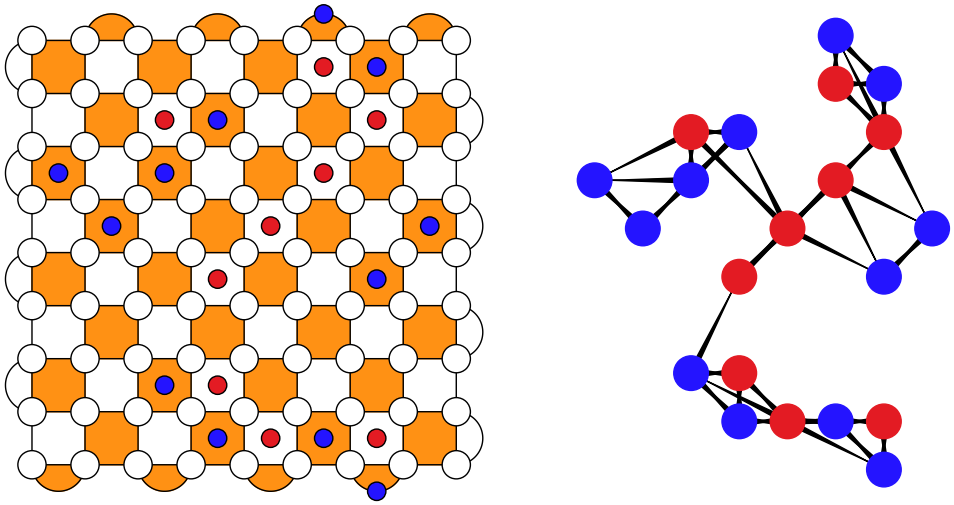


Machine Learning Assisted Quantum Error Correction Using Scalable Neural Network Decoders

Decoding Surface Code Syndromes with Graph and Convolutional Neural Networks

Master's thesis in Physics

PONTUS HAVSTRÖM
OLIVIA HEUTS

DEPARTMENT OF PHYSICS
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Master's thesis 2023

Machine Learning Assisted Quantum Error Correction Using Scalable Neural Network Decoders

Decoding Surface Code Syndromes
with Graph and Convolutional Neural Networks

PONTUS HAVSTRÖM
OLIVIA HEUTS

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2023

Machine Learning Assisted Quantum Error Correction Using Scalable Neural Network Decoders
Decoding Surface Code Syndromes with Graph and Convolutional Neural Networks
PONTUS HAVSTRÖM
OLIVIA HEUTS

Supervisor and Examiner: Mats Granath

Master's Thesis 2023
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Illustration of a syndrome on the rotated surface code with code distance $d = 9$, and a graph representation of the syndrome.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2023

Machine Learning Assisted Quantum Error Correction Using Scalable Neural Network Decoders
Decoding Surface Code Syndromes with Graph and Convolutional Neural Networks
Pontus Havström & Olivia Heuts
Department of Physics
Chalmers University of Technology

Abstract

A necessary condition for fault-tolerant quantum computers is the implementation of quantum error correction, as the sensitive nature of quantum technology causes unavoidable errors on qubits. Topological stabilizer codes, such as the surface code and its variations, are promising candidates for near term implementations of quantum error correcting codes. In surface codes, multiple physical qubits are encoded to represent a single logical qubit with a higher tolerance for errors than the individual physical qubits. Errors on data qubits cannot be measured directly, and have to be corrected based on incomplete observations of the system from ancilla qubit measurement syndromes. Classical algorithms called decoders are used to determine correction operators based on the syndromes, which is a non-trivial and computationally expensive task. In practice, the error decoding must be fast, and as such it is of interest to develop decoders that rapidly determine correction operations while still remaining sufficiently accurate.

Decoders based on neural networks have been shown to yield high decoding accuracy for small distance surface codes, while also having fast decoding time once trained. Many such decoders are however not necessarily scalable and have been designed for a specific code size. In this thesis, we develop two types of neural network based decoders using the deep learning architectures Graph Neural Networks (GNN) and Convolutional Neural Networks (CNN), both of which in principle allow for decoding arbitrarily large codes. We apply the decoders to the rotated surface code under depolarizing noise with perfect syndrome measurements, and evaluate their performance based on their accuracy, computational speed and scalability to large code distances. We show that the the decoders perform on par with the commonly used Minimum Weight Perfect Matching (MWPM) decoder at small codes and low physical error rates, with the CNN decoder outperforming the MWPM decoder for code distance $d = 7$. We also find that using a sparse graph representation of syndromes yields a favorable computational complexity for the GNN decoder on large-distance codes.

Keywords: quantum computing, quantum error correction, topological stabilizer codes, surface codes, neural network decoders, convolutional neural networks, graph neural networks.

Acknowledgements

We would like to extend our deepest gratitude towards our supervisor Mats Granath who introduced us to this project and supported us all the way to our goal, gladly engaging in discussion and answering questions arising on the way.

We also thank Karl Hammar for offering assistance in running his existing decoder which we utilized in this project.

Furthermore, computations were facilitated by resources provided by the Swedish National Infrastructure for Computing (SNIC) at Chalmers Centre for Computational Science and Engineering (C3SE).

Finally, we are eternally grateful for our families and friends who have shown interest in our work and offered invaluable support and encouragement along the way.

Pontus Havström & Olivia Heuts, Gothenburg, November 2022

Contents

List of Figures

List of Tables

1

Introduction

Quantum computers have the potential of solving computational problems which are intractable when using a classical computer. This is achieved by devising algorithms that utilize the principles of superposition, entanglement and interference in two-level quantum mechanical systems: quantum bits (qubits). Examples of problems where quantum computers offer such an advantage are the famous Shor's algorithm for integer factorization [1], searching an unstructured list [2], and perhaps most notably the task of simulating other quantum systems [3], which quickly becomes difficult for a classical computer. While great strides of progress are being made towards building functional quantum computers, some major obstacles still remain — one of these being the challenge of handling noise [4]. For it to be possible to perform operations and measurements with qubits in practice, as well as due to limitations when manufacturing and preserving physical qubits, some unwanted interaction with the environment of the system is inevitable, introducing errors to the quantum information.

However, performing quantum computations in the presence of noise can be done by implementing so called quantum error correcting codes. In general, this is done by encoding one logical qubit using many physical qubits in ways that protect against errors while still being able to perform operations and measurements on the logical qubit [5]. A promising type of error correction code for attempting to build fault-tolerant quantum computers in the near future are variations of *surface codes*, where qubits are encoded on a two-dimensional lattice. This type of code has the benefit of tolerating a relatively high error probability, while also being practical for experimental implementation due to its two-dimensional physical configuration [6].

When using a surface code, errors are indirectly observed as a syndrome, corresponding to several possible errors that are affecting the system. For a given syndrome, a sequence of decoding operations should then be determined which correct the underlying error [7]. One can use decoders based on directly determining the most likely error for a measured syndrome, but this approach comes at the cost of computational complexity. In practical implementation of quantum computers, the error decoding must be fast, and as such it is of interest to develop decoders that rapidly determine correction operations while still remaining sufficiently accurate. Additionally, larger codes can protect against a larger number of errors (for sufficiently low error probabilities) [8], meaning that it is important that the error decoder scales well with the size of the code.

Utilizing machine learning can potentially pave the way for fast and accurate error decoders, by training models to determine a correction operation for a given syndrome. Additionally, the syndromes in surface codes can be interpreted as graphs, inviting the idea of using the class of Graph Neural Networks (GNNs) which has recently been of great interest across multiple fields, showing promise in several other applications of geometric deep learning [9]. Another class of machine learning models that may be well-suited for decoding are Convolutional Neural Networks (CNNs), which have proven effective visual classification properties, and can be adapted to allow for inputs of arbitrary size. As such, GNNs and CNNs can possibly provide fast error decoding in surface codes of increasing size – an important step towards realising scalable quantum computing in practice.

1.1 Thesis Aim and Objectives

The objective of this thesis project is to develop error correction decoders for syndromes in topological error correcting codes, with the aim of providing insight in ways to achieve fast and accurate error correction as a step towards practical fault-tolerant quantum computing. While previous studies have shown promising results for decoders based on a wide variety of artificial neural networks [10–16], what models are best suited for quantum error decoding still remains an open question. A key challenge has also been to produce decoders that generalize well to larger code distances [17]. For this reason, we will restrict our study to decoders based on two specific classes of artificial neural networks that have the potential to be invariant to the size of the code: Convolutional Neural Networks (CNNs) and Graph Neural Networks (GNNs).

Thus, a secondary aim is to assess the potential of using these models as decoders by evaluating their performance on the basis of their computational efficiency, their accuracy, and their scalability in relation to the size of the error correcting code to which the decoder is applied. A positive conclusion in regards to any of these performance metrics would be of interest for future studies of machine learning assisted quantum error decoders. Additionally, as the two classes of networks will be evaluated in tandem, this study offers the possibility to make comparisons between different neural network decoders under similar simulation conditions.

In conclusion, the objective of this thesis is concretized in the following research question:

RQ. Can artificial neural networks perform error decoding for simple noise models on the rotated surface code that is accurate, fast and that generalizes to large code distances by utilising the geometric structure of syndromes?

This question will be explored based on the two aforementioned classes of neural networks, CNNs and GNNs, where the geometric structure of syndromes is utilized but represented in different ways. This is summarized in the following subquestions:

1. How well can the decoding be performed by representing syndromes as *grids* and assign their equivalence classes using a *Fully Convolutional Network*?

2. How well can the decoding be performed by representing syndromes as *graphs* and assign their equivalence classes using a *Graph Neural Network*?

2

Quantum Computing

The computers of today are under constant development to become smaller and faster, but this process is approaching its physical limits as computer parts are getting closer to the size of an atom. This entails certain problems, for instance, the transistors which makes up the gates of a computer that processes data will eventually reach a size where electrons are at risk of tunnelling right through them. In a quantum computer on the other hand, these quantum properties are instead used to its advantage.

The foremost advantage of quantum computing however is ultimately the potential computational speed for certain problems. Quantum computers have the possibility of solving otherwise intractable computing problems. This is achieved by devising algorithms that utilize quantum mechanical phenomena such as superposition, entanglement and interference in two-level quantum mechanical systems known as quantum bits or "qubits".

In this chapter we will deeper explain relevant concepts regarding quantum computing such as qubits, operators, error correction and other useful knowledge to comprehend the project.

2.1 Quantum Bits and Pauli Operators

A qubit is the quantum computing equivalence of a bit in a classical computer. Bit is a portmanteau of the words "**bi**nary" and "digi**t**" and represents information binary in a classical computer as either 0 or 1, thus a classical bit will always be in one of two states. Qubits however, work in fundamentally different ways.

The two quantum states corresponding to 0 and 1 in a classical bit are:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{2.1}$$

However, the quantum mechanical phenomena of superposition makes it possible for a quantum system to be in a combination of these two states at the same time. This superposition of states is a linear combination of the so called "basis states" written:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{2.2}$$

under the condition that:

$$||\alpha||^2 + ||\beta||^2 = 1 \tag{2.3}$$

where α and β are complex numbers and each term the probability for the state to collapse into either $|0\rangle$ or $|1\rangle$ respectively when measured.

To modify the qubits different operations are performed on them. The most central operators for a single qubit are the identity matrix together with the three Pauli matrices:

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2.4}$$

Any unitary, or probability conserving, transformation can be expressed in terms of these operators.

The identity matrix operates by returning the input unchanged. When applied on the basis states follows: $I|0\rangle = |0\rangle$ or $I|1\rangle = |1\rangle$. This can be verified with vector operations:

$$I(\alpha|0\rangle + \beta|1\rangle) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \alpha|0\rangle + \beta|1\rangle \tag{2.5}$$

The X operation is referred to as a bit-flip since it flips the qubit state $|0\rangle$ to $|1\rangle$ and vice versa. This can be seen with the following equation

$$X(\alpha|0\rangle + \beta|1\rangle) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix} = \beta|0\rangle + \alpha|1\rangle \tag{2.6}$$

What happens is that the probabilities for the two states switch with each other. On the basis states this yields: $X|0\rangle = |1\rangle$ or $X|1\rangle = |0\rangle$.

The Z operator performs a so called phase-flip as follows:

$$Z(\alpha|0\rangle + \beta|1\rangle) = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ -\beta \end{bmatrix} = \alpha|0\rangle - \beta|1\rangle \tag{2.7}$$

This negative term might seem insignificant since $||-\beta||^2 = ||\beta||^2$ and thus the input state and the output state will collapse into the same state. But as long as the qubit isn't measured, this negative sign can still affect the computations involving multiple qubits and is thus important to keep track of. $Z|0\rangle = |0\rangle$, $Z|1\rangle = -|1\rangle$.

These operators are all self-inverse, meaning that $X^2 = Y^2 = Z^2 = I$. Thus adding the same operator twice to a state will return it to it's original state.

Furthermore it is possible for multiple qubits to be entangled with each other. The qubits are thus not independent of each other like in a classical computer, but rather part of a common large state, a linear combination of all possible states of multiple qubits. Two entangled qubits no longer have independent probabilities of states,

but instead a probability distribution of all possible combinations of states between the two of them.

This means that the number of possible states for the system increase exponentially with the number of entangled qubits since the number doubles for every added qubit. The number of possible states in the probability distribution becomes 2^n for two-level systems where n is the number of qubits. Entanglement means that two systems that are too far apart to influence each other can still exert correlated behaviour though their behaviours are individually random. Changing the state of an entangled qubit will immediately change the state of the paired qubit, thus entanglement can be used for transferring information with qubits regardless of the physical proximity of the qubits.

It is then quantum interference between the qubits that makes this superposition of states actually useful in the computer. The state of a qubit is described by a quantum wave function and is thus subject to constructive and destructive interference. The last step of a computation is to measure the results, collapsing the superposition state into definitive information in one of the possible basis states. In a quantum computing algorithm, prior to measurement, interference can then be used to weaken the amplitude of the state in some basis states and increase it in others by applying certain gates, which allows for biasing of the measurement towards desired states. Together with the possible entanglement of an exponential number of basis states, this allows for devising algorithms that utilize these properties of a quantum system to solve certain problems efficiently in ways that a classical computer cannot.

