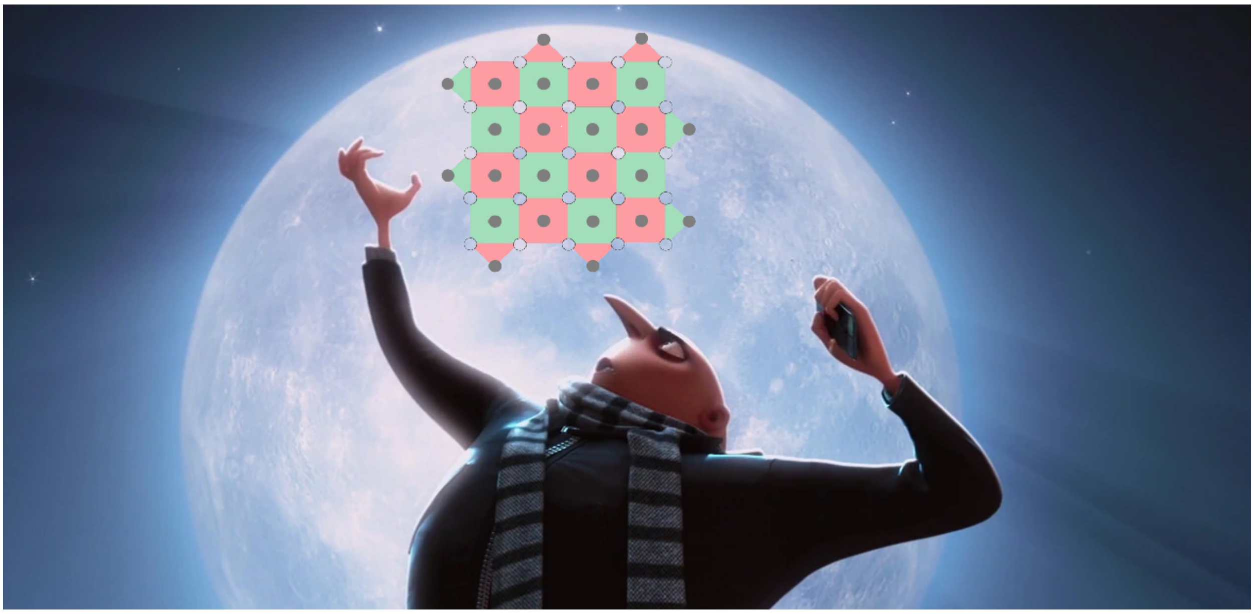
CHALMERS
UNIVERSITY OF TECHNOLOGY



Sequential Graph-Based Decoding of the Surface Code using a Hybrid Graph and Recurrent Neural Network Model

A work in progress

Master's thesis in Complex Adaptive Systems

OLE FJELDSÅ

GUSTAF JONASSON JOHANSSON

DEPARTMENT OF PHYSICS

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2025
www.chalmers.se

Master's thesis 2025

Sequential Graph-Based Decoding of the Surface Code using a Hybrid Graph and Recurrent Neural Network Model

OLE FJELDSÅ, GUSTAF JONASSON JOHANSSON

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2025

Sequential Graph-Based Decoding of the Surface Code using a Hybrid Graph and Recurrent Neural Network Model

OLE FJELDSÅ
GUSTAF JONASSON JOHANSSON

© OLE FJELDSÅ, GUSTAF JONASSON JOHANSSON, 2025.

Supervisor: Mats Granath, Department of Physics, University of Gothenburg.
Examiner: Mats Granath, Department of Physics, University of Gothenburg.

Master’s Thesis 2025
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

We need to make a serious cover

Cover: Gru evaluating the surface code .

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2025

Sequential Graph-Based Decoding of the Surface Code using a Hybrid Graph and Recurrent Neural Network Model

OLE FJELDSÅ
GUSTAF JONASSON JOHANSSON
Department of Physics
Chalmers University of Technology

Abstract

In order to achieve reliable quantum computation with noisy qubits, quantum error correction (QEC) is necessary. Quantum error-correcting codes mitigate the inherent noise in quantum systems by distributing the logical state over several qubits, thereby introducing redundancy. One such promising code is the surface code. It encodes the logical qubit using a two-dimensional lattice of physical data qubits and ancilla qubits. By taking and decoding measurements on the ancilla qubits of the surface code, one can deduce whether a logical bit- or phase-flip has occurred. However, this is a complex and potentially time-consuming task. Multiple decoding algorithms exist, such as the classical minimum-weight perfect matching (MWPM) decoder. In recent years, data-driven algorithms have been shown to decode the surface code with a high degree of accuracy. In this thesis, we present a machine learning approach to decoding the surface code using a combination of graph neural networks (GNN) and recurrent neural networks (RNN). Specifically, graph representations are constructed over a short, sliding time window of syndrome measurement data. Each representation is processed by a GNN and its output is used as a learned high-dimensional embedding for an RNN. This enables continuous decoding of measurement patterns over longer time series. While the decoder is trained on relatively short syndromes, it is able to generalize for new data and longer syndromes, outperforming the classical MWPM algorithm across both short and long time series. This work opens up a new approach to reliable and potentially fast decoding of QEC codes.

Keywords: Quantum error correction, graph neural networks, recurrent neural networks, gated recurrent unit, surface code.

Acknowledgements

We are grateful to Mats Granath and Moritz Lange for their invaluable guidance, insightful discussions, and patience throughout this project. We also thank Blaž Pridgar for his technical help and contributions to our model analysis. Finally, we appreciate Tobias Wallström and Eduardo Manoni for their spirited banter.

Are we happy with this?

Computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC) at Chalmers Centre for Computational Science and Engineering (C3SE), partially funded by the Swedish Research Council through grant agreements no. 2023-05353 and no. 2024014.

Update to the correct grant and agreement numbers.

Ole Fjeldså and Gustaf Jonasson Johansson, Gothenburg, June 2025

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

CPU	Central Processing Unit
GNN	Graph Neural Network
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
LSTM	Long Short-Term Memory
MWPM	Minimum Weight Perfect Matching
RNN	Recurrent Neural Network

Contents

List of Acronyms **ix**

List of Figures **xiii**

1 Introduction **1**

2 Theory **3**
2.1 Graph theory 3
2.2 Quantum computing 3
2.2.1 Qubit 3
2.2.2 Pauli operators 4
2.2.3 Quantum circuits 5
2.2.4 Quantum gates 5
2.2.5 Entanglement 6
2.3 Quantum error codes 6
2.3.1 Bit flip repetition code 7
2.3.2 The stabilizer formalism 9
2.3.3 The surface code 9
2.3.4 Minimum-weight perfect matching 10
2.4 Neural networks 11
2.4.1 Feed forward, fully connected neural networks 12
2.4.2 Recurrent neural networks 13
2.4.3 Gated recurrent unit 14
2.4.4 Graph convolution layer 15
2.4.5 Global mean pool 15

3 Methods **16**
3.1 Data generation 16
3.2 Neural network architecture 18
3.3 Training setup 21

4 Results **23**
4.1 Decoder training 23
4.2 Decoder performance 24

5 Conclusion **28**

References **31**

A Appendix **I**
- A.1 Appendix 1 I
- A.2 Appendix 2 I

List of Figures

2.1 The Bloch sphere. Any state corresponds to a point on the sphere, the positive and negative z direction represent the basis states $|0\rangle$ and $|1\rangle$. The position of the state is defined by (2.1). 4
2.2 Entanglement of two qubits. The state vector at each time represents the probabilities of the different possible states as $(p_{00}, p_{01}, p_{10}, p_{11})^T$. 6
2.3 The two qubit encoder. 7
2.4 Surface code four-cycle. The left hand side diagram shows a simplified representation of the quantum circuit diagram on the right. Pink represents X stabilizers while blue represents Z stabilizers. The data qubits are represented by circles and the ancilla qubits squares. . . . 9
2.5 Left: a tiling of the surface code four-cycle 2.4 creating the planar $[[13, 1, 3]]$ surface code. Right: the $[[9, 1, 3]]$ rotated surface code made by rotating the four-cycle tiling. 10
2.6 BEHOLD: the surface code. 10
2.7 Creating the MWPM of a graph. Here the nodes are 2D Cartesian coordinates and the edge weights are the distances between them. . . 11
2.8 Left: surface code with a syndrome from the logical coset $\mathcal{Z}$. Middle: the same syndrome with the fully connected graph between all X stabilizers and all X virtual boundary checks. Right: a MWPM of the syndrome. For all figures pink plaquettes represent activated X stabilizers while blue plaquettes represent activated Z stabilizers. The light blue and pink plaquettes outside the surface code represent the additional virtual boundary checks, note that only one virtual boundary check is ever used and only if there is an odd number of activated stabilizers. 11
2.9 Deep neural network with input $\mathbf{X} = \{x_0, x_1, \dots, x_{nx}\}$, a sequence of hidden layers, and an output $\mathbf{Y}' = \{y'_0, y'_1, \dots, y'_{ny}\}$. Each layer $\mathbf{X}_l$ is updated according to (2.18) with the input of each layer being the output of the previous. 12
2.10 Fundamental structure for a recurrent neural network. The hidden layer(s) feed their state to the output and to the output for the current time step. The figure shows both the unrolled and rolled up schematic at time t. 13
2.11 Circuit diagram of a single GRU block. r_t, z_t, n_t and h_t are calculated according to (2.20). 14

2.12 The update of a single node feature x'_1, according to (2.21) during graph convolution. 15

3.1 Diagram showing how graphs are created according to a sliding window schema. The circles represent all nodes that are present in the corresponding time step t. 17
3.2 Time required to generate syndrome graphs based on surface codes of distances $d = 3, 5, 7$, with $d_T = 2$. The left panel shows how the time to generate syndrome graphs changes with syndrome length. The right panel instead shows how the time to generate syndrome graphs changes with error rate p, and fixed syndrome length $T = 99$. The error bars represent one standard deviation. The results are based on $5 \cdot 10^3$ randomly generated syndromes per data point. 18
3.3 Overview of the entire model pipeline. **A**: Transfer surface codes to graph over all detector events. **B**: Embedding from graph to embedding tensor using graph convolutions. **C**: four layer Gated Recurrent Unit. 20
3.4 Proportion of class 0 syndromes as a function of syndrome length for code distances $d = \{3, 5, 7\}$, with error rate $p = 0.003$. The error bars represent one standard deviation. The results are based on roughly $2.5 \cdot 10^5$ randomly generated syndromes per data point. 22

4.1 Logical accuracy (left) and model output (right) as a function of training epoch. Each epoch consists of roughly $5 \cdot 10^5$ randomly generated syndromes. 24
4.2 Logical accuracy (left) and logical failure rate (right) as a function of surface code cycles. The vertical bars indicate standard deviation. All models were tested with $5 \cdot 10^5$ syndromes per data point. 25
4.3 Logical accuracy (left) and logical failure rate (right) as a function of surface code cycles up to 10^4 cycles. The vertical bars indicate standard deviation. All models are tested with $6 \cdot 10^4$ syndromes per data point. 25
4.4 Logical accuracy (left) and logical failure rate (right) as a function of error rate. The vertical bars indicate standard deviation. All models are tested with syndrome length $T = 99$ with $5 \cdot 10^5$ syndromes per data point. 26
4.5 Decoding time plotted as a function of code cycles. Each data point represents the mean decoding time of $2.5 \cdot 10^4$ randomly generated syndromes. In addition, the decoding time of our model has been normalized with the number of graphs present in the syndromes, because of the recurrent nature of our decoder. 27

1

Introduction

Quantum computing has the potential to transform how we solve complex problems in fields such as drug design [1], chemistry [2], and financial market analysis [3]. Perhaps its most well-known application is prime factorization [4], [5]. Unlike classical computers, which process information using bits that exist in one of two distinct states (0 or 1), quantum computers use qubits that can exist in superpositions of both states simultaneously. This unique property, combined with entanglement and other quantum phenomena, enables quantum algorithms to outperform classical methods in certain tasks. However, qubits are highly fragile, with their quantum state easily disrupted by environmental noise or imperfect control signals. Even the slightest interference can cause decoherence, collapsing the superposition and erasing valuable information. To counteract this, quantum error correction (QEC) is essential for preserving coherence and enabling reliable quantum computation.

