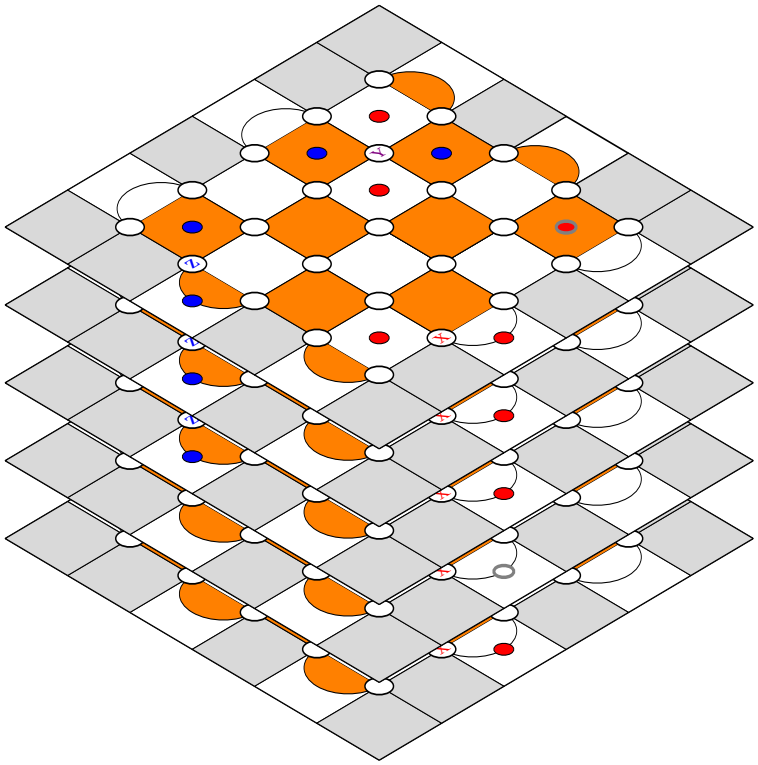

Deep Reinforcement Learning for Quantum Error Correction

on partially observable Markov decision processes

Master's thesis in Complex Adaptive Systems

Marvin Becker
Karl-Rehan Chiu Falck

Department of Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021
www.chalmers.se

Master's thesis 2021

Deep Reinforcement Learning for Quantum Error Correction

on partially observable Markov decision processes

Marvin Becker
Karl-Rehan Chiu Falck

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics

Chalmers University of Technology
Gothenburg, Sweden 2021

Deep Reinforcement Learning for Quantum Error Correction
on partially observable Markov decision processes
Marvin Becker
Karl-Rehan Chiu Falck

Supervisor & Examiner: Mats Granath, Department of Physics

Master's Thesis 2021
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: $d = 5$ surface code subject to $h = 5$ measurement cycles, creating a syndrome volume. Different qubit error types as well as syndrome measurement errors are shown.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021

Deep Reinforcement Learningfor Quantum Error Correction
on partially observable Markov decision processes

Marvin Becker, Karl-Rehan Chiu Falck
Department of Physics
Chalmers University of Technology

Abstract

Recent advancements in quantum computing have supported and reinforced their promising computational abilities which for certain problems far exceeds that of classical computers.
However, one of the main difficulties in realising larger quantum computers lies in the instability of quantum bits often resulting in erroneous states. While there are schemes in place to encode logical qubit states, such as repetition codes, the very quantum nature of these qubits makes it impossible to directly observe their quantum states without causing it to decohere. To circumvent this, parity measurements of localized groups of qubits, so-called syndrome measurements, can be performed as is the case in the surface code.
The encoding leads to improved stability and error resistance of the encoded logical qubit. The challenge of decoding the surface code in the presence of potential errors still remains: Multiple possible error configurations can lead to the same observed syndrome state, thus making the decoding task non-trivial. This has been addressed with different approaches such as the utilization of decoder agents trained via deep learning.
In this study, we deploy techniques from the realm of reinforcement learning such as Deep Q-learning, Proximal Policy Optimization, and Hindsight learning to train agents capable of decoding errors on the surface code. To better reflect a realistic setting, we also acknowledge the possibility of erroneous syndrome measurements as part of the training. The entirety of these syndrome measurements constitutes the observable state and introducing measurement errors breaks the Markov assumption which is central to many reinforcement learning algorithms and leaves us with a partially observable Markov decision process (POMDP). To deal with this, we introduce the dimension of time to our problem formulation by keeping track of the evolution of both qubits and syndromes. Our work includes a brand new state formulation for the syndrome evolution and utilizes different strategies to train agents on this newly-formulated problem.

Keywords: Deep Learning, Reinforcement Learning, Quantum Error Correction, Surface Code, Quantum Bits

Acknowledgements

First and foremost, I would like to thank Mats Granath for the supervision during this project. His helpfulness, input, and general interest in advancing the topic made it easy to engage in this work and stay motivated throughout the process.

I am also grateful for the cooperation with Adam Olsson and Gabriel Lindeby which enabled us to build on top of the code and concepts they developed for their Master's thesis.

Lastly, I want to thank my friends and fellow students who made studying during the last years such an enjoyable endeavor.

Marvin Becker, Gothenburg, June 2021

Like Marvin I want to thank Mats Granath for the supervision and input during the project. In addition I want to thank him also for taking time to take the time to give a rundown of quantum mechanics to a relative beginner making it possible for me to engage in the project.
Also, thanks are also extended to Adam Olsson and Gabriel Lindeby for helping us building on top of their code which significantly sped up our own progress.
Another thank you to Evert van Nieuwenburg who took the time to discuss the project.
Finally, thanks to all the professors, teachers, teaching assistants and students that taught, worked with or helped me along the way. They made the journey both enjoyable and interesting.

Karl-Rehan Chiu Falck, Gothenburg, June 2021

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Overview . . . 1
1.2 Previous Research . . . 3

2 Definitions **5**

3 Background **7**
3.1 Quantum Bits . . . 7
3.1.1 Encodings . . . 8
3.1.2 Surface Code . . . 9
3.1.3 Decoding . . . 11
3.1.4 Syndrome Measurements . . . 11
3.2 Machine Learning . . . 12
3.2.1 Artificial Neural Networks . . . 12
3.2.1.1 Convolutional Layers . . . 12
3.2.1.2 GRU and Gated Transformer . . . 13
3.3 Reinforcement Learning . . . 13
3.3.1 Deep Q-learning . . . 15
3.3.2 Proximal Policy Optimization - PPO . . . 17
3.3.3 Hindsight experience replay . . . 18

4 Method and Implementation **21**
4.1 Environment . . . 21
4.1.1 Qubits . . . 21
4.1.2 Syndromes . . . 22
4.1.3 Action Space . . . 25
4.1.4 Measurement Errors . . . 26
4.2 Reinforcement Learning . . . 27
4.2.1 Deep Q Learning . . . 27
4.2.2 PPO . . . 29
4.2.3 Hindsight . . . 31
4.2.4 Rewards . . . 31
4.3 Training on GPU Cluster . . . 32

4.4 Neural Network Architectures 32
4.4.1 Convolutional Network Configurations 33
4.4.2 2D Convolution and Recurrent Module 33
4.4.3 3D Convolution 33

5 Results **35**
5.1 Results 35
5.1.1 General Training 35
5.1.2 Hindsight Training 40
5.1.3 Performance Comparison 40
5.1.3.1 Hindsight Learning Comparison 42
5.1.4 Stack Depth 42
5.1.5 Of Failure and Lifetime 47
5.1.6 Network Activation 49

6 Conclusion **53**
6.1 Impact of Stack Depth 53
6.2 Threshold Analysis 53
6.3 Network Architectures 54
6.4 Learning Strategies 54
6.5 Future Work 55
6.5.1 Generalized Architecture 55
6.5.2 Split Input Toggle 56
6.5.3 Non-Constrained Channel List 56
6.5.4 Sparse Rewards 56
6.5.5 Training for Equivalence Classes 56

7 Ethical Discussion **59**

A Neural Network Architectures **I**

List of Figures

3.1 Bloch sphere representation of a quantum two-state system 8
3.2 Surface code grid 10
3.3 Non-trivial loop acting as a logical Pauli-X operator. 10
3.4 Non-trivial loop acting as a logical Pauli-Z operator. 10
3.5 Example of trivial loops for different qubit operations. 10
3.6 Basic fully-connected multilayer perceptron 13
3.7 Depiction of convolutional operation 14
3.8 General reinforcement learning schema 15

4.1 Surface code grid and syndrome matrix embedding 22
4.2 Layout of the syndrome stack 23
4.3 Infected syndrome stack 24
4.4 Syndrome stack tensor 25
4.5 General outline of the process structure for Deep Q Learning. 28
4.6 General outline of the process structure for PPO learning. 30

5.1 Evolution of number of steps per episode during training runs 37
5.2 Evolution of ground state rate and syndrome annihilation likelihood during training runs 37
5.3 Evolution of Q values and surface code energy during training runs . 38
5.4 Results illustrating strengths and weakness of hindsight learning . . . 41
5.5 Results illustrating that the weaknesses of hindsight does not always hold 41
5.6 General architecture and strategy comparison for $d = 5$ 43
5.7 Comparison of results between hindsight with aspect to the fail rate . 44
5.8 Comparison of fail rate when faced with different stack depths 45
5.9 Impact of stack depth on performance for 3D Conv and 2D Conv models, error rate rescaled to better reflect the expected number of errors introduced 46
5.10 Fail rate and lifetime analysis for different system sizes 47
5.11 Log scale fail rate threshold analysis 48
5.12 Birds-eye view of the surface code grid used in network activation analysis 49
5.13 Network activation analysis results and state representation. 51

List of Tables

A.1 Overview of used neural network architectures I

1
Introduction

The code used in this thesis can be found at https://github.com/mbecker12/surface-rl-decoder.

1.1 Overview

Quantum computers have been a hot topic for some time, promising significant speed-up for certain computational problems. While quantum computers may not yield improvements in every single aspect of computing, they can lead to substantial speedup in specific groups of problems, for example combinatorial problems. Quantum algorithms also introduce us to a whole new crowd of complexity classes [1, 2, 3] whose solutions are beyond the reach of classical computers for sufficient problem sizes.

One of the most famous examples is Shor's algorithm [4], which would threaten encryption methods used worldwide by helping to solve the problem of prime factorization. Less adversarial examples include works in airline scheduling [5] or solutions to exact-cover problems [6].

There exists a flock of useful quantum algorithms, like e.g. quantum Fourier transform, Grover's algorithm [7, 8, 3]. Furthermore, there are different paradigms within quantum computation: adiabatic quantum computations, quantum annealing, measurement based models, and circuit-based models. The efforts in this work can be best thought of as aiding circuit-based models. Otherwise, our work is totally agnostic to the physical qubit realization and treats qubits in a more conceptual manner.

Despite all their upside, their quantum nature makes quantum computers and quantum bits in particular susceptible to noise and disturbances. Quantum computers are usually operated at extreme conditions, meaning near-zero Kelvin temperatures. Even then, the slightest perturbation can cause the quantum state of a qubit to change or decohere. It is virtually unavoidable that a qubit will be subject to error. This causes qubits to have a drastically higher error rate than classical bits. At this point, we would like to refer the reader to the following blog article which covers the basic principles of quantum bits and error correction [9].

There are different strategies to deal with errors in computations in general. Maybe the simplest way to think of is repetitive codes where one bit of information gets repeated to make the information more resistant to outside noise. An interesting aspect to this is the fact that it is not possible to clone a qubit (see no-cloning theorem [10, 11]) but luckily there are other ways to replicate the state of the encoded qubit. We will however not go into further detail about replication of qubit information in this work (e.g. Nielsen and Chuang [3] explains a suitable mechanism). The surface code that we will introduce and use in this work can be seen as a more sophisticated, topological version of such repetition codes [12, 13].

The crucial task beyond the actual encoding is to spot errors and decode them if present. The decoding process is unaware of the actual physical qubit errors. Instead, it can only observe an indirect state representation, namely the syndrome. The syndrome is created by measuring the error parity of many local groups of qubits. Hence, one syndrome defect can only narrow down the occurrence of an error to a small group of qubits. There exist different approaches to decoding errors on the surface code in particular. In this project, we made use of another major advancement in computing in recent years besides quantum computing: We focused on training an agent using deep reinforcement learning to decode error chains on the surface code.

Moreover, in our project, we acknowledge the fact that local parity measures on the surface code, which yield the state for the decoding, are themselves prone to errors. This fact has been assumed to be negligible in previous work that we build on, or it has been disregarded to focus advancements in other aspects of quantum error correction [14, 15, 16].

In the lingo of reinforcement learning, this would make our problem a partially observable Markov decision process (POMDP) [17, 18]. This means that the system does not completely fulfill the Markov property that the current state is sufficient to describe the sequence of all predecessor states. In other words, the state at one particular time step cannot be trusted to convey all information about the system due to possible hidden variables or mechanisms. These hidden variables in our case are the measurement errors of the local parity checks which we will talk about in more detail in section 3.1.4.

1.2 Previous Research

Different strategies have been pursued in order to encode logical qubits and thus make them more fault-tolerant, among those strategies are the surface code or the toric code [12, 13, 19, 20].

Again, for these encodings, there exist different methods to decode a present syndrome configuration, such as the Minimum-Weight Perfect Matching (MWPM) decoder [21].

A more detailed discussion of error correction in the presence of measurement errors with differing underlying syndrome error models was done in [22].

Other approaches aimed at reducing qubit errors focus on inferring the noise spectrum surrounding the qubit and using this knowledge to extend coherence times [23].

Recently, with the emergence of machine learning and specifically deep reinforcement learning (RL), motivated among other things by the successes of AlphaZero and the like, reinforcement learning agents have been trained to deal with the decoding task [14, 15, 24, 25].

2

Definitions

- **Quantum bit** or qubit for short, is the smallest unit of quantum information and consists of a superposition of 0 and 1.
- **Physical qubit** is an actual qubit that holds the superposition of 0 and 1.
- **Ancillary** (or **ancilla**) qubits are special qubits meant to store the information of local error parity checks of physical qubits. This is needed as measuring the state of qubits directly causes their state to decohere. This way, the presence of errors is detected by a measuring of the ancilla qubits to check for odd parity which yields the eigenvalue -1.
- **Decoherence** of a quantum system causes the system to lose its quantum behavior, for example superposition.
- **Quantum superposition** describes a quantum system being comprised of multiple states at the same time as opposed to classical systems which can only ever be in one state at any given time.
- **Logical qubit** is the abstract, encoded qubit that can be used in a quantum algorithm. This can be the composition of several physical qubits in a certain array in order to provide stability and error resistance.
- **Code size**, d, describes the number of qubits along any row or column in our implementation of the rotated surface code. It also describes the code distance used for error correction purposes.
- **Stack depth**, h, measures the number of successive measurement cycles used to decode errors.
- **Syndrome defect** is a single activated stabilizer measurement, i.e. vertex or plaquette with eigenvalue -1. They are the result of a separate quantum circuit which performs an error parity check on local qubit groups. At the end of this quantum circuit, the result qubit is measured. This measurement defines whether a syndrome defect is present or not and can be stored classically.
- **Syndrome** describes the entirety of all syndrome defects on the surface code.
- **Syndrome stack** is defined as the entirety of multiple layers of the surface code representing different time steps at which measurements are performed.
- **Physical qubit error** is an error on a physical qubit as part of the surface code. Such an error can be discretized to a bit flip (X operation), phase flip (Z operation) or both (Y operation). Any of these operations result in an error and must be corrected. We use p_{err} to denote the probability of introducing such an error upon initialization of an episode.

- **Syndrome measurement error** is a measurement error of the ancillary qubits. As the syndrome measurements are also made up of quantum circuits, they are just as prone to errors as the physical qubits. They are introduced with probability p_{msmt} at the beginning of an episode.
- **Action history** is a list of previously chosen actions in an episode to correct errors.
- **Action tuple** is a conclusive description of an action to be performed, defined by (qubit x-coordinate, qubit y-coordinate, operator number). For the operator number, we use the following mapping to Pauli operators: $0 = I, 1 = X, 2 = Y, 3 = Z$.
- **Episode** is the sequence of steps from an initial (potentially erroneous) input state until either the maximum number of possible actions is reached or the terminal action was called by the agent.
- **Agent** is the artificially intelligent object which makes decisions based of a policy to maximize its rewards.
- **Environment** is the space with which the agent interacts with. In the case of this study it is the surface code.
- **Supervised Learning** teaches the agent a policy based on labeled data.
- **Unsupervised learning** teaches the agent a policy based on unlabeled data.
- **Reinforcement learning** teaches the agent a policy based on the cumulative reward given by its interactions with the environment.
- **Reward function** Determines how the agent is rewarded based on its actions.
- **Terminal action** is the action to be chosen by the agent if it is convinced that the present state of the surface code cannot be improved or is completely void of errors.

3
Background

Quantum computers can yield certain advantages over classical computers. However, as mentioned before, the quantum nature of the constituent components poses some problems on its own.
In this section, we give a brief description and summary of quantum bits as well as strategies to make them more error-resilient.
Lastly, we will talk about machine learning and neural networks and how those can contribute to the qubit decoding problem at hand.

3.1 Quantum Bits

Quantum bits (qubits) in a quantum computer are the analogous counterpart to bits in classical computers. They are comprised of quantum physical systems. The intention is to make use of the quantum properties of those systems when doing computations.

These quantum properties allow the qubits to not be restricted to only two states, as is the case with classical bits, which can be either in state 0 or 1.

