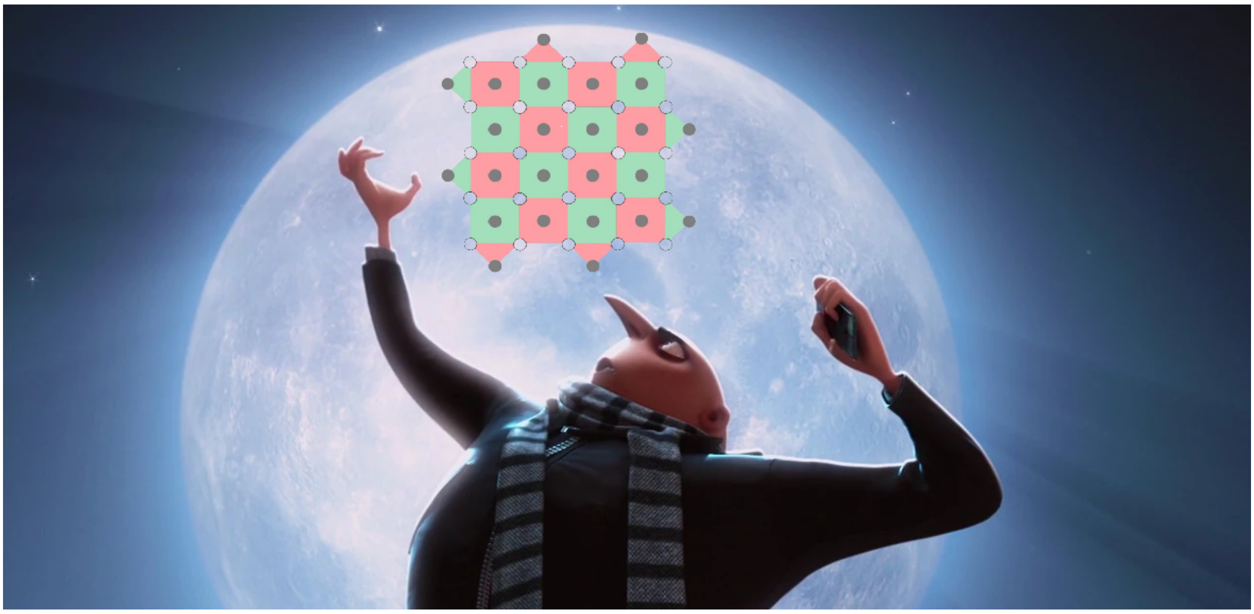
CHALMERS
UNIVERSITY OF TECHNOLOGY



Sequential Graph-Based Decoding of the Surface Code

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

OLE FJELDSÅ, GUSTAF JONASSON JOHANSSON

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2025

Sequential Graph-Based Decoding of the Surface Code

OLE FJELDSÅ
GUSTAF JONASSON JOHANSSON

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
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 Lastnamesen 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

Contents

List of Figures

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.

This is mostly the introduction from the planning report. When the results are done we should give a quick mention of them at the end of the introduction.

To allow for error correction in a classical computer we can use a simple repetition code. For example, a simple three bit encoding $\{0, 1\} \rightarrow \{000, 111\}$ 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 logical qubits as a two-dimensional lattice of physical qubits with continuous boundary conditions.

Efficiently decoding the surface code is a challenging problem. One widely used approach is the Minimum Weight Perfect Matching (MWPM) algorithm [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 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 to generalize the decoding over variable time frames.

Add something about our results here "we show that..."

2

Theory

2.1 Quantum computing

There is already a lot of written thory in the ”notes” document. Have a look there for theory regarding quantum computing.

This chapter handles the basics of quantum computing. It especially needs to explain:

- ✓ The qubit
- ✓ The Pauli operators/group
- ✓ Quantum circuits
- □ Important gates
- ✓ Entanglement
- □ Commuting/anti-commuting operations.

Most of this chapter originally comes from the quantum computing course at freecodecamp.org [23]. This chapter needs a lot more references.

2.1.1 Qubit

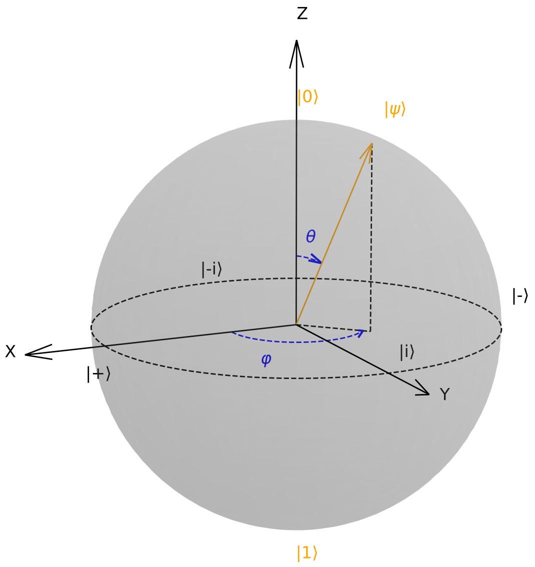


Figure 2.1: The Bloch sphere. Any state corresponds to a point on the sphere, the positive and negative Z direction represent the negative and positive basis states $|0\rangle$ and $|1\rangle$. The position of the state is defined by (2.1).