To allow for error correction in a classical computer we can use a simple repetition code. For example, a simple three bit encoding $\{0, 1\} \to \{000, 111\}$ can correct a single bit-flip error and detect a two bit-flip error. Unfortunately, such an error correction scheme is not possible for qubits due to the no cloning theorem [6], the presence of both bit-flips and phase-flips, and the wave-function collapse resulting from reading the qubit.

To perform error correction on qubits, stabilizer codes are used. A stabilizer code is a type of quantum error-correcting code that protects quantum information by encoding so-called logical qubits as a larger set of physical qubits by entanglement. It is defined by a set of stabilizer operators, which are elements of the Pauli group that commute with each other and stabilize the encoded logical states. Errors are detected by measuring these stabilizers without collapsing the quantum state of the logical qubit. The surface code [7], [8] is one such stabilizer code. It is a topological error-correcting code that protects quantum information by encoding a logical qubit as a two-dimensional lattice of physical qubits.

Efficiently decoding the surface code is a challenging problem. One widely used approach is the Minimum Weight Perfect Matching (MWPM) algorithm [8], [9], a classical method that performs well in general but struggles with biased X/Z error rates and imperfect measurements. Maximum likelihood decoders [8], on the other hand, achieve excellent accuracy but are computationally infeasible due to their exponential time complexity. A promising alternative is machine learning-based decoders, a number of which have been proposed [10]–[21]. Machine learning models have the potential to decode rapidly once trained and are adaptable to different noise models, offering both flexibility and efficiency.

This work builds on previous work from from the Department of Physics at University of Gothenburg [22] using graph neural networks (GNN) to classify the most likely error class of a graph of stabilizer measurements. We build on that by breaking down graphs of longer syndromes into overlapping sub-graphs. These graphs are then processed by a combination of a GNN and a recurrent neural network (RNN) to generalize the decoding over variable time frames, in a similar fashion to [23].

We show that the combined GNN-RNN approach outperforms MWPM over long time series for simulated stabilizers under surface level noise. The decoder is able to generalize over several thousand measurement cycles while only being trained on syndromes of 49-99 stabilizer cycles.

2

Theory

This chapter provides the necessary theoretical background for this thesis. It covers the basics of graph theory, quantum computing, quantum error correction and machine learning with neural networks.

2.1 Graph theory

This section provides a brief introduction to the graph theory concepts necessary for understanding this thesis.

A directed graph is defined as a pair $\mathcal{G} = (N, E)$, where N is a set of nodes and $E \subseteq \{(x, y) \mid x, y \in N^2,\ x \neq y\}$ is a set of directed edges between them. Graphs serve as versatile models for relational structures, where nodes represent entities and edges denote relationships. An undirected graph can be obtained from a directed graph by ensuring that each edge is bidirectional and equally weighted in both directions. The number of nodes in a graph $|N|$ is called the order of the graph and the number of edges in a graph $|E|$ is called the size of the graph. There are several different ways of deciding the weights of an edge, if for example the nodes are coordinates the weights between them could be the distance between the nodes.

2.2 Quantum computing

This section needs more references.

This section provides the necessary background in quantum computing to build the concepts of quantum error correction.

2.2.1 Qubit

The qubit is the quantum computer equivalent to the bit in a classical computer. It represents the basic unit of information in quantum computing. The qubit $|\psi\rangle$ is written as

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

where $|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ are orthogonal basis vectors, and α, β are complex values with $|\alpha|^2 + |\beta|^2 = 1$. Here α, β give us the probabilities of measuring a qubit as $|0\rangle$ or $|1\rangle$. For example, the probability of measuring a zero is equal to $|\alpha|^2$.

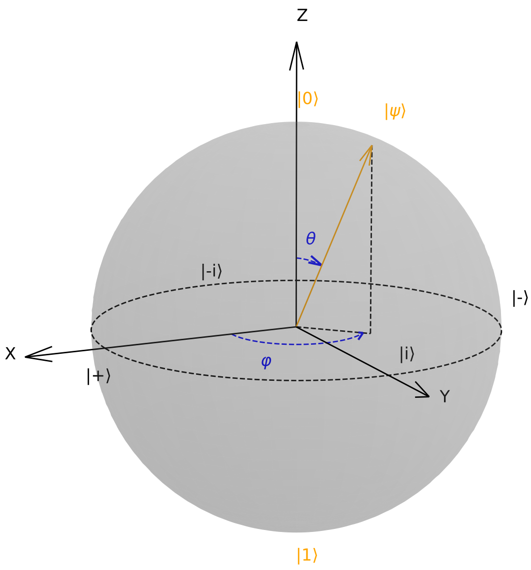


Figure 2.1: The Bloch sphere. Any state corresponds to a point on the sphere, the positive and negative z direction represent the basis states $|0\rangle$ and $|1\rangle$. The position of the state is defined by (2.1).

Figure 2.1 shows a geometric interpretation of the qubit called the Bloch sphere with the qubit state represented by a point on the sphere as

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle \tag{2.1}$$

In addition to the basis states we have the notable states $|+\rangle$, $|-\rangle$, $|i\rangle$ and $|-i\rangle$ on the positive and negative x and y axis of the sphere. All of these states have an equal probability of being measured as $|0\rangle$ or $|1\rangle$ but have different phases.

2.2.2 Pauli operators

The Pauli operators are

$$\mathbb{1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \; X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \; Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \; Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{2.2}$$

Each of the operators X, Y, and Z have the effect of flipping the qubit π radians around the x, y, or z or axis on the Bloch Sphere. As we can see from figure 2.1, the X operator corresponds to a bit-flip error, the Z operator corresponds to a phase-flip error, and the Y operator can be decomposed to $Y = XZ$ up to one phase. Applying the X or Z operator to a qubit $|\psi\rangle$ yields

Be more precise here.

$$X|\psi\rangle = \alpha X|0\rangle + \beta X|1\rangle = \alpha|1\rangle + \beta|0\rangle \tag{2.3a}$$
$$Z|\psi\rangle = \alpha Z|0\rangle + \beta Z|1\rangle = \alpha|0\rangle - \beta|1\rangle \tag{2.3b}$$

The Pauli group on a single qubit is defined as the Pauli operators

$$\mathcal{G}_1 = \{\pm\mathbb{1}, \pm i\mathbb{1}, \pm X, \pm iX, \pm Y, \pm iY, \pm Z, \pm iZ\}, \tag{2.4}$$

with the $\pm 1, \pm i$ terms to ensure that $\mathcal{G}_1$ is closed under multiplication. Every member of $\mathcal{G}_1$ has eigenvalue ± 1 or $\pm i$, is unitary, hermitian and self-inverse. Additionally, any two members of the Pauli group satisfy the commutation and anti-commutation relations

$$[\sigma_j, \sigma_k] \;= \sigma_j\sigma_k - \sigma_k\sigma_j = 2i\varepsilon_{jkl}\sigma_l \tag{2.5a}$$

$$\{\sigma_j, \sigma_k\} = \sigma_j\sigma_k + \sigma_k\sigma_j = 2\delta_{jk}\mathbb{1} \tag{2.5b}$$

where ε_{jkl} is the Levi-Civita symbol and δ_{jk} is the Kronecker delta

$$\varepsilon_{jkl} = \begin{cases} 1 & \text{if } (j,k,l) \in \{(x,y,z),(y,z,x),(z,x,y)\} \quad \text{(even permutation)} \\ -1 & \text{if } (j,k,l) \in \{(x,z,y),(y,x,z),(z,y,x)\} \quad \text{(odd permutation)} \\ 0 & \text{otherwise} \end{cases}$$

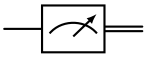
$$\delta_{jk} = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{otherwise} \end{cases}$$

The general Pauli group is made up of all operators formed from tensor products of elements from $\mathcal{G}_1$. If we for example have a three qubit system we can apply the operation

$$\mathbb{1} \otimes X \otimes Y = X_2Y_3,$$

where the left hand side is the tensor product of three operations (applied to their respective qubit), and the right hand side is the support notation of the same operation.

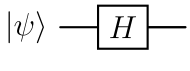
2.2.3 Quantum circuits

Qubits and operators are visualized using quantum circuit diagrams. Each qubit is represented by a horizontal line going from left to right trough time. The different operations applied to the qubits are placed along these lines. Single qubit operations like H, X and Z are represented as a single letter in a box placed on the line as in (2.6). Multiple qubit gates are represented using vertical connections between qubits as in (2.7) or rectangles covering multiple lines as in figure 2.3. Single lines represent qubits while double lines represent classical information, the measurement of a qubit is represented by the meter symbol  .

2.2.4 Quantum gates

Besides the Pauli operators, some additional quantum operators are needed, namely the quantum gates H, $CNOT$ and CZ. The Hadamard gate is defined as

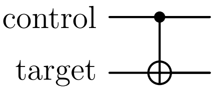
$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$



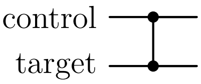
(2.6)

It has the effect $H|0\rangle = |+\rangle$, $H|1\rangle = |-\rangle$, and vice versa. In this way the Hadamard gate can be used to bring a qubit in and out of superposition. The $CNOT$ and CZ gates are defined as

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.7}$$



$$CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \tag{2.8}$$



These gates work as conditional X and Z gates. If the control is equal to $|1\rangle$, apply X or Z to the target qubit. If the control is in superposition the CNOT gate swaps the probabilities of measuring $|10\rangle$ and $|11\rangle$.

TODO

One important phenomenon enabled by controlled qubits is *phase kickback* ...

2.2.5 Entanglement

expand on why we need entanglement, and how it works. Mention the bell states, mention that we will use entanglement to enable QEC codes.

One of the distinguishing features of quantum systems is the ability to entangle Figure 2.2 shows how one can entangle two qubits. As we can see from the figure, measuring one of the qubits at the end of the circuit gives the state of the other qubit instantly.

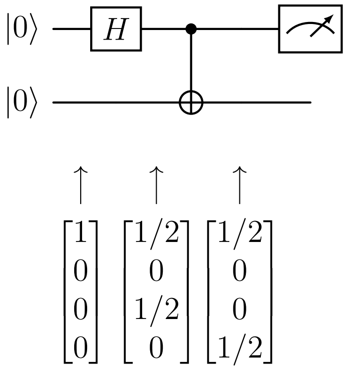


Figure 2.2: Entanglement of two qubits. The state vector at each time represents the probabilities of the different possible states as $(p_{00}, p_{01}, p_{10}, p_{11})^T$.

2.3 Quantum error codes

In order to facilitate reliable QEC, one needs to introduce redundancy in the way a qubit is represented. In practice, this means distributing the logical state over multiple qubits. If only one qubit is used, it is not possible to deduce if its value is the result of some computation, or an error. Using multiple qubits increases the reliability of the system by taking advantage of the fact that an error happening in multiple qubits is less likely than an error happening in just one qubit. This section covers the basics of QEC.

I feel like we should mention that this section leans heavily on [24].

2.3.1 Bit flip repetition code

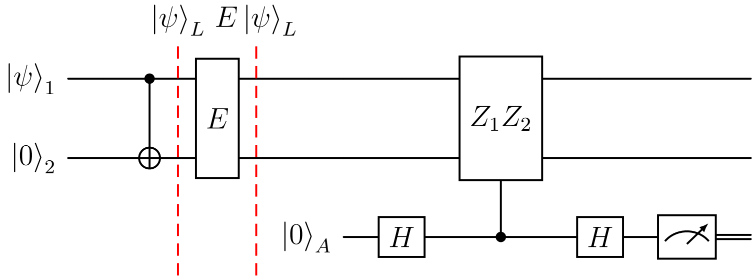


Figure 2.3: The two qubit encoder.