For our work, we consider qubits made up of quantum mechanical two-level systems, say with states $|0\rangle$ and $|1\rangle$. Those kinds of qubits can exist in a superposition of both states: $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where $\alpha, \beta \in \mathbb{C}$, constrained to $|\alpha|^2 + |\beta|^2 = 1$. This allows for a representation of multiple different values in just one (quantum) bit.

Besides that, if multiple qubits are involved, those qubits can be put in an entangled state where the state of one depends on the state of the other. The precise state is however unknown until a measurement collapses the quantum state into one specific state. It is precisely these quantum properties (superposition and entanglement) that gives quantum computers their potentially supreme computing power.

Qubits can also be thought of as unit vectors on the Bloch sphere [3]. A visual representation of this concept is shown in Fig. 3.1. One can act with certain unitary operations on the qubits to change their state. Each such operation can be represented as a rotation on the Bloch sphere. We constrain ourselves to the set of Pauli operators in this work:

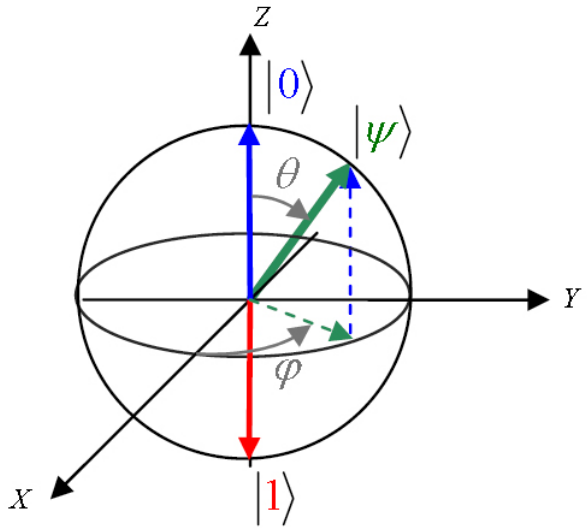

Figure 3.1: Depiction of the Bloch Sphere. A qubit can be thought of as a unit vector of the Bloch sphere. Qubit operations act as rotations on the sphere, with the Pauli-X operator causing the qubit to be flipped by π radians around the x-axis, and the Pauli-Z operator causing a similar rotation around the z-axis [26]. Image from [27].

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

3.1.1 Encodings

Working with qubits is inherently more error-prone than classical bits. For one, one needs to preserve the quantum nature of the qubits to utilize the aforementioned quantum properties. For this, the qubits need to be protected against external noise, one contributing factor to that is also temperature. To minimize thermal fluctuations, the temperatures of physical qubit systems need to be very low in general. Still, for a multitude of reasons, the quantum state can change or it can collapse (so-called decoherence). Such processes potentially introduce errors in a quantum circuit.

To make quantum computers more error-resilient, one approach is to encode one logical qubit with multiple physical qubits.
The simplest of such codes is the 3 qubit bit-flip code which protects against Pauli-X errors. An analogon to that would be the 3 qubit phase-flip code, protecting against Pauli-Z errors. An extension of these codes, namely the Shor code, can be used to protect both against bit-flip and phase-flip errors using nine qubits [3].

A significant hurdle when working with quantum encodings is the fact that one cannot measure the qubits individually to learn about potential errors. This would cause the qubits to decohere and lose their quantum properties. Instead, one is limited to certain local measurements involving the error parity of at least two qubits. In the case of the 3 qubit bit-flip code, one would check for an X error (corresponding to a bit-flip) on qubit 1 or 2 simultaneously and store this bit of information on a classical bit. This check will be performed by a quantum circuit with a designated result qubit (or ancilla qubit) at the end. The measurement of this ancilla qubit provides us with information of whether an error is present or not. The same is then done for qubits 2 and 3, the result is measured on another quantum circuit and its corresponding ancilla qubit. Analyzing the measurements of these ancilla qubits can give clues about where an error occurred.

Note that even though quantum errors are generally continuous, theorem 10.2 in Nielsen and Chuang [3] shows that a set of discrete error correction operators is able to correct errors by an arbitrary noise process on a given code. The authors summarize that it is possible to discretize quantum errors and to correct these discretized errors, it is sufficient to be able to act with the four Pauli matrices (I, X, Y, Z) [3].

For our intents and purposes, the effect of two successive operations of Pauli matrices is equivalent to the application of another Pauli matrix. In very simplified form, for us this implies

$$\begin{aligned} X\,Z &\propto Y, & X\,Y &\propto Z, & Y\,Z &\propto X, \\ \text{and } X\,X &= I, & Y\,Y &= I, & Z\,Z &= I. \end{aligned}$$

This means that applying one operator to another discretized qubit error causes the error to change. The second line shows that applying the same operator as the suspected discretized qubit error corrects the qubit error.

3.1.2 Surface Code

One example of a more sophisticated encoding is the surface code [12, 13]. In the surface code, $d \times d$ qubits are arranged in a lattice (see Fig. 3.2). Syndrome measurements can be performed to measure the parity of local groups of four qubits with regard to the existence of Pauli-X or -Z errors. The result of syndrome measurements done on ancillary qubits can be stored in on a classical computer. The blank and orange faces in the qubit grid in Fig. 3.2 can be seen as placeholders for those ancillary qubits.

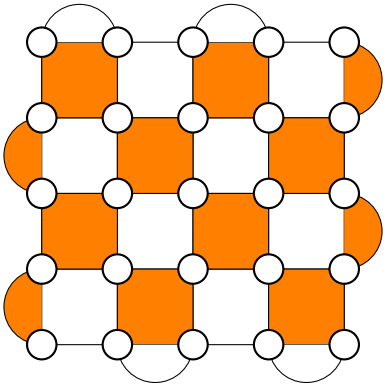

Figure 3.2: One layer of the qubit grid, i.e. the rotated surface code with $d = 5$. The white nodes correspond to physical qubits. The blank faces denote plaquette syndromes (consisting of Z operator stabilizers; measuring parity of adjacent X errors) and the orange faces show vertex syndromes (consisting of X operator stabilizers; measuring parity of adjacent Z errors). The style of the figure was very much inspired by [20].

The surface made up of the $d \times d$ qubit grid represents one logical qubit. More specifically, in our implementation, we use the rotated surface code as described for example in [20].

Logical operators on this encoded logical bit can be achieved by chains of Pauli-X (or -Z) operators along the horizontal (vertical) axis (see Figs. 3.3, 3.4). Those chains or any operator that commutes with the vertex (plaquette) stabilizers will cause the qubit state of the entire grid to change comparable to applying a Pauli-X (Z) operator to a single qubit.

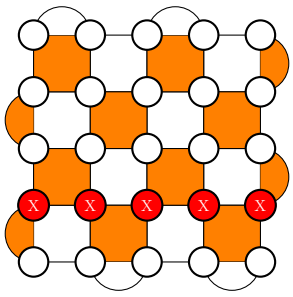

Figure 3.3: Non-trivial loop acting as a logical Pauli-X operator.

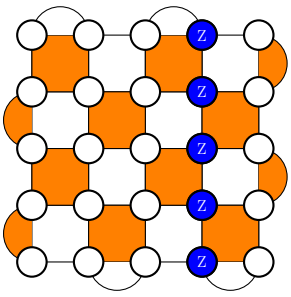

Figure 3.4: Non-trivial loop acting as a logical Pauli-Z operator.

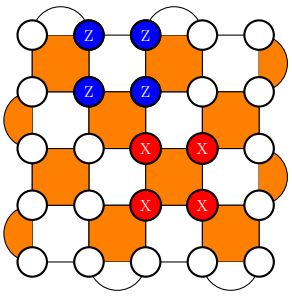

Figure 3.5: Example of trivial loops for different qubit operations.

Besides these logical operators which are also called non-trivial loops[1], there exist trivial loops (see Fig. 3.5). These trivial loops cause no syndrome measurements despite the presence of qubit errors and do not cause the logical state to change. Their presence at the end of a decoding episode would be considered a successful outcome of the decoding process.

Generally, one can think of a larger grid size, or code distance, d as beneficial for error correction purposes. A larger code makes it less likely to randomly introduce logical errors. In contrast, a larger qubit grid also introduces more possible error chains which can cause the same syndrome state. The ultimate goal would be to have a decoding agent learn the correct statistics about the possible error chains, a task that gets more difficult with growing code size [16].

3.1.3 Decoding

One or more active syndrome defects indicate the presence of qubit errors[2]. The difficulty lies in the fact that multiple qubit error configurations can lead to the same syndrome state. It is the task of a potential decoder to infer which qubit error chains are most likely to have caused the obtained syndrome. With that knowledge, the correct operations to remove the errors from physical qubits can be applied to restore the logical qubit ground state, in the most probable way.

3.1.4 Syndrome Measurements

The local parity checks performed on local qubit groups – the syndrome measurements – are themselves nothing but the results of their own quantum circuits. On these quantum circuits, the final, ancillary qubit is measured and the result is stored and used to denote the presence of syndrome defects. Therefore, these parity checks are just as error-prone as the actual qubits that make up the encoding qubit grid in the surface code. Hence, the syndrome measurements can occasionally show the wrong outcome. This work introduces such syndrome measurement errors and aims to train a decoding agent to be aware of their existence. Moreover, due to the makeup of the syndrome measurement circuits and possible entanglements between qubits, the syndrome measurement errors can cause ripple effects to the physical lattice qubits [22]. In this work however, we treat syndrome measurement errors as events independent of lattice qubits so that a syndrome measurement has no consequences on the physical qubits.

[1] We call these *loops* because the concept originates from the toric code, where the qubit grid is embedded on a torus. Hence, a full string of operators which wraps around the torus can be considered a (non-trivial) loop. We merely adopted the terminology from there.

[2] In this short subsection, we disregard syndrome measurement errors. The presence of these is discussed in section 3.1.4

3.2 Machine Learning

Machine learning is a wide subject within computer science with various implementations and use cases. The general idea behind machine learning is to have a machine process data and learn to detect patterns, predict or select a response and so forth. There are three major branches within machine learning [28, 29, 30], supervised learning where the output is compared with already known targets, unsupervised learning where the target outputs is non-predetermined and finally reinforcement learning which deals with how intelligent agents should behave in a given environment.

3.2.1 Artificial Neural Networks

The idea behind artificial neural networks is to mimic the behaviour of the nerve cells within animals but in a simplified way [31]. A network is made up of neurons (a unit) in different layers going from input layer to a series of hidden layers to an output layer as shown in figure 3.6. In the simplest of models, as the one shown, the connections between the nodes of different layers are connected by weights w_{ij}, with ij for connection from unit i to j, and input a_i, we have thus

$$a_j = g\left(\sum_{i=0}^{n} w_{ij} a_i\right), \tag{3.1}$$

with $g(*)$ being the activation function (may be chosen arbitrarily but common choices are ReLU, Sigmoid and Heaviside step function) is applied to the sum.

There are many types of networks with different types of connections for different tasks. As this work is not about different types of architectures, we will simply briefly explain the different types of networks that are utilised for the network in the method section 4 and ask the reader to refer to other literature if one wishes to understand the details of the different network modules.

3.2.1.1 Convolutional Layers

Convolutional layers, as the name suggests, convolve kernels (or filters) over the input data in order to create feature maps (different maps for each filter). These maps can hold information of specific features depending on the kernel and is thus an invaluable asset where the output needs to take into consideration the positions of the input data relative one to another (an example would be image classification).

In our work, we utilise networks of both 2D convolutional layers and 3D convolutional layers to enable the agent to analyze the syndrome state. This is important as it is crucial to retain information about the spatial syndrome positions in order to find the optimal action for the correction, see 3.1.2.

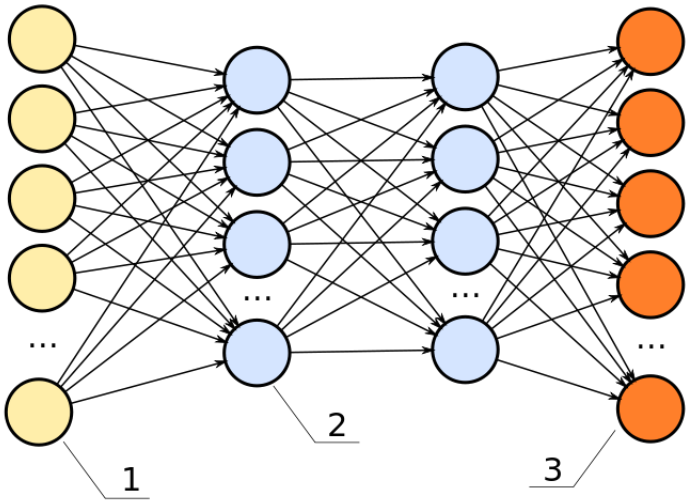

Figure 3.6: An example of how an artificial neural network can be constructed, with nodes feeding forward through weights (edges) from input layer (1, left) to output layer (3, right) through a number of hidden layers (2). Image from [32].

3.2.1.2 GRU and Gated Transformer

As we include the aspect of time in the input in order to distinguish true errors from measurement errors, one also has to take into consideration that the 2D convolutional module does not consider time in its convolutions (whereas this happens implicitly in the 3D convolutional layers). Therefore, layers that explicitly take the sequence of time slices into consideration become necessary. The different layers used for this work are GRU (gated recurrent unit) and gated transformer. GRU [34] layers use past input together with the new input and thus are able to take time into consideration as the feature maps are fed into the network. The gated transformer [35] adds aspects of positional embedding for the sequence input and is thus also another aspect worth studying.

3.3 Reinforcement Learning

In many problems the agent will have to interact with an environment. This poses the problem that one may not know the entire potential state of the environment or what actions the agent should take to achieve the desired results.

Within the environment, the agent takes an action dependant on the state representation and the action policy. The environment then returns the reward and the new state representation. The goal of the agent is to maximise its cumulative reward over the course of each episode.

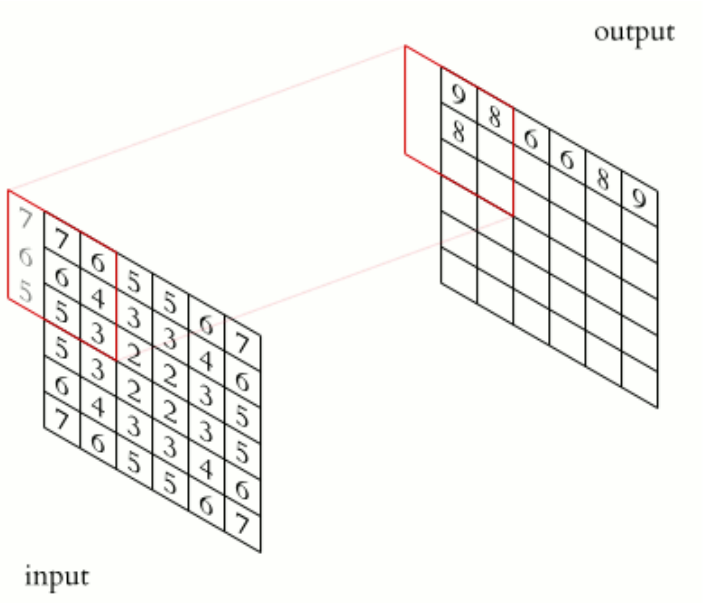

Figure 3.7: Convolutional layer at work. The kernel convolves the input data to extract a feature map. Image from [33].

The optimal value of a state s_t is determined by the Bellman equation [36, 37]

$$V(s_t) = \max_a \left(R(s_t, a) + \gamma \sum_{s'_{t+1}} P(s_t, a, s'_{t+1}) V(s'_{t+1}) \right), \tag{3.2}$$

where $V(s_t)$ is the value of the current state, $R(s_t, a)$ is the reward for taking the optimal action (as noted by $\max_a(.)$), γ is the discount factor for future rewards, s'_{t+1} is the potential future state by taking action a, $P(s_t, a, s'_{t+1})$ is the probability of an action at state s_t resulting in state s'_{t+1} and $V(s'_{t+1})$ is the value of the future state. Note that the reason for the discount factor is to ensure that the agent values rewards closer to the current point in time as otherwise it sees no issues with simply taking an unnecessary long path towards the goal. As one can see, the formula is recursive and one could find the exact value for each state if one knew all possible states and all possible rewards for each possible action. Possessing the information would enable one to immediately determine the optimal action policy but that is however for many problems not the case and instead one will have to try to estimate these values through interacting with the environment.

The way to achieve this, varies with different methods but in essence the agents learn through exploration and exploitation. The exploration is conducted by taking random actions at times, resulting in potential new states which hopefully results in better cumulative rewards than what the current best action policy suggests and updates the policy depending on the results. This is in contrast with exploitation in which the agent chooses the best action in accordance to its action policy in order

to optimise its reward based on its current knowledge. As this is a trade-off when learning there are various ways of implementing this, the most common one being the ε-greedy policy which in essence says:

random number $r \in [0, 1]$
if $r < \varepsilon$ **then**
| return `random_action`
else
| return `predicted_best_action`
end

Algorithm 1: ε-greedy algorithm

Usually one selects a high value of ε for the initial phase to promote exploration whilst towards the end of the training ε is lowered to a very small number to utilise the exploitation more and focus on fine-tuning the network.

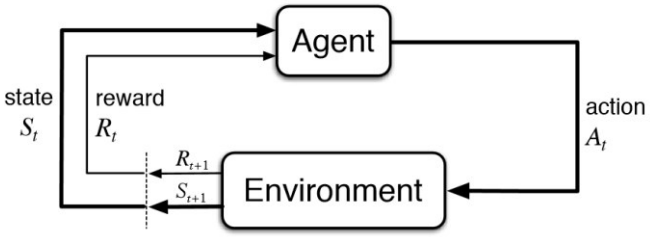

Figure 3.8: The agent interacts with the environment and then receives a reward and the new state representation. Picture taken from [38].

