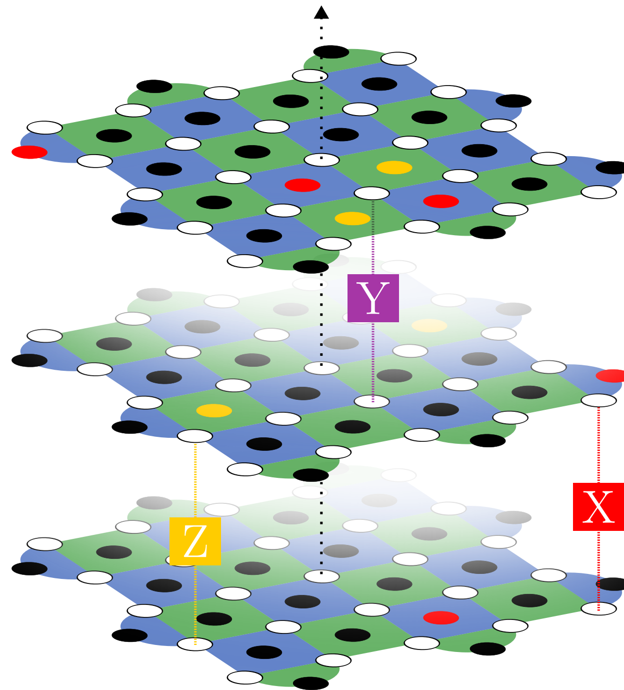
CHALMERS
UNIVERSITY OF TECHNOLOGY



Data-driven decoding of the surface code using a neural matching decoder

A decoder combining a graph neural network with a minimum-weight perfect matching algorithm

Master's thesis in Physics and Complex Adaptive Systems

ISAK BENGTSSON

FRIDA FJELDDAHL

DEPARTMENT OF PHYSICS
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024
www.chalmers.se

Master's thesis 2024

Data-driven decoding of the surface code using a neural matching decoder

ISAK BENGTSSON, FRIDA FJELDDAHL

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2024

Data-driven decoding of the surface code using a neural matching decoder
ISAK BENGTSSON
FRIDA FJELDDAHL

© ISAK BENGTSSON, FRIDA FJELDDAHL, 2024.

Supervisor: Mats Granath, Department of Physics
Examiner: Mats Granath, Department of Physics

Master's Thesis 2024
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Three cycles of stabilizer measurements on the rotated surface code.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2024

Data-driven decoding of the surface code using a neural matching decoder

ISAK BENGTSSON
FRIDA FJELDDAHL
Department of Physics
Chalmers University of Technology

Abstract

Quantum error correction is a prerequisite to achieve fault-tolerant quantum computation with noisy qubits. A promising approach is the surface code, in which the qubits have nearest-neighbour connectivity and are arranged on a two-dimensional grid. Concurrent stabilizer measurements across the grid project the qubits to a two-dimensional code space, encoding a single logical qubit. Subsequent stabilizer measurements detect if the logical qubit leaves the code space, but the measurements must be interpreted by a quantum error correction decoder to find what the actual errors are. Decoding the surface code is computationally intensive, and the decoder must be both fast and accurate if quantum error correction is to work. In this thesis, we present the novel neural matching decoder (NMD). Our decoder combines a graph neural network (GNN) with a more traditional choice, a minimum-weight perfect matching (MWPM) algorithm. Our approach is purely data-driven and does not require any prior information about the noise processes in the quantum circuits, instead the NMD is trained on 10^9 simulated error syndromes. This stands in contrast to conventional decoders, that require detailed noise models to achieve high decoding accuracies. Given a surface code of size $d = 5$ influenced by circuit-level noise for $d_t = 5$ time steps, we show that the NMD can reach relatively high accuracies and approaches the results of a conventional MWPM decoder. This indicates that the NMD can learn the underlying noise model directly from a distribution of error syndromes. It also shows that it is possible to create a decoder by combining neural networks with a conventional algorithm. This is interesting from a computational perspective, as it could help limit the computational resources needed by the decoder whilst retaining a high accuracy. Our work opens up a new direction in the field of quantum error correction decoders, by combining machine learning with a graph algorithm.

Keywords: quantum error correction, surface code, MWPM, graph neural networks, GNN.

Acknowledgements

We would like to express our gratitude towards Mats Granath, who has supported us throughout the project and always been eager to discuss theory and ideas. It has always been very appreciated. Furthermore, we would like to thank Moritz Lange for explaining his work on a graph neural network decoder.

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.

Isak Bengtsson and Frida Fjelddahl, Gothenburg, May 2024

List of Acronyms

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

ATN	Attention
FF	Feed-forward
GCN	Graph Convolutional Network
GNN	Graph Neural Network
MLP	Multi-Layer Perceptron
MSE	Mean Squared Error
LS	Local Search
MWPM	Minimum Weight Perfect Matching
NMD	Neural Matching Decoder
QEC	Quantum Error Correction
SGD	Stochastic Gradient Descent

Contents

List of Acronyms ix

List of Figures xiii

List of Tables xvii

1 Introduction 1
- 1.1 Scope . . . 2

2 Theory 3
- 2.1 Quantum computation . . . 3
  - 2.1.1 The qubit state . . . 3
  - 2.1.2 Quantum logical gates . . . 5
  - 2.1.3 Noise and errors . . . 7
- 2.2 Quantum error correction . . . 8
  - 2.2.1 Repetition code . . . 8
  - 2.2.2 Stabilizer formalism . . . 11
  - 2.2.3 Surface codes . . . 13
  - 2.2.4 Error syndromes and correction operators . . . 17
- 2.3 Decoding an error syndrome . . . 18
  - 2.3.1 Syndrome graphs . . . 20
  - 2.3.2 Minimum-weight perfect matching decoders . . . 21
- 2.4 Graph neural networks . . . 22
  - 2.4.1 Fundamentals of a neural network . . . 22
  - 2.4.2 Exploiting graph structures in neural networks . . . 24
  - 2.4.3 Neural network training . . . 26

3 Methods 29
- 3.1 Neural matching decoder . . . 29
  - 3.1.1 Decoder overview . . . 30
  - 3.1.2 Sampling of error syndromes . . . 31
  - 3.1.3 Measurements to syndrome graphs . . . 32
  - 3.1.4 Network architectures . . . 34
- 3.2 Network training . . . 37
  - 3.2.1 Surrogate loss . . . 38
  - 3.2.2 Local search . . . 39

4 Results **43**
4.1 Training setup 43
4.1.1 Training with local search 44
4.1.2 Training with SGD 47
4.2 Decoding accuracy 51

5 Discussion **53**
5.1 Ethical considerations 53
5.2 Conclusion and future improvements 54

Bibliography **57**

List of Figures

2.1 The state $|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle$ visualised in the Bloch sphere. 4

2.2 A set of symbols used to represent gates and other operations in a quantum circuit. 7

2.3 A circuit used to entangle two qubits giving the Bell state $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. 7

2.4 The small circuit used to create a logical state $|\psi\rangle_L = \alpha|000\rangle + \beta|111\rangle$ in the repetition code. 9

2.5 An example of a repetition code circuit that protects against bit-flips. The first three qubits are entangled to give the logical state $|\psi\rangle_L = \alpha|0\rangle_L + \beta|1\rangle_L$, in which every qubit can be affected by single qubit errors e. The last two qubits are ancillas, responsible for the stabilizer measurements that are used to infer what errors have happened. After the stabilizer measurements have been used to infer an error, a potential correction operator can be applied. 11

2.6 (a) A planar surface code with two interleaved grids of data and measurement qubits are shown schematically in a. The data qubits are marked with open circles and the measurement qubits are marked with filled circles. The color of the squares indicates whether a measurement qubit is a measure-X or measure-Z qubit, meaning it measures either a stabilizer $X_aX_bX_cX_d = X_{abcd}$ or $Z_aZ_bZ_cZ_d = Z_{abcd}$. Green squares represent measure-X and blue squares measure-Z. The dashed red lines going across the grid indicate the logical operators Z_L and X_L. (b) The name ordering used to describe which data qubits a measurement qubit is coupled to. Note that the operations on data qubits are performed in lock-step, which means all a qubit operations are performed simultaneously across the whole surface code before all simultaneous b qubit operations, and so on [21]. 14

2.7 From the planar surface code, a central portion of qubits can be extracted and instead used to construct a rotated surface code. Additional weight-2 stabilizers are added along the boundaries. The blue squares represent Z stabilizers and the green ones X stabilizers. A rotated surface code is more spatially efficient and uses a factor of two less qubits for a given code distance. The logical operators X_L and Z_L acts on all qubits across a horizontal and vertical path respectively. 16

2.8 Two examples of error chains demonstrating the "lost" parity changes inside the error chain. 18

2.9 Examples of different errors with the same syndrome that require different correction operators. Z_L is a valid correction operators for (**a**), but applying Z_L to (**b**) results in a logical phase flip. 19
2.10 Examples of undetectable errors on the surface code. One causes a logical phase-flip while the other stabilizes the logical state. 19
2.11 Three cycles of stabilizer measurements shown together with the corresponding syndrome graph. The nodes are given by comparing the parity of the stabilizer measurements between consecutive rounds. To get the nodes corresponding to the first time step, it is assumed that we know the initial stabilizer configuration. Red nodes represent Z-stabilizers and yellow ones X-stabilizers. 20
2.12 A feed-forward network with an input layer for three inputs $\{x_0, x_1, x_2\}$, a series of hidden layers and an output layer for a single output y_0. . 23
2.13 A graph convolution updating node feature vector $\mathbf{x}_0$ based on its neighbouring node feature vectors $\mathbf{x}_j$ and the edge weights ϵ_{0j}. The function g is a non-linear activation function. 25

3.1 An overview of the neural matching decoder. The pre-processing step takes an error syndrome and creates a syndrome graph, which is fed to a GNN that updates the graph representation in a series of message passing layers. After the GNN, the syndrome graph is split in two subgraphs, one associated with nodes originating from X-stabilizers and one associated with nodes from Z-stabilizers. The subgraphs are processed in two parallel heads, giving a new set of edge weights for the original syndrome graph. Finally, the equivalence class of the error syndrome is predicted via two parallel matching modules utilizing a MWPM algorithm. 30
3.2 The matching module that processes syndrome graphs with updated edge weights by use of the MWPM algorithm. The edges belong to one of two classes, here shown as dashed or solid lines marked with $c \in \{0, 1\}$. By counting how many of the edges included in the perfect matching that represent an error chain extending to the boundaries of the surface code, the equivalence class can be predicted. 31
3.3 An example of how two edges can be created for any pair of nodes in the syndrome graph. The inside edge is drawn in turquoise and the outside edge is drawn in orange. 33
3.4 A schematic illustration of the GNN used in the NMD. The initial edge weights w_{ij} are used together with the class labels c_k to create a scalar edge feature embedding. This embedding is used to weight the sums in a series of graph convolutions, which are followed by ReLU-activations. 34
3.5 The two different heads that were explored shown together with the split of the original syndrome graph and subsequent edge embedding. The feed-forward head used a mean pooling operation to extend the edge feature vector, whilst the self-attention head used a simpler representation but added a skip connection and layer normalisation. . . . 36

4.1 Prediction performances on datasets with fixed error rate $p = 0.001$ for networks trained with classic accuracy as metric. Training performance shows classic accuracy for a dataset of 500'000 syndromes sampled from STIM, while validation performance shows logical accuracy for a dataset of 40000 syndromes. 44

4.2 Prediction performances on datasets with fixed error rate $p = 0.001$ for networks trained with balanced accuracy as metric. Training performance shows balanced accuracy for a dataset of 500'000 syndromes sampled from STIM, while validation performance shows logical accuracy for a dataset of 40'000 syndromes. 45

4.3 Prediction performances on datasets with mixed error rates $p = 0.001 - 0.005$ for networks trained with classic accuracy as metric. Training performance shows classic accuracy for a dataset of 500'000 syndromes sampled from STIM, while validation performance shows logical accuracy for a dataset of 40'000 syndromes. 46

4.4 Prediction performances on datasets with mixed error rates $p = 0.001 - 0.005$ for networks trained with balanced accuracy as metric. Training performance shows balanced accuracy for a dataset of 500000 syndromes sampled from STIM, while validation performance shows logical accuracy for a dataset of 40000 syndromes. 47

4.5 The class-wise percentage of correct and incorrect predictions when using classic accuracy or balanced accuracy as training metric. 47

4.6 Training loss as a function of training epoch for a fixed error rate (a) and a mixed error rate (b). 48

4.7 Logical accuracy as a function of training epoch when the networks were trained on simulations with a fixed error rate $p = 0.001$. The accuracy is computed for a separate validation set that consisted of the same error syndromes during the entire training process. 49

4.8 Logical accuracy as a function of training epoch when the networks were trained on simulations with a fixed error rate $p = 0.001 - 0.005$. The accuracy is computed for a separate validation set that consisted of the same error syndromes during the course of a training run. However, training with SGD was slow and the training had to be restarted before the three ATNx-networks reached 1000 epochs. Each restart initialise a new validation set, which could have a slightly different distribution of error syndromes. This likely explains the small jumps in logical accuracy seen for the ATNx-networks. 50

4.9 The logical failure rate seen as a function of various heads used in the NMD when evaluated on 10^7 syndrome graphs sampled from simulations with error rate $p = 0.001$. Different markers and colors are used to indicate whether the networks were trained with stochastic gradient descent (SGD) or local search (LS), and if a dataset with error syndromes sampled from simulations of fixed error rate $p = 0.001$ or a mixed error rate $p = 0.001-0.005$ was used during training. Also note a dashed line that indicates the logical accuracy of a MWPM-decoder (implemented in PyMatching [29]) and a dotted line giving the logical accuracy of the GNN-decoder developed in [38], when evaluated on the same dataset. 51

List of Tables

2.1 Possible stabilizer measurements for a repetition code with only single-qubit errors. 11

4.1 The different network architectures used in the NMD. FFx corresponds to heads having only dense layers, whilst ATNx represents networks with a self-attention layer. GCN stands for graph convolutional network and MLP stands for multi-layer perceptron. 43

1 Introduction

The concept of quantum computing traces its roots back to Feynman [20], who suggested that a computer based on the principles of quantum mechanics could offer several advantages over a classical computer. In contrast to classical computing that relies on binary bits to encode information, quantum computers use a quantum analogue known as qubits. Qubits benefit from quantum effects such as superposition and entanglement, which makes it possible to solve certain problems much more efficiently. By today, quantum computers have been realised experimentally and have introduced a completely new paradigm for computing [3]. Applications include Shor's prime factoring algorithm [47], optimization algorithms [19] and efficient simulations of quantum systems [13].

However, there are still several obstacles to overcome before a quantum computer can run the suggested algorithms for any problem of significant size. One of the primary challenges lies in the vulnerability of qubits, the fundamental units of a quantum computer. Qubits are very susceptible to noise and the information they hold can easily be corrupted. In order to reach a real quantum advantage, quantum computing must be made fault-tolerant by correcting for the physical errors affecting the qubits. This is known as quantum error correction [46, 23], a strategy that makes use of multiple physical qubits to encode logical qubits that are more robust toward errors.

One of the most promising approaches to quantum error correction is the surface code [21, 7], where qubits are arranged on a two-dimensional grid with nearest-neighbour connectivity. By having a portion of the qubits set aside for parity measurements, the state of the system can be inferred in a non-destructive way through consecutive measurement cycles. However, the measurement outcomes are often hard to interpret and degenerate with respect to the possible errors that could have caused them. An implementation of the surface code must therefore be paired with an algorithm that computes the most likely error given a set of parity measurements, a quantum error correction decoder. The decoder is essential for the survival of the logical qubit state and must run synchronised with the quantum computer, decoding parity measurements between the logical qubit operations. Logical operations are estimated to run at the order of 0.1-10 µs [21], and the decoder will have to run at the same rate to not slow down computations. The decoder must therefore be both fast and accurate if quantum error correction is to work.

There have been several suggestions of quantum error correction decoders for the

surface code, ranging from accurate but slow maximum-likelihood decoders [9] to faster but less accurate ones such as union-find [15]. A popular compromise is to construct decoders based on the graph algorithm minimum-weight perfect matching (MWPM), which has been shown to produce accurate decoders in several cases [16, 29, 30]. However, a disadvantage common to all these decoders is that their accuracy is dependent on a detailed description of the noise processes affecting the qubits. In a practical realisation of a quantum computer, the noise processes are likely to be complex and difficult to measure and model completely. An alternative is to use a data-driven approach and construct a decoder based on a neural network. The decoder is trained on a labelled dataset of parity measurements and learns the underlying noise model implicitly. A potential drawback is the requirement of a large dataset, but several works have solved this by using synthetic data generated in simulations. This has shown to produce highly accurate decoders, for instance the recurrent-neural network presented in [5] or the graph neural network shown in [38].

In this work, we present a novel decoder that combines a graph neural network with an MWPM algorithm. This allows us to limit the number of parameters in the neural network whilst retaining a relatively high accuracy. Importantly, it preserves the data-driven property, not requiring an a priori model for the errors. Both the computational size and a data-driven approach can be beneficial in future applications, where quantum error correction decoders will be implemented in hardware for quantum circuits without a known noise model. More research is needed to fully explore the potential of combining MWPM algorithms with neural networks, and hopefully this work can serve as a starting point for future endeavours.

1.1 Scope

The aim of this thesis is to develop a new type of quantum error correction decoder that combines a graph neural network and an MWPM algorithm to decode error syndromes from the surface code. Surface codes can be defined in different ways, but for the purpose of this report only the Calderbank-Shor-Steane rotated surface code with weight-4 X-stabilizers and weight-4 Z-stabilizers is considered [8, 50]. The surface codes are scalable in the number of physical qubits, but this work only treats codes of size $d = 5$ (having $d \times d = 25$ data qubits) that are measured for $d_t = 5$ cycles. Furthermore, all error syndromes processed by the decoder are sampled from simulated quantum circuits. To introduce errors, the noise model circuit-level noise has been used throughout all simulations. This is the noise model most relevant for fault-tolerant quantum computations, as it affects all qubits and gates used within the circuit [49].

2

Theory

This chapter provides a theoretical background to quantum computing, quantum error correction and machine learning. The emphasis is placed on areas most relevant for understanding the neural matching decoder.

2.1 Quantum computation

The fundamental building block of a quantum computer is the qubit, the quantum analogue of a bit in a classical computer. A qubit is a quantum system with two well-defined energy levels, represented in Dirac notation as the ground state $|0\rangle$ and the excited state $|1\rangle$. In vector notation, the states are written as

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

The states $|0\rangle$ and $|1\rangle$ form an orthonormal basis for the qubit, $\langle 0|1\rangle = \langle 1|0\rangle = 0$ and $\langle 0|0\rangle = \langle 1|1\rangle = 1$, which often is referred to as the z-basis, or the computational basis. Qubits can be realised using several different physical implementations, such as ion-traps or superconducting anharmonic oscillators (transmons) [10, 35].

The following section will introduce the mathematics used to describe the state of one or several qubits. Then follows an introduction to quantum logical gates, explaining how different operators act on qubits. After that, different noise processes and what errors they cause in quantum computers are discussed.

2.1.1 The qubit state

The qubit is a quantum system. As such, it is not restricted to being in either state $|0\rangle$ or state $|1\rangle$, but rather, it can be in a superposition of the ground state $|0\rangle$ and the excited state $|1\rangle$. A general qubit state $|\psi\rangle$ is described by

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle,$$

where α and β are complex numbers and $|\alpha|^2 + |\beta|^2 = 1$. If we exclude an global phase that does not have any physical meaning, the alternative parameterization

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

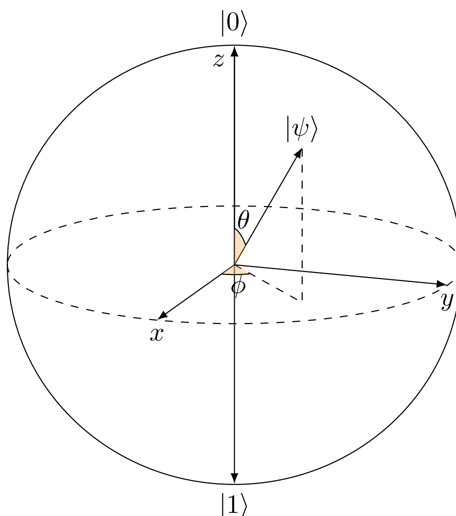


Figure 2.1: The state $|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle$ visualised in the Bloch sphere.

can be used to map qubit states to the Bloch sphere shown in Figure 2.1, via spherical coordinates (θ, ϕ).

For a quantum system of n qubits, the joint qubit state $|\Psi\rangle$ is described in a 2^n-dimensional Hilbert space as

$$|\Psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_{n-1}\rangle \otimes |\psi_n\rangle .$$

Both $|\psi\rangle$ and $|\Psi\rangle$ are known as pure states, because they can be described by single state vectors. However, if the state preparation is unknown, or if a system of one or more qubits is entangled with an external environment, the state cannot be described by a single state vector. Instead, the state must be described as an ensemble of pure states, a mixed state. The mixed state is often represented by the density operator ρ, a Hermitian matrix with $\mathrm{tr}(\rho) = 1$. All density operators can be decomposed in pure states $|\psi_i\rangle$,

$$\rho = \sum_i p_i |\psi_i\rangle \langle\psi_i| ,$$

but the decomposition is not unique and the interpretation of p_i can depend on the context. In the context of state preparation, p_i represents a classical probability. Consider for instance a state preparation where we only know that we will get states $|a\rangle$ with probability p_a and states $|b\rangle$ with probability p_b. The density matrix will then be given by

$$\rho = p_a |a\rangle \langle a| + p_b |b\rangle \langle b| ,$$

for the classical probabilities p_a and p_b. If the density matrix represents a pure state $|\psi\rangle$, we have $\rho = |\psi\rangle\langle\psi|$ which fulfils $\rho = \rho^2$ and $\mathrm{tr}(\rho^2) = 1$.