To allow for redundancy in a quantum code we can not simply copy bits. Instead we entangle the state of the qubits, expanding their Hilbert space. This spreads the state of a qubit $|\psi\rangle$ over the logical state $|\psi\rangle_L$. Here we show the two qubit encoder to exemplify this.

$$|\psi\rangle \xrightarrow{\text{two qubit encoder}} |\psi\rangle_L = \alpha\,|00\rangle + \beta\,|11\rangle = \alpha\,|0\rangle_L + \beta\,|1\rangle_L\,.$$

This has the effect of changing the Hilbert space of the qubit. Before encoding, the qubit is parametrized within $|\psi\rangle \in \mathcal{H}_2 = \text{span}\{|0\rangle\,,|1\rangle\}$. After the encoding this changes to the Hilbert space

$$|\psi\rangle_L \in \mathcal{H}_4 = \text{span}\{|00\rangle\,,|01\rangle\,,|10\rangle\,,|11\rangle\}$$

that we can split up into two mutually orthogonal subspaces

$$\mathcal{C} = \text{span}\{|00\rangle\,,|11\rangle\}, \qquad\qquad \mathcal{F} = \text{span}\{|01\rangle\,,|10\rangle\}.$$

In the two qubit system we have the following bit-flip errors:

Error	$\mathbb{1}_1\mathbb{1}_2$	$X_1\mathbb{1}_2$	$\mathbb{1}_1X_2$	X_1X_2
Syndrome	0	1	1	0

Table 2.1: Transposed syndrome table for the two qubit code.

If we now apply the bit flip X_1 to $|\psi\rangle_L$ (i.e. applying X to $|\psi\rangle_{L_1}$) we get

$$X_1\,|\psi\rangle_L = \alpha\,|10\rangle + \beta\,|01\rangle \in \mathcal{F} \subset \mathcal{H}_4$$

To distinguish $\mathcal{C}$ from $\mathcal{F}$ we use the controlled-Z_1Z_2 gate. The Z_1Z_2 operator applied to $|\psi\rangle_L$ yields the eigenvalue $(+1)$, i.e. $Z_1Z_2\,|\psi\rangle_L = (+1)\,|\psi\rangle_L$ in the following way

Should (2.9) and (2.10) be shortened down.

$$\begin{aligned}
Z_1Z_2\left|\psi\right\rangle_L &= Z_1Z_2(\alpha\left|00\right\rangle + \beta\left|11\right\rangle) \\
&= \alpha\left|00\right\rangle + \beta Z_1Z_2\left|11\right\rangle \\
&= \alpha\left|00\right\rangle + \beta Z\left|1\right\rangle \otimes Z\left|1\right\rangle \\
&= \alpha\left|00\right\rangle + \beta(-1)\left|1\right\rangle \otimes (-1)\left|1\right\rangle \\
&= \alpha\left|00\right\rangle + (-1)(-1)\beta\left|11\right\rangle \\
&= \alpha\left|00\right\rangle + \beta\left|11\right\rangle \\
&= \left|\psi\right\rangle_L
\end{aligned} \tag{2.9}$$

If we instead apply the operator to $\left|\psi\right\rangle_L$ with the X_1 error we end up with the eigenvalue (-1)

$$\begin{aligned}
Z_1Z_2X_1\left|\psi\right\rangle_L &= Z_1Z_2(\alpha X_1\left|00\right\rangle + \beta X_1\left|11\right\rangle) \\
&= Z_1Z_2(\alpha\left|10\right\rangle + \beta\left|01\right\rangle) \\
&= \alpha Z_1\left|10\right\rangle + \beta Z_2\left|01\right\rangle \\
&= \alpha Z_1\left|1\right\rangle \otimes \left|0\right\rangle + \beta\left|0\right\rangle \otimes Z_2\left|1\right\rangle \\
&= \alpha(-1)\left|1\right\rangle \otimes \left|0\right\rangle + \beta\left|0\right\rangle \otimes (-1)\left|1\right\rangle \\
&= (-1)(\alpha\left|1\right\rangle \otimes \left|0\right\rangle + \beta\left|0\right\rangle \otimes \left|1\right\rangle) \\
&= (-1)(\alpha\left|10\right\rangle + \beta\left|01\right\rangle \\
&= -\left|\psi\right\rangle_L
\end{aligned} \tag{2.10}$$

Note that in the last case the change is applied to the entangled ancilla qubit since $Z_1Z_2X_1\left|\psi\right\rangle_L = (+1)X_1\left|\psi\right\rangle$. The operator Z_1Z_2 is said to *stabilize* $\left|\psi\right\rangle_L$ as it leaves it unchanged.

The ancilla is set up as $\left|0\right\rangle_A$ to begin with before applying the Hadamard gate to it, changing it to the state $\left|+\right\rangle = \frac{1}{\sqrt{2}}(\left|0\right\rangle + \left|1\right\rangle)$. At this point, the ancilla is prepared and ready to be used in the controlled-Z_1Z_2 operation.
Applying the controlled-Z_1Z_2 gate yields the state

$$\frac{1}{\sqrt{2}}\left(\left|0\right\rangle\left|\psi\right\rangle_L + \left|1\right\rangle Z_1Z_2|\psi\rangle_L\right). \tag{2.11}$$

Since $|\psi\rangle_L$ is an eigenstate of Z_1Z_2 with eigenvalue λ, we get

$$Z_1Z_2|\psi\rangle_L = \lambda|\psi\rangle_L \tag{2.12}$$

and the total state becomes:

$$\frac{1}{\sqrt{2}}\left(\left|0\right\rangle|\psi\rangle_L + \lambda\left|1\right\rangle|\psi\rangle_L\right). \tag{2.13}$$

We now apply H again to our ancilla qubit

$$\begin{aligned}
H\left(\tfrac{1}{\sqrt{2}}(\left|0\right\rangle + \lambda\left|1\right\rangle)\right) &= \tfrac{1}{\sqrt{2}}\left(H\left|0\right\rangle + \lambda H\left|1\right\rangle\right) \\
&= \tfrac{1}{\sqrt{2}}\left(\tfrac{1}{\sqrt{2}}(\left|0\right\rangle + \left|1\right\rangle) + \lambda\tfrac{1}{\sqrt{2}}(\left|0\right\rangle - \left|1\right\rangle)\right) \\
&= \tfrac{1}{2}((\left|0\right\rangle + \left|1\right\rangle) + \lambda(\left|0\right\rangle - \left|1\right\rangle)) \\
&= \tfrac{1}{2}\left((1+\lambda)\left|0\right\rangle + (1-\lambda)\left|1\right\rangle\right).
\end{aligned} \tag{2.14}$$

Since $\lambda = \pm 1$ if $\lambda = 1$, then $\left|0\right\rangle$ is measured and if $\lambda = -1$, then $\left|1\right\rangle$ is measured. Thus, measuring the ancilla qubit tells us whether an $X_1\mathbb{1}_2$ or $\mathbb{1}_1X_2$ error has occurred.

2.3.2 The stabilizer formalism

Finish this

The stabilizer formalism describes the procedure to produce a general $[[n,k,d]]$ stabilizer code. Here n is the total number of qubits, k is the number of logical qubits and d is the stabilizer code distance. The code distance of a quantum error correction code is the minimum number of physical qubits that can go undetected and cause a logical error. In the case of the two qubit encoder we can detect but not correct a single bit-flip meaning that if the qubit was only susceptible to bit-flip errors it would be of distance $d = 2$. As qubits are susceptible to phase-flip errors as well, we also need a stabilizer code to take the logical Z operator into account when determining the code distance.

The stabilizers $\mathcal{S}$ of an $[[n,k,d]]$ code is a subgroup of the n-qubit Pauli group $\mathcal{G}_n$. $\mathcal{G}_n$ is made up of all operators of length n formed from tensor products of elements from $\mathcal{G}_1$(2.4). As with $\mathcal{G}_1$ the members of $\mathcal{G}_n$ either commute or anti-commute. All stabilizers must stabilize all logical states of the of the code space $\mathcal{C}$.

2.3.3 The surface code

The surface code four-cycle is the building block of the surface code. Figure 2.4 shows the quantum circuit diagram of the four-cycle and a simplified representation of the quantum circuit.

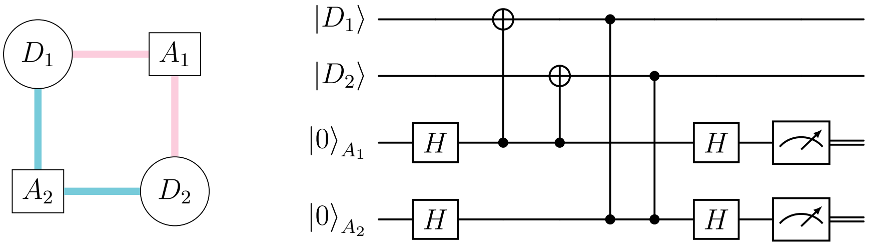


Figure 2.4: Surface code four-cycle. The left hand side diagram shows a simplified representation of the quantum circuit diagram on the right. Pink represents X stabilizers while blue represents Z stabilizers. The data qubits are represented by circles and the ancilla qubits squares.

While X and Z anti-commute ($XZ = -ZX$) the stabilizers $X_{D_1}X_{D_2}$ and $Z_{D_1}Z_{D_2}$ commute since they intersect non-trivially on an even number of qubits.

mention that this is a building block and has distance $d = 0$

$$\begin{aligned}
X_{D_1}X_{D_2}Z_{D1}Z_{D2} &= X_{D_1}Z_{D1}X_{D_2}Z_{D2} \\
&= -Z_{D1}X_{D_1}X_{D_2}Z_{D2} \\
&= -Z_{D1}X_{D_1}Z_{D2}X_{D_2} \\
&= (-1)^2 Z_{D1}Z_{D2}X_{D_1}X_{D_2} \\
&= Z_{D1}Z_{D2}X_{D_1}X_{D_2}
\end{aligned} \tag{2.15}$$

Using the four-cycle we can build the surface code, the left hand side of figure 2.5 show the $[[13,1,3]]$ planar surface code, as we can see from the diagram each stabilizer commutes as they still intersect non-trivially on an even number of qubits.

From here we can improve the cost of a logical qubit by rotating the planar surface code as seen on the left hand side of figure 2.5. This reduces the amount of physical qubits needed to encode the logical qubit while keeping the code distance $d = 3$. As we can see from the diagram the required commutation relations still hold.

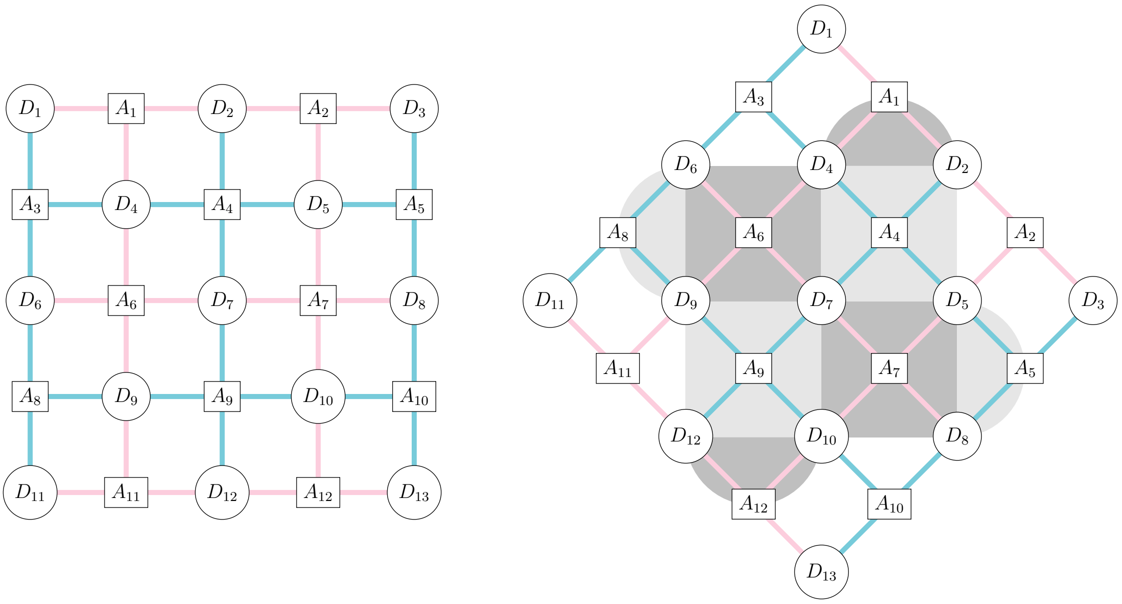

D_1 A_1 D_2 A_2 D_3
A_3 D_4 A_4 D_5 A_5
D_6 A_6 D_7 A_7 D_8
A_8 D_9 A_9 D_{10} A_{10}
D_{11} A_{11} D_{12} A_{12} D_{13}
D_1
A_3 A_1
D_6 D_4 D_2
A_8 A_6 A_4 A_2
D_{11} D_9 D_7 D_5 D_3
A_{11} A_9 A_7 A_5
D_{12} D_{10} D_8
A_{12} A_{10}
D_{13}

Figure 2.5: Left: a tiling of the surface code four-cycle 2.4 creating the planar [[13, 1, 3]] surface code. Right: the [[9, 1, 3]] rotated surface code made by rotating the four-cycle tiling.