3.3.1 Deep Q-learning

One of the most commonly used learning methods is Q-learning. It predicts the best action by calculating the Q-values, also known as the action values ($Q(s_t, a_t)$ which is an estimation of how rewarding an action will be, future actions considered. For each action given current state by updating $Q(s_t, a_t)$ with

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)], \tag{3.3}$$

where r_t is the immediate reward for time step t, γ is the discount factor, α the learning rate and $\max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$ is the maximum future estimated reward (by the same algorithm). This means that the algorithm enables the agent to learn to consider future states it may visit recursively as it explores the environment. However, storing the values for each state becomes impractical if the number of states is too large and a function estimator of the Q-values becomes more practical. Deep Q-learning (DQN) is the implementation of Q-learning with two deep (multi-layered) neural networks to predict the Q-values. One network performing online learning and the other offline (also called the target network). The online network takes as input the state representation and calculates the predicted Q-values for each action. If ε-greedy policy is implemented then `predicted_best_action` is the

action with the highest Q-value. It stores the data in a replay buffer which enables it to decorrelate data by randomly sampling from it. The loss is calculated from each mini-batch (samples from the replay buffer) as the average difference of the output between online network predicted Q-values and offline network's predicted future maximum discounted Q-values added to the immediate reward of the current step. That is

$$L = \frac{1}{N}\sum_{i=1}^{N}(Q(s_i, a_i) - (r_i + \gamma \max_a Q(s_{i+1}, a)). \tag{3.4}$$

The online network is then updated through backpropagation in one optimization step using the loss [31]. Finally, after certain periods (chosen arbitrarily) the offline network is updated by copying over the parameters of the online network.

The algorithm is summarised below in algorithm 2.

```
Initialize network and parameters
Initialize memory M
for all epochs do
    while game not done do
        Read state s_t from environment
        if exploration then
            Select random action a_t
        end
        else
            Select predicted action a_t
        end
        Perform action a_t
        Receive reward r_t, observe new state s_{t+1}
        Store step (s_t, a_t, r_t, s_{t+1}) in replay memory M
    end
    mini-batch ← random sampling of steps from M
    for all steps ∈ mini batch do
        Predict Q-values for s_t with online network
        Predict Q-values for s_{t+1} with offline network
        calculate loss
        update online network
    end
    Decrease ε
    if Time to update offline network then
        update offline network
    end
end
```

Algorithm 2: Deep Q-learning

3.3.2 Proximal Policy Optimization - PPO

In contrast to Deep Q Learning explained in section 3.3.1, Proximal Policy Optimization (PPO) is an on-policy method. Usually on-policy methods are restrained by the fact that they can only use *one* current batch of episode samples. Otherwise, the learner would get an outdated estimate of the policy gradient. This also means that samples cannot be reused [36].

PPO however, makes use of a clipped loss function to limit the impact of one sample on the overall network output and by that the agent's policy [39]. These rather conservative policy updates reduce variance and enable us to reuse samples from a central replay buffer. This makes PPO more sample-efficient compared to its predecessor on-policy algorithms [36].

Besides that, PPO being an actor-critic method, rather than a pure policy gradient method (although the terminology is vague and sometimes inconsistent [36]), requires a value function approximator $V(s;\theta)$ and a policy approximator $\pi(a|s;\phi)$. This can be realized with one neural network with two separate heads splitting off from the network's (convolutional) base. One head will be responsible for the representation of the policy with an output of size $||A||$ (see. Eq. 4.1), and one head with an output of size 1 representing the value approximation of the current state. Using a split network reduces the number of parameters in the network and enables both outputs to learn simultaneously.

PPO updates the underlying network weights of the agent based on multiple random samples from the same episodes in the replay buffer. This introduces stochasticity while making sure that it is likely to use all present episodes in the buffer in multiple optimization steps.

In each optimizer step, the policy gets updated according to the following objective function [36]:

$$A^{\text{GAE}}_{\text{weighted}} = \frac{\pi(a|s;\phi)}{\pi(a|s;\phi^-)} A^{\text{GAE}} \tag{3.5}$$

$$A^{\text{GAE}}_{\text{clipped}} = \text{clamp}\left(\frac{\pi(a|s;\phi)}{\pi(a|s;\phi^-)}, 1-\epsilon, 1+\epsilon\right) A^{\text{GAE}} \tag{3.6}$$

$$J(\phi,\phi^-) = \mathbf{E}_{(s,a,A^{\text{GAE}})}\left\{\min\left[A^{\text{GAE}}_{\text{weighted}}, A^{\text{GAE}}_{\text{clipped}}\right]\right\} \tag{3.7}$$

A^{GAE} is the generalized advantage estimator (GAE) defined as follows:

$$A^{\text{GAE}} = \sum_{t=0}^{T} \gamma^t \left(r_t + \gamma V(s') - V(s)\right), \tag{3.8}$$

where γ is the discount factor.

Similarly, the value output is subject to the loss function

$$L(\theta, \theta^-) = \mathbf{E}_{(s,a,G,V)} \left\{ \max \left[G - V(s;\theta), G - (V + \text{clamp}\left(V(s;\theta) - V, -\delta, \delta\right)) \right] \right\} \tag{3.9}$$

G is the return $G = \sum_t^T \gamma^t r_t$ of an episode, r_t is the reward gained in step t.
ϕ describes the parameters involved in the policy approximation, and θ describes the parameters used for value approximation. Due to the two-headed network approach, the parameters covered by ϕ and the ones covered by θ overlap in the parameters of the network's shared base. The superscript in ϕ^-, θ^- denotes the old parameter states, before the optimization step.
The clamp$(\cdot, \cdot, \cdot)$ function makes sure that the value in question (first argument) is between the lower bound (second argument) and upper bound (third argument). Otherwise, it returns the first argument unchanged. ϵ and δ are clipping parameters to be chosen for the training setup.
The expectation values $\mathbf{E}_{(\cdot)}$ in Eqs. 3.7 and 3.9 are to be taken over all samples in a batch.

We also introduce an entropy term $\mathcal{H} = \mathcal{H}(\cdot|s')$ to the total loss function. This leverages randomness in the agent's policy and aims to counteract specific values from shooting off for no valid reason. This way, the neural network will have to be really sure about an action with a large policy probability since this larger value comes at an entropy cost. We can then update the complete two-headed network by combining the objective functions with reasonable weighting w_V, w_H:

$$\mathcal{L}(\phi, \theta) = J(\phi, \phi^-) + w_{\text{V}} L(\theta, \theta^-) + w_{\text{H}} \mathcal{H}(\phi|s') \tag{3.10}$$

3.3.3 Hindsight experience replay

Among the many existing reinforcement learning algorithms that exist, we have also chosen hindsight experience replay which was devised for the purpose of learning from failed attempts [40].

This algorithm is especially useful when the state space is large and impractical to explore using more traditional algorithms. A simple example would be trying to shoot a ball into a goal. There are many ways to shoot it and the ball can end up in many places but all attempts except the ones that end up in the goal will not yield positive results and all the agent can do is infer that it failed from those. Humans however can figure out from the failed attempts what they can do to improve their attempts and eventually score. At this point one could simply use the idea that one should use shaped rewards along the lines of $r = -||s - g||^2$ (absolute distance between state s and goal g). This works for simple problems but if it is more complex such as problems where we are unable to determine a reward function analogous to $-||s - g||^2$. If this is the case and a sparse reward scheme is the only decent choice, we have a problem that the agent struggles with learning and exploring (as it has no immediate incentive to explore).

The solution to this is the idea of introducing multiple false goals from the set $G \subseteq S$ for $S = \{s_0, s_1, ...s_T\}$. One such false goal denoted by $g' \in G$ is then set to some state in the sequence, that is $g' = s_t,\ t \in [1, T]$. The pseudo code is as shown below in algorithm 3. This allows for the algorithm to learn how to achieve states further from the initial state and thus promotes exploration.

for *episode* $= 1, M$ **do**
  Sample goal **g** and initial state $\mathbf{s_0}$
  for $t = 0, T-1$ **do**
    Sample action $\mathbf{a_t}$ using the policy
    $a_t \leftarrow \pi_b(s_t)$
    execute action a_t and observe new state s_{t+1}
  end
  for $t = 0, T-1$ **do**
    $r := r(s_t, a_t, g)$
    store transition $(s_t, a_t, r_t, s_{t+1}, g)$ in **R**
    Sample set of additional goals for replay $\mathbf{G} \subseteq \mathbf{S}$ (current episode)
    for $g' \in \mathbf{G}$ **do**
      $r' := r(s_t, a_t, g')$ store transition $(s_t, a_t, r'_t, s_{t+1}, g')$ in **R**
    end
  end
  for $t = 1, N$ **do**
    Sample minibatch **B** from replay buffer **R**
    Perform one step of optimization using off-policy algorithm **A** and mini-batch **B**
  end
end

Algorithm 3: Hindsight learning

4
Method and Implementation

In this section, we will take a look at how the theoretic framework was implemented. There are different aspects to be covered: The implementation of the environment which includes the qubit grid and syndrome state as well as the realization of syndrome measurement errors, and beyond that different learning programs. Our code is publicly available at https://github.com/mbecker12/surface-rl-decoder.

4.1 Environment

As a central part of any reinforcement learning project, we first define the environment with which the agent is supposed to interact. Essentially, the environment consist of the qubit grid with corresponding ancillary qubits storing the syndrome measurement results. Moreover, the environment needs to support different functionalities: initializing qubits and resulting syndrome states, computing a syndrome state from a given qubit arrangement, storing syndrome measurement errors as well as actual qubit errors and being able to tell the expected (measurement error-free) syndrome state apart from the observed state (which may include syndrome measurement errors).

On intialization, random qubit errors are imposed on the surface code. From there, the actual state, namely the entirety of syndromes, can be calculated.

Additionally, we store an action history, both to keep track of the number of performed actions and to be able to apply the whole suggested action sequence to the initial qubit errors to receive the resulting final state. In the following subsections, we are going to take a closer look at how the qubits, syndromes, and measurement errors are implemented.

4.1.1 Qubits

The implementation of qubits is straight-forward: The physical qubits of a $d \times d$ surface code are stored in a $(d \times d)$-matrix. They can take on one of the following states: $\{0, 1, 2, 3\}$, where the numbers indicate which net operator has been applied to the qubit, $\{I, X, Y, Z\}$, respectively.

Qubit errors are introduced at the initialization of a new episode with probability p_{err}.

4.1.2 Syndromes

The entirety of local syndrome measurements is embedded in a $((d+1) \times (d+1))$-matrix. This representation is an efficient way to store and depict the information about syndrome measurements without being too bloated. With this implementation, certain entries of the syndrome matrix will always remain 0 since they cannot be mapped to a real syndrome measurement, see Fig. 4.1 for more details.

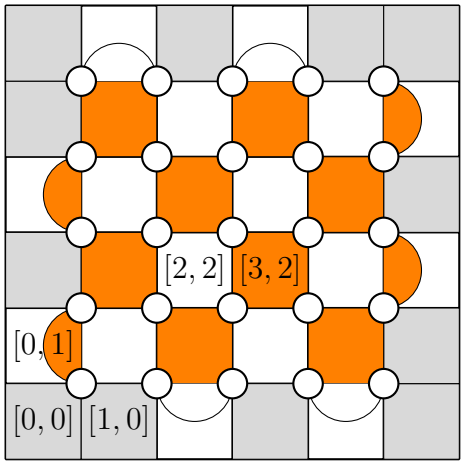

Figure 4.1: Matrix representation of syndromes. The syndrome matrices that make up the observational system state are embedded in $(d+1) \times (d+1)$-dimensional matrices. The faces of the matrix show the x- and y-indices for the python/numpy implementation. Grey fields are always 0. In the code, this is handled by applying binary mask arrays. All other fields contain information about the activation of its corresponding syndrome defect measurement.

Besides, the environment needs to be able to compute a syndrome state from the qubits and the information about errors on those qubits. With the aforementioned syndrome embedding, this computation can be implemented efficiently. The key to calculating whether a syndrome measurement should be set to true is by looking at a local group of four qubits (two qubits at the edges) and comparing their error types and parity of errors. This can be handled by applying a roll operation[1] in which the current qubit matrix gets shifted in certain directions, i.e. $\varnothing$, *up*, *left*, as well as *up + left*. This creates copies of the qubit matrix for each direction where, when stacked on top of each other, the qubits in one location correspond to the qubits subject to one local syndrome measurement. The syndrome defect can then be inferred by modular addition of all rolled copies of the qubit matrix. Thanks to this trick, the computation can be done in a vectorized way.

Now, since we also simulate syndrome measurement errors in our system, meaning that the syndromes themselves which make up the state of our environment cannot be fully trusted, we find ourselves in a partially observable Markov decision process (POMDP) if we were to use only one instance of the syndrome grid.

[1] https://numpy.org/doc/stable/reference/generated/numpy.roll.html

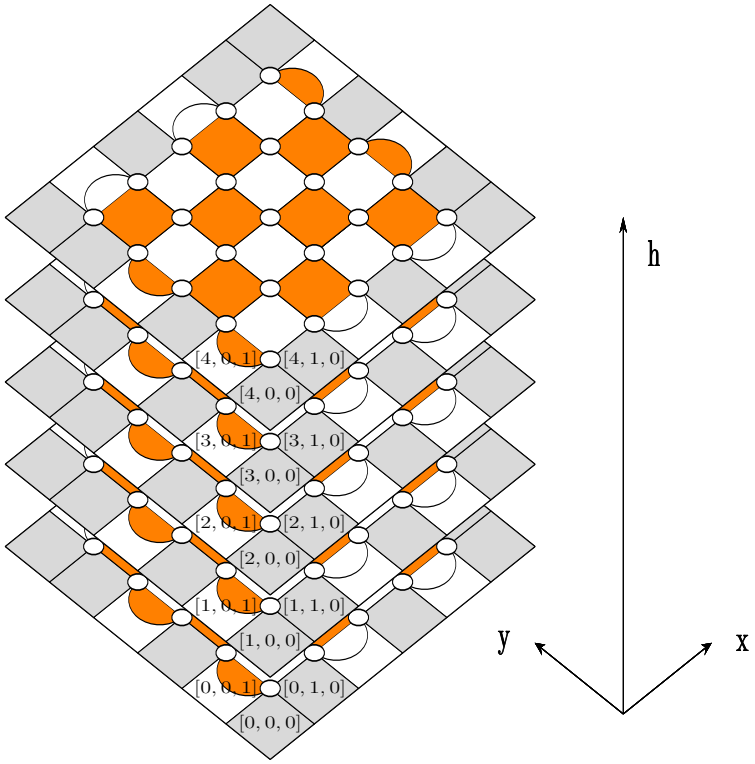

Figure 4.2: Depiction of the syndrome stack. The indices denote 3-dimensional coordinates [height, x, y] of the ancillary qubits in the state tensor. This is the basic shape of the realization of the state in our RL setup.

To solve this problem, we stack multiple instances of the syndrome grid (same for the qubit grid, for that matter) to emulate the notion of time evolution of the same surface code instance, see Fig. 4.2. This can be thought of as successive measurement cycles of the same surface code instance. Time runs from the bottom of the stack to the top and can be thought of as different time steps in one iteration of a quantum circuit: Between different actions in a quantum circuit, we take a snapshot of the syndrome state. At the end of the quantum circuit, we want to know which errors have accumulated and shall be corrected. The goal is then to have a syndrome-free top layer of the syndrome stack.

In our implementation, we introduce random syndrome measurement errors in the syndrome stack at the beginning of an episode with probability p_{msmt}.

This workaround causes other problems: How do we treat the added dimension of time in our implementation? How do we teach an agent do differentiate between real syndromes and mere syndrome measurement errors? We aim to answer those questions in Sections 4.4 ff.

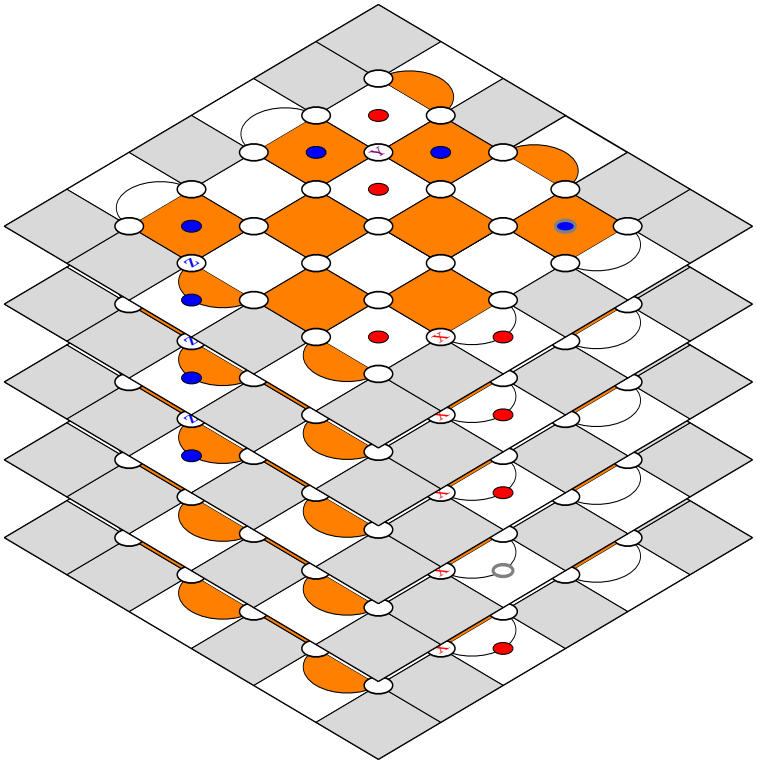

Figure 4.3: Depiction of the syndrome stack with different types of errors present. Erroneous node contain the letter denoting the Pauli operation corresponding to the error (X, Y, or Z). Blue circles show vertex syndrome defects, caused by Z-type errors; red circles show plaquette syndrome defects, caused by X-type errors. Grey circles around syndrome defects (or missing defects) show measurement errors.