When representing single qubits, the density matrix is given by a Hermitian matrix $\rho \in \mathbb{C}^{2\times 2}$. Using the matrix representation of the Pauli operators

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

the density matrix can then be decomposed as

$$\rho = \frac{1}{2}\left(\mathbb{1} + r_x X + r_y Y + r_z Z\right).$$

The vector $\vec{r} = (r_x, r_y, r_z)$ is known as the Bloch vector and must have a length $|\vec{r}| \leq 1$. It can be be used to visualize the state represented by the density matrix on the Bloch sphere. Pure states are mapped to the surface of the Bloch sphere whilst mixed states are found in the interior.

2.1.2 Quantum logical gates

Quantum logical gates are the basic operations used to manipulate the qubits in a quantum computer. A gate that operates on n qubits is mathematically represented by a unitary operator $U \in \mathbb{C}^{2^n \times 2^n}, \ U^\dagger = U^{-1}$. Often gates act on one, two or three qubits. Before moving on to discussing some common gates, we present an important property for relations between quantum gates, *commutation*. If two quantum operators commute, the order in which they are applied to some arbitrary quantum state does not matter. For commuting operators A, B,

$$[A, B] = AB - BA = 0.$$

If applying two operators in different order swaps the signs in the quantum state, they are said to *anti-commute*. For anti-commuting operators A and B,

$$\{A, B\} = AB + BA = 0.$$

It can easily be proven that commuting operators must have the same eigenstates. The proof is left out, but the physical implication of this is that the order of measurements of commuting operators does not matter to the outcome. If two operators A and B commute, then there exists a set of basis states $|\lambda_i\rangle$ that are eigenstates of both A and B. Using this set of basis states, any state $|\psi\rangle$ can be written as

$$|\psi\rangle = \sum_i \alpha_i |\lambda_i\rangle.$$

Since any $|\lambda_i\rangle$ is an eigenstate of A and B, we get

$$A|\lambda_i\rangle = a_i|\lambda_i\rangle$$
$$B|\lambda_i\rangle = b_i|\lambda_i\rangle.$$

Let us say that a measurement of A on $|\psi\rangle$ gives us the result a_i. This happens with probability $|\alpha_i|^2$ (assuming no degeneracy, the proof for this case is left out). After being measured, $|\psi\rangle$ has collapsed into $|\lambda_i\rangle$, and measuring B on $|\lambda_i\rangle$ gives us the result b_i with probability 1. The overall probability of getting result b_i when measuring B after first measuring A is then $|\alpha_i|^2$, since it requires the result of the A measurement to be a_i. If we measure B first, we get the result b_i with probability $|\alpha_i|^2$, and the rest follows exactly as above. If A and B do not commute, they cannot both have a definite value at the same time. This is the basis for the Heisenberg uncertainty principle, but we refer to other works for the details [45].

The most common qubit gates include the Pauli operators $\sigma = \{X, Y, Z\}$, the controlled-X (CNOT or CX) and the Hadamard gate (H). The Pauli operators act on qubits as

$$
\begin{aligned}
X\left(\alpha\left|0\right\rangle + \beta\left|1\right\rangle\right) &= \alpha\left|1\right\rangle + \beta\left|0\right\rangle \\
Y\left(\alpha\left|0\right\rangle + \beta\left|1\right\rangle\right) &= i\left(\alpha\left|1\right\rangle - \beta\left|0\right\rangle\right) \\
Z\left(\alpha\left|0\right\rangle + \beta\left|1\right\rangle\right) &= \alpha\left|0\right\rangle - \beta\left|1\right\rangle
\end{aligned}
$$

and satisfy the commutation relations

$$
\begin{aligned}
[\sigma_i, \sigma_j] &= \sigma_i\sigma_j - \sigma_j\sigma_i = 2i\epsilon_{ijk}\sigma_k \\
\{\sigma_i, \sigma_j\} &= \sigma_i\sigma_j + \sigma_j\sigma_i = 2\delta_{ij}\mathbb{1},
\end{aligned}
$$

where ϵ_{ijk} is the Levi-Civita symbol

$$
\epsilon_{ijk} = \begin{cases} +1, & \text{if } (i,j,k) \text{ is } (x,y,z),\ (y,z,x) \text{ or } (z,x,y) \\ -1, & \text{if } (i,j,k) \text{ is } (x,z,y),\ (y,x,z) \text{ or } (z,y,x) \\ 0, & \text{if } i=j,\ j=k \text{ or } k=i \end{cases}
$$

and δ_{ij} is the Kronecker delta

$$
\delta_{ij} = \begin{cases} 1, & \text{if } i=j \\ 0, & \text{otherwise} \end{cases}.
$$

Due to their effects on the qubit state, X is referred to as a bit-flip, whilst Z is known as a phase-flip. The Y-gate can be seen as a combination of X and Z, $Y = iXZ$.

The Hadamard gate creates equal superpositions of the computational basis $\{\left|0\right\rangle, \left|1\right\rangle\}$. Its matrix representation is given by

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

and it operates as

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

The two states $\left|+\right\rangle$ and $\left|-\right\rangle$ are often referred to as the X-basis (or Hadamard basis), since $X\left|+\right\rangle = \left|+\right\rangle$ and $X\left|-\right\rangle = -\left|-\right\rangle$.

Controlled gates are applied on two qubits, with one qubit acting as the control and the second as a target. This can create entanglement, a necessary requirement for universal quantum computing [17]. One example of a controlled gate is controlled-X, often written as CX or CNOT. If the control qubit is in an excited state $\left|1\right\rangle$, it will act with X on the target qubit. For a control qubit in state $\left|0\right\rangle$, no gate will be applied to the target qubit.

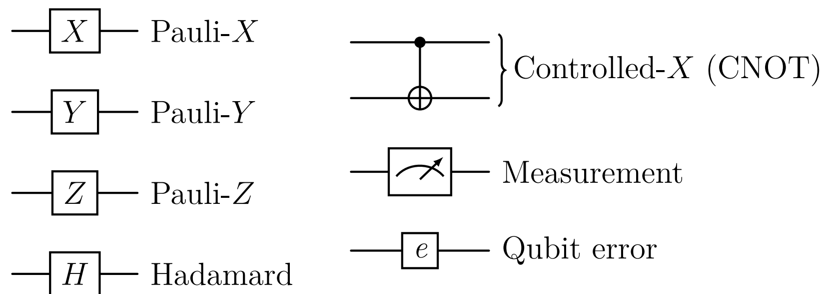


Figure 2.2: A set of symbols used to represent gates and other operations in a quantum circuit.

A common way to model quantum computation is to use a quantum circuit. In a quantum circuit, qubits are initialised to known states, gates are applied sequentially and measurements are performed to determine future actions or to do final readouts. A table showing some common gates and operations together with their circuit symbols is given in Figure 2.2. In Figure 2.3 a simple circuit is shown, where two qubits are entangled to create a so called Bell state. Using a Hadamard gate, the first qubit is put in an equal superposition, resulting in a state $\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$. The CNOT then flips the second qubit conditioned on the first one, resulting in the Bell state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. This state is entangled because it cannot be written as a product over the two qubits. When studying quantum circuits, a set of gates known

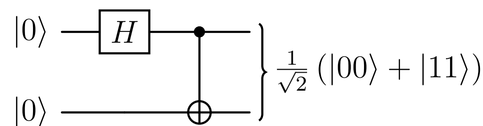


Figure 2.3: A circuit used to entangle two qubits giving the Bell state $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

as the Clifford gates is often encountered. The Clifford gates can be generated from multiplications of the Hadamard gate, the CNOT gate and the phase gate $S = \sqrt{Z}$. They are significant because of the Gottesman-Knill theorem, which states that a quantum circuit consisting only of Clifford gates can be simulated efficiently on a classical computer. The Clifford gates normalise the n-qubit Pauli group by conjugation, which means that a Clifford gate C fulfils $C\sigma_i C^\dagger = \sigma_j$ for $\sigma_i, \sigma_j \in \{X, Y, Z\}$. Since the product of two Pauli operators is either the identity or another Pauli operator, they belong to the Clifford group trivially [24].

2.1.3 Noise and errors

In an ideal quantum computer free from noise, qubits would always be prepared in the desired states and only evolve to new states when acted upon by quantum logical gates. For a qubit visualised in the Bloch sphere, this would correspond to deterministically rotating the Bloch vector to new precise locations exactly when a

gate is applied. In reality, qubits are highly susceptible to noise and their states are constantly influenced by the external environment. This means that a qubit will not remain in any given state for long, we get *qubit decoherence*. Qubit decoherence mainly follows from relaxation and dephasing. Relaxation is when the excited state $|1\rangle$ returns to the ground state $|0\rangle$, and dephasing means that the phase information between $|0\rangle$ and $|1\rangle$ is corrupted or lost. In the Bloch sphere, relaxation errors can be thought of as happening in the θ-axis whilst dephasing happens in the ϕ-axis [37].

There are several ways to describe the effects of noise in quantum systems, but the general idea is to model an error channel $\epsilon(\rho)$ describing how the density operator ρ transforms in the presence of noise. One approach is to construct an error channel $\epsilon(\rho)$ with Pauli operators describing the errors acting on the system,

$$\begin{aligned}\epsilon(\rho) = &(\mathbb{1} - p_X - p_Y - p_Z)\,\rho \\ &+ p_X X\rho X + p_Y Y\rho Y + p_Z Z\rho Z,\end{aligned}$$

which happens with probabilities p_i. If $p_i = p$, we call the channel a *depolarizing channel*. The Pauli channel cannot describe arbitrary qubit errors, but it is often used as a simple error model in discussions related to quantum error correction [43, 25].

2.2 Quantum error correction

To mitigate the effects of noise on a quantum computer, the information stored in the qubit states can be protected by the means of quantum error correction (QEC). The existence of feasible QEC schemes was first shown in [46], and since then much research has been devoted to the area. QEC can be implemented in various ways, but all must encode quantum information in a way that allows for the detection and correction of errors without compromising the information itself. Since a quantum state collapses upon measurement, direct measurement of the qubits used in computations is not possible. The usual solution to this problem is to use stabilizer codes, a prominent group of QEC codes that rely on a set of commutative parity measurements. The measurements project a n-qubit state to a lower dimensional *code space* representing one or more logical qubits. The parity measurements are done continuously, and an error is detected whenever the state leaves the code space [23].

In this section, QEC will be introduced by first explaining how the *repetition code* works. This will be followed by a more formal description of stabilizer formalism, the language we often use to describe QEC codes. Then follows a section on the surface code, one of the most promising QEC codes to date. In relation to the surface code, error syndromes and correction operators will also be discussed.

2.2.1 Repetition code

As an introduction to quantum error correction, let us consider the simple repetition code. The aim is to create a QEC code protecting a state against bit-flips. Instead

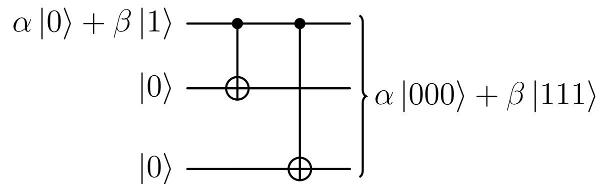


Figure 2.4: The small circuit used to create a logical state $|\psi\rangle_L = \alpha|000\rangle + \beta|111\rangle$ in the repetition code.

of using single qubits to represent the states $|0\rangle$ and $|1\rangle$, we use three qubits and define *logical states* $|0\rangle_L$ and $|1\rangle_L$ as

$$\begin{aligned}|0\rangle_L &= |0\rangle \otimes |0\rangle \otimes |0\rangle = |000\rangle \\ |1\rangle_L &= |1\rangle \otimes |1\rangle \otimes |1\rangle = |111\rangle ,\end{aligned}$$

which means that we can encode an arbitrary logical qubit state by the superposition

$$|\psi\rangle_L = \alpha|0\rangle_L + \beta|1\rangle_L .$$

The space spanned by $|0\rangle_L$ and $|1\rangle_L$ is defined as the code space. To obey the no-cloning theorem [42], which states that it is impossible to copy a quantum state, the logical state is prepared using the entanglement circuit seen in Figure 2.4. Having encoded a logical state $|\psi\rangle_L$, the three entangled qubits can be treated as one logical qubit. This means that we need logical counterparts to our usual qubit operators. To do a bit-flip, we want an operator that performs the flip $|0\rangle_L \to |1\rangle_L$ and vice versa. This is satisfied by $X_L = X \otimes X \otimes X = X_1X_2X_3$. The repetition code only protects against bit-flips, and there is no need for a special construct to create a logical counterpart to the operator Z (Y can always be decomposed into X and Z, so it needs no special treatment either). Acting with Z on any qubit in the logical state will yield the expected outcome, $Z_i|0\rangle_L = |0\rangle_L$ and $Z_i|1\rangle_L = -|1\rangle_L$ $i \in \{1,2,3\}$. Note that the logical operator X_L still fulfils the expected commutation relations with Z_i,

$$\begin{aligned}\{X_L, Z_i\} &= \{X_1X_2X_3, Z_i\} \\ &= X_1X_2X_3Z_i + Z_iX_1X_2X_3 \\ &= X_1X_2X_3Z_i - X_1X_2X_3Z_i \\ &= 0 \\ [X_L, Z_i] &= [X_1X_2X_3, Z_i] \\ &= X_1X_2X_3Z_i - Z_iX_1X_2X_3 \\ &= 2X_1X_2X_3Z_i \\ &= -2iY_L.\end{aligned}$$

The last equality can be understood by the action of Y on a single qubit state, $Y|0\rangle = i|1\rangle$ and $Y|1\rangle = -i|0\rangle$. A logical Y should yield the same result on a logical state, a combined bit-flip and phase change.

Now, let us focus on how we can detect bit-flip errors in the state $|\psi\rangle_L$. Depending on which qubit is flipped, the initial state $|\psi\rangle_L$ will turn into one of the corrupted states

$$\begin{aligned}|\psi\rangle_c &= \alpha\,|100\rangle + \beta\,|011\rangle \\ |\psi\rangle_c &= \alpha\,|010\rangle + \beta\,|101\rangle \\ |\psi\rangle_c &= \alpha\,|001\rangle + \beta\,|110\rangle\,.\end{aligned}$$

In a classical repetition code, the erroneous bit could have been detected by directly looking at the sequence. However, if a qubit is measured the quantum state will collapse, destroying the encoded information. For example, consider what would happen if we were to measure the state of the logical qubit $|\psi\rangle_L$ directly. We do this by projective measurements, using projectors

$$\begin{aligned}P_0 &= |000\rangle\,\langle 000| \\ P_1 &= |111\rangle\,\langle 111|\,.\end{aligned}$$

Assuming our logical qubit has not been subjected to errors, we get

$$\begin{aligned}P_0\,|\psi\rangle_L &= |000\rangle\,\langle 000|\,(\alpha\,|000\rangle + \beta\,|111\rangle) = \alpha\,|000\rangle = \alpha\,|0\rangle_L \\ P_1\,|\psi\rangle_L &= |111\rangle\,\langle 111|\,(\alpha\,|000\rangle + \beta\,|111\rangle) = \beta\,|111\rangle = \beta\,|1\rangle_L\,,\end{aligned}$$

which means that we will measure $|0\rangle_L$ with probability $|\alpha|^2$ and $|1\rangle_L$ with $|\beta|^2$. However, the projective measurement have collapsed the logical state and the information encoded in the superposition is destroyed.

To get around this problem, we can do parity measurements that do not measure the state of a single qubit directly. The repetition code makes use of two parity measurements, $Z \otimes Z \otimes \mathbb{1} = Z_1 Z_2$ and $\mathbb{1} \otimes Z \otimes Z = Z_2 Z_3$. The states $|0\rangle$ and $|1\rangle$ are eigenstates of the Pauli Z-operator with eigenvalues ± 1, which means that measuring $Z_1 Z_2$ and $Z_2 Z_3$ will yield $+1$ if both qubits are the same and -1 if they are not. Thus, $Z_1 Z_2$ checks for a bit-flip in qubit 1 or 2 whilst $Z_2 Z_3$ checks for a bit-flip in qubit 2 or 3. The stabilizer measurements are done with the help of two ancillary qubits that are measured in the $\{|0\rangle\,, |1\rangle\}$-basis. How this is done in a circuit is shown in Figure 2.5, where the small entanglement circuit from Figure 2.4 is extended to include qubit errors e and the stabilizer measurements used to infer the errors. For single qubit bit-flips, the possible errors and the stabilizer measurements are summarized in Table 2.1. The set of stabilizer measurements are known as the *error syndrome*. In the case of the repetition code, the error syndromes are unique given that only one qubit at a time is affected by an error. If two errors act on the system, for example $X_1 X_2\,|\psi\rangle_L = \alpha\,|110\rangle + \beta\,|001\rangle$, the error cannot be uniquely identified. Measuring $Z_1 Z_2$ would yield $+1$ whilst $Z_2 Z_3$ yields -1. There is no way to distinguish this error from that of a single bit-flip on the third qubit, X_3. Furthermore, a phase error such as $|\psi\rangle_L \longrightarrow \alpha\,|000\rangle - \beta\,|111\rangle$ would also go unnoticed. To protect against an arbitrary error, a more sophisticated code is needed.

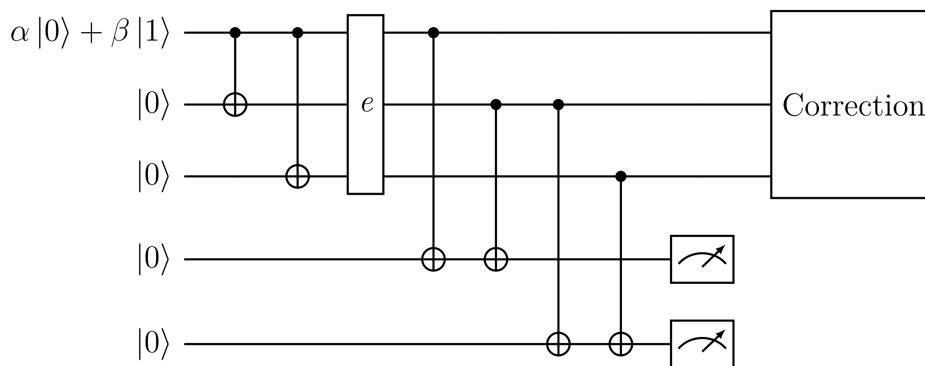


Figure 2.5: An example of a repetition code circuit that protects against bit-flips. The first three qubits are entangled to give the logical state $|\psi\rangle_L = \alpha |0\rangle_L + \beta |1\rangle_L$, in which every qubit can be affected by single qubit errors e. The last two qubits are ancillas, responsible for the stabilizer measurements that are used to infer what errors have happened. After the stabilizer measurements have been used to infer an error, a potential correction operator can be applied.

Operation	Z_1Z_2	Z_2Z_3
$\mathbb{1}$	+1	+1
X_1	-1	+1
X_2	-1	-1
X_3	+1	-1

Table 2.1: Possible stabilizer measurements for a repetition code with only single-qubit errors.

2.2.2 Stabilizer formalism

By introducing the concepts of stabilizers, QEC codes can be created in a more general way. Let us begin with some important definitions. Given an operator U and a state $|\psi\rangle$, we say that the state $|\psi\rangle$ is *stabilized* by U if U leaves the state unchanged, $U|\psi\rangle = |\psi\rangle$. For instance, the Bell state

$$|\psi\rangle = \frac{1}{\sqrt{2}} \left(|00\rangle + |11\rangle \right),$$

is stabilized by the the operators X_1X_2 and Z_1Z_2 since

$$X_1X_2 |\psi\rangle = |\psi\rangle$$
$$Z_1Z_2 |\psi\rangle = |\psi\rangle .$$

To formalise our language, consider the Pauli group for single qubits,

$$G_1 = \{\pm\mathbb{1}, \pm i\mathbb{1}, \pm X, \pm iX, \pm Y, \pm iY, \pm Z, \pm iZ\} = \langle X, Y, Z\rangle$$

The Pauli group is multiplicatively generated by the Pauli X, Y and Z operators, and the factors ± 1 and $\pm i$ are needed to ensure the group stays closed under group

multiplication. Generalising to higher dimensions, G_n denotes the Pauli group on n qubits and consists of all n-fold tensor products of Pauli matrices. Assuming S a subgroup of G_n, define V_s as the set of 2^n-dimensional qubit states fixed by every element of S. For this to be possible for a non-trivial V_s, S must be an abelian subgroup (all operators in S must commute and thus share common eigenstates), and $-\mathbb{1} \notin S$. V_s is called the *vector space stabilized* by S and S is the *stabilizer group* of V_s. To illustrate, consider $n = 3$ qubits and the group $S = \{\mathbb{1}, X_1X_2, X_2X_3, X_1X_3\}$. X_1X_2 fixes the set

$$v_1 = \{|{+}{+}{+}\rangle, |{+}{+}{-}\rangle, |{-}{-}{+}\rangle, |{-}{-}{-}\rangle\},$$

X_2X_3 fixes

$$v_2 = \{|{+}{+}{+}\rangle, |{+}{-}{-}\rangle, |{-}{+}{+}\rangle, |{-}{-}{-}\rangle\}$$

and X_1X_3 fixes

$$v_3 = \{|{+}{+}{+}\rangle, |{-}{+}{-}\rangle, |{+}{-}{+}\rangle, |{-}{-}{-}\rangle\}.$$

Since V_s is defined as the set fixed by every element of S, V_s must correspond to the subspace spanned by $|{+}{+}{+}\rangle$ and $|{-}{-}{-}\rangle$, as these are common to all three operators. To save some work, we could have described the group S by its *generators* instead. A list of elements $g_1 \dots g_l$ generates a group S if every element in S can be written as a product of elements in the list $g_1 \dots g_l$. We write this as $S = \langle g_1, \dots, g_l\rangle$. For $S = \{\mathbb{1}, X_1X_2, X_2X_3, X_1X_3\}$,

$$\begin{aligned}(X_1X_2)(X_1X_2) &= (X_1X_2)(X_2X_1) = \mathbb{1}^2 = \mathbb{1} \\ (X_1X_2)(X_2X_3) &= X_1X_3,\end{aligned}$$

and we could instead have written $S = \langle X_1X_2, X_2X_3\rangle$. Since stabilizers leave the state unchanged and form an abelian group, any product of elements in S must also stabilize the state. To check if a state is stabilized by S, it is therefore enough to check that the state is stabilized by the generators of S.