Figure 2.6: BEHOLD: the surface code.

TODO The logical X/z direction . . .

TODO The logical coset . . .

2.3.4 Minimum-weight perfect matching

A matching of a graph $\mathcal{G}$ is a set of edges such that no edge shares a node with another edge. A perfect matching includes every node in $\mathcal{G}$. In order for the perfect matching to be minimum-weight, it must minimize the sum of the edge weights. Figure 2.7 shows an example of the minimum-weight perfect matching (MWPM) of a graph.

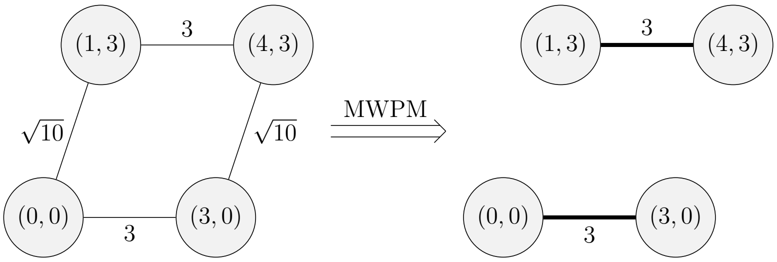


Figure 2.7: Creating the MWPM of a graph. Here the nodes are 2D Cartesian coordinates and the edge weights are the distances between them.

The MWPM of a graph can be found using the blossom algorithm [9] developed by Jack Edmonds in the 1960s, and it turns out this algorithm can be used to decode quantum stabilizer codes. In the QEC space it is known as the MWPM decoder [25]. Figure 2.8 shows an example of a MWPM on the surface code. Note the bright coloured "stabilizers" on the boundary of the surface code. If an odd number of X or Z stabilizers are activated, the virtual boundary stabilizers facilitate the creation of a MWPM, since an even number of nodes is required for this.

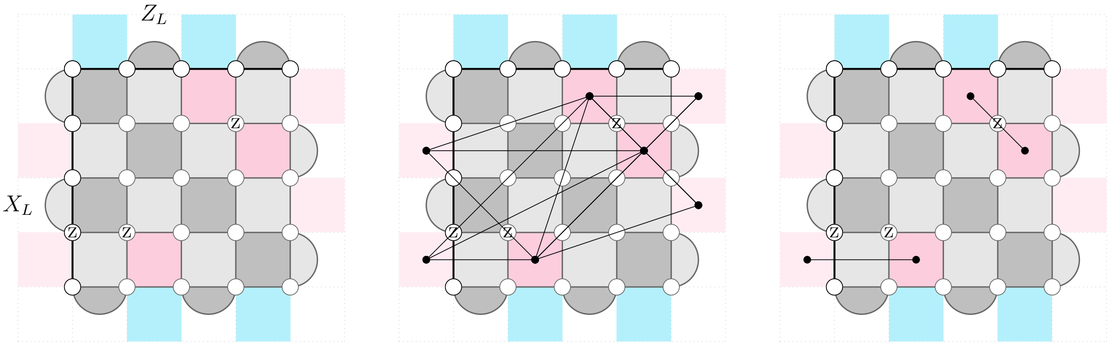


Figure 2.8: Left: surface code with a syndrome from the logical coset $\mathcal{Z}$. Middle: the same syndrome with the fully connected graph between all X stabilizers and all X virtual boundary checks. Right: a MWPM of the syndrome. For all figures pink plaquettes represent activated X stabilizers while blue plaquettes represent activated Z stabilizers. The light blue and pink plaquettes outside the surface code represent the additional virtual boundary checks, note that only one virtual boundary check is ever used and only if there is an odd number of activated stabilizers.

2.4 Neural networks

This project is based on developing a QEC decoder using a data-driven approach. In particular, our decoder is based on neural networks. As such, this section covers the basics of neural networks as it relates to this work.

2.4.1 Feed forward, fully connected neural networks

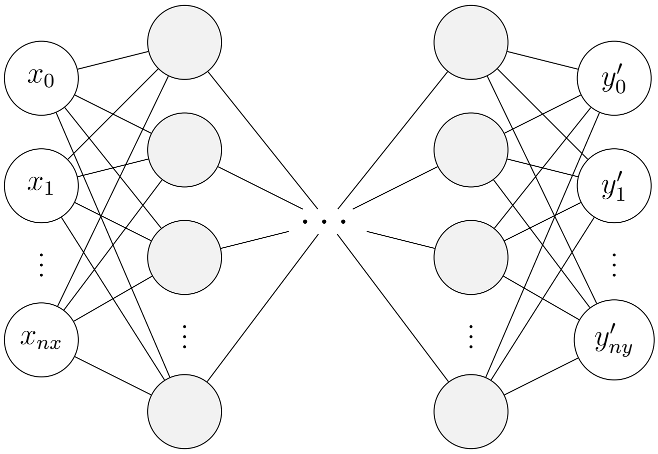


Figure 2.9: Deep neural network with input $\mathbf{X} = \{x_0, x_1, \ldots, x_{nx}\}$, a sequence of hidden layers, and an output $\mathbf{Y}' = \{y'_0, y'_1, \ldots, y'_{ny}\}$. Each layer $\mathbf{X}_l$ is updated according to (2.18) with the input of each layer being the output of the previous.

The building block of neural networks is the (artificial) neuron, first proposed as a mathematical model for the nervous system [26]. The neuron N can be written as

$$N(\mathbf{X}) = f\left(b + \sum_{x_j \in \mathbf{X}} x_j w_j\right) \tag{2.16}$$

where $\mathbf{X}$ is the input tensor to the neuron, w_j is the weight from input x_j to the neuron, b is the bias for the neuron, and f is a (typically non-linear) activation function. From this we can create layers of neurons with output $\mathbf{X}_l$

$$\mathbf{X}_l(\mathbf{X}) = \begin{bmatrix} N_0 \\ N_1 \\ \vdots \\ N_n \end{bmatrix}, \qquad N_i(\mathbf{X}) = f\left(b_i + \sum_{x_j \in \mathbf{X}} x_j w_{ji}\right) \tag{2.17}$$

This can be rewritten as the matrix operation

$$\mathbf{X}_l(\mathbf{X}) = f\left(\mathbf{W}_l \mathbf{X} + \mathbf{B}_l\right) \tag{2.18}$$

where $\mathbf{W}_l$ is a $n \times k$ matrix, $\mathbf{X}$ is a $k \times d$ matrix, $\mathbf{B}_l$ is a $n \times d$ matrix, n is the number of neurons in the layer and d is the number of samples in the input tensor. These layers can be stacked after each other creating a deep neural network as seen in figure 2.9.

The neural network is initialized with random weights and biases. To allow it to make precise predictions it is trained on known input output pairs $\{\mathbf{X}, \mathbf{Y}\}$ and every weight and bias is adjusted using backpropagation gradient descent [27]. In this process every parameter p (weights and biases) is updated according to $p' = -\eta\frac{\partial E(\mathbf{Y},\mathbf{Y}')}{\partial p}$. Here, $E(\mathbf{Y},\mathbf{Y}')$ is a chosen error function indicating the performance of the network and η is a learning rate ensuring appropriate size of the parameter update. When using the notation (2.18) we get the update rule

$$\begin{aligned}\nabla_l &= \mathbf{W}_l^T \nabla_{l+1} && \text{(2.19a)}\\ \nabla_W &= \nabla_{l+1}\mathbf{X}_{l-1} && \text{(2.19b)}\\ \mathbf{W}_l' &= -\eta\nabla_W && \text{(2.19c)}\\ \mathbf{B}_l' &= -\eta\nabla_l && \text{(2.19d)}\end{aligned}$$

where the gradient of the output layer $\nabla_{\mathbf{Y}'}$ is $\frac{\partial E(\mathbf{Y},\mathbf{Y}')}{\partial \mathbf{Y}'}$. This update is passed backwards trough the network one layer at a time.

2.4.2 Recurrent neural networks

To allow a neural network to handle time series data of variable length it is useful to give it some sort of ”memory”. The recurrent neural network (RNN) [28] allows for this by connecting the hidden state (sequence of hidden layers) h of the network at time t to the hidden state at time $t+1$ as seen in figure 2.10.

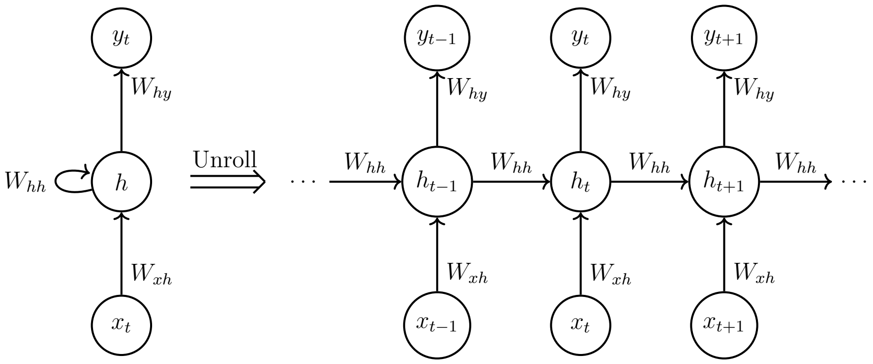


Figure 2.10: Fundamental structure for a recurrent neural network. The hidden layer(s) feed their state to the output and to the output for the current time step. The figure shows both the unrolled and rolled up schematic at time t.

The hidden state is is similar to the feed forward network and the parameters are updated using backpropagation.

2.4.3 Gated recurrent unit

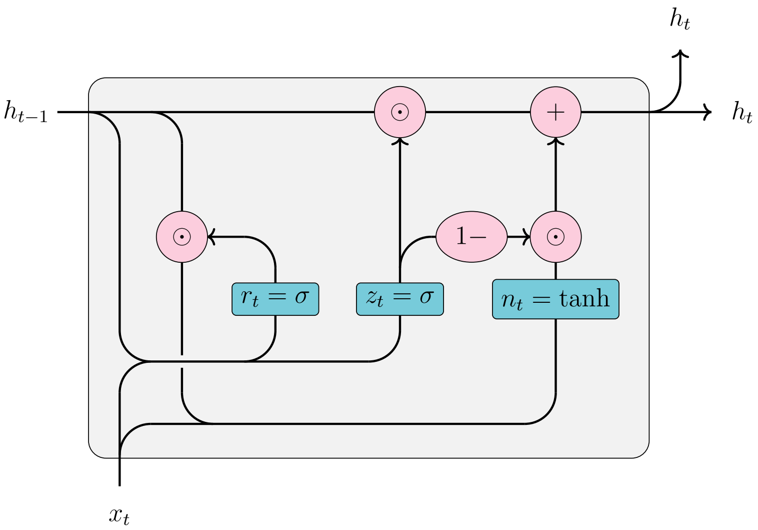


Figure 2.11: Circuit diagram of a single GRU block. r_t, z_t, n_t and h_t are calculated according to (2.20).

Using fully connected layers for the hidden state of the RNN can run into the vanishing gradient problem for longer time series [29], [30], which makes network training difficult. One approach to mitigating this is to use long short-term memory (LSTM) [31] blocks for the hidden state. LSTM circumvents the vanishing gradient by keeping separate long term and short term hidden states. The gated recurrent unit (GRU) [32] builds on the LSTM but removes the cell state vector, combining long and short term memory, giving it the parameters

$$
\begin{aligned}
r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) && \text{(2.20a)}\\
z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) && \text{(2.20b)}\\
n_t &= \tanh(W_{in}x_t + b_{in} + r_t \odot (W_{hn}h_{(t-1)} + b_{hn})) && \text{(2.20c)}\\
h_t &= (1 - z_t) \odot n_t + z_t \odot h_{(t-1)} && \text{(2.20d)}\\
\sigma(x) &= \frac{1}{1 + \exp(-x)}
\end{aligned}
$$

Here the reset gate r_t decides how much of the previous state to remember. If $r_t = 0$ the new state forgets the entirety of the old state. The update gate z_t decides how much of the past hidden state to keep. If $z_t = 1$ we take none of the new state and all of the old state and vice versa. The candidate state n_t gives a candidate for a new hidden state before the update gate decides how much of it to accept. Figure 2.11 gives a more illustrative view of the GRU block.