2.2 Error Correction

Every computer is subject to the risk of errors during a computation and quantum computers are certainly not an exception. One of the biggest obstacles within quantum computing development today is their extreme sensitivity to noise. Quantum systems are very vulnerable since quantum mechanical properties kick in first at quite extraordinary circumstances such as temperatures close to absolute zero or extreme pressures only encountered in outer space. This makes the conditions for quantum technology rather unstable and quantum computers are thus extra prone to errors from surrounding noise. The probability p of an error occurring during an operation in a classical computer is approximated to $p \approx 10^{-18}$ while the best quantum computers today can reach an error rate approximated to $p \approx 10^{-2}$ [5]. This number likely changes rapidly with research and does not apply to all quantum computers. Quantum error correction is thus essential to achieve fault-tolerant quantum computation.

A further obstacle with quantum computing however is that it is not possible to examine whether a qubit has been compromised mid computation. Quantum mechanics states that when a quantum system is observed its states "collapse" into a basis state of the measurement and the computation is ruined. Additionally, due to the no-cloning theorem it is not possible to duplicate a qubit state and instead examine a clone not part of the computation to try and circumvent the original

problem. This theorem states that there exists no such operation U that changes a known state ϕ into a copy of an unknown state ψ[5]:

$$U|\varphi\rangle|\psi\rangle = |\psi\rangle|\psi\rangle \tag{2.8}$$

Thus there is a need for other means of checking on the qubits.

A quantum error is when a qubit unintentionally is subject to an operation (equation 2.4). Thus the possible errors to which a qubit can be subjected to corresponds to the operators that can be applied to them. While a classical computer is only at risk for a bit-flip error, also known as an X-error as in equation (2.6), a quantum computer additionally faces the risk of a phase-flip error or Z-error as in equation (2.7). An X-error and a Z-error occurring on the same qubit results in a Y-error. In addition to this the error can also cause a change in α and β resulting in an infinite amount of possible errors. However there is a theorem claiming that being able to correct the X- and Z-errors means that the rest of the errors can be corrected as well.[5]

2.2.1 Encoding

A fundamental way of managing errors is to use encoding, meaning that multiple physical qubits represent one so called "logical qubit". The simplest case of encoding qubits is the repetition code where more than two qubits are repeated and their majority value reads as the value of the logical qubit. Consider for instance the encoded logical qubit:

$$|0\rangle_L = |0\rangle \otimes |0\rangle \otimes |0\rangle = |000\rangle \tag{2.9}$$

where $|0\rangle_L$ is the logical qubit and $|000\rangle$ represents the three encoded physical qubits. If a bit-flip were to happen to one of the physical qubits the value of the logical qubit would remain the same according to:

$$X_3 |0\rangle_L = |001\rangle \tag{2.10}$$

since a majority of the physical qubits remain error free, it will still be interpreted as a logical 0-qubit in the computation. If, on the other hand, a majority of the physical qubits were to flip as following:

$$X_{2,3} |0\rangle_L = |011\rangle \tag{2.11}$$

then there would be a logical error since this would now be interpreted as a 1-qubit in the computation. To solve an error an operator that can be applied to the entire logical qubit is needed, a logical operator.

2.2.2 Logical Operators and Code Distance

In order to perform operations on logical qubits a logical operator that can operate on all of the encoded qubits is needed. Such an operator is the tensor product of the individual operators acting on the qubits, but are oftentimes written with the

tensor product symbol suppressed. For the three-qubit repetition code, the logical X operator $X \otimes X \otimes X$ is written as XXX or $X_{1,2,3}$, and acts as a bit-flip on the logical qubit, transforming the computational basis state $|000\rangle$ into $|111\rangle$ and vice versa.

For quantum error correcting codes, the logical operator is closely related to the code distance, denoted d, which is the number of qubits that are acted upon by a non-identity logical operator of minimal weight. In other words, the distance of a quantum code is the smallest number of errors on the physical qubits which cause an undetectable logical operation on the logical qubit. The number of detectable errors are then $d - 1$, and the number of correctable errors are $\lfloor \frac{d}{2} \rfloor$.

In the three-qubit repetition code, the minimum weight of the logical X operator is three, meaning that the maximum number of X errors that can be corrected is $\lfloor \frac{3}{2} \rfloor = 1$. Two X errors would be incorrectly resolved and cause a logical X operation when corrected, and three X errors would correspond to the logical X operation itself.

To determine the distance of a quantum error correcting code, however, one must consider the minimal weight of all logical operators. For the three-qubit repetition code, applying Z to a single qubit results in a phase-flip of the logical qubit, and as such one possible logical Z operation of minimal weight is ZII. Thus, the three-qubit repetition code cannot detect any Z errors, and its code distance with respect to any possible error is $d = 1$. This illustrates how the distance d can be used as a measure of the robustness of a code, in the sense that it determines the maximum number of simultaneous errors that can be corrected, assuming a sufficiently low probability of errors occurring on the physical qubits.

2.2.3 Ancilla Qubits

In order to reset a faulty qubit one first needs to be able to identify whether an error has occurred. As mentioned in the beginning of this section, an obstacle with quantum computing is that it is not possible to measure whether an individual qubit has been compromised mid computation. According to quantum mechanics when a quantum system is observed its superposition states collapses, observing a qubit during a computation would thereby ruin the computation. A way around this issue is to implement additional qubits not part of the computation. These helper qubits or "ancilla qubits" are prepared in state $|0\rangle$ and connect to two or more data qubits to perform parity checks by applying an operator to two of these qubits at a time, this operator will then either commute or anticommute with the error. An ancilla qubit with an X-operator, an X-ancilla qubit, would therefore be able to detect a Z-error on a data qubit since these operators anticommute while a Z-ancilla qubit will detect X-errors. The ancilla qubit will thus signal when an error has occurred, and this is called a syndrome measurement. This kind of measurement provides information about potential errors without disclosing any stored information from the data qubits and the quantum superposition remains intact.

For example, the simplest possible repetition code as the one in section 2.2.1 is able to detect bit-flip errors on an encoded state:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \quad \rightarrow \quad |\psi_3\rangle = \alpha|000\rangle + \beta|111\rangle \tag{2.12}$$

An error occurring on the first qubit would yield the state:

$$|\psi_{3,\text{err}}\rangle = \alpha|100\rangle + \beta|011\rangle \tag{2.13}$$

This error can then be detected by first performing a parity check on qubit 1 and 2 by applying the operator $Z_{1,2}$ and then repeating this for qubits 2 and 3 with $Z_{2,3}$. These operations would not affect the original state since:

$$Z_{1,2}\,|\psi_3\rangle = \alpha|000\rangle + \beta|111\rangle = Z_{2,3}\,|\psi_3\rangle\,, \tag{2.14}$$

However, the operator $Z_{1,2}$ will change the phase of the defected state accordingly:

$$Z_{1,2}\,|\psi_{3,err}\rangle = -\alpha|100\rangle - \beta|011\rangle = -\,|\psi_{3,err}\rangle \tag{2.15}$$

Assuming that the probability of more than one qubit flipping is negligible this yields the information that either qubit 1 or qubit 2 has flipped. Performing the second parity check, $Z_{2,3}$ will then determine exactly which qubit through the method of exclusion.

A code of this size can however only detect bit-flip errors. In order to also detect errors unique for quantum computing, such as the phase-flip, there are more complicated types of codes with elaborate systems of data and ancilla qubits. In this work we consider the surface code, which is a more elaborate error correcting code that is considered to be promising for near term fault-tolerant quantum computers.

2.3 Surface Codes

Surface codes are quantum error correction schemes with data qubits and ancilla qubits on a square lattice. There are several different kinds of surface codes with slightly different configurations such as the toric code whose defining property is its periodic boundary conditions meaning that the qubit grid in theory would be shaped like a torus where the qubits at the top of the square are connected to the ones at the bottom, and the same applies to the qubits at the left and the right edge. A version of this is the planar code which is built the same but without the periodic boundary conditions, in theory making it a two dimensional square. It was realized that rotating the planar surface code 45° would decrease the amount of physical qubits needed per logical qubit and thus the rotated surface code was created. An example of a rotated surface code grid can be seen in Figure 2.1 where every white circle is a data qubit and the ancilla qubits reside in each of the squares as well as the arches on the edges, performing parity checks on the four respectively two neighboring data qubits in the corners. The orange fields contain X-ancilla qubits and the white fields contain Z-ancilla qubits. The outcome of the ancilla measurements is called the error syndrome and the goal is then to find the corresponding errors in order to be able to correct them. Deducing the errors occurring on the data qubits in a system like this typically requires a complex decoder.

The shortest chain of errors that can cause an undetectable error is called the code distance (d). In the case of rotated surface code it is the same as the number of qubits on an edge of the code which is seven in the examples seen in Figure 2.1. The code distance is thereby often used to refer to the size of a surface code.

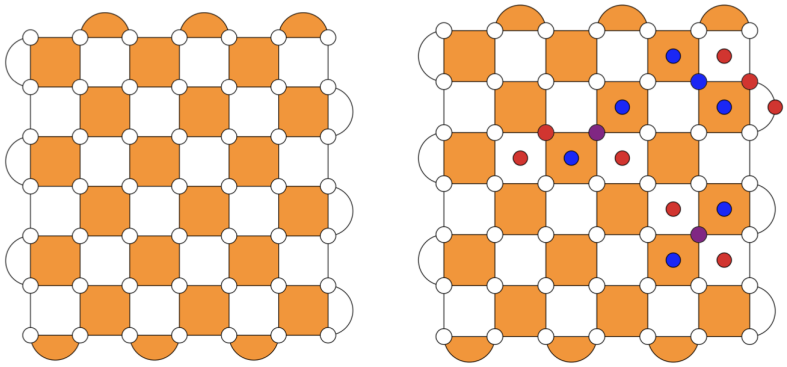


Figure 2.1: Rotated surface codes without errors and with errors. Blue dots in orange fields indicate an X-ancilla signaling about a neighboring Z-error, red dots in white fields indicate a Z-ancilla reacting to a neighboring X-error. A blue or red dot between squares mean a Z-error respectively X-error on that qubit, a purple dot represents a Y-error.

3

Machine Learning with Artificial Neural Networks

An artificial neural network is a kind of supervised machine learning algorithm made to recognize patterns by building a network of artificial neurons. These artificial neurons are called nodes and are connected to each other in different ways depending on the kind of network. This chapter aims to further explain the staples of machine learning as well as the two networks, graph neural network and convolutional neural network, used in this project.

3.1 Artificial Neurons

Between every connection in an artificial neural network a weight and a bias is applied on the passing data. Thus the input to an artificial neuron consists of the output from multiple or all neurons in a previous layer, times the weight for that particular connection with a bias of the neuron in question added as following:

$$z_j = \sum_{i=1}^{n} w_i x_i + b_j \tag{3.1}$$

Where z_j represents the input to neuron j, x_i is the output from node i with w_i as the weight between node j and i, b_j is the bias of the neuron in question and n the number of neurons connected to the current neuron. This is a linear equation and constitutes the first part of a neuron and performs a kind of linear regression. Next this function acts as input to another, predetermined function. A so called "activation function". A common activation function is the Rectified Linear Unit activation function or "ReLU" defined:

$$f(z_j) = \max(0, z_j) \tag{3.2}$$

Where $f(z_j)$, also denoted as y_j then becomes the output of the neuron and the process is repeated for the next layer of neurons. The linear function and the activation function are what make up an artificial neuron and multiple neurons together compose a layer in the network. Since the activation function is "reshaped" by the parameter values it will turn out differently for each neuron depending on the weights and biases of said neuron. The objective of the training of the network is then to optimize the parameters of the network by finding the parameter values

that in the end best fit the activation functions in the final layer of neurons to the data.

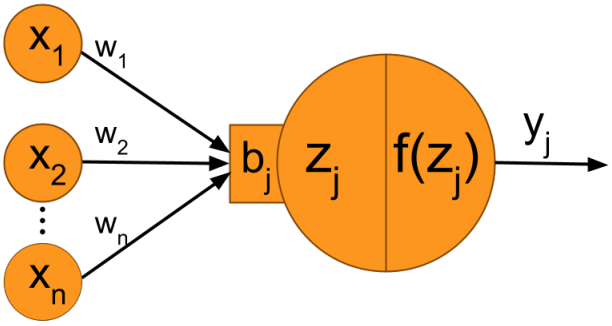


Figure 3.1: Drawn model of an artificial neuron

3.2 Backpropagation and Gradient Descent

The training of the network is implemented with the help of backpropagation which is a way of estimating the weights and biases in the network. The term backpropagation derives from the fact that the estimation starts with the last parameter in the network, b_n for a network with n number of neurons, and works its way to the beginning. The first step in this procedure is thus to evaluate how well the prediction in the final step fits the data and this is done by calculating the loss function.

A loss function is a way of measuring how well a neural network models a data set by looking at the difference between the current output and the desired output. There are several different loss functions for different objectives. Below, the cross entropy (CE) loss function can be seen:

$$\mathrm{CE} = -\sum_{i=1}^{N} y_i \cdot \log\left(\hat{y}_i\right) \tag{3.3}$$

N is the number of samples tested against, y_i is the correct value from the data set while $\hat{y}_i$ the predicted value from the network for the same input. This loss function is specifically used to optimize classification models. The output of a classification model will be an array of the probabilities for each class and these will be compared with the true labels. The measurement from the loss function is then used as a feedback signal in the backpropagation where the aim of the training is to minimize the loss by adjusting the weights and biases. This is done with an optimization algorithm called "gradient descent" which estimates the parameters for when the loss function is as small as possible.

To find the minima of the loss function differentiation is necessary. Since the loss function depends on every weight and bias in the network it is essentially a multivariable function of all w and b. For the general case it can be written as following: $L(y, \hat{y})$ where $y = f(wx + b)$. L can thus be differentiated with respect to these variables with the help of the chain rule and partial derivatives.