Fig. 4.3 shows an exemplary state of the syndrome stack with an X error present throughout all measurement cycles. The syndrome defects caused by this error are only being interrupted by a syndrome measurement error (grey circle) in a single layer. On the left-hand side, there is an additional Z error which only popped into existence after two measurement cycles and persisted throughout the remaining measurements. Furthermore, there is a Y error depicted near the top of the grid, causing syndrome defects in all its surrounding faces. Lastly, a single, stray measurement error causes a pseudo-syndrome defect on the right-hand side of the topmost layer. An optimal decoder agent should learn to ignore this syndrome defect on the grounds that it was created spontaneously with no other hints of a real physical qubit error.

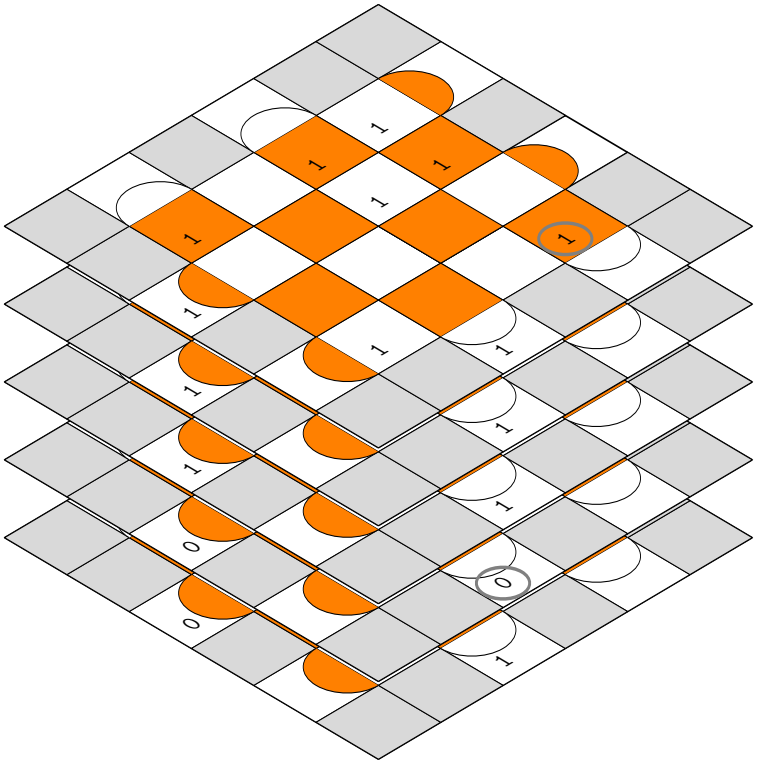

Figure 4.4: Visualization of state representation in tensor form. All syndrome defects – whether they were caused by real physical qubit errors or syndrome measurement errors – are stored as a `1`, all the other entries of the tensor are `0`. Color information, i.e. the syndrome type, is neither stored in the state (meaning the state representation in the RL sense) nor is it passed on to the agent explicitly.

Fig. 4.4 shows a sparse representation of what the state would look like for the syndrome depicted in Fig. 4.3. This is stored in an $h \times (d+1) \times (d+1)$ tensor and fed as input to a neural network.

4.1.3 Action Space

To be able to pick the correct action based on the networks output, we have to define a mapping between the flat q-value array and a $(3 \times d \times d)$-dimensional space which describes all possible qubit operations.

It is important to note that the agent also has to learn to trigger the `terminal` action on its own, bringing the action space dimension to

$$||A|| = 3d^2 + 1. \tag{4.1}$$

This can prove difficult to handle for the agent, especially with growing code sizes: The amount of *regular* actions, namely Pauli operations, by far outweighs the single terminal action. This leaves the terminal action with a very small probability to be explored by randomly chosen actions. With our reward scheme, calling the terminal action at the wrong time, e.g. with syndromes still remaining, will lead to punish-

ment in the form of a relatively large negative reward. Hence, it becomes even less likely for the agent to discover the terminal action in a beneficial scenario to learn to trigger this action reliably.

We generally assume that there are 3 possible actions per qubit, namely Pauli-X, -Y, and -Z operators; this is the explanation for the straying 3s to appear in action space dimension and corresponding code snippets. The Pauli operators are mapped to $o \in \{1, 2, 3\}$, for X, Y, Z ($o = 0$ is reserved for the identity operation). Further, let $i \in \{0, ..., 3d^2\}$ be any index to a Q-value in the Q-value array with $i_{\max}$ being the index of the maximum Q-value, d is the code distance, $x, y \in \{0, ..., d-1\}$ are the qubit coordinates on the grid in x- and y-direction, respectively.

(Note: due to python indexing which starts at 0, the index space spans $3d^2 + 1$ integers. The +1 is added for the terminal action which is part of the action space, cf. Eq. 4.1.)

if $i = 3d^2$ **then**
  return $(0, 0, a_{\text{terminal}})$
else
  $a \leftarrow i \bmod 3$
  $o \leftarrow a + 1$
  $g \leftarrow \lfloor (i - a)/3 \rfloor$
  $x \leftarrow g \bmod d$
  $y \leftarrow \lfloor g/d \rfloor$
  return (x, y, o)
end

In the above algorithm, a_{terminal} is usually set to 4.

4.1.4 Measurement Errors

Within this collection of qubit errors and resulting syndrome states, there might be faulty syndrome measurements.

The measurement errors are denoted by a mask matrix in the shape of the syndrome stack. This mask is created upon initialization of a new state and stored separately. To obtain the physically relevant, measurable state, the corresponding syndromes which are caused by physical qubit errors are then flipped. See Fig. 4.3 for a depiction of measurement errors.

At the same time, the original configuration of qubit errors is stored, completely unimpeded by measurement errors. This way, one can apply all actions in the proposed action sequence to the original qubit error configuration to get the resulting correct syndromes without any measurement errors. This is used to judge the performance of the action sequence in a setting where all syndrome measurements are perfect. This is desirable, as otherwise the syndrome measurement errors, if they

are not being detected as such, could lead to harmful decisions by the agent. This way, we can check if the agent is still able to retain the logical state and get rid of all qubit errors even in the presence of faulty measurements.

Since the syndrome state is not guaranteed to be the real state of the system, we refrain from terminating an episode automatically. Instead, the agent needs to learn when to decide to terminate an episode. Ideally, it would do so if it reckons that the current state cannot be improved and all physical errors have been corrected.

4.2 Reinforcement Learning

In this project, we rely on reinforcement learning to train an agent to decode qubit errors and reinstate the desired logical state of the surface code. For this, we pursue different strategies within reinforcement learning and we try out different neural network architectures in these agents.

In this section, we will describe further how we set up the different RL methods.

4.2.1 Deep Q Learning

For the Deep Q Learning setup, we follow the example of Gabriel Lindeby and Adam Olsson [16] and deploy a distributed program made up of different sub-processes. In this setup, our program consists of one learner process, one central replay memory, and multiple actor processes which in turn spawn multiple environments to run different episodes in parallel within these environments. A depiction of the relation of the different processes is given in Fig. 4.5

All the processes are triggered from one central main script. In this script, all the configuration is handled and the needed parameters are passed to the sub-processes. Also, the communication channels between the different sub-processes are set up.

The **learner process** contains the online policy network which is trained. Here, we also have an offline target network with the same architecture but frozen weights; this network does not get updated by gradient descent steps. Instead, after a defined number of learning steps, this target network gets updated with the current weights of the policy network. This technique helps stabilize the Q-values which the network is tending towards, otherwise there is a chance that the Q-values estimated by the neural network would diverge. The learner process then executes a learning step by calculating the loss according to the Bellman equation (Eq. 3.4), and updates the online network's parameters accordingly.
At the same time as the target network is updated, the learner process sends out the current network weights to the different actor processes so that those can update their local policy network as well.

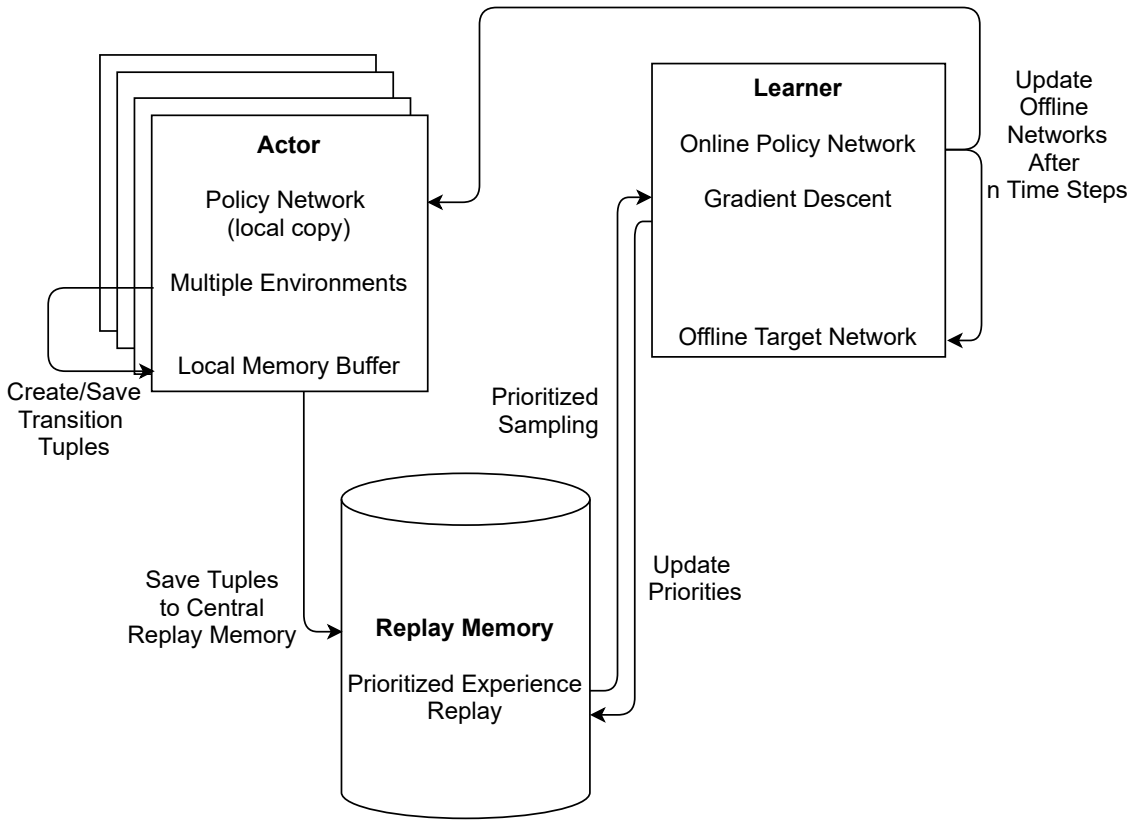

Figure 4.5: General outline of the process structure for Deep Q Learning.

Besides, in the learner process, there is an evaluation routine which is triggered after a defined number of steps which calculates different metrics to gain insight into the performance of the agent's policy.

The **replay memory process** consists of a few key components: A container to store and sample transition tuples, and communication channels from the actors and to the learner.
In the simplest realization, one such container to store data could just be a long list or a huge array in which one can save transition tuples. The sampling would then happen with uniform probabilities for each sample.

As a more sophisticated approach, we use *prioritized experience replay* [41]. Here, the container which stores the data is a sum tree. This in turn, consists of a way to store data at the leafs of this tree as well as a way of measuring and storing the priority at each leaf. The whole tree is then built by summing up the priorities of the two child leaves. In our case, the priority of each data point, i.e. transition tuple, is the absolute loss that the policy network outputs.
The sum tree structure allows sampling with a notion of priority, hence the name. For our case, using the absolute loss, which depends on the difference between target and policy network, allows us to sample data points in which the agent learns a lot with higher frequency.
The samples are provided by the different actor processes and are stored in the data

container, depending on the realization, along with the priority. On the other end of the process, whenever the learner process requests a batch of a fixed size, the desired number of samples is drawn from the replay memory instance and sent via the communication queue to the learner process.

The **actor processes** constantly step through the environment in the meantime. Albeit a little counter-intuitive, each actor process spawns multiple environments. This means that a multitude of realizations of the time-stacked surface code is provided in each actor process. Thus, the actor can step through multiple environments at the same time where the states of all the environments can be fed to the policy network as a batch, further accelerating the process. If one environment encounters a terminal state, be it by reaching the step limit or through the agent declaring an episode to be terminal, the environment is simply reset to a new state, and another decoding attempt starts. Technically speaking, the environment object stays the same, only the key quantities (state, qubits, initial errors, measurement errors, etc.) are reset and randomly re-initialized.
Deploying multiple actors and moreover multiple environments allows more exploration. Especially with the implementation of *prioritized experience replay*, we can step through a lot of environments and can be sure that the learner will sample the most educational samples most often.
The actor will not greedily follow the action with the optimal Q-value proposed by the network. Instead, as is common in Q Learning, we introduce an exploration parameter ϵ which dictates the probability with which a random action is chosen. This random action is chosen with a probability proportional to the Q-values of the policy network.

With each step in the environments, the resulting transition tuple (S, A, R, S', T) is first stored in a local memory buffer. Once this local buffer is filled up, all tuples in the local buffer are sent to replay memory via the corresponding communication channel.

To be able to step through the environments based on a policy, each actor has a local policy neural network with the same architecture as the one in the learner process. In fact, the actors' policy networks get updated regularly with the new learned weights of the learner's policy network. For this, each actor process has its own receiving communication channel with the learner.

4.2.2 PPO

Motivated by the advancements of [42] on another quantum computing related RL problem, we decided to venture into PPO as well. The implementation of PPO training is strongly based on [36] (see Fig. 4.6) and therefore diverts from the overall setup seen in [16].

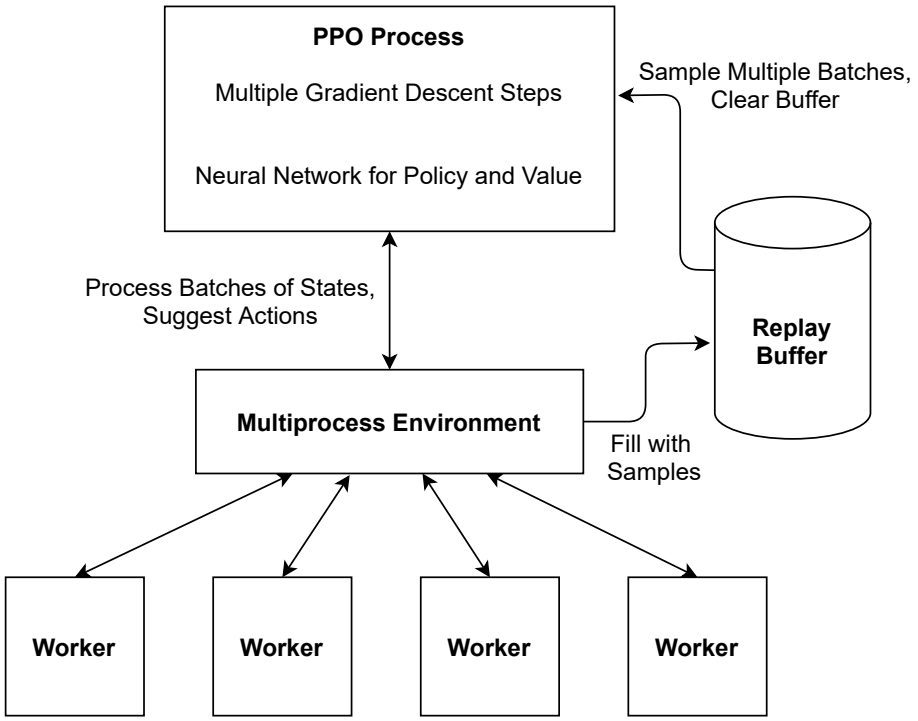

Figure 4.6: General outline of the process structure for PPO learning.

There are however some similarities to [16]: Our PPO setup still includes multiple sub-processes executing sample generating steps through independent environments. Here, we call these **worker processes** instead of actors. The workers themselves do not contain a local copy of the current two-headed policy-value network (mentioned in 3.3.2). Instead, all the current workers' states are collected and sent to the central neural network as a batch to infer the next action for each worker. The workers will then execute the step suggested by the network.

After an episode is finished, the statistics of an episode (e.g. GAE, values, rewards) are saved to a central **replay buffer**.

Once the replay buffer is sufficiently filled, we sample n iterations of b episodes from the replay buffer to perform the learning on n different batches of size b. This **optimization step** updates the central entity of the neural network governing the agent's policy π. In this step, the sampled transitions are again judged by the current version of the network. After each iteration, the network parameters and thus policy π and values V will change. But since the algorithm uses clipped quantities to compute the gradients, the assumption is that the updated network does not stray away too far from the original one that took the sampled actions in the first place.

4.2.3 Hindsight

Using the same configuration as described in the above Deep Q Learning section (4.2.1) but adding that at each step each actor stores the data in a buffer before enough is collected and sent of to the replay memory. Before the collected data is sent off, it is also filled with multiple instances of copied steps from the recent buffer with the addition of the reward being set to the maximum possible value to simulate an achieved perfect goal and thus promote exploring in the vicinity of that area.