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

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

2.1.2 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 $|\psi\rangle$ yield

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

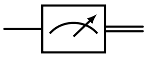
$$\mathcal{G}_1 = \{\pm\mathbb{1}, \pm i\mathbb{1}, \pm X, \pm iX, \pm Y, \pm iY, \pm Z, \pm iZ\},$$

with the $\pm 1, \pm i$ terms to ensure that $\mathcal{G}_1$ is closed under multiplication. 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

$$\mathbb{1} \otimes X \otimes Y = X_2 Y_3,$$

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.

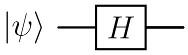
2.1.3 Quantum circuits

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 (2.4). Multi line qubits are represented using vertical connections between qubits as in (2.5) 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  .

2.1.4 Quantum gates

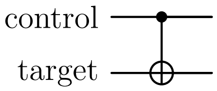
Besides the Pauli operators we need some additional quantum operators, namely the quantum gates H, $CNOT$ and CZ. The Hadamard gate is defined as

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \tag{2.4}$$

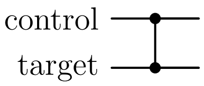


It has the effect $H|0\rangle = |+\rangle$, $H|1\rangle = |-\rangle$, and vice versa. In this way the Hadamard gate can be used to bring a qubit in and out of superposition. The $CNOT$ and CZ gates are defined as

$$CNOT = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{pmatrix} \tag{2.5}$$



$$CZ = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & -1\end{pmatrix} \tag{2.6}$$



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 $|11\rangle$.

rest is explained in phase kickback

2.1.5 Phase kickback

2.1.6 Entanglement

expand on why we need entanglement

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.

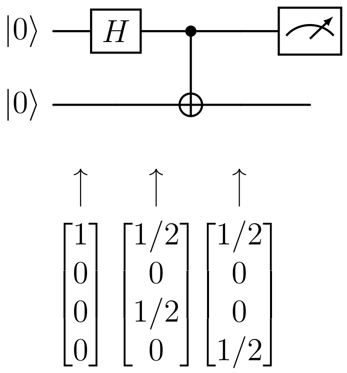

$|0\rangle$
H
$|0\rangle$
$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$
$\begin{bmatrix}1/2\\0\\1/2\\0\end{bmatrix}$
$\begin{bmatrix}1/2\\0\\0\\1/2\end{bmatrix}$

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

2.2 QEC

This section covers the quantum error correction parts of the thesis. I feel like we should mention that this chapter leans heavily on other peoples work (done in citations I guess). Especially [24] is used a lot, both for content and structure, this section would probably look quite different if not for this paper.

□ Repitition code using the the previous section as tools.

□ The stabilizer formalism.

□ The rotated surface code.

In order to facilitate reliable QEC, one needs to introduce redundancy in the way a qubit is represented. In practice, this means using multiple qubits to encode a single so-called logical qubit. If only one qubit is used, it is not possible to surmise 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 two different methods of encoding logical qubits.

2.2.1 Bit flip repetition code

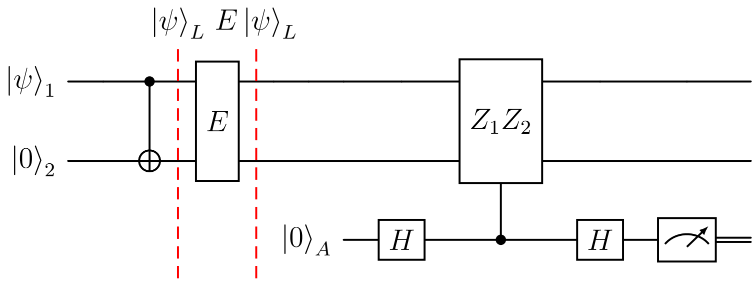


Figure 2.3: The two qubit encoder.

To allow for redundancy in a quantum code we can not simply copy bits. Instead we expand the Hilbert space of the qubits, to do this we spread 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 parametrised 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

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

On the two qubit system we have the following bit flip errors.

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

Table 2.1: 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

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

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

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

Note that in both cases the change is applied to the entangled ancilla qubit. 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

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

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

$$Z_1Z_2|\psi\rangle_L = \lambda|\psi\rangle_L$$

and the total state becomes:

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

We now apply H again to our entangled ancilla

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

Since $\lambda = \pm 1$:

- If $\lambda = 1$, then $|0\rangle$ is measured.
- If $\lambda = -1$, then $|1\rangle$ is obtained.

Thus, measuring the ancilla qubit tells us whether an error has occurred.