2.4.4 Graph convolution layer

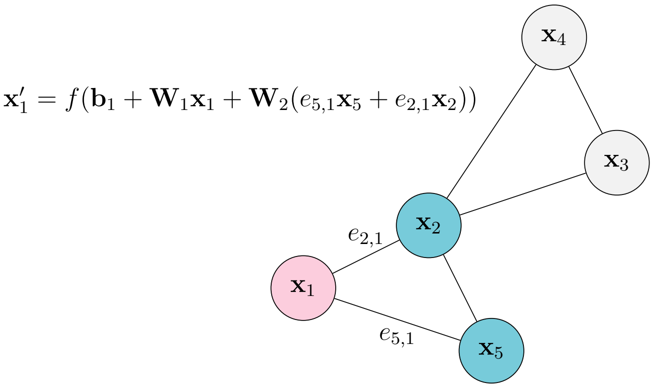

$\mathbf{x}_1' = f(\mathbf{b}_1 + \mathbf{W}_1\mathbf{x}_1 + \mathbf{W}_2(e_{5,1}\mathbf{x}_5 + e_{2,1}\mathbf{x}_2))$
$\mathbf{x}_4$
$\mathbf{x}_3$
$\mathbf{x}_2$
$e_{2,1}$
$\mathbf{x}_1$
$e_{5,1}$
$\mathbf{x}_5$

Figure 2.12: The update of a single node feature x_1', according to (2.21) during graph convolution.

The graph convolution layer maps every node feature $x_1, x_2, \ldots$ to a new state according to the graph neural network operator [33]

$$\mathbf{x}_i' = f\left(\mathbf{b}_i + \mathbf{W}_1\mathbf{x}_i + \mathbf{W}_2 \sum_{j\in\mathcal{N}_i} e_{ji}\mathbf{x}_j\right) \tag{2.21}$$

Like other neural networks the weights and biases ($\mathbf{W}_i$, $\mathbf{b}_i$) are trainable parameters initialized randomly and updated using backpropagation. e_{ji} are the edge weights of the graph and f is the actiavtion function. The graph neural network is permutation equivariant, meaning that it does not change the structure of the graph - only the node features. Figure 2.12 shows the update rule acting on a single node feature during a graph convolution.

2.4.5 Global mean pool

The global mean pool of a graph $\mathcal{G}$ is the mean of every node feature across the graph nodes $N_\mathcal{G}$ according to

$$\text{gmp}(\mathcal{G}) = \frac{1}{|N_\mathcal{G}|}\sum_{\mathbf{x}\in N_\mathcal{G}} \mathbf{x} \tag{2.22}$$

where $|N_\mathcal{G}|$ is the number of nodes in the graph. This reduces (pools) the state of the graph into a single vector representation of the same size as a single node feature vector. The fixed size of the pool makes it a suitable embedding between a graph and neural network structures demanding fixed size inputs.

3

Methods

This chapter covers the methods that were employed when creating our quantum error correction decoder. First, we explain how data for model training and inference was generated. Next, we describe the neural network architecture. Finally, we cover how the neural network was trained.

3.1 Data generation

In order to generate data for use during training and inference, the Python package Stim [34] was used. Stim can be used to simulate quantum stabilizer circuits, such as the rotated surface code in our case. When simulating stabilizer circuits, bit-flip and phase-flip errors are applied with error rate p using a noise model called *circuit-level noise*. This noise model consists of *depolarizing noise* that applies X, Y, and Z errors with equal probability $p_X = p_Y = p_Z = \frac{p}{3}$, in addition to bit-flip errors after ancilla qubit measurements and resets.

After simulating the circuit for T cycles, or time steps, Stim returns a vector of so-called detection events from $T + 1$ time steps, and a boolean value indicating if the simulation has resulted in a logical bit-flip or phase-flip. This value is used as the label when training and evaluating the neural network. Stim simulates a real experiment and as such, the data qubits at the end of the simulation can only be measured in either the X basis or the Z basis. If the data qubits are measured in the X basis, logical phase-flips can be detected, and if they are measured in the Z basis, logical bit-flips can be detected. It is this final measurement of the data qubits that causes the $+1$ in the $T+1$ expression. The so-called detection events are also boolean values, indicating whether or not each stabilizer measurement differs from the error-free case.

Having generated a syndrome, a series of graphs each spanning d_T time steps are created. The nodes correspond to detection events, each represented by a feature vector $[x, y, t_r, Z, X]$ where x, y, t_r are node coordinates and X, Z are boolean values indicating stabilizer type. The x and y coordinates indicate the location of the stabilizer on the surface code, $x, y \in [0, d]$, the top left being the origin. The t_r coordinate indicates the relative time step of the detection event within the graph, $t_r \in [0, d_T - 1]$.

Each node has an edge to its twenty nearest neighbours. These edges are created using a k-nearest neighbour algorithm. Generally, the graphs contain less than twenty nodes, and as such are undirected and fully connected. In principle, though, the graphs can contain more than twenty nodes, and in these cases the graphs are not fully connected, and potentially not undirected. The number of nodes in a graph has a positive correlation with the code distance d, the number of time steps the graph spans d_T, and the error rate p. Next, the edge weights are created. These are calculated as the square of the inverse supremum norm between two nodes. For instance, the edge weight between nodes i and j, e_{ij}, is:

$$e_{ij} = (max(|x_i - x_j|, |y_i - y_j|, |t_{r_i} - t_{r_j}|))^{-2} \tag{3.1}$$

The graphs are created according to a sliding window schema, such that all nodes (except for nodes in the first and last time step) are present in multiple overlapping graphs, see Figure 3.1. Should a graph be empty, such as if there are no detection events until $t = 5$ in Figure 3.1, leading to an empty Graph 1, the next graph, i.e. Graph 2, takes its place as the first graph of the syndrome. As such, syndromes contain a varying number of graphs for a given syndrome length T.

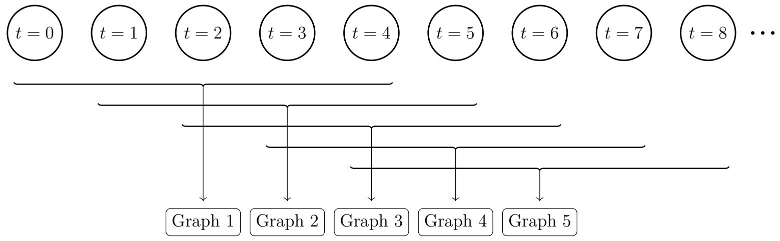


Figure 3.1: Diagram showing how graphs are created according to a sliding window schema. The circles represent all nodes that are present in the corresponding time step t.

Since a given node can belong to multiple graphs, something has to be done to avoid creating a single graph that spans the entire syndrome. The solution we settled on was making copies of each node. Each node is copied $min(t, d_T - 1)$ times, where t corresponds to the time step the node belongs to. Nodes belonging to time step $t = 0$ are only present in one graph, and therefore do not need to be copied. Nodes belonging to time step $t = 1$ are present in two graphs, and as such need to be copied once. Next, the time coordinate t_r of each node copy is adjusted accordingly. For instance, nodes in time step $t = 4$ belonging to Graph 1 in Figure 3.1 have time coordinate $t_r = 4$, and the "same" nodes in Graph 2 instead have time coordinate $t_r = 3$. Each node is also assigned an index indicating to which graph it belongs. This index is used when computing the edges.

maybe this phrasing is bad. A bit unclear

Unfortunately, creating the syndrome graphs by first copying the nodes takes a significant amount of time, as the number of nodes is increased by roughly a factor of d_T. During training of our neural network, around 30%-50% of the time is dedicated to data generation. Section 3.3 covers this in greater detail. Further, during inference, the vast majority of the time is dedicated to data generation, as the model is evaluated on syndromes of much longer length than those on which the model is trained, and this takes a long time. Figure 3.2 shows how the time required to generate syndrome graphs varies with syndrome length T and error rate p for code distances $d = 3, 5, 7$.

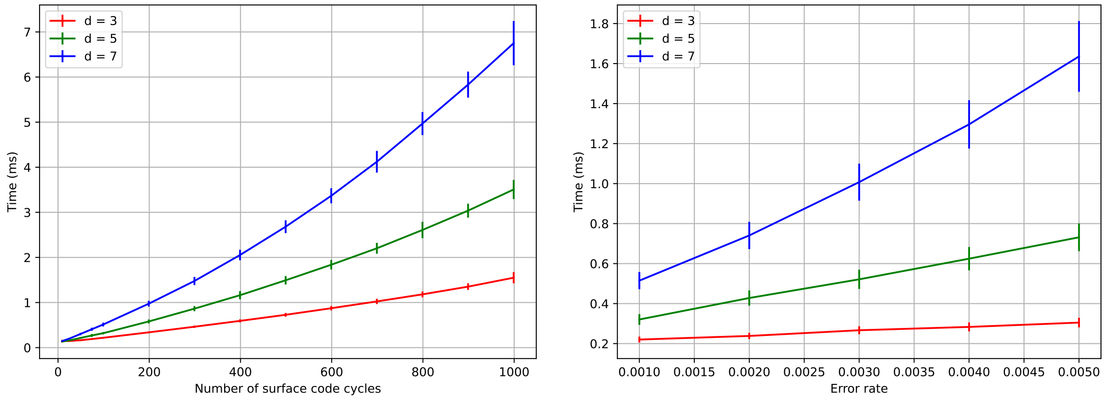


Figure 3.2: Time required to generate syndrome graphs based on surface codes of distances $d = 3, 5, 7$, with $d_T = 2$. The left panel shows how the time to generate syndrome graphs changes with syndrome length. The right panel instead shows how the time to generate syndrome graphs changes with error rate p, and fixed syndrome length $T = 99$. The error bars represent one standard deviation. The results are based on $5 \cdot 10^3$ randomly generated syndromes per data point.

3.2 Neural network architecture

The neural network is trained to detect either logical bit-flips or phase-flips. First, the syndrome graphs are fed through a GNN whose purpose is to create a high dimensional embedding vector of each node. The GNN consists of a series of graph convolutions that successively increase the size of the embedding vectors. The rectified linear unit (ReLU) activation function, see (3.2), is used following each graph convolution.

$$ReLU(x) = max(0, x) \tag{3.2}$$

Next, graph embeddings are obtained by averaging the node features of each graph, using an operation called global mean pool, see section 2.4.5. The graph embeddings are then fed sequentially through an RNN, which in our case is a multi-layer GRU. At each time step, the GRU outputs a number of hidden states, one for each layer. These, in combination with the next graph embedding, are used as input to the GRU in the following time step.

Finally, the hidden state corresponding to the last layer of the final time step is decoded by feeding it through a dense layer, which maps the hidden state vector to a scalar. A sigmoid function is then applied to limit the scalar to the interval (0, 1). In order to determine if a logical bit-flip or phase-flip has occurred, this number is rounded to the nearest integer. A zero indicates that no error has occurred, whereas a one indicates that an error has indeed occurred. It is important to note that a given network can only detect one type or error. If, in the last measurement round, the data qubits are measured in the X basis, the network learns to detect logical phase-flips, and if the data qubits are measured in the Z basis, the network learns to detect logical bit-flips. Therefore, one needs two separate networks to deduce if the syndrome belongs to the logical X, Y, Z, or I class. We exclusively trained networks to detect logical bit-flips, but it is trivial to train networks to detect logical phase-flips, as the only difference is which data qubits are measured in the last round. Figure 3.3 shows an overview of the model pipeline.