Using the stabilizer formalism, a $[[n, k]]$ *stabilizer code* is defined as the vector space V_S stabilized by a subgroup S of G_n such that $-\mathbb{1} \notin S$. For n qubits encoding k logical qubits, the stabilizers S have $n - k$ independent and commuting generators, $S = \langle g_1, \dots, g_{n-k}\rangle$ [43]. The vector space V_s is synonymous with the term *code space* introduced in Section 2.2.1 and its basis consists of the joint +1-eigenspace of the stabilizers in S. The basis states are referred to as *logical states*.

To compare QEC codes, the term *code distance* is often used. Code distance refers to the minimum-weight Pauli operator needed to form a logical operator. The weight t of a Pauli operator is defined as the number of qubits it acts on non-trivially. As an example, consider the repetition code discussed in 2.2.1. The logical operator Z_L can be given by any $Z_L = Z_i, i \in \{1, 2, 3\}$, resulting in an operator with weight $t = 1$. Thus, the repetition code has a code distance $d = 1$. The code distance d is sometimes included in the description of a code, using the notation $[[n, k, d]]$ with n and k as before.

2.2.3 Surface codes

One of the most promising approaches to fault-tolerant quantum computing is the surface codes. Surface codes evolved from the *toric codes* introduced by Kitaev [34] and constitute a family of stabilizer codes defined on a two-dimensional lattice of qubits. They only require nearest-neighbour connectivity and have a comparatively high tolerance towards errors. This makes surface codes a viable option for fault-tolerant quantum computing. In 2022, Google Quantum AI used the surface code to make the first experimental demonstration of how QEC performance improves with an increasing number of qubits [1].

In a surface code, the qubits are arranged in two interleaved grids consisting of qubits responsible for information encoding, *data qubits*, and qubits used for stabilizer measurements, *measurement qubits* (also known as *ancillary qubits* or *ancillas*). The qubits have nearest-neighbour connectivity, with four neighbours for every qubit except the ones along the boundary. The measurement qubits are divided into two types, "measure-X" and "measure-Z" qubits, distributed in a checkerboard pattern throughout the code. Measure-X qubits are used to force neighbouring data qubits into eigenstates of the operator $X_aX_bX_cX_d = X_{abcd}$, whilst the measure-Z qubits force neighbouring data qubits into eigenstates of the operator $Z_aZ_bZ_cZ_d = Z_{abcd}$. The measurement qubits therefore measure X or Z stabilizers. To avoid possible confusion we here clarify that measure-X qubits measure X-stabilizers, and therefore detect Z-errors (phase-flips), while measure-Z qubits measure Z-stabilizers that detect X-errors (bit-flips).

A schematic overview of a planar surface code is given in Figure 2.6a, with open circles representing data qubits and filled circles showing measurement qubits. Green squares indicate measure-X qubits and blue ones measure-Z qubits. The name ordering used for the stabilizers X_{abcd} and Z_{abcd} is shown in Figure 2.6b.

To operate a surface code, we begin by initialising all measurement qubits in the ground state $|0\rangle$. The data qubits can be initialised in a random configuration as long as they remain in the computational basis $\{|0\rangle, |1\rangle\}$. Then follows a cycle of concurrent stabilizer measurements, projecting the qubits to a simultaneous eigenstate of all Z_{abcd} and X_{abcd} stabilizers. This gives a ± 1 configuration of eigenvalues that define the *quiescent state*. After having performed the first series of stabilizer measurements, the aim of our surface code is to detect any error affecting the quiescent state so they can be analyzed and, if necessary, corrected. If all X_{abcd} and Z_{abcd} stabilizers commute with each other, we can measure them again and again without changing the outcome of the measurements, given that no errors occur. It is trivial to see that X_{abcd} and Z_{abcd} commute for all stabilizers not sharing any qubits, since operators acting on different qubits always commute. However, all measure-X stabilizers will also share two data qubits with a neighbouring measure-Z qubit and vice versa. Assuming qubits a and b are shared, we have the commutator

$$\begin{aligned}
&[X_aX_bX_cX_d, Z_aZ_bZ_eZ_f] = \\
&\quad X_aX_bX_cX_dZ_aZ_bZ_eZ_f - Z_aZ_bZ_eZ_fX_aX_bX_cX_d = \\
&\quad X_aX_bX_cX_dZ_aZ_bZ_eZ_f - Z_aX_aZ_bX_bX_cX_dZ_eZ_f.
\end{aligned}$$

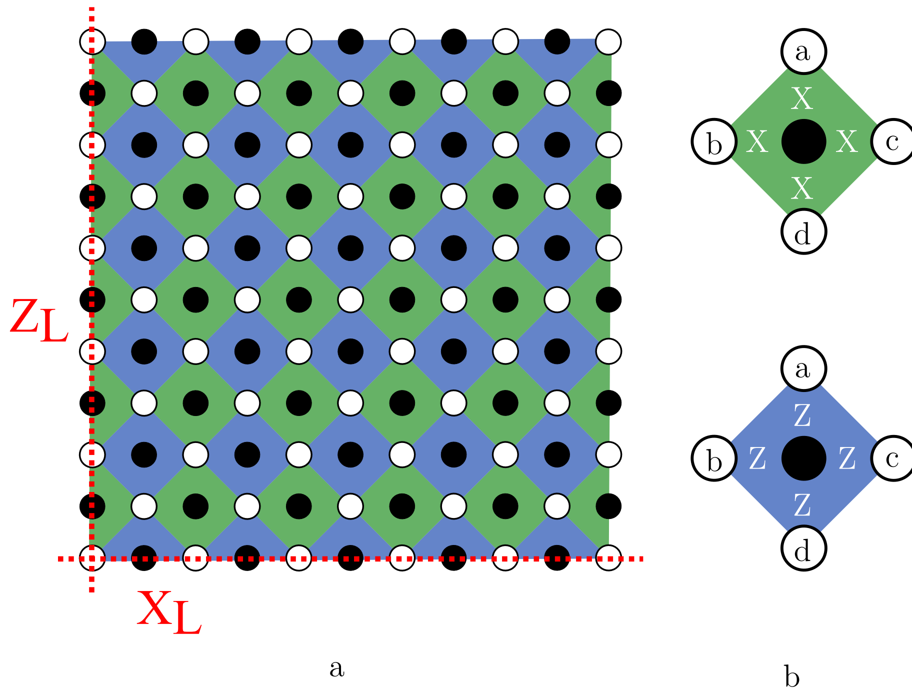


Figure 2.6: (a) A planar surface code with two interleaved grids of data and measurement qubits are shown schematically in *a*. The data qubits are marked with open circles and the measurement qubits are marked with filled circles. The color of the squares indicates whether a measurement qubit is a measure-X or measure-Z qubit, meaning it measures either a stabilizer $X_aX_bX_cX_d = X_{abcd}$ or $Z_aZ_bZ_cZ_d = Z_{abcd}$. Green squares represent measure-X and blue squares measure-Z. The dashed red lines going across the grid indicate the logical operators Z_L and X_L. (b) The name ordering used to describe which data qubits a measurement qubit is coupled to. Note that the operations on data qubits are performed in lock-step, which means all a qubit operations are performed simultaneously across the whole surface code before all simultaneous b qubit operations, and so on [21].

Pauli operators acting on the same qubit satisfy $[\sigma_i, \sigma_j] = 2\delta_{ij}\mathbb{1}$, which means that commuting Z_a through X_a introduces a minus sign. However, since the same is true for Z_b and X_b, we can rearrange to get

$$X_a X_b X_c X_d Z_a Z_b Z_e Z_f - X_a X_b X_c X_d Z_a Z_b Z_e Z_f = 0,$$

showing that all stabilizers in the surface code commute [21]. The general idea of QEC codes is to use several physical qubits to encode a fewer number of logical qubits. The repetition code discussed in 2.2.1 uses three physical qubits to encode one logical qubit. To understand how many logical qubits the surface code represents, consider the relation between the Hilbert space of the data qubits and the constraints enforced by the measurement qubits. Assume a surface code defined on a square, with $D = d^2 + (d-1)^2 = 2d^2 - 2d + 1$ data qubits and $M = 2d^2 - 2d$ measurement qubits. The Hilbert space of the data qubits has 2^D dimensions, but when the surface code is operated the stabilizers project the qubits into a quiescent state. The quiescent state corresponds to a specific configuration of binary measurements across the grid and uniquely identifies the logical subspace. Each measurement qubit therefore effectively halves the number of dimensions in the Hilbert space, which leaves

$$\mathcal{N} = \frac{2^D}{2^M} = 2^{D-M} = 2^{2d^2-2d+1-2d^2-2d} = 2$$

degrees of freedom among the data qubits. The surface code therefore encodes exactly one logical qubit with general state

$$|\psi\rangle_L = \alpha\,|0\rangle_L + \beta\,|1\rangle_L\,. \tag{2.1}$$

For the surface code to be useful, it must be possible to change the logical state of the encoded qubit by logical Pauli operators. The logical operator must commute with all stabilizers but also act non-trivially on the quiescent state $|\psi\rangle_q$. Given the logical states $|0\rangle_L$ and $|1\rangle_L$, we want the logical bit-flip operator to act as

$$\begin{aligned} X_L\,|0\rangle_L &= |1\rangle_L \\ X_L\,|1\rangle_L &= |0\rangle_L \end{aligned} \tag{2.2}$$

and the logical phase-flip operator to act as

$$\begin{aligned} Z_L\,|0\rangle_L &= |0\rangle_L \\ Z_L\,|1\rangle_L &= -\,|1\rangle_L \end{aligned}\,. \tag{2.3}$$

For simplicity, assume that all data qubits are initialised in $|0\rangle$. All Z stabilizer measurements will then yield the eigenvalue $+1$. Saying that our logical operator needs to commute with all stabilizers is equivalent to requiring that the stabilizer measurements continue to yield $+1$. The only way to do this whilst altering the quiescent state non-trivially is to flip an even number of data qubits for every stabilizer. If only one out of the four data qubits coupled to a measurement qubit is flipped, the stabilizer will measure -1 and project the surface code into another quiescent state $|\psi\rangle_q'$. To always flip qubits in pairs, a logical bit-flip X_L on the surface code must connect the left boundary to the right one via bit-flip operators X. The logical Z_L is

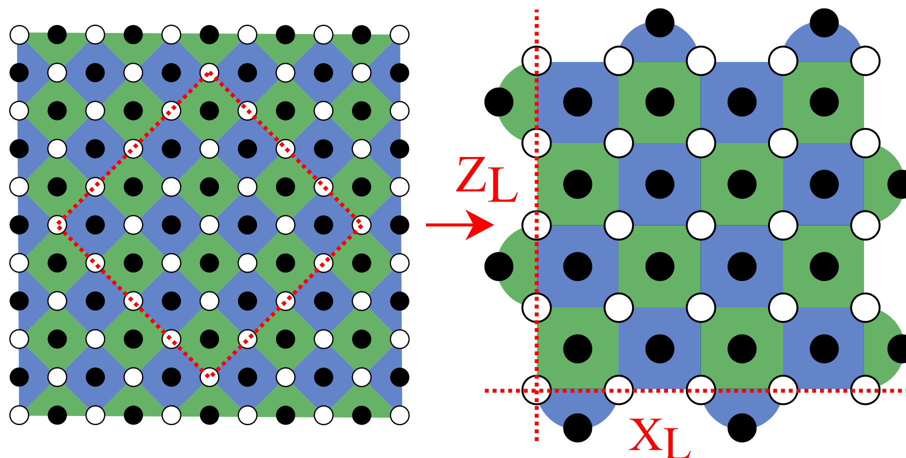


Figure 2.7: From the planar surface code, a central portion of qubits can be extracted and instead used to construct a rotated surface code. Additional weight-2 stabilizers are added along the boundaries. The blue squares represent Z stabilizers and the green ones X stabilizers. A rotated surface code is more spatially efficient and uses a factor of two less qubits for a given code distance. The logical operators X_L and Z_L acts on all qubits across a horizontal and vertical path respectively.

defined based on the same arguments but must instead connect the upper boundary with the lower one using Z operators on the data qubits. The logical operators shown in Figure 2.7 are not the only possible ones. Any logical operator that can be formed from another by applying a stabilizer is simply a different representation of the same operator. On a surface code with d data qubits along any boundary, a logical operator needs at least d operators acting on the state. This gives the surface code a code distance d [21].

In Section 2.2.1 we discussed the repetition code and defined $|0\rangle_L = |000\rangle$ and $|1\rangle_L = |111\rangle$. For the surface code, the definition of $|0\rangle_L$ and $|1\rangle_L$ is more complex. $|0\rangle_L$ now denotes a superposition of states that both satisfy the ± 1 configuration from our stabilizer measurements, and are stabilized by our Z_L operator (i.e. have a $+1$ eigenvalue for Z_L). Naturally, $|1\rangle_L = X_L |0\rangle_L$, which yields a state that both satisfies the ± 1 configuration from our stabilizer measurements, and has eigenvalue -1 for Z_L. As with the previously defined *code space*, this is equivalent to saying that the quiescent state is spanned by $|0\rangle_L$ and $|1\rangle_L$.

A common improvement on the planar surface code shown in 2.6a is to extract a central portion of qubits and rotate them, giving the rotated surface code as seen in Figure 2.7. Additional stabilizers only acting on pairs of data qubits are added along the boundaries, but the logical operators can still be defined as strings of operators acting on each data qubit across the grid. For a given code distance, this reduces the number of physical qubits needed by a factor of two compared to the planar surface code. This makes the rotated surface code the preferred option for experimental realizations.

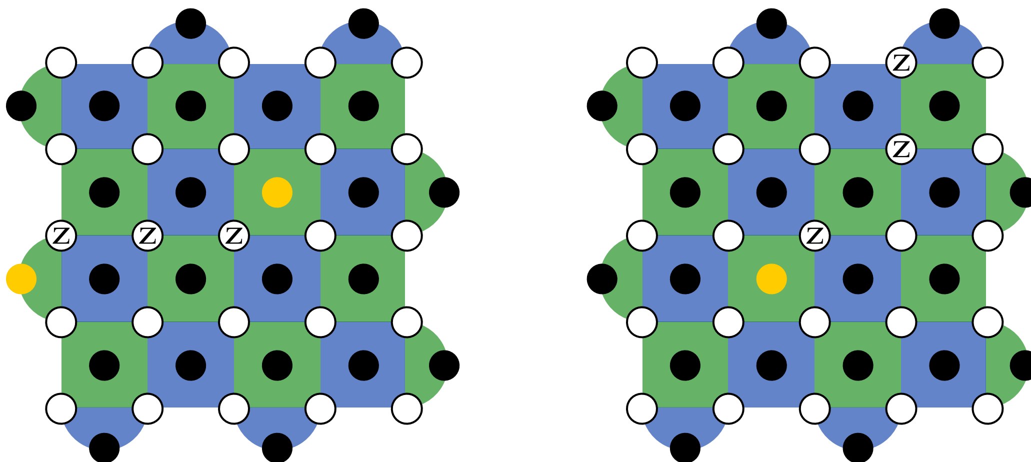
2.2.4 Error syndromes and correction operators

The aim of the surface code is to protect the state of the logical qubit by detecting the errors affecting it. In Section 2.2.1 the concept of an error syndrome was introduced, referring to a specific combination of parity measurement outcomes that could be used to infer if a bit-flip in the repetition code happens. For the surface code, an error syndrome is constructed by measuring all stabilizers simultaneously in repeating cycles.

During initialisation, the surface code is projected into a quiescent state corresponding to a specific configuration of ± 1 stabilizer measurements. If no errors occur, subsequent stabilizer measurements will yield the same ± 1 configuration, and the physical qubits remain in the same quiescent state. However, if a bit-flip or a phase-flip occurs, the erroneous qubit will cause a parity change in the neighbouring stabilizers (assuming no other errors) and the initial ± 1-configuration will no longer be intact. We call this a positive *detector measurement*. By continuously performing stabilizer measurements across the surface code, we can track how errors change stabilizer parities between measurement rounds. These positive detector measurements will then collectively be referred to as the error syndrome.

When only considering single qubit errors, or errors on multiple qubits that do not share any stabilizers, identifying errors from error syndromes is trivial. However, if the same type of Pauli error occurs on two or more adjacent qubits, the parity of their shared stabilizer is flipped twice, corresponding to an *error chain*. The only stabilizers that show a parity change are the ones at the boundaries of the error chain. Two different examples of error chains are demonstrated in Figure 2.8. *Undetectable errors* occur if the error chain represents a logical operator, or if the error is a stabilizer on the surface code. The second case clearly does not present a problem as it leaves the logical state unchanged (shown in Figure 2.10b), but the first case (shown in Figure 2.10a), is a type of error that cannot be handled by stabilizer measurements, a *logical error*. Up to a threshold error rate p_{th} on the physical qubits, scaling up the size of the surface code can help protect against logical errors [21].

Any error syndrome could be caused by multiple different qubit errors, and require different correction operators. Up until now, we have not distinguished between errors on different physical qubits in the surface code. However, we have now introduced the concepts necessary to make this distinction. Not every qubit error that changes the quiescent state, i.e. produces a parity change in a stabilizer measurement, lead to a change in the logical state [21]. For example, say we are performing a Z_L-measurement on the logical qubit. By definition, any error operator on the original quiescent state that commutes with Z_L will not have an impact on the measurement result. Since any operation on different qubits commute, we can conclude that any Pauli error on a qubit that is not acted on by Z_L will commute with Z_L. Any Z-error on the Z_L qubits will also commute with Z_L. Additionally, X-errors on an even number of Z_L qubits will commute with Z_L. The only Pauli errors on the quiescent state that do not commute with Z_L are X-errors on an odd number of Z_L qubits. These errors will affect Z_L measurements, but the effect can be coun-



(a) Z error chain where both error chain boundaries show a parity change

(b) Z error chain ending at qubit with only one X stabilizer, resulting in only one detector measurement

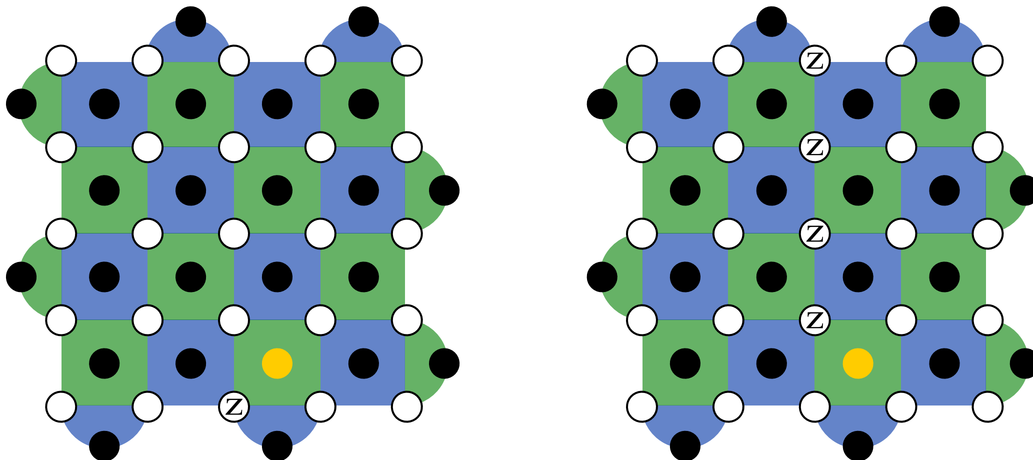
Figure 2.8: Two examples of error chains demonstrating the "lost" parity changes inside the error chain.

tered by applying X_L as a correction operator. The same argument can be made to say that the only Pauli errors on the quiescent state that do not commute with X_L are Z-errors on an odd number of X_L qubits. The errors in this case will affect X_L measurements, and can be countered by applying Z_L as a correction operator. To simplify discussion of different types of errors we introduce the term *equivalence class*. The four different equivalence classes are defined as follows:

- **Class I:** Errors that commute with both X_L and Z_L. These errors do not require a logical operator to be corrected (in fact, applying one will result in a logical error).
- **Class X:** Errors that commute with X_L, but not Z_L. These errors are corrected by applying X_L.
- **Class Z:** Errors that commute with Z_L, but not X_L. These errors are corrected by applying Z_L
- **Class Y:** Errors that do not commute with either X_L or Z_L. These errors are corrected by applying both X_L and Z_L.

2.3 Decoding an error syndrome

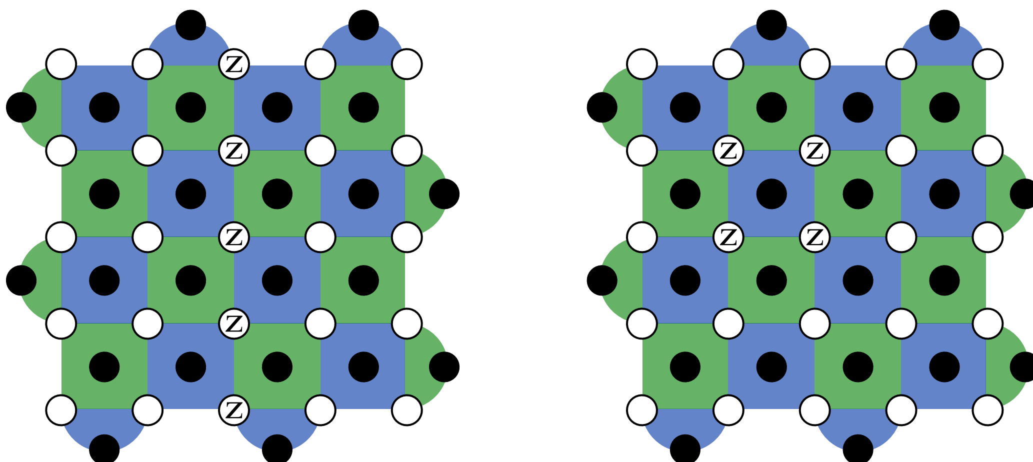
A stabilizer code is useful only if we can deduce what kind of error a syndrome represents. As discussed in the section above, syndromes can be degenerate and often correspond to several possible qubit errors. For errors in class X, Y and Z, logical operators are valid correction operators, while for errors in class I, logical operators will cause a logical error. The syndromes for errors in different equivalence classes can look identical, so given a syndrome, we must therefore find the most likely



(a) A syndrome caused by an error belonging to class Z.

(b) The same syndrome as in (**a**) that is caused by an error belonging to class I.

Figure 2.9: Examples of different errors with the same syndrome that require different correction operators. Z_L is a valid correction operators for (**a**), but applying Z_L to (**b**) results in a logical phase flip.



(a) Logical phase-flip error.

(b) Error operator is a stabilizer.

Figure 2.10: Examples of undetectable errors on the surface code. One causes a logical phase-flip while the other stabilizes the logical state.

equivalence class.

The algorithm responsible for processing the error syndrome is typically referred to as a *decoder*. For surface codes, the standard algorithm is the minimum-weight perfect matching (MWPM) decoder [30], but several other classical algorithms such as union-find [15] and belief-matching [31] have also been suggested. There are also optimal maximum-likelihood decoders that always yield the most likely equivalence class, but they are not scalable and requires abundant computational resources even at moderate code sizes [16, 9, 26]. An alternative approach that recently has shown great success is the use of neural networks. Two examples are the transformer-based recurrent neural network suggested in [5] and the graph neural network presented

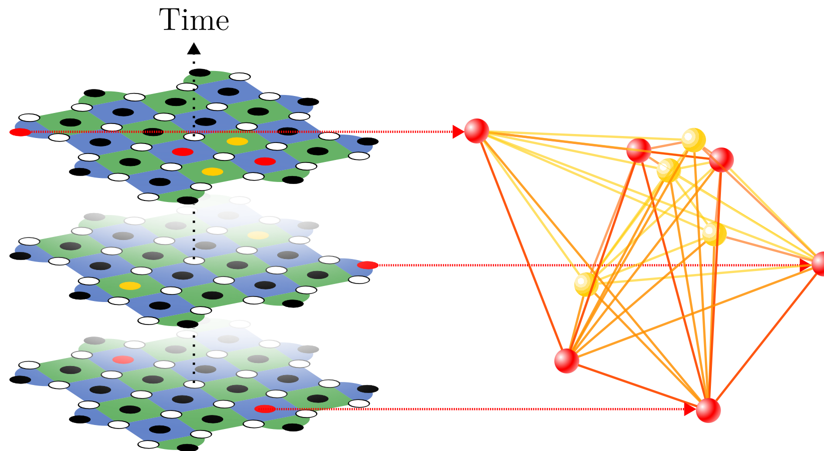


Figure 2.11: Three cycles of stabilizer measurements shown together with the corresponding syndrome graph. The nodes are given by comparing the parity of the stabilizer measurements between consecutive rounds. To get the nodes corresponding to the first time step, it is assumed that we know the initial stabilizer configuration. Red nodes represent Z-stabilizers and yellow ones X-stabilizers.

in [38].

This section will describe how an error syndrome can be mapped to a different mathematical object, a graph. A graph representation is used in several decoders, including the MWPM decoder, belief-matching and the graph neural network decoder referred to above. After having introduced the graph formalism, a section explaining how MWPM decoders work follows.

2.3.1 Syndrome graphs

Before discussing how the MWPM-decoder works, we must understand how error syndromes can be translated to mathematical objects that can be interpreted by an algorithm. There is no singular mathematical representation used by all decoders, but the MWPM-decoder and several others use *graphs*. Let us first define what a graph is, and then discuss how it can be used to represent an error syndrome.

A graph is a pair $G = (V, E)$, where V represents a set of vertices and E gives a set of vertex pairs $\{v_i, v_j\}$. The vertices are also known as *nodes*. The pairs $\{v_i, v_j\}$ each represent an edge, which graphically can be interpreted as lines connecting vertices v_i and v_j. Graphs can be *directed*, which means that an edge $e = \{v_i, v_j\}$ is considered as having node v_i as a source and node v_j as a target. When there exists an edge $\{v_i, v_j\}$ for every edge $\{v_j, v_i\}$, we call the graph *undirected*. A graph with n vertices is *complete* when every vertex is connected to all other vertices via $n\,(n-1) \div 2$ edges. The *degree* of a vertex is given by the number of edges incident to it, and the entire graph has a degree corresponding to the vertex with the highest number of incident edges. A vertex v_i is also said to have a *neighbourhood* $\mathcal{N}\,(i)$, which is given by the set of vertices connected to v_i.

To map an error syndrome to a graph, we create one node for every stabilizer that

has had its parity flipped between consecutive measurement rounds. In other words, we create one node for every positive detector measurement. As the stabilizers are scattered across a square lattice, the nodes of the syndrome graph can be viewed as having coordinates (x, y, t) representing where and when a specific stabilizer had its parity changed. Depending on which decoder that will be used, the nodes can be connected via edges in different ways. A general idea is that the edges connecting different pairs of nodes should be weighted in a way that illustrates how likely an error chain between those two nodes are. An example is shown in Figure 2.11, where a (complete) syndrome graph has been formed based on three cycles of stabilizer measurements. In the figure its assumed that the initial stabilizer configuration is known, otherwise we would not know that the nodes for the first time step give positive detector measurements.

2.3.2 Minimum-weight perfect matching decoders

Minimum-weight perfect matching (MWPM) decoders rely on a MWPM algorithm. Given an undirected graph with nodes V, and edges E with weights W, a *matching* is a subset of E that do not share any common nodes, and a *perfect matching* is a matching set that includes all nodes in V. The MWPM problem involves finding a set of perfectly matching edges, with the lowest sum of weights.

Since stabilizer measurements of X- or Z-errors on the surface code often appear in pairs, whether caused by a single qubit error or an error chain, MWPM can be used to find connected measurements. The nodes in this case correspond to detector measurements, and the edge weights are set to correspond to the probability of the error chain between each node pair. If one qubit error occurs with some probability p, two qubit errors occur with some probability proportional to p^2, so the probability of a longer error chain occurring is lower. This means that finding the MWPM set of a syndrome graph corresponds to finding the shortest, and therefore most likely, error chains. In Section 2.2.4, cases where single qubit errors, or error chains, do not correspond to pairs of detector measurements are presented. In these cases it is possible to get a graph with an odd number of nodes, which clearly cannot yield a perfect matching. Different MWPM decoders have different approaches to tackling this problem, but generally one or more special *virtual nodes* are added to the graph. MWPM decoders perform quite well on codes with low error probabilities, but as error probabilities increase and longer error chains occur more frequently they become less suitable for the task. Additionally, since they only consider pairs of X- and Z-stabilizer measurements separately, they do not take into account possible correlations between X- and Z-errors. By itself, MWPM is too simplistic to be used for correcting more complex error models, but its performance can be improved by combining it with for example message passing algorithms such as belief propagation [16, 29, 31].

2.4 Graph neural networks

There are many data structures besides quantum error syndromes that can be represented by graphs, including social networks, molecules and power grid systems [53]. These are all examples of fields where we have access to large amounts of data. Social media have billions of users, known molecules can be counted in the millions and a power grid system is monitored around the clock. For the last decade, machine learning and neural networks have come to dominate problems in data intensive fields like these. Convolutional neural networks and vision transformers dominate all tasks in computer vision [52, 6], recurrent neural networks are better at time-series forecasting than most classical methods [28] and generative AI can produce human-like text in a way we have never seen before [27]. This makes a strong argument for applying machine learning methods in other fields that have access to an abundance of data. However, most neural networks are designed to process data with a fixed shape. In order to leverage information encoded in graphs, a neural network must be able to handle inputs of both varying size and shape. The typical way to do this is to use a special kind of neural network, a *graph neural network*.

This section will begin by introducing the fundamental concepts of a neural network and then discuss the unique properties of a graph neural network. This is followed by a section explaining how neural networks can be trained.

2.4.1 Fundamentals of a neural network

A neural network is a computational model inspired by the human brain. Similarly to how our brains process information by feeding it to millions of neurons, neural networks approximate complex functions by composing many simple operations. Mathematically, a neural network can be represented as a non-linear parameterized function $F(\boldsymbol{\theta}; \mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n) \to \mathbf{y}_0, \mathbf{y}_1 \ldots \mathbf{y}_n$ that maps an arbitrary number of tensors $\mathbf{x}_i$ to an arbitrary number of tensors $\mathbf{y}_i$ as a function of its *parameters* $\boldsymbol{\theta}$ (also known as *weights and biases* or just *weights* for short). To understand how a neural network is constructed, let us begin by considering its smallest building block, the artificial neuron.

An artificial neuron represents a function that takes n inputs x_i. The neuron computes a weighted sum with weights w_i, adds a bias b and then wraps the entire summation in a non-linear activation function g,

$$y = g\left(\sum_{i=0}^{n}(w_i x_i) + b\right). \tag{2.4}$$

The activation functions g are typically monotonically increasing and continuous. Common examples include $\tanh x$, the logistic function $\sigma(x)$ and the rectified linear unit (ReLU),

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad \sigma(x) = \frac{e^x}{1 + e^x}, \quad \mathrm{ReLU}(x) = \max(0, x).$$

In general we want g to also be at least piecewise differentiable, which is true for $\tanh x$ and $\sigma(x)$ (differentiable everywhere) as well as ReLU (differentiable everywhere except at $x = 0$).

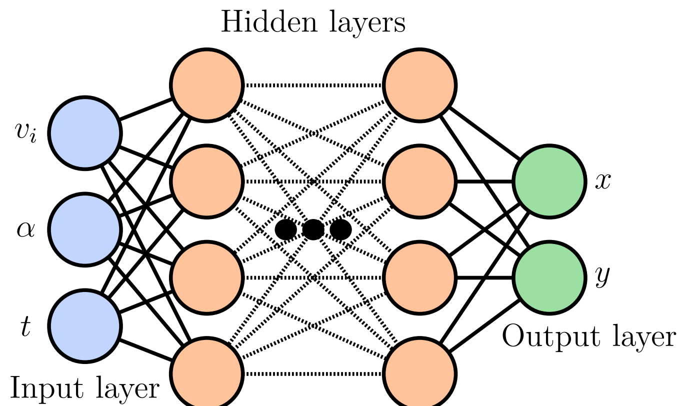


Figure 2.12: A feed-forward network with an input layer for three inputs $\{x_0, x_1, x_2\}$, a series of hidden layers and an output layer for a single output y_0.

To construct an *artificial neural network*, neurons are stacked together in *layers*. The first layer of a neural network is often referred to as the *input layer*. Then follows an arbitrary number of *hidden layers*, before a final *output layer*. The number of neurons in the output layer is chosen such that it corresponds to the dimensionality of the desired output. For a general layer k with m inputs and n neurons, Equation 2.4 can be computed for all neurons using matrix multiplication

$$\mathbf{y}^{(k)} = g\left(\mathbf{W}^{(k)}\mathbf{x}^{(k)} + \mathbf{b}^{(k)}\right), \tag{2.5}$$

where $\mathbf{W} \in \mathcal{R}^{n\times m}$ is a weight matrix and $\mathbf{b} \in \mathcal{R}^{n}$ is a bias vector. This type of mapping is usually called a *linear layer* or a *dense layer*. In a feed-forward network, the output of one layer becomes the input to the next one, formally $\mathbf{x}^{(k)} = \mathbf{y}^{(k-1)}$. A feed-forward network with l layers can then be written as a mapping between real-valued vectors,

$$y^{(l)} = g^{(l)}\left(\mathbf{W}^{(l)}\left(g^{(l-1)}\left(\mathbf{W}^{(l-1)}\left(\dots g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{y}^{(0)} + \mathbf{b}^{(1)}\right)\dots\right) + \mathbf{b}^{(l-1)}\right) + \mathbf{b}^{(1)}\right)\right),$$

with $\mathbf{y}^{(l)}$ denoting the output of the network and $\mathbf{y}^{(0)}$ representing its input. The activation functions $g^{(l)}$ introduce non-linearity to the otherwise linear matrix operations, which makes it possible for the neural network to approximate any continuous function [40].

As an example of a feed-forward network, consider a neural network tasked with predicting a projectile trajectory in a two-dimensional space. Given a vector of three inputs, initial velocity v_i, launch angle α and elapsed time t, the network is designed to predict the coordinate (x, y) of the projectile at time t. A schematic illustration of such a network is seen in Figure 2.12. The input layer takes three inputs, (v_i, α, t), which are fed through an arbitrary number of hidden layers with no restrictions on

dimensionality. To make sure the neural network predicts a coordinate vector (x, y), the output layer has two neurons.

There are many ways to create more complex neural network layers. An important example is the self-attention layer first presented in [51]. The idea is to use an attention-mechanism that allows a sequence of inputs to exchange information via a series of learnable transformations. Assuming the inputs are stacked as vectors along the rows of a matrix $\mathbf{X}$, the self-attention layer begins by transforming $\mathbf{X}$ to three new matrices, a key matrix $\mathbf{K}$, a query matrix $\mathbf{Q}$ and a value matrix $\mathbf{V}$. This is done using learnable weight matrices $\mathbf{W}_i$ according to

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}_K, \quad \mathbf{V} = \mathbf{X}\mathbf{W}_V.$$

Then follows the attention mechanism given by

$$\text{Attention} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d}}\right)\mathbf{V}, \tag{2.6}$$

where d represents the column dimension of the input matrix $\mathbf{X}$. The softmax function operates along the row-dimension of $\mathbf{Q}\mathbf{K}^T/\sqrt{d}$ and normalize the input rows having d elements to probability distributions that sum to 1. This is done by applying the transformation

$$p_i = \frac{e^{z_i}}{\sum_{i=0}^{d-1} e^{z_i}} \tag{2.7}$$

for each element z_i in the input rows. By transforming the inputs $\mathbf{X}$ to several sets of key, query and value matrices (using different sets of weight matrices $\mathbf{W}_i$) we can compute several outputs for a single input $\mathbf{X}$. To give the final output, the outputs from each attention-mechanism are concatenated and projected to the desired dimension via another learnable weight matrix $\mathbf{W}_0$. If we transform the input to n sets of key, query and value matrices we call it a multi-headed self-attention layer with n heads.

2.4.2 Exploiting graph structures in neural networks

To create a neural network optimized for processing data represented by graphs, we can create alternative mappings that utilize the information inherent in the graph structure. The advantage of using graphs to describe data instead of vectors or matrices comes from their ability to express complex relations between different objects. As described in Section 2.3.1, a graph describes relationships by connecting nodes with edges. In its simplest form, a graph can be used to describe connections between a set of labelled objects represented by the nodes. However, when representing data that is supposed to be processed by a neural network, graphs are usually designed to encode more information. To create numerical representations of the nodes, they are labelled and assigned *node feature vectors* that can describe the object the node represents. If the graph is used to describe data where colors of objects are important, the node feature vectors could be three-dimensional RGB values for instance. In a similar fashion, edges are either given scalar values, perhaps conveying how strongly two objects are related, or *edge feature vectors* that

can encode more complex information. The goal of a graph neural network is to update these vectors based on a learnable parameterization similarly to how the linear layer in Equation 2.5 does it, but ideally in a way that preserves the information contained in the graph structure [53]. To preserve the graph structure, graph neural

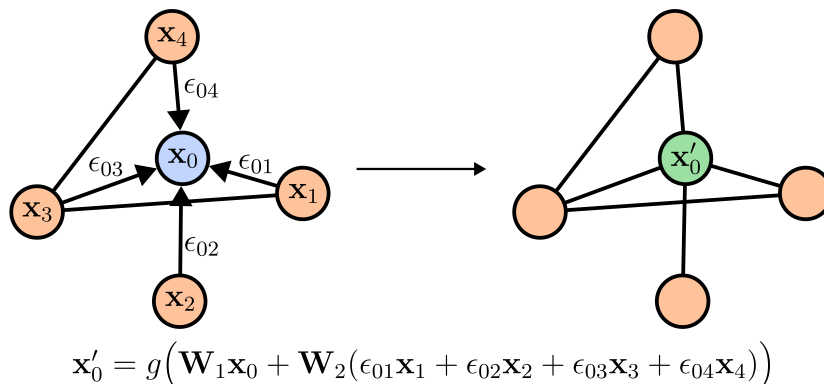


Figure 2.13: A graph convolution updating node feature vector $\mathbf{x}_0$ based on its neighbouring node feature vectors $\mathbf{x}_j$ and the edge weights ϵ_{0j}. The function g is a non-linear activation function.

networks are usually based on layers with permutation equivariant maps that update the node feature vectors via message passing algorithms. A permutation equivariant map means that the node feature vectors are transformed without changing the number of nodes and their connecting edges. For a node n_i, a message passing layer updates the node feature vector $\mathbf{x}_i$ by taking into account both its own values, and the values of the node feature vectors in the neighbourhood of n_i, $\mathcal{N}(i)$. A general message passing layer can be expressed as

$$\mathbf{x}'_i = g\left(\mathbf{x}_i, \bigoplus_{j\in\mathcal{N}(i)} f\left(\mathbf{x}_i, \mathbf{x}_j, \epsilon_{ij}\right)\right), \tag{2.8}$$

with g, f being differentiable functions and $\oplus$ a permutation invariant aggregation operator accepting an arbitrary number of inputs. Having a message passing layer in a graph neural network means that each node feature vector is subject to the above operations. One example of a message passing layer is the graph convolution operator presented in [41],

$$\mathbf{x}'_i = g\left(\mathbf{W}_1\mathbf{x}_i + \mathbf{W}_2 \sum_{j\in\mathcal{N}(i)} \epsilon_{ij}\mathbf{x}_j\right), \tag{2.9}$$

which assumes scalar edge features ϵ_{ij} and uses a weighted sum over the neighbouring node features to update a node feature. This is illustrated in Figure 2.13, where the node feature vector $\mathbf{x}_0$ gets updated based on the neighbouring node feature vectors $\mathbf{x}_j$ and the edge weights ϵ_{0j}. A network which takes graphs as inputs and transforms them with a series of message-passing layers is called a graph neural network (GNN).

2.4.3 Neural network training

Let us return to the example of predicting a projectile trajectory from Section 2.4.1. Suppose we have created a neural network and now wish to accurately predict the coordinate of a projectile by feeding it with an initial velocity v_i, the launch angle α and an elapsed time t. When we feed inputs to the network, they will pass through the linear mappings and the activation functions. The output will thus be a function of all values in the weight matrices and the bias vectors, the parameters $\boldsymbol{\theta}$. After having decided on a specific set of layers in the network, a *network architecture*, one must therefore go on to find a set of parameters that can transform the inputs to the desired outputs.

Depending on the specific problem one is trying to solve and the data available, an optimal set of parameters for the neural network can be found in different ways. Most methods are iterative, and the process of updating the parameters is usually referred to as *training* the network. A common scenario is to have access to a labelled dataset, which consists of a set of inputs paired with outputs that are known to be correct. The correct outputs are often referred to as *labels*. For the problem of predicting a projectile trajectory, a labelled dataset could for instance be created by studying high speed footage of many projectile trajectories whilst annotating the projectile coordinates as a function of initial velocity, launch angle and elapsed time. When we have access to a labelled dataset, we can use *supervised machine learning* and formulate the task of finding the optimal set of weights as an optimization problem.

In supervised machine learning, we have a dataset $\mathcal{D}$ consisting of inputs $\mathbf{x}_i$ paired with the desired outputs $\mathbf{y}_i$, $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}$. In general we can have multiple inputs and outputs, but let us consider the case of single inputs and single outputs for simplicity. Given a neural network $F(\boldsymbol{\theta}; \mathbf{x_i})$, our task is to find $\boldsymbol{\theta}$ such that the neural network predict outputs $\hat{\mathbf{y}}_i$ that approximates the desired outputs $\mathbf{y}_i$ within an acceptable margin of error. To be able to quantify how well the neural network does this, we define a *loss function* $L(\mathbf{y}_i, \hat{\mathbf{y}}_i)$ that measures the discrepancy between the prediction $\hat{\mathbf{y}}_i$ and the label $\mathbf{y}_i$. By minimizing the loss function with respect to the network weights $\boldsymbol{\theta}$, we hope to find the weights that parameterize the neural network in an optimal way. How well we succeed in doing this depends on how accurately the loss function can quantify the performance of the network and how efficient our optimization algorithm is.

There are two major categories of loss functions that are commonly used, the ones suitable for *regression problems* and the ones used for *classification tasks*. In a regression problem, the neural network is designed to predict a continuous output represented by floating-point numbers. Predicting the coordinate of a projectile is an example of this. To measure how well the network predicts a continuous output, the *mean squared error* (MSE) loss function is often used. Given a set of n predictions $\hat{y}_i$ and labels y_i, the MSE loss is defined as

$$\text{MSE} = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \hat{y}_i)^2 .$$

When dealing with a classification task, the neural network aims to predict which class a set of inputs corresponds to. This means that the possible outputs are confined to finite set of discrete classes. When designing a neural network to predict which of the k classes an input belongs to, the output layer of the network is commonly chosen to have k neurons. Each neuron represents a class, and the neuron outputs are interpreted as the probability that an input belongs to a specific class. The labels are transformed to have the same shape as the output of the neural network, using *one-hot encoding*. One-hot encoding for dimension k takes an integer i and transforms it by creating a vector of length k filled with zeros except at element i, which gets a 1. If a we have inputs belonging to three classes, a label indicating class 2 would for instance be transformed as

$$\text{Class 2} \longrightarrow \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}.$$

If the neuron outputs are to be interpreted as probabilities they must all be in the range $p \in [0, 1]$ and sum to one. To achieve this, the output layer uses the softmax function defined in Equation 2.7 as an activation function. After the softmax has been applied, the output probabilities $\hat{p}_j$ can be compared to the one-hot encoded label vectors using the *cross-entropy loss* H,

$$H = -\sum_{j=0}^{k-1} y_j \log(p_j), \tag{2.10}$$

with y_j representing element j of the label vector and p_j the prediction from neuron j. Note that unlike for the MSE loss, we have omitted the sum over several outputs here. In general, H is computed and averaged over a set of output vectors [40].

Having chosen a loss function, the next step is to minimize it with respect to the parameters in the network. This is generally a computationally intensive task, as the number of weights can be very high (in the order of millions or even billions) [48, 14]. If the neural network and the loss function are only composed of differentiable functions, gradient-based algorithms can be used. In practice this is almost always the case, with different forms of gradient descent being the most common optimization algorithms used. Gradient descent is an iterative algorithm that involves computing the partial derivative of every parameter in the network with respect to the loss function. This had been impossible for large networks if it were not for a technique known as *backpropagation* [44]. Backpropagation utilize the chain rule and computes gradients starting from the output layer, capturing how changes in every parameter contributes to the overall loss. A basic implementation of gradient descent updates the network parameters θ according to

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} \left(L\left(\mathbf{y}, \hat{\mathbf{y}}\right)\right).$$

The *learning rate* η controls how much the network parameters are allowed to change during an iteration. To give faster convergence and make sure the algorithm does not get stuck in local minima, several improvements can be made by adding additional terms. A common choice is to use the Adam optimizer, which implements

an algorithm for stochastic gradient descent (SGD) that estimate first-order and second-order moments to help avoid local minima [40, 33].

Apart from choosing a suitable loss function and having an efficient optimization algorithm, the ability of a neural network to do well on a given task is highly dependent on the access to large amounts of data. During training, the dataset is split into *batches* which are fed to the network in parallel to allow for efficient computations. Iterating through all batches compromise an *epoch*, which is a common measure of how long the neural network has been trained. The number of epochs needed to reach convergence can vary depending on how large the dataset is and how the neural network is constructed.

3
Methods

In this chapter, we present the methods employed in the development of our novel quantum error correction decoder, the *neural matching decoder* (NMD). Apart from explaining how the NMD was constructed, we also discuss how error syndromes were collected and mapped to syndrome graphs, explain the different network architectures that were evaluated and discuss two different methods used to train the networks.

3.1 Neural matching decoder

The NMD was designed to decode error syndromes from the rotated surface code by combining a graph graph neural network (GNN) and a classical minimum-weight perfect matching (MPWM) algorithm. When decoding an error syndrome, the NMD predicts which equivalence class an error syndrome belongs to. For the rotated surface code, any given error syndrome can belong to either the identity class I, the logical X-class X_L, the logical Z-class Z_L or the logical Y-class Y_L. The logical Y-class can be decomposed in terms of X and Z, and we can therefore split the problem in two parts and predict which of the classes $\{I, X_L\}$ and $\{I, Z_L\}$ the error syndrome belongs to separately. A subtlety is that the definitions of the equivalence classes are dependent on how the logical operators on the surface code are defined. The NMD has been used for rotated surface codes with logical operators defined according to Section 2.2.3.

By leveraging a GNN in the NMD, we adopt a data-driven methodology and use supervised machine learning to train the decoder to identify the equivalence class of an error syndrome without direct information about the underlying noise processes. Accurately describing all noise processes in a real quantum circuit is a complex task, which makes a decoder that does not rely on noise modeling practical. By combining the GNN with a MWPM algorithm, we aim to reduce the computational complexity of the decoder whilst retaining a high decoding accuracy. This is important in future applications, where decoders might be used for real-time decoding in fault-tolerant quantum computers.

The sections below will begin by giving an overview of the different parts of the NMD. Then follows a section on what type of data that has been used to train the GNN, and a description of how the error syndromes have been mapped to syndrome graphs. Next, the different neural network architectures that were explored are

presented. Finally, we explain two methods that were used to train the neural networks used in the decoder.

3.1.1 Decoder overview

An overview of the NMD is shown in Figure 3.1, where we divide the decoder into one pre-processing step followed by three major components, a GNN, two *heads* and two *matching modules*. The pre-processing step converts the error syndromes to syndrome graphs, in which we encode information that can be of use for the decoder. How this is done will be discussed in the two following sections. For now, let us assume that the syndrome graphs have already been created and are ready to be processed by the NMD. The NMD begins by feeding the syndrome graphs to the GNN, which consists of a series of message passing layers. At the end of the GNN, the decoder splits each syndrome graph into two subgraphs and continues to decode the error syndrome via two parallel heads. One head will use a subgraph with nodes originating only from X-stabilizers, and predict whether the error syndrome belongs to the identity class I or the logical Z-class. The other head will use a subgraph with nodes only from Z-stabilizers and predict whether the error syndrome belongs to the identity class I or the logical X-class. If none of the heads predict the identity class, the error syndrome will belong to the logical Y-class. The heads are designed

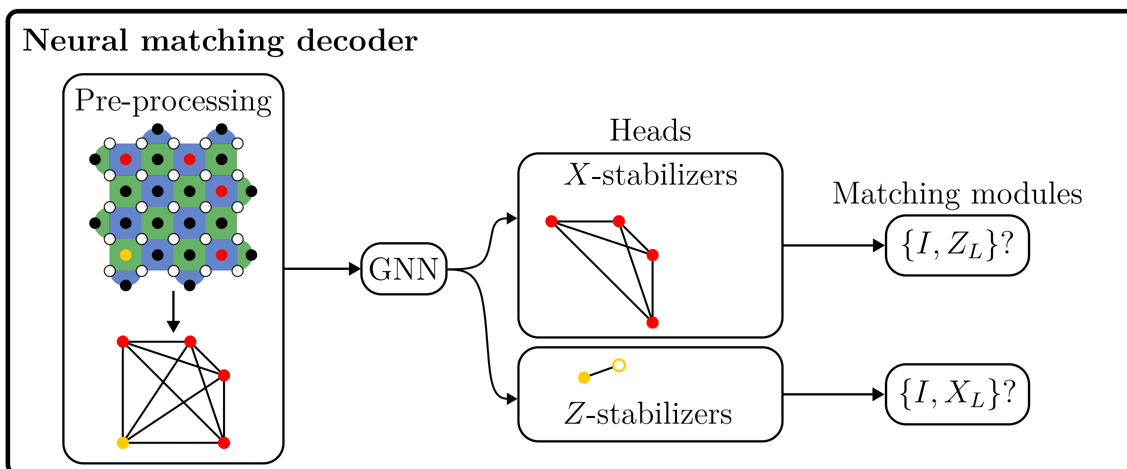


Figure 3.1: An overview of the neural matching decoder. The pre-processing step takes an error syndrome and creates a syndrome graph, which is fed to a GNN that updates the graph representation in a series of message passing layers. After the GNN, the syndrome graph is split in two subgraphs, one associated with nodes originating from X-stabilizers and one associated with nodes from Z-stabilizers. The subgraphs are processed in two parallel heads, giving a new set of edge weights for the original syndrome graph. Finally, the equivalence class of the error syndrome is predicted via two parallel matching modules utilizing a MWPM algorithm.

to compute vectors that have one element for every edge in the subgraphs. These vectors are interpreted as a new set of edge weights and replace the ones initially used in the subgraphs. When the initial edge weights have been replaced, the subgraphs are sent to the matching module that contains the MWPM algorithm. The MWPM

algorithm is implemented using Blossom V [36] and returns the subset of edges that yields the perfect matching with the lowest total weight. For a more extensive discussion on MWPM and the definition of a perfect matching, see Section 2.3.2.

By using two different types of edges in the syndrome graphs, the subset of edges given by the perfect matching can be used to infer the equivalence class of the error syndrome. The first edge type represents error chains confined to the bulk of the surface code, whilst the second type represents error chains with two parts, extending to the code boundaries. Using the definition of a rotated surface code from Section 2.2.3, the logical Z-operator is situated on the left boundary whereas the logical X-operator is situated along the lower boundary. This means that error chains confined to the bulk of the code belong to the identity class. However, an error chain extending to the boundaries belongs to the logical X- or logical Z-class. If an odd number of error chains belong to the logical X- or Z-class, the entire error syndrome will belong to that logical class. The subset of edges used in the perfect matching can be interpreted as the error chains that are most likely to have yielded a given error syndrome. An illustration of this is shown in Figure 3.2. By counting

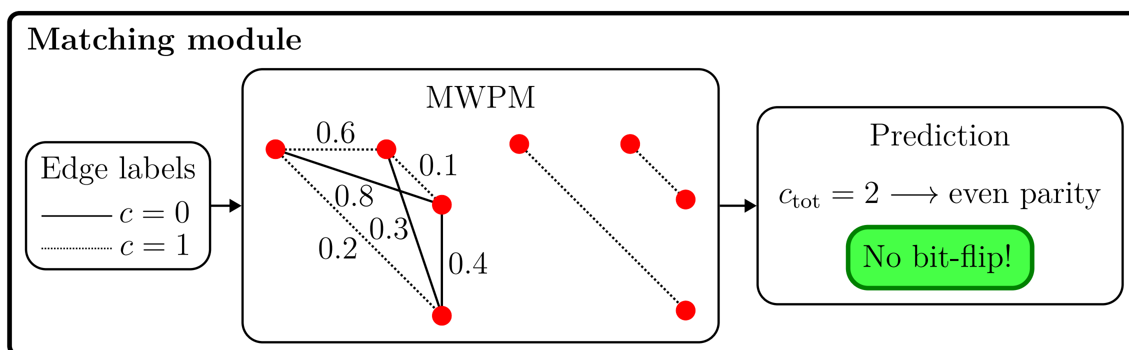


Figure 3.2: The matching module that processes syndrome graphs with updated edge weights by use of the MWPM algorithm. The edges belong to one of two classes, here shown as dashed or solid lines marked with $c \in \{0, 1\}$. By counting how many of the edges included in the perfect matching that represent an error chain extending to the boundaries of the surface code, the equivalence class can be predicted.

if an odd or even number of those edges represent error chains extending to the boundaries, the matching module can therefore predict the most plausible logical X- or Z-class of an error syndrome.

3.1.2 Sampling of error syndromes

In contrast to decoders purely based on classical algorithms, the performance of the NMD is highly dependent on the availability of a large dataset for training the neural networks. Given the scarcity of experimental realizations of the surface code, the NMD was trained solely on error syndromes sampled from a simulated rotated surface code. The quantum circuits used in the surface code were simulated using Stim, a Python package for high performance simulation and analysis of quantum

stabilizer circuits [22]. This approach facilitated the sampling of an arbitrary number of error syndromes.

When simulating the surface code, qubit errors are applied based on a noise model called *circuit-level noise*. This means that all qubits in the quantum circuit are subjected to noise, including the measurement qubits. We used the same setup as in [38], defining an error probability p that all applied errors are proportional to. Bit-flip or phase-flip errors are applied to the data qubits both at the start of a cycle of stabilizer measurements and before each measurement within the cycle. For the measurement qubits, bit-flip or phase-flip errors are applied after each Clifford gate and after the resets following each measurement. The stabilizers were measured concurrently for d_t rounds. An error syndrome is created by comparing the measurement outcomes between consecutive rounds, as described in Section 2.2.4. After d_t rounds, all data qubits were measured in either the X-basis or the Z-basis. This measurement collapsed the quiescent state, but also gave direct information about what error chains the qubit errors had caused during the d_t rounds. Depending on if the measurement was done in the Z- or X-basis, STIM used this information to deduce whether the error syndrome belonged to the identity class I, the logical X-class or the logical Z-class. This became our label when we trained the NMD. Note that X and Z do not commute, which meant that for any given error syndrome we could never know if it belonged to both the logical Z- and the logical X-class. Instead we either ran what in STIM is called *memory-Z experiments* and got a label $l \in \{I, X_L\}$ or we ran *memory-X experiments* and got $l \in \{I, Z_L\}$. After the data qubits had been measured, the parities of all stabilizers at the final round could be inferred without measurement errors yielding a so called *perfect stabilizer measurement*. In total this gave error syndromes that could have positive detector measurements in a maximum of $d_t + 1$ time steps.

The NMD was trained to decode error syndromes from a specific code size d being measured a certain number of rounds d_t. By initialising the simulations before we started to train the NMD, error syndromes could be sampled very efficiently. This enabled us to sample a new set of error syndromes before every forward pass, effectively generating new training data on the go. This optimized memory usage and allowed us to train the network on an unbounded dataset. The NMD was only trained on rotated surface codes with $d = 5$ and $d_t = 5$, but the error rate p varied in the range $p = 0.001 - 0.005$. For any given simulation, we always used a depolarizing error channel which meant that the probability of a bit-flip error was equal to the probability of a phase-flip error.

3.1.3 Measurements to syndrome graphs

Before feeding the error syndromes to the NMD, they were mapped to syndrome graphs using a procedure similar to the one described in Section 2.3.1. Positive detector measurements were used to create nodes associated with node feature vectors $\mathbf{x}_i$. The node feature vectors had five elements, $\mathbf{x}_i = (X, Z, x, y, t)$, where $X, Z \in \{0, 1\}$ indicated what kind of stabilizer a node represented, $x, y \in [0, d+1]$ described where in the surface code the stabilizer was physically situated and $t \in [0, d_t+1]$ indicated

at what time step a parity change was detected.

After the nodes had been created, edges were generated using a k-nearest neighbour search which allowed each node to have k outgoing edges to its k nearest neighbours. For most trials, we set $k = 20$. In general this does not result in an undirected graph, so for all node pairs with only a directed edge between them, another edge in the opposite direction was added. This ensured that the edge set defined an undirected graph. When the edges had been created, they were all duplicated to yield an edge set twice as big. This resulted in two undirected edges for every connected node pair. One was used to represent error chains confined to the bulk of the surface code, and the other was used to describe error chains extending to the boundaries of the code. For every pair of nodes, the syndrome graph thus described two possible error chains that could have caused the parity changes represented by the nodes. To distinguish between the two types of edges, all edges were associated with an edge class $c \in \{0, 1\}$. An edge representing an error chain in the bulk was labelled with $c = 0$ and an edge describing an error chain extending to the boundaries was labelled $c = 1$. From now on, an edge with label $c = 0$ will be referenced to as an *inside edge* whilst an edge with label $c = 1$ will be called an *outside edge*.

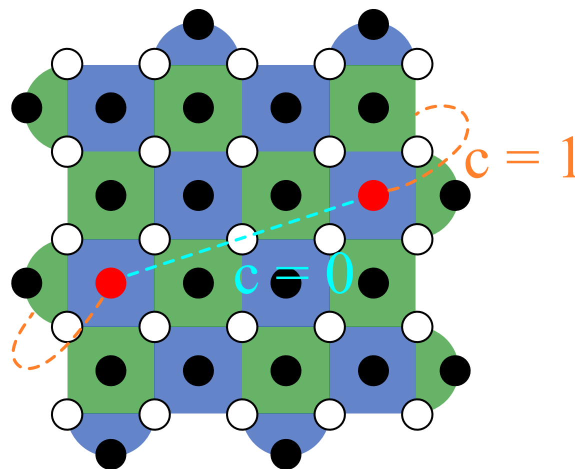


Figure 3.3: An example of how two edges can be created for any pair of nodes in the syndrome graph. The inside edge is drawn in turquoise and the outside edge is drawn in orange.

In addition to a class label, all edges were given initial weights w_{ij} related to length of the error chains they represented. The inside edges were given weights based on the inverse square of the Euclidean distance d_{ij} between nodes i and j, $w_{ij} = 1/d_{ij}^2$. For the outside edges, a modified Euclidean measure taking into account how the error chains were assumed to extend to the surface code boundaries was used. If the error syndrome originated from a memory-Z experiment, the absolute distance between node i and node j was subtracted from the total width of the surface code in the latitudinal axis. For error syndromes originating from a memory-X experiment, the absolute distance between node i and j was instead modified in the same way but for the longitudinal axis. The modified distance measure $d_{ij}^{'}$ was then used to compute weights $w_{ij}^{'} = 1/d_{ij}^{'2}$. The two distance measures are illustrated in Figure 3.3.

A problem with how the error syndromes were mapped to syndrome graphs is that there are no guarantee that the graphs would get an even number or nodes or edge sets that can be used to find perfect matches. When the NMD uses the matching module to predict the equivalence class of an error syndrome, the MWPM algorithm must be able to find a perfect matching. Since the two heads have one matching module each, the two subgraphs with nodes originating from either only X- or only Z-stabilizers must both fulfil the requirements for the existence of at least one perfect matching. To ensure this, virtual nodes were added whenever the subgraphs had an odd number of nodes. The virtual nodes were given feature vectors $\mathbf{x}_{\text{virtual}} = (X, Z, 0, 0, 0)$, with $X, Z \in \{0, 1\}$ indicating the type of stabilizer it represented. They were connected to all other nodes with two undirected edges each, one for each edge class, and were given initial edge weights w_{ij} and classes c like the ordinary edges. It was empirically observed that this always made it possible for the MWPM algorithm to find a perfect matching.

3.1.4 Network architectures

The NMD used a GNN that remained similar for all trials, see Figure 3.4 for a schematic illustration. It consisted of two message-passing layers utilizing the graph convolution operation in Equation 2.9. The scalar edge features ϵ_{ij} used in the weighted sum of the graph convolution were computed via an embedding layer according to Equation 2.5. As an input to the embedding layer, a concatenation of the initial edge weights w_{ij} and the class labels c given to the edges of the syndrome graphs were used. Before the concatenation, the class labels c were one-hot encoded such that $c = 0$ was represented by the vector $\mathbf{c}_0 = (0, 1)$ and $c = 1$ was given by $\mathbf{c}_1 = (1, 0)$. The message passing layers had dimensions in the range 32-512 across all trials. A graph normalisation layer as described in [12] was added before continuing with the transformations.

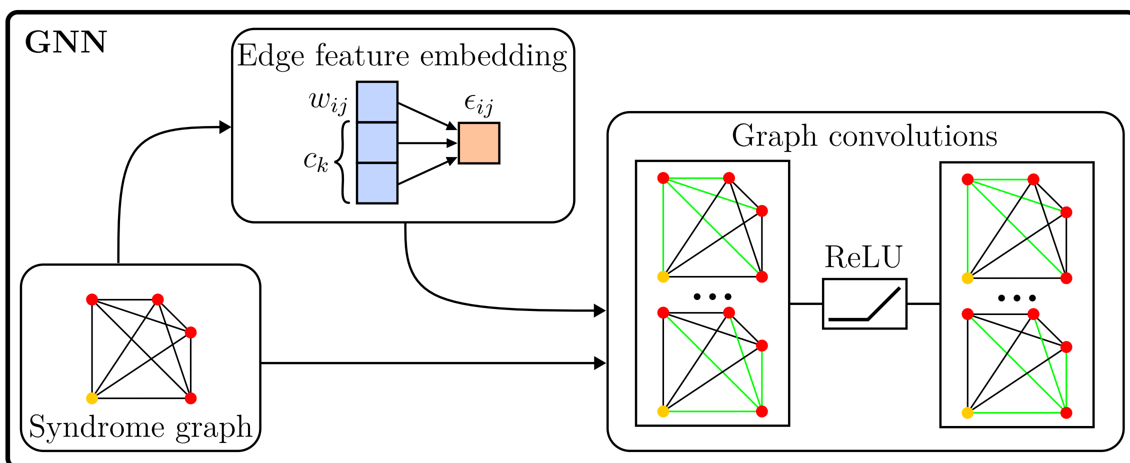


Figure 3.4: A schematic illustration of the GNN used in the NMD. The initial edge weights w_{ij} are used together with the class labels c_k to create a scalar edge feature embedding. This embedding is used to weight the sums in a series of graph convolutions, which are followed by ReLU-activations.

After the graph normalisation layer, the syndrome graphs were split in two subgraphs with nodes originating from only X- or Z-stabilizers. The subgraphs were handled independently by two parallel heads. Two types of heads were explored, one with a simple feed-forward network and one utilizing attention-mechanisms. The simple feed-forward network used a series of linear layers according to Equation 2.5, whilst the other type of head implemented the self-attention layer described in Equation 2.6. Both heads are shown in Figure 3.5, with the left side of the figure illustrating the feed-forward alternative and the right side showing the self-attention option. Note that only one head is shown for each explored alternative, displaying how one of the subgraphs would be processed. Since the heads were tasked with computing vectors that could be interpreted as a set of edge weights, the graph structure was abandoned at this point. To do this without losing valuable information, every edge in the subgraphs was given a vector representation, an edge feature vector. The edge feature vector for an edge between node i and node j was created based on the node feature vectors $\mathbf{x}_i$ and $\mathbf{x}_j$. Since the edges in the syndrome graphs were given initial weights w_{ij} and class labels c, this information was also exploited via another embedding layer. In contrast to the first embedding layer, which was used to compute scalar edge features ϵ_{ij} before the graph convolution, this embedding layer computed longer vector representations $\boldsymbol{\omega}_{ij}$. The inputs were the same however, the initial edge weights w_{ij} and one-hot encoded class label embeddings c_k, but the size of the embedding layer was chosen such that it transformed its inputs to the same lengths as the node feature vectors. This meant that for every edge in the subgraph, we had three vectors with useful information, two node feature vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ as well as an embedding vector $\boldsymbol{\omega}_{ij}$. An edge between node i and j was therefore given an edge feature vector $\mathbf{e}_{ij}$ by concatenation,

$$\mathbf{e}_{ij} = \mathbf{x}_i \| \boldsymbol{\omega}_{ij} \| \mathbf{x}_j.$$

This was done per subgraph within each head. The edge feature vectors were stacked in matrices with one row for every edge in the subgraphs.

In a feed-forward network, data encoded in rows of matrices cannot exchange information. Since the NMD predicts the equivalence class of an error syndrome by checking the class labels of all edges used in the perfect matching, the set of edge weights that yields the perfect matching leading to the correct equivalence class is likely to depend on the entire syndrome graph. For the trials using heads with feed-forward networks, a global node feature vector was therefore computed by taking the mean of all node feature vectors in the syndrome graph (before we split it into two subgraphs),

$$\bar{\mathbf{x}} = \frac{1}{n} \sum_{i=0}^{n-1} \mathbf{x}_i.$$

This global node feature vector $\bar{\mathbf{x}}$ was used to extend the edge feature vectors further,

$$\mathbf{e}'_{ij} = \mathbf{x}_i \| \boldsymbol{\omega}_{ij} \| \bar{\mathbf{x}} \| \mathbf{x}_j.$$

By doing so, every edge feature vector belonging to a certain subgraph was given information about both its immediate neighbourhood and the entire error syndrome.

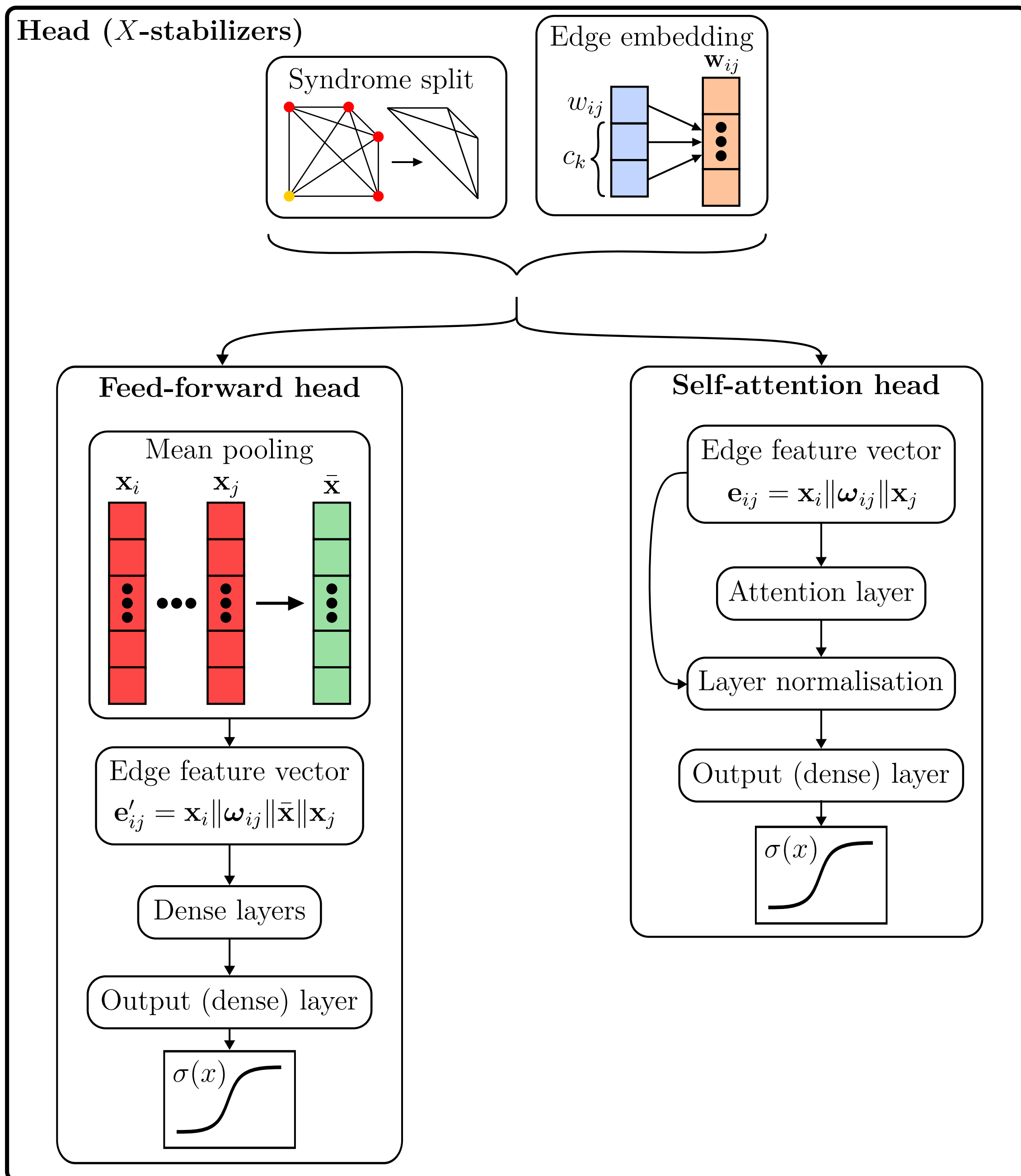


Figure 3.5: The two different heads that were explored shown together with the split of the original syndrome graph and subsequent edge embedding. The feed-forward head used a mean pooling operation to extend the edge feature vector, whilst the self-attention head used a simpler representation but added a skip connection and layer normalisation.

For the trials with heads using a self-attention layer, the global node feature vector was consider superfluous and was not added to the concatenation. In a self-attention layer, information is exchanged in a much more flexible manner and any given edge feature vector can be transformed as a function of the others. This limits the advantage of adding a global node feature vector.

In terms of sizes, the trials using heads with feed-forward networks employed two to four linear layers with dimensions in the range 32-512. The trials using heads with a self-attention layer transformed the edge feature vectors without changing their dimensions. A skip connection feeding a copy of the edge feature vectors past the self-attention layer was also used. The copy was added to the transformed edge feature vectors after the self-attention layer and their summation was sent through the layer normalisation function described in [4]. As a general rule, the activation function ReLU was used between all layers throughout the network.

Both types of heads finished with a linear layer projecting the edge feature vectors to scalars, which were interpreted as new sets of edge weights for the syndrome graphs. At this point, the syndrome graphs still had two types of edges for every pair of nodes. The MWPM algorithm can only handle one edge per node pair, which meant that we needed to choose which one to keep. Inspired by how the MPWM algorithm matches edges that give a minimum total weight, we compared the newly computed edge weights for every pair of edges belonging to the same node pair and kept the edge with the lowest weight. This gave us syndrome graphs having only one edge per node pair. The updated edge weights were then sent through the logistic function defined in Equation 2.4.1, which re-scaled them to the range $e_{ij} \in [0, 1]$. The syndrome graphs, equipped with new edge weights, were finally sent to the matching module where the equivalence class of the error syndromes could be predicted.

3.2 Network training

To create a working neural matching decoder (NMD), the graph neural network and the two neural network heads must be parameterized with a suitable set of parameters. In order to find the optimal parameters, we use supervised machine learning and train the decoder with labelled syndrome graphs. The syndrome graphs are created based on error syndromes sampled using Stim, as described in Section 3.1.2. Note that the two heads in the NMD are independent, with one being responsible for the logical X-class and the other being responsible for the logical Z-class. Since the error syndromes sampled using Stim only had one label, identity or logical X/Z-class, the two heads had to be trained separately. With the noise model defined in Section 3.1.2 there is no statistical difference between a bit-flip error and phase-flip error, which means that it was sufficient to show how well the NMD performed when training only one head.

The neural networks in the NMD are responsible for computing new edge weights for the syndrome graphs. A syndrome graph is considered to have an optimal set of edge weights when the minimum-weight perfect matching (MWPM) algorithm finds a perfect matching that gives the correct equivalence class of error syndrome. The matching module in the NMD does this by counting if an odd or even number of edges in the perfect matching represent an error chain extending to the boundaries of the surface code. This means that until we have passed the new edge weights through the MWPM algorithm in matching module, we have no way of knowing if the neural networks have given us an optimal output. In other words, the labels

given to the syndrome graphs cannot be used to verify the correctness of the neural network outputs before the matching module has finished.

Needing to pass the neural network outputs through the matching module before comparing the predicted equivalence class to our label becomes a problem when training the networks. The MWPM algorithm is not differentiable, which means that there is no obvious differentiable loss function that can compare our predictions to the labels. This prevents us from directly training the networks using gradient descent, which is by far the most common way to train a neural network.

To circumvent the issue with a non-differentiable loss function, two different approaches were tested. One alternative was to create a surrogate loss function that could be differentiated, allowing us to use gradient descent based on an indirect comparison of the predictions and the labels. The second alternative was to use gradient-free optimization, employing a method called *local search*. Both options are presented in the sections below.

3.2.1 Surrogate loss

The surrogate loss function was designed to bypass the MWPM algorithm by creating *pseudo-labels* that could be compared directly to the set of edge weights computed by the neural network head. This was done dynamically when training the NMD, right after the matching module had processed the syndrome graph with the newly computed edge weights e_{ij}. The matching module begins by feeding the syndrome graph through the MWPM algorithm, which returns the subset of edges that yield the perfect matching with the lowest weight. By continuing to run the matching module, we get a prediction of what equivalence class the error syndrome belongs to. If the predicted equivalence class is correct (which we check by comparing to the label of the error syndrome), we generate a set of pseudo-labels l_{ij} according to

$$l_{ij} = \begin{cases} 0, & \text{if } e_{ij} \text{ is one of the edges in the perfect matching.} \\ 1, & \text{otherwise.} \end{cases} \tag{3.1}$$

Note that $e_{ij} \in [0, 1]$ since the final layer in the network heads is followed by the logistic function. In some sense, this makes the label l_{ij} represent an ideal edge weight given the perfect matching that was found.

However, if the predicted equivalence class is wrong, it becomes harder to define what an ideal edge weight should have been. The naive choice is to invert the pseudo-label used when the prediction is correct, giving

$$l_{ij} = \begin{cases} 1, & \text{if } e_{ij} \text{ is one of the edges in the perfect matching,} \\ 0, & \text{otherwise,} \end{cases}$$

but there is no guarantee that the subset of edges *not* used in the perfect matching represent another possible perfect matching or even one that would give the correct equivalence class. Instead we used a brute-force approach where new edge sets were

generated by randomly sampling $e_{ij} \sim U[0, 1)$. For each new edge set, we ran the MWPM algorithm until a perfect matching that could be used to infer the correct equivalence class was found. When that was successful, a set of pseudo-labels were created using the same approach as in Equation 3.1. To avoid a scenario where we never would find a perfect matching yielding the correct equivalence class, we reverted to the naive choice of pseudo-labels if the method was not successful within 30 iterations.

When a pseudo-label had been created, the cross-entropy loss was computed by use of Equation 2.10 for every computed edge weight e_{ij}. The subset of edges used in the perfect matching typically consisted of fewer edges than the ones that were not matched. When training the network, this could create a bias towards computing edge weights with higher values, as this would lower the value of the loss function faster. To avoid this, we normalised the loss of each edge weight within a syndrome graph based on a ratio of how many edges were used in the perfect matching.

Since the cross-entropy loss is differentiable, the pseudo-labels made it possible to compute a gradient with respect to the parameters in the network. This allowed us to train the NMD using stochastic gradient descent (SGD), which was implemented with the Adam optimizer. We used a learning rate of $\eta = 0.0001 - 0.00001$ for all trials and trained with batch sizes of 10´000-80’000 syndrome graphs. Since our dataset was generated on the go by sampling error syndromes from a simulated surface code, the usual definition of an epoch did not apply. Instead we defined an epoch to comprise 10^6 syndrome graphs, and trained the NMD for 1000 epochs. In practice this meant that the NMD was trained using 10^9 uniquely sampled error syndromes.

3.2.2 Local search

As an alternative to a having a loss function based on pseudo-labels, we attempted to train the NMD with a gradient-free optimization method called local search (LS). The implementation was similar to a variant presented in [2], using only prediction accuracy as a training metric. The dataset contained more samples with equivalence class I than equivalence class X/Z, the latter making up around 15% of the data set. The classic accuracy metric, calculated as

$$A = \frac{\text{correct predictions}}{\text{all predictions}},$$

does not take into account the prediction performance of individual classes. For an imbalanced dataset with significantly more samples of one class than the other, this could skew training towards choosing the majority class, since statistically that is more likely to be correct. In our case, always guessing equivalence class I would yield an overall accuracy score of around 85%, but the prediction accuracy of equivalence class X/Z would be 0%. To avoid this, an alternative *balanced accuracy* was implemented that took the average of the *sensitivity*

$$\text{Sensitivity} = \frac{\text{TP}}{\text{TP+FN}}$$

and *specificity*

$$\text{Specificity} = \frac{\text{TN}}{\text{TN+FP}}$$

where TP, TN, FP and FN denote true positive, true negative, false positive and false negative respectively. Conversely, sensitivity could be said to represent the fraction of correct positive predictions (correctly predicting that a sample does belong to a class), while specificity represents the fraction of correct negative predictions (correctly predicting that a sample does not belong to a class). The decoder predicts whether the logical state has changed, i.e. if a sample belongs to class X/Z (depending on head), which means all correct predictions of class X/Z belongs to the true positives. If the decoder correctly predicts that the logical state has not changed, i.e. the sample belongs to class I, this prediction belongs to the true negatives. Now, for the scenario we described above, where the decoder simply classifies all samples as class I, the sensitivity would be zero, while the specificity would be one. The balanced accuracy score takes the average of the sensitivity and the specificity,

$$A_{balanced} = \frac{\text{sensitivity+specificity}}{2}.$$

So, instead of the relatively high training signal of 85% we would get from using classic accuracy, we would get a training signal $A_{balanced} = \frac{0+1}{2} = 0.5$, so 50% prediction accuracy, the exact same result we would get if the dataset contained 50% class I samples and 50% class X samples. Training was performed using both accuracy and balanced accuracy as metric, and since no distinction between them is necessary for describing the method, in this section we will simply refer to both as *accuracy*.

The first step in the optimization process consisted of testing the classification performance of the initial network with randomly generated parameter values on a fixed training set of around 500'000 syndrome graphs. The resulting accuracy was set as the initial *best value*, and the initial network parameters were set as the *elite vector*. The N trainable parameters θ in the network were flattened into one N-dimensional vector and partitioned into equal, smaller batches of S parameters each. The batches represented "local" subspaces and were partitioned randomly, not according to their position in the parameter vector or restricted to network layers. The number of batches is denoted by $K = N/S$.

Each optimization step then consisted of generating a noise vector of size S from a uniform random distribution $U(-r, r)$, where r denotes the *search radius*. This noise vector was added to the S parameters in one partition. The classification performance of the modified network was tested, and if the resulting accuracy was higher than the current best value, the modified vector replaced the current elite vector and the measured accuracy became the new best value. If performance was worse, or equal, the modified parameters were reverted back to their previous values. This meant that after each optimization step, the network parameters were always set to the current elite vector. To allow the algorithm to escape local maxima, a *score decay* was implemented. In each optimization step, the current best accuracy was decreased by some small amount, allowing the model to expand its search to parameters that initially might perform slightly worse, but with some further updates

could lead to better performance. One full cycle consisted of all partitions being modified with the accuracy of the NMD tested once. Training consisted of multiple cycles, and after each cycle the partitions were randomly re-selected. This process continued until some time limit was reached.

LS optimization does not utilize gradients for updating network parameters in the way SGD does. This meant mini-batch training was not a suitable approach, and the full training set was used in each optimization step. However, processing the full data set requires a lot of memory, so to reduce memory footprint the data set was split into smaller sets that were evaluated separately. The network performance on each small subset was saved, and once predictions had been made for the full training set the total accuracy score was calculated and sent to the optimization algorithm as training signal.

Since LS used a fixed training set, we needed to ensure the network was not over-fitting to the training set. This was done by measuring network performance on a smaller, fixed validation set each time a new best accuracy was found. If performance started to drop on the validation set, while continuing to increase on the training set, we knew the network had started to over-fit, and training performance was no longer a reliable indicator of the network's classification performance. To further monitor progress, network performance was evaluated every 500th optimization step regardless if a new best set of network parameters had been found in that step.

The training parameters search radius, score decay and number of dimensions per partition were chosen empirically. Network performance for different values was repeatedly measured, and the values yielding the best results were chosen.

4

Results

In this chapter, we present the performance of the neural matching decoder (NMD) for a series of different network architectures and the training methods stochastic gradient descent and local search. Finally, the prediction performance of the NMD is benchmarked against the performance of previously implemented MWPM and GNN decoders.

4.1 Training setup

Several versions of the NMD was created by using networks with different heads and various sizes. The different network architectures used are described in Table 4.1, and are abbreviated FFx if the head used a feed-forward network or ATNx if the head employed a self-attention layer.

Name	GCN dimensions	MLP dimensions	# attention heads	# parameters
FF1	256, 512	256	N/A	6.6×10^5
FF2	256, 512	256, 128, 128, 64	N/A	7.2×10^5
ATN1	128, 256	N/A	1	4.2×10^6
ATN2	256, 512	N/A	1	1.7×10^7
ATN3	128, 256	N/A	3	4.2×10^6

Table 4.1: The different network architectures used in the NMD. FFx corresponds to heads having only dense layers, whilst ATNx represents networks with a self-attention layer. GCN stands for graph convolutional network and MLP stands for multi-layer perceptron.

Two different types of datasets were used to train the networks. The first dataset was constructed by sampling error syndromes from a simulation that had a fixed probability $p = 0.001$ for all error rates. For the second dataset, the probability p of all error rates in the simulation was instead varied in steps of 0.001 between $p = 0.001$ and $p = 0.005$. During training, error syndromes originating from simulations with different p:s were used in equal amounts. A higher error rate means (statistically) more positive detector measurements, which meant the second dataset included larger syndrome graphs than the dataset constructed based on a fixed error rate.

The networks were trained using either NVIDIA A40 with 48 GB of memory or NVIDIA A100 with 40 GB of memory. The wall times required to train the networks

varied depending on the size of the model and the dataset used. No network was trained for more than a week, but even the smaller ones required at least a day or two.

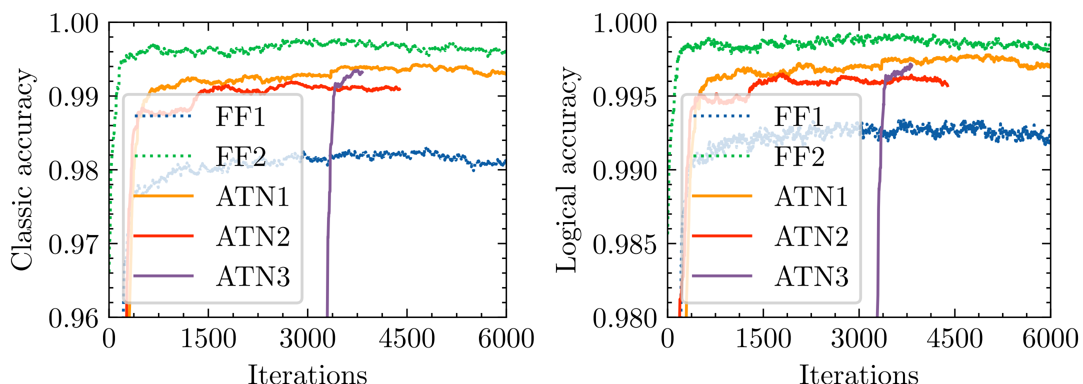
Two performance metrics called *logical accuracy* and *logical failure rate* are used for evaluations and are defined as follows. Given a set of N error syndromes sampled from STIM, only n of them have a non-zero number of positive detector measurements and are sent to the decoder. This leaves $N - n$ *trivial syndromes* that are assigned to the identity class directly. Out of the n error syndromes sent to the decoder, the correct equivalence class is predicted for k of them. The logical accuracy is then defined as

$$\alpha_L = \frac{k + (N - n)}{N},$$

and we define the logical failure rate as $1 - \alpha_L$.

4.1.1 Training with local search

Figures 4.1-4.4 show the results of training the networks described in Table 4.1 using the optimization algorithm local search (LS). Each network was trained on two datasets with 500'000 error syndromes, one with a fixed error rate $p = 0.001$ and the other with a mixed error rate $p = 0.001 - 0.005$. To measure how well the optimization progressed, a validation set with 40'000 error syndromes was used. The validation set consisted of error syndromes sampled from simulations with the same ratio of error rates as the training set. Note that the datasets were uniquely generated for each network, which means there could be some minor differences in the training and validation data used for each network. Also note that the training results are the values of the metric the networks were trained on (classic or balanced accuracy), and the validation results show logical accuracy.

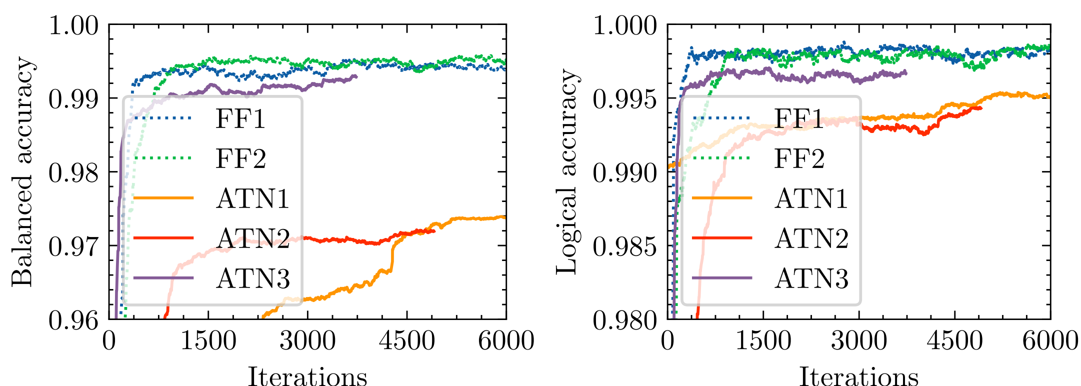


(a) Training performance represented by classic accuracy score.

(b) Validation performance represented by logical accuracy score.

Figure 4.1: Prediction performances on datasets with fixed error rate $p = 0.001$ for networks trained with classic accuracy as metric. Training performance shows classic accuracy for a dataset of 500'000 syndromes sampled from STIM, while validation performance shows logical accuracy for a dataset of 40000 syndromes.

Clearly, LS had the capacity to significantly improve the performance of all networks. This was the case regardless of the starting performance of the initial, randomly generated network parameters. Figures 4.1 and 4.2 show network performance on the data set with fixed error rate, using classic and balanced accuracy, respectively. In both cases, the rate of improvement was very fast, with most networks reaching an accuracy of over 98% in just a few hundred training iterations. After that, improvement slowed, and plateaued at around 99% accuracy for most networks. The best performing network overall seems to be FF2, but the comparative performance of the other networks varied more. A notable outlier is the training performance of ATN1 and ATN2, plotted in Figure 4.2a, showing a much slower rate of improvement, and significantly lower training performance than validation performance (Figure 4.2b). The difference between training performance and validation performance can be attributed to the different scoring metrics. In this case, ATN1 and ATN2 had a much lower prediction performance for class X than the other networks, which was more evident from the balanced accuracy score than the logical accuracy score. Differences in performance can in part be attributed to the stochastic nature of the method. Even with the same hyperparameters and settings, there was no guarantee that performances would be the same, and sometimes the algorithm was unable to escape local optima. Another factor to take into account was the number of trainable parameters in each network. Time complexity for training scaled directly with the number of network parameters, which means the networks with self-attention heads (ATNx) were slower to train, and in most cases took more iterations to reach its highest performance. Therefore, ATN networks were given more training time. FF network training ran for a total of three days, while ATN network training ran for a total of five days. However, in almost all training runs, performance improved the most within the first 1000 training iterations, and continuing training after that point only resulted in slight improvements. We can therefore assume that all networks were given a "fair" time frame to reach an optimal performance.

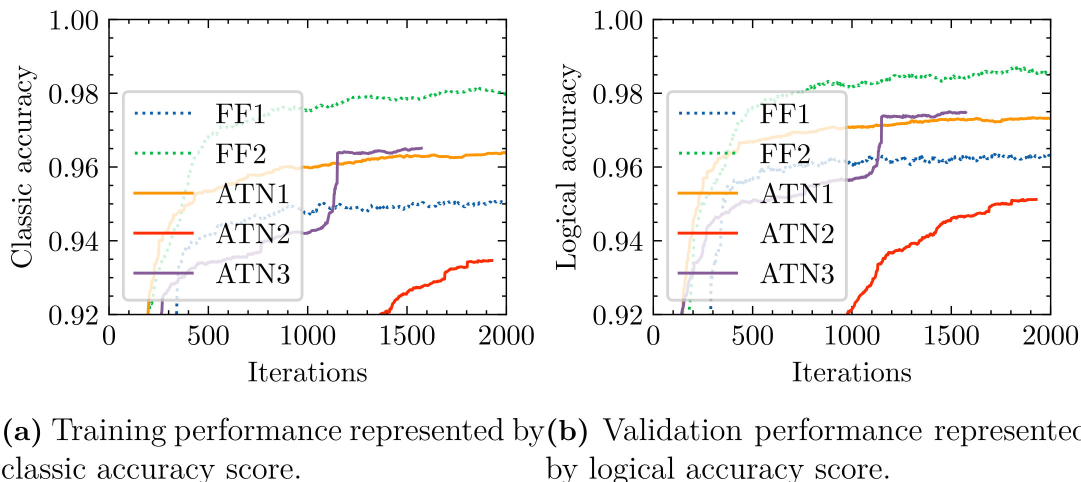


(a) Training performance represented by balanced accuracy score.

(b) Validation performance represented by logical accuracy score.

Figure 4.2: Prediction performances on datasets with fixed error rate $p = 0.001$ for networks trained with balanced accuracy as metric. Training performance shows balanced accuracy for a dataset of 500'000 syndromes sampled from STIM, while validation performance shows logical accuracy for a dataset of 40'000 syndromes.

On the dataset with mixed error probabilities, comparative network performances were largely similar, albeit with lower accuracies for all networks (see figures 4.3 and 4.4). The lower scores can be attributed to the fact that higher error probabilities means syndrome graphs are more likely to contain more nodes, making the classification task more complex.



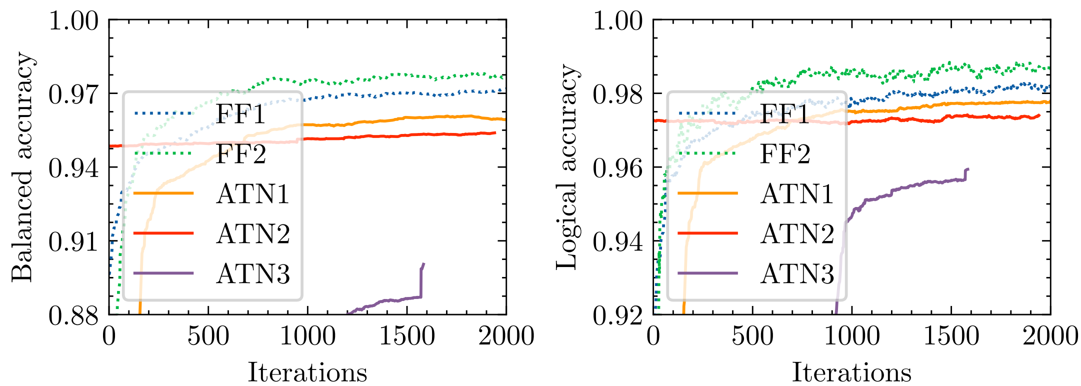
(a) Training performance represented by classic accuracy score.

(b) Validation performance represented by logical accuracy score.

Figure 4.3: Prediction performances on datasets with mixed error rates $p = 0.001-0.005$ for networks trained with classic accuracy as metric. Training performance shows classic accuracy for a dataset of 500'000 syndromes sampled from STIM, while validation performance shows logical accuracy for a dataset of 40'000 syndromes.

Classification performance on training and validation sets seemed to correspond well for all networks, with no sharp drops in validation accuracy that could indicate overfitting to the training set. Therefore, we can assume that the size of the training set was large enough to adequately represent variations in syndromes for the given error probabilities.

The overall performances of the networks do not reveal obvious differences between using classic accuracy or balanced accuracy as training metric. However, when comparing class-wise performance in figure 4.5, some of the behavior discussed in Section 3.2.2 can be observed. While the classification performance on the I and X class are almost equal for balanced accuracy, they differ by 1.6% for classic accuracy. More samples of class X are incorrectly labeled as class I when using classic accuracy, indicating that there could be some bias towards choosing class I. On the flip side, more samples of class I are incorrectly labeled as class X when using balanced accuracy, possibly indicating that balancing the accuracy score gives too much weight to class X. It is possible that performance could be further improved by implementing some alternative form of weighting of scores that takes into account the prior probability of belonging to each class. The apparent trade-off leads to the question, is correct identification more important for one of the classes? In this case, it could be argued that no such distinction exists. If a sample of class X is mislabeled as class I it will not be corrected, leading to a persisting error in logical state measurements. On the other hand, if a sample of class I is mislabeled as class X, a correction will

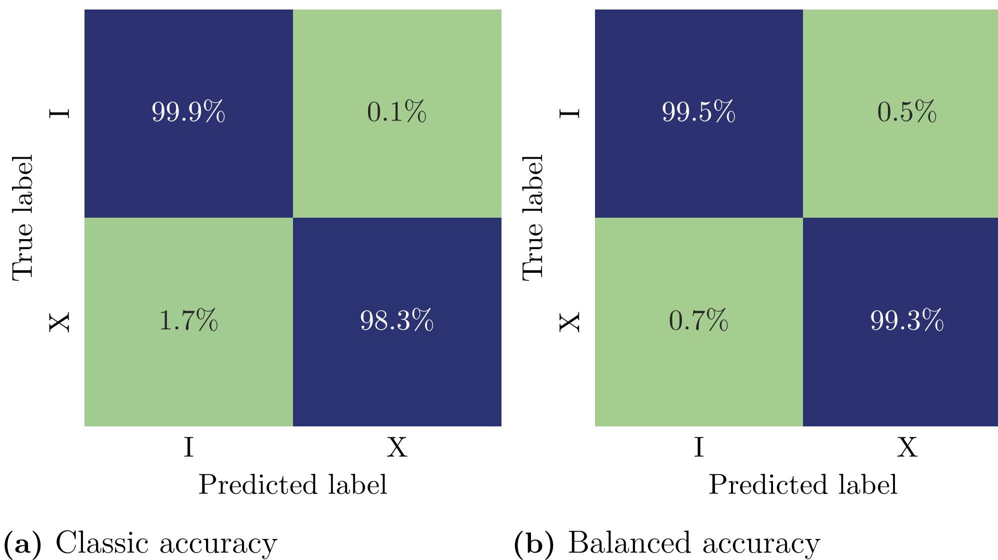


(a) Training performance represented by balanced accuracy score.

(b) Validation performance represented by logical accuracy score.

Figure 4.4: Prediction performances on datasets with mixed error rates $p = 0.001 - 0.005$ for networks trained with balanced accuracy as metric. Training performance shows balanced accuracy for a dataset of 500000 syndromes sampled from Stim, while validation performance shows logical accuracy for a dataset of 40000 syndromes.

be applied unnecessarily, leading to a logical bit flip. The choice of training metric could therefore be made based on which yields the best overall accuracy.



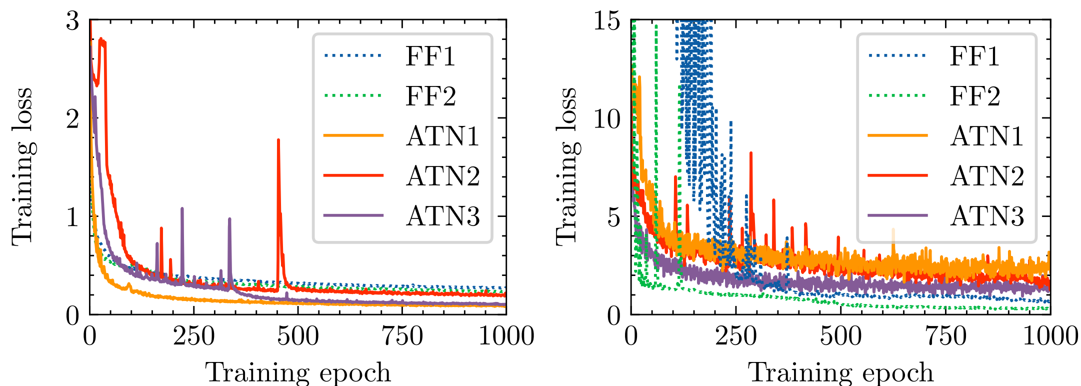
(a) Classic accuracy

(b) Balanced accuracy

Figure 4.5: The class-wise percentage of correct and incorrect predictions when using classic accuracy or balanced accuracy as training metric.

4.1.2 Training with SGD

When using a surrogate loss to train the networks with SGD, the five different network architectures listed in Table 4.1 were used with the two types of datasets described in Section 4.1. The training losses are shown in Figure 4.6, with the



(a) Fixed error rate $p = 0.001$.

(b) Mixed error rate $p = 0.001 - 0.005$.

Figure 4.6: Training loss as a function of training epoch for a fixed error rate (a) and a mixed error rate (b).

networks trained on datasets with a fixed error rate shown in Figure 4.6a and the networks trained on datasets with mixed error rates seen in Figure 4.6b. Apart from a few spikes, the losses approach zero as expected from a network converging towards the optimal set of parameters. The range of loss values is higher for the networks trained on a dataset with mixed error rates, which likely can explained by the increased number of nodes in the syndrome graphs. This probably makes it harder for the networks to compute ideal edge weights initially. All networks appear to converge to minima in the loss function.

Since the surrogate loss described in Section 3.2.1 works by creating pseudo-labels not directly connected to the true labels of error syndromes, it is important to verify that a decrease in the training loss also led to networks that performed better. To do this, the performance of the networks were measured once every epoch on fixed validation datasets independent of the training datasets. The validation datasets were created from simulations with the same error rates as the corresponding training datasets. We expected the NMD to be more accurate for error syndromes with a fewer number of nodes, and created validation sets with 60'000 error syndromes for the networks trained on a fixed error rate of $p = 0.001$ whilst we only used 30'000 error syndromes for the validation set when training the networks on a mixed error rate $p = 0.001 - 0.005$. The number of error syndromes was chosen small enough to not impact the training speed, but large enough to indicate how the networks performance evolved and allow for some statistical stability in the metrics.

In Figure 4.7, the logical accuracy of the NMD is shown as a function of training epoch for the dataset having a fixed error rate $p = 0.001$. The dotted lines indicate heads having feed-forward networks, whilst the solid lines show the networks with self-attention layers. As can be seen in the figure, all networks converged to high logical accuracies. This indicates that the design of the surrogate loss works. The FF2 network appears to have converged slower than the others, and could perhaps have been trained for a longer time. The ATN3 network, the only one having three attention heads (within the self-attention layer) instead of one, reached the highest

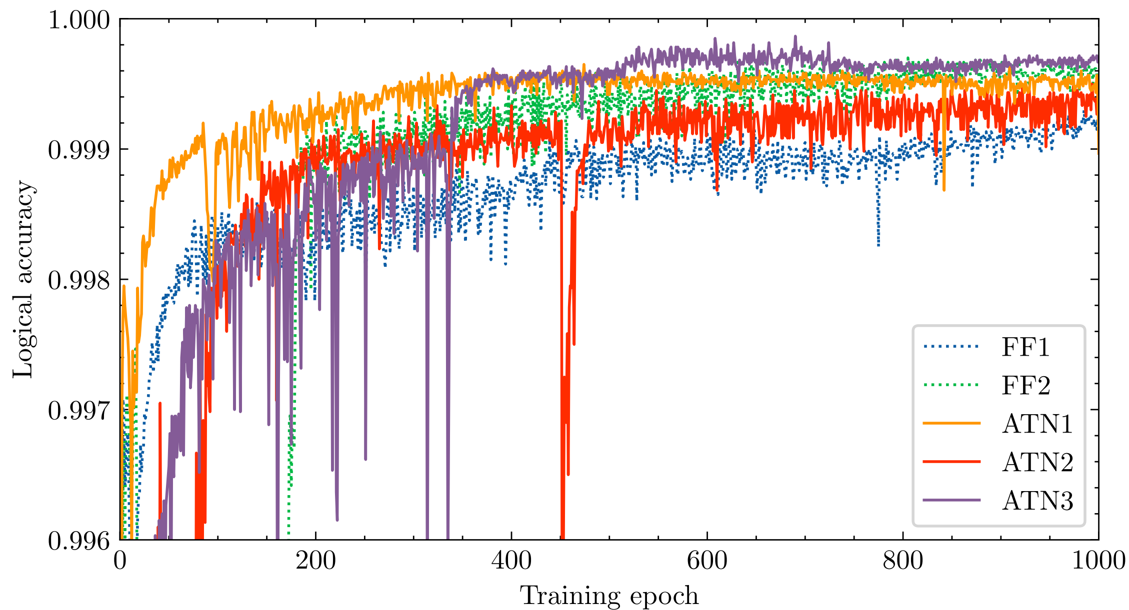


Figure 4.7: Logical accuracy as a function of training epoch when the networks were trained on simulations with a fixed error rate $p = 0.001$. The accuracy is computed for a separate validation set that consisted of the same error syndromes during the entire training process.

logical accuracy.

The logical failure rates for the networks trained (and evaluated) on error syndromes sampled from simulations with mixed error rates are seen as a function of training epoch in Figure 4.8. The peak logical accuracy is lower than in Figure 4.7, which is to be expected as the networks now are evaluated on error syndromes with error rates ranging from $p = 0.001 - 0.005$. What is more interesting is that the feed-forward networks appear to converge slower compared to when they were trained on datasets with fixed error rates, especially the FF1-network. Judging from the slope between epoch 800 and 1000, the FF2 network would likely have reached higher logical accuracies had it been trained longer. This behaviour is not seen for the networks employing self-attention heads, which might indicate that self-attention layers can adapt to syndrome graphs having more nodes better than the linear layers in the feed-forward networks do.

Another difference between the logical accuracy of the ATNx-networks and the FFx-networks is the small jumps appearing, most notably around epoch 500 for ATN2. Training with SGD on a dataset of mixed error rates was slow and the training had to be restarted before the ATNx-networks reached 1000 epochs. When the training restarts, a new validation set that can have a slightly different distribution of error syndromes is generated. This likely explains why the small jumps in the accuracy curves appear.

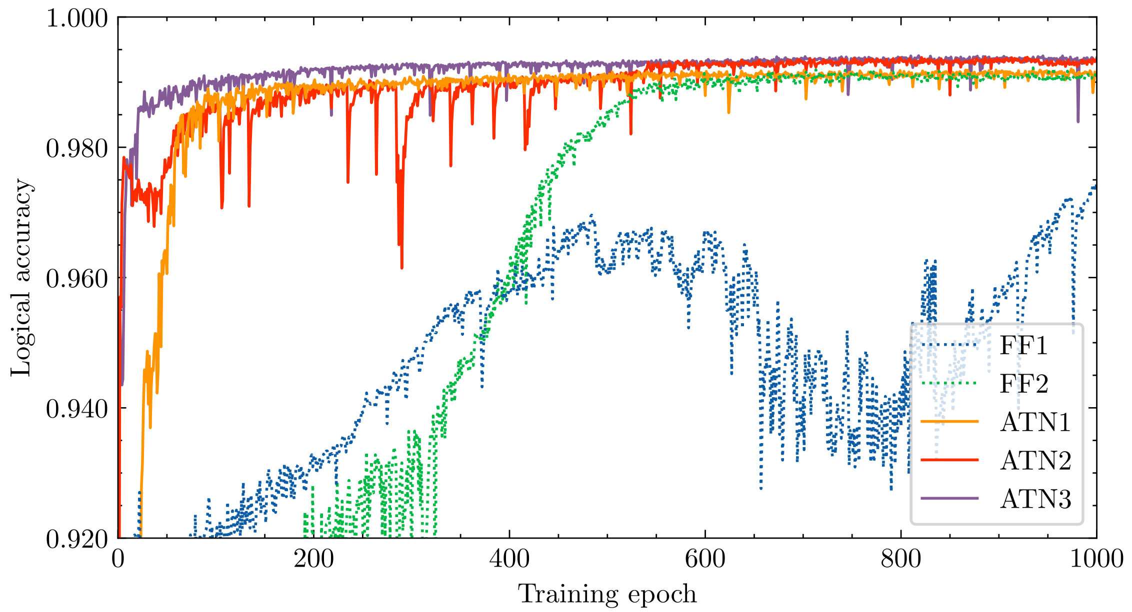


Figure 4.8: Logical accuracy as a function of training epoch when the networks were trained on simulations with a fixed error rate $p = 0.001 - 0.005$. The accuracy is computed for a separate validation set that consisted of the same error syndromes during the course of a training run. However, training with SGD was slow and the training had to be restarted before the three ATNx-networks reached 1000 epochs. Each restart initialise a new validation set, which could have a slightly different distribution of error syndromes. This likely explains the small jumps in logical accuracy seen for the ATNx-networks.

4.2 Decoding accuracy

The validation sets used to evaluate how well the training of the networks converged were not large enough to be statistically significant when doing a final evaluation of the performance of the NMD. Instead, a separate test set consisting of 10^7 error syndromes sampled from a simulation subjected to qubit errors with probability $p = 0.001$ was used to evaluate the final performance of the NMDs. In Figure 4.9,

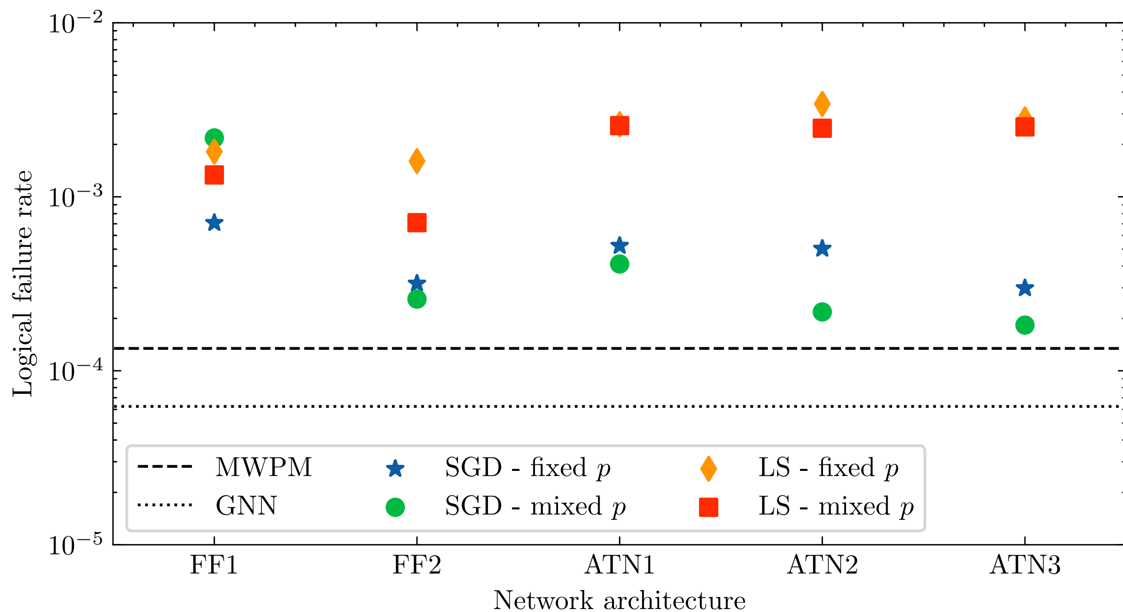


Figure 4.9: The logical failure rate seen as a function of various heads used in the NMD when evaluated on 10^7 syndrome graphs sampled from simulations with error rate $p = 0.001$. Different markers and colors are used to indicate whether the networks were trained with stochastic gradient descent (SGD) or local search (LS), and if a dataset with error syndromes sampled from simulations of fixed error rate $p = 0.001$ or a mixed error rate $p = 0.001 - 0.005$ was used during training. Also note a dashed line that indicates the logical accuracy of a MWPM-decoder (implemented in PyMatching [29]) and a dotted line giving the logical accuracy of the GNN-decoder developed in [38], when evaluated on the same dataset.

the logical failure rate of each version of the NMD that was trained is shown. The markers represent NMDs trained with stochastic gradient descent (SGD) and local search (LS), both when trained on error syndromes sampled for simulations with a fixed error rate $p = 0.001$ and when trained on error syndromes sampled from simulations with mixed error rate $p = 0.001 - 0.005$. The dashed and dotted line show how well a MWPM decoder implemented in PyMatching [29] and the GNN decoder from [38] performs on the same error syndromes respectively.

Except from a single outlier given by the green dot for FF1, the networks that were trained using error syndromes sampled from simulations with mixed error rates perform slightly better. The FF1 network showed a steep gradient between epochs 800 and 1000 in Figure 4.8, and it appears the training has not yet converged. This

would explain why it shows a higher logical failure rate than expected from the general trend.

Another trend is that the networks trained with SGD performed better than the ones trained using LS. The difference appears to be slightly smaller for the FFx-networks, which could indicate that LS works better for smaller networks. In terms of the number of parameters, the ATNx-networks are much larger which might make it harder for the gradient-free optimization routine implemented in LS to converge.

The best result achieved by the NMD was for the ATN3 network trained using SGD and a dataset with error syndromes sampled from mixed simulations, resulting in a logical failure rate of 1.83×10^{-4}. This is is not far from the performance of the MWPM decoder, but the pure GNN decoder outperforms our results by some margin. When comparing to the MWPM decoder, it is important to note that the NMD has learned the underlying noise distribution causing the errors directly from the dataset used in training. This is not the case for the MWPM decoder, which utilizes the same noise model as Stim has employed in the simulations when decoding the error syndromes.

5

Discussion

To conclude our work on the neural matching decoder (NMD), the sections below discuss ethical considerations, reflect on the results and suggest future outlooks.

5.1 Ethical considerations

Both quantum computing and machine learning can have a great societal impact, and the ethical implications of our research are important to address. The last years have for instance shown how deepfakes, synthetic text, images or videos generated with machine learning, can pose a security threat by its power to mimic reality. In a video released shortly after Russia's invasion of Ukraine in March 2022, Ukraine's President Volodymyr Zelensky was seen to ask his fellow countrymen to put down arms and surrender to Russia. The video was not a real recording of Zelensky, but a deepfake used to deceive Ukrainians for the benefit of Russia's ambitions [11]. In 2022 the deepfake could be identified from an odd pixelation, but it becomes increasingly hard to distinguish them from reality. Whenever we develop machine learning algorithms, we must be vary of its potential to do harm.

However, the graph neural network developed in this thesis cannot be used for tasks other than quantum error correction. A more relevant ethical implication is how quantum error correction decoders are an important step towards enabling fault-tolerant quantum computers that can run complex quantum algorithms. Given a fault-tolerant quantum computer, Shor's algorithm will enable its user to break many common encryption protocols which will have a severe impact on information security [32]. To mitigate these risks, it is important to develop new encryption protocols that are more robust towards quantum algorithms.

Another ethical implication of both quantum computing and machine learning is the potential environmental impacts. Estimates show that training a large natural language processing model can have a carbon footprint equal to that of 125 round-trip flights between New York and Beijing [18]. It has also been suggested that simulating large quantum circuits comes with a high carbon footprint, significantly higher than that of training a standard machine learning model [39]. However, shifting the perspective, we could instead argue that quantum computing and machine learning can give us the technological breakthroughs needed to mitigate climate change.

5.2 Conclusion and future improvements

This thesis presented a novel quantum error correction decoder, the neural matching decoder. The decoder combines a graph neural network (GNN) with a minimum-weight perfect matching (MWPM) algorithm and we show how it is capable of decoding the rotated surface code of size $d = 5$ for $d_t = 5$ time steps with a relatively high logical accuracy. The performance cannot yet match the benchmark MWPM decoder, or a decoder based on a pure GNN, but it shows potential with a best logical failure rate of 1.83×10^{-4}. It was possible to train the networks using the gradient-free method local search, but the more conventional approach based on a pseudo-loss and stochastic gradient descent still gave slightly better results. Furthermore, it is clear that training the networks on error syndromes sampled from simulations with a mixed error rate lead to lower logical failure rates even when evaluated on a fixed error rate at the lowest end of the range. A possible explanation is that simulations with higher error rates give more syndrome graphs with a higher number of nodes, allowing the network to adapt to these better. Simulations with lower error rates can still yield syndrome graphs with a high number of nodes, but it is not as likely to happen compared to the simulations with higher error rates. Error syndromes sampled from simulations with a wider range of error rates will expose the network to syndrome graphs with more varied sizes, allowing it to generalise better in training.

To improve the NMD, we suggest two major approaches. The first one is related to the construction of the syndrome graphs. The syndrome graphs are created based on the error syndromes, but their construction is relatively complex and require two types of edges for every pair of connected nodes. With the present setup in the matching module, edges with two distinct types are necessary to be able to determine which equivalence class the error syndrome belongs to. This is done by computing if an odd or even number of the matched edges belong to the type of edge that represents an error chain extending to the boundaries. It might be that this construction is unnecessarily complicated. An alternative could be to add two artificial nodes that represent the identity class and a logical operator X_L or Z_L respectively. The artificial nodes would be connected to all others with some constant initial weight. The equivalence class of an error syndrome could then possibly be determined by checking if the decoder is more likely to create a matching to the identity node or the one related to a logical operator.

The second area for improvement relates to the software implementation of the network training and the MWPM algorithm. Usually all computations when training a neural network are done in parallel, utilizing GPU-acceleration to feed batches of data through the network. However, the neural matching decoder relies on a MWPM algorithm (Blossom V [36]) that runs on a single CPU-core. This meant that whilst the forward pass and the backward propagation in the network ran in parallel on the GPU, the matching module had to move all syndrome graphs to the CPU and process them one at a time before computing a loss or predicting an equivalence class. In practice this gave a significant computational overhead which forced us to limit the size of the training datasets. Future developments of the neural matching decoder might consider options that can mitigate the bottleneck of the MWPM

algorithm. A potential solution could be to run the MWPM for several syndrome graphs at a time. Since different syndrome graphs do not share any edges, the MWPM algorithm cannot match edges across graphs and it should be possible to parse which subset of matched edges that belong to which graph. This might limit the number of iterative steps needed on the CPU, thus lowering the computational time.

An interesting direction for future works could be to explore how well a NMD could scale to larger code sizes. One of the motivations behind the NMD was that the combination of a data-driven approach based on a GNN and a MWPM algorithm could maintain a high accuracy with a limited computational cost. This remains an open question, as this work primarily shows how its possible to achieve a working implementation of the NMD for a specific code size $d = 5$. A relevant question is how well the NMD will perform when decoding larger code sizes whilst keeping its computational size fixed (at least in terms of network parameters). This is going to be important in practical realisations of fault-tolerant quantum computers.

Bibliography

[1] R. Acharya, I. Aleiner, R. Allen, and OTHERS. Suppressing quantum errors by scaling a surface code logical qubit. 2022. URL https://arxiv.org/pdf/2207.06431.pdf.

[2] A. Aly, G. Guadagni, and J. B. Dugan. Derivative-free optimization of neural networks using local search. In *2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pages 0293–0299, 2019. doi: 10.1109/UEMCON47517.2019.8993007.

[3] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574 (7779):505–510, Oct. 2019. ISSN 1476-4687. doi: 10.1038/s41586-019-1666-5. URL http://dx.doi.org/10.1038/s41586-019-1666-5.

[4] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. 2016. URL http://arxiv.org/pdf/1607.06450.

[5] J. Bausch, A. W. Senior, F. J. H. Heras, and OTHERS. Learning to decode the surface code with a recurrent, transformer-based neural network. 2023. URL https://arxiv.org/pdf/2310.05900.pdf.

[6] A. Berroukham, K. Housni, and M. Lahraichi. Vision transformers: A review of architecture, applications, and future directions. In *2023 7th IEEE Congress on Information Science and Technology (CiSt)*. IEEE, Dec. 2023. doi: 10.1109/cist56084.2023.10410015. URL http://dx.doi.org/10.1109/CiSt56084.2023.10410015.

[7] D. Bluvstein, S. J. Evered, A. A. Geim, S. H. Li, H. Zhou, et al. Logical quantum processor based on reconfigurable atom arrays. *Nature*, 626(7997): 58–65, dec 2023. ISSN 1476-4687. doi: 10.1038/s41586-023-06927-3. URL http://dx.doi.org/10.1038/s41586-023-06927-3.

[8] H. Bombin and M. A. Martin-Delgado. Optimal resources for topological two-dimensional stabilizer codes: Comparative study. *Physical Review A*, 76(1), July 2007. ISSN 1094-1622. doi: 10.1103/physreva.76.012305. URL http://dx.doi.org/10.1103/PhysRevA.76.012305.

[9] S. Bravyi, M. Suchara, and A. Vargo. Efficient algorithms for maximum likelihood decoding in the surface code. *Physical Review A*, 90(3), Sept. 2014. ISSN

1094-1622. doi: 10.1103/physreva.90.032326. URL http://dx.doi.org/10.1103/PhysRevA.90.032326.

[10] C. D. Bruzewicz, J. Chiaverini, R. McConnell, and J. M. Sage. Trapped-ion quantum computing: Progress and challenges. *Applied Physics Reviews*, 6(2), May 2019. ISSN 1931-9401. doi: 10.1063/1.5088164. URL http://dx.doi.org/10.1063/1.5088164.

[11] E. Busch and J. Ware. The weaponisation of deepfakes. Dec. 2023.

[12] T. Cai, S. Luo, K. Xu, D. He, T.-Y. Liu, and L. Wang. Graphnorm: A principled approach to accelerating graph neural network training. 2020. URL http://arxiv.org/pdf/2009.03294.

[13] A. M. Dalzell, S. McArdle, M. Berta, and OTHERS. Quantum algorithms: A survey of applications and end-to-end complexities. 2023. URL http://arxiv.org/pdf/2310.03011.pdf.

[14] M. Dehghani, J. Djolonga, B. Mustafa, P. Padlewski, J. Heek, J. Gilmer, A. Steiner, M. Caron, R. Geirhos, I. Alabdulmohsin, and OTHERS. Scaling vision transformers to 22 billion parameters. 2023. URL http://arxiv.org/pdf/2302.05442.

[15] N. Delfosse and N. H. Nickerson. Almost-linear time decoding algorithm for topological codes. *Quantum*, 5:595, December 2021. ISSN 2521-327X. doi: 10.22331/q-2021-12-02-595. URL http://dx.doi.org/10.22331/q-2021-12-02-595.

[16] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill. Topological quantum memory. *Journal of Mathematical Physics*, 43(9):4452–4505, August 2002. ISSN 1089-7658. doi: 10.1063/1.1499754. URL http://dx.doi.org/10.1063/1.1499754.

[17] D. Deutsch. Quantum theory, the church–turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 400(1818):97–117, July 1985.

[18] P. Dhar. The carbon impact of artificial intelligence. *Nature Machine Intelligence*, 2:423–425, Aug. 2020.

[19] E. Farhi, J. Goldstone, S. Gutmann, J. Lapan, A. Lundgren, and D. Preda. A quantum adiabatic evolution algorithm applied to random instances of an np-complete problem. *Science*, 292(5516):472–475, Apr. 2001. ISSN 1095-9203. doi: 10.1126/science.1057726. URL http://dx.doi.org/10.1126/science.1057726.

[20] R. P. Feynman. Simulating physics with computers. *International journal of theoretical physcis*, 21:467–488, June 1982.

[21] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A*, 86(3),

sep 2012. doi: 10.1103/physreva.86.032324. URL https://doi.org/10.1103%2Fphysreva.86.032324.

[22] C. Gidney. Stim: a fast stabilizer circuit simulator. *Quantum*, 5:497, jul 2021. ISSN 2521-327X. doi: 10.22331/q-2021-07-06-497. URL https://doi.org/10.22331/q-2021-07-06-497.

[23] D. Gottesman. Stabilizer codes and quantum error correction. 1997. URL https://arxiv.org/pdf/quant-ph/9705052.pdf.

[24] D. Gottesman. The heisenberg representation of quantum computers. 1998. URL http://arxiv.org/pdf/quant-ph/9807006.

[25] D. Gottesman. An introduction to quantum error correction and fault-tolerant quantum computation. 2009. URL http://arxiv.org/pdf/0904.2557.pdf.

[26] K. Hammar, A. Orekhov, P. W. Hybelius, A. K. Wisakanto, B. Srivastava, A. F. Kockum, and M. Granath. Error-rate-agnostic decoding of topological stabilizer codes. *Physical Review A*, 105(4), Apr. 2022. ISSN 2469-9934. doi: 10.1103/physreva.105.042616. URL http://dx.doi.org/10.1103/PhysRevA.105.042616.

[27] W. Hariri. Unlocking the potential of chatgpt: A comprehensive exploration of its applications, advantages, limitations, and future directions in natural language processing. 2023. URL http://arxiv.org/pdf/2304.02017.

[28] H. Hewamalage, C. Bergmeir, and K. Bandara. Recurrent neural networks for time series forecasting: Current status and future directions. *International Journal of Forecasting*, 37(1):388–427, Jan. 2021. ISSN 0169-2070. doi: 10.1016/j.ijforecast.2020.06.008. URL http://dx.doi.org/10.1016/j.ijforecast.2020.06.008.

[29] O. Higgott. Pymatching: A python package for decoding quantum codes with minimum-weight perfect matching. 2021. URL http://arxiv.org/pdf/2105.13082.

[30] O. Higgott and C. Gidney. Sparse blossom: correcting a million errors per core second with minimum-weight matching. 2023. URL http://arxiv.org/pdf/2303.15933.pdf.

[31] O. Higgott, T. C. Bohdanowicz, A. Kubica, and OTHERS. Improved decoding of circuit noise and fragile boundaries of tailored surface codes. 2022. URL http://arxiv.org/pdf/2203.04948.pdf.

[32] M. Kaplan, G. Leurent, A. Leverrier, and M. Naya-Plasencia. *Breaking Symmetric Cryptosystems Using Quantum Period Finding*, page 207–237. Springer Berlin Heidelberg, 2016. ISBN 9783662530085. doi: 10.1007/978-3-662-53008-5_8. URL http://dx.doi.org/10.1007/978-3-662-53008-5_8.

[33] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. 2014. URL http://arxiv.org/pdf/1412.6980.

[34] A. Kitaev. Fault-tolerant quantum computation by anyons. *Annals of Physics*, 303(1):2–30, January 2003. ISSN 0003-4916. doi: 10.1016/s0003-4916(02)00018-0. URL http://dx.doi.org/10.1016/S0003-4916(02)00018-0.

[35] M. Kjaergaard, M. E. Schwartz, J. Braumüller, P. Krantz, J. I.-J. Wang, S. Gustavsson, and W. D. Oliver. Superconducting qubits: Current state of play. *Annual Review of Condensed Matter Physics*, 11(1):369–395, March 2020. ISSN 1947-5462. doi: 10.1146/annurev-conmatphys-031119-050605. URL http://dx.doi.org/10.1146/annurev-conmatphys-031119-050605.

[36] V. Kolmogorov. Blossom v: a new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation*, 1(1):43–67, apr 2009. ISSN 1867-2957. doi: 10.1007/s12532-009-0002-8. URL http://dx.doi.org/10.1007/s12532-009-0002-8.

[37] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver. A quantum engineer's guide to superconducting qubits. *Applied Physics Reviews*, 6(2), June 2019. ISSN 1931-9401. doi: 10.1063/1.5089550. URL http://dx.doi.org/10.1063/1.5089550.

[38] M. Lange, P. Havström, B. Srivastava, and OTHERS. Data-driven decoding of quantum error correcting codes using graph neural networks. 2023. URL https://arxiv.org/pdf/2307.01241.pdf.

[39] J. Li, Q. Guan, D. Tao, and W. Jiang. Carbon emissions of quantum circuit simulation: More than you would think. 2023. URL http://arxiv.org/pdf/2307.05510.

[40] B. Mehlig. *Machine Learning with Neural Networks: An Introduction for Scientists and Engineers*. Cambridge University Press, Oct. 2021. ISBN 9781108494939. doi: 10.1017/9781108860604. URL http://dx.doi.org/10.1017/9781108860604.

[41] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. 2018. URL http://arxiv.org/pdf/1810.02244.

[42] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

[43] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, June 2012. ISBN 9780511976667. doi: 10.1017/cbo9780511976667. URL http://dx.doi.org/10.1017/CBO9780511976667.

[44] R. Rojas. *The Backpropagation Algorithm*, page 149–182. Springer Berlin Heidelberg, 1996. ISBN 9783642610684. doi: 10.1007/978-3-642-61068-4_7. URL http://dx.doi.org/10.1007/978-3-642-61068-4_7.

[45] J. J. Sakurai and J. Napolitano. *Modern Quantum Mechanics*. Cambridge University Press, 3 edition, 2020.

[46] P. W. Shor. Scheme for reducing decoherence in quantum computer memory. *Physical Review A*, 52(4):R2493–R2496, oct 1995. ISSN 1094-1622. doi: 10.1103/physreva.52.r2493. URL http://dx.doi.org/10.1103/PhysRevA.52.R2493.

[47] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5): 1484–1509, oct 1997. ISSN 1095-7111. doi: 10.1137/s0097539795293172. URL http://dx.doi.org/10.1137/S0097539795293172.

[48] S. Sobczak, R. Kapela, K. McGuinness, A. Swietlicka, D. Pazderski, and N. E. O'Connor. Restricted boltzmann machine as an aggregation technique for binary descriptors. *The Visual Computer*, 37(3):423–432, Dec. 2019. ISSN 1432-2315. doi: 10.1007/s00371-019-01782-8. URL http://dx.doi.org/10.1007/s00371-019-01782-8.

[49] A. M. Stephens. Fault-tolerant thresholds for quantum error correction with the surface code. *Physical Review A*, 89(2), Feb. 2014. ISSN 1094-1622. doi: 10.1103/physreva.89.022321. URL http://dx.doi.org/10.1103/PhysRevA.89.022321.

[50] Y. Tomita and K. M. Svore. Low-distance surface codes under realistic quantum noise. *Physical Review A*, 90(6), Dec. 2014. ISSN 1094-1622. doi: 10.1103/physreva.90.062320. URL http://dx.doi.org/10.1103/PhysRevA.90.062320.

[51] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. 2017. URL http://arxiv.org/pdf/1706.03762.

[52] X. Zhao, L. Wang, Y. Zhang, X. Han, M. Deveci, and M. Parmar. A review of convolutional neural networks in computer vision. *Artificial Intelligence Review*, 57(4), Mar. 2024. ISSN 1573-7462. doi: 10.1007/s10462-024-10721-6. URL http://dx.doi.org/10.1007/s10462-024-10721-6.

[53] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020. ISSN 2666-6510. doi: 10.1016/j.aiopen.2021.01.001. URL http://dx.doi.org/10.1016/j.aiopen.2021.01.001.

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