2.2.2 Stabilizers

Write about the stabilizer formalism here

2.2.3 The surface code

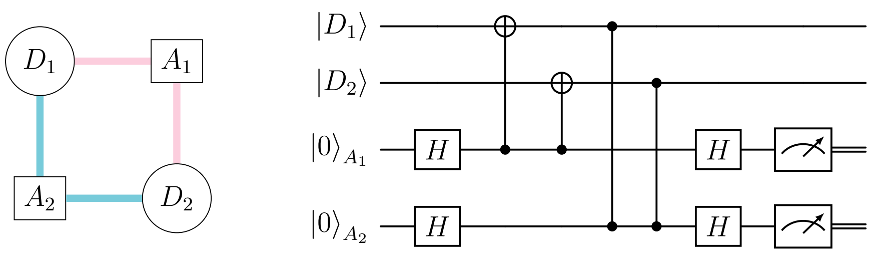


Figure 2.4: Surface code four-cycle. The left hand side diagram shows a simplified representation of the quantum circuit diagram on the right. Pink represents X stabilizers while blue represents Z stabilizers.

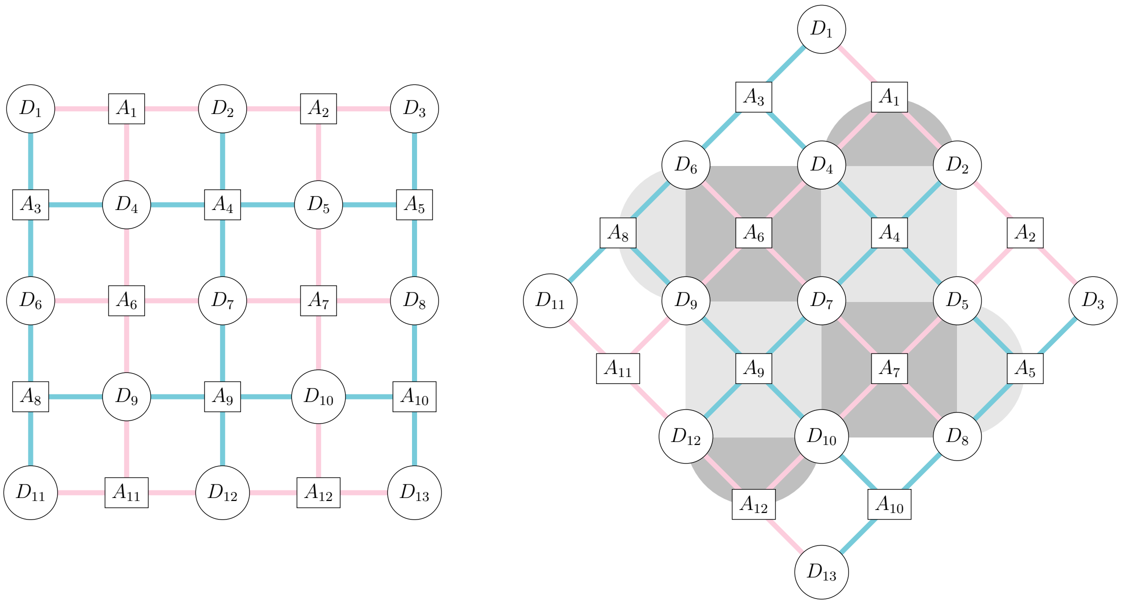

D_1 A_1 D_2 A_2 D_3
A_3 D_4 A_4 D_5 A_5
D_6 A_6 D_7 A_7 D_8
A_8 D_9 A_9 D_{10} A_{10}
D_{11} A_{11} D_{12} A_{12} D_{13}
D_1
A_3 A_1
D_6 D_4 D_2
A_8 A_6 A_4 A_2
D_{11} D_9 D_7 D_5 D_3
A_{11} A_9 A_7 A_5
D_{12} D_{10} D_8
A_{12} A_{10}
D_{13}

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

- Surface code four cycle
- concatenation
- Scaling
- what is a bit/phase flip
- describe difficulties here. why can it not just be read straight of.

- Introduce logical basis X_L Z_L and so on

2.2.4 Minimum weight perfect matching

A graph is a pair $\mathcal{G} = (\mathbf{N}, \mathbf{E})$, where $\mathbf{N}$ is a set of nodes, and $\mathbf{E}$ are edges connecting them.

2.3 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.

2.3.1 Feed forward, fully connected neural networks

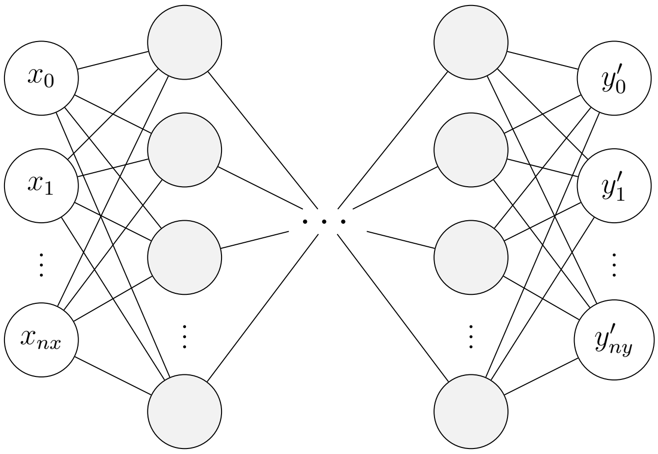


Figure 2.6: Deep neural network with input $\mathbf{X} = \{x_0, x_1, \ldots, x_{nx}\}$, a sequence of hidden layers, and an output $\mathbf{Y}' = \{y'_0, y'_1, \ldots, y'_{ny}\}$. Each layer $\mathbf{X}_l$ is updated according to (2.9) with the input of each layer being the output of the previous.