For the case of optimizing the last parameter in the network, b_n, the derivative is taken with respect to said parameter and the chain rule yields:

$$\frac{dL}{db_n} = \frac{dL}{dy} \cdot \frac{dy}{db_n} \tag{3.4}$$

When this gradient is equal to zero the loss function is at a (possibly local) minima and the model at its most accurate (in some vicinity of the minima). To find this minima in an efficient way gradient descent is used. Gradient descent works by iteratively calculating a next guess based on the gradient of the current point. Once the gradient of the starting point is calculated a scaled step is taken in the opposite direction of said gradient. This process is repeated until the loss gradient converges towards zero. The algorithm can be expressed as:

$$a_{m+1} = a_m - \gamma \nabla L(b_n = a_m) \tag{3.5}$$

Where a_m is the starting point and a_{m+1} the new guess. γ is the learning rate, a hyper-parameter chosen when training the network.

The corresponding parameter value to the minima of the loss function will yield the highest accuracy for the model and the backpropagation algorithm can move on to the next parameter. This process is repeated multiple times for every parameter in the network and for every training round until the overall training accuracy converges.

3.3 Convolutional Neural Networks

A commonly occurring artificial neural network is the convolutional neural network, CNN. This is a deep learning algorithm oftentimes used for classifying data in a grid pattern such as image classification and analysis due to their ability to detect patterns and spatial features in a grid. The defining characteristic of a CNN is the existence of convolutional layers which maintain the relation between neighboring matrix elements and can make use of information such as correlated pixels or error chains.

The first layer in a CNN is the input layer which is an array of the data with dimensions $n_h \times n_w \times n_c$ where n_h is the height of the array, n_w is the width and n_c is the number of channels in the input array (or depth). A convolutional layer would then start with applying what is called a filter to the input layer. A filter is in turn a smaller array, usually of size 3x3, of arbitrary weights that will be determined by backpropagation as the network is training.

A two dimensional convolutional operation with a filter of odd dimensions is defined as follows:

$$z_{ij} = \sum_{k=l=1}^{f,f} w_{kl} x_{k'l'} \tag{3.6}$$

with:

$$k' = k + i - \frac{f+1}{2}, \qquad l' = l + j - \frac{f+1}{2} \tag{3.7}$$

z_i a matrix element of the output matrix $\mathbf{z}$, also known as a feature map. $\mathbf{w}$ is an array of weights constituting the filter and $\mathbf{x}$ the input matrix. The convolutional operation is in other words the dot product between the filter $\mathbf{w}$ and a fragment of $\mathbf{x}$ centered around the matrix element x_i with height and width f. This process is pictured in figure 3.2. After the convolutional operation a bias is added in a similar manner as with single artificial neurons and the last step is an activation function such as the ReLU function described in section 3.1 equation 3.2

The matrix element z_i represents the filters correlation to that particular area of the data matrix. The filter then moves to the right and in that way "walks" across the array with a certain stride size, s, and the procedure is repeated all over the array. This way the feature map is formed and will act as input to the next layer in the network. The feature map will be on the form:

$$\frac{(n_h - f + 1)}{s} \times \frac{(n_w - f + 1)}{s} \times f_n \tag{3.8}$$

where f is the size of the filter, f_n the number of filters applied and s is the stride size.

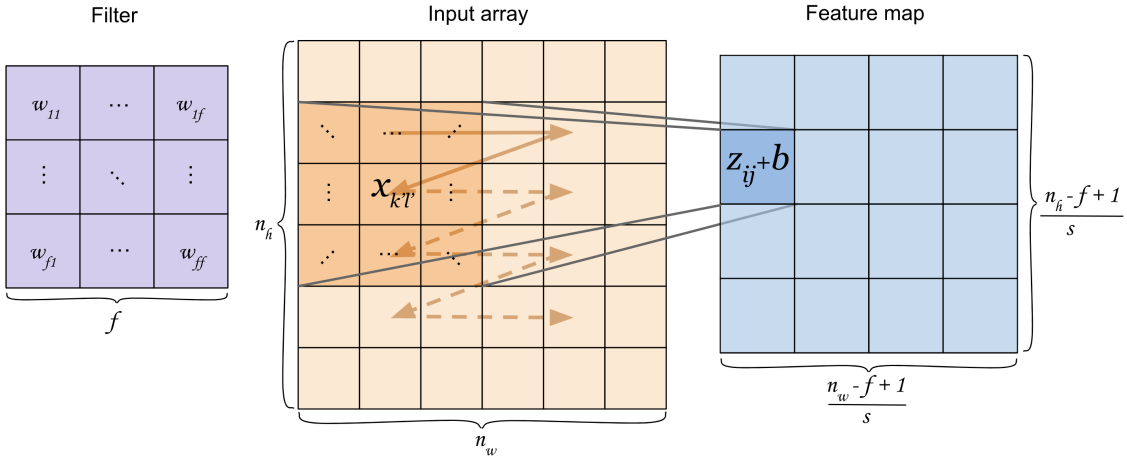


Figure 3.2: A 2D convolutional operation with a square filter. A filter of size f is applied to an area centered on x_i resulting in a matrix element in the feature map. The arrows in the input array represent the movement of the center of the filter.

Other common layers in a convolutional network are pooling layers. These work similarly to a convolution layer with a filter size f and stride s but the operation applied is fixed and can not be learned. The max-pooling layer for instance will return the maximum value of its f^2 inputs. Pooling layers can be used to downsample the data while keeping important information, usually by setting $s = f$.

A pooling layer will return a matrix on the form:

$$\frac{(n_h - f + 1)}{s} \times \frac{(n_w - f + 1)}{s} \times n_c \tag{3.9}$$

where n_c is the number of channels of the input data to the layer.

By choosing the filter size to be the same as that of the input array of the layer global max pooling is achieved. A global max pooling layer only looks at the single highest value for each layer in the array and will thus always produce an array of dimensions $1 \times 1 \times n_c$. This array can then make up the input data to a fully connected neural network with regular connected layers.

3.4 Graph Neural Networks

Graph neural networks (GNNs) are a class of neural networks models that have shown great promise for various prediction tasks in graph representation learning, such as node classification, link prediction and graph classification. The essence of a GNN is that the input data is represented as graphs, a collection of nodes and edges connecting the nodes, can be converted to a set of arbitrarily high-dimensional node feature vectors containing information about the structure and node information of the graph. While the field of GNNs is relatively new, it is rapidly growing and a large number of models have been proposed in the past decade. The main components of understanding the theory of GNNs are graph representations, as well as the computational functions for propagation, aggregation, and pooling of node information. In this chapter, we will describe these constituents piece by piece, to finally arrive at an understanding of GNNs as a whole. We will start by presenting the fundamental definition of a graph.

A graph G is a structure defined by a set of nodes (or vertices) V and edges E, denoted $G = (V, E)$. Most generally, the edges of a graph represent connections between separate units of information, represented by the nodes. An example of a graph is a social network of friends, where the nodes may represent individual persons with edges connecting friends in the network. We denote a node by its index as $v_i \in V$, and an edge as the pair of node indices connected by the edge: (i, j). The number of nodes and edges in a graph are represented as $|V|$ and $|E|$, respectively. Edges may either be directed, in which case they are defined by an ordered pair of nodes $(i \to j)$ where the connection is seen to only exist from v_i to v_j, or undirected, defined by an unordered pair of nodes where there is a single connection describing both directions. A graph is called complete if there exists an edge between every pair of nodes in the graph.

A graph in itself describes only the connectivity of a set of nodes. Further information can be included in the graph in the form of node features, here defined as a D-dimensional vector $\mathbf{x}_i$ for each node v_i. The node features assign numerical information to each node in the graph, and are used as the input features in a GNN. To represent all node features in a graph, we define the $|V| \times D$ dimensional node feature matrix as $\mathbf{X}$, where each row is a node feature $\mathbf{x}_i$. Additionally, edges can be assigned features, acting as weights $e_{i,j}$ on the connection between node v_i and v_j.

In the context of GNNs, a node is said to have a neighbourhood, which is the set of nodes connected to it by edges. Here, we define the neighbourhood of a node v_i as

$$\mathcal{N}_i = \{v_j \mid (i, j) \in E\}$$

for an undirected graph, using the same notation as Bronstein et al. [18]. Similarly, the node feature neighbourhood is defined as

$$\mathbf{X}_{\mathcal{N}_i} = \{\mathbf{x}_j \mid v_j \in \mathcal{N}_i\}$$

which is the set of node feature vectors of the nodes in the neighbourhood $\mathcal{N}_i$. Note that $\mathbf{X}_{\mathcal{N}_i}$ is technically a multiset, meaning that elements of $\mathbf{X}_{\mathcal{N}_i}$ may occur more than once, as different nodes may have identical node feature vectors. Here, however, we consider each node feature $\mathbf{x}_i$ to be unique, described by its node index i [18].

3.4.1 Graph Layers

A GNN layer is a general term for a function that performs a transformation on the graph. In a GNN, layers are used as functional modules that can be combined sequentially to obtain embedded representations of the features of a graph. Here, we consider two principal types of GNN layers: node-level layers and global pooling layers. Node-level layers act as transformations of the node features, without transforming the connectivity or structure of the graph in any way. In general, a node-level layer $\mathbf{F}(\mathbf{x_i})$ can be described by a neighbourhood aggregation function, as

$$\mathbf{F}(\mathbf{x}_i) = \mathbf{x}'_i = \text{aggregation}(\{\mathbf{x}_i, \mathbf{X}_{\mathcal{N}_i}\}) \tag{3.10}$$

where $\mathbf{X}_{\mathcal{N}_i}$ is the node feature neighbourhood of node v_i and $\mathbf{x}'_i$ is its transformed node feature vector. The aggregation function determines how the node features of neighbouring nodes are combined to form the transformed node feature vector $\mathbf{x}'_i$, and is applied to each node in the graph, producing a transformed graph. Note that the node-level layer $\mathbf{F}(\mathbf{x_i})$ may change the dimensionality of the transformed node feature vectors $\mathbf{x}'_i$, but does not affect the number of nodes or their connectivity through edges. Such a node-level layer is often called a *graph convolution*, and can be seen as a generalization of the convolution operator seen in CNNs to graph structured data. The key property of a graph convolution is that it does not make any assumption on the number of nodes from which information is aggregated, unlike the convolutional operations in a CNN which assume a grid structure with an equal number of connections at each "node".

A simple GNN layer is the graph convolution presented in [19], defined as

$$\text{GraphConv}(\mathbf{x}_i) = \mathbf{x}'_i = f\left(\mathbf{x}_i\mathbf{W}_1 + \sum_{j \in \mathcal{N}_i} e_{j,i}\mathbf{x}_j\mathbf{W}_2\right) \tag{3.11}$$

where $\mathbf{W}_1$ and $\mathbf{W}_2$ are weight matrices of dimension $D \times D'$, where D is the number of features for the input node, and D' are the number of features in the transformed node feature vector, and f is an activation function introducing nonlinearity to the layer, such as ReLU defined in Eq. 3.2. This operation aggregates information from the node neighbourhood of each node as a weighted sum, determined by the edge features. The entries in $\mathbf{W}_1$ and $\mathbf{W}_2$ are the learnable parameters of this layer when used in a GNN.

In a global pooling layer, the feature information of all nodes in a graph is combined to a single graph-level feature vector. An example of a global pooling layer is mean pooling, defined as

$$\text{GlobalMeanPool}(G = (V, E)) = \frac{1}{|V|} \sum_{v_i \in V} \mathbf{x}_i \tag{3.12}$$

where the node features are averaged over all nodes in the graph.

3.4.2 GNNs for Prediction Tasks

By applying graph convolutions and pooling layers, one may obtain a transformed embedding of the initial information in a graph, which can be used for various classification tasks. For example, the dimensionality of the node features can be transformed using a series of graph convolution layers to match a node-level prediction task. For graph classification, a single feature vector for an entire graph can be obtained by subsequently performing a global pooling of the node features. As such, the complex structure of a graph is transformed to a single vector space embedding. This vectorized representation of the graph can then be used as an input to a classifier, such as a fully connected neural network, to learn a mapping from an input graph to a predicted class.

4

Methods

In this section, we first describe how we map the decoding problem for the rotated surface code to a classification problem solvable by a neural network. We then describe how labelled training data for supervised learning was generated. Finally, we present how syndromes were represented as input data for the respective models, as well as the architectures of the neural network decoders.

4.1 Decoding the Surface Code with Neural Networks

The decoding task consists of, for an observed syndrome S caused by the underlying data qubit error E, finding any error configuration C that returns the state to the codespace without causing a logical error in the process. Applying any operations that would produce the same syndrome S when measured by the ancilla qubits ensures that the state is returned to the codespace. However, choosing an arbitrary operation may result in the product of the error and the correction $E \cdot C$ causing an unwanted logical operation – which would be considered a failed decoding of the syndrome. An optimal decoder would always choose a correction that has the lowest probability of causing such a logical error. Since multiple errors E can cause the same syndrome S, but no direct observation of the error can be made, it is not a trivial task to find the optimal solution.

Training a neural network to learn the mapping from syndrome to correction offers a possible way to solve the decoding task in an efficient way. To do this, we consider the decoding of the surface code as a classification problem, an area where neural networks have proven to be successful in general. All the possible error chains (sets of operations on the data qubits) on the surface code can be divided into four *equivalence classes*. When decoding a syndrome, choosing any correction operation belonging to the same equivalence class as the underlying error which caused the syndrome ensures that the decoding is successful. A selection of a correction operator belonging to a different equivalence class than the underlying error will lead to an undetected logical operation, and is considered a failed decoding. As such, our general approach of applying neural networks as decoders is to represent the syndromes as input data and, through a feed-forward pass through the network, return a prediction of the most probable equivalence class.