4.2.4 Rewards

Many reinforcement learning problems yield very sparse rewards, with rewards often only given at the end of each episode. This can make it more difficult for the neural network to learn compared to other fields of deep learning such as supervised learning.
Indeed, our main rewards consist of final rewards which take into account the state and the qubit configuration at the end of an episode. We check if logical errors are present in the qubit grid, i.e. if the agent's actions have led to a logical operator which causes the qubit state to leave the expected one. In such a case, the agent get punished with a large negative reward. Additionally, we count the number of remaining physical syndromes (excluding possible syndrome measurement errors) and deal out a negative reward proportional to the number of those. Lastly, if both those checks are negative, this means that there are no syndromes that were overlooked and the surface code is still in the expected quantum state. Since this is the desired outcome for all scenarios, we reward the agent with a large positive value.

As mentioned above, only giving out rewards at the end of an episode gives a very sparse feedback signal. Intending to aid training, we added an intermediate reward scheme: For this, we look at how the state evolves in each step, comparing the previous state to the new one. If the number of syndrome defects in the top layer increases, which means that the agent has accidentally created syndrome defects, the agent gets punished with a negative reward. On the other hand, a decrease in number of syndrome defects in the transition from old to new state, yields a positive reward. Here, we weigh the negative reward more strongly than the analogous positive reward. With this, we intend to stop the agent from stubbornly repeating the same action on the same qubit, effectively flipping the state of one physical qubit eternally (or until the episode is terminated due to the maximum number of steps per episode).

This intermediate reward scheme gives a more frequent feedback signal to the agent, which can be especially useful when sampling transition tuples from a central replay memory.

4.3 Training on GPU Cluster

Among the resources available to us were runtime on the Alvis cluster[2], provided by C3SE[3], which consists of a system built around several GPU nodes with the intention to aid machine learning research.

In order to run multiple instances of the program on the cluster with all the proper settings, code, packages, environments, and ensurance of reproducibility both Singularity[4] and Docker[5] containers were produced containing all the necessary code and packages. To prepare for the eventuality of different execution environments and to alleviate possible difficulties when installing packages on a remote cluster, we felt that building cohesive containers was necessary.

As the Alvis cluster is only capable of supporting singularity containers, one such container was built from a docker container.

4.4 Neural Network Architectures

As can be seen in previous work done by others such as Gabriel Lindeby and Adam Olsson [16, 15, 14] there are many kinds of artificial neural network architectures that have been utilised mainly consisting of multiple convolutional layers which then finally feeds into linear layers to determine the action.

All Q-learning networks contain an output layer with the number of neurons equal to the size of the action space (Eq. 4.1). The networks used for PPO learning contain one head of linear layers consisting of the same number of neurons. These networks contain an additional head, whose output is made up of one neuron to predict the value of the input state.

As there are many architectures to consider, we have determined to simply make it so that the network constructor can construct multiple types of networks using a configuration dictionary holding necessary information for the network construction rather than hard-code the architecture for every new iteration of a network. Some specifics of the architectures that one should take note of is briefed about below.

A more detailed summary of network configurations is given in Appendix A.

[2]https://www.c3se.chalmers.se/about/Alvis/
[3]https://www.c3se.chalmers.se/
[4]https://sylabs.io/singularity/
[5]https://www.docker.com/resources/what-container

4.4.1 Convolutional Network Configurations

The network takes as part of the input a single list. The list configures the initial convolutional layers by taking the numbers in the list as the number of input channels and creates as many convolutional layers as there are numbers in the list with ReLU layers in-between them.

4.4.2 2D Convolution and Recurrent Module

The 2D convolutional network uses as the name suggests 2D convolutional layers for the initial part. However, before feeding the output from the convolutional layers into the linear layers it can be fed into either an LSTM, RNN, or gated transformer. The convolutional module will treat each slice of the syndrome stack as a single one-channel input. At the end of the convolutional module, the resulting feature maps of size $(d+1) \times (d+1)$ are flattened and saved as a sequence of vectors. This sequence is then fed to the recurrent unit of choice. Those recurrent units are there to consider the time aspect of the input as the 2D convolutional layers only considers each time slice by itself.

There also exists a version of the 2D Conv network type without any recurrent unit. In that case, the feature map from the last convolutional layer is simply flattened and fed to a fully-connected layer.

4.4.3 3D Convolution

As the 3D-convolutional network already convolves through the time dimension, there is no need for an LSTM, RNN, or gated transformer. The output can directly be fed into the linear layers, hopefully with sufficient information about the temporal properties of the input state.

5
Results

5.1 Results

In this section, we will present and discuss our results and training progress for decoder agents on the surface code.

5.1.1 General Training

We start with a general comparison of the training process of the different learning strategies we discussed earlier. The aim is to discuss the behavior of different agents during the learning process to gain knowledge about possible adaptability, e.g. for larger system sizes.

All values shown in this section stem from our custom tensorboard[1] value logging during randomly initialized evaluation episodes which are completely independent of the actual gradient descent learning steps. That said, the amount of steps that can be seen on the x-axes do correspond to the number of actual learner steps that have passed before the evaluation routine is performed. The evaluation routine is triggered after every n learner steps (n is usually around $100 - 250$) and has the agent facing a multitude of episodes simultaneously (usually around 128 episodes per execution of the evaluation routine).

Fig. 5.1 shows how the number of steps per episode evolves over time for different agents and at different system sizes. This is a good clue of when the agent has learned to use the terminal action. Whether or not the terminal action was used in a beneficial scenario can be seen by looking at other metrics which we will discuss below. Since we refrained from showing unsuccessful examples in this graph, we can safely assume that the agent will eventually have learned to use the terminal action to its advantage.

Every time the number of steps per episode drops rapidly, this is a hint that the agent has learned through exploring different scenarios that calling the terminal action in the right situation can be advantageous: Calling the terminal action on a (nearly) syndrome-free state yields a high reward and causes the agent to learn its benefit.

[1] https://www.tensorflow.org/tensorboard

In the mentioned figure, Fig. 5.1, we see that all the 3D Conv agents take a little time to bring their steps per episode down but once they have learned to use the terminal action, the number of steps per episode drops rather quickly. This also hints at the strong learning connected to those events: With a large reward looming for a perfectly solved episode, this will cause the loss function (Eqs. 3.4, 3.10) to be large as well which will also lead to steeper gradients for the network parameters.

We can further see a hierarchy of training speed based on the number of steps: Focusing on the 3D Conv Q-learning agents for now, we see that the agent trained on a smaller system ($d = h = 5$) *discovers* the terminal action sooner than an agent trained on a larger system (e.g. $d = h = 7$). This behavior is expected, as in the latter case, the action space is vastly larger (see Eq. 4.1), therefore it takes longer to randomly explore the terminal action in a beneficial scenario. Looking at the 3D Conv PPO ($d = h = 5$) agent now, we can see that the drop in steps per episode happens even earlier than for the same architecture trained on the same system size. The only difference is the training algorithm used.

While this might seem very promising for the PPO algorithm, we were not able to reproduce its training ability on larger systems; already at $d = h = 7$, the algorithm causes the agents to get stuck. We observed that the agents were either stuck at the maximum number of allowed steps per episode or in other cases they immediately dropped to one step per evaluation episode. Surely, this behavior will also be reflected in the workers, making it vastly more difficult to explore beneficial actions. We hypothesize that PPO training is very sensitive to the chosen hyperparameters and despite trying out different hyperparameter settings, we could not make the agents learn a useful decoding scheme at larger systems using the PPO algorithm.

On that note, we suspect that for the underlying system a better learning strategy going forward is one that includes a certain filtering of feasible actions. Since we would not want an agent to operate on qubits unrelated to any syndrome measurement, we can discard any actions on those from the temporary action space. This mechanism could then possibly be combined with Q-learning, PPO, or any other learning algorithm of choice. The added computational overhead of filtering feasible actions is assumed to be outweighed by the benefit of having a reduced action space.

Looking at the remaining graphs in Fig. 5.1, namely the 2D Conv + GRU agents trained by transfer learning on system sizes $d = h = 5$ and $d = h = 7$, respectively, we see a more gradual transition towards fewer steps per episode than for their 3D Conv counterparts. In general, these agents seemed to have more difficulties learning which could be related to their amount of added complexity due to the recurrent unit between the convolutional module and the output module. As we see in further plots, the latter agent ($d = h = 7$) has not even shown full convergence to comparable performance in terms of Q-values or ground state rate as other agents have achieved.

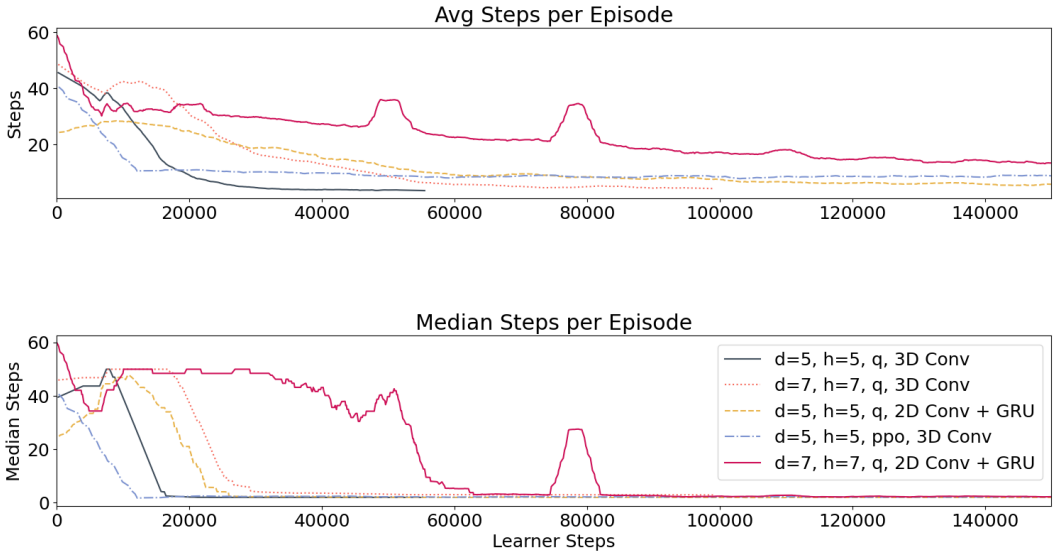

Figure 5.1: Comparison of average number of steps per evaluation episode for different learning strategies. A sudden drop of steps per episode hints at the agent having discovered suitable scenarios where it is reasonable to call the terminal action. The suddenness of the drop in number of steps is even more pronounced in the bottom plot which shows the median number of steps. Note that the x-axis shows the number of learner steps that have passed while the y-axis shows the number of steps per evaluation episode.

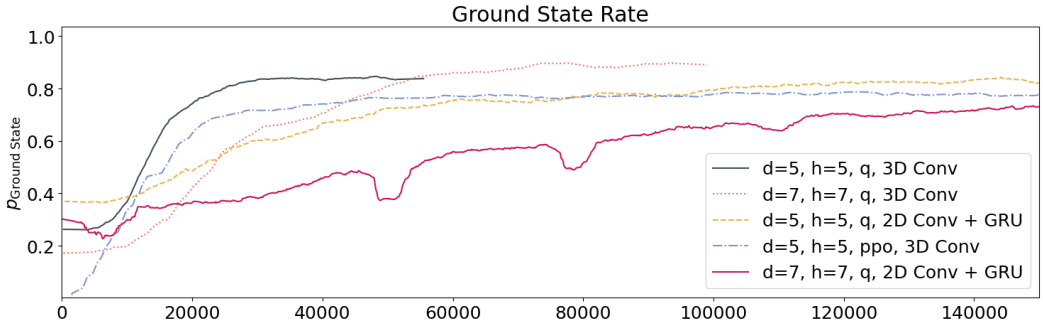

Figure 5.2: Comparison of average ground state rate for each evaluation episode for different learning strategies, meaning the percentage of episodes that have neither remaining syndromes nor an unwanted logical operator.

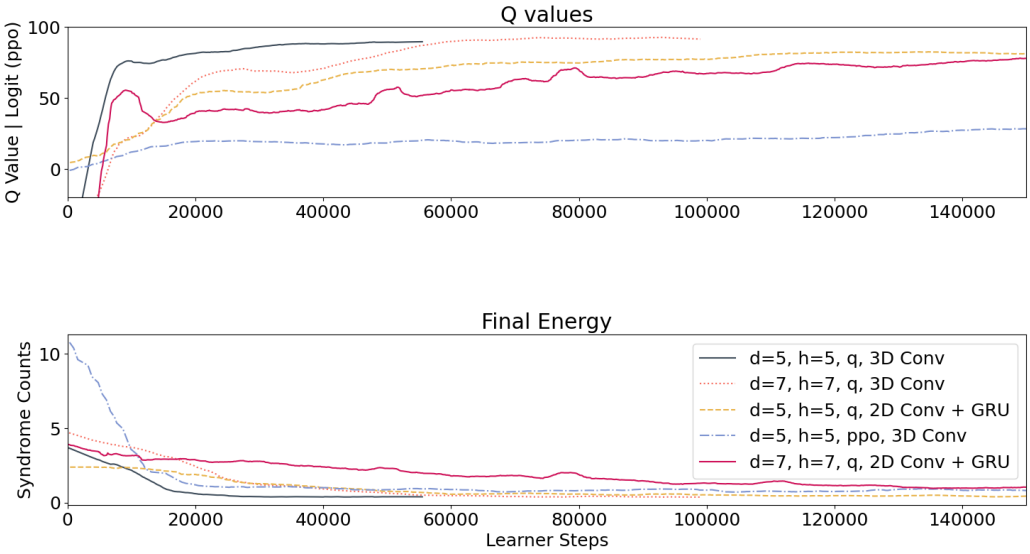

Figure 5.3: Top: Evolution of maximum Q value in each step and in each episode. Comparison between different learning strategies. For the PPO learning, instead of showing Q values, the graph (red) shows the logits corresponding to the greedily-chosen action.
Bottom: Shows the evolution of average final energy of the syndrome stack, i.e. the number of syndromes that remain after the final step.

If we look at Fig. 5.2, we see the evolution of how often the agent manages to recover the ground state of a given system in the evaluation routine. In our work, the ground state describes the original qubit state without any unwanted logical operations on the encoded qubit. Again, we see the trend that this rate is growing slower for the larger system ($d = h = 7$) and even slower for the complex 2D Conv + GRU model. The curve for this model type at the larger system still has not reached the possible level of performance which we would expect from looking at the other agents.

Interestingly, the more complex models start out at a higher ground state rate, possibly due to the aspect of transfer learning where the bottom-most convolutional layers were taken from a fully-trained network. That could arguably make it easier for the network to infer possible good actions.

Lastly, we compare the evolution of the maximum Q-value (or logit[2], in the case of PPO) as well as the evolution of final syndrome energy in Fig. 5.3. The top graph shows the average Q-value across all steps and all episodes in one execution of the evaluation routine. With the maximum reward at $R_{\max} = 100$ for a perfectly solved episode, we expect the graph to converge somewhere close to but below $R_{\max}$ since this value should get discounted at earlier steps in an episode.

For the PPO run, of course we cannot keep track of any Q-values, instead we show the evolution of the corresponding maximum logit value, averaged over steps and episodes. Since the logits values will be converted to probabilities to construct the policy π, we cannot give an estimate or expectation of where the value should converge to. On the contrary, we even see the maximum logits value to grow slightly towards the end of the learning process where we are rather certain that the agent has properly learned the decoding task. This hints at the agent becoming more and more certain about its actions, although to prove this claim, one would have to look at the difference to the next-largest logit or even take into account the whole distribution of logits or any other measurement that could translate to certainty in a decision.

The final energy is simply a measure of how many syndromes (excluding syndrome measurement errors) remain after the final step of an episode, for the bottom graph, this value is averaged across all episodes in one evaluation routine execution. We expect this value to converge towards 0, so that no syndromes are left after a decoding cycle.

We recognize similar trends as before: Agents on smaller systems learn and thus converge quicker to the expected values, where the more complex 2D Conv + GRU architecture takes longer to train.

One interesting aspect is that the PPO run starts at much higher energy values than all the other agents. We are not sure why that is but we suspect that the answer could also lead towards stabilizing the PPO training and thus enabling training for larger systems. One possible explanation is that agents in the Q-learning tend to repeat one particular action indefinitely while they are still fairly untrained. This limits the number of introduced syndromes in the worst case. PPO-trained agents may choose actions more randomly across the qubit grid, leading to a lot of different chosen actions in one episode, of which the majority will be detrimental while the agent is still untrained.

[2]Here, *logit* describes the unnormalized values from which a probability distribution can be constructed, for example by applying the softmax function.

5.1.2 Hindsight Training

Looking at the results produced by implementing Hindsight learning into the deep Q-learning one can see the advantages and disadvantages of it. The figures 5.4, 5.5, illustrates these. Figure 5.4 shows how similarly configured agents with only one or two adjustments from each other with either regards to using hindsight, 2D or 3D convolutional layers and stack depth 7 or 9. If one inspects them, one sees that agents utilising hindsight tend to learn faster initially as it is visible the initial phase for both depth 7 and 9 (when compared to the same agents training on the same depth) on the ground state and final energy plots that the network using 2D and hindsight. This is somewhat expected as hindsight does promote exploration and learning how to access further states from the initial state.