First draft, not re-searched, might need some fact checking

The building block of neural networks is the (artificial) neuron, first proposed as a mathematical model for the nervous system [25]. The neuron N can be written as

$$N(\mathbf{X}) = f\left(b + \sum_{x_j \in \mathbf{X}} x_j w_j\right) \tag{2.7}$$

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 $\mathbf{X}_l$

$$\mathbf{X}_l(\mathbf{X}) = \begin{bmatrix} N_0 \\ N_1 \\ \vdots \\ N_n \end{bmatrix}, \qquad N_i(\mathbf{X}) = f\left(b_i + \sum_{x_j \in \mathbf{X}} x_j w_{ji}\right) \tag{2.8}$$

This can be rewritten as the matrix operation

$$\mathbf{X}_l(\mathbf{X}) = f\left(\mathbf{W}_l \mathbf{X} + \mathbf{B}_l\right) \tag{2.9}$$

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 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 2.6.

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 [26]. In this process every parameter p (weights and biases) is updated according to $p' = -\eta \frac{\partial E(Y,Y')}{\partial p}$, here $E(y, y')$ 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 (2.9) we get the update rule

$$\begin{aligned} \nabla_l &= \mathbf{W}_l^T \nabla_{l+1} && \text{(2.10a)} \\ \nabla_W &= \nabla_{l+1} \mathbf{X}_{l-1} && \text{(2.10b)} \\ \mathbf{W}_l' &= -\eta \nabla_W && \text{(2.10c)} \\ \mathbf{B}_l' &= -\eta \nabla_l && \text{(2.10d)} \end{aligned}$$

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

2.3.2 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) [27] 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 2.7.

maybe explicitly mention that the hidden state is the output from the rnn

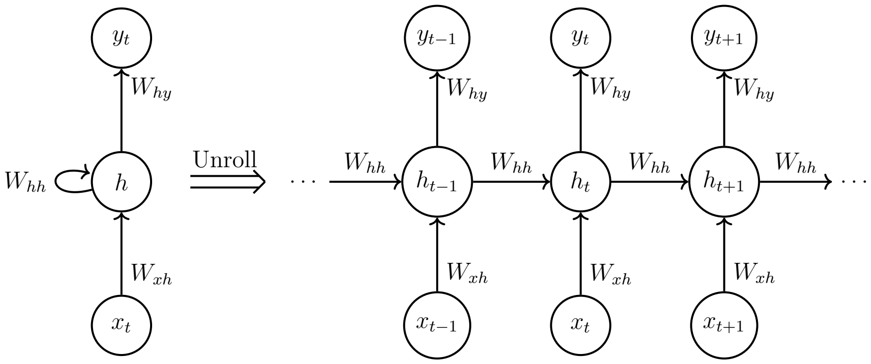


Figure 2.7: Fundamental structure for a recurrent neural network. The hidden layer(s) feed their state to the output and to the output for the current time step. The figure shows both the unrolled and rolled up schematic at time t.

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

2.3.3 Gated Recurrent Unit (GRU)

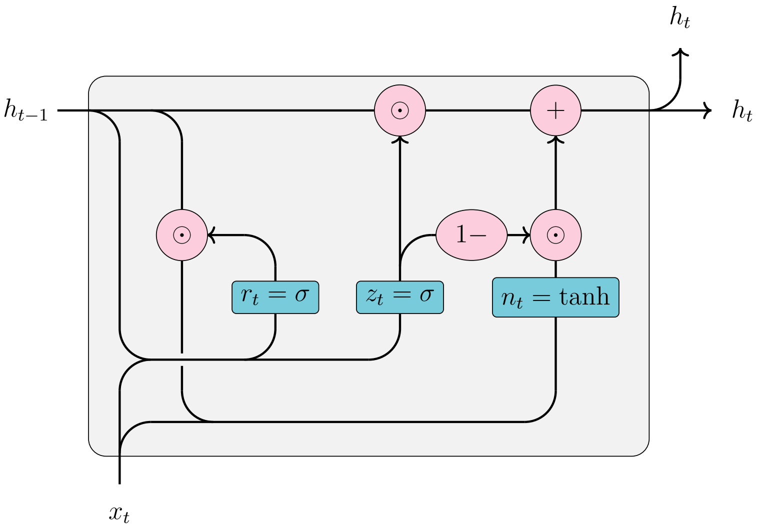


Figure 2.8: Circuit diagram of a single GRU block. r_t, z_t, n_t and h_t are calculated according to (2.11).