4.1.1 Specific Representation of Equivalence Classes

To uniquely determine the equivalence class to which an error chain belongs, it is necessary to choose a single representation of the four logical operators. Following the convention used in [20], we choose to represent the logical Pauli X operator, X_L, to be the length-d chain of errors on the western edge of the surface code lattice, and the logical Pauli Z operator, Z_L to be the length-d chain of errors on the northern edge of the lattice. The equivalence class of any error chain can then be uniquely determined from whether or not they commute with these specific representations of the logical operators. More concretely, this leads to an error chain belonging to equivalence class X if it has an odd number of X errors on the northern edge, and conversely it belongs to class Z if it has an odd number of Z errors on the west edge. If none of these conditions are met, the error chain belongs to class I (corresponding to no logical operator). If both of the conditions are met simultaneously (odd parity of X on the northern edge *and* odd parity of Z on the western edge), the error chain belongs to class Y, which in the context of the surface code corresponds to the product $X_L \cdot Z_L$.

4.1.2 Generating Training Data

In this study, we solely consider errors caused by the *depolarizing* noise model, where each physical qubit undergoes a Pauli X, Z or Y error with equal probability $p/3$, with p being the *physical error rate* – the probability of an error occurring on each individual qubit. Training and test data was generated by simulating depolarizing noise at different physical error rates to the rotated surface code, and determining the syndromes consistent with the resulting error chains.

Training the networks to find the most probable equivalence class of a given syndrome was done with supervised learning, where the training data consists of (syndrome, equivalence class) pairs. To learn the input-output relation, it is then necessary to determine a method for labelling the syndromes used for training with a target equivalence class. In previous studies of neural network based decoders, a common approach has been to use the equivalence class of the error chain which caused the syndrome during sampling of data as the target. However, since each syndrome is not caused by a unique error chain, this approach relies on repeatedly sampling the same syndrome caused by different errors to obtain an estimate of the probability distribution of the equivalence classes of that syndrome. As the number of possible syndromes scales exponentially with the code distance (2^{d^2-1}), this method of labelling training data quickly becomes problematic. For code distances beyond the smallest codes, it becomes unfeasible to sample more than a single error chain for each syndrome in a reasonably large dataset, which results in a poor estimate of the equivalence class probability distribution.

4.1.2.1 Reference Decoders

To circumvent this issue, labelled data was instead generated by using existing decoders to predict the most probable equivalence class of the syndromes used to train the networks. Two such reference decoders were employed. The first being

the Effective Weight Decoder (EWD), which uses Metropolis-based Monte Carlo sampling to determine the frequency of the most probable error chains in each equivalence class for a given syndrome [20]. EWD is near-optimal, at the cost of having a relatively slow decoding time, both in absolute terms and with respect to the time complexity scaling with code distance d. At low error rates, the time complexity of the EWD decoder was approximated to $\mathcal{O}(d^5)$ for code distances up to $d = 15$, with an indication of even having superpolynomial decoding time.

Due to the poor time complexity of the EWD decoder, a faster but less accurate decoding algorithm based on Minimum-Weight Perfect Matching (MWPM) was used when generating training data for large code distances (above $d = 9$). The MWPM decoder is also the standard algorithm used to benchmark decoders on the surface code in the literature, and will be used in this study to benchmark the neural network decoders. It should be noted that using reference decoders to generate training data for supervised learning means that the performance of the neural network decoders is strictly bounded by that of the reference decoder. In the ideal case, if a neural network decoder would achieve a perfect accuracy with respect to the reference decoder, it would exactly mimic the performance of the reference decoder. With this said, if the neural networks can achieve a high accuracy when trained with the near-optimal EWD targets, it could possibly outperform less accurate decoders such as MWPM with a faster runtime than the EWD decoder once trained.

Several smaller datasets were initially generated for various code distances d and error rates p when exploring models in an early stage of the project. Finally, the primary dataset used for training the final models was a set of $5 \cdot 10^5$ unique syndromes with equivalence class targets determined by EWD, randomly generated from depolarizing noise with an error rate of $p = 11\%$ on the $d = 7$ surface code. The choice of error rate for sampling the training data was in part guided by previous studies of neural network decoders [12, 13], as well as our own early results, which indicated that using training data generated at a relatively high error rate gave a higher performance at that error rate while also retaining accuracy for lower error rates, compared to training at lower error rates.

As described by Varsamopoulos et al. [12], training a neural network decoder with data sampled at a fixed error rate will optimize the performance of the decoder at that error rate, and may lead to poor generalization to unseen data sampled at other (especially larger) error rates. In a realistic experimental scenario, there will be some underlying error model to which the performance of the decoder can be specifically tailored to by adjusting the training error rate. In this work however, we only evaluate the general performance of a theoretical decoder under the depolarizing error model, tested at various physical error rates after training. For this reason, we choose a fixed training error rate that is large enough to give a diverse set of training syndromes while remaining sufficiently low such that the shortest error chains are more probable to cause those syndromes. This is to retain similar performance when testing the decoder at error rates lower than the one used for training data sampling, as the minimum weight error chains will be most probable to cause a syndrome at low error rates.

For the $d = 7$ dataset, the specific choice of generating training data with the

physical error rate $p = 11\%$ was also done to follow the practice used in previous work by Overwater et al. [13], where the pseudo-threshold of the MWPM decoder was chosen as error rate for the training data sampling. For the rotated surface code with code distance $d = 7$, the pseudo-threshold of MWPM (the physical error rate where the decoder performs as well as a single unencoded qubit) is approximately 0.114. This choice was found to give a satisfactory trade-off between producing diverse data without sacrificing performance when evaluating the performance of the decoders on test data generated at various lower error rates.

The size of our $d = 7$ training set is significantly smaller than what has been used for supervised learning in similar studies of neural network decoders [12, 14], but due to the high accuracy of EWD, each syndrome in the dataset is ensured to have a near-optimal estimate of its most probable equivalence class. Additionally, due to the exponential scaling of the number of possible syndromes with respect to code distance, generating this many syndromes at $d = 7$ with an error rate of $p = 11\%$ is far from having sampled the entire syndrome space.

More specifically, the total number of possible unique syndromes for the $d = 7$ rotated surface code is $2^{7^2-1} \approx 2.8 \cdot 10^{14}$, meaning that the training data for $d = 7$ contains less than $2 \cdot 10^{-9}$ of the possible syndromes (although some syndromes may be more common than others for a specific physical error rate). For this reason, it is expected that the training data has a low degree of overlap with syndromes sampled during testing, and that a high decoding accuracy during testing indicates that the model is able to generalize and not just naively repeat learned patterns from the training data.

4.2 Convolutional Neural Network Decoder

The rotated surface code is structured on a grid and thus yield a quite intuitive matrix representation of the syndromes. A convolutional neural network is well suited for processing and classifying matrix based data and in this section we go through how the syndromes are represented in a matrix as well as the architecture of the convolutional neural network that was built.

4.2.1 Matrix Representation of Syndromes

The syndromes generated from the data generating algorithms are already on matrix form and paired with the most likely equivalence class by default. This makes preparing the data rather convenient since it is already on the right form and only minor cleaning of the data set is necessary. This "translation" from syndrome on the surface code to error matrix can be seen in 4.1 where each "1" is an error-signaling ancilla qubit and the 0's are neutral ancilla qubits. In some places there are zeroes where no qubit exists. These are simply fillings to achieve a square matrix.

The syndrome matrix will be of size $(d+1)\times(d+1)$ where d is the code distance. This representation of the data does not immediately take into consideration whether the error is measured by an ancilla qubit measuring X or one measuring Z, but instead

we rely on the network to recognize this from the positions of the signaling ancilla qubits.

Figure 4.1: Syndrome from a code of distance d=7 and error probability p=0.11 represented as a so called syndrome matrix.

4.2.2 CNN Architecture

A convolutional neural network was built based on fully convolutional neural networks which can take input data of any shape and is thus not restricted to just one matrix size. For this network the python library Tensorflow was used which is well suited for building and training models. Several convolutional layers and their settings were chosen iteratively by trial and error. For the convolutional layers the filter size was chosen to be “3” and the stride size to “1”. The activation function ReLU was chosen which is a standard activation function.

Since every convolutional layer decreases the matrix size, the matrices would soon reach a size too small for the next layer to handle depending on the size of the input matrices. To prevent this, padding was added between every layer to make it possible for the network to handle a larger variety of code distances. This was done by setting padding to “same” in the convolutional layers, which pads the output matrix with a layer of zeroes around all edges. This way the shape of the matrices could remain throughout the convolutional layers. The network also contains two max pooling layers with a pool size of 2 represented in orange in figure 4.2.

The last layer of the convolutional part is a global max pooling layer which picks the highest value from each layer of filters and thereby turn any input size into 1x1 vector with the same depth as the amount of filters in the layer before. Since this amount of filters is determined in the network architecture the output of this layer will always have the same dimensions regardless of the shape of the input to the network. The convolutional part of the neural network can be seen in Figure 4.2. With a fixed output shape from the convolutional part the data can then pass on to a fully connected network with dense layers that require a predetermined input dimension. This part consists of several dense layers with a max width of 128 nodes which narrows down to four output nodes that represent the four equivalence classes.

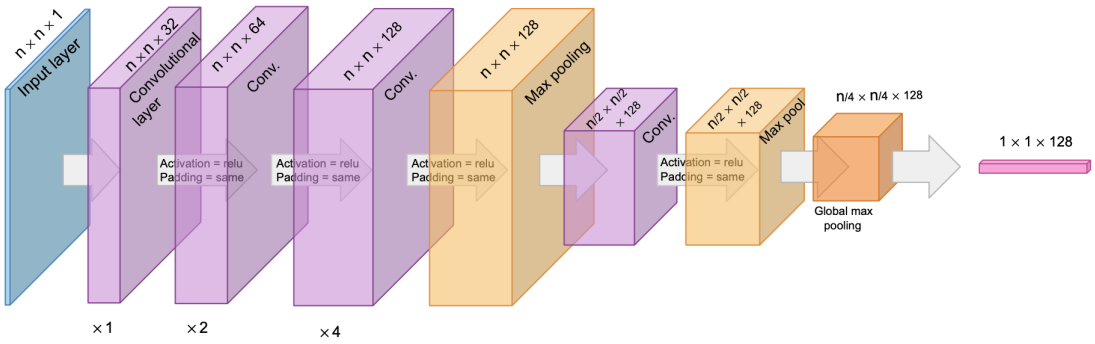


Figure 4.2: Fully convolutional part of the CNN. Not to scale.

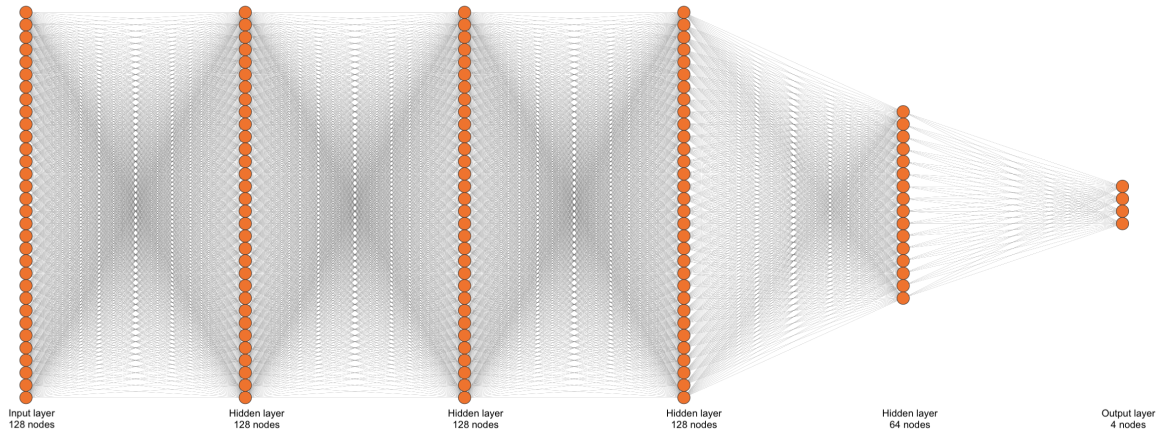


Figure 4.3: Figure representing the number of layers and connectivity of the fully connected part of the CNN as well as the number of output nodes. The number of nodes pictured in the other layers is not accurate but was scaled down to enhance visibility.

4.3 Graph Neural Network Decoder

While the ancilla qubit lattice on which the syndromes are measured are naturally grid structured, a graph representation of the syndromes may be considered by only including information from the activated stabilizer generators that anti-commute with the error (corresponding to the non-zero values in the qubit matrix shown in Figure 4.1. These -1 eigenvalue outcomes of individual ancilla qubit measurements are referred to as defects, with a syndrome being the collection of all defects on the surface code after the stabilizer measurements. As the decoding problem consists of mapping an input syndrome to a single predicted equivalence class, a GNN was designed to perform graph classification on graph representations of syndromes. Here we describe how syndromes were represented as graph structured data, as well as the architecture of the implemented GNN decoder.

4.3.1 Graph Representation of Syndromes

The syndromes generated for training and testing are initially represented as matrices, and do not have a natural graph representation. As such, the choice of how syndromes are to be mapped to graphs is an important part of applying a GNN to the decoding problem. Graphs were constructed by representing each individual defect in the syndrome as a node. This is essentially the only information available to the decoder – no information is known about the states of the data qubits.

There are however two types of defects; those measured by X-ancilla qubits and those measured by Z-ancilla qubits. The X-ancillas are shown as defects in the syndrome if they measure an odd parity of Z errors on the four adjacent data qubits, whereas the Z-ancillas measure defects if there are an odd parity of X errors on adjacent data qubits. Note that Y errors (which occur with the same probability as X and Z in the depolarizing noise model) can be considered as overlapping X and Z errors. In the visualization of the surface code, X- and Z-ancillas are represented by orange and white faces on the grid, respectively. The ancilla type of each defect was chosen as the first node feature, as they are directly related to the underlying data qubit error which caused the syndrome. For example, an error consisting purely of Pauli X operators will only be detected Z-ancillas. To distinguish between the two defect types in the node feature representation, they were one-hot encoded in the first two elements of the node feature vector.

Furthermore, another important property of the defects is their position relative to the code boundaries (and to other defects). In essence, the equivalence class of a string of errors is closely related to its endpoints, where the defects will be measured. In our specific representation of the equivalence classes, strings of X-errors that end on the north boundary of the code will belong to equivalence class X, as they cause odd parity of X-errors on the northern boundary of the code. Strings of X-errors that are contained within the boundaries of the code or run to the south boundary will not contribute to odd parity on the northern boundary. Similarly holds for how strings of Z-errors are positioned relative to the west and east boundaries.