However, the agents in these figures stagnates in learning and does not as one sees from the charts learn to use the terminal action as the average and median steps per episode is at their maximum per episode for all episodes. This is also likely due to the same reason as to why hindsight improves the exploration, the false goals. As in the case of our implementation of hindsight the false goals are sampled randomly from previously taken actions which are unlikely to include the terminal action thus promoting actions that is aimed towards changing the current state which may or may not be the optimal option. The problem increases with the code size as there are now more actions and states that are not the terminal one. This in turn shows on the learning progress as the agent being impeded by what was initially its advantage. With that said, the agents can still learn to use the terminal action and achieve great results as shown by figures at 5.5. However, the results show clearly that one has to be aware of these unintentional effects of the method and plan accordingly. With that said, it should be noted over the several runs done that using hindsight caused the agent to have a higher chance to learn at all as when training at certain code sizes and error rates. An example being that over multiple runs at $d = 7$ and $p_{err} = 0.01$ most runs without hindsight failed while most runs with hindsight succeeded at learning.

5.1.3 Performance Comparison

Figure 5.6 shows how different agents, comprised of different neural network architectures or trained by different learning strategies, fare on the base system of $d = h = 5$. We take into account the potentially varying number of syndrome layers. These layers correspond to one measurement cycle. Therefore, a deeper stack inherently captures a longer life cycle of the surface code grid. For this reason, and for better comparability, we scale the failure rate to be the fail rate per cycle (i.e. per layer). This allows better comparison to the one-layer setup [14, 15, 16] and to lifetime analysis done by Sweke et al. [24].

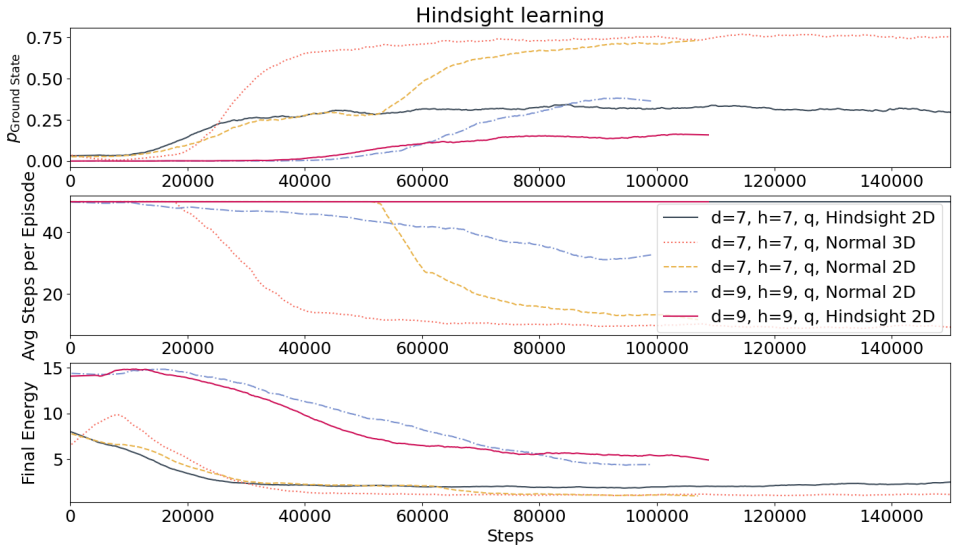

Figure 5.4: Plots of the evaluations of agents with similar configuration. Difference being in usage of hindsight, stack depth and whether it is using 3D or 2D convolutional layers. Images picked to illustrate certain specific strengths and weakness of hindsight

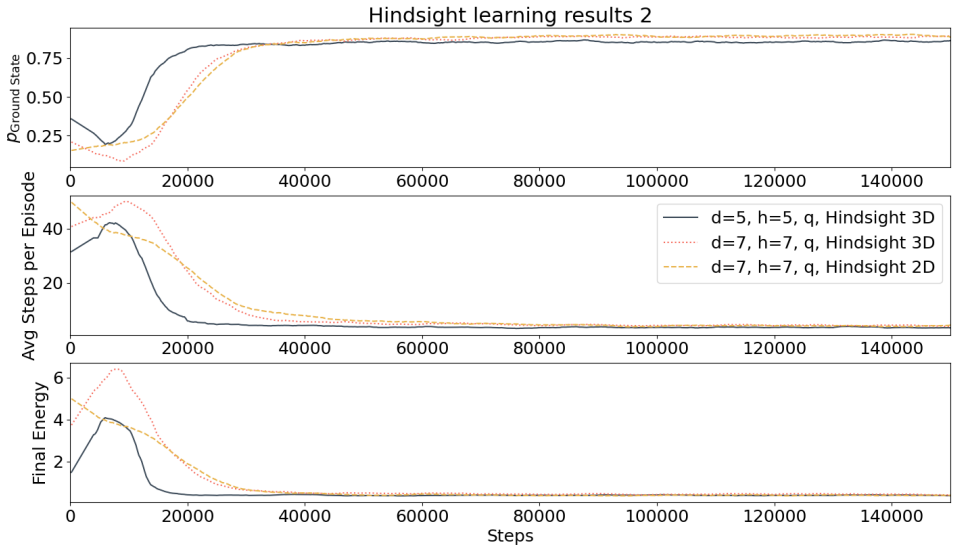

Figure 5.5: Plots showing that agents utilising hindsight *can* learn the terminal step and achieve great results as opposing the previous plots.

We immediately notice the formation of some sort of band structure regarding agent fail rate. Astoundingly, the transfer-trained complex 2D Conv + GRU architecture (black circles) performs comparable to an agent trained without any measurement errors (orange triangles). The former does not cause as many episodes with remaining syndromes however, so there seems to be some kind of recognition of measurement errors in the training process. Still, the performance of this architecture is underwhelming and below expectations.

In the second band, we find 3D Conv models each trained by different learning algorithms, namely Q Learning and PPO. These seem to perform on an identical level within error bars. The most surprising however is probably the performance of the plain 2D Conv model. Despite having no clear module dedicated to infering time series relations, this model has the lowest fail rate and the lowest percentage of episode with remaining syndromes across all tested error rates.

We expected other achitectures like 3D Conv, with its inherent time-dimensional convolutions, or 2D Conv + GRU, with the built-in recurrent network unit to analyse time series, to show better performance. The plain 2D Conv model simply creates a feature vector for each layer in a syndrome stack and then combines those to feed them to a fully-connected network head. Of course, we don't deny the ability of a fully-connected module to learn, but conceptually, the afore-mentioned architectures seem like more natural choices for handling the time aspect in the task at hand. We still suspect the benefits of these architectures to have a stronger impact with growing system sizes; in these applications, the plain 2D Conv model will most likely suffer from too large dimensions in the fully-connected network.

5.1.3.1 Hindsight Learning Comparison

Comparing the fail rate of agents with hindsight and without one can see that for $d = 5$ the hindsight improves the result but for larger code sizes this is no longer the case as to which one is performing better (see figure 5.7). This is again most likely due to the same reason as stated in the earlier section 5.1.2. Among the larger code sizes the hindsight implementation starts hindering the exploitation part of the agent and thus starts performing sub-optimal and the results are lowered to the same as without hindsight or in some cases even lesser performance.

5.1.4 Stack Depth

One area of interest is how the stack depth impacts the agents' performance. The hope is that with growing stack depth, i.e. a larger collection of measurement cycles, the agent is able to resolve qubit errors further in time and possibly identify physical errors as they happen at a distinct point in time. Depending on the interpretation of the error rate however, more layers also leads to more errors being introduced, both physical and measurement errors alike. All training and evaluations in this section were done for $d = 5$ and varying h.

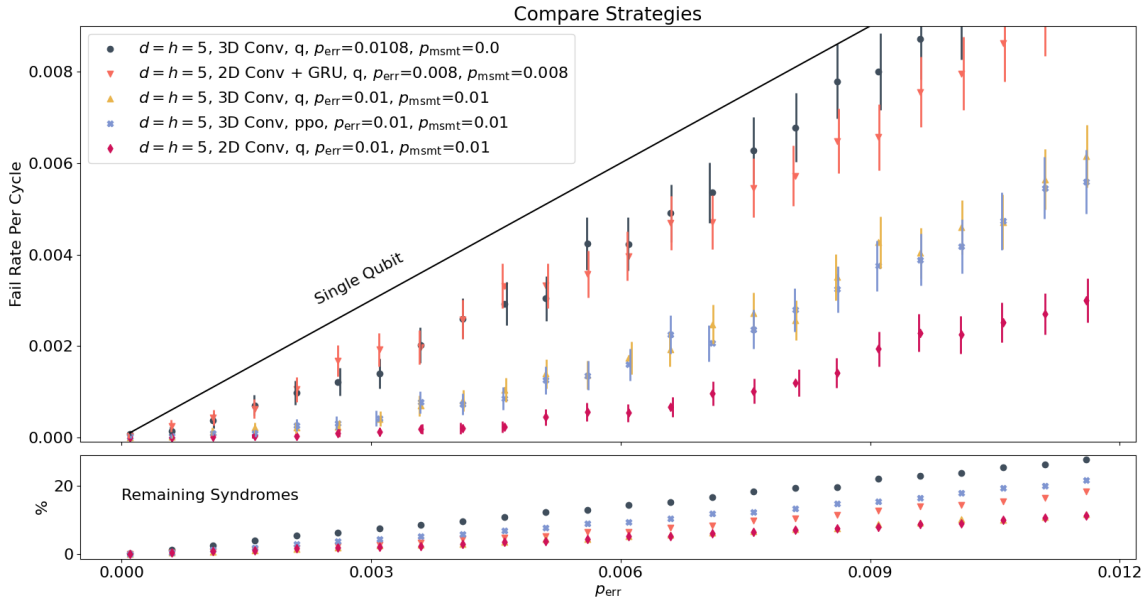

Figure 5.6: Comparison of different agents' performance on the same base system. The bottom part shows the percentage of episodes that had syndromes left and are regarded as having an undetermined outcome. Each data point in the bottom graph corresponds to the the data point at the same error rate in the top graph. Note that the error rate used in the legend refers to the static error rate the agent was trained on. The error rate in the x-axis corresponds to the spectrum of error rates in the post-training evaluation.

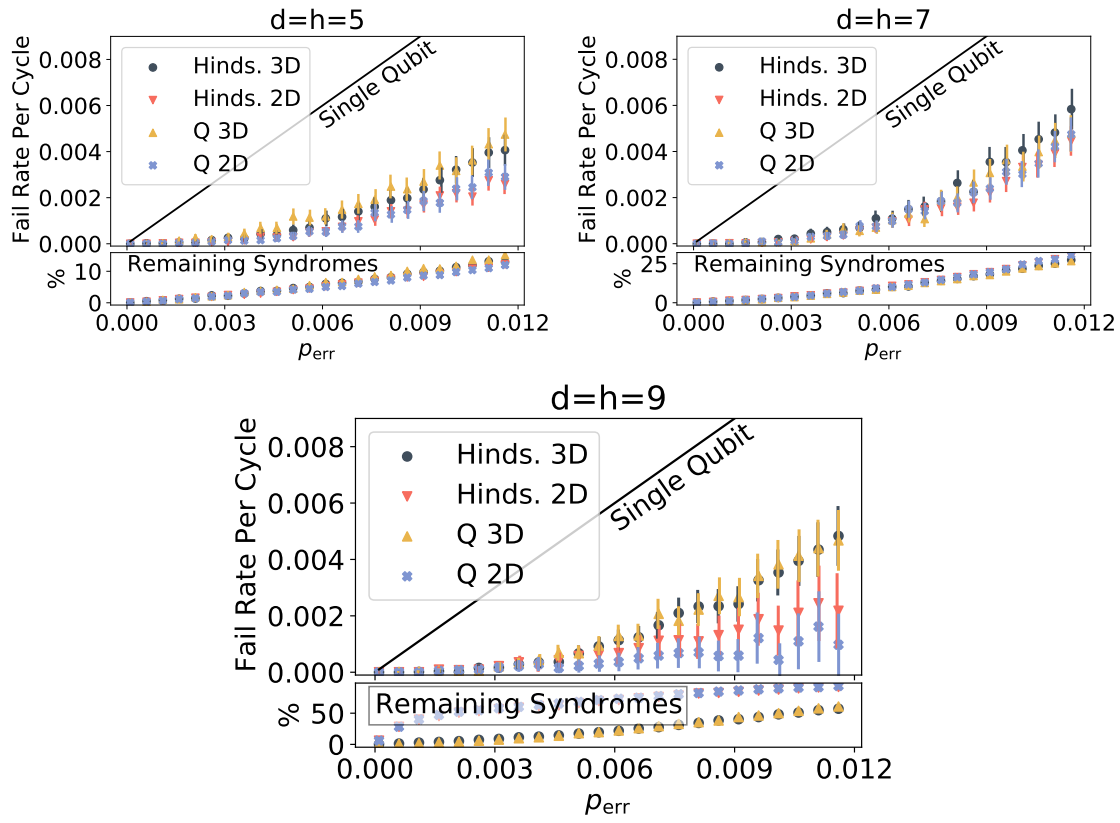

Figure 5.7: Comparison of different agents' performance with aspect to the fail rate. The figure are of agents using different code sizes and stack depth with the uppermost using $d = h = 5$, middle $d = h = 7$, and bottom $d = h = 9$. As shown, the performance of agents utilising hindsight surpasses that of those without at smaller code sizes but struggles more at larger sizes.

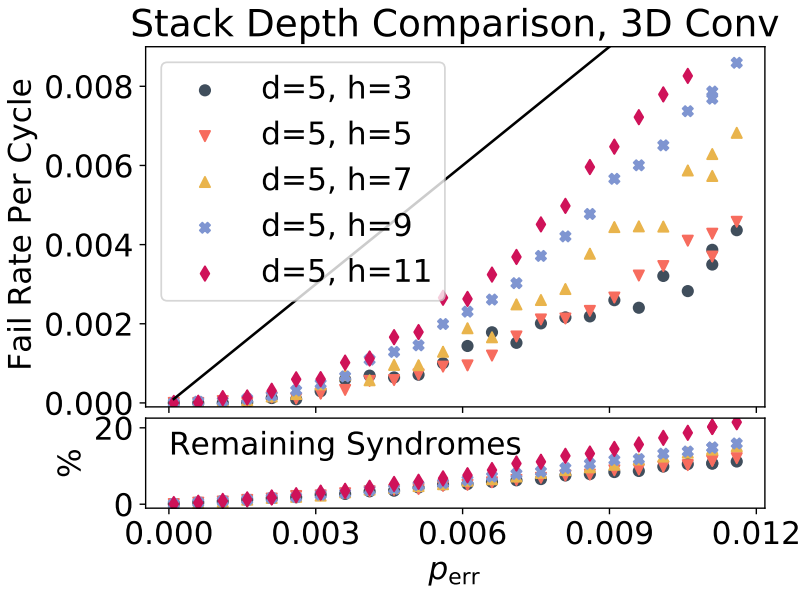

Figure 5.8: Comparison of how agents perform on systems with differing stack depth at different error rates. The code size is $d = 5$ for all. The black line shows the single-qubit error rate for comparison. We used 3D Conv agents for this comparison. The bottom graph shows the percentage of evaluation episodes with syndromes remaining at the end of each decoding cycle and shares the same x-axis as the top plot. These episodes are excluded from the main analysis.

Generally, we analyze the performance in terms of the fail rate or logical error rate; a measurement of what fraction of episodes had an unwanted non-trivial loop introduced.

The first result to be discussed is the performance of agents on systems with different stack depths, where the expected number of introduced errors is roughly the same. We arrive at the expected number of errors the following way: $\overline{n_{\text{err}}} = d^2 h p_{\text{err}}$. Since we chose $p_{\text{err}} = p_{\text{msmt}}$ in all runs, this means that there is an equal number of physical qubit errors and syndrome measurement errors to be expected to occur in the stack, i.e. $\overline{n_{\text{err}}} = \overline{n_{\text{msmt}}}$. The difference between the two error types is however, that a physical error introduced in one of the lower layers will persist vertically (i.e. in the time dimension) in the stack whereas measurement errors are confined to the point in the syndrome stack where they occur.

If we compare the systems across a range of error rates, like in Fig. 5.8, we see that in fact the growing number of errors outweighs the benefits of better temporal resolution, as we discussed above. For the same error rate - interpreted as the probability of introducing an error at each qubit in each layer - the systems with the lowest stack depth yield the lowest failure rate. The low number of errors in-

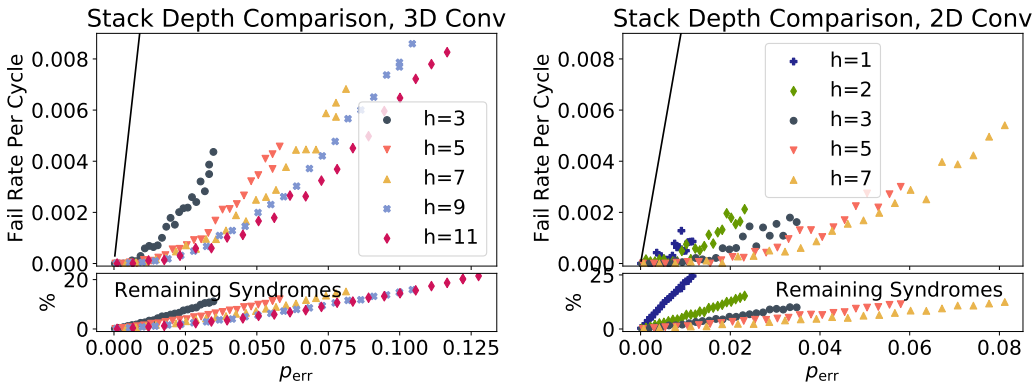

Figure 5.9: Comparison of how agents perform on systems with differing stack depth when faced with comparable numbers of errors expected in the system. The code size is $d = 5$ for all. The error rate is rescaled such that the error rate $p_{\mathrm{err}}^{\text{one layer}}$ corresponds to the value of the standard 2-dimensional surface code without any additional layers, therefore each vertical slice preserves the expected number of errors in the syndrome stack. Left: 3D Conv agents, right: plain 2D Conv agents. The bottom graph shows the fraction of evaluation episodes with syndromes remaining at the end of each decoding cycle and shares the same x-axis as the top plot. These episodes are excluded from the main analysis.

troduced in the case of $h = 3$ seems to be beneficial for the agent. For $h = 11$ on the other hand, the agent will be faced with too many net errors, both physical and measurement errors, to be able to restore the qubit state.

Since the above discussion might be unfairly biased towards low stack depths, we rescale the error rate to be interpreted as the error rate as it would be in the case of a single layer of the surface code instance. This way, we can again compare the performance at similar expected total numbers of introduced errors. As a reference, at $p_{\mathrm{err}}^{\text{one layer}} = 0.05$ for example, we expect $\overline{n_{\mathrm{err}}} = 1.25$.

In Fig. 5.9, we see that with this rescaled error rate, deeper syndrome stacks prove to be more beneficial again. Faced with the same number of errors, the agents seem to be able to resolve error chains better when a larger temporal sequence is available. Also, in this representation, measurement errors, which again are more localized, are sparser and therefore possibly easier to spot for the agent. A deeper syndrome stack allows the agent to be more certain about syndromes caused by physical errors since their average depth in the syndrome stack can be larger.

The right-hand side of Fig. 5.9 shows similar results as described before but for 2D Conv agents trained on varying stack depths. Here, we see a similar trend but generally the error rates are lower for the same stack depth. A more detailed analysis of the impact of the time dimension on an agent's performance is given in Sec. 5.1.6.

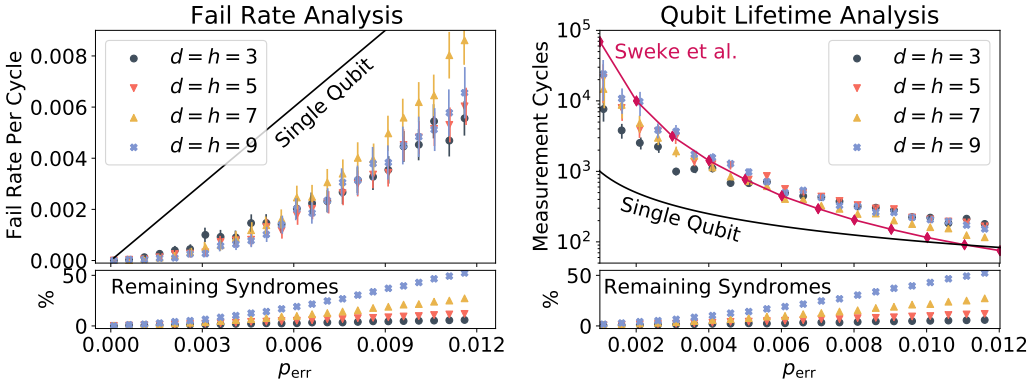

Figure 5.10: Left: Fail rate or logical error rate for different system sizes. The bottom graph shows the fraction of evaluation episodes with syndromes remaining at the end of each decoding cycle and shares the same x-axis as the top plot. These episodes are excluded from the main analysis. Right: Different expected lifetimes of logical qubits. This can be interpreted as the average number of measurement cycles until a logical error will occur. Sweke et al. performed measurements on $d = h = 5$.

5.1.5 Of Failure and Lifetime

One important question is how the agents' performance scales with growing code size. With varying code size, it seems to be common (according to the analysis by e.g. [22, 24]) to choose the stack depth $h = d$ when taking into account syndrome measurement errors. In this section, we present the results of agents trained on different system sizes.

In Fig. 5.10, on the left-hand side we see how agents trained on different system sizes compare to one another. There is no clear visible trend in the behavior of success rate, nor is there a clear threshold in error rate that one can see in the linear plot, as we would expect from [22].

The only things that really stand out could be caused by agents performing below their potential: For $d = h = 7$, we see a significantly higher fail rate towards the right end of the graph. For $d = h = 9$, the performance seems on par with smaller systems but we notice in the top part of the figure that here we have a significant number of episodes with remaining syndromes which we choose to disregard since it is not entirely clear how to handle these episodes.

Looking at Fig. 5.11 which shows the same data in logarithmic scale, there is a slight indication of a threshold error rate $p_{\text{err}}^{\text{thr}}$ above $p_{\text{err}} = 0.005$. The motivation for this conjecture comes from the performance in terms of fail rate at opposite ends of the depicted error rates: On the left-hand side, smaller system sizes show a larger error rate than larger systems. On the other end, the small systems $d = 3$ and $d = 5$ show the lowest fail rate. This leads one to believe that a crossover point could exist in between. Such a crossover point would define the threshold error rate (see [22]). However, with the present data, which turns out to be quite noisy, it is not

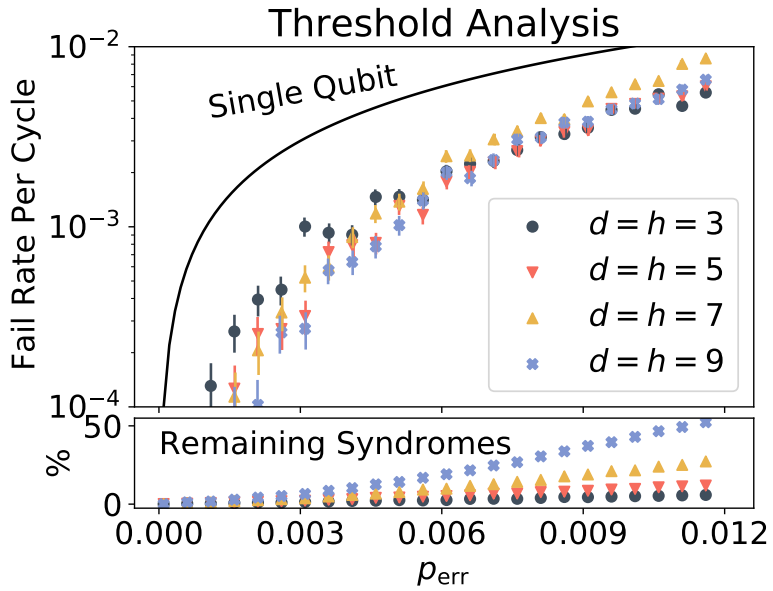

Figure 5.11: Visualization of fail rate for different system sizes in logarithmic scale. The solid black line shows the fail rate of a single qubit for comparison. The bottom graph shows the percentage of episodes with remaining syndromes at the end of a decoding episode for the corresponding agent above. Note that both plots share the same x-axis.

possible to find one particular point of intersection as the threshold error rate of the surface code. The search for such a point is further hampered by the underwhelming performance of the agent at $d = 7$, showing somewhat anomalous behavior and unexpectedly high fail rates especially towards larger error rates. We conclude that for a threshold analysis, more work needs to be done: On one hand, the agents should be trained to near-optimal performance to enable a fair comparison; secondly, more post-training evaluation samples are needed to improve the statistics at such low fail rates.

Rescaling the fail rate to represent the fail rate per cycle also allows to calculate the expected lifetime of the logical qubit. We do so in the right plot in Fig. 5.10 where we also compare to the results from Sweke et al. [24]. Their paper explicitly focused on maximizing qubit lifetime in their training process, so their setup as well as their state formulation in their implementation is different from ours.

The results show that our agents have a higher expected lifetime towards higher error rates than the Sweke models. In fact, whilst the Sweke measurements drop below the single-qubit line at some point, our agents consistently stay above this line for the investigated error rates. On the other side of the graph, where there is a miniscule amount of errors introduced, Sweke et al. obtain better qubit lifetimes.

5.1.6 Network Activation

To investigate the decision making process of the agents a bit more and confirm the proper working of the neural networks, we will take a look at the Q value output of a network when faced with different state inputs. One goal is to confirm that the network takes into account the measurement history. An alternative scenario one could think of is an agent only focusing on the top layer and ignoring the rest of the stack. This would defeat the purpose of using multiple syndrome layers and would not progress the problem formulation beyond the already investigated single layer setup.

For this, we prepare a number of deterministic syndrome states, which all are some thinkable variations of the surface code state in Fig. 5.12, which we will refer to as basic state. The basic state in Fig. 5.12 shows one qubit error which exists at least in the top-most layer and – depending on the scenario – might exist in lower layers as well. We now construct different possible scenarios that could alter the basic state and hence alters the input to the neural network. In turn, this should also influence the Q value output of the network, which is exactly what we want to investigate in this section. Some state alterations may include syndrome measurement errors (states 1, 3, 4, 5) and longer qubit error existence than merely in the top-most layer (states 1, 4, 5, 6). The states are described in more detail in the following paragraphs.

The actual derived states we chose for this analysis are shown on the right-hand side of Fig. 5.13. Whilst Fig. 5.12 gives a birds-eye view of the basic system setup, the columns on the right-hand side of Fig. 5.13 show how these two active syndrome defects are represented in the vertical time dimension. A filled space corresponds to an active syndrome defect hinting at the presence of qubit errors in one of the surrounding qubits. The color coding corresponds to the index of each state variation that we prepared.

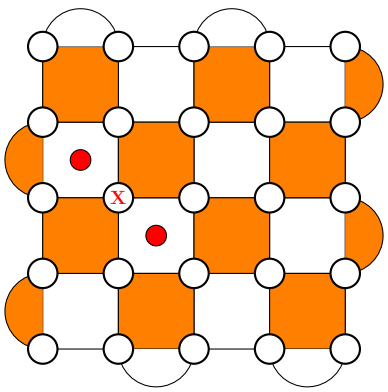

Figure 5.12: Birds-eye view of the basic status of the surface code grid used for the analysis of network activation. Qubit $(1, 2)$ is subject to a Pauli-X error which results in the two adjacent vertex defects to be activated. For simplicity, only the top layer of the basic state is shown.

We will first describe the different states, and which alteration to the basic state is depicted, in more detail (Fig. 5.13, right):
State 1 (black) can be thought of as two syndrome measurement errors occurring on syndrome $(2, 3)$, thereby inactivating the syndrome defect. The qubit error depth also happens to be two in this scenario. This is a highly unlikely case with two syndrome measurement errors happening in the same place in successive measurements. For this and other states, different interpretations of potential causes for the syndrome state are possible which emphasizes the general decoding problem at hand.
State 2 (orange) sees a qubit error introduced in the final measurement cycle. This corresponds exactly to the basic state in Fig. 5.12.
State 3 (yellow) can be thought of a syndrome measurement introduced at $(2, 3)$ in the final layer.
State 4 (blue) can be seen as a physical error happening in the second-to-last cycle with a measurement error introduced in the final layer to negate the syndrome defect at $(2, 3)$.
State 5 (red) represents a qubit that persists through all measurement cycles only to be interrupted by a measurement error in the last layer.
State 6 (green) serves as a reference which should give the agent the highest certainty: Here, there is only one qubit error that exists in all measurement cycles without any interfering measurement errors.

To the left of that, in Fig. 5.13, we see the output of the neural network's final layer which we interpret as Q values. We show the Q value proposed by the network (y axis) for each action index (x axis). Each action index in turn corresponds to a 3-tuple which encodes which action should be performed on which qubit in the form of (x-coordinate, y-coordinate, operator index). The chosen action tuple is also shown on the right hand side on the figure. A deterministic agent will choose the action corresponding to the highest Q value. Ideally, this would be a clear spike in Q values for one particular action.

If we again analyze the states one by one, we get the following results:
State 1 (black) shows a relatively low Q value, even more so it proposes a completely different action from all the other examples. This reflects the very uncertain nature of the input state where only one syndrome defect is active whereas a qubit error of type X usually causes two syndrome defects (unless it is on the edge of the grid). The chosen action is also adjacent to the single activated syndrome defect.

State 2 (orange) chooses the expected action $(1, 2, 1)$ to counteract the X error on the qubit at $(1, 2)$ despite a very short syndrome presence.

State 3 (yellow) shows a large spike at action $(0, 0, 4)$ which corresponds to the terminal action. Hence we can say that the network was able to identify a syndrome defect isolated in space and time as a syndrome defect.

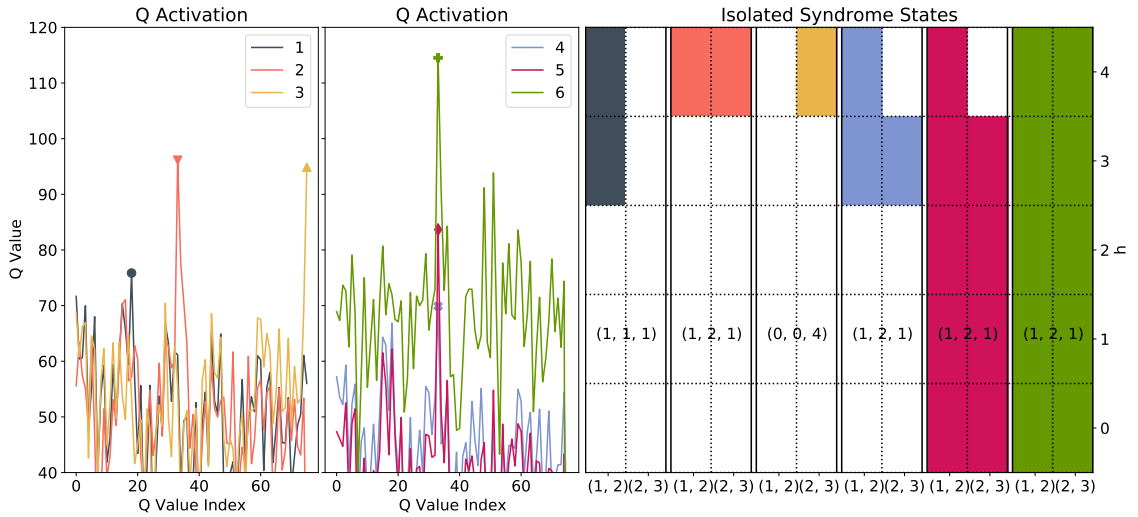

Figure 5.13: Left: Q value output for different scenarios of a 2D Conv network at $d = 5$. Right: Representation of the two relevant syndrome defects in time. The color coding serves to distinguish different states and their activation.

States 4, 5, and 6 (blue, red, green) are more similar to each other. For all of these, the agent was able to suggest the expected action $(1, 2, 1)$. The difference lies in the certainty of the chosen action which is manifested in the Q value output. State 6 with no interruption to any of the two relevant syndrome defects yields the highest Q value output for the chosen action. A syndrome measurement in the stack (state 5) causes the Q value for the chosen action to be lower. A late error with syndrome defect (state 4) makes the network even less certain with the Q value for the proposed action plummeting to around 70 compared to the previous cases. One can actually see the same action index as for state 1 to have a large activation.

With this analysis we can see that the trained networks are able to react reasonably to different scenarios and are able to differentiate between input states that are similar in nature.

6
Conclusion

We developed a new state representation for the partially observable Markov decision process (POMDP) problem of decoding the surface code in the presence of faulty measurement errors. Based on this, we were able to develop and train agents for the decoding task using different neural network architectures and reinforcement learning techniques. We compared the different results and gave insight and motivations for choosing different architectures and learning strategies.

6.1 Impact of Stack Depth

In section 5.1.4, we investigate the impact of the number of considered measurement cycles. Figure 5.9 shows that on a rescaled error rate, deeper syndrome stacks do provide some advantage over shallower syndrome stacks. This might suggest that at least with a given expected number of errors, the better temporal resolution and increased sparsity of more layers makes it easier for the agent to decode the syndrome. However, tracking more measurement cycles also means introducing more qubit and measurement errors, as Fig. 5.8 suggests. Therefore, we cannot make a definite statement about the optimal number of layers, and stick to the convention of $d = h$ in the case of $p_{\text{err}} = p_{\text{msmt}}$.

6.2 Threshold Analysis

Section 5.1.5 deals with the fail rate at different system sizes d. While on first glance, almost all agents perform almost equally well, Fig. 5.11 hints at the possible presence of a threshold error rate for the surface code used in this work. The threshold error rate $p_{\text{err}}^{\text{th}}$ is an important quantity to describe the capability of an encoding when scaling the code distance d towards higher values. To quantify an approximate value for $p_{\text{err}}^{\text{th}}$, more precise evaluation runs need to be done.

We also obtained results on par with and arguably better than Sweke et al. [24] in terms of average lifetime of the logical qubit. Our model seems to perform better for higher error rates p_{err} although it was not trained on the task to keep the qubit intact as long as possible.

6.3 Network Architectures

Surprisingly, for the base system $d = h = 5$, the plain 2D convolutional network performs best. When scaling up the surface code to $d = 7$ or higher, the learning progress is significantly hampered and training is not as reliable if at all possible.

Another attempt was to combine the 2D convolutional structure with a recurrent unit like GRU, LSTM, or Transformer modules. This architecture proved difficult to train and the only way to obtain successful learning was with the help of transfer learning, where we used pre-trained 2D convolutional layers. Even then, the training time needed for convergence was significantly longer than its plain 2D Conv or 3D Conv counterparts. We also saw that even then, the performance was not better than the plain architectures which we blame on the added model complexity.

The most reliable architecture from a generalization standpoint was one based on 3D convolutions. The family of 3D Conv models was able to extend to larger systems and across a broad range of error rates when training.

An overview of used network architectures is given in Appendix A.

6.4 Learning Strategies

Beyond comparing different network architectures, we also tried to venture into different learning algorithms from the field of reinforcement learning.

Based on previous work [16], we were able to build our Deep Q Learning program on a computationally optimized base structure. We suspect that this gave the Q Learning approach a large advantage in general training setup.

Furthermore, we implemented a PPO realization to delve into policy gradient-based learning algorithms, with the hope that such a method would improve learning in a high-dimensional action space. And indeed, for the base system it looked promising, showing much faster convergence and seemingly discovering the terminal action in advantageous situations much quicker. Alas, we were not able to transfer these learning advancements to larger systems; the agents were stuck in local optima from which they did not seem to escape even after hours of training. We suppose that this algorithm is very sensitive to the choice of hyperparameters, including but not limited to local memory buffer configuration and training parameters such as learning rate and gradient clipping. Our hope is, that with more time for hyperparameter tuning, one could still make training via PPO work.

Lastly, while hindsight proved effective at improving learning, the current implementation of Deep Q Learning made it difficult to implement hindsight sampling of steps focused on the end of an episode, which would have increased the chances of learning the terminal action. This is due to the steps being gathered into a local

buffer before being put into the replay memory. Because of this, the local buffer can hold fractions of episodes where the location of the beginning and end of an episode are not defined. This in turn makes it hard to specifically select near terminal (or terminal) steps without either making the code error-prone or sacrificing the current implementation of the Deep Q Learning. The latter sacrifice would also slow down the training unless replacing the optimized structure which would partly go against the point of hindsight learning.

With that said, the problems only seem to have to be considered once the action space is large as agents with small action space still learn the terminal action. One could also attempt to implement hindsight learning to subside over time and thus have the training gain the initial strength of hindsight and ignore the problems for latter stages. However, if one wants to use hindsight to the end, one could use another implementation of off-policy learning method than ours.

6.5 Future Work

Here, we aim to touch upon some points that could benefit the general program setup in the future.

6.5.1 Generalized Architecture

In the future, one might want to investigate the possibility of using a network architecture that is more generalizable. Right now, each system size requires at least the fully-connected hidden layers to vary in size. It might be beneficial to employ a fixed architecture that is capable of training on different network sizes. One key concept for this might be a fully-convolutional approach, where the output is also given by a convolutional layer, motivated by the example in [42]. Or one has to implement some kind of pooling (maybe inspired by ROI pooling [43]) to feed the convolutional layers' output to fully-connected layers of fixed size.

Using a generalized architecture might also help leverage the advantages of transfer learning: Since the syndromes are local phenomenons, in principle at least the earliest convolutional layers in the architecture should work the same for different system sizes. Because of that, one could train a network on a smaller system first, and use the very same network on a larger system after some training for fine-tuning the network's parameters. In this kind of approach, one should also consider using batch normalization layers so that the hidden feature maps are not strongly dependent on the system size.

6.5.2 Split Input Toggle

Initially an idea of potentially splitting the input into different channels was explored. The different channels would accept either plaquette syndromes, vertex syndromes or the combination of both. These different channels could be constructed as one sees fit before combining the result. In our case we had separate convolutional layers for each type of syndrome defect before combining the results of the output. The idea was to see if they could adapt better if there were parts of the network looking into specific types of syndromes.
This was however put on the side as copying the online network to the offline network seemed to be having issues and with regards to the time and priorities the idea was suspended. However it would be interesting to see what would happen if agents had access to this kind of information structure.

6.5.3 Non-Constrained Channel List

Another idea that was implemented at some stage was to simply give the constructor of a specific neural network agent class a list from which it would loop through and construct a sequence of layers using the list as a blueprint. This was however scrapped as problems arose when copying and sending the network parameters between different subprocesses and due to lack of time to look into the issues. It may be interesting however for future works to look into this as it would make it easier to observe performance of different networks with different layer configurations.

6.5.4 Sparse Rewards

Lastly, the most unbiased way to actually reward an agent is to simply show it whether or not it succeeded at eradicating the errors without the "training wheels" of intermediate rewards. Attempts were made at this but no agent so far learned it. It is however proposed to try out smaller code sizes along with a pre-trained model (pre-trained with intermediate rewards) to fine-tune itself. Note also that a pre-trained model could be stuck in a local minimum from the previous training. Another thing to consider would be the addition of hindsight experience replay (we suggest improving the implementation also) as some of the earliest works was on bit-flip problems [40] which is similar to the "qubit-flip problem" of this task. It should however be noted that hindsight learning was tried without a pre-trained model for $d = 5$ and $p_{err} = 0.005$ without learning. One may also consider revising the implementation of training the agent to suit whatever strategy is tried.

6.5.5 Training for Equivalence Classes

With our current reward scheme, the agent is looking for an efficient way to get rid of any syndrome due to physical qubit errors and then terminate a decoding episode. Since one syndrome could have been caused by different qubit and measurement errors, the agent could receive conflicting rewards when faced with this special case: While in one case the network proposes the optimal correction sequence for the un-

derlying qubit error chain, another state of the same phenotype could have a vastly different error chain. In an extreme case, proposing the same actions again for this latter state could lead to the introduction of an unwanted logical operation.

Our actual goal is to teach the agents the most probable correction chain. For this, one might have to consider the equivalence class of the underlying error chain. Predicting the correct equivalence class with Monte Carlo methods is computationally costly, which is why we would like to train neural networks to perform this task instead.

Also, for this task, sparse rewards are considered beneficial. With the intermediate rewards that we introduced, the main goal was shifted towards annihilating syndromes. Thence, the impact on the final reward of whether a correction chain was in the right or wrong equivalence class would be diminished.

7
Ethical Discussion

With all things said and done, there are both pros and cons if quantum computing became part of modern technology. While some difficult problems could be solved, new ones would arise.

As mentioned in the introduction, Shor's algorithm could solve the problem of prime factorization which would crack the current RSA encryptions used by the banking world. Multiple other encryption companies could also be ruined with their encryption rendered useless. However, there are encryptions emerging to combat the dangers of quantum computing ([44, 45]). Thus, one would think that there is no problem, as all that would happen is simply that the encryption systems would be updated.
With that said, everything would have been fine if that is all that would have happened. However, what if someone copied and stored important data right now, even though it is still encrypted, information which is not meant for public eyes. Things such as government secrets, military secrets, transactions or encrypted messages, things that carries significance even long after they have been encrypted. If that someone simply waits around for when quantum computing is at a stage where it can crack the encryption, he or she can use it however they please.

Another related topic which may touch some and not others but is very relevant for some countries (that is not to say they are not the only ones using this) is that of cryptocurrencies as these utilise (as the names states) cryptography. While the need for cryptocurrencies (and decentralised finance) is not perceived as high by most people in the developed world, the need for it in other areas is quite high. This is especially true for areas where either banking services are not readily available or corruption and inflation runs rampant. In these areas cryptocurrency has grown to find usage as a store of value and as everyday payment. An example of this is El Salvador which recently (as of writing this article) has accepted bitcoin as legal tender.
If quantum computing becomes a reality it could potentially ruin these systems which in some areas is of tremendous importance unless something changes.

Bibliography

[1] J. Watrous, "Quantum Computational Complexity," *arXiv:0804.3401 [quant-ph]*, Apr. 2008, arXiv: 0804.3401. [Online]. Available: http://arxiv.org/abs/0804.3401

[2] T. Tusarova, "Quantum Complexity Classes," *arXiv:cs/0409051*, Sep. 2004, arXiv: cs/0409051. [Online]. Available: http://arxiv.org/abs/cs/0409051

[3] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*, 10th ed. USA: Cambridge University Press, 2011.

[4] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," 1994. [Online]. Available: https://ieeexplore.ieee.org/document/365700

[5] P. Vikstål, M. Grönkvist, M. Svensson, M. Andersson, G. Johansson, and G. Ferrini, "Applying the Quantum Approximate Optimization Algorithm to the Tail-Assignment Problem," *Physical Review Applied*, vol. 14, no. 3, p. 034009, Sep. 2020, publisher: American Physical Society. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevApplied.14.034009

[6] A. Bengtsson, P. Vikstål, C. Warren, M. Svensson, X. Gu, A. F. Kockum, P. Krantz, C. Križan, D. Shiri, I.-M. Svensson, G. Tancredi, G. Johansson, P. Delsing, G. Ferrini, and J. Bylander, "Improved Success Probability with Greater Circuit Depth for the Quantum Approximate Optimization Algorithm," *Physical Review Applied*, vol. 14, no. 3, p. 034010, Sep. 2020, publisher: American Physical Society. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevApplied.14.034010

[7] D. Coppersmith, "An approximate Fourier transform useful in quantum factoring," *arXiv:quant-ph/0201067*, Jan. 2002, arXiv: quant-ph/0201067. [Online]. Available: http://arxiv.org/abs/quant-ph/0201067

[8] L. K. Grover, "A fast quantum mechanical algorithm for database search | Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing." [Online]. Available: https://dl.acm.org/doi/10.1145/237814.237866

[9] A. Cho, "The biggest flipping challenge in quantum computing," Jul. 2020. [Online]. Available: https://www.sciencemag.org/news/2020/07/biggest-flipping-challenge-quantum-computing

[10] W. K. Wootters and W. H. Zurek, "A single quantum cannot be cloned," *Nature*, vol. 299, no. 5886, pp. 802–803, Oct. 1982, number: 5886 Publisher: Nature Publishing Group. [Online]. Available: https://www.nature.com/articles/299802a0

[11] D. Dieks, "Communication by EPR devices," *Physics Letters A*, vol. 92, no. 6, pp. 271–272, Nov. 1982. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0375960182900846

[12] S. B. Bravyi and A. Y. Kitaev, "Quantum codes on a lattice with boundary," *arXiv:quant-ph/9811052*, Nov. 1998, arXiv: quant-ph/9811052. [Online]. Available: http://arxiv.org/abs/quant-ph/9811052

[13] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, "Topological quantum memory: Journal of Mathematical Physics: Vol 43, No 9." [Online]. Available: https://aip.scitation.org/doi/10.1063/1.1499754

[14] P. Andreasson, J. Johansson, S. Liljestrand, and M. Granath, "Quantum error correction for the toric code using deep reinforcement learning," *Quantum*, vol. 3, p. 183, Sep. 2019, publisher: Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften. [Online]. Available: https://quantum-journal.org/papers/q-2019-09-02-183/

[15] D. Fitzek, M. Eliasson, A. F. Kockum, and M. Granath, "Deep Q-learning decoder for depolarizing noise on the toric code," *Physical Review Research*, vol. 2, no. 2, p. 023230, May 2020, publisher: American Physical Society. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevResearch.2.023230

[16] A. Olsson and G. Lindeby, "Distributed Training for Deep Reinforce-ment Learning Decoders on the Toric Code," *Chalmers University of Technology*, Jun. 2020. [Online]. Available: https://hdl.handle.net/20.500.12380/300977

[17] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra, "Planning and acting in partially observable stochastic domains," *Artificial Intelligence*, vol. 101, no. 1, pp. 99–134, May 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S000437029800023X

[18] K. J. Åström, "Optimal control of Markov processes with incomplete state information," *Journal of Mathematical Analysis and Applications*, vol. 10, no. 1, pp. 174–205, Feb. 1965. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022247X6590154X

[19] A. Y. Kitaev, "Fault-tolerant quantum computation by anyons," *Annals of Physics*, vol. 303, no. 1, pp. 2–30, Jan. 2003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0003491602000180

[20] D. K. Tuckett, A. S. Darmawan, C. T. Chubb, S. Bravyi, S. D. Bartlett, and S. T. Flammia, "Tailoring Surface Codes for Highly Biased Noise," *Physical Review X*, vol. 9, no. 4, p. 041031, Nov. 2019, publisher: American Physical Society. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevX.9.041031

[21] A. G. Fowler, "Optimal complexity correction of correlated errors in the surface code," *arXiv:1310.0863 [quant-ph]*, Oct. 2013, arXiv: 1310.0863. [Online]. Available: http://arxiv.org/abs/1310.0863

[22] A. M. Stephens, "Fault-tolerant thresholds for quantum error correction with the surface code," *Physical Review A*, vol. 89, no. 2, p. 022321, Feb. 2014, arXiv: 1311.5003. [Online]. Available: http://arxiv.org/abs/1311.5003

[23] D. F. Wise, J. J. L. Morton, and S. Dhomkar, "Using deep learning to understand and mitigate the qubit noise environment," *PRX Quantum*, vol. 2, no. 1, p. 010316, Jan. 2021, arXiv: 2005.01144. [Online]. Available: http://arxiv.org/abs/2005.01144

[24] R. Sweke, M. S. Kesselring, E. P. L. v. Nieuwenburg, and J. Eisert, "Reinforcement learning decoders for fault-tolerant quantum computation," *Machine Learning: Science and Technology*, vol. 2, no. 2, p. 025005, Jan. 2021, publisher: IOP Publishing. [Online]. Available: https://doi.org/10.1088/2632-2153/abc609

[25] P. Baireuther, T. E. O'Brien, B. Tarasinski, and C. W. J. Beenakker, "Machine-learning-assisted correction of correlated qubit errors in a topological code," *Quantum*, vol. 2, p. 48, Jan. 2018, publisher: Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften. [Online]. Available: https://quantum-journal.org/papers/q-2018-01-29-48/

[26] "Quantum logic gate," May 2021, page Version ID: 1026044985. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Quantum_logic_gate&oldid=1026044985

[27] "File:Sphere bloch.jpg - Wikimedia Commons." [Online]. Available: https://commons.wikimedia.org/wiki/File:Sphere_bloch.jpg

[28] S. Arora, "Supervised vs Unsupervised vs Reinforcement," Jan. 2020. [Online]. Available: https://www.aitude.com/supervised-vs-unsupervised-vs-reinforcement/

[29] "NVIDIA Blog: Supervised Vs. Unsupervised Learning," Aug. 2018. [Online]. Available: https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/

[30] M. Kenyon, "Supervised v. Unsupervised v. Reinforcement Learning: An Introduction," May 2020. [Online]. Available: https://thesharperdev.com/supervised-v-unsupervised-v-reinforcement-learning-an-introduction/

[31] B. Mehlig, *Artificial Neural Networks*, 2019, type: Electronic Article. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2019arXiv190105639M

[32] Zufzzi, "Neural network bottleneck achitecture," Oct. 2010. [Online]. Available: https://commons.wikimedia.org/wiki/File:Neural_network_bottleneck_achitecture.svg

[33] M. Plotke, "English: 2D Image-Kernel Convolution Animation," Jan. 2013. [Online]. Available: https://commons.wikimedia.org/wiki/File:2D_Convolution_Animation.gif

[34] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," *arXiv:1406.1078 [cs, stat]*, Sep. 2014, arXiv: 1406.1078. [Online]. Available: http://arxiv.org/abs/1406.1078

[35] E. Parisotto, H. F. Song, J. W. Rae, R. Pascanu, C. Gulcehre, S. M. Jayakumar, M. Jaderberg, R. L. Kaufman, A. Clark, S. Noury, M. M. Botvinick, N. Heess, and R. Hadsell, "Stabilizing Transformers for Reinforcement Learning," *arXiv:1910.06764 [cs, stat]*, Oct. 2019, arXiv: 1910.06764. [Online]. Available: http://arxiv.org/abs/1910.06764

[36] M. Morales, *Grokking Deep Reinforcement Learning*, 1st ed. Shelter Island, New York: Manning Publications, Nov. 2020.

[37] L. Baird and A. W. Moore, "Gradient descent for general reinforcement learning," *Advances in neural information processing systems*, pp. 968–974, 1999.

[38] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html

[39] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv:1707.06347 [cs]*, Aug. 2017, arXiv: 1707.06347. [Online]. Available: http://arxiv.org/abs/1707.06347

[40] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight Experience Replay," *arXiv:1707.01495 [cs]*, Feb. 2018, arXiv: 1707.01495. [Online]. Available: http://arxiv.org/abs/1707.01495

[41] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," *arXiv:1511.05952 [cs]*, Feb. 2016, arXiv: 1511.05952. [Online]. Available: http://arxiv.org/abs/1511.05952

[42] T. Fösel, M. Y. Niu, F. Marquardt, and L. Li, "Quantum circuit optimization with deep reinforcement learning," *arXiv:2103.07585 [quant-ph]*, Mar. 2021, arXiv: 2103.07585. [Online]. Available: http://arxiv.org/abs/2103.07585

[43] T. Grel, "Region of interest pooling explained," Feb. 2017, section: Data science. [Online]. Available: https://deepsense.ai/region-of-interest-pooling-explained/

[44] P. Pradhan, S. Rakshit, and S. Datta, "Lattice based cryptography : Its applications, areas of interest & future scope," 03 2019, pp. 988–993.

[45] S. Galbraith and F. Vercauteren, "Computational problems in supersingular elliptic curve isogenies," *Quantum Information Processing*, vol. 17, pp. 1–22, 2017.

A

Neural Network Architectures

For the most part, we utilized the neural network architectures summarized in Tab. A.1:

Module \Architecture	2D Conv	2D Conv + GRU	3D Conv
Conv Layers, Channels	[16, 32, 64, 16]	[16, 32, 64, 16]	[32, 64, 32, 16, 8]
# GRU Layers	-	2	-
GRU Layers, # Neurons	-	256	-
Fully-Connected Layers, # Neurons	[512, 128]	[512, 128]	[512, 256]

Table A.1: Overview of used neural network architectures

In the convolutional layers, we always chose stride = 1, kernel size = 3 (in all convolutional dimensions), and padding such that the output feature maps retain the shape of the input tensors.

Department of Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se