Using fully connected layers for the hidden state of the RNN can run into the vanishing gradient problem for longer time series [28], [29]. One approach to mitigating this is to use long short-term memory (LSTM) [30] 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) [31] builds on the LSTM but removes the cell state vector, combining long and short term memory, giving it the

Should we describe LSTM in greater detail? We already have a short section about it but it does not directly apply to our thesis.

parameters

$$r_t = \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \tag{2.11a}$$
$$z_t = \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \tag{2.11b}$$
$$n_t = \tanh(W_{in}x_t + b_{in} + r_t \odot (W_{hn}h_{(t-1)} + b_{hn})) \tag{2.11c}$$
$$h_t = (1 - z_t) \odot n_t + z_t \odot h_{(t-1)} \tag{2.11d}$$
$$\sigma(x) = \left(1 + e^{-x}\right)^{-1} = \frac{1}{1 + e^{-x}} = \frac{1}{1 + \exp(-x)}$$

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 2.8 gives a more illustrative view of the GRU block.

2.3.4 Graph convolution layer

[32]–[34]
A graph is a pair $\mathcal{G} = (\mathbf{N}, \mathbf{E})$, where $\mathbf{N}$ is a set of nodes, and $\mathbf{E}$ are edges connecting them. Many problems can be mapped to graphs and graphs come in many shapes and sizes.

move citations into the running text, maybe we can give the definition of a graph when explaining MWPM

Make this short introduction a bit longer, mention that GCN is a type of GNN

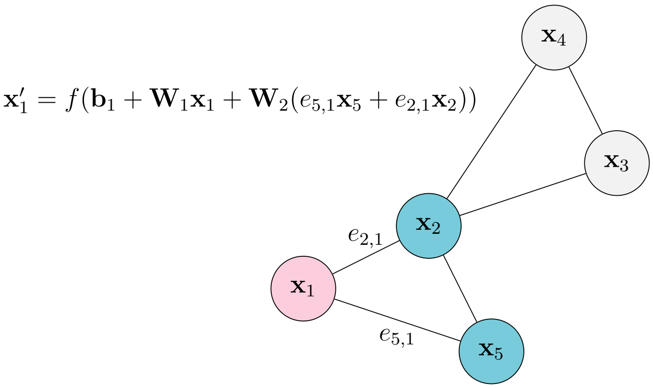

$\mathbf{x}_1' = f(\mathbf{b}_1 + \mathbf{W}_1\mathbf{x}_1 + \mathbf{W}_2(e_{5,1}\mathbf{x}_5 + e_{2,1}\mathbf{x}_2))$
$\mathbf{x}_4$
$\mathbf{x}_3$
$\mathbf{x}_2$
$e_{2,1}$
$\mathbf{x}_1$
$e_{5,1}$
$\mathbf{x}_5$

Figure 2.9: The update of a single node feature x_1', according to (2.12) during graph convolution.

The graph convolution layer maps every node feature $x_1, x_2, \ldots$ to a new state according to (2.12). Like the 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 2.9 shows the update rule acting on a single node feature during a graph convolution.

$$\mathbf{x}_i' = f\left(\mathbf{b}_i + \mathbf{W}_1\mathbf{x}_i + \mathbf{W}_2 \sum_{j\in\mathcal{N}_i} e_{ji}\mathbf{x}_j\right) \tag{2.12}$$

2.3.5 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

$$\mathrm{gmp}(\mathcal{G}) = \frac{1}{|N_{\mathcal{G}}|} \sum_{\mathbf{x} \in N_{\mathcal{G}}} \mathbf{x} \tag{2.13}$$

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 [35] 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* . 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.

mention more here, like when the errors are applied, we should do this under ”surface code” as we explain the equivalence classes

Having generated a syndrome, a series of graphs each spanning d_T time steps are created. The graphs consist of a set of a nodes N, and a set of edges E. 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 20 nearest neighbours. These edges are created using a k-nearest neighbour algorithm. Generally, the graphs contain less than 20 nodes, and as such are undirected and fully connected. In principle, though, the graphs can contain more than 20 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

There should probably just be a short chapter about graphs in theory.

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:

maybe motivate why 20 neighbours

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

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.

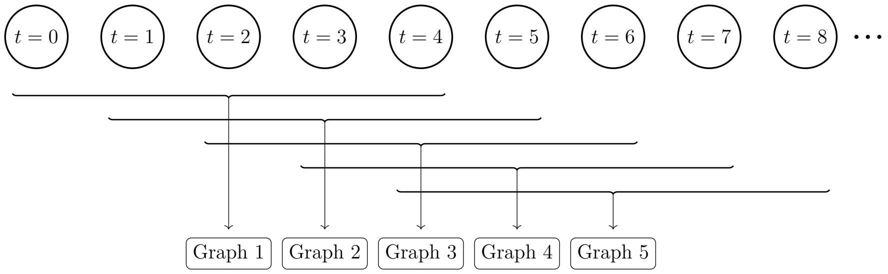


Figure 3.1: Diagram showing how graphs are created according to a sliding window schema. The circles represent all nodes that are present in the corresponding time step t.