To attempt to capture this unobservable information about the positions of the underlying errors, two node features were defined as the distance of the defect to the north boundary, and the distance of the defect to the west boundary. These distances were scaled by the code size d, so that the positional node feature captures the relative position of the defects. This was done in an attempt to keep the graph representation as general as possible, with the aim of allowing the model to generalize to arbitrary code distances.

The defect type and position relative to the boundary together form the node feature vectors, defined as

$$\mathbf{x}_i = (X, Z, D_{north}, D_{west}) \tag{4.1}$$

where $X = 1$ if the defect corresponds to a X-ancilla and 0 otherwise, $Z = 1$ if the defects corresponds to a Z-ancilla. $D_{north} = \frac{k}{d}$ is the normalized distance to the north boundary, with k being the row index in the syndrome matrix and d being the code distance. Similarly $D_{west} = \frac{l}{d}$ where l is the column index in the syndrome

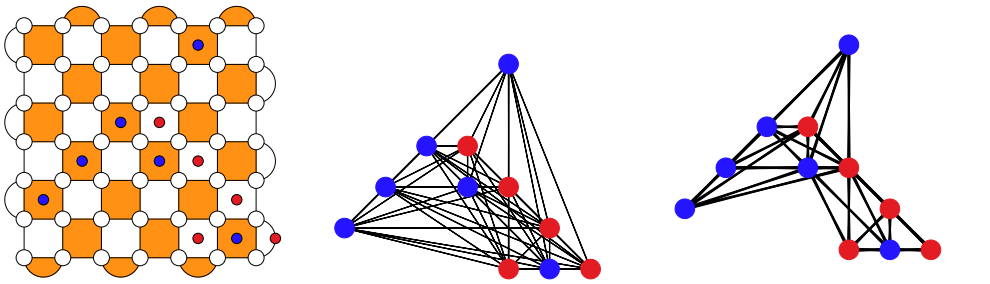
matrix. The integer values k and l range from 0 to d, and as such, D_{west} and D_{north} are normalized to lie within the interval $[0, 1]$ for all defects and code distances.

This constitutes the definition of node features, but to complete the graph representation we also require a definition of edges connecting the nodes in the graph. Two different ways of determining which edges to be included in the graph were tested. Firstly, all graphs were defined to be complete, meaning that there is an edge connecting every pair of nodes. Upon training the GNN model with syndromes from larger code distances, it was found that such a dense graph representation was severely prohibitive to the runtime of the GNN decoder. As such, an alternative graph representation was also implemented, where each node is connected to its m nearest neighbours. Using a value of $m = 5$ resulted in only slightly lower performance at small code distances, while drastically improving the runtime of the GNN decoder for larger code distances, as the number of nodes scale with d^2.

Finally, the edges were assigned a single-valued feature $e_{i,j}$ which weights the contribution of individual neighbouring nodes in the graph convolution described by Equation 3.11. All edge weights were defined based on the shortest distance of a possible error chain connecting two defects. This distance can be calculated as the maximum of the difference between the row and column indices of two defects in the syndrome matrix. As it is desired that short distances correspond to larger edge weights, the inverse distance was used. From early tests, it was also found that using a squared inverse distances was favorable, to further diminish the contribution of edges between nodes far apart. The edge weights between nodes i and j with row and column indices (k_i, l_i) and (k_j, l_j) were defined as

$$e_{i,j} = \left(\frac{1}{\max(|\mathrm{k_i} - \mathrm{k_j}|,\ |\mathrm{l_i} - \mathrm{l_j}|)} \right)^2 \tag{4.2}$$

To illustrate the graph representation, the mapping of a syndrome on the $d = 7$ surface code is shown together with its corresponding complete graph in Figure 4.4b. The same syndrome mapped to a graph where each defect is only connected by edges to its $m = 5$ closest neighbours is shown in Figure 4.4c. Note that in cases where the edges would become directed by having connections between pairs of nodes in only one direction, such edges were made to be undirected by adding an edge in the opposite direction. This means that there are cases where the node degree may become slightly larger than m for certain nodes. This was done to avoid constructing graphs where information can only propagate in one direction between clusters of nodes within the graph.



(a) Syndrome. **(b)** Complete graph. **(c)** Sparse graph with $m = 5$.

Figure 4.4: Mapping of a syndrome to graphs with different connectivity for the $d = 7$ surface code. In **(b)**, as a complete graph where all nodes are connected by edges, and in **(c)** as a sparse graph, where each node is connected to its $m = 5$ nearest neighbours. **(a)** shows the syndrome on the rotated surface code, where X- and Z-ancillas are represented by orange and white faces, respectively. X- and Z-defects are shown as blue and red circles.

To exemplify the node features, the top most node in Figures 4.4b and 4.4b has node feature vector

$$\mathrm{x} = (1, 0, \frac{1}{7}, \frac{5}{7})$$

In the case when an empty syndrome is encountered, meaning that no defects were measured by the ancilla qubits, it is not possible to construct a graph, as there are no nodes. In this case, the GNN decoder was set to always output class I, as such syndromes are most commonly caused by no error having occurred for sufficiently low physical error rates, in which I is the correct equivalence class.

4.3.2 GNN Architecture

To implement the graph neural networks and graph representations, we used the popular library PyTorch Geometric [21], where many GNN layers are available in an object oriented environment, as well as efficient batching of graph representation data.

The final GNN architecture used for decoding syndromes on the surface code consisted of three sequential GraphConv layers (Equation 3.11) with ReLU activation functions and an increasing number of output neurons in each layer. Each graph convolution propagates information between the nodes, increasing the dimensions in the transformed node feature vectors while retaining the same structure of the graph. The graph convolutions are followed by a GlobalMeanPool layer to obtain a single vector representation of the entire graph by averaging the transformed node feature vectors. This final *graph embedding* was then fed through three dense layers, with ReLU activation functions in the hidden layers, acting as a classifier.

Two classification schemes were tested. Firstly, the training targets were one-hot encoded as 4-dimensional vectors corresponding to the four equivalence classes. In this

representation, a softmax activation function was applied to the output layer of the GNN, so that the network returns a probability distribution of the four equivalence classes for a given syndrome graph. The prediction of the network was defined as the class with highest probability. When training the network with this multiclass classification scheme, cross entropy was used as loss function.

Secondly, as has been suggested in previous studies of classification based neural network decoders [12], the equivalence classes were instead represented by a two-bit binary number. This can be done since the condition for equivalence class X and Z do not depend on each other, and since class Y corresponds to the combination of class X and Z, while class I corresponds to when neither the condition for class X nor Z holds true. In the second classification scheme, only two output neurons are needed – each performing binary classification. For this scheme, an elementwise sigmoid function was applied to the output layer of the GNN, meaning that the network returns two probabilities – representing the conditions for class X and class Z, respectively. Consequently, the network prediction was in this case determined by rounding each output probability to zero or one, and mapping the resulting two-bit binary number to the corresponding equivalence class as follows:

$$00 \to I \quad 10 \to X$$

$$11 \to Y \quad 01 \to Z$$

Again, note that this mapping is done due to the fact that class Y corresponds to both conditions being true, and class I corresponds to neither condition being true. For this classification scheme, the loss function used when training was the sum of two individual binary cross entropy functions, one for each output neuron. Using the binary classification scheme was found to give a slight performance improvement, and was used in the final GNN model.

It should be noted that the number of weights in the first layer of the GNN in general depends only on the number of node features, and not the number of nodes. The global pooling then effectively reduces the number of nodes to one. As such, in principle, arbitrarily large graphs can be processed with the exact same network architecture. Simple neural network decoders based on fully-connected input layers are tailored to a specific code distance d (as this determines the number ancilla qubits), and can not be used for decoding larger (or smaller) surface codes than the ones used during training.

Attempts were made to improve the performance of the GNN by using more expressive graph convolution layers that have been shown to perform well in other applications of graph classification, but none of the alternative GNN layers tested (SIGN [22], GIN [9]) gave any indication of improving performance, and were thus not studied further.

4.3.3 Comparison of Alternative Graph Representations

The final graph representation was determined from early iterative testing of decoding performance of the GNN when using different graph representations. A

comparison of the same model trained on alternative graph representations is presented here, to give a motivation for the choices made when determining which node features to include and how to represent edges. The same model, using the final GNN architecture presented in the previous section, was trained on a smaller dataset with 50 000 unique syndromes sampled at code distance $d = 7$ with an error rate of $p = 11\%$. 80% of the data was used for training, while 20% was used for validation. Figure 4.5 shows the validation accuracy averaged over each epoch during training for five different graph representations: three using different node features, one where the inverse distance edge weights are squared, and one where edges are only drawn to the $m = 5$ nearest neighbours of each node.

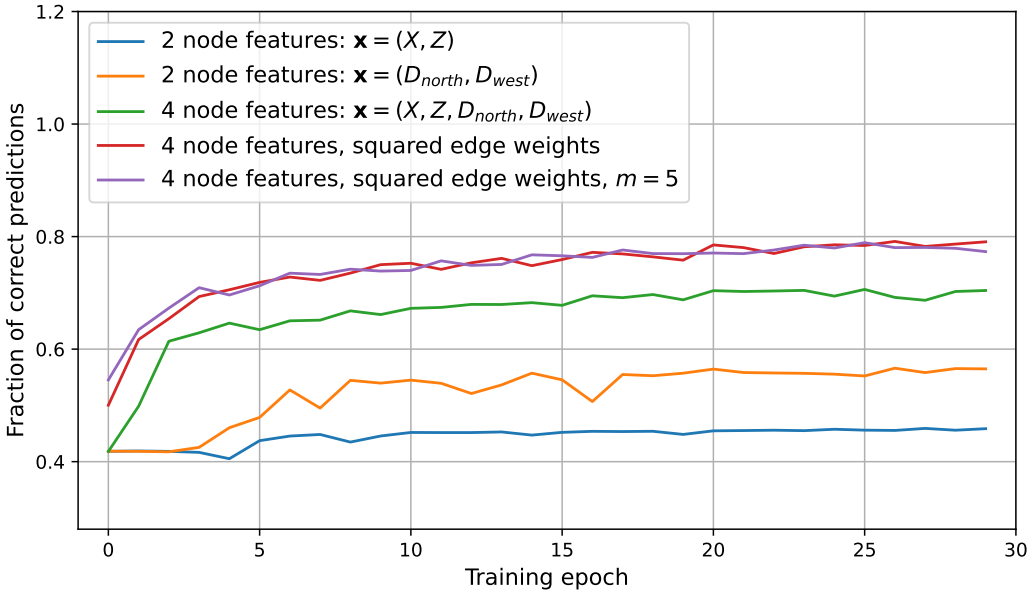


Figure 4.5: Comparison of the average validation accuracy over 30 epochs of training the same model on the same data ($d = 7$, $p = 11\%$), using different graph representations of the syndromes. The results show that using all four proposed node features described in Equation 4.1 improves the performance. Additionally, squaring the inverse distance between nodes used as edge weights further improves the performance. Finally, the performance is similar when reducing the connectivity of the graphs by drawing edges to the $m = 5$ nearest neighbours of each node.

Firstly, the performance when using different node features is compared. Using only the defect type node features results in the validation accuracy converging at 45% for this dataset. Similarly, using only the node features describing the relative distance to the west and north boundaries of the code gives a higher, but still poor, validation accuracy of 55%. When using all four node features, according to the definition presented in Equation 4.1, the model is able to better learn the input-output relation, and reaches a validation accuracy of 70% for this example dataset. Beyond this, the performance can be seen to increase further when also raising the

edge weights (the inverse distance between defects) to the power of 2, according to the definition used in 4.2. With this graph representation, the validation accuracy reaches around 79% in the same number of epochs on the same data. Additionally, when using the same node and edge features, but limiting the connectivity of the graph by only including the $m = 5$ shortest edges for each node, it can be seen that the performance on the validation data is not affected significantly for this code distance and error rate.

5

Results and Discussion

In this section we present and discuss the findings of our study. Results pertaining to the CNN and the GNN will be treated separately here, and the conclusions of these results will be compared in the next chapter.

The decoding performance of the trained models was evaluated by repeatedly generating random errors (allowing for duplicates) on the surface code under depolarizing noise at various physical error rates p, determining the syndrome consistent with each error, and using the syndrome as input in the trained decoder. The output equivalence class predicted by the decoder is compared to the true equivalence class of the underlying error, where each incorrect prediction is counted as a logical error. Repeating this procedure and calculating the fraction of incorrect predictions gives an approximation of the logical error rate of the decoder, defined as

$$\text{Logical error rate} = \frac{\text{Number of incorrect predictions}}{\text{Number of decoded syndromes}} \tag{5.1}$$

At the point where the logical error rate of the decoder is equal to the physical error rate, called the pseudo-threshold, a single unencoded qubit would perform better than the surface code with the decoder. The logical error rate is expected to increase as the physical error rate increases, but a higher pseudo-threshold is preferable, as it leads to higher tolerable physical error rates, as well as better performance at lower error rates.

5.1 Convolutional Neural Network Decoder

In this section we present the results for the CNN decoder. We will go through the training process and then look at how the decoder perform against the existing decoders. Lastly we will look at some other aspects such as performance on larger code sizes.

5.1.1 Performance and Accuracy

General accuracy and performance of the network was mainly measured on a model trained on data with code distance $d = 7$ and error rate $p = 0.11$ (see Section 4.1.2.1 for a motivation of the choice of physical error rate for training). This code distance yields a syndrome matrix of size 8×8 which is the smallest matrix this network architecture can handle. Smaller surface codes than that is barely useful to train on since the number of possible syndromes is so small that a neural network decoder is not necessary.

In order to create a model to outperform MWPM we trained on the more accurate data set generated with EWD. The accuracy against the EWD labels reached during training for this model reached an asymptote at 0.95 after around 20 epochs of training, and since EWD itself has an accuracy of 0.934 at $p = 0.11$ the overall accuracy against the true labels can be estimated to $0.95 \cdot 0.934 = 0.887$ which is higher than the accuracy of 0.886 for MWPM during the same conditions. This is confirmed in Figure 5.1 where the performance of the model exceeds the estimation from the algorithm for every p shown in the figure. Thus this model was successfully able to outperform the MWPM algorithm under these conditions.

Furthermore it was examined how well the CNN decoder predicts other code distances than what it was trained for. This is shown in Figure 5.2 and it can be seen that neither code size outperforms MWPM in this scenario and that the two code sizes actually deviates quite identically for low error rates. For lower error rates we can then assume that it is as difficult for the model to predict code sizes one size smaller as one size larger than what was originally trained for. However as p increases we can see that the predictions on larger code sizes seem to be the more challenging ones.

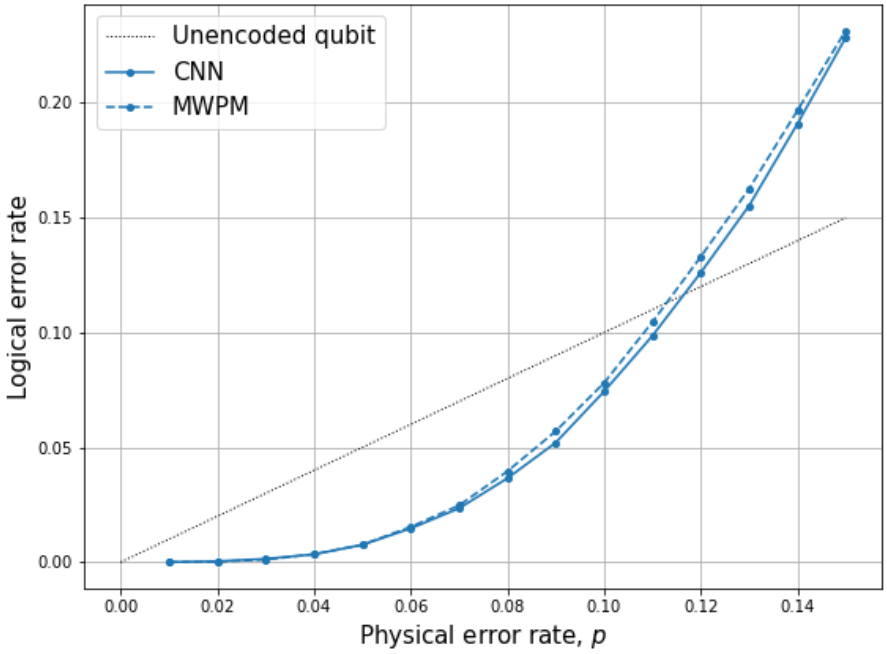


Figure 5.1: Performance of a model trained on code distance $d = 7$, error rate $p = 0.11$ and 500 000 unique syndromes generated from EWD showcasing how the model outperforms the MWPM algorithm on code distance $d = 7$.

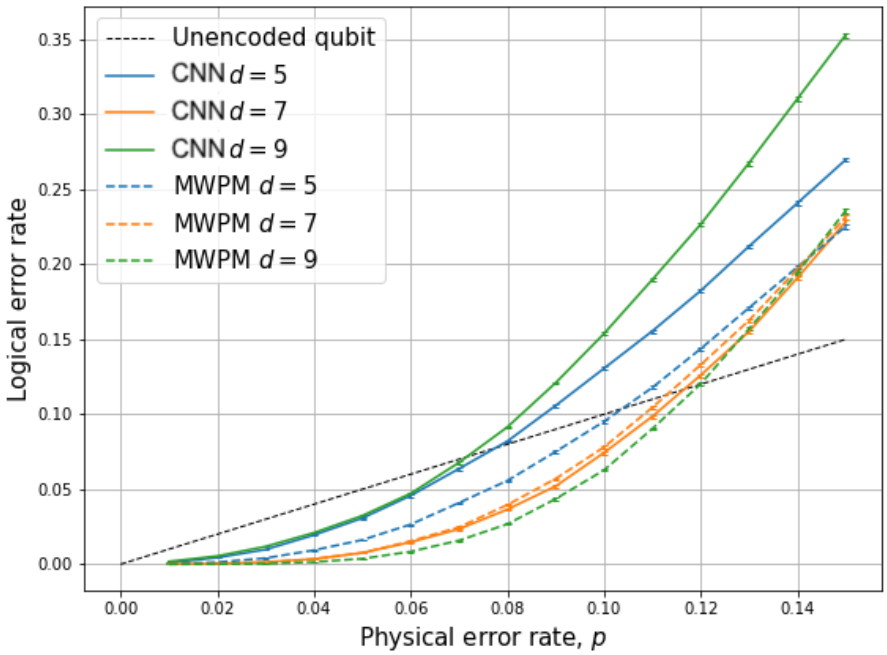


Figure 5.2: Performance of a model trained on code distance $d = 7$, error rate $p = 0.11$ and 500 000 unique syndromes generated from EWD showcasing how the model classifies codes of one size smaller respectively larger than what was trained on.

Furthermore it was noticed that training at a higher physical error rate p was beneficial, as it improved the decoding performance at all error rates. Our findings show that for the $d = 7$ code, training on data generated at a physical error rate of $p = 11\%$ improved the decoding performance with respect to training at a lower error rate of $p = 5\%$. This is shown in Figure 5.3, where two instances of the CNN decoder was trained separately with data generated at $p = 5\%$ and $p = 11\%$ for the $d = 7$ surface code. Even at $p = 5\%$, marked with an orange vertical line in the figure, the model trained at data sampled from the higher error rate of $p = 11\%$ performs better, as it has a lower logical error rate.

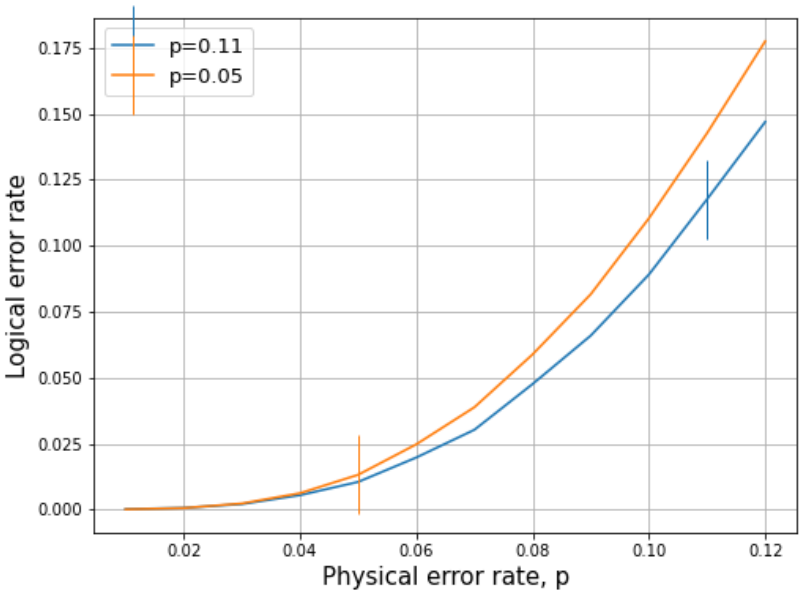


Figure 5.3: Figure depicting the performance of two models trained on data with a physical error rate $p = 0.05$ respectively $p = 0.11$. The vertical lines marks the physical error rate on the x-axis that respective model was trained on.

5.1.2 Performance on Larger Code Distance

We also examined how scalable the CNN was to larger code sizes. The largest code size trained on was $d = 15$ and mixed error rates between 0.01 and 0.07. In this case we used data generated from MWPM since the aim of this inquiry was mainly to look into scalability and not to outperform MWPM. EWD takes a lot of time and computer power for code sizes this large and it felt justified to use MWPM.

Under the conditions this network was trained the results are not very promising. As seen in figure 5.4 the accuracy quickly deviates even within the error rates it was trained for and MWPM is far superior. However, the conditions were not ideal due to difficulties generating training data. It would be preferable to train on higher error rates and larger data sets as well as being able to train on data generated

with EWD to evaluate this better. During training the test accuracy converged at around 0.82 compared to 0.98 for a model trained on MWPM data for $d = 7$ and $p = 0.5$. Thus the scalability under these circumstances were not ideal.

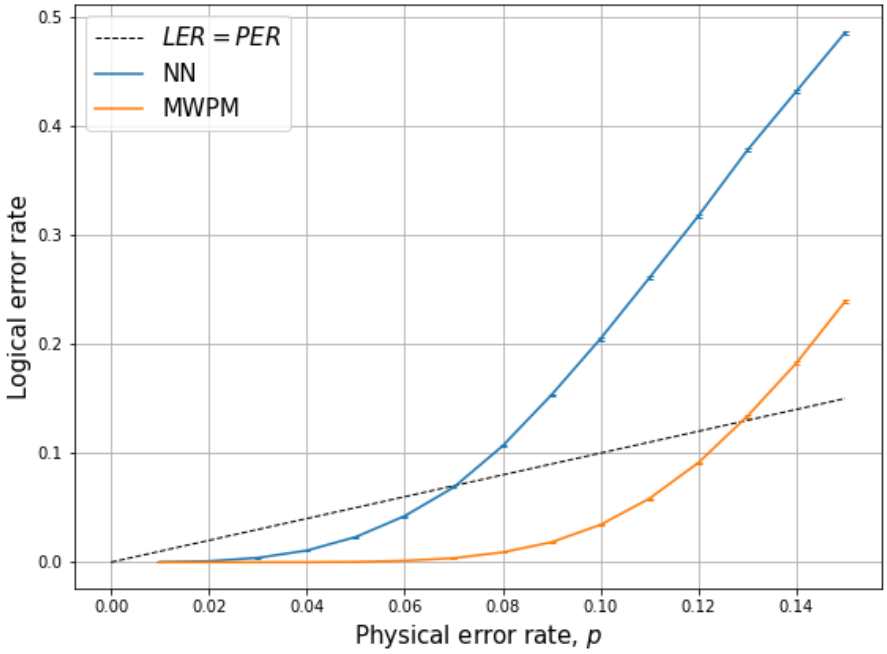


Figure 5.4: Performance of a model trained on code distance $d = 15$, error rates between 0.01 and 0.07 and around 100 000 unique syndromes generated from MWPM.

5.2 Graph Neural Network Decoder

Here we present the main results for the GNN decoder. First, the performance of the best GNN model is shown and benchmarked against the MWPM decoder. Followed by this is an analysis of the time complexity of the trained GNN decoder with respect to the code distance.

The final best performing GNN architecture using complete graphs was trained on the dataset consisting of 500 000 unique syndromes sampled from depolarizing noise on the $d = 7$ surface code with an error rate of 11%, with target equivalence classes predicted by EWD. The training progress for this model and dataset is shown in Figure 5.5, where a training/validation split of 80%/20% was used, with Adam [23] as optimization algorithm using an initial learning rate of 0.01. The learning rate was manually decreased, which gave slight improvements to the validation accuracy. After 150 epochs, training was concluded to have converged. The model reached a final validation accuracy of around 89% compared with the EWD target labels.

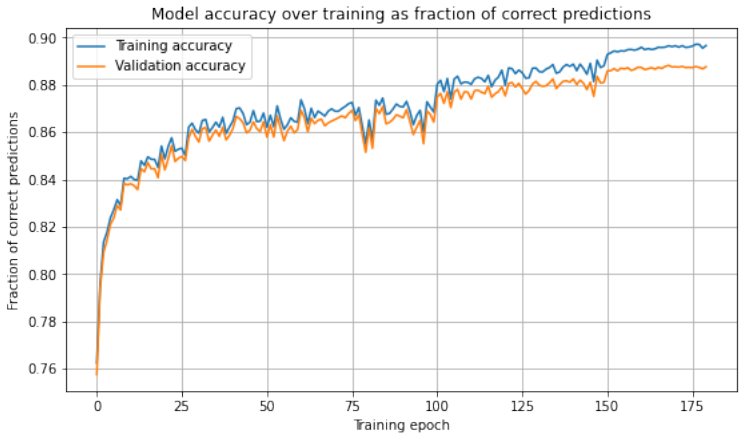


Figure 5.5: Training progress of the GNN decoder with a dataset consisting of 500 000 unique syndromes sampled from the depolarizing noise model at a physical error rate of $p = 11\%$. The learning rate was manually lowered, after 100 and 150 training epochs, at which points momentary gain in accuracy can be seen.

5.2.1 Decoding Performance

The logical error rate as defined in of the trained GNN decoder was evaluated from decoding 100 000 syndromes at each physical error rates p, with values of p ranging from 1% to 15%, and benchmarked against the MWPM decoder applied to the same syndromes. Figure 5.6 shows the GNN decoding performance at code distance $d = 7$, the same code distance that the model was trained on. It is seen that the GNN network performs worse than the MWPM decoder at all physical error rates p, having a higher logical error rate, but has a similar performance at low error rates.

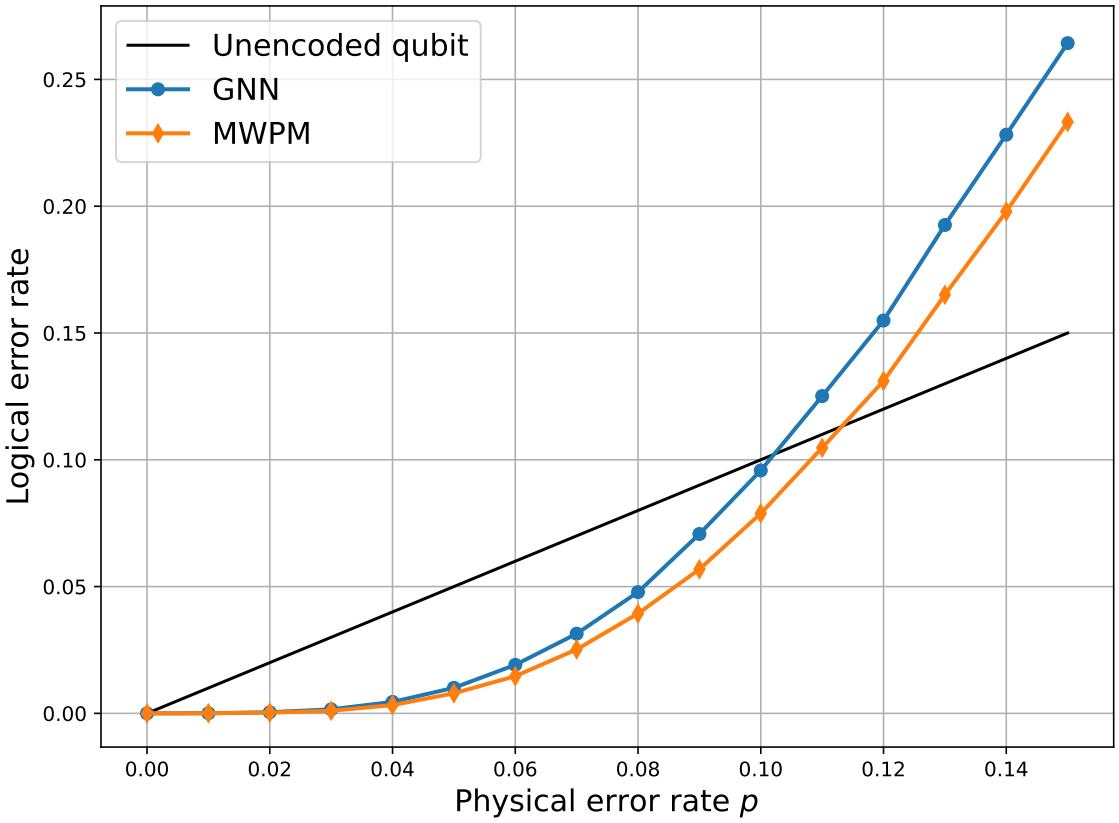


Figure 5.6: Logical failure rate as a function of physical error rate p for code distance $d = 7$, using the GNN decoder trained on $d = 7$ data benchmarked against the MWPM decoder. The GNN decoder performs worse than the MWPM decoder, especially at higher error rates.

Next, the same model, trained at code distance $d = 7$, was used to decode syndromes on a smaller code at $d = 5$. In this case, the GNN decoder performs close to identically to the MWPM decoder. This confirms that the GNN decoder is able to generalize to smaller code distances than ones seen during training. This shows some indication of the GNN being able to learn a generalized representation of syndromes, and decoding them with an accuracy comparable to MWPM. However, smaller code sizes have a smaller syndrome state space, and are generally easier to decode. As such, it would be of more interest to see if the GNN is able to generalize to *larger* code distances.

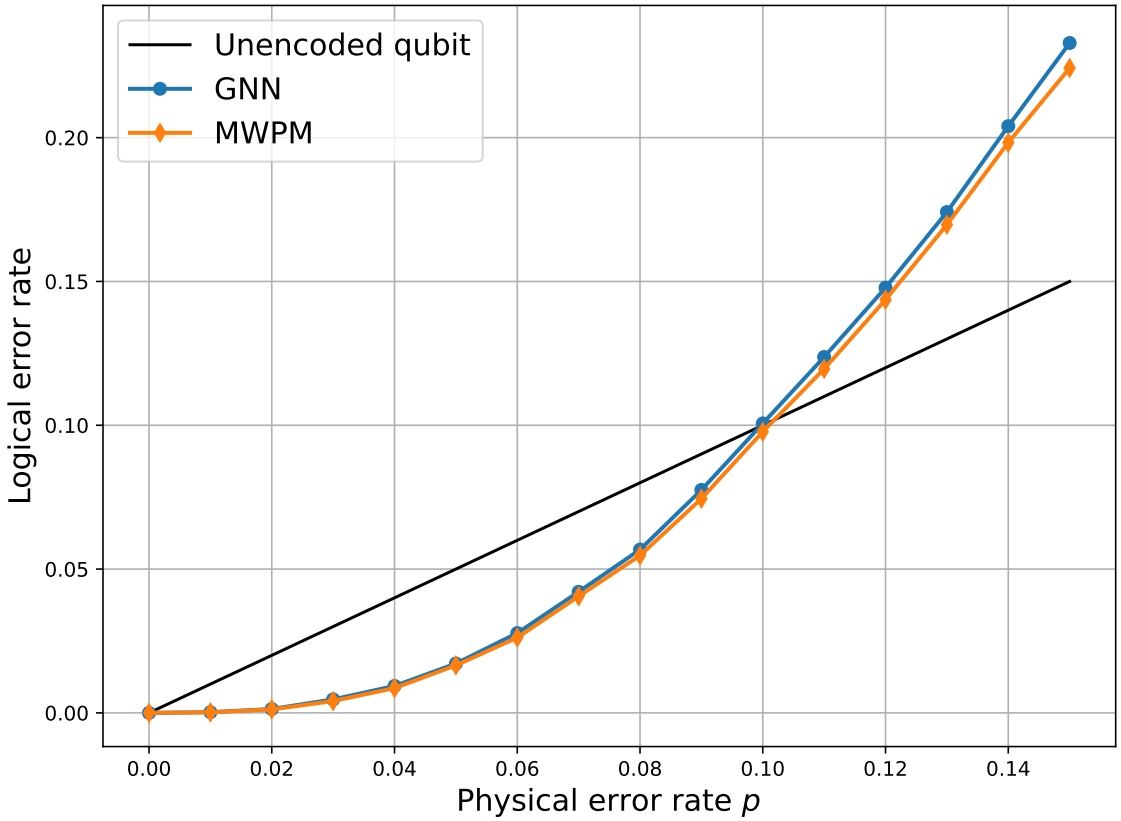


Figure 5.7: Logical failure rate as a function of physical error rate p for code distance $d = 5$, using the GNN decoder trained on $d = 7$ data benchmarked against the MWPM decoder. The GNN decoder performs almost identically to the MWPM decoder, albeit slightly worse, even though it has not been trained on data generated at $d = 5$.

In Figure 5.8, the decoding performance of the GNN model trained at $d = 7$ is shown for code distances $d = \{5, 7, 9\}$, to show the GNN results from Figure 5.6 and Figure 5.7 together with the performance at a larger code size than was used during training. Comparing the results at $d = 5$ and $d = 7$ shows that below error rates of approximately $p = 11\%$, the logical failure rate on $d = 7$ is lower than for $d = 5$, beyond which the opposite is true. This could be interpreted as the threshold where investing in a larger code distance (which requires more physical qubits to encode a single logical qubit) becomes favorable. However, this trend does not continue for $d = 9$. At low error rates below such a threshold, decoders are expected to perform better as the code distance increases. As such, it is clear that the GNN does not generalize as well to larger code distances as it does to smaller.

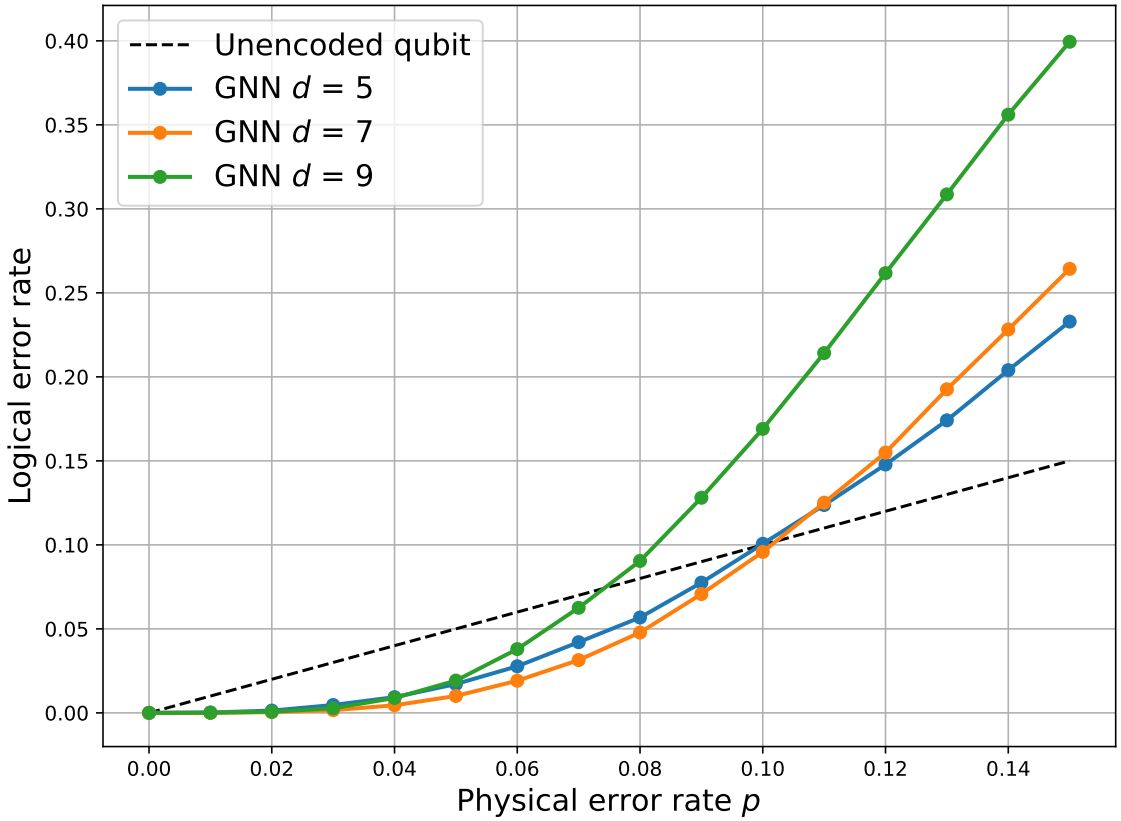


Figure 5.8: Logical failure rate as a function of physical error rate p for code distances $d = \{5, 7, 9\}$, using the GNN decoder trained on $d = 7$. The performance at $d = 9$ indicates that the model generalizes well to larger code distances for low error rates.

5.2.2 Decoding Time Complexity of the Trained Model

Next, we investigate how the runtime of the GNN decoder scales as the code distance increases. While accuracy is one important measure of decoders, a fast runtime is crucial as well due to the strict delay requirements when correcting errors in possible future real quantum computing systems. While optimizing neural network takes substantial time, neural network based decoders have the potential to achieve fast decoding times once trained. Here, we only consider the runtime, and not the decoding accuracy, as the number of samples used in the runtime analysis were too few to accurately estimate a logical error rate. The GNN decoding time depends on the number of nodes in the graph representation. Even for a constant code distance, the number of nodes will vary depending on the syndrome that was sampled, which complicates the runtime analysis and requires averaging over multiple syndromes at the same code distance. Additionally the number of nodes also depend on the error rate, however here runtime for a constant error rate of $p = 0.01$ was considered to determine the runtime scaling solely with respect to code distance.

Average decoding times were estimated by measuring the time spent to evaluate predictions of equivalence classes by the decoders for multiple randomly generated

syndromes at increasing code distances with a constant error rate of $p = 0.01$. This was done both for the GNN decoder using complete graphs, where every pair of nodes in the graphs are connected with an edge, as well as for the GNN decoder when using the sparse graph representations where each node is only connected to it's $m = 5$ nearest neighbours. The decoding time scaling of the GNN, as well as the time to construct input graphs from syndromes, is shown in Figure 5.9, and is compared with the operation time of the local matching variant of PyMatching [24], a fast implementation of the benchmark MWPM decoder, all run on the same machine using a CPU for all computations.

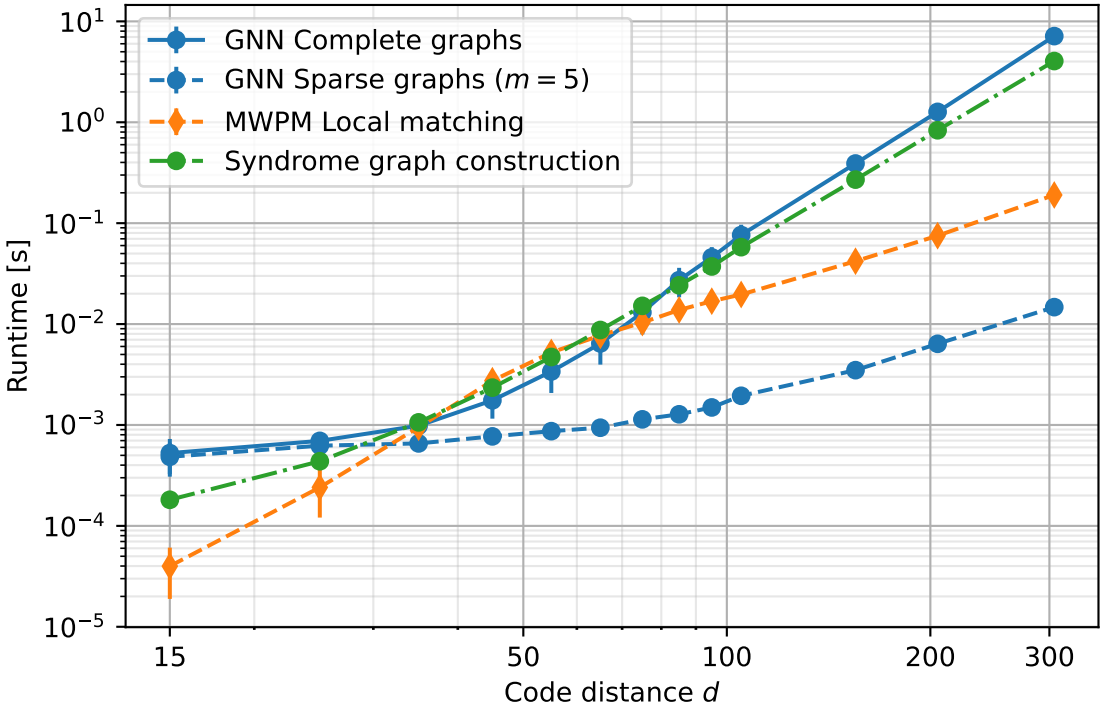


Figure 5.9: Experimentally determined average decoding times of the GNN decoder, as well as the time to construct graph inputs from syndromes, with increasing code distance at a physical error rate of $p = 0.01$. Shown is both the decoding time when using complete graphs, as well as sparse graphs with $m = 5$ nearest edges per node. For comparison, the average decoding time of the local matching variant of PyMatching [24], a computationally efficient implementation of the MWPM decoder, is also shown.