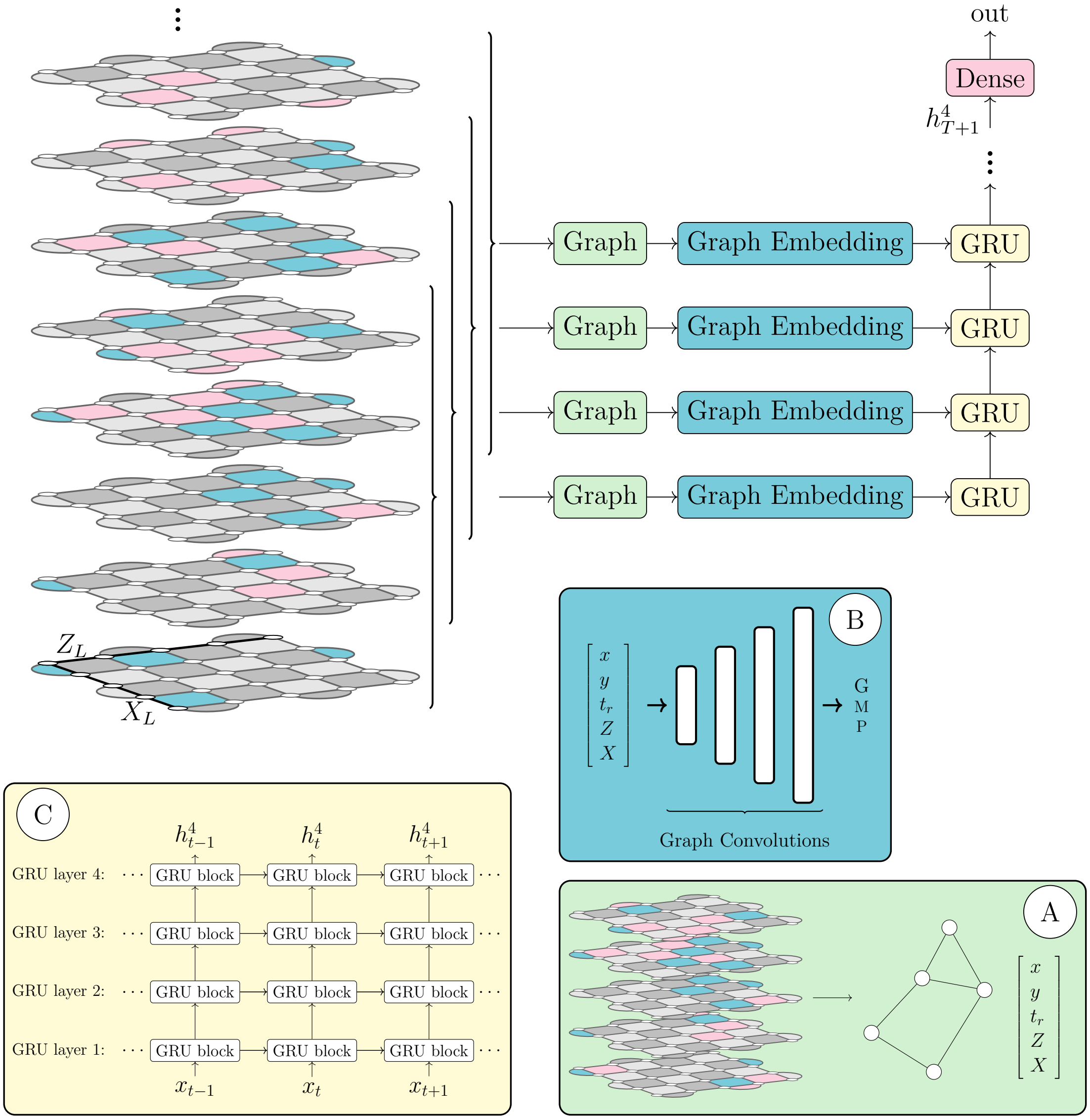

out
Dense
h^4_{T+1}
Graph
Graph Embedding
GRU
Z_L
X_L
B
x y t_r Z X
G M P
Graph Convolutions
C
h^4_{t-1} h^4_t h^4_{t+1}
GRU layer 4:
GRU layer 3:
GRU layer 2:
GRU layer 1:
GRU block
x_{t-1} x_t x_{t+1}
A

Figure 3.3: Overview of the entire model pipeline. **A**: Transfer surface codes to graph over all detector events. **B**: Embedding from graph to embedding tensor using graph convolutions. **C**: four layer Gated Recurrent Unit.

Improve this figure text, Add an actual syndrome to the surface code.

A benefit of using an RNN is that, during inference, the whole syndrome does not need to be fed through the neural network at once, only the part that corresponds to the last graph and the previous hidden state. In other words, decoding a syndrome at time step $t = 7999$ theoretically takes the same amount of time as decoding a syndrome at $t = 99$. This is not the case for other decoders, such as MWPM.

Our decoder was implemented using PyTorch [35] and PyTorch Geometric (PyG) [36]. PyTorch is a machine learning library that enables efficient tensor computing. PyG is built upon PyTorch and provides functionality to implement graph neural networks.

3.3 Training setup

Since our decoder is based on a neural network, a significant amount of data is needed to train it. As mentioned in section 3.1, data was obtained by generating it using Stim. Instead of generating fixed training and test sets, data was generated one batch at a time during training. As such, every batch that is fed through the network is unique. This approach has both benefits and drawbacks. First, it decreases the tendency for the network to overfit on the provided data. On the other hand, a significant amount of time during training is dedicated to data generation. Further, because Stim runs on the CPU, every new batch of graphs that is generated must be transferred to the GPU, which slows down training further. Generating the data and transferring it to the GPU takes roughly 30%-50% of the total training time.

Despite not using a fixed training set, the training was split into so-called epochs. Generally, one epoch corresponds to feeding the whole data set through the neural network once. We settled on 256 batches counting as one epoch, where each batch consists of 2048 syndromes. Therefore, one epoch worth of data corresponds to roughly $5 \cdot 10^5$ syndromes. The number of epochs required for the loss to converge is positively correlated with code distance. Chapter 4 covers this in greater detail. It is important to note that we did not make use of a test set as a way of evaluating the performance of the model and the convergence of the loss during training. The reason for this is that with no fixed training set, the model does not overfit on the provided data, as each batch is randomly sampled from a state space much larger than than the total number of syndromes generated during training.

double check

During training, the network learns to predict a label that is either 0 or 1. The probability of a logical bit-flip or phase-flip error occurring is positively correlated with syndrome length and error rate p. As such, the relative class proportion for short syndromes is skewed towards class 0, i.e. the dataset is unbalanced. This can cause issues when training machine learning models [37]. Figure 3.4 shows the class 0 proportion as a function of syndrome length for code distances 3, 5, and 7. We can see that for shorter syndrome lengths, the dataset is indeed imbalanced. However, the class 0 proportion seems to converge to 0.5 as the syndrome length increases. The models we trained were all based on syndromes with roughly equal class proportions.

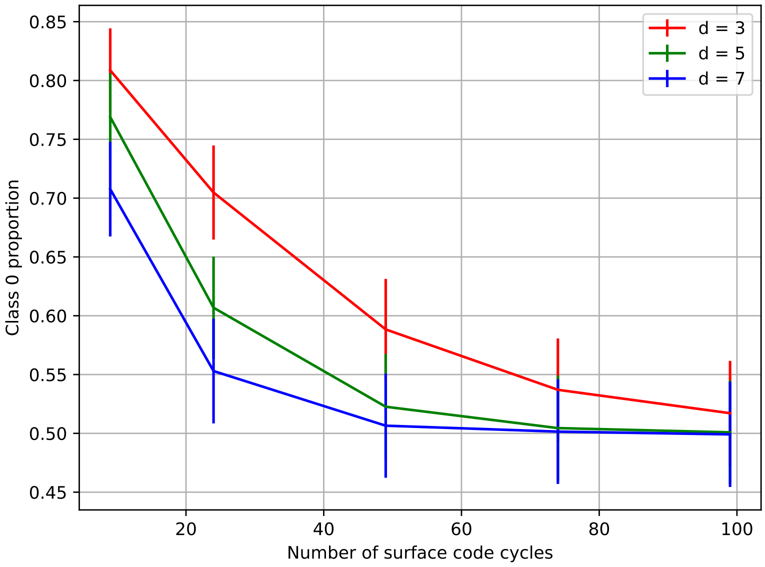


Figure 3.4: Proportion of class 0 syndromes as a function of syndrome length for code distances $d = \{3, 5, 7\}$, with error rate $p = 0.003$. The error bars represent one standard deviation. The results are based on roughly $2.5 \cdot 10^5$ randomly generated syndromes per data point.

It is important to note that syndromes containing no detection events are always excluded, both during training and inference. As nodes in the syndrome graphs are based on detection events, the absence of them implies that all the graphs belonging to a syndrome are empty. Syndromes without any detection events belong to class 0, and are referred to as *trivial* cases. These cases are also excluded from model performance metrics, such as accuracy.

Our decoder was trained using the Adam (Adaptive Moment Estimation) optimizer [38], with betas $\beta_1 = 0.9$, $\beta_2 = 0.999$. Adam is similar to ordinary gradient descent, but it leverages momentum and adaptive learning rates for each parameter. Momentum means that part of the previous update is added onto the next. This helps speed up convergence. The decoder was optimized by minimizing the binary cross-entropy (BCE) loss function:

$$BCE = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(O_i) + (1 - y_i) log(1 - O_i)] \tag{3.3}$$

where N is the number of observations, $y_i \in \{0, 1\}$ is the target label, and $O_i \in (0, 1)$ is the output from the network. The initial learning rate, η_0, was set to 10^{-3}. During training, the learning rate was decreased exponentially to 10^{-4} according to $\eta_t = \eta_0 0.95^t$, where t corresponds to the current epoch.

4

Results

With a method for obtaining data and a neural network architecture in place, models can be trained and evaluated. In this chapter we first cover the training of the models before presenting their performance.

4.1 Decoder training

Three models were trained, one for each distance $d \in \{3, 5, 7\}$. All three models were trained with a varying error rate $p \in \{0.001, 0.002, 0.003, 0.004, 0.005\}$ that was uniformly sampled on a batch-by-batch basis. Because the number of epochs required for the loss to converge is positively correlated with code distance, the distance 7 model was trained for 500 epochs, the distance 5 model for 300 epochs, and the distance 3 model for 150 epochs. Since each epoch consists of roughly $5 \cdot 10^5$ syndromes, the distance 7 model was trained on $2.5 \cdot 10^8$ syndromes, the distance 5 model on $1.5 \cdot 10^8$ syndromes, and the distance 3 model on $7.5 \cdot 10^7$ syndromes.

The three models were trained with slightly different syndrome graph settings. Both the distance 5 and 7 model were trained on syndromes of length $T = 49$, with sliding window size $d_T = 2$. The distance 3 model was instead trained on syndromes of length $T = 99$ with sliding window size $d_T = 5$. There are two reasons for this. First, the time required to train models increases with code distance, syndrome length, and sliding window size. The distance 7 model took roughly four days and eighteen hours to train on an Nvidia A40 GPU. Had it instead been trained on syndromes of length $T = 99$ with $d_T = 5$, it would have taken more than a week. Therefore, in the interest of time, we settled on shorter syndromes. Secondly, we experimented with multiple different syndrome lengths and sliding window sizes especially on distance 3 models, as they can be trained relatively quickly. We found that models that were trained on syndromes of length $T = 49$ had similar performance to those trained using $T = 99$. A few distance 3 models were trained on syndromes of length $T = 24$, too. While these models performed relatively well on shorter syndromes, they failed to generalize for longer syndromes.

The state space grows exponentially with code distance. As such, the amount of information that the neural network needs to encode also grows exponentially. Initially, the models had the same number of parameters regardless of code distance, roughly $5.3 \cdot 10^5$, with a graph embedding vector of length 256, and the hidden state of the GRU a vector of length 128. This configuration worked well for code distances 3 and 5, but it was not enough for code distance 7. Therefore, the sizes of the graph embedding vector and hidden state vector for the code distance 7 model were doubled to 512 and 256, respectively, increasing the number of parameters to roughly $2.1 \cdot 10^6$. See appendix A.2 for a more detailed breakdown of the number of parameters and architecture of each model.

The results from training are summarized in figure 4.1. The left panel shows the logical accuracy as a function of training epoch. As expected, the distance 3 model converges at a lower logical accuracy compared to the two other models. The right panel shows the mean model output for class 0 and class 1, also as a function of training epoch. The model output can be interpreted as the confidence of the models. Based on this interpretation, the distance 3 model is less confident with respect to which input belongs to which class.

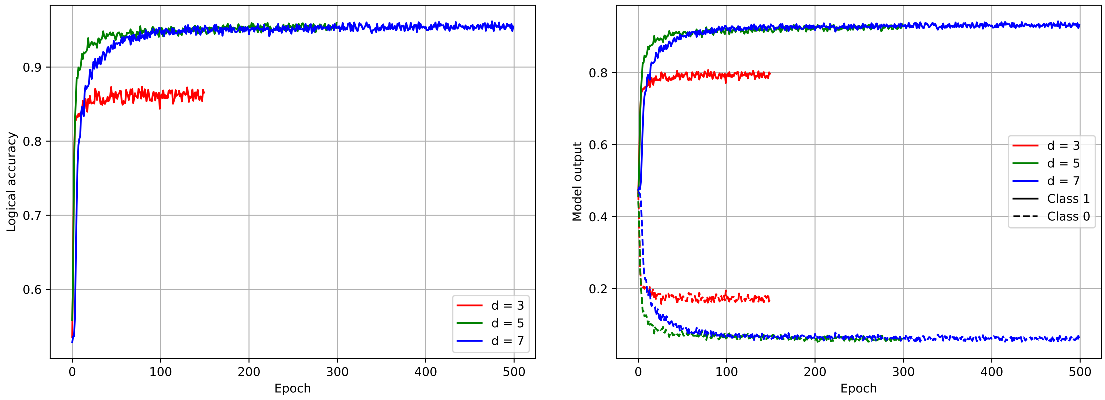


Figure 4.1: Logical accuracy (left) and model output (right) as a function of training epoch. Each epoch consists of roughly $5 \cdot 10^5$ randomly generated syndromes.

4.2 Decoder performance

Having trained the models, it is time to evaluate their performance. The MWPM decoder implemented by the PyMatching library [39] is used as a benchmark for all performance metrics, and is represented by dashed lines in figures 4.2 through 4.5. Figure 4.2 shows the logical accuracy and failure rate of the three models as a function of surface code cycle. The distance 3 and 5 models both outperform MWPM over the entire domain. However, the distance 7 model has similar or worse performance compared to MWPM.

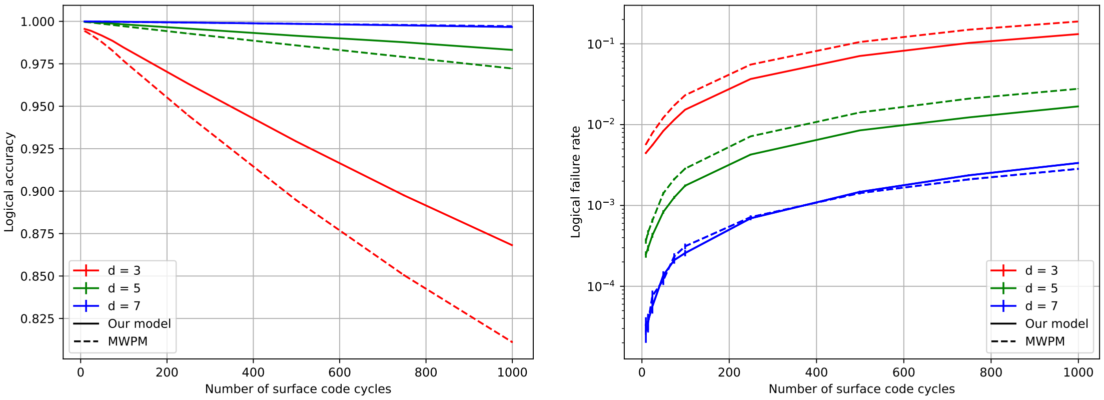


Figure 4.2: Logical accuracy (left) and logical failure rate (right) as a function of surface code cycles. The vertical bars indicate standard deviation. All models were tested with $5 \cdot 10^5$ syndromes per data point.

Figure 4.3 shows similar data compared to figure 4.2, but here the x-axis is logarithmic and the models are evaluated on syndromes up to 10^4 surface code cycles. As seen in the figure, the performance of the distance 3 model drastically decreases, approaching a logical accuracy of 0.5. This is equivalent to randomly guessing when the dataset is balanced and there are two classes, as in this case. The distance 5 model continues to confidently outperform MWPM whereas the distance 7 model is noticeable worse than MWPM. While all three models are trained on syndromes shorter than a hundred surface code cycles, the figures suggest that they are able to generalise for much longer time series. The distance 5 model, especially, performs well relative to MWPM for long time series.

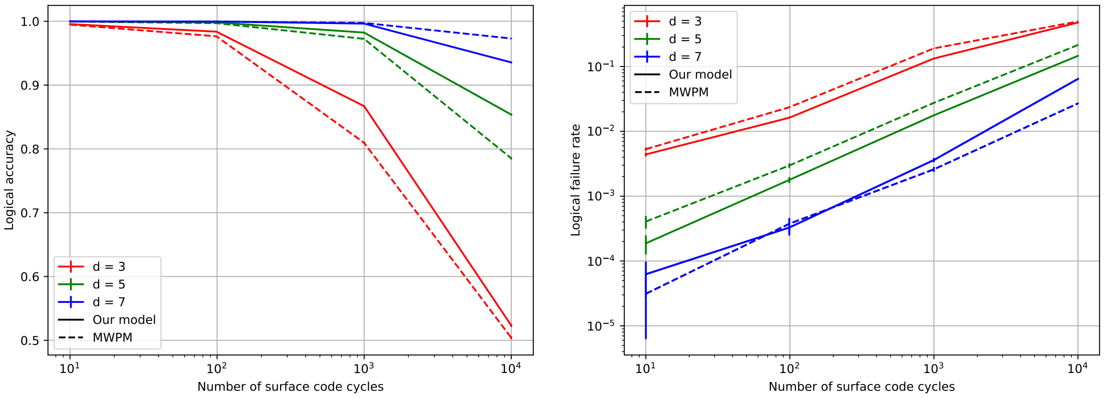


Figure 4.3: Logical accuracy (left) and logical failure rate (right) as a function of surface code cycles up to 10^4 cycles. The vertical bars indicate standard deviation. All models are tested with $6 \cdot 10^4$ syndromes per data point.

Next, the performance of the models is evaluated with respect to error rate p. Figure 4.4 shows model performance for a fixed syndrome length $T = 99$ as a function of error rate. The distance 3 and 5 models both outperform MWPM for all error rates $p \in \{0.001, 0.002, 0.003, 0.004, 0.005\}$. The distance 7 model, on the other hand, has similar performance as compared to MWPM for $p = 0.001$, but its relative performance decreases as the error rate increases.

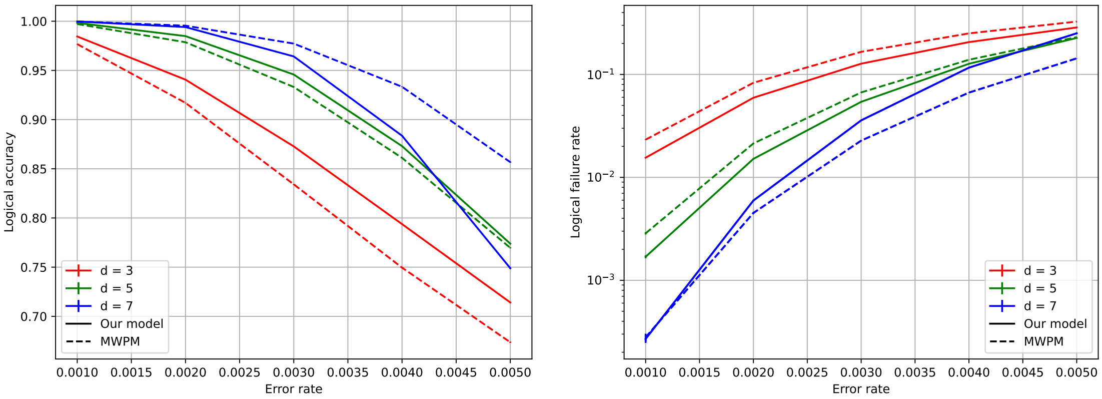


Figure 4.4: Logical accuracy (left) and logical failure rate (right) as a function of error rate. The vertical bars indicate standard deviation. All models are tested with syndrome length $T = 99$ with $5 \cdot 10^5$ syndromes per data point.

Finally, the time required to decode syndromes is evaluated. While our model is slower than MWPM on any fixed length syndrome decoding, the use of an RNN ensures that adding a new time step to the syndrome is constant time as only the new graph and the previous hidden state need to be evaluated, as mentioned in section 3.2. Meanwhile, the MWPM decoder must evaluate the entire syndrome all over again. Figure 4.5 shows how the decoding time of a single time step with our decoder converges to constant time after some initial computational overhead while the decoding time increases linearly with surface code cycles for MWPM (note the logarithmic y-axis).

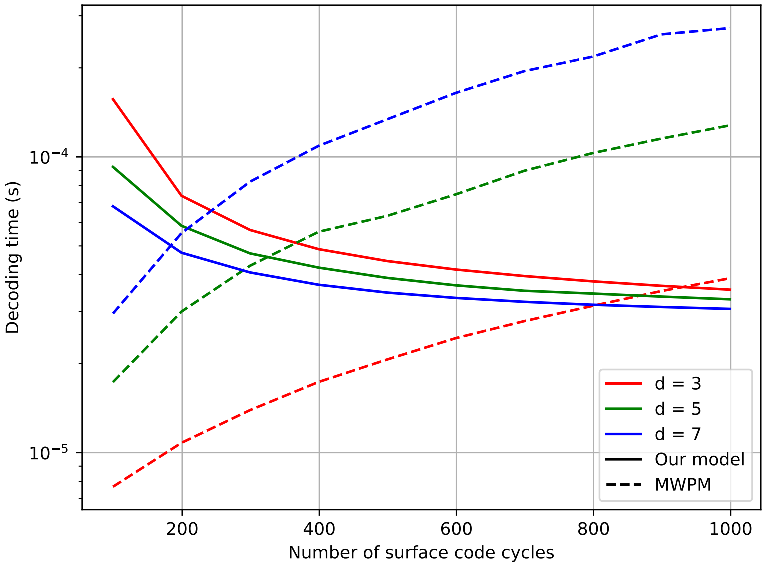


Figure 4.5: Decoding time plotted as a function of code cycles. Each data point represents the mean decoding time of $2.5 \cdot 10^4$ randomly generated syndromes. In addition, the decoding time of our model has been normalized with the number of graphs present in the syndromes, because of the recurrent nature of our decoder.

It is interesting that the decoding time for the distance 7 model appears to be lower than that for distance 3 and 5. The reason for this is likely related to the fact that, on average, a distance 7 syndrome contains more graphs than a distance 3 or 5 model for a given syndrome length, all else equal. Since the decoding time is normalized with the number of graphs, the number in the denominator associated with the distance 7 model is larger than those of distance 3 and 5, resulting in a lower decoding time.

5

Conclusion

This work introduced a data-driven approach to decoding the surface code using a combination of graph neural networks and recurrent neural networks. The decoder was trained on hundreds of millions of simulations of the surface code under surface level noise. It is able to outperform the classical MWPM algorithm on code distances 3 and 5 for thousands of surface code cycles using an error rate $p = 0.001$. However, it falls behind on code distance 7. Further, our decoder outperforms MWPM for all error rates $p \in \{0.001, 0.002, 0.003, 0.004, 0.005\}$ on code distances 3 and 5 for a fixed syndrome length $T = 99$. When it comes to decoding time, MWPM is faster than our decoder when decoding a whole syndrome at once. However, when decoding the syndrome in real time, i.e. at every time step, our decoder only needs information about the previous hidden state and the state of the surface code over the last d_T time steps. Therefore, the decoding time is constant with respect to surface code cycles. This is unlike MWPM whose decoding time grows linearly with surface code cycles.

There are a couple of problems with the approach presented in this work. The main problem relates to the amount of time required for data generation. Since we chose to always generate new data, as opposed to having a fixed data set, data must continuously be generated on the CPU and transferred to the GPU on a batch-by-batch basis during training and inference. This takes a long time, around 30%-50% of the total training time. This problem is exacerbated by the fact that the way in which the syndrome graphs are created in an overlapping fashion also takes a long time. The reason this takes a long time is that all nodes belonging to a batch are put in the same two-dimensional tensor of shape $[N, 5]$ where N is the total number of nodes in a batch and 5 is the number of node features. Ideally, a three-dimensional tensor of shape $[B, N_i, 5]$, where B is the batch size and N_i the number of nodes in syndrome i, would be used, since the node duplication could be vectorized with this approach. However, because the number of nodes varies from syndrome to syndrome, i.e. N_i is not constant across a batch, this does not work. Further, the graph convolution layers [40] of PyTorch Geometric expect a two-dimensional input. As such, much of the node duplication is done inside of a Python for-loop, as opposed to being done in a vectorized fashion, adding a further performance penalty.