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

elaborate here maybe?

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 $d = \{3, 5, 7\}$.

expand on this, explain plot

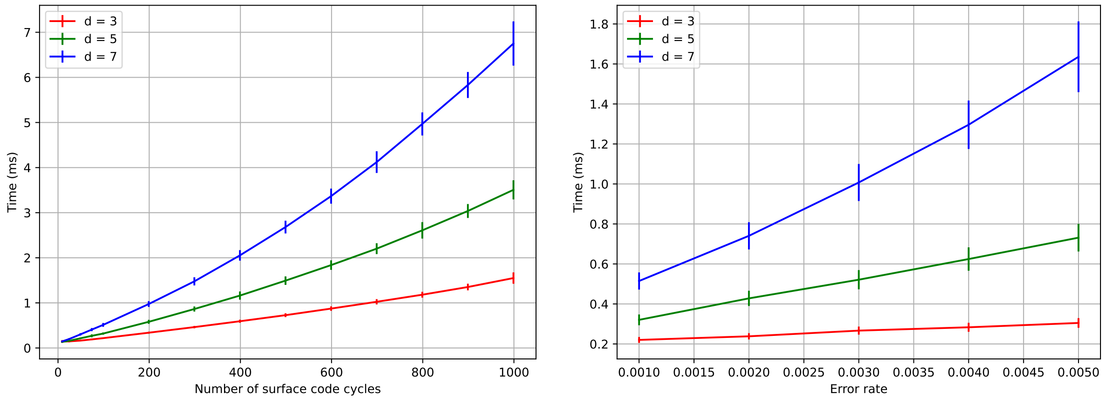


Figure 3.2: Time required to generate syndrome graphs based on surface codes of distance $d = \{3, 5, 7\}$, with $d_T = 2$. The left panel shows how the time to generate syndrome graphs changes with syndrome length. The right panel instead shows how the time to generate syndrome graphs changes with error rate p, and fixed syndrome length $T = 99$. The error bars represent one standard deviation. The results are based on $5 * 10^3$ 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 2.3.5. 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 range (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 X, Y, Z, or I class. We exclusively trained networks to detect logical bit-flips, but it is trivial to . Figure 3.3 shows an overview of the model pipeline.

Not necessarily, but two Dense networks could be good

write that we its easy to train it to learn phase-flips too

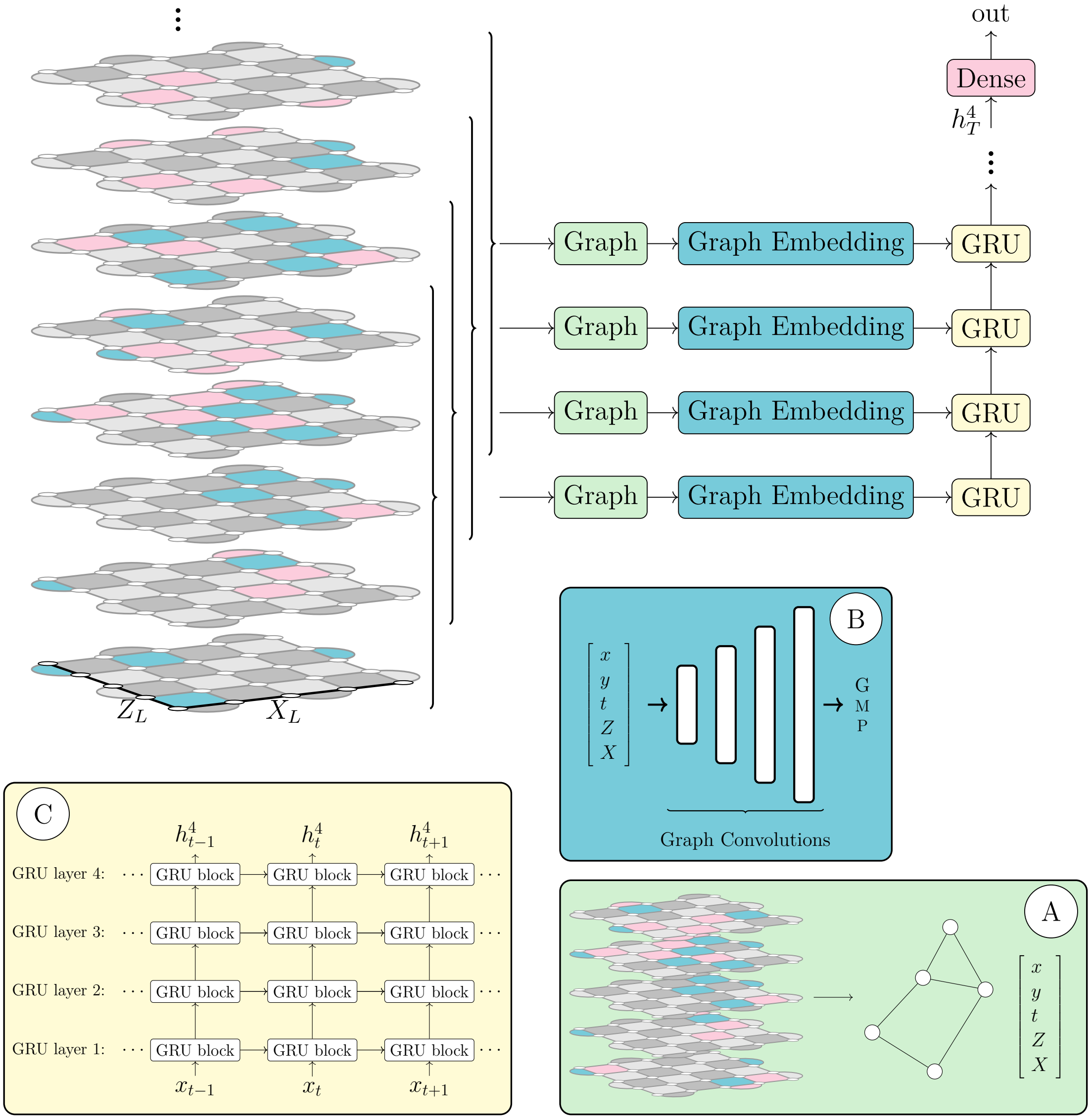


Figure 3.3: Overview of the entire model pipeline. **A**: Transfer surface codes to graph over all detector events. **B**: Embedding from graph to embedding tensor using graph convolutions. **C**: four layer Gated Recurrent Unit.

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. In other words, decoding a syndrome at time step $t = 7999$ takes the same amount of time as decoding a syndrome at $t = 99$. This is not the case for other decoders, such as MWPM.

this assumes that all the previous graphs have been fed through the neural net

Our decoder was implemented using PyTorch [36] and PyTorch Geometric (PyG) [37]. 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, we obtain data by generating it using Stim. Instead of generating fixed training and test sets, data is 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 $5 * 10^5$ 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 unique.

We can maybe mention that this is done in large part just for convention? Besides, people like to watch training plots with performance as a function of epochs

We could mention the probability of getting two equal syndromes or something like that?

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 [38]. 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.

why is d3 higher than d7? because less qubits? shouldnt d7 be more resistant.

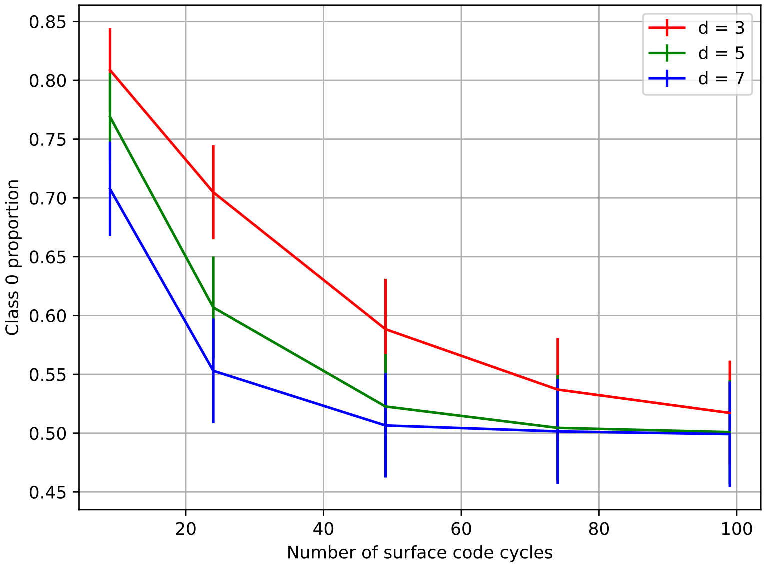


Figure 3.4: Proportion of class 0 syndromes as a function of syndrome length for code distances $d = \{3, 5, 7\}$, with error rate $p = 0.003$. The error bars represent one standard deviation. The results are based on roughly $2.5 * 10^5$ randomly generated syndromes per data point.

It is important to note that we always exclude syndromes that contain no detection events, 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 [39], 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:

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

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 was decreased exponentially to 10^{-4} according to $\eta_t = 0.95^t \eta_0$, where t corresponds to the current epoch.

4

Results

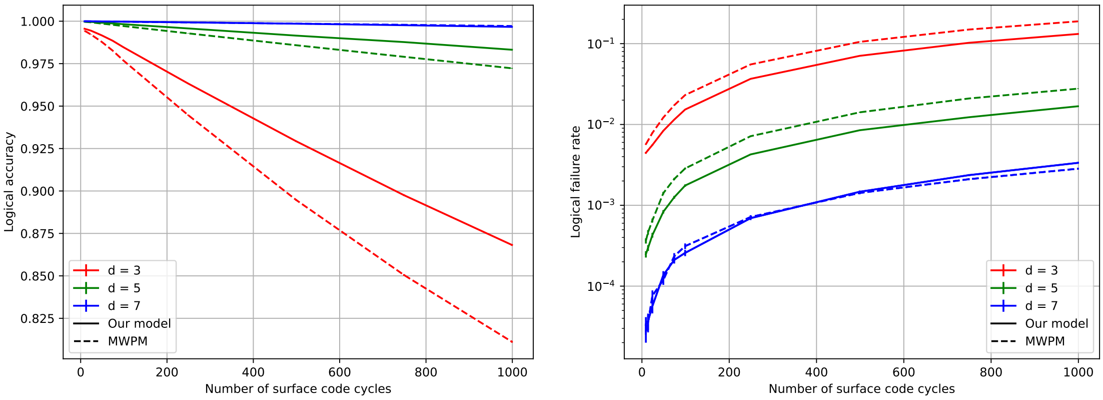


Figure 4.1: Caption

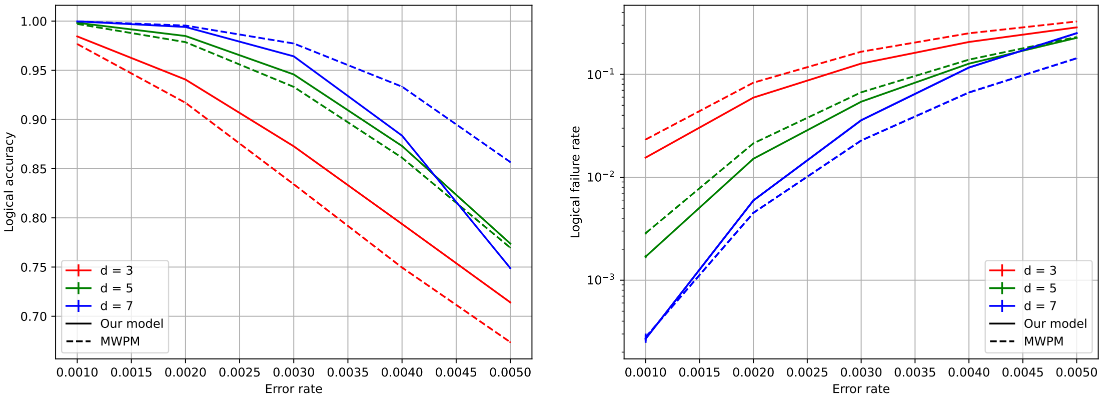


Figure 4.2: Caption

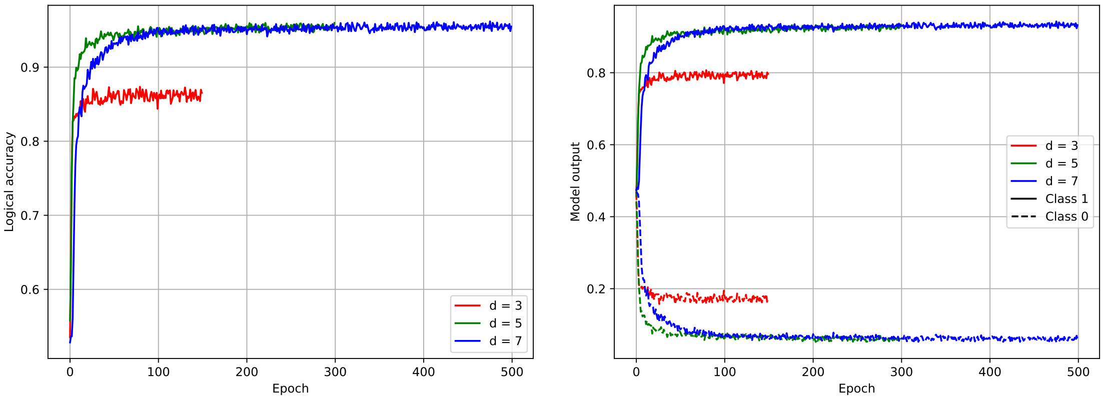

Logical accuracy
0.6
0.7
0.8
0.9
0
100
200
300
400
500
Epoch
d = 3
d = 5
d = 7
Model output
0.2
0.4
0.6
0.8
0
100
200
300
400
500
Epoch
d = 3
d = 5
d = 7
Class 1
Class 0

Figure 4.3

5
Conclusion

The RNN can indeed generalize over longer time series beating mwpm on sizes 3 and 5.

There are several things that can be improved

- Data generation is in need of a speedup, espercially the sliding window generation. This could be done before creating the sparse events as the uniform shape lends itself better to vectorization.
- Backpropagation on the bit/phase flip for each time step.
-

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] quantum-soar. "Quantum computing course – math and theory for beginners," freeCodeCamp.org. (May 2024), [Online]. Available: `https://www.youtube.com/watch?v=tsbCSkvHhMo`.

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

[26] 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`.

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

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

[29] 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`.

[30] 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`.

[31] 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`.

[32] 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`.

[33] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 2, 2005, 729–734 vol. 2. DOI: `10.1109/IJCNN.2005.1555942`.

[34] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[35] 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`.

[36] 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 Sup-*

port 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`.

[37] 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`.

[38] 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`.

[39] 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`.

A
Appendix 1

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

B

Appendix 2

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