Firstly, it should be noted that the GNN has a significant overhead time at the smallest code distances. This is likely to depend on how the GNN was implemented in the code, which was not done in a way to necessarily minimize absolute runtime, in contrast to the PyMatching implementation of MWPM which is more heavily optimized for runtime.

What is more interesting is how the decoders scale with with the code distance.

At low code distances, the runtime with sparse and complete graphs are virtually identical, which is expected as the average number of nodes in each graph is small, and that the limit of $m = 5$ nearest edges will result in sparse graphs that have similar connectivity to the complete graphs.

For increasing code distances, it is clear that the decoding when using complete graphs scales poorly, with roughly two orders of magnitude longer operation time compared to the sparse graph implementation. The GNN decoder using complete graphs scales worse than the MWPM algorithm. Using a sparse graph representation shows how the decoding time of the GNN could be improved upon. However, the sparse graph representation is not necessarily favorable in terms of accuracy at these large code distances where accuracy estimations have not been made. Further studies focusing on decoding larger codes would be needed to draw conclusions about how the edge density of the graphs affect the decoding accuracy at very large code distances.

In general, the time complexity of the GNN implemented here is limited by the GraphConv layers, which iterate over all edges connected to each node. In a complete graph with n nodes, each node has $(n-1)$ edges, whereas in the sparse graph representation with only $m = 5$ nearest neighbour edges, there are (on average) only m edges per node. Limiting the number of edges to a constant value m could then improve the expected worst case runtime of the GraphConv layers from $\mathcal{O}(n \cdot (n-1)) = \mathcal{O}(n^2)$ to $\mathcal{O}(n \cdot m)$. As the number of nodes n in the worst case scales with the code distances squared, the time complexity of the GNN is then expected to be $\mathcal{O}(d^4)$ with complete graphs, and $\mathcal{O}(d^2 \cdot m)$ with sparse graphs.

Furthermore, at large code distances, it is then expected that the decoding time of the GNN decoder (a single pass through the neural network) and the graph construction depend on the code distance d as power functions $T(d) = Cd^{\alpha}$. This is confirmed by the results seen in Figure 5.9, where the decoding times appear as linear in the log-log graph at large code distances.

To numerically evaluate the runtime scaling exponents α, linear regression was applied to the logarithms of the empirically determined runtimes and code distances presented in Figure 5.9. This can be seen as calculating the slope of the decoding time as a function of code distance in the log-log graph. For this analysis, only the runtimes at code distances $d > 100$ were used to give an accurate estimate of the runtime scaling at large code distances, as the runtime at small distance codes showed greater variance due to the limited number of errors occurring at a physical error rate of $p = 0.01$. The numerical estimation of the exponents α of the empirical runtime scaling seen in Figure 5.9 are shown in Table 5.1.

For complete graphs, the empirically estimated runtime scaling at large code distances was found to be $\mathcal{O}(d^{4.25})$. Using sparse graphs reduces the empirical time complexity of the GNN decoder to $\mathcal{O}(d^{1.91})$. This is due to the number of edges being limited to the fixed value m, which is significantly lower than the average number of nodes in the graphs at large code distances. By limiting the connectivity of the graphs, a significant improvement in time complexity is thus made, possibly without the sacrifice of predictive accuracy based on the indicative results seen ear-

lier in Figure 4.5 where sparse and complete graphs perform similarly at a $d = 7$ code with a high error rate of $p = 11\%$ (resulting in graphs with a larger number of nodes on average than the cutoff of $m = 5$).

Decoding method	α
GNN complete graphs	4.25 ± 0.03
GNN sparse graphs	1.91 ± 0.12
MWPM	2.13 ± 0.07
Graph construction	3.99 ± 0.01

Table 5.1: Empirically determined exponents α of the decoding times and graph construction seen in Figure 5.9 under the assumption that they behave as power functions $T(d) = Cd^{\alpha}$ with respect to code distance d at large distances ($d > 100$), where C is some constant. Computed with linear regression of the logarithms of the measured runtimes and code distances at $d \in \{105, 155, 205, 305\}$. The uncertainty represents one standard error.

It should be noted, however, that this comparison only considers the execution time of the actual neural network. Prior to this, a graph must be constructed from the syndrome to be given as input to the GNN. The time complexity of the graph construction is limited by the computation of edge weights, both when using complete graphs (where all edges are included) and for the sparse graphs (where the m nearest nodes must be determined for each node). In this work, the graph construction was done using a naive algorithm that considers every pair of nodes in the graph and calculates the corresponding edge weight from the distance between the defects according to Equation 4.2. This naive graph construction has a similar time complexity to the GNN operating on complete graphs, with an estimated runtime exponent of $\alpha = 3.99$ as seen in Table 5.1.

As the graph construction is a necessary step of the GNN decoder that should be included in the time complexity analysis, the total execution time of the GNN decoder with sparse graphs in our implementation – from syndrome matrix to prediction – is dominated by the graph construction, with an empirical time complexity of $\mathcal{O}(d^{3.99})$. If using sparse graphs, it would then be necessary to implement a more efficient graph construction algorithm that does not consider all possible edges in the graph when calculating edge weights. One such possible algorithm is Dijkstra's algorithm, which can be used to find the m nearest nodes for each source node in the graph [25]. This approach would rely on traversing a weighted graph connecting all possible defects, however constructing such a matching graph would only have to be one once for a given code distance, prior to running the graph conversion and decoding of syndrome measurements, and would not affect the actual decoding time. Implementing a faster algorithm for constructing the input graphs, in combination with the sparse graph representation, could then possibly result in a GNN that has favorable scaling, even with respect to the fast local matching variant of the MWPM-based decoder PyMatching, which was found to have an empirical time complexity of $\mathcal{O}(d^{2.13})$.

6

Conclusions and Outlook

In this chapter we go deeper into similarities and differences between the two networks and discuss prospects.

6.1 General Conclusions

Two neural network based decoders were developed and implemented to decode errors in the rotated surface code. By training a CNN model on data generated by a slow but accurate algorithm, EWD, it was able to outperform a faster but slightly less accurate, established reference decoder, MWPM. The GNN model was able to reach a performance level similar to that of MWPM for small surface codes, but was not able to outperform MWPM. However, the GNN model shows signs of being able to learn a generalized representation of syndromes, managing to retain performance at smaller code distances than ones seen during training which was proved to be a harder task for the CNN model. The generalizability to larger code distances was however limited for both of the models.

Using sparse graph representations for the GNN model was shown to yield a significantly faster run time than when using complete graphs, potentially allowing for a GNN decoder whose decoding time scales better with code distance than the MWPM decoder. However, the total decoding time of the GNN decoder using sparse graphs is limited by the time required to construct graphs from syndromes. To capitalize on the decreased computational complexity of using sparse graphs, a faster algorithm for determining which edges should be included in the graphs is required. For future studies, a possible alternative to achieve this is applying Dijkstra's algorithm for finding the shortest path between nodes in a weighted graph with a preconstructed matching graph, similar to the method used in the local matching version of PyMatching [24].

6.2 Training at Larger Code Distances and Higher Error Rates

While the models studied here are able to decode arbitrarily large codes in principle, the results show that it is likely necessary to train the models at larger code distances in order to achieve better performance on larger codes. Training against

the near-optimal EWD decoder proved to be effective at d=7, with the CNN outperforming the MWPM decoder at this code distance. An interesting continuation of the study is to see if the same approach can be brought to larger code distances, or if the increasing syndrome state space and slow decoding time of EWD makes this approach less feasible for larger systems.

Alternatively, one could apply a different data sampling method when training the decoders presented in this work at larger code distances; such as training against the true equivalence classes of the underlying errors (without a reference decoder) with either a much larger set of randomly generated syndromes, or with an unlimited set of continuously generated syndromes where each batch of training data is freshly generated, as done in previous studies of neural network decoders [12, 13].

Furthermore we saw in Figure 5.3 that training a CNN on a higher error rate also was beneficial when classifying lower error rates, yet it is reasonable to believe that training on a too high error rate would eventually result in a lower accuracy for low error rates. An interesting study would thus be to examine and trying to find an optimal error rate to train on that yield the highest accuracy for lower error rates.

6.3 Matrix Representations of Syndromes

As mentioned previously, a limitation for the project was to only use the default matrix representation of the syndromes in the CNN, meaning that no distinction is made between X errors and Z errors. In an extension of the project it could be worth looking at finding a matrix representation where a distinction is made between the two kinds of defects. They could be represented by different numbers for instance. Another option considered was to represent the data as a three dimensional array with two layers. The first layer would contain 1's where an X-defect is measured and the second layer contains 1's where a Z-defect is measured, similar to how colours are represented in three different channels in a regular rgb-picture.

The matrices also contain zeros where no qubit actually resides in order to form square matrices. It could be worth considering other means of filling out these spaces since there is a difference between a zero that could potentially be a syndrome and a zero where a syndrome could never occur. It might be useful for the network to be able to differ between these two instances, using for instance negative numbers for the fillings instead of zeros is something to be considered in the future.

6.4 Noise Models and Imperfect Syndrome Measurements

This study only considered the depolarizing noise model. There are several other noise models which may be more realistic representations of errors in real quantum systems, such as biased noise or spatially inhomogeneous noise. A possible area for future work could be to evaluate the proposed decoders on such alternative noise models. Additionally, this work only dealt with data qubit errors. In experimental

implementations of error correcting codes, it is possible that the decoding also must consider errors on the ancilla qubits measuring the syndromes. The decoder must then consider multiple consecutive layers of repeated syndrome measurements. This is a problem that could possibly be suitable for a GNN decoder, as graphs can be constructed to include the information from the different layers. In a CNN decoder this problem could be tackled with three dimensional input data where different layers in an array represent different layers of qubits.

6.5 Neural Network Based Decoders Hereafter

Machine learning assisted quantum computing is a growing field gaining more and more attention. Our project enforces the statement that it is an area worth investing in. We show that a machine learning model can outperform common decoding algorithms and expand on ways to improve and further develop such models.

Bibliography

[1] Peter W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[2] Lov K Grover. "A fast quantum mechanical algorithm for database search". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 1996, pp. 212–219.

[3] Richard P Feynman. "Simulating physics with computers". In: *Feynman and computation*. CRC Press, 2018, pp. 133–153.

[4] Steven M Girvin. "Introduction to quantum error correction and fault tolerance". In: *arXiv preprint arXiv:2111.08894* (2021).

[5] Michael A Nielsen and Isaac L Chuang. "Quantum computation and quantum information". In: *Phys. Today* 54.2 (2001).

[6] Austin G Fowler et al. "Surface codes: Towards practical large-scale quantum computation". In: *Physical Review A* 86.3 (2012), p. 032324.

[7] Eric Dennis et al. "Topological quantum memory". In: *Journal of Mathematical Physics* 43.9 (2002), pp. 4452–4505.

[8] Sebastian Krinner et al. "Realizing Repeated Quantum Error Correction in a Distance-Three Surface Code". In: *arXiv preprint arXiv:2112.03708* (2021).

[9] Zonghan Wu et al. "A comprehensive survey on graph neural networks". In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24.

[10] Philip Andreasson et al. "Quantum error correction for the toric code using deep reinforcement learning". In: *Quantum* 3 (2019), p. 183.

[11] David Fitzek et al. "Deep Q-learning decoder for depolarizing noise on the toric code". In: *Physical Review Research* 2.2 (2020), p. 023230.

[12] Savvas Varsamopoulos, Koen Bertels, and Carmen Garcia Almudever. "Comparing neural network based decoders for the surface code". In: *IEEE Transactions on Computers* 69.2 (2019), pp. 300–311.

[13] Ramon WJ Overwater, Masoud Babaie, and Fabio Sebastiano. "Neural-Network Decoders for Quantum Error Correction Using Surface Codes: A Space Exploration of the Hardware Cost-Performance Tradeoffs". In: *IEEE Transactions on Quantum Engineering* 3 (2022), pp. 1–19.

[14] Spiro Gicev, Lloyd CL Hollenberg, and Muhammad Usman. "A scalable and fast artificial neural network syndrome decoder for surface codes". In: *arXiv preprint arXiv:2110.05854* (2021).

[15] Paul Baireuther et al. “Machine-learning-assisted correction of correlated qubit errors in a topological code”. In: *Quantum* 2 (2018), p. 48.
[16] Xiaotong Ni. “Neural network decoders for large-distance 2d toric codes”. In: *Quantum* 4 (2020), p. 310.
[17] Christopher Chamberland and Pooya Ronagh. “Deep neural decoders for near term fault-tolerant experiments”. In: *Quantum Science and Technology* 3.4 (2018), p. 044002.
[18] Michael M Bronstein et al. “Geometric deep learning: going beyond euclidean data”. In: *IEEE Signal Processing Magazine* 34.4 (2017), pp. 18–42.
[19] Christopher Morris et al. “Weisfeiler and leman go neural: Higher-order graph neural networks”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 33. 01. 2019, pp. 4602–4609.
[20] Karl Hammar et al. “Error-rate-agnostic decoding of topological stabilizer codes”. In: *arXiv preprint arXiv:2112.01977* (2021).
[21] Matthias Fey and Jan E. Lenssen. “Fast Graph Representation Learning with PyTorch Geometric”. In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019.
[22] Fabrizio Frasca et al. “Sign: Scalable inception graph neural networks”. In: *arXiv preprint arXiv:2004.11198* (2020).
[23] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).
[24] Oscar Higgott. “PyMatching: A Python package for decoding quantum codes with minimum-weight perfect matching”. In: *ACM Transactions on Quantum Computing* 3.3 (2022), pp. 1–16.
[25] Edsger W Dijkstra et al. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

DEPARTMENT OF PHYSICS
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2022
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY