation of \( \pi \) around the Y-axis,

---

2 [https://www.smbc-comics.com/comic/the-talk-3](https://www.smbc-comics.com/comic/the-talk-3)
3. the Pauli-Z gate, a rotation of $\pi$ around the Z-axis,
4. the S gate, a rotation of $\pi/2$ around the Z-axis,
5. the T gate, a rotation of $\pi/4$ around the Z-axis,
6. and the Hadamard gate, a rotation of $\pi$ around the Z-axis followed by a rotation of $\pi/2$ around the Y-axis.

This gate set can be extended with two-qubit gates where the first qubit controls if the second qubit is rotated with either an X, Y or Z gate. These controlled gates are called the cX gate, cY gate and cZ gate, respectively. Since the X gate is semantically equal to a NOT gate, the cX gate is equal to the classical CNOT gate.

The Hadamard gate changes the perspective from which you are looking at the sphere from X-axis to the Z-axis and vice versa. So applying a Hadamard, an X and a Hadamard gate is the same as only applying a Z gate. Similarly, applying a Hadamard, a Z and a Hadamard gate has the same effect as only applying an X gate. Applying two Hadamard gates after each other is the same a doing nothing, you change perspective and you change it back. This means that a CNOT (i.e. cX) and a cZ can be interchanged by adding a Hadamard gate before and after it on the target qubit.

The collection of CNOT, Hadamard and S gate is called the Clifford group and it has been proven that combining the Clifford group with the T gate is approximately universal for quantum operations (Backens, 2014). Thus we can combine the gates from this Clifford+T gate set to construct the other quantum gates previously described. This means that we only need to be able to rotate qubits, possibly under the control of another qubit, to do any quantum calculation.

An important example of such calculation for the remainder of this chapter is the SWAP gate. The result of calculating this gate over two qubits is that the location of the qubits is swapped. The SWAP can be constructed by applying a sequence of 3 CNOT gates where the middle one is reversed (the control and target is switched around, see Figure 2.1).

In order to describe more complicated quantum computations and even quantum programs, a standardized notation of connecting these basic gates is needed. One such universal method is in a quantum circuit. It is very similar to the classical circuit notation that represents the sequence of gates using wires that represent the (quantum) bits. A simple example of this is shown in Figure 2.1.

The quantum circuit notation describes calculations on a gate level. First this gate needs to be applied on this qubit, than this other one, etc. This has the drawback that it may not be obvious that two circuits perform the same calculations at first glance. This is partially because some gates (such as the SWAP) can be constructed from other gates. So if one circuit uses SWAP gates and the other only CNOT gates, they will look very different, but what they calculate, their semantics, are the same, see Figure 2.1 for an example.

Another, lesser known, diagrammatic language for quantum computations is ZX-calculus (Coecke and Duncan, 2011). With this notation it is easier to reason that two diagrams are semantically the same. On the other hand, an algorithm that
distills the basic quantum gates from such a diagram is yet to be made (Duncan et al., 2019).

In practice, quantum computer programs are written in a plain text programming language. Analogous to classical programming languages, many different ones have been developed. Most quantum languages were either inspired by or built upon classical programming languages. Examples are Open QASM (Cross et al., 2017), Scaffold (Murali et al., 2019), Quipper (Green et al., 2013) and Quil (Smith et al., 2016).

Qubits, quantum circuits and languages merely describe which calculations need to be performed. These operations can then be executed on a quantum computer, which will be discussed in the next section.

2.3 NISQ devices

Quantum circuits describe a set of operations to be calculated. This can be done with a quantum computer. In essence, this is a machine with a physical realisation of qubits and quantum gates.

Just like classical bits can be physically created in different ways, semiconductors on a chip, light switches, or domino’s, qubits can also be realised in multiple ways. At the time of writing, existing quantum computers use superconductors, but other quantum technologies are being researched, such as ion traps (Cirac and Zoller, 1995), linear optical devices (Knill et al., 2001), quantum dots (Loss and DiVincenzo, 1998), and NV-centres (Fuchs et al., 2011). Although the exact technology behind quantum computers is not important for this thesis, what is important is the number of qubits on these devices and their limitations.

These existing quantum computers are categorized as Noisy Intermediate-Scale Quantum (NISQ) devices. As the name suggests, NISQ devices are characterized by a small number of noisy qubits. Current quantum computers range from 16 (e.g. IBM and Rigetti) to 72 (Google) qubits (Zulehner et al., 2018; Preskill, 2018). Note that we are ignoring the D-Wave quantum computers with a few thousand qubits, because they are not general purpose quantum computers (Preskill, 2018).
The noisy nature of NISQ devices means that the qubits will not perfectly represent their logical semantics. The qubits have a short coherence time, after which the qubits lose all their information. The operations that are performed are also imperfect, so applying gates will introduce errors into the result. This means that all calculations need to be performed as quick as possible with as little gates as possible (Cowtan et al., 2019; Li et al., 2018).

Another limitation of the current superconducting devices is that gates cannot be applied between arbitrary qubits. Typically, a quantum computer is specified by a fixed topology describing between which qubits a multi-qubit gate can be applied. This topology is called the coupling graph. Usually, the coupling graph only supports a single type of two-qubit gate, such as a CNOT (e.g. IBM) or cZ (e.g. Rigetti) gate. All CNOT, respectively cZ, gates that are not allowed in the topology, as well as all other multi-qubit gates would need to be constructed from the 2-qubit gate in the coupling graph (Cowtan et al., 2019). In some devices, this 2-qubit gate is also directional, meaning that they can only be applied with the specified control and target qubit. Directional architectures will be ignored for the remainder of this research, because they can be reversed by adding Hadamard gates (Paler et al., 2018).

2.4 Quantum circuit mapping

As explained in section 2.3, the currently existing quantum computers only allow the calculation of a CNOT gate between qubits as specified in it’s coupling graph. This means that some CNOT gates in a quantum circuit cannot be directly calculated on a particular quantum computer. And because qubits cannot be copied (see section 2.1), classical routing algorithms cannot be applied for this problem (Cowtan et al., 2019).

Instead, the given circuit needs to be adjusted such that it adheres to the quantum computer’s connectivity constraints. This will result in a new circuit that is semantically the same (Cowtan et al., 2019). However, the adjusted circuit still needs to have as little gates as possible to reduce noise. Furthermore, those gates need to be parallelized as much as possible due to the short decoherence times of NISQ devices. The problem of finding such an optimal circuit is called the quantum circuit mapping problem.

The main approach to solving this problem is by swapping the qubits across the quantum computer architecture such that the CNOT in the original circuit can be executed. Some different solutions to this approach will be discussed in section 2.4.1. However, swapping the qubits will always result in more gates when the original circuit does not fit the architecture. Instead, recent solutions have been proposed to re-synthesize the CNOTs under the constraints of the coupling graph which will be discussed in section 2.4.2.

Both of these approaches are very sensitive to the original qubit placement in the quantum computer since realistic topologies are generally not symmetric. The problem of placing the qubits such that the resulting mapped circuit has the least
amount of CNOT gates is called the qubit allocation problem and it is proven to be NP-complete (Siraichi et al., 2018).

However, we have kept this problem outside of the scope of this research. Approximate solutions can be found using heuristics (such as Paler, 2018) or with a genetic algorithm (Kissing and Meijer-van de Griend, 2019).

2.4.1 Qubit routing

One way of solving the quantum circuit mapping problem is from the point of view that the problem is caused by the qubits being in the wrong place on the quantum computer. Thus the qubits need to be moved such that all 2-qubit gates can be executed under the coupling graph constraints. The problem of deciding where the qubits need to be moved is called the qubit routing problem which is solved by adding SWAP gates to the original circuit to move the qubits to their intended location. In some cases the problem is simplified by moving the qubits back after applying the CNOT gate, but this will most likely result in more added gates than if qubits remain in their new position (Paler et al., 2018).

The problem of finding the optimal SWAP gates to place in the circuit is at least NP-hard (Cowtan et al., 2019). The search space is $2^n q! |V|^n |n!$ where $q$ is the number of qubits in the circuit, $n$ the number of CNOT gates and $|V|$ is the number of qubits that the quantum computer can hold (Paler et al., 2018). This shows that for circuits with larger amounts of qubits, the kind that cannot be simulated classically, it is computationally infeasible to find an exact solution (Paler, 2018).

Nevertheless, this does not mean that near optimal solution cannot be found using exact methods. The Scaffold compiler uses the Z3 SMT solver to find optimal solutions to a set of sub-problems that approximate the full qubit routing solution (Murali et al., 2019). Wille et al. (2019) recently proposed a formulation to find the exact minimal number of swaps, but their results that it could take more than 8 minutes for a single 5 qubit circuit, depending on the circuit. They did propose a few heuristics that might improve the runtime, but at the cost of minimality of the solution with respect to the number of added SWAP gates.

In general, scalable qubit routing algorithms rely on heuristics. This includes the current best quantum circuit compilers: T|ket> (Cowtan et al., 2019) and QuilC (Smith et al., 2016).

An approximate solution to the qubit routing problem can also be found using graph search. The original circuit can be described as a dependency graph which can be traversed to find the SWAP locations (Li et al., 2018). Alternatively, the search space of the routing problem itself can be described as a graph to be traversed (Paler et al., 2018).

A side effect of using graph search is that general optimization algorithms can be used to improve the runtime. Zulehner et al. (2018) used the well-known $A^*$ algorithm to traverse the graph more efficiently. Paler et al. (2018) proposed to use an ant colony or genetic algorithm, but this has not been implemented yet.
Other algorithms that stem from artificial intelligence have also been used to solve the routing problem. NASA recently described a quantum circuit compiler that uses automated reasoning (Venturelli et al., 2019). Even neural networks have been applied to this problem. Herbert and Sengupta (2018) used a Double Deep Q-Network (DDQN) to teach a neural network to find the best locations for SWAP gates, which was the inspiration for this thesis.

Although better algorithms for qubit routing are still being developed, approaching circuit mapping from a swapping perspective will always increase the number of gates (otherwise it was already solved) (Herbert and Sengupta, 2018). Even if the original circuit is assumed to be optimized before routing, as in the T|ket> compiler (Cowtan et al., 2019) (at the time of writing one of the best quantum circuit compilers), this might not result in a mapped circuit with the least possible amount of CNOT gates. The mapped circuit might have a minimal number of added SWAP gates, but that does not mean that a semantically equivalent circuit with less CNOT gates does not exist. It is possible that the original CNOTs in the circuit are placed in a sub-optimal location for the routing algorithm. It might also be possible that there exists a semantically equivalent sequence of CNOTs which can be routed with less SWAPs for a given sequence of CNOTs in the original circuit. The qubit routing problem assumes that this is not true. The next section will describe a set of algorithms that map circuits by changing the CNOT sequences in stead of the qubit locations.

2.4.2 CNOT circuit re-synthesis

Another perspective to the cause of the quantum circuit mapping problem is that it is caused by that the wrong CNOT gates are in the circuit. The CNOTs should have been chosen such that they adhere to the quantum computer coupling graph. This means that the sequences of CNOT gates in the original circuit need to be re-synthesized in a way that all resulting CNOT gates adhere to the quantum computer’s coupling constraints. The problem of re-synthesizing a minimal sequence of mapped CNOTs is what we call the constrained CNOT circuit re-synthesis problem. Solving the quantum circuit mapping problem from this angle has the added benefit of flexibility. The resulting circuit might use other CNOTs to obtain the same result. A re-synthesis algorithm can also be directly added to the circuit optimization algorithms discussed in section 8.3.1 whenever the algorithm synthesizes CNOT gates. As a result, the optimized circuit is immediately mapped to the given quantum computer architecture, improving the total compile time.

As explained in section 2.2, a quantum circuit can be considered a series of rotations with sequences of CNOTs in between. In section 2.3, it was explained that the coupling constraints of a quantum computer only affect multi-qubit gates. Usually, the quantum computer only directly supports a single type of multi-qubit operation, which is either a CNOT or a cZ gate. A cZ gate can be created from a CNOT gate and Hadamard gates (see section 2.2), so we can construct mapped circuits by only mapping the sequences of CNOT gates in between the sequences of single-qubit gates.
The main constrained CNOT circuit re-synthesis procedure starts with representing the sequences of CNOTs in the original circuit as a parity matrix. From classical computing, we know that the result of applying a sequence of CNOTs over bits can be represented as the set of parities, one for each bit. Each parity represents which CNOTs are applied to the corresponding bit. This can be represented in a matrix: The parity matrix. Different sequences of CNOTs that are semantically the same also have the same parity matrix. The same holds for quantum CNOT circuits where the parity matrix’s size corresponds to the number of qubits in the circuit: $q \times q$.

A parity matrix can be constructed starting from the identity matrix. For each CNOT in the sequence with control qubit $i$ and target qubit $j$, the $i^{th}$ row in the parity matrix is added to $j^{th}$ row. After all CNOTs are processed, the resulting parity matrix corresponds to the full CNOT sequence.

Note that parity matrices are Boolean matrices, so $1 + 1 = 0$. Therefore, two CNOTs in a row cancel each other out and creation of the parity map can be undone by adding the sequence of CNOTs in reverse. This will result in the identity matrix again. In fact, a semantically equivalent CNOT sequence can be extracted from the parity matrix by finding a sequence of row additions, elementary row operations, such that the resulting parity map is identity once again. The equivalent circuit is then the extracted CNOT sequence in reverse. There are different ways to obtain the identity matrix from the same parity matrix which means that different, yet semantically equivalent, CNOT sequences can be synthesized.

Adding a row of a matrix to another row in the matrix is called an elementary row operation and a well-known algorithm for reducing a matrix to an identity matrix with elementary row operations is called Gaussian elimination or Gauss-Jordan elimination. However, Gaussian elimination has no restrictions on which elementary row operation is allowed. But the connectivity restriction of a quantum computer implies that rows cannot be added to each other arbitrarily. Thus, another, restricted algorithm needs to be used. Two algorithms for this have been proposed: Kissinger and Meijer-van de Griend (2019) adjusted the original Gaussian elimination procedure to adhere to a given coupling graph, and Nash et al. (2019) restricted the CNOT synthesis procedure from Amy et al. (2014) to a given coupling graph.

However, these two algorithms both do not guarantee that the resulting sequence of mapped CNOTs is minimal. Just like qubit routing algorithms, they are very sensitive to the initial qubit placement on the quantum computer.

Both algorithms reduce the parity matrix to the pure identity matrix. However, a better solution might be found if the algorithm would reduce to a permutation of the identity matrix which is equivalent to moving the qubits to another location. A straightforward way of implementing this in the existing algorithms is by permuting the columns of the original parity matrix beforehand. But, this would require the permutation in advance and these algorithms are sensitive to the choice of such permutation. Nevertheless, the permutation can be optimized with a genetic algorithm, for instance (see Kissinger and Meijer-van de Griend, 2019).

In our research, we will teach a neural network to extract a minimal amount
of mapped CNOTs from a parity matrix. The exact procedure will be described in Chapter 4, but first an overview of the required knowledge about deep reinforcement learning is given in Chapter 3.
Chapter 3

Deep reinforcement learning

In our research, we applied a technique called deep reinforcement learning to the CNOT circuit synthesis problem. Since we do not expect the reader to have any knowledge about artificial intelligence techniques, this chapter will give a basic introduction into the concepts required to understand deep reinforcement learning. After this high level introduction, the chapter will go into more details about deep reinforcement learning in particular.

This chapter will start with an introduction into artificial neural networks in section 3.1 as the term “deep” in deep reinforcement learning refers to deep neural networks. Different neural network architectures are discussed in section 3.1.1, how neural networks are trained is discussed in section 3.1.2 and transfer learning is described in section 3.1.3.

Afterwards, reinforcement learning is described (section 3.2) with Q-learning as a specific algorithm (section 3.2.1). This is followed by the combination of neural networks and Q-learning called Deep Q-Networks (section 3.2.2). The latter section will be discussed in a bit more detail, since this is the technique that was used in our research.

3.1 Artificial neural networks

Artificial neural networks are a set of algorithms that are combined to approximate a (unknown) function (Csáji, 2001). This is done by generalizing example input-output behaviour of the function to approximate. For example, an artificial neural network can “learn” a function from an image to a corresponding label. This is called classification and given enough examples, an artificial neural network will give the label of a dog when given the image of a dog as input (for example in Krizhevsky et al., 2012).

Artificial neural networks are inspired by biological neural networks. Since most neural networks in this thesis are artificial, we will simply use the term neural networks for artificial neural networks and add the word biological when needed. In biological neural networks, biological neurons send signals to other neurons and when enough signals are received, the receiving neurons will start sending signals themselves. In this way, information is combined and propagated such that an action might be taken. For example, if you stump your toe, neurons in your foot
will send a signal that your toe hurts to the brain. If your toe hurts enough, you will scream: “Ouch!”

Similarly, artificial neural networks describe a collection of artificial neurons and their connections, usually visualized as a graph. Some nodes are assigned as inputs and they will receive their values from an external source, an image for example. Other nodes are assigned to be outputs and their values will be read and interpreted, e.g. the classification labels.

### 3.1.1 Neural network architecture

Neural networks are structured such that the nodes can be ordered in layers. The input nodes form the input layer, the output nodes form the output layer and the nodes in between are structured into hidden layers. The layers are structured by their connectivity, the first hidden layer contains all nodes connected to the input layer and the last hidden layer contains all nodes connected to the output layer. Note that this is a simplified explanation that is generally true, but some neural networks exist that do not follow this structure (e.g. U-net Ronneberger et al., 2015).

The values of the nodes are mathematically represented in a vector. The connections to the nodes in each layer have a weight that describes how important the incoming signal is. From the weights and the input nodes, the outgoing signal is calculated. This is essentially the weighted sum of the connected nodes. Mathematically, the weights can be represented as a matrix and the entire neural network becomes a sequence of matrix multiplications. However, this just corresponds to a linear function that will always go through the origin. Thus, a bias is added to the weighted sum, such the resulting function can be offset from the origin.

The nodes between each layer can be connected in different ways resulting in different kinds of layers. The simplest kind of layer is the Fully Connected layer (FC), where each node in the layer is connected to every node in the previous layer. A neural network consisting of only fully connected layers is called a Multi-Layer Perceptron (MLP).

However, this will result in too many connections to effectively train the model if the input dimensions are large, for example with RGB images. This is what convolutional layers are used for. The layers use a sliding window over the nodes in the previous layer, calculates its convolution and learns a weight for each convolution to calculate the values of the new nodes. This reduces the dimensions of the layers and thus the number of weights while condensing the information which reduces train and run time. A neural network that uses convolutional layers is called a Convolutional Neural Network (CNN). Convolutional and fully connected layers require that the input has a fixed dimension, but that is not always possible. For example when the input is a variable length sequence, such as a sentence.

A Recurrent Neural Network (RNN) can handle inputs with variable sizes. These neural networks use a memory cell as a layer (e.g. a Long Short Term Memory (LSTM) or a Gated Recurrent Unit (GRU)). The memory cell combines the input with
a memory representation into a new memory representation and an output. The memory representation is called the hidden state and it is supposed to encode all previous inputs. This is a useful technique for variable size inputs, but it is more difficult to find the appropriate weights of the memory cell. Which of these neural network types are most suitable for a task depends on the specific task. Each type has its strengths and weaknesses, but the exact hyper parameters need to be found through experimentation.

A deep neural network is a neural network that has many layers, as opposed to a shallow neural network that only has a few layers. Usually, CNNs are deep, where simple MLPs are shallow.

However, with these structures, a neural network is only able to approximate linear functions. This is resolved by adding a type of filter called an activation function after a layer. The activation function is inspired by the biological action potential of a neuron and it is applied to the value of every node it was calculated from the previous nodes. If this function is non-linear, the neural network can approximate non-linear functions as well. A well-known example is the Rectified Linear Unit (ReLU), which is a function that clips all negative values to 0 (Nair and Hinton, 2010).

### 3.1.2 Training procedure

This describes how a neural network represents a non-linear function. Which function that is is determined by it’s weights. When training a neural network, the weights are adjusted such that they will better fit the given training examples. When training, the neural network predicts the output for a given training example which can be compared to the actually desired output, the ground truth. From the predicted output and the ground truth, a score can be calculated to reflect the performance of the neural network, the loss. Common loss functions are the mean squared error, mean absolute error and cross entropy. The loss reflects the prediction error of the network and it is minimized by adjusting the weights with an algorithm called back-propagation (Rumelhart et al., 1988). With back-propagation, the gradient of the loss with respect to the weights can be calculated. Essentially, the loss is distributed over the weights such that weights that influenced the outcome more get a higher gradient proportional to the loss. The gradients are used to make small adjustments to the weights and the process is continued, this process is called gradient descent.

The function describing how the gradients affect the weights is called the optimizer. The most commonly used optimizer is Stochastic Gradient Descent (SGD) which adjusts the weights with a fraction of the gradients. The fraction is called the learning rate and it influences how fast the neural network learns. A high learning rate means that it gets to a minimum faster, but the changes in the weights could be so large that a better minimum is missed. A low learning rate results in longer training time, but the chance of missing a minimum is reduced. However, a low learning rate has a higher probability to get stuck in a local minimum. Which learning rate is best needs to be determined experimentally for each problem and...
neural network. An extension of SGD that uses a flexible learning rate is \textit{Root Mean Square Propagation} (RMSProp) (Tieleman and Hinton, 2012). It uses the root mean square of the running average of the gradients to scale the learning rate.

Since the loss of different training examples might vary, neural network models are trained using \textit{batches}. A batch is a subset of the training data, typically 16, 32, or 64 examples, over which the loss is combined. This helps stabilize the values of the gradients over training steps.

\subsection{3.1.3 Transfer learning}

Since the behaviour of a neural network is determined by its weights, and the weights are determined by training, it is possible to give a neural network a warm start by initializing its weights based on another neural network that was already trained. This process is called \textit{transfer learning}, because the “knowledge” of one neural network is transferred to another one (Pratt, 1993). If both networks have the same structure and all weights are transferred then this will create a copy of the original network.

Transfer learning can help to train a network with less training examples, when a trained network for a similar task is available. The main idea is that the trained network has “learned” patterns that are not only useful for the original task, but also for other, similar tasks. A good example of this is in classification. A good general purpose image classifier, such as ImageNet (Krizhevsky et al., 2012), can distinguish dogs from elephants, but in order to do that, it has found patterns in images that are needed to see the difference between a dog and an elephant. Some of these patterns may also be useful to detect diseases from CT scans (Shin et al., 2016).

In general, the weights of the last few layers are not transferred to the new neural network. Instead these weights are learned with training, such that the new neural network learns the output for the new task, resulting in the new function to be approximated. In some cases, the transferred weights are also adjusted during training, allowing the network to adjust the old patterns. This is useful when the training examples for the old and new neural network are too different.

\section{3.2 Reinforcement learning}

\textit{Reinforcement Learning} (RL) is a paradigm in artificial intelligence that constitutes a collection of algorithms. RL is used for problems that can be described as a sequence of decisions to be made. The machine learning algorithm that makes these decisions is called the \textit{agent}. The agent exists within a pre-defined dynamic \textit{environment} which it can observe to obtain the current \textit{state}. The agent can also take \textit{actions} in the environment. Taking an action has a chance that the environment transitions into another state. If the state changes to another state because the agent performed an action, the agent receives a reward. A transition is a tuple containing the previous state, the action taken, the new state and the received reward. The agent will keep taking actions indefinitely, unless some stopping criterion is
defined. To solve the decision making problem, the agent needs to learn which actions to take in which situation to maximize the reward. The set of possible states, actions, transition probabilities and rewards is called a Markov Decision Process (MDP).

For example, the agent is in a room with a light and a switch (the environment). The agent can observe whether the light is on or off (the states) and it can flick the switch (an action) to turn the light on or off (a transition) or it can do nothing (another action) to keep the light on or off (another transition). If, after an action, the light is on, the agent receives a reward of 1 and otherwise it receives a reward of 0. Now, the agent can learn to optimize the reward, causing that it will turn on the lights when they are off and otherwise do nothing.

When training a RL agent, the agent takes actions in the (simulated) environment and observes the resulting state. Initially, the agent is only aware of the current state, which actions it can take and that it wants to obtain as much reward as possible. From that it can take actions, learn the entire MDP, and search it to find the best actions. This is called model-based reinforcement learning. However, in most problems, formulating the MDP is simple, but searching it is hard. So model-based RL would give no advantage. For these problems, model-free reinforcement learning is used. As the name suggests, these algorithms do not learn the full model, but they only learn the expected reward of taking an action in a particular state.

Learning which actions to take can be done from example (on-policy) or by trial-and-error (off-policy). The latter needs to balance exploration (trying new actions) with exploitation (enforcing known good actions). How this is balanced is determined by the exploration policy. There are two widely used exploration policies: $\epsilon$-greedy and softmax. In $\epsilon$-greedy exploration, the model chooses the option with the best reward (greedy) with chance $1 - \epsilon$ and otherwise it takes a random action. In softmax exploration, the sampling distribution is proportional to the softmax of the expected rewards of all actions.

Based on the observations of the transitions, actions and rewards, the agent can learn to estimate which action would give the highest reward. Although several algorithms for this exist, we will only discuss the algorithm called Q-learning (see section 3.2.1) since we used this in our research.

### 3.2.1 Q-learning

Q-learning (Watkins and Dayan, 1992) is a reinforcement learning algorithm that uses dynamic programming. It approximates a function that assesses the quality of a taken action in a particular state: $Q : S \times A \rightarrow \mathbb{R}$. The values of each $Q(s, a)$ are stored in a table, the Q-table, and updated while traversing the environment.

Initially, the Q-table is filled with zeroes. If, at time $t$, the agent takes an action $a_t$, it transitions from state $s_t$ to state $s_{t+1}$, and observes reward $r_t$, the table will be updated according to

$$Q^{new}(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a(Q(s_{t+1}, a)))$$
where $\alpha$ is the \textit{learning rate} and $\gamma$ is the \textit{discount factor}. Similar to the learning rate in neural networks, the $\alpha$ controls how much the Q-values are adjusted during training. The Q-function tries to predict the total expected reward, including the future rewards. The discount factor controls how much the future rewards are counted towards the current rewards. If the discount factor is small, the actions with the highest Q-value are those that directly result in a higher reward. Whereas a larger discount factor will result in a Q-table that prefers actions that result in a higher reward later. Note that if $s_{t+1}$ is a final state, meaning that the agent is done, there are not next actions to take and $Q(s_t, a_t)$ will be equal to $r_t$. Given enough examples, this algorithm was proven to converge towards the optimal value function (van Hasselt, 2010).

However, it was also shown that Q-learning tends to overestimate the quality of the actions (van Hasselt, 2010). Which can be resolved using a second Q-table. Both tables approximate the Q-function, but when training, only one table is updated at random. When updating, the chosen table ($Q^A$) uses the other table ($Q^B$) to estimate the Q-value of the next action:

$$Q^A(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q^A(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot Q^B(s_{t+1}, \arg\max_{a\in \mathcal{A}}(Q^A(s_t, a))))$$

Note that if the other table is chosen, $Q^B(s_t, a_t)$ is updated with the same equation as above where $Q^A$ and $Q^B$ interchanged. This is called \textit{double Q-learning} and it has similar convergence conditions to the original Q-learning algorithm without overestimating the Q-values (van Hasselt, 2010).

### 3.2.2 Deep Q-Networks

The concept of Q-learning can be adapted such that a neural network can be trained to approximate the Q-function. This is useful because deep neural networks can learn a low dimensional encoding for high dimensional data (Arulkumaran et al., 2017) which reduces the state space of the Q-function. A neural network that is trained for Q-learning is called a Deep Q-Network (DQN) (Mnih et al., 2013) and can be used to automatically play Atari games (Mnih et al., 2013) and StarCraft II (Vinyals et al., 2017) for instance.

Unlike the table-based Q-learning, a Q-network is trained to learn the Q-values of each action given the state: $Q : S \rightarrow \mathbb{R}^{|\mathcal{A}|}$. This is purely a semantic difference, because it learns to predict a row from the Q-table instead of a cell. However, if the Q-network work be trained to learn $Q : S \times \mathcal{A} \rightarrow \mathbb{R}$, the Q-value would have to be recalculated for every $a \in \mathcal{A}$ to find the best action. The new formulation calculates all Q-values at once. Afterwards, it only requires a pass through the vector to find the index of the largest value which corresponds to the action to be taken.

Another difference in the Q-function in a DQN is that it does not keep track of the old Q-values. Instead, it updates

$$Q(s_t, a_t) = r_t + \gamma \cdot \max_a(Q(s_{t+1}, a))$$
This is because the neural network can directly approximate the true value of the expected future reward, whereas the Q-table tries to store which action would give the highest expected future reward.

The Deep Q-Network is trained from example transitions that the agent encountered when exploring the problem space. This is called experience replay. The transitions are temporarily stored in a buffer, called the replay memory. The replay memory is used to randomly sample a batch of transitions to learn from. The size of the memory is limited and if it is full, new transitions replace the oldest transitions. In every training loop, a single transition is observed through simulation and added to the replay memory. Then, a batch is selected to train the network on. An extension of this is the prioritized replay memory proposed by Schaul et al. (2015). It takes into account how well the DQN has learned each transition. If the training loss of a transition is high, the network can still learn a lot from it, so it should be sampled with a higher probability. This introduces an intended sampling bias that can be alleviated by weighting the resulting training loss accordingly. These weights are called importance sampling weights.

The architecture of the DQN depends on the states that need to be transformed into a Q-value vector. If the states are relatively small, the DQN can be created from fully connected layers (e.g. in Herbert and Sengupta, 2018). If the DQN states are images, they can best be represented with convolutional layers (e.g. in Mnih et al., 2013) and if the state space is a sequence or has a variable size, it can best be represented in a recurrent neural network (Hausknecht and Stone, 2015; Sorokin et al., 2015). The DQN basically needs to learn a suitable low dimensional representation such that it can predict the Q-values.

Just like normal Q-learning, the DQN will overestimate the Q-value which can also be alleviated with a double version, a Double Deep Q-Network (DDQN) (van Hasselt et al., 2016) by keeping track of an old version of the network and using that the estimate the Q-value of the best next action (w.r.t. the current DDQN version).

Another disadvantage of the original DQN design is that it needs to predict the Q-value from a single representation. But the Q-value should consist of two components, the quality of the state it is in (the value) and the quality of taking an action in the current state (the advantage). A Dueling network (Wang et al., 2015) replaces the last layer of the DQN (or DDQN) by two fully connected layers that are in parallel (as opposed to sequential). One layer predicts the value function, $V: \mathcal{S} \to \mathbb{R}$ and the other predicts the advantage function, $Adv: \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. The output is calculated from

$$Q(s, a) = V(s) + (A(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}} A(s, a')) \quad (3.1)$$

If the reinforcement learning problem can be subdivided into a hierarchy of performing different tasks, it could be learned in a hierarchical Deep Q-Network (h-DQN). This is essentially multiple DQNs stacked together. The highest DQN takes the input and decides on a sub-goal to achieve, then another DQN decides on how to achieve said sub-goal. If the full task is too complicated to learn from trial-and-error with simple actions, then explicitly structuring the task into a hierarchy can
help improve performance.

The original DQN training procedure loop contains a single simulation step (usually on the CPU) followed by a single training step (usually on the GPU). This means that the simulation needs to wait for the training and vice versa. It would be faster if the simulation and the training loops were on different threads, such that they do not have to wait for each other as much. This can be done with an asynchronous Deep Q-Network (Mnih et al., 2016) and it has the added benefit that it can have multiple simulators, each using a different exploration policy with different parameters.

Another extension of deep Q-learning is n-step Q-learning. Instead of calculating the rewards per single step and updating the neural network accordingly, it simulates multiple steps and calculates a more accurate Q-value for each transition based to the actually received reward in the future steps:

\[
R_t = r_t + \gamma R_{t+1}
\]

\[
R_n = \begin{cases} 
0, & \text{if } s_n \text{ is terminal} \\
\max_a Q(s_n, a), & \text{otherwise}
\end{cases}
\]

\[
Q(s_t, a_t) = R_t
\]

where \(R_t\) is the observed accumulated reward over the next \(n - t\) steps, \(r_t\) is the observed reward at step \(t\) and \(\max_a Q(s_n, a)\) is accumulated reward predicted from the neural network. Asynchronous n-step Q-learning has been shown to stabilize training and to outperform asynchronous one-step Q-learning on Atari games (Mnih et al., 2016).

An extensive overview of deep reinforcement learning techniques can be found in Arulkumaran et al. (2017).
Chapter 4

Methods

In our research, we apply a Deep Q-Network (DQN) to find a solution for the constrained CNOT circuit re-synthesis problem. To do this, we need to redefine the problem as a Reinforcement Learning (RL) problem and train an agent to find a solution.

However, our approach to constrained CNOT circuit re-synthesis makes three assumptions about the given circuit and the implementation of the quantum computer. First of all, we ignore the direction of the CNOT gates as they are physically implemented on the quantum computer. We can do this because the direction of a CNOT gate can be reversed with Hadamard gates and the IBM quantum computers that have this restriction have been discontinued. More recent quantum computers have a native implementation of the allowed CNOT gates in both directions. Secondly, we assume that the phase and Hadamard gates have already been optimized in the given quantum circuit. And lastly, we assume that every physical CNOT gate adds the same, constant noise when used in the quantum computer, but we do propose an extension to take this into account in section 8.2.3.

In this chapter, we describe how we designed and trained our network for constrained CNOT circuit re-synthesis. First, we give a description of the RL environment that we have used to train the agent in section 4.1. In particular, how we designed the reward function (section 4.1.1) and how the agent can be used to extract CNOT circuits from parity matrices (section 4.1.2). Then, we describe our phased training procedure (section 4.2) since the structure of the training procedure influenced the design of our neural network. The design of the neural network is described in section 4.3. Lastly, we describe how the network can be used to do CNOT circuit re-synthesis, with the focus on techniques for when the network does not find a solution in a reasonable number of steps (section 4.4).

4.1 The reinforcement learning environment

In this section, we describe how we transformed the constrained CNOT circuit re-synthesis problem into a Reinforcement Learning (RL) problem. Naively, the states in this problem are the remaining circuits to be re-synthesized and the actions are the CNOTs to be placed. However, this requires a suitable representation of the circuit as input to the DQN. Such a representation might exist, but it will be difficult
for the network to learn whether two such representations are semantically the same.

However, the problem of finding which CNOTs need to be performed to create a semantically equivalent mapped circuit is equivalent to finding which allowed rows need to be added to which rows in the parity matrix representing the CNOT circuit to obtain the identity matrix (see section 2.4.2). Using the perspective of a parity matrix, the state is simply the remaining parity matrix and the actions are allowed elementary row operations. Thus, we train a DQN to learn how to optimally reduce the parity matrix to the identity under the given connectivity constraints. This results in an optimized and constrained version of Gaussian elimination.

Learning which actions to take requires a reward to be associated to every state-action-new state tuple. The choice of reward function determines the behaviour of the neural network, so it is important to design a suitable reward function. We describe our decisions in section 4.1.1. Afterwards, we describe how we generate the initial states and how the agent traverses the environment from those states in simulation (section 4.1.2).

4.1.1 The reward function

A RL agent learns which actions to take based on the rewards it observes when traversing the environment and accumulating the expected final reward. Naturally, the nature of the reward function will cause a bias in the agent’s behaviour. For example, positive rewards will encourage the agent to take as many actions as possible. Since the Q-function accumulates the expected reward over time, the agent will learn that taking more actions means more rewards. As a result, the agent might never learn to find a solution, because it is more beneficial to stay close to the solution and collect more rewards. This can be desirable for tasks where the agent needs to stay in an equilibrium. On the other hand, negative rewards encourage the agent to find a solution as quick as possible to avoid accumulating penalties which is beneficial for optimization problems. For our purpose, we require a negative reward, because we want the agent to find a solution with a minimal number of steps.

The value of the reward should reflect the progress that the agent is taking. If the agent is rewarded for undesired actions, it will learn the wrong behaviour. However, the reward does not need reflect this perfectly. Consider the Q-function from section 3.2.2:

\[
Q(s_t, a_t) = r_t + \gamma \cdot \max_a(Q(s_{t+1}, a))
\]

The quality of an action is determined by the immediate reward \(r_t\) and the expected future reward \(\max_a(Q(s_{t+1}, a))\). As a result, the agent can learn that taking a poor action (small \(r_t\)) right now, might get the agent into a state with a better reward. A commonly used reward function is:

\[
r_t(s_t, a_t, s_{t+1}) = \begin{cases} 
1, & \text{if } s_{t+1} \text{ is final} \\
0, & \text{otherwise}
\end{cases}
\]
In which case, the agent is only rewarded if it reaches the end goal and it will learn to take the actions that will eventually reach the end goal. However, the agent will learn this by trial and error. In the cases in which it is not rewarded, the agent does not know whether it got closer to the goal or further away. For problems where it might not be clear close a particular state is to the goal, this strategy is very useful. However, if the reward can be designed such that the agent can know whether it is on the right path before it reaches a goal, it should find better policies. For constrained CNOT circuit re-synthesis, the agent should be taking a path that reduces the number of 1s in the parity matrix, bringing it closer to the identity matrix. Therefore, we could choose a negative reward that resembles that:

\[
 r_t = -\frac{-q + \sum_{i=0}^{q} \sum_{j=0}^{q} P_{ij}q^2}{q^2}
\]

where \(q\) is the number of qubits and \(P\) is the parity matrix of the new state. Note that we consider any permutation of the identity matrix to be a valid final state. Because, semantically, the permutation of the identity matrix is equivalent to moving the qubits on the quantum computer.

The steepness of the reward function describes how important it is to take the right actions. If there is no difference in the reward for a given action, the agent will not favor one action above the other and it will care more about the expected future reward. For our task, it is most important to take the right action when the agent is close to finishing and it is less important at the start, when the parity matrix is close to random. The steepness close to the solution can be adjusted by taking the exponent before negating:

\[
 r_t = -\left(\frac{-q + \sum_{i=0}^{q} \sum_{j=0}^{q} P_{ij}q^2}{q^2}\right)^\beta
\]

where \(\beta < 1\) is a chosen parameter. We empirically found that the best value was \(\beta = 0.3\).

The Q-function is in essence a partial sum of the actions to take to get to the goal. To train a neural network to approximate that sum, it is required that the Q-value is bounded. If it is not bounded, the Q-value stays changing and the network cannot keep up. If the action sequence \(\{a_t\}\) is known, we can rewrite our Q-function to the partial sum:

\[
 Q(s_t, a_t) = \sum_{\tau=0}^{t} \gamma^\tau r_\tau
\]

If the reward is bounded by \(R\) on either side and the agent takes an infinite number of steps, \(t \to \infty\), the upper-bound and lower-bound of \(Q(s_t, a_t)\) is a geometric series:

\[
 Q(s_\infty, a_\infty) = \sum_{\tau=0}^{\infty} R\gamma^\tau
\]
Chapter 4. Methods

The discount factor $\gamma$ is chosen to be positive and smaller than 1, therefore the geometric series is convergent and its sum is:

$$Q(s_\infty, a_\infty) = \sum_{\tau=0}^{\infty} R^{\tau} = \frac{R}{1 - \gamma}$$

Our reward function is designed such that $0 \geq r_t \geq -1$, and we used $\gamma = 0.9$ as discount factor. Thus, our Q-values should never be smaller than $Q(s_\infty, a_\infty) = \frac{-1}{1-0.9} = -10$ or larger than 0.

4.1.2 Extraction simulation procedure

To simulate the extraction procedure, the environment needs to generate the initial state. We did this by creating a parity matrix that corresponds to a random CNOT circuit with $n$ gates. We started with an empty circuit which corresponds to the identity matrix. Then, we randomly placed the qubits on the quantum computer’s architecture by permuting the identity matrix. Followed by randomly picking $n$ CNOT gates to be added to the circuit. For training and validation purposes, we only generated CNOT gates that are fitted to the quantum computer, but we used arbitrary CNOT gates for testing. For each of the $n$ CNOT gates, we added the corresponds rows in the permuted identity matrix, resulting in the parity matrix. This is the initial state of the environment.

For our experiments we have used 4 fictional quantum computer connectivity constraints: the 2, 3, and 4 qubit line coupling graphs and the 3 qubit circle coupling graph. We have chosen these graphs, because the 3 qubit line graph is the smallest constrained coupling graph. The other coupling graphs are used for comparison with coupling graphs with more, or less qubits (i.e. 4 qubit line, and 2 qubit line, respectively) or with less restrictions (i.e. 3 qubit circle). These graphs are visualized in Figure 4.1.

![Figure 4.1: A graphical representation of the four fictional quantum computer coupling graphs that we have used in our experiments.](image)
The quantum computer coupling graph depends on the size of the parity matrix and the number of actions. The parity matrix has the size $q \times q$ where $q$ is the number of qubits (nodes) in the graph. The number of actions are the number of CNOT gates that can be placed between qubits. This is double the amount of edges in the coupling graph, because each connection allows a CNOT gate in both directions.

When simulating, the RL agent picks an action according to a predefined set of rules, the selection policy. This action corresponds to a CNOT gate to be added at the beginning of the new circuit which is equal to adding a row in the parity matrix. Afterwards, the reward is calculated and it is checked whether the new state is a final state, i.e. whether it is a permutation of the identity matrix. The new state is stored as the current state in the environment.

### 4.2 Phased training procedure

Teaching an agent to do constrained Gaussian elimination by trial and error is not a very effective strategy. Technically, the agent can take an infinite number of actions without ever reaching the goal. Thus, we need to restrict the path that the agent takes. Unfortunately, just restricting the total number of action by some predetermined upper-bound for a random parity matrix will still result in a search space that is very large. On top of that, the path that lead to a minimal solution is very narrow. At the start of training, the agent does not know where to go and if it takes the wrong actions, it might get in a worse state or walk in circles. For example, adding the same CNOT gate for a second time in a row will negate the first gate, thus ending up at a previous state.

An advantage of this particular problem space is that the task of adding the right rows in the parity matrix can be split into simpler sub-problems. Finding the right CNOT gate to extract in a parity matrix of a fitted CNOT circuit with a single gate is simpler than extracting gates from a parity matrix of a fitted CNOT circuit with more gates. At the same time, if the agent knows how to extract a CNOT circuit with a single gate, it can use that knowledge to extract more gates. For any CNOT circuit with an arbitrary number of gates, the agent will eventually have to extract the last gate, which is equivalent to extracting gates from a circuit with a single gate.

We used this property of the problem space to restrict our training examples in phases. We first trained the network on CNOT circuits with a single gate and restricted the number of steps that the agent was allowed to take to 2. Once the agent was good enough (details below), we reset the entire training procedure, except for the network’s weights, and restarted training on CNOT circuits with an extra gate and allowing an extra step: $n_s = n_g + 1$. We continued incrementing the CNOT circuit size until the theoretical upper-bound of (unrestricted) gates (Patel et al., 2008), that can be extracted from a parity matrix, was reached: $n_g = \lceil \frac{q^2}{\log_2(q)} \rceil$. Although this upper-bound only holds for fully connected coupling graphs, we have empirically observed that this is enough to train the neural network for arbitrary
CNOT circuits. If, for future coupling graphs, this is not the case, the model can continue to be trained in the same manner by raising the upper-bound. Note that we trained the network on CNOT circuits that only have fitted CNOT gates. As a result, we know the upper-bound of the number of CNOTs to be extracted, it should not be more than the original number of gates. If we used arbitrary CNOTs, this property would not hold.

After every 1000 circuits that the agent had simulated, we validated the neural network’s performance by testing it on 10000 random CNOT circuits with the same number of gates that it was training with. We did this by simply picking the actions with the highest expected reward as predicted by the neural network. We considered the neural network “good enough” if it would extract 99% of the validation CNOT circuits within \( n_s = n_g + 1 \) steps. Whenever that happened, the network would go to the next phase, incrementing the number of gates for simulation and restarting the training procedure while keeping the weights.

Since it might be possible that the neural network would never reach the 99% validation target, we also employed a technique to try to get the neural network out of a plateau. If the network’s validation percentage would not increase for more than 10 iterations (after \( 10 \times 1000 = 10000 \) simulated circuits), we would decrease the learning rate and hope that the neural network would improve. The learning rate was decreased by multiplying it with 0.9. If the learning rate would get smaller than \( 10^{-5} \), we assume that it cannot learn anymore in the current phase and the next phase would start.

Using this training procedure and the generation procedure for the initial state (see section 4.1.2), we can make an analysis of the search space that the agent is learning in each phase. The number of possible initial states with \( q \) qubits, \( n_a \) actions and \( n_g \) gates is \( q!n_a^{n_g} \). From there, the agent can take \( n_a^{n_g} = n_a^{n_g+1} \) steps. Resulting in \( q!n_a^{n_g} \cdot n_a^{n_g+1} = q!n_a^{2n_g+1} \) different simulations that the neural network is trained with. Thus, when the number of gates, \( n_g \), is incremented when going to the next phase, the search space is multiplied by \( n_a^2 \). Therefore, the search space increases exponentially when switching phases. However, this is better than training without this phased procedure. In which case the neural network would start to learn by trial and error in a search space of \( q!n_a^{\lceil \frac{q^2}{\log_2(q)} \rceil} \) in stead of \( q!n_a^3 \).

4.3 Neural network model

We based our neural network design on the architecture used by (Herbert and Sen-gupta, 2018). They used a simple fully connected architecture with three hidden layers of size 32 with ReLU activation functions in between. Their original network was a double DQN, but we have opted for a dueling network for the CNOT circuit re-synthesis task. The dueling architecture, as explained in section 3.2.2, predicts a value of the state and the advantage for taking an action in that state separately. The two predicted numbers are then combined using equation 3.1 to make the Q-function. As opposed to the (double) DQN that predicts the Q-function directly. Note that the double Q-learning nature of calculating the prediction of the
Q-function within the dueling architecture is kept, such that the Q-values are not overestimated.

We also augmented the dueling architecture by making it asynchronous and used n-step Q-learning in stead of one-step Q-learning, since this was shown to improve training on Atari games (Mnih et al., 2016).

The advantage of using n-step Q-learning is that the number of steps in each simulation is restricted and relatively small. Therefore, the agent can attempt to extract the max amount of steps. If the agent happens to be finished, we know the exact expected future reward for each transition. Otherwise we can make an estimation of future reward on the final state by predicting it from the neural network itself and calculating the expected future rewards for each of the actions previously taken.

However, when the neural network is not yet trained, the predicted estimation might not represent the actual value. Although we do not know which exact value should be predicted, we know that the value should be negative and we can use the partial sum from section 4.1.1 as a lower-bound of the expected reward. If the predicted value does not lie between 0 and the lower-bound, we know that it is wrong. If the predicted value is wrong, we can assume that the neural network is not trained for the given state. This means that the state is new, which should
probably require more CNOT gates to extract than it is currently training on. Therefore, we can use the lower-bound as the expected reward instead. This prevents the expected rewards from exploding due to wrong estimations. In turn, this prevents the loss from exploding and thus it prevents the weights from being adjusted too much and the learning process from getting stuck.

The advantage of using asynchronous updates is that we can use separate threads to simulate the extraction procedure and generate training data, and a thread to consume the training data and train on it. This results in a speed-up because the training loop does not need to wait for the simulation loop and vice versa.

Asynchronous training also allows for multiple simulating threads that use different action selection policies, allowing more variation in the training data. We used three different action selection protocols: \( \epsilon \)-greedy, softmax and oracle. The first two are well known, but the latter is novel. Since we generate the CNOT circuits to be extracted first, we can build an action selection protocol that simply selects the next CNOT gate from the original circuit. Although this is not an optimal extraction, it is a valid one and it will generate training data that guides the neural network towards a valid extraction. The other selectors will teach the neural network to find a better path around the valid paths.

Another adjustment we did to the action selection was to the \( \epsilon \)-greedy selection. Usually, the \( \epsilon \) is slowly decreased while training. However, we used the phased training procedure to our advantage. Because of the phased training, we know that if the agent takes the right action first, it is in familiar territory and it should know the right action to take. So, we decrease \( \epsilon \) for each step taken instead.

We trained our network with 10 different simulation threads. Five of them are oracle selectors, 3 of them were \( \epsilon \)-greedy starting with \( \epsilon = 0.9 \) and decreasing it over 5 steps to a random number between 0.01 and 0.1, and the last 2 selectors used a softmax selector with a random temperature between 1 and 500. This distribution of selection policies causes that 50% of the training data is guided, 30% of the data takes random steps first and greedy steps later, and 20% of the data takes one of the better actions, as predicted by the neural network. Thus, exploiting a possible converging path with the oracle selectors, randomly exploring the outer rim of the search space with \( \epsilon \)-greedy, and learning the difference between the best two (or more) predicted actions with the softmax selector. Together, these threads should generate training data with the desired variability to train the network.

We trained our neural network by storing the simulated transitions in a replay memory with a maximum size of 10000. From this memory, we sampled a batch of 64 transitions to train every time a new transition was added to the replay memory. Every 100 simulations, i.e. episodes, we updated the target network. For our Q-function, we used a discount factor of \( \gamma = 0.9 \). We started with a learning rate of \( 10^{-3} \) which was gradually decreased until it reached below \( 10^{-5} \), as explained before. Our code can be found on GitHub\(^1\)

\(^1\)https://github.com/Quantomatic/pyzx/blob/dqn/pyzx/reinforcement_learning.py, it can be run with “python -m pyzx rl”.

---

Constrained quantum CNOT circuit re-synthesis using deep reinforcement learning  
Chapter 4. Methods
We applied *transfer learning* in a somewhat unconventional way. Instead of only transferring the weights of the hidden layer, we have also transferred parts of the input and output layers. We did this because different coupling graphs can contain overlapping structures for which the pre-trained neural network has already learned certain heuristics. To be more precise, when adding an extra qubit to the coupling graph, the pre-trained neural network has already learned the Q-values for all parity matrices where the bottom row and last column have a one where they overlap and zeroes everywhere else. Similarly, adding a new connection between qubits in the quantum computer requires the neural network to learn new shortcuts using that new connection, because the pre-trained neural network had already learned to extract without the new connection.

In order to also transfer this knowledge to the new neural network, we have mapped the weights of the input layer and output layer to the new neural network in a way that corresponds to the same parity matrix positions and CNOT locations in the coupling graph.

### 4.4 Neural CNOT circuit re-synthesis procedure

Once a network is trained, it can be used for CNOT circuit re-synthesis. By default, this is done by taking the actions with the highest Q-value, as predicted by the neural network. This is a simple process that will converge to the goal, if the neural network is properly trained.

However, if the network is not properly trained, it might get stuck and take actions indefinitely or it could extract more CNOT gates than necessary. A simple way to deal with this is by restricting the number of actions that can be taken, as we have done during training. We could use a theoretical upper-bound or use a baseline algorithm to find this max amount of steps. However, when using the network for re-synthesis, we do want the network to extract the full circuit. It is not desired if the network does not reach the goal of the permuted identity matrix.

Although it is possible to fall back on a baseline method when the network does not converge, it would mean that some circuits would not benefit from the neural network approach.

Another method is by building a tree with the possible best actions and traversing that tree depth first. If the network converges, it will find the solution in linear time. However, if it does not converge, it can backtrack and try the second best action instead.

In theory, this tree search can be done over all possible actions, but even when considering just the two best actions, this method is exponentially slower than the greedy version. If the network is poorly trained and most predicted best actions are actually bad actions, the agent needs to backtrack a lot more, resulting in a longer training time. However, we have found that in practice (see the 3 qubit line transfer results in Table 5.1b in Chapter 5), if the model is properly trained, it usually takes the best actions and it does not take a lot more time to backtrack with respect to the greedy approach.
Chapter 5

Results

We have trained several neural networks as described in Chapter 4 on a desktop computer with an Intel i7-6700K @ 4.00 GHz CPU. We did not train using a GPU because the asynchronous learning implementation required that the network was stored on the CPU. However, because our neural network is very small, the computation bottleneck was not training the neural network, but simulating the extraction procedure, so this was not a problem.

We did several experiments for different fictional quantum computer connectivity constraints: the 2, 3 and 4 qubit line coupling graphs and the 3 qubit circle (fully connected) coupling graph (as shown in Figure 4.1). We compared the performance of the neural networks versus the performance of the Steiner-Gauss algorithm from Kissinger and Meijer-van de Griend (2019). We used the naive Steiner-Gauss algorithm and the version that attempts to optimize the final qubit placement with a genetic algorithm (referred to with genetic Steiner). Note that we did not use the genetic algorithm to optimize the initial qubit placement, because our agents were not trained for this task.

We generated all possible parity matrices with the rank equal to the number of qubits to test our extraction procedure on. The rank restriction was needed because the goal state is a permuted identity matrix, which can only be obtained if the parity matrix is of full rank. This results in 6 unique $2 \times 2$ matrices, 168 unique $3 \times 3$ matrices and 20160 unique $4 \times 4$ matrices for the coupling graphs with the corresponding number of qubits. As performance metric, we chose to count the number of extracted CNOT gates. We try to minimize the amount of extract CNOTs with our algorithm. However, we do not know the minimal number of CNOT gates to be extracted from these parity matrices. Therefore, we have also tested our agent on 10000 randomly generated CNOT circuits with $1$ to $\left\lceil \frac{q^2}{\log_2(q)} \right\rceil$ gates, those results can be found in Appendix B.

In this chapter, we will first discuss the neural network’s performance versus the baselines for each of the four quantum computer coupling graphs (section 5.1). Then, we will discuss the effects of the quantum computer constraints on the training time and network performance (section 5.2). Lastly, we will discuss the effects of transfer learning on the training time and network performance (section 5.3). To make comparisons between the baselines and the different neural networks easier, we have combined the results for the different methods into four different

---

1 When implementing our algorithm, PyTorch could not share GPU tensors across processes on Windows
tables, one for each coupling graph. The results for the 2, 3 and 4 qubit line architectures and the 3 qubit circle architecture can be found in Table 5.1a, 5.1b, 5.1d and 5.1c, respectively.

Note that, for practical purposes, we limited the maximum amount of time that the 4 qubit line neural network was allowed to search for a solution with backtracking over the two best actions to 2 seconds per circuit.

### 5.1 Neural network performance

We first compare the two baseline methods, for each coupling graph. Both methods do not require training and they will always obtain a valid re-synthesized circuit. It is noteworthy that the genetic Steiner algorithm takes significantly longer than the Steiner-Gauss algorithm, but it does generate better circuits (less gates) on average, as can be seen in Table 5.1a, 5.1b, 5.1c and 5.1d.

The network for re-synthesis on the 2 qubit line coupling graph took 2 minutes to train. The trained model was able to successfully extract the test circuits from all parity maps in less than a second, as can be seen in Table 5.1a. On average, the neural network extracts the same number of CNOT gates as the genetic Steiner, but the network is able to do this much faster than the genetic Steiner algorithm. This is because the latter iteratively searches for an optimal final qubit placement, while the former has learned this and immediately extracts the right gates. The Steiner-Gauss algorithm, on the other hand, is much faster than the neural network, but it’s solution is also much worse.

The network trained for the 3 qubit line architecture took 15 minutes to train and was able to extract circuits from all parity matrices. However, unlike the 2 qubit network, this neural network performs slightly worse than the genetic Steiner baseline (see Table 5.1b). Nevertheless, the neural network does perform much better than the Steiner-Gauss algorithm at only a fraction of the run time of the genetic Steiner algorithm.

Similarly, the network trained for the 3 qubit circle architecture took 18 minutes to train, was also able to extract test circuits from all parity matrices (see Table 5.1c). However, this neural network was able to beat the genetic Steiner baseline on average, even though its longest extracted CNOT sequence was longer than the genetic Steiner one. This shows that the genetic Steiner does not necessarily find the minimal semantically equivalent CNOT circuit, since our network can find one with less gates for some parity matrices.

Unfortunately, the network trained for the 4 qubit line architecture was unable to extract all parity matrices within the maximal number of steps that the Steiner-Gauss baseline needed and within 2 seconds per parity matrix, even after training for almost 4 days (see Table 5.1d). The 5.38% of parity matrices that it was able to solve were not trivially small. Backtracking the action selection procedure and trying the second best action does improve the percentage of re-synthesized circuits to 18.67, but the performance on the backtracked circuits is worse than the genetic Steiner baseline. This shows that backtracking does not always find the best solution.
TABLE 5.1: The extracted CNOT count for all parity matrices with rank equal to the number of qubits for different coupling graphs of the Steiner-Gauss and genetic Steiner-Gauss baselines versus our approach trained from scratch and with transfer learning.

<table>
<thead>
<tr>
<th>Training time</th>
<th>% Complete</th>
<th>Testing time</th>
<th>CNOT gate count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Mean</td>
<td>Max</td>
</tr>
<tr>
<td>Steiner-Gauss</td>
<td>100</td>
<td>998 ns</td>
<td>1.5</td>
</tr>
<tr>
<td>Genetic Steiner-Gauss</td>
<td>100</td>
<td>1.11 s</td>
<td>0.67</td>
</tr>
<tr>
<td>Greedy actions (from scratch)</td>
<td>100</td>
<td>3.03 ms</td>
<td>0.67</td>
</tr>
<tr>
<td>Best 2 actions (from scratch)</td>
<td>100</td>
<td>2.00 ms</td>
<td>0.67</td>
</tr>
<tr>
<td>Greedy actions (transferred)</td>
<td>100</td>
<td>2.02 ms</td>
<td>0.67</td>
</tr>
<tr>
<td>Best 2 actions (transferred)</td>
<td>100</td>
<td>2.00 ms</td>
<td>0.67</td>
</tr>
</tbody>
</table>

(a) 2 qubit line coupling graph (6 unique parity matrices)

<table>
<thead>
<tr>
<th>Training time</th>
<th>% Complete</th>
<th>Testing time</th>
<th>CNOT gate count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Mean</td>
<td>Max</td>
</tr>
<tr>
<td>Steiner-Gauss</td>
<td>100</td>
<td>14 ms</td>
<td>5.14</td>
</tr>
<tr>
<td>Genetic Steiner-Gauss</td>
<td>100</td>
<td>38.6 s</td>
<td>2.79</td>
</tr>
<tr>
<td>Greedy actions (from scratch)</td>
<td>100</td>
<td>218 ms</td>
<td>2.98</td>
</tr>
<tr>
<td>Best 2 actions (from scratch)</td>
<td>100</td>
<td>176 ms</td>
<td>2.98</td>
</tr>
<tr>
<td>Greedy actions (transferred)</td>
<td>100</td>
<td>196 ms</td>
<td>2.79</td>
</tr>
<tr>
<td>Best 2 actions (transferred)</td>
<td>100</td>
<td>173 ms</td>
<td>2.88</td>
</tr>
</tbody>
</table>

(b) 3 qubit line coupling graph (168 unique parity matrices)

<table>
<thead>
<tr>
<th>Training time</th>
<th>% Complete</th>
<th>Testing time</th>
<th>CNOT gate count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Mean</td>
<td>Max</td>
</tr>
<tr>
<td>Steiner-Gauss</td>
<td>100</td>
<td>12 ms</td>
<td>4.07</td>
</tr>
<tr>
<td>Genetic Steiner-Gauss</td>
<td>100</td>
<td>36.8 s</td>
<td>2.07</td>
</tr>
<tr>
<td>Greedy actions (from scratch)</td>
<td>100</td>
<td>147 ms</td>
<td>2.04</td>
</tr>
<tr>
<td>Best 2 actions (from scratch)</td>
<td>100</td>
<td>119 ms</td>
<td>2.04</td>
</tr>
<tr>
<td>Greedy actions (transferred)</td>
<td>100</td>
<td>147 ms</td>
<td>2.03</td>
</tr>
<tr>
<td>Best 2 actions (transferred)</td>
<td>100</td>
<td>119 ms</td>
<td>2.03</td>
</tr>
</tbody>
</table>

(c) 3 qubit circle coupling graph (168 unique parity matrices)

<table>
<thead>
<tr>
<th>Training time</th>
<th>% Complete</th>
<th>Testing time</th>
<th>CNOT gate count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Mean</td>
<td>Max</td>
</tr>
<tr>
<td>Steiner-Gauss</td>
<td>100</td>
<td>3.09 s</td>
<td>11.30</td>
</tr>
<tr>
<td>Genetic Steiner-Gauss</td>
<td>100</td>
<td>2h 2m</td>
<td>6.54</td>
</tr>
<tr>
<td>Greedy actions (from scratch)</td>
<td>100</td>
<td>2m 54s</td>
<td>2.94</td>
</tr>
<tr>
<td>Best 2 actions (from scratch)</td>
<td>100</td>
<td>9h 9m</td>
<td>16.26</td>
</tr>
<tr>
<td>Greedy actions (without phased training)</td>
<td>100</td>
<td>9h 1s</td>
<td>0.49</td>
</tr>
<tr>
<td>Best 2 actions (without phased training)</td>
<td>100</td>
<td>11h 10m</td>
<td>10.96</td>
</tr>
<tr>
<td>Greedy actions (transferred)</td>
<td>100</td>
<td>2m 48s</td>
<td>3.40</td>
</tr>
<tr>
<td>Best 2 actions (transferred)</td>
<td>100</td>
<td>8h 29m</td>
<td>15.46</td>
</tr>
</tbody>
</table>

(d) 4 qubit line coupling graph (20160 unique parity matrices)

Aside from circuit quality, the backtracking algorithm for a poorly trained model takes too long for practical use. The reported testing times include a limitation of the testing time per circuit of 2 seconds. This means that the backtracking algorithm may have been able to re-synthesize more circuits, but that this would have taken significantly more time. We had to add this limitation, because initial experiments without limitation did not finish for 10000 circuits within 3 days for a single 4 qubit network.

To show the benefit of our phased training technique, we have also added the
performance of a neural network trained without phased training for the 4 qubit line architecture to Table 5.1d. This neural network trains significantly faster, but it is unable to find a solution for more than 99% of the parity matrices, even with backtracking. This is significantly worse in comparison with the phased trained neural network. Also note that the training time of this network is smaller than that of the phased trained network because it has less phases to train on.

All and all, these results show that, if a trained neural network can extract a given CNOT circuit, it will do this with similar circuit quality as the Steiner-Gauss algorithm, but in significantly less time.

5.2 Effects of the coupling graphs

Given the performance of the trained neural networks for the four different quantum computer architectures, we can investigate the effect of the number of qubits and the qubit connectivity on the training time and the network’s performance.

Looking at the training time of the 2, 3 and 4 qubit line neural networks (Table 5.1a, 5.1b and 5.1d), we can easily see that a network for a quantum computer with more qubits requires a longer training time. This is expected, since the size of the search space and the number of training phases increase with the number of qubits. We also observe an increase in testing time, again, this is expected because larger coupling graphs might require more steps to synthesize a circuit. To give a better idea of the difference in training behavior between all coupling graphs, a plot of the loss and validation over training episode can be found in Appendix A in Figure A.1.

Similarly, the difference in training time and performance between the networks trained for the 3 qubit line and 3 qubit circle connectivity constraints can be used to investigate the effect of qubit connectivity. More connections between the qubits requires more training time, as can be seen in Table 5.1b and 5.1c. This is expected because more connections is equivalent to more actions, which increases the search space as well. This difference is smaller with respect to the number of qubits, because the number of actions does not effect the number of training phases. The testing time, however, decreases with the number actions. This is most likely because more connections result in more semantically equivalent paths through the quantum computer architecture, thus the network can find a solution faster. Which is also reflected in the gate counts between the two networks. Again, a plot of the loss and validation of training episodes can be found in Appendix A, in Figure A.3.

5.3 Quality of transfer learning

We also experimented with the transfer learning capabilities of our trained neural networks. We have investigated transfer learning over different number of qubits and different number of actions.
For transfer learning over different number of qubits, we took the neural network trained for the 3 qubit line coupling graph and mapped the weights to a network for the 2 and 4 qubit line architectures. Their performances can be found in Table 5.1a and 5.1d, respectively. We observe that decreasing the number of qubits in the quantum computer from 3 to 2 results in a network that trains faster. In fact, the resulting network was immediately able to extract all validation circuits, as can be seen in Figure A.4b in Appendix A.

On the other hand, we observe that increasing the number of qubits in the coupling graph from 3 to 4 qubits results in a network that seems to train slower. However, we do find evidence in the number of successfully extracted circuits and the testing times that the transfer learned network is performing slightly better than the network trained from scratch (see Table 5.1d). However, the quality of those extracted circuits is slightly worse. Nevertheless, it is possible that some of the circuits, that the transfer learned network can extract, simply require more gates in the first place. Since the gate count of the transfer learned network is lower on average than that of the genetic Steiner baseline.

For transfer learning over different number of actions, we took the neural network trained for the 3 qubit line architecture and mapped the weights to a network for the 3 qubit circle architecture and vice versa. Notably, both transferred networks took longer to train than the corresponding network that was trained from scratch (see Table 5.1b and 5.1c). The transfer learned 3 qubit line network was only able to extract 98.8% of the parity matrices, but it was able to extract all by backtracking and trying the second best action instead without increasing the testing time. Both transfer learned model seems to perform slightly better than the base network that was learned from scratch. In fact, the transfer learned 3 qubit circle network slightly outperforms the genetic Steiner baseline. The training loss and validation for these networks are also plotted against each other in Appendix A: Figure A.5 for the 3 qubit line architecture and Figure A.6.
Chapter 6

Discussion

The results of our research have strong dependencies on our design decisions. In this chapter, we will discuss our Reinforcement Learning (RL) environment design (section 6.1), the effects of our phased training procedure (section 6.2), our choice to use neural networks (section 6.3), the quality of our transfer learning technique (section 6.4) and other limitations of our constrained CNOT circuit re-synthesis approach (section 6.5).

6.1 Reinforcement Learning (RL) environment design

Our choices in the design of our reinforcement learning environment directly effect the state- and action space of the reinforcement learning problem. As such, those choices will influence the performance of the neural network, because the network learns to approximate the Q-function \( Q : \mathcal{S} \rightarrow \mathbb{R}^{[A]} \). Therefore, the neural network can only directly learn from the information that is given in the state and it needs to generalize all other patterns.

We designed the environment in such a way that the state of the agent is equivalent to the circuit to be extracted. This allows the agent to learn which actions to take to eventually find a solution to the constrained CNOT circuit re-synthesis problem, but it does not give any information about the quantum computer coupling graph that it is mapping to. As a result, the neural network needs to generalize which actions would reduce the resulting circuit depth, by trial and error. If the environment state would include a representation of the quantum computer connectivity constraints, it might help the neural network to find a better approximation of the Q-function.

However, this would only be informative for the network if the coupling graph is also part of the reward function. For example, by scaling the reward with respect to the qubit distance on the quantum computer without penalizing better states. If the new state information is not reflected with a different reward, then the neural network might not learn a different expected reward function.

Since this is an initial research, we did not sufficiently experiment with the design of the environment to claim this is the best representation. In fact, we expect that adding extra state information, such as the distance between qubits, will improve the results, if done appropriately.
Chapter 6. Discussion

We also expect that the number of actions in the environment has an influence on the quality of our reward function. As described in section 4.1.1, we defined our reward function as

\[ r_t = -\left( -Q - \sum_{q_i=0} \sum_{P_{ij}} q_i^2 \beta \right) \].

The \( \beta \) determines the steepness of the reward function close to the results. Thus the agent gets a better reward if it takes an action that brings it close to the goal than if it would stay away from the goal. This is good because for random parity matrices, taking the right action is not very important, but when the agent only needs to place one gate, it needs to be the right one.

We have chosen our value for \( \beta \) based on the training behaviour of the 3 qubit line graph, but it is possible that different quantum computer connectivity constraints require a different value for \( \beta \) or a different reward function all together. Since training our neural network for the 4 qubit graph took almost 4 days, we were not able to experiment with other \( \beta \) values or completely different reward function for more qubits. But this may be a reason why the neural network trained for the 4 qubit line graph performed so poorly.

Similarly, we would not be surprises if our choice of state space and reward function influenced the transfer learning results. We will discuss this in more detail in section 6.4.

6.2 Phased training approach

Although we have shown that phased training improves the quality of the trained network, the phased training approach also has some drawbacks.

The biggest challenge with phased training is deciding when the training algorithm should go to the next phase. If the neural network trains too long on data from the same phase, it runs the risk of over-fitting on the given data, thus poorly generalizing on new data. However, if the neural network is not trained enough, it might never learn necessary features of the previous phase in the new phase, thus resulting in a poorly trained network.

The phased training approach specifically assumes that, in any phase, if the agent picks the right action first, it will end up in a state where the agent already knows the right actions to take next. However, if the agent takes the wrong action, it will stay in unknown territory and it might not return to a previously trained state. Since an agent that is trained for a coupling graph with more qubits can usually take more actions, the chance of picking the right action first is smaller. Similarly, later phases are allowed more steps, so the agent might take more steps in the wrong direction while simulating.

This required balance of the generalization ability of the neural network when switching to the next phase might be another cause of the poor performance of the neural network trained for the 4 qubit line graph. It is very possible that the network is over-fitting in the first phase, when the search space is still very small. Then, in later phases, it is not able to generalize anymore and it gets stuck in a subpar result. When the performance is stagnant during training, the training procedure will switch to the next phase, but the network is not good enough yet for
the phased training assumption. Thus, the network's performance will continue to decrease in future phases.

Similarly, it might also influence the transfer learning results, as will be discussed in section 6.4.

6.3 Use of neural networks

Although neural networks are commonly presented as a universal solution to any problem, they too have drawbacks. For example, deep learning algorithm assume that the data that it is trained on is sampled independently and that it comes from the same distribution. However, this is not strictly the case in our approach (Mnih et al., 2013). Our data is sampled from simulations, so they have a strong sequential dependency. Secondly, our phased training approach generates data from different distributions for every episode since they are generated from circuits with different number of gates.

It is also difficult to make conclusive statements about differences between neural network performance when their differences are very small. Since neural networks are randomly initialized and the training procedure is optimized by parallelization, every trained network learns a slightly different approximation. If the performance difference introduced by these random factors is larger than the effect of some adjustment in the neural network, it is impossible to attribute the performance difference to the adjustment or the randomness. This makes it very difficult to fine-tune the right hyper-parameters for a neural network and to compare neural networks with similar performances.

Our decision of neural network architecture is also a design decision that affects our results. We have used the basic structure form Herbert and Sengupta (2018) and replaced the simple double DQN with a dueling version that uses asynchronous learning. With respect to the scope of our research, this is a good starting point, but it is very possible that a different neural network architecture would have given better results.

The chosen neural network design seems to work well for the 2 and 3 qubit coupling graphs, but the poor results of the 4 qubit line neural network shows that this neural network is not scalable to circuits with more qubits. Thus, more experimentation with different neural network structures and different hyper parameters is still needed.

6.4 Transfer learning approach

Transfer learning is known for the ability to give a neural network a warm-start with respect to the weights, thus training faster and resulting in better models. However, this is not fully reflected in our results.

One reason why the transfer learned networks took longer to train might be caused by the weight transfer between the input and output layers. Usually, weights
are transferred between full layers, but for the input and output layers, we have done a partial transfer of the weights. If the pre-trained network had a different number of qubits or actions, then we still transferred a selection of the weights. Although we made sure that the weights corresponded to the same inputs and outputs, it is possible that the non-transferred weights would not be updated properly during training. The uniformly initialized non-transferred weights and the transferred weights should follow a very different distribution which the back-propagation algorithm might not be able to accommodate.

This could be alleviated by fixing the weights of the hidden layers in the neural network, such that they are not adjust during training. However, only adjusting the weights of the first and last layer might restrict the network to the point that it cannot generalize anymore to the new coupling graph.

In general, poor transfer learning results show that the transferred network does not generalize to the new task as well as could be expected. Longer training times mean that the network takes longer to learn, which can be caused by a poor starting location: The transferred weights. In our research, these weights can be unsuitable for training due to our initialization technique, as described before, but it can also mean that the features learned by the pre-trained network are not suitable for the new coupling graph.

It is possible that the pre-trained network is unsuitable because it is over-fitted for the old coupling graph. This is desirable for the original task, but when transfer learning, it could mean that the new network starts in a local optimum and the new network will never learn how to better accommodate the new training data.

In particular, the old network was trained with the phased training approach that exposes the network to a subspace of the entire search space. It would therefore not be surprising if the old network was over-fitting for original coupling graph. However, it is also possible that the original transferred weights are suitable for the new network, but that this is reverted when training the new network starting at phase 1 instead of a later phase. For example, if the old network had learned to chose actions that are sub-optimal for the new coupling graph, then this would need to be unlearned by the new network. But in doing so, the new network might unlearn patterns that it will need in later phases.

### 6.5 Other limitations

One of the main problems with this research is that we have only successfully tested it on very small coupling graphs. Because these graphs are toy examples, they do not accurately represent the current existing quantum computer architectures. Thus, these trained models are not directly applicable to existing quantum compilers. However, the techniques described in this thesis might still be used for partial extraction of quantum circuits, which will be described in more detail in section 8.1.

Nevertheless, if, in future work, a neural network does learn to do constrained CNOT circuit re-synthesis for realistic quantum computer connectivity constraints,
it’s solutions might still not be optimal. The neural network learns an approximation, which might be better than previous approaches, but it could possibly still be improved.

Secondly, it is possible that the neural network’s performance is inconsistent in the sense that it would extract CNOTs from some parity matrices more optimally than others. This kind of biased behaviour depends on the trained model and the training data.

Lastly, the current neural network structure only allows the extraction of square matrices of full rank, but for some applications, such as the circuit optimization technique by Duncan et al. (2019), it might be required that matrices with flexible number of columns is extracted using the same CNOT constraints. Our current approach uses fully connected layers that require a fixed input size, so it cannot accommodate matrices of a variable size. The network can be adapted for this by adding a recurrent layer. We will discuss this in more detail in section 8.3.
Chapter 7

Conclusion

For this master thesis, we reached the possibility to train a reinforcement learning agent to perform constrained CNOT circuit re-synthesis from a parity matrix. We used a neural network to learn a Q-function over states and actions in the reinforcement learning environment. This deep Q-network predicts for each action an approximation of the expected final reward when taking that action in a given state. Using this prediction, the reinforcement learning agent can decide which actions it should take in the current state. Taking an action results in a new state where the agent can decide again. This continues until the agent reaches the goal.

In the case of constrained CNOT circuit re-synthesis, the actions that the agent picks are the CNOT gates that it can place in a circuit that are allowed by the quantum computer connectivity constraints. The state is the parity matrix of the CNOT circuit that still needs to be synthesized to generate a semantically equivalent CNOT circuit. Once the agent has reached a state where the parity matrix is equal to a permutation of the identity matrix, it is done. The permutation describes where the final qubits are placed on the quantum computer after finishing the calculations.

We used a simple fully connected dueling neural network with 3 layers of 32 nodes and ReLU activation functions in between the layers. We experimented using four different fictional quantum computer coupling graphs. We used a 3 qubit line graph as it is the smallest graph that is not fully connected. This graph was adjusted to add or remove a qubit, resulting in the 4 and 2 qubit line graphs, respectively. We also used a 3 qubit circle graph to investigate the effect of qubit connectivity, since it is allowed two extra actions (the new CNOT gate in both directions).

We proposed a novel phased training approach to guide the training process and we have shown that this improves the network quality. We also showed that the trained networks outperformed the Steiner-Gauss baseline on the 2 and 3 qubit coupling graphs. These networks performed similar to the genetic Steiner baseline, and the neural networks trained for the 3 qubit circle graph even slightly outperformed the genetic Steiner baseline. However, the current design of the environment and neural network was not sufficient to scale our approach to a 4 qubit line coupling graph. We expect that this is due to our choice of hyper-parameters, but more research is required to make conclusive statements.

We also experimented with transfer learning to initialize most of the network weights and showed that this can improve on the networks trained from random
initialization. This shows that the trained networks can generalize towards a different coupling graph. However, this does require longer training and the observed performance improvement is rather small. This could be due to that the heuristics that the agent uses might unsuitable for other coupling graphs and the transferred agent needs to unlearn these heuristics and find new ones. This could also be a feature of the constrained CNOT circuit re-synthesis problem, but it is most likely due to over-fitting of the network.

Once we have designed a neural network to do constrained quantum CNOT circuit re-synthesis that does work for realistic quantum computer coupling graphs, the network would only need to be trained once per architecture. Such a trained network can then be used to re-synthesize aggregated clusters of CNOT gates within a full quantum circuit, such that the resulting circuit is mapped to the specified architecture.

For this master thesis, we focused on pure CNOT circuits and we only investigated the performance of the network on fictional architectures. In theory, this approach is scalable because parity matrices scale quadratically with respect to the number of qubits. However, we were not able to show that such a neural network can be trained for larger quantum computers. Nevertheless, if such a neural network can be trained, it can also be adjusted for full end-to-end circuit re-synthesis. A few approaches will be described in section 8.3.
Chapter 8

Future work

As we have made clear in the discussion and conclusion, it is necessary to experiment with more neural network architectures and hyper parameters, before we can make any general conclusions about the usability of our proposed technique. However, that can be seen as the start of a new field of research. In this chapter, we will describe three directions in which our research can be extended. First of all, we could apply our current approach to larger coupling graphs by using partially trained neural networks (section 8.1). Secondly, we could add new features to the neural network, such that the trained agent would synthesize better CNOT circuits (section 8.2). And lastly, we could extend the current approach to be able to do full constrained quantum circuit re-synthesis (section 8.3).

8.1 Using partially trained networks

We have shown in our results that, in its current form, our proposed technique only works for trivial quantum computer coupling graphs. However, this does not mean that these trained neural networks could not be used for quantum computer with other coupling graphs. In fact, we suggest three possible uses of our constrained CNOT circuit re-synthesis technique that might be directly usable in quantum circuit mapping techniques.

First of all, we suggest to use the two 3 qubit neural networks for the extraction of partial CNOT circuits. Larger quantum computer coupling graphs can be cut into pieces of 3 connected qubits that either form a line or a circle. Similarly, we can cut a CNOT circuit into partial CNOT circuits of two types: (1) collections of CNOTs that are already mapped or one qubit away on the quantum computer architecture, and (2) other partial CNOT circuits. The former can be solved and optimized using our neural network and the latter can be mapped with previously existing techniques, such as swapping and the Steiner-Gauss algorithm. Although this will most likely result in worse solutions with respect to the pure genetic Steiner algorithm, there might be a specific class of quantum algorithms that would benefit from this type of mapping. For example, the Toffoli gate over 3 qubits is decomposed into single CNOT gates with rotations in between. If these 3 qubits are placed on the quantum computer such that they form a line, our neural network can be used to map it to the quantum computer architecture.
Secondly, our 3 qubit neural networks could be used in a way that is similar to the swapping approach, but with moving 3 qubits over the architecture at a time, instead of 2. Adding a single SWAP gate to the circuit is equivalent to adding a sequence of CNOT gates which parity matrix is equivalent to a SWAP. Similarly, our neural network can be used to add sequences of CNOT gates that relocate 3 qubits to a specified location. This would allow a SWAP-based approach to consider routing more qubits at a time.

Lastly, it is most likely possible to train our neural network for shallow circuits (with little gates) on larger coupling graphs. We did not explicitly discuss this in our results, but in the validation plots in Figure A.1b, we can see that the neural networks are able to learn strategies for extracting CNOT circuits with a small number of gates, even for the 4 qubit graph. Although the trained models on such a small number of gates is not able to re-synthesize arbitrary CNOT circuits, it might still be useful. In practice, the sequences of CNOT gates that occur in realistic quantum circuits are usually quite shallow. Again, take the Toffoli gate as an example, the decomposed version of this gate contains sequences of only a single CNOT gate between the rotations. Thus a neural network that is trained for such shallow circuits on a larger coupling graph, would be able to map these CNOTs nonetheless. Secondly, this type of shallow circuit could be combined with the fully trained 3 qubit neural networks as described above, instead of SWAP gates or Steiner-Gauss. Similarly, such a shallow network could also be used to iteratively map a few CNOT gates in the entire sequence of CNOTs to be mapped. However, the merit of these usages still need to be investigated.

8.2 Extensions

Aside from directly applying our technique as a partial solution to the quantum circuit mapping problem, we could also try to extend our approach to have certain desirable properties. First of all, we will describe how the neural network design could be extended to also propose a good initial qubit placement (section 8.2.1). Secondly, our current neural network does not take the possibility of parallel actions into account. We propose the novel idea of a parallel Q-function to avoid an exponential action space in section 8.2.2. Lastly, we suggest to research the use of error estimation as part of the reward function to better reflect the true quality of the synthesized circuit.

8.2.1 Initial qubit placement

Our current research did not take the initial qubit placement on the quantum computer into account. As briefly discussed in section 2.4, the choice of placing the qubits onto the coupling graph directly affects the possible solutions to be found with our algorithm. Currently, we naively placed the qubits on the quantum computer, but we suggest three possible approaches to find a more suitable initial qubit placement.
Chapter 8. Future work

First of all, we could approximate the initial qubit placement using our constrained CNOT circuit re-synthesis on the leading CNOT gates in the circuit. Our technique will reduce the parity matrix to a permutation of the identity matrix. This permutation describes the new qubit locations on the quantum computer. As such, we can remove all leading CNOT gates from the circuit and start with the qubits in their new locations. Unfortunately, realistic quantum programs will not have any leading CNOT gates in the circuit for this very reason.

Another approach is inspired by the reverse traversal technique from (Li et al., 2018) and it is based on the fact that quantum circuits are reversible. If a quantum circuit mapping technique allows the movement of qubits across the quantum computer such that it results in some final qubit placement, then this final qubit placement can be used as initial qubit placement for mapping the reversed circuit. Thus, a pass of the mapping algorithm on the reversed circuit will again obtain a final qubit placement which can be used as the initial qubit placement of the original circuit (the reversed, reversed circuit). This reversed traversal technique can be continued until the mapping algorithm does not improve the final circuit anymore, obtaining the final mapping. We propose that the reversed traversal can also be used in a Particle Swarm Optimization (PSO) algorithm. A PSO starts with a number of random starting points (a particle swarm) and each iteration the points (particles) are adjusted to find an optimum.

We suggest to approximate the qubit allocation problem by generating a collection of random qubit placements (particles) and adjusting them using the reversed traversal technique. If this results in a better circuit, the particle is updated with the better qubit placement, otherwise it randomly adjusted with a mutation, as is known from a genetic algorithm. Such a mutation could, for example, be swapping two random qubits in the initial qubit placement. This PSO has successfully been implemented and can be found on GitHub\(^1\), but it was not used for our research.

Lastly, we could adjust our neural network to find the initial qubit placement before picking which gates should be added to the re-synthesized circuit. This can be done with a hierarchical DQN (Kulkarni et al., 2016), a type of DQN that uses a hierarchy of tasks for the agent to learn. For qubit allocation, we suggest to split the constrained CNOT circuit re-synthesis task into two sub-tasks: placing the qubits and picking the CNOT gates. The part of the h-DQN that picks the gates can be equivalent to the neural network that we have used in our research. The part that places the qubits will be new. We suggest this part of the neural network predicts a sequence that is also a permutation given the parity matrix. The predicted permutation is then used to permute the parity matrix which is then given as input to the second neural network to extract the CNOT gates. The reward of the qubit placing neural network should correspond to the number of actions taken by the gate picking agent. The neural network can predict a permutation by predicting which qubits should be placed in the next available slot, given the parity matrix and the previously placed qubits. Such a complicated neural network architecture will require a significant amount of time to find the right hyper-parameters, so it is left

\(^1\)https://github.com/Quantomatic/pyzx/blob/steiner_decomp/pyzx/scripts/circuit_router.py
for future work.

### 8.2.2 Parallel action taking

A significant drawback in our approach is that it does not take parallel CNOT gates into account. Since the current day qubit decoherence times are very low, the quality of quantum circuits is not only determined by the number of gates in the circuit, but also by how many gates can be placed in parallel. For simplicity, we did not take this into account when designing our neural network, but ideally it should. To this end, we suggest a novel technique to allow parallel action taking by a deep Q-learning agent.

The main problem of parallel actions in Q-learning is that actions are discounted over time. Thus, aggregating actions that could be taken in parallel will not represent the expected reward adequately. However, simply redefining the set of actions as the set of all actions that can be taken in parallel will result in an exponentially large action space which is not scalable. This problem was also raised by Herbert and Sengupta (2018) in the SWAP finding DQN and addressed by making the Q-function into a value function for the state and selecting actions based on which action results in the better state.

Instead, we propose to implement parallel action taking as part of the discount factor. The original discount factor discounts all future actions, but this is not desired for actions that can be taken in parallel. We suggest a conditional discount factor that only discounts if the next action cannot be taken together with the set of previous actions:

\[
\gamma(a_{prev}, a) = \begin{cases} 
0, & \text{if } a_{prev} \text{ is not a valid set of parallel actions} \\
0, & \text{if } a \text{ can be taken in parallel with } a_{prev} \\
\gamma, & \text{otherwise}
\end{cases}
\]

where \(\gamma\) is a scalar like the original discount factor.

However, in the Q-function, we need to know the next action to take in order to calculate the expected future reward. This can be found by looping over all possible future parallel actions and picking the best set:

\[
Q(s_t, a_t, a_{prev}) = r_t + \max_{s_{a_x} \in \mathcal{P}(A)} \left( \sum_{i=0}^{\left|s^a\right|} \gamma(a'_{prev}, s^a_i) \cdot Q(s_{i+t+1}, s^a_i, \emptyset) \right)
\]

where \(a'_{prev}\) is the set consisting of \(a_{prev}\), \(a_t\) and \(\{s^a_j \mid j \in [\ldots i]\}\). Note that the sum in our parallel Q-function will be a sum of zeroes except when the previously taken actions \(a'_{prev}\) can be taken in parallel, but not together with \(s^a_i\). In that case, the set of previous actions is empty, resulting in the \(\emptyset\) in the equation. Also note that this Q-function can be optimized by only considering \(s^a \in \mathcal{P}(A)\) such that \(a_{prev}\) is a valid set of parallel actions. Unfortunately, implementing this parallel Q-function in our technique would not have been able to show whether this parallel Q-function would work in general, since we only investigated trivial coupling graphs.
8.2.3 Error estimation

Another improvement to our technique is a reward function that better represents the expected error in the resulting circuit. We optimize the number of gates because gates introduce small errors in the result which we want to minimize. However, it might be possible that in a quantum computer some CNOT gates are more noisy than others. In that case, we would prefer to use the less noisy gates over the more noisy ones. We can teach our reinforcement learning agent that more noisy gates are less preferred by adding the gate fidelities to the environment state and to reflect their error rates in the reward function. Ideally, we would like to build an error expectation approximation algorithm that can be used in the reward function such that our RL agent does not just reduce the number of gates in the circuit, but it reduces the error introduced when calculating the gates.

A very powerful property of using neural networks for this problem is that the RL agent should, in theory, be able to learn heuristics for which gates to extract given the gate fidelities even when the gate quality changes over time. If it is possible to monitor or estimate the quality of the gates in the quantum computer, then the circuit can be optimized for those specific values in real time.

However, this will require a lot of knowledge about the exact functioning of the quantum computer in question and it might not be possible to make a suitable prediction of the quality of the final result. Besides, this extension is only fruitful if we were able to train a neural network on realistic quantum computer coupling graphs which still requires more research. Nevertheless, this would be a powerful application of our neural constrained CNOT circuit re-synthesis technique.

8.3 Full quantum circuit re-synthesis

In this research, we only focused on the constrained re-synthesis of quantum circuit containing just CNOT circuits. In this section, we will describe how our algorithm can be used for the constrained re-synthesis of arbitrary quantum circuits by integrating it into existing algorithms (section 8.3.1) or by adjusting the neural network to accommodate all gates (section 8.3.2). For this we assume that a suitable RL agent can be trained for the required quantum computer.

8.3.1 Integration in other algorithms

We can use our CNOT circuit re-synthesis technique for the mapping and optimization of arbitrary quantum circuits by integrating it into established quantum circuit optimization techniques that use Gaussian elimination for the synthesis of CNOT circuits by simply replacing the Gaussian elimination algorithm with our RL agent. An example of would be is the T-par algorithm by Amy et al. (2014).

Similarly, we could add our RL agent to the optimization algorithm proposed by Duncan et al. (2019). However, this would require a slight adjustment to our current neural network. As discussed in section 6.5, our technique is currently only...
able to extract CNOT circuits from square parity matrices, but the algorithm proposed by Duncan et al. (2019) also extracts CNOTs from parity matrices with more columns than rows. In our proposed technique, it is possible to adjust the neural network such that it can handle parity matrices with arbitrary number of columns using a Recurrent Neural Network (RNN). Such a network is able to make a condensed representation of sequences of arbitrary lengths once trained. We suspect that a suitable RNN can be designed to learn the Q-function for the RL agent, but this requires more research.

Another way to combine our technique with the algorithm proposed by Duncan et al. (2019) is to use our technique for post-processing the circuit. Namely, using the Gaussian elimination procedure to generate the initial CNOT gates and then apply our neural network to the CNOT sequences between the rotations in the synthesized circuit. However, the strength of the intermediate representation of their algorithm lies in picking which rotations should be applied first. This can also be added as actions of our RL agent, as will be described next.

### 8.3.2 Neural full circuit re-synthesis

We could also extend our constrained CNOT circuit re-synthesis agent to be able to extract full quantum circuits. To do this, we require a representation of the quantum circuit to be extracted that can be used as a state for our neural network. Niepert et al. (2016) described a method for creating a graph representation to be used in a Convolutional Neural Network (CNN). It was meant for use in classification and regression problems, but it might also be applicable to a Deep Q-Network (DQN).

A possible start of such a quantum circuit representation is the graph-like ZX-diagram described by Duncan et al. (2019). We refer the unfamiliar reader to the original paper for a detailed explanation of their algorithm and terminology. A graph-like ZX-diagram can be represented with two matrices: The parity matrix that is the frontier, and the connectivity matrix that describes the remainder of the diagram. The first matrix has a flexible number of columns. The second has both a variable number of rows and columns. Therefore, a neural network that is able to describe states in such a format would be a combination of a RNN and a nested RNN.

The actions in this reinforcement learning problem are again the CNOT gates that can be placed according to the parity matrix. But this can also be extended with the phase gate in the frontier that can be placed in the circuit next. Otherwise this would be done automatically whenever a phase becomes available. It is also possible to describe this problem with a hierarchical DQN, where the task of picking which phase gates to place next is a sub-task to achieve with our proposed RL agent.

An interesting benefit of such using a graph-like ZX-diagram to represent the quantum circuit is that it should theoretically be possible to teach a RL agent to extract any kind of quantum gate. For example, the set of gates directly implemented by the physical quantum computer. These will be the actions of the agent. If the
Chapter 8. Future work

A graph-like ZX-diagram is completely described by the state, then applying the extracted gate to the diagram would result in a new state. Thus, actions can be taken until the resulting state is trivial which corresponds to the empty circuit.

This is a very powerful benefit of using neural networks for quantum circuit mapping, because the physical implementation of quantum computers will most likely change as better quantum computers are developed. Since quantum computing is such a relatively young field, it should be desirable to have a general quantum circuit mapping technique that requires the least amount of assumptions on the quantum computer architecture. Such that we do not need to go back to the drawing board when new quantum computers have a different implemented set of basic gates or other constraints. Instead, we would only need to define a new reinforcement learning environment, possibly adjust the neural network design, and train a new RL agent to find new heuristics automatically.
Bibliography


Appendix A

Training loss and validation plots

This appendix is intended to show the training progression of our different neural networks. We show plots of the calculated loss and the percentage of correctly extracted validation circuits for the different coupling graphs. Below, we will explain how these values are calculated and we describe some visible artifacts in the plots that are caused by the phased training approach.

To compare the difference between training the neural network for coupling graphs for different number of qubits, we show the plots in Figure A.1. However, since training for the 4 qubit line graph takes much longer than the other connectivity constraints, we show the zoomed in plot, without the 4 qubit line results, in Figure A.2.

Similarly, to compare the effect of different connectivity constraint in quantum computers with the same number of qubits, we show a plot with the training performance of the neural networks for the 3 qubit line and circle coupling graph in Figure A.3.

We also created some figures to compare the difference in training behaviour for neural networks trained from scratch and trained with transfer learning. Figure A.4 shows this for the 2 qubit line graph and Figure A.7 shows this for the 4 qubit line graph. We also show the effects of transfer learning from the 3 qubit line to the 3 qubit circle graph and vice versa, in Figure A.6 and A.5.

Figure A.7 also shows the training behaviour of the neural network trained without phased training, such that it can be directly compared to the 4 qubit line network trained from scratch and with transfer learning. Again, since the neural network without phased training trained a lot faster than the other 4 qubit line networks, we have also created a separate plot containing only the loss and validation for the network trained without phased training.

The loss is a number describing the distance between the values predicted by the neural network and the simulated ground truth. This number is used to update the weights in an attempt to decrease the loss through back-propagation.

We used the Hubert loss, also known as the Smooth L1 loss, defined by:

\[ \text{loss}(x, y) = \frac{1}{n} \sum_{i} z_i \]
where $z_i$ is given by:

$$
z_i = \begin{cases} 
0.5(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\
|x_i - y_i| - 0.5, & \text{otherwise}
\end{cases}
$$

Because the simulated state transitions that the neural networks is trained on can vary significantly, the loss is relatively unstable, which can be seen in the plots.

For validation of the neural network performance, we generate a validation set of 1000 CNOT circuits in the same way as the initial training circuits. Then, we used the greedy action taking approach to try to find a solution within the same number of steps that is allowed during training. From that, we calculate the percentage of CNOT circuits that are extracted within the specified number of steps, as described in section 4.2. This is the validation value that is plotted in all figures in this appendix. For the validation percentage, a higher score is better.

Since we used a phased training approach, as described in section 4.2, the neural network is exposed to simulations obtained from different distributions during training. The switch from one phase to the next is clearly visible in both the loss and the validation in most figures.

When the neural network's validation reaches above 99%, the training phase is increased and the neural network is trained on a larger dataset. Because this new dataset contains previously unseen transitions, the neural network has not learned their corresponding value and mispredicts that value. Therefore, the loss suddenly increases. Similarly, the neural network is also validated on previously unseen CNOT circuits. Thus if the network was not able to generalize over the previous phase, it is unable to find a solution and the validation percentage decreases. While training, the network learns to extract more CNOT circuits and the validation percentage goes up, either until the 99% threshold is reached or the validation percentage hits a plateau.

![Figure A.1: The progression of the training loss (A.1a) and the amount of extracted circuits (A.1b) over simulations seen (episodes) for the 2 qubit, 3 qubit and 4 qubit line graphs.](image)
**Appendix A. Training loss and validation plots**

**Figure A.2:** The progression of the training loss (A.2a) and the amount of extracted circuits (A.2b) over simulations seen (episodes) for the 2 qubit and 3 qubit line graphs.

**Figure A.3:** The progression of the training loss (A.3a) and the amount of extracted circuits (A.3b) over simulations seen (episodes) for the 3 qubit line and circle graphs.

**Figure A.4:** The progression of the training loss (A.4a) and the amount of extracted circuits (A.4b) over simulations seen (episodes) for the 2 qubit line graph trained from scratch versus the network transfer learned from a network trained for the 3 qubit line graph.
Appendix A. Training loss and validation plots

**Figure A.5**: The progression of the training loss (A.5a) and the amount of extracted circuits (A.5b) over simulations seen (episodes) for the 3 qubit line graph trained from scratch versus the network transfer learned from a network trained for the 3 qubit circle graph.

**Figure A.6**: The progression of the training loss (A.6a) and the amount of extracted circuits (A.6b) over simulations seen (episodes) for the 3 qubit circle graph trained from scratch versus the network transfer learned from a network trained for the 3 qubit line graph.
**Figure A.7**: The progression of the training loss (A.7a) and the amount of extracted circuits (A.7b) over simulations seen (episodes) for the 4 qubit line graph trained from scratch with and without phased training versus the network transfer learned from a network trained for the 3 qubit line graph.

**Figure A.8**: The progression of the training loss (A.8a) and the amount of extracted circuits (A.8b) over simulations seen (episodes) for the 4 qubit line graph trained without phased training.
Appendix B

Results on random CNOT circuits

This appendix contains the performance results of our neural networks on 10000 random CNOT circuits. Many of these CNOT circuits will correspond to the same parity matrices, as there are only 6, 168 and 20160 unique parity matrices for coupling graphs with 2, 3, or 4 qubits. However, from the parity matrix itself, we do not know the length of the CNOT sequence to be extracted. Instead, by generating the CNOT circuits, we can use the length of the original generated CNOT circuit to calculate to calculate percentage of added CNOT gates with respect to the original circuit.

As performance metric, we use the percentage in overhead with respect to the original CNOT gate count:

\[
f(c_{\text{old}}, c_{\text{new}}) = \frac{|c_{\text{new}}| - |c_{\text{old}}|}{|c_{\text{old}}|} \cdot 100\%\]

where \(c_{\text{old}}\) and \(c_{\text{new}}\) are the old and new circuit, respectively, and we use \(|\cdot|\) to indicate the number of CNOT gates in the circuit. We use the percentage, as opposed to the pure gate count, because the test dataset consists of 10000 random CNOT circuits with 1 to \(\lceil \frac{q^2}{\log_2(q)} \rceil\) gates.

Table B.1 shows that the results discussed in Chapter 5 follow a similar pattern in overhead percentage. It is noteworthy that the 4 qubit line neural networks take much more steps than needed, specifically when backtracking over the 2 best actions (see Table B.1d). This shows that the extra circuits that were extracted with the backtracking were found by accident.
TABLE B.1: The CNOT circuit re-synthesis overhead (%) on 10000 CNOT circuits for different coupling graphs of the Steiner-Gauss and genetic Steiner-Gauss baselines versus our approach trained from scratch and trained with transfer learning.

<table>
<thead>
<tr>
<th></th>
<th>Training time</th>
<th>% Complete</th>
<th>Testing time</th>
<th>Overhead (%)</th>
<th>Mean</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Steiner-Gauss</strong></td>
<td>-</td>
<td>100</td>
<td>540 ms</td>
<td>-22.53</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>100</td>
<td>43m 54s</td>
<td>-59.04</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Greedy actions (from scratch)</td>
<td>2 m</td>
<td>100</td>
<td>4.06 s</td>
<td>-59.04</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Best 2 actions (from scratch)</td>
<td>100</td>
<td>4.26 s</td>
<td>-59.04</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greedy actions (transferred)</td>
<td>1m 40s</td>
<td>100</td>
<td>4.18 s</td>
<td>-59.04</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Best 2 actions (transferred)</td>
<td>100</td>
<td>3.96 s</td>
<td>-59.04</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(A) 2 qubit line coupling graph

<table>
<thead>
<tr>
<th></th>
<th>Training time</th>
<th>% Complete</th>
<th>Testing time</th>
<th>Overhead (%)</th>
<th>Mean</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Steiner-Gauss</strong></td>
<td>-</td>
<td>100</td>
<td>1m 23s</td>
<td>57.11</td>
<td>800</td>
<td>300</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>100</td>
<td>55m 4s</td>
<td>-25.8</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>Greedy actions (from scratch)</td>
<td>15 m</td>
<td>100</td>
<td>17.51 s</td>
<td>-22.12</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>Best 2 actions (from scratch)</td>
<td>100</td>
<td>12.26 s</td>
<td>-22.12</td>
<td>400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greedy actions (transferred)</td>
<td>4h</td>
<td>99.1</td>
<td>15.73 s</td>
<td>-25.61</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>Best 2 actions (transferred)</td>
<td>100</td>
<td>12.60 s</td>
<td>-24.40</td>
<td>400</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(B) 3 qubit line coupling graph

<table>
<thead>
<tr>
<th></th>
<th>Training time</th>
<th>% Complete</th>
<th>Testing time</th>
<th>Overhead (%)</th>
<th>Mean</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Steiner-Gauss</strong></td>
<td>-</td>
<td>100</td>
<td>1.13 s</td>
<td>23.26</td>
<td>600</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>100</td>
<td>52m 30s</td>
<td>-50.58</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Greedy actions (from scratch)</td>
<td>18 m</td>
<td>100</td>
<td>12.38 s</td>
<td>-51.21</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Best 2 actions (from scratch)</td>
<td>100</td>
<td>8.89 s</td>
<td>-51.21</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greedy actions (transferred)</td>
<td>1h2m</td>
<td>99.1</td>
<td>12.28 s</td>
<td>-51.26</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Best 2 actions (transferred)</td>
<td>100</td>
<td>9.14 s</td>
<td>-51.26</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(c) 3 qubit circle coupling graph

<table>
<thead>
<tr>
<th></th>
<th>Training time</th>
<th>% Complete</th>
<th>Testing time</th>
<th>Overhead (%)</th>
<th>Mean</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Steiner-Gauss</strong></td>
<td>-</td>
<td>100</td>
<td>1.62 s</td>
<td>127.90</td>
<td>1700</td>
<td>1700</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>100</td>
<td>1h 8m</td>
<td>0.44</td>
<td>600</td>
<td></td>
</tr>
<tr>
<td>Greedy actions (from scratch)</td>
<td>3d 20h</td>
<td>14.76</td>
<td>1m 29s</td>
<td>-9.95</td>
<td>700</td>
<td></td>
</tr>
<tr>
<td>Best 2 actions (from scratch)</td>
<td>3h 58m</td>
<td>28.98</td>
<td>3h 58m</td>
<td>155.91</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>Greedy actions (without phased training)</td>
<td>14h 38m</td>
<td>1.8</td>
<td>1m 41s</td>
<td>-66.59</td>
<td>0</td>
<td>2000</td>
</tr>
<tr>
<td>Best 2 actions (without phased training)</td>
<td>5h 24m</td>
<td>2.66</td>
<td>5h 24m</td>
<td>411.39</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>Greedy actions (transferred)</td>
<td>3d 22h</td>
<td>18.52</td>
<td>1m 28s</td>
<td>-8.41</td>
<td>900</td>
<td>2000</td>
</tr>
<tr>
<td>Best 2 actions (transferred)</td>
<td>3h 32m</td>
<td>36.89</td>
<td>3h 32m</td>
<td>177.87</td>
<td>2000</td>
<td></td>
</tr>
</tbody>
</table>

(d) 4 qubit line coupling graph
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$-greedy exploration</td>
<td>19</td>
</tr>
<tr>
<td>action</td>
<td>18</td>
</tr>
<tr>
<td>activation function</td>
<td>17</td>
</tr>
<tr>
<td>advantage</td>
<td>21</td>
</tr>
<tr>
<td>agent</td>
<td>5, 18</td>
</tr>
<tr>
<td>AI; see artificial intelligence</td>
<td>1, 4</td>
</tr>
<tr>
<td>artificial intelligence</td>
<td>1, 4</td>
</tr>
<tr>
<td>$\epsilon$-greedy exploration</td>
<td>19</td>
</tr>
<tr>
<td>action</td>
<td>18</td>
</tr>
<tr>
<td>activation function</td>
<td>17</td>
</tr>
<tr>
<td>advantage</td>
<td>21</td>
</tr>
<tr>
<td>agent</td>
<td>5, 18</td>
</tr>
<tr>
<td>artificial intelligence</td>
<td>4</td>
</tr>
<tr>
<td>asynchronous deep q-network</td>
<td>22</td>
</tr>
<tr>
<td>back-propagation</td>
<td>17, 40, 55</td>
</tr>
<tr>
<td>batch</td>
<td>18, 30</td>
</tr>
<tr>
<td>classification</td>
<td>15</td>
</tr>
<tr>
<td>convolutional layer</td>
<td>16</td>
</tr>
<tr>
<td>convolutional neural network</td>
<td>16, 49</td>
</tr>
<tr>
<td>cross entropy</td>
<td>17</td>
</tr>
<tr>
<td>deep neural network</td>
<td>17</td>
</tr>
<tr>
<td>deep q-network</td>
<td>2, 3, 20, 23, 49</td>
</tr>
<tr>
<td>discount factor</td>
<td>20, 30</td>
</tr>
<tr>
<td>double deep q-network</td>
<td>12, 21</td>
</tr>
<tr>
<td>double q-learning</td>
<td>20</td>
</tr>
<tr>
<td>dueling network</td>
<td>21</td>
</tr>
<tr>
<td>environment</td>
<td>4, 5, 18, 37</td>
</tr>
<tr>
<td>episode</td>
<td>30, 56–59</td>
</tr>
<tr>
<td>experience replay</td>
<td>21</td>
</tr>
<tr>
<td>exploitation</td>
<td>19</td>
</tr>
<tr>
<td>exploration</td>
<td>19</td>
</tr>
<tr>
<td>exploration policy</td>
<td>19</td>
</tr>
<tr>
<td>fully connected layer</td>
<td>16</td>
</tr>
<tr>
<td>gated recurrent unit</td>
<td>16</td>
</tr>
<tr>
<td>gradient descent</td>
<td>17</td>
</tr>
<tr>
<td>ground truth</td>
<td>17</td>
</tr>
<tr>
<td>hidden layer</td>
<td>16</td>
</tr>
<tr>
<td>hidden state</td>
<td>17</td>
</tr>
<tr>
<td>hierarchical deep q-network</td>
<td>21</td>
</tr>
<tr>
<td>hubert loss</td>
<td>55</td>
</tr>
<tr>
<td>importance sampling weights</td>
<td>21</td>
</tr>
<tr>
<td>input layer</td>
<td>16</td>
</tr>
<tr>
<td>learning rate</td>
<td>17, 20, 30</td>
</tr>
<tr>
<td>long short term memory</td>
<td>16</td>
</tr>
<tr>
<td>loss</td>
<td>17</td>
</tr>
<tr>
<td>loss function</td>
<td>17</td>
</tr>
<tr>
<td>markov decision process</td>
<td>19</td>
</tr>
<tr>
<td>mean absolute error</td>
<td>17</td>
</tr>
<tr>
<td>mean squared error</td>
<td>17</td>
</tr>
<tr>
<td>model-based reinforcement learning</td>
<td>19</td>
</tr>
<tr>
<td>model-free reinforcement learning</td>
<td>19</td>
</tr>
<tr>
<td>multi-layer perceptron</td>
<td>16</td>
</tr>
<tr>
<td>n-step q-learning</td>
<td>22</td>
</tr>
<tr>
<td>off-policy</td>
<td>19</td>
</tr>
<tr>
<td>on-policy</td>
<td>19</td>
</tr>
<tr>
<td>optimizer</td>
<td>17</td>
</tr>
<tr>
<td>output layer</td>
<td>16</td>
</tr>
<tr>
<td>parallel q-function</td>
<td>45</td>
</tr>
<tr>
<td>particle swarm optimization</td>
<td>46</td>
</tr>
<tr>
<td>prioritized replay memory</td>
<td>21</td>
</tr>
<tr>
<td>q-learning</td>
<td>19</td>
</tr>
<tr>
<td>q-table</td>
<td>19</td>
</tr>
<tr>
<td>rectified linear unit</td>
<td>17</td>
</tr>
<tr>
<td>recurrent neural network</td>
<td>16, 49</td>
</tr>
<tr>
<td>reinforcement learning</td>
<td>4, 18, 23, 37</td>
</tr>
<tr>
<td>replay memory</td>
<td>21, 30</td>
</tr>
<tr>
<td>root mean square propagation</td>
<td>18</td>
</tr>
<tr>
<td>selection policy</td>
<td>27</td>
</tr>
<tr>
<td>shallow neural network</td>
<td>17</td>
</tr>
<tr>
<td>smooth $l_1$ loss</td>
<td>55</td>
</tr>
<tr>
<td>softmax exploration</td>
<td>19</td>
</tr>
</tbody>
</table>
state, 18
stochastic gradient descent, 17
target network, 30
transfer learning, 3, 18, 31
value, 21
asynchronous deep q-network, 22
back-propagation, 17, 40, 55
batch, 18, 30
bloch sphere, 7
circuit, 8
classification, 15
clifford group, 8
clifford+t, 8
CNN; see convolutional neural network, 16, 49
cnot gate, 8
collapse, 7
constrained cnot circuit re-synthesis, 1, 2, 5, 12
convolutional layer, 16
convolutional neural network, 16, 49
coupling graph, 3, 10
cross entropy, 17
cx gate, 8
cy gate, 8
cz gate, 8
DDQN; see double deep q-network, 12, 21
decoherence time, 2
depth neural network, 17
depth q-network, 2–4, 20, 23, 39, 49
discount factor, 20, 30
double deep q-network, 12, 21
double q-learning, 20
DQN; see deep q-network, 2–4, 20, 23, 39, 49
dueling network, 21
dynamic programming, 19
elementary row operation, 13
environment, 4, 5, 18, 37
episode, 30, 56–59
error correction codes, 2
experience replay, 21
exploitation, 19
exploration, 19
exploration policy, 19
FC; see fully connected layer, 16
frontier, 49
fully connected layer, 16
gate, 6
gated recurrent unit, 16
gauss-jordan elimination; see gaussian elimination, 13
gaussian elimination, 13
genetic steiner, 32
geometric series, 25
gradient descent, 17
ground truth, 17
GRU; see gated recurrent unit, 16
h-DQN; see hierarchical deep q-network, 21
hadamard gate, 8
hidden layer, 16
hidden state, 17
hierarchical deep q-network, 21
hubert loss, 55
importance sampling weights, 21
input layer, 16
interference, 7
learning rate, 17, 20, 30
long short term memory, 16
loss, 17
loss function, 17
LSTM; see long short term memory, 16
markov decision process, 19
MDP; see markov decision process, 19
mean absolute error, 17
mean squared error, 17
MLP; see multi-layer perceptron, 16
model-based reinforcement learning, 19
model-free reinforcement learning, 19
multi-layer perceptron, 16
n-step q-learning, 22
NISQ; see noisy intermediate-scale quantum, 9