The neural network architecture described in this paper only decodes the hidden state of the RNN corresponding to the last time step. This is because, by default, Stim only returns a label indicating which class the syndromes belong to for the last time step. However, with a bit of extra work, it is possible to extract the syndrome class for each time step. This could potentially improve both the performance and training time of the decoder, since all hidden states could be used during training, instead of neglecting all but the last one.

References

[1] R. Santagati, A. Aspuru-Guzik, R. Babbush, *et al.*, "Drug design on quantum computers," *Nature Physics*, vol. 20, no. 4, pp. 549–557, Mar. 2024, ISSN: 1745-2481. DOI: `10.1038/s41567-024-02411-5`. [Online]. Available: `http://dx.doi.org/10.1038/s41567-024-02411-5`.

[2] H. Shang, L. Shen, Y. Fan, *et al.*, "Large-scale simulation of quantum computational chemistry on a new sunway supercomputer," Dec. 2022. DOI: `10.1109/SC41404.2022.00019`.

[3] N. Stamatopoulos, D. J. Egger, Y. Sun, *et al.*, "Option pricing using quantum computers," *Quantum*, vol. 4, p. 291, Jul. 2020, ISSN: 2521-327X. DOI: `10.22331/q-2020-07-06-291`. [Online]. Available: `http://dx.doi.org/10.22331/q-2020-07-06-291`.

[4] P. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134. DOI: `10.1109/SFCS.1994.365700`.

[5] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997, ISSN: 1095-7111. DOI: `10.1137/s0097539795293172`. [Online]. Available: `http://dx.doi.org/10.1137/S0097539795293172`.

[6] W. K. Wootters and W. H. Zurek, "A single quantum cannot be cloned," *Nature*, vol. 299, no. 5886, pp. 802–803, Oct. 1982. DOI: `10.1038/299802a0`. [Online]. Available: `http://dx.doi.org/10.1038/299802a0`.

[7] A. Kitaev, "Fault-tolerant quantum computation by anyons," *Annals of Physics*, vol. 303, no. 1, pp. 2–30, Jan. 2003, ISSN: 0003-4916. DOI: `10.1016/s0003-4916(02)00018-0`. [Online]. Available: `http://dx.doi.org/10.1016/S0003-4916(02)00018-0`.

[8] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, "Topological quantum memory," *Journal of Mathematical Physics*, vol. 43, no. 9, pp. 4452–4505, Sep. 2002, ISSN: 1089-7658. DOI: `10.1063/1.1499754`. [Online]. Available: `http://dx.doi.org/10.1063/1.1499754`.

[9] J. Edmonds, "Paths, trees, and flowers," *Canadian Journal of Mathematics*, vol. 17, pp. 449–467, 1965. DOI: `10.4153/CJM-1965-045-4`.

[10] R. Acharya, L. Aghababaie-Beni, I. Aleiner, *et al.*, *Quantum error correction below the surface code threshold*, 2024. arXiv: `2408.13687 [quant-ph]`. [Online]. Available: `https://arxiv.org/abs/2408.13687`.

[11] J. Bausch, A. W. Senior, F. J. H. Heras, *et al.*, "Learning high-accuracy error decoding for quantum processors," *Nature*, vol. 635, no. 8040, pp. 834–840,

Nov. 2024, ISSN: 1476-4687. DOI: `10.1038/s41586-024-08148-8`. [Online]. Available: `http://dx.doi.org/10.1038/s41586-024-08148-8`.

[12] G. Torlai and R. G. Melko, "Neural decoder for topological codes," *Physical Review Letters*, vol. 119, no. 3, Jul. 2017, ISSN: 1079-7114. DOI: `10.1103/physrevlett.119.030501`. [Online]. Available: `http://dx.doi.org/10.1103/PhysRevLett.119.030501`.

[13] S. Krastanov and L. Jiang, "Deep neural network probabilistic decoder for stabilizer codes," *Scientific Reports*, vol. 7, no. 1, Sep. 2017, ISSN: 2045-2322. DOI: `10.1038/s41598-017-11266-1`. [Online]. Available: `http://dx.doi.org/10.1038/s41598-017-11266-1`.

[14] S. Varsamopoulos, B. Criger, and K. Bertels, "Decoding small surface codes with feedforward neural networks," *Quantum Science and Technology*, vol. 3, no. 1, p. 015 004, Nov. 2017, ISSN: 2058-9565. DOI: `10.1088/2058-9565/aa955a`. [Online]. Available: `http://dx.doi.org/10.1088/2058-9565/aa955a`.

[15] 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, ISSN: 2521-327X. DOI: `10.22331/q-2018-01-29-48`. [Online]. Available: `http://dx.doi.org/10.22331/q-2018-01-29-48`.

[16] N. P. Breuckmann and X. Ni, "Scalable neural network decoders for higher dimensional quantum codes," *Quantum*, vol. 2, p. 68, May 2018, ISSN: 2521-327X. DOI: `10.22331/q-2018-05-24-68`. [Online]. Available: `http://dx.doi.org/10.22331/q-2018-05-24-68`.

[17] S. Varsamopoulos, K. Bertels, and C. G. Almudever, "Comparing neural network based decoders for the surface code," *IEEE Transactions on Computers*, vol. 69, no. 2, pp. 300–311, Feb. 2020, ISSN: 2326-3814. DOI: `10.1109/tc.2019.2948612`. [Online]. Available: `http://dx.doi.org/10.1109/TC.2019.2948612`.

[18] P. Baireuther, M. D. Caio, B. Criger, C. W. J. Beenakker, and T. E. O'Brien, "Neural network decoder for topological color codes with circuit level noise," *New Journal of Physics*, vol. 21, no. 1, p. 013 003, Jan. 2019, ISSN: 1367-2630. DOI: `10.1088/1367-2630/aaf29e`. [Online]. Available: `http://dx.doi.org/10.1088/1367-2630/aaf29e`.

[19] B. M. Varbanov, M. Serra-Peralta, D. Byfield, and B. M. Terhal, "Neural network decoder for near-term surface-code experiments," *Phys. Rev. Res.*, vol. 7, p. 013 029, 1 Jan. 2025. DOI: `10.1103/PhysRevResearch.7.013029`. [Online]. Available: `https://link.aps.org/doi/10.1103/PhysRevResearch.7.013029`.

[20] H. Jung, I. Ali, and J. Ha, "Convolutional neural decoder for surface codes," *IEEE Transactions on Quantum Engineering*, vol. 5, pp. 1–13, 2024. DOI: `10.1109/TQE.2024.3419773`.

[21] V. K, D. S, and S. T, "Rl-qec: Harnessing reinforcement learning for quantum error correction advancements," in *2024 International Conference on Trends in Quantum Computing and Emerging Business Technologies*, 2024, pp. 1–5. DOI: `10.1109/TQCEBT59414.2024.10545200`.

[22] M. Lange, P. Havström, B. Srivastava, *et al.*, *Data-driven decoding of quantum error correcting codes using graph neural networks*, 2023. arXiv: `2307.01241 [quant-ph]`. [Online]. Available: `https://arxiv.org/abs/2307.01241`.

[23] J. Bausch, A. W. Senior, F. J. H. Heras, *et al.*, "Learning to decode the surface code with a recurrent, transformer-based neural network," Oct. 2023. [Online]. Available: `https://doi.org/10.48550/arXiv.2310.05900`.

[24] J. Roffe, "Quantum error correction: An introductory guide," *Contemporary Physics*, vol. 60, no. 3, pp. 226–245, Jul. 2019, ISSN: 1366-5812. DOI: `10.1080/00107514.2019.1667078`. [Online]. Available: `http://dx.doi.org/10.1080/00107514.2019.1667078`.

[25] A. deMarti iOlius, P. Fuentes, R. Orús, P. M. Crespo, and J. Etxezarreta Martinez, "Decoding algorithms for surface codes," *Quantum*, vol. 8, p. 1498, Oct. 2024, ISSN: 2521-327X. DOI: `10.22331/q-2024-10-10-1498`. [Online]. Available: `http://dx.doi.org/10.22331/q-2024-10-10-1498`.

[26] W. Mcculloch and W. Pitts, "A logical calculus of ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 127–147, 1943.

[27] S. Linnainmaa, "Algoritmin kumulatiivinen pyöristysvirhe yksittäisten pyöristysvirheiden taylor-kehitelmänä (the representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors)," *Thesis*, 1970. [Online]. Available: `https://people.idsia.ch/~juergen/linnainmaa1970thesis.pdf`.

[28] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, pp. 179–211, 1990.

[29] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," Apr. 1991.

[30] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994. DOI: `10.1109/72.279181`.

[31] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, Nov. 1997. DOI: `10.1162/neco.1997.9.8.1735`.

[32] K. Cho, B. van Merrienboer, C. Gulcehre, *et al.*, *Learning phrase representations using rnn encoder-decoder for statistical machine translation*, 2014. arXiv: `1406.1078 [cs.CL]`. [Online]. Available: `https://arxiv.org/abs/1406.1078`.

[33] C. Morris, M. Ritzert, M. Fey, *et al.*, *Weisfeiler and leman go neural: Higher-order graph neural networks*, 2021. arXiv: `1810.02244 [cs.LG]`. [Online]. Available: `https://arxiv.org/abs/1810.02244`.

[34] C. Gidney, "Stim: A fast stabilizer circuit simulator," *Quantum*, vol. 5, p. 497, Jul. 2021, ISSN: 2521-327X. DOI: `10.22331/q-2021-07-06-497`. [Online]. Available: `https://doi.org/10.22331/q-2021-07-06-497`.

[35] J. Ansel, E. Yang, H. He, *et al.*, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*, ACM, Apr. 2024. DOI: `10.1145/3620665.3640366`. [Online]. Available: `https://pytorch.org/assets/pytorch2-2.pdf`.

[36] M. Fey and J. E. Lenssen, *Fast graph representation learning with pytorch geometric*, 2019. arXiv: `1903.02428 [cs.LG]`. [Online]. Available: `https://arxiv.org/abs/1903.02428`.

[37] M. Altalhan, A. Algarni, and M. Turki-Hadj Alouane, “Imbalanced data problem in machine learning: A review,” *IEEE Access*, vol. 13, pp. 13 686–13 699, 2025. DOI: `10.1109/ACCESS.2025.3531662`.

[38] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: `1412.6980 [cs.LG]`. [Online]. Available: `https://arxiv.org/abs/1412.6980`.

[39] O. Higgott, *Pymatching: A python package for decoding quantum codes with minimum-weight perfect matching*, 2021. arXiv: `2105.13082 [quant-ph]`. [Online]. Available: `https://arxiv.org/abs/2105.13082`.

[40] *Conv.graphconv*, 2025. [Online]. Available: `https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.conv.GraphConv.html#torch_geometric.nn.conv.GraphConv`.

A

Appendix

A.1 Appendix 1

The code for this project can be found at github.com/Olfj/QEC_GNN-RNN

A.2 Appendix 2

Network parameters for the models presented in this thesis.

Layer	**Input Features**	**Output Features**	**Param #**
GraphConv 1	5	32	352
GraphConv 2	32	64	4,160
GraphConv 3	64	128	16,512
GraphConv 4	128	256	65,792
GRU	256, 128	128	445,440
Linear	128	1	129
Total			**532,385**

Table A.1: Layer-wise input/output features and parameter counts of the distance three and five GRUDecoder model

Layer	**Input Features**	**Output Features**	**Param #**
GraphConv 1	5	32	352
GraphConv 2	32	64	4,160
GraphConv 3	64	128	16,512
GraphConv 4	128	256	65,792
GraphConv 5	256	512	262,656
GRU	512, 256	256	1,775,616
Linear	256	1	257
Total			**2,125,345**

Table A.2: Layer-wise input/output features and parameter counts of the larger distance seven GRUDecoder model

DEPARTMENT OF PHYSICS
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY