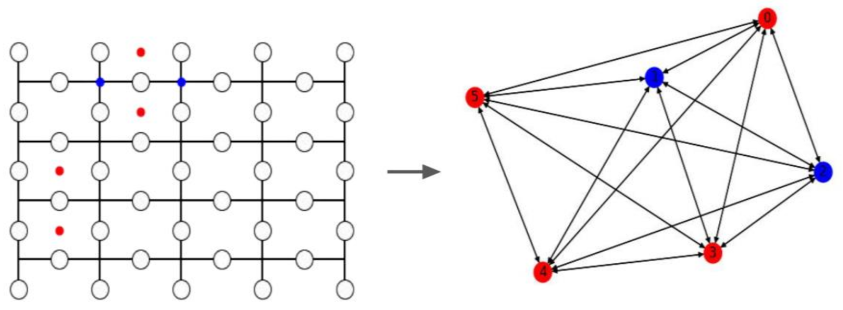
DEPARTMENT OF PHYSICS

QUANTUM ERROR CORRECTION USING GRAPH NEURAL NETWORKS



Valdemar Bergentall

Degree project for Master of Science (120 hec) with a major in Physics
2021, 120 HEC
Second Cycle

Master's thesis 2021

Quantum error correction using graph neural networks

VALDEMAR BERGENTALL

UNIVERSITY OF GOTHENBURG

Department of Physics
University of Gothenburg
Gothenburg, Sweden 2021

Quantum error correction using graph neural networks
VALDEMAR BERGENTALL

Supervisor: Mats Granath, Department of Physics, Gothenburg University
Examiner: Johannes Hofmann , Department of Physics, Gothenburg University

Master's Thesis 2021
Department of Physics
University of Gothenburg

Cover: Schematic figure showing how a syndrome of the surface code is mapped to graph. Where blue nodes represent vertex defects and red nodes represents plaquette defects.

Typeset in LaTeX
Gothenburg, Sweden 2021

Abstract

A graph neural network (GNN) is constructed and trained with a purpose of using it as a quantum error correction decoder for depolarized noise on the surface code. Since associating syndromes on the surface code with graphs instead of grid-like data seemed promising, a previous decoder based on the *Markov Chain Monte Carlo* method was used to generate data to create graphs. In this thesis the emphasis has been on error probabilities, $p = 0.05$, 0.1 and surface code sizes $d = 5, 7, 9$. Two specific network architectures have been tested using various graph convolutional layers. While training the networks, evenly distributed datasets were used and the highest reached test accuracy for $p = 0.05$ was 97% and for $p = 0.1$ it was 81.4%. Utilizing the trained network as a quantum error correction decoder for $p = 0.05$ the performance did not achieve an error correction rate equal to the reference algorithm *Minimum Weight Perfect Matching.* Further research could be done to create a custom-made graph convolutional layer designed with intent to make the contribution of edge attributes more pivotal.

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

Acknowledgements

First I would like to thank my supervisor, Mats Granath, for guidance and continuous discussions regarding possible advantages that could be made to improve my study. In addition I would like to thank Evert van Nieuwenburg and Basudha Srivastava for joining in on weekly meetings giving me valuable feedback. Final thanks to Karl Hammar who helped me with the *Markov Chain Monte Carlo*-decoder

Contents

List of Figures

List of Tables

1 Introduction

This introduction will be divided into two parts, the first being a background of quantum computers and the most significant problems with quantum computing and the second part will be an introduction to Geometric Machine Learning.

1.1 Quantum Error Correction

With the first quantum computers being built at this time the road ahead to a fully functioning and reliable quantum computer is still distant. Not only is it a theoretical challenge to determine how to best construct it but also the practical part of assembling. Over the last years the enthusiasm regarding quantum computers has skyrocketed and almost all big tech companies want to be in the frontline. In this master thesis project focus has been on one of the key obstacles regarding a quantum computer: correcting errors.
Comparing classical- to quantum computers the operation differs notably, since in a classical computer information is stored in binary form, and the error that occurs is then only the simple bit-flip, while in the quantum computer the quantum bits can be in a superposition of 0 and 1 and in addition to the bit-flip error, a relative phase exists where errors can occur. With this said one could state that while the binary operation that takes place in a classical computer is simple, it would indicate that the correction to such error should also be rather simple, and with the more complex operation in quantum computers the correction of an error should be harder. While this of course is one part of the struggle with quantum error correction, this is not the only one.

In classical error correction what is most commonly used is so called majority voting, which means that instead of having single bits, each bit is encoded into a multiple of physical bits which leads to the possibility that, if an error occurs on one of the physical bits, a majority voting is done to see whether this is an actual logical error or not. One thing to note in the classical case is that a copy or clone of a bit could be produced. This becomes a problem since the non-cloning theorem states[1] that in the realm of quantum mechanics there is no unitary operator that could act on a state and clone it. Thus, the idea of using majority voting is not a possibility. Another difficulty is measuring the states since in quantum mechanics, when a measurement of a state is done, the state collapses into the eigenstate of the measured observable. For a long time this was seen as dead-end for a practical quantum computer[2]. However, in the 90s Peter Shor introduced the idea of error correcting code which included nine-qubits[3]. This code was able to correct single-qubit errors. Since then the interest reignited and multiple of other encoding of qubits have been theorised, such as Kiteav's surface code[4] which is the encoding used in this thesis.

1.2 Geometric Machine Learning

In the last decade the field of machine learning has found its way to a wide range of sciences. It has somewhat become a crossroad where data science meets every other science profession resulting in many revelations, from image recognition[5] to finding out whether a molecule is suited for a antibiotic[6].

Focusing on geometric machine learning, we first want to resonate the purpose of it. In standard convolution neural networks (CNNs) the data is presented on a grid making it euclidean, where for example pixels could be viewed in a euclidean way. In some cases the euclidean way of presenting data meets its restrictions. For example, mapping information about e.g a molecule's connections, features of each atom etc. to a grid would be rather inefficient compared to mapping it to a non-euclidean structure such as an actual graph where the nodes could represent the atoms with node features mirroring the attributes of the atom but also including the connection between the atoms as edge attributes. Thus we will end up with both the structure compositions and the essence of the relation composition. Now the question is if there are smart operations that could help us with the machine learning on them.

In recent years several graph convolutions methods have been theorized and tested[14][15], and have shown great results compared to standard CNN. These different operations will be gone through in detail in the graph neural network section to give a sense of what could be the strengths and what could be the shortcomings.

Another topic within machine learning is the three paradigms that are used to process the data. The paradigms are **supervised learning**, **unsupervised learning** and **reinforcement learning**[7]. In supervised learning each data sample has a corresponding target label, using the labeled data to conclude broad information of the data and then utilize a trained model to predict labels on unlabelled data. This is the paradigm that is used for most classification tasks and it is the one used in this thesis. In unsupervised learning the data samples do not have labels connected to them, thus the learning is more of a cluster understanding, meaning that the machine learns to separate information to different clusters. Before mentioning reinforcement learning there is a sub-branch within unsupervised learning called semi-supervised learning where just a fraction of the data have labels. Then we have reinforcement learning which makes use of a reward system: The model is rewarded based on its actions and outcomes. This is used when training models to perform at super-human level, e.g. in games such as go[8].
Both supervised learning and Reinforcement learning have been used for quantum error correction [9][10].
Now knowing the paradigms we can categorise the typical tasks done with geometric

machine learning. In this thesis, as already mentioned, the attention is focused on graph classification, which falls under supervised learning. With graph neural networks node classification is also common, this task often falls under semi-supervised learning since here, instead of having multiple graphs it could only be one graph with a great amount of nodes. A good example hereof is a typical social network[11]. Now we do not need a target label for each node in the social network but still cluster the nodes together that have similar features. It is semi-supervised learning that has really sparked the interest for graph neural networks. There is also link prediction[12] where the edges between the nodes are of interest, this is also mostly occurring in semi-supervised learning.

2 Graph neural networks

The data structures used in geometric Machine learning are non-euclidean, for instance graphs, manifolds and more. Most common today is to map the data to a graph. We begin with a proper description of the graph data structure. A graph consists of a set of nodes and edges, $G = (N, E)$, where edges $e_{ij} \in E$ indicates a connection between two nodes, $i, j \in N = \{1, .., m\}$. The edges could be directed, meaning that depending on which of the nodes that is source and destination node respectively could indicate different features, or the edge could be undirected where the edge does not depend on where it starts and ends up.
To represent this data, the adjacency matrix A_{ij} is an appropriate choice. The adjacency matrix is a square matrix of the size of the number of nodes in the graph. Non-zero elements in the matrix indicates a connection between nodes. In the most simple case a connection is shown as one and no connection as zero. If there are no self-loops (an edge with the same destination as source), the adjacency matrix has no trace (zeros on diagonal). If the edge weight (e.g. length) is included, the ones could be replaced with the weights in the matrix.
Information stored in graphs is often in the form of a feature vectors $\vec{x}_i$. Every node has a node feature vector and its dimension is $\mathbb{R}^d$ where d is how many features each node holds.

2.1 Message-Passing

The idea with the graph data is now to calculate what is most commonly called node embeddings. We can think of it as mapping our nodes to an embedding space, and this node embedding recaps the information of the node features and the node's neighbourhood, $\mathcal{N}(i)$ [13]. To gather the information about the nodes neighbourhood we aggregate information from each connecting node. This process is called the message-passing in graph neural networks and can be formulated as:

$$\vec{x}_i^{k+1} = \gamma(\vec{x}_i^k, \, \square_{j \in \mathcal{N}(i)} \, \phi(\vec{x}_i^k, \vec{x}_j^k, e_{ij})) \tag{2.1}$$

where x is the node embedding, e represents edges, superscripts indicates which iteration in the message-passing procedure it is, and the subscripts specifies the node. γ and ϕ are differentiable functions i.e neural networks and $\square$ is a permutation invariant function such as summation or mean. The second term in the parentheses can be seen as the aggregation of information from the neighbouring nodes. The initial node embeddings ($\vec{x}^0$) are then the node features of the input graph. Essentially this

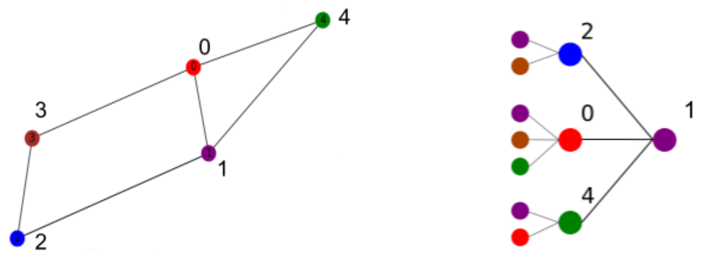


Figure 2.1: Example of a simple graph where we want to aggregate information to node 1 where two iterations are used in the message-passing procedure, as can be seen by the neighbouring nodes aggregating information form their adjacent nodes.

message-passing procedure is quite simple, by aggregating information from the local neighbourhood in the first iteration $k = 1$, and then in each subsequent iteration the respective neighbourhoods are included.
Given (2.1) the question arises how this □-function could be defined to gather the best information from neighbouring nodes. In the most simple cases this could just be a summation of the aggregated information. However, this could be a complication since with a large amount of degree differences between nodes (great number of fluctuations of neighbouring nodes) problems such as numerical instability arise. One possibility is a symmetric normalization of aggregated information. This could be written as

$$\vec{x}_i{}' = \mathbf{W} \sum_{j \in \mathcal{N}(i) \cup i} \frac{e_{j,i}}{\sqrt{d_j d_i}} \vec{x}_j \tag{2.2}$$

where $\mathbf{W}$ is a trainable weight matrix and d denotes the number of adjacent nodes of the specific node. It should be noted that here self-loops are also included. In (2.2) we can now see that the aggregated information is normalized by how many adjacent nodes each respective node has. This kind of normalization used was first introduced by Kipf and Welling in 2016 [14] and goes by the name Graph convolutional network (GCN). This is defined as node-wise and we can also define it in matrix form using the node feature matrix X which takes the size $\mathbb{R}^{d \times m}$ since that is how it is implemented in `pytorch geometric` which is a geometric deep learning extension library which include various convolutional- and pooling layers,

$$X' = \sigma(D^{-\frac{1}{2}} \tilde{A} D^{-\frac{1}{2}} X \mathbf{W}) \tag{2.3}$$

where X is the node feature matrix, A is the adjacency matrix where self-loops are included ($\tilde{A} = A + I$), $\mathbf{D}$ is a diagonal node degree matrix defined as $D_{ii} = \sum_j A_{ij}$ and σ is a non-linear activation function such as tanh.

A remark on this convolution layer is that it is isotropic, meaning that the aggregations obtained from all the neighbouring nodes are equally crucial, and this could lead to a lower performance since sometimes the aggregation should be more important from a specific neighbouring node. To deal with this a more elaborate layer could be used, namely GAT[15] (Graph Attention Layer). The GAT layer instead uses the attention mechanism which results in a selective feedback from the adjacent nodes.

Below we will again use $X = \{\vec{x}_1, \vec{x}_2, .., \vec{x}_m\}$ where m is the number of nodes and $\vec{x}_i \in \mathbb{R}^d$ meaning the number of node features. The output dimension post of this convolution layer is $\vec{x}_i{}' \in R^{d'}$, where d' could either be larger or smaller than d. The procedure of the layer is first having a linear transformation with trainable weights, then the attention mechanism is added. Let us write

$$\vec{x}_i{}' = \sigma(\sum \alpha_{ij} \mathbf{W} \vec{x}_j) \tag{2.4}$$

where α is the attention coefficient, which could be defined as

$$\alpha_{ij} = \frac{exp(\sigma(\vec{a}[\mathbf{W}\vec{x}_i \| \mathbf{W}\vec{x}_j])}{\sum_{k \in \mathcal{N}(i)} exp(\sigma(\vec{a}[\mathbf{W}\vec{x}_i \| \mathbf{W}\vec{x}_k])}. \tag{2.5}$$

$\vec{a}$ is a trainable attention vector, $\|\|$ is a concatenation operator, which is multiplied with the concatenation of the products of weight matrices and the node feature vectors. Here we can see that each exponential is divided with a summation of exponentials, this is called softmax and works as a normalization factor in this case. Finally the σ, which again is a non-linear activation function. Looking at the function of this layer it could be seen that it actually takes all the other nodes into account, meaning that structural information is not explicitly included. A simple change is to add the adjacency matrix after the concatenation so that attention coefficients between nodes that are not connected become zero.

$$\alpha_{ij} = \text{Softmax}(\sigma(a[\mathbf{W}\vec{x}_i \| \mathbf{W}\vec{x}_j])\tilde{A}_{ij}). \tag{2.6}$$

This is the description for one attention head, it is also possible to have multiple attention heads, we can simply write this as

$$\vec{x}_i' = \Big\|_{k=1}^{K} \sigma(\sum_j \alpha_{ij}^k \mathbf{W}^k \vec{x}_j) \tag{2.7}$$

where K is the number of heads. Note that here we get K independent attention coefficients.

Another layer that will be used in this work is the GraphConv layer defined as

$$\vec{x}_i{}' = \sigma(\mathbf{W}_1 \vec{x}_i + \mathbf{W}_2 \sum_{j \in \mathcal{N}(i)} e_{ij} \vec{x}_j) \tag{2.8}$$

in the `Pytorch Geometric` library, where in this case the aggregation is a summation over the nodes weighted by the edge weights. However the implementation of this in `Pytorch Geometric` allows the user to change the other permutation-invariant, differentiable functions such as a mean or maximum.

Now with a fundamental understanding of how the graph convolutional layers work there is difficulty to create deep-networks since after a certain amount of iterations of the message-passing procedure the node embeddings for each node become very alike, this is called over-smoothing[13]. There is no specific layer that deals with this directly but in recent years architectures of the networks have been proposed where instead of only relying on the output from the final convolution layer, a concatenation of results from previous layers is done. In this way both a larger depth could be used but then also keeping the behaviour of the node's previous attributes.

2.2 Graph Pooling Layers

In standard convolution neural networks pooling layers are essential, not only to reduce the data samples but also to harvest the key features of the data input making the network more receptive to new information.
The most simple versions of pooling-layers for graphs is called global-pooling where all node features in each separate graph is pooled. The ordinary global-pooling layers are mean- and max-pooling, where we can write the mean-pooling as

$$X' = \frac{1}{m}\sum_{i=1}^{m} \vec{x}_i \tag{2.9}$$

where m is the number of nodes in the input graph. Note that the dimension of X' is then compressed to $\mathbb{R}^d$ in global pooling.
However, global-pooling is not always the most effective and performs poorly with large graphs, since the substantial decrease of sample size also results in important features being diluted. There are other pooling methods such as as TopK pooling where particular nodes of each graph are selected depending on a score calculated for each node. This instead leads to a lesser compression of the nodes in the graph compared to the compression of graphs in global pooling. There are also pooling layers used specifically for node classification and link-prediction which sort nodes into specific clusters and then making use of both the graph and the cluster to pool the data.
In the thesis the focus will be on non-global pooling layers such as TopK-pooling because of the advantages it has on the graphs we are dealing with.

The procedure of TopK-pooling is as follows: a trainable projection vector called $\vec{p}$ is introduced, the product of the projection vector and the input feature matrix X is divided by the norm of $\vec{p}$ giving a score vector $\vec{y}$, corresponding to a score for each node:

$$\vec{y} = \frac{X\vec{p}}{||\vec{p}||} \tag{2.10}$$

The length of the score vector is the the amount of nodes in the input graph. The next step is to select k nodes to keep from each graph. The parameter k can either be fixed integer or a fraction of the total number of nodes depending on the wanted output. In this thesis we want the pooled graphs to have the same sizes, thus k will be fixed, while in other tasks when perhaps the graph sizes differ a lot, a fraction is more suitable so the impact of the pooling will not have a bias effect on specific graphs. The indices of the top scoring nodes are extracted and then used to calculate the newly pooled graph.

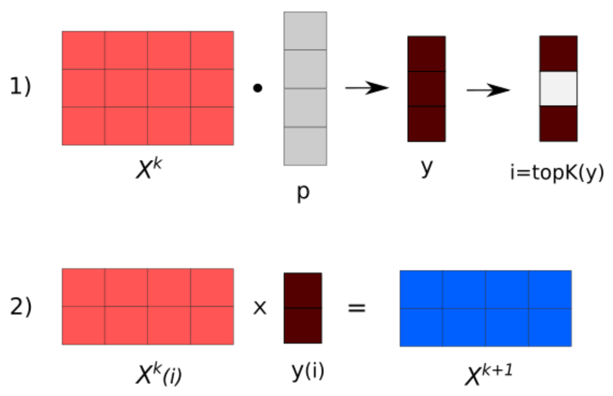


Figure 2.2: In 1) the node feature matrix X (with 4 features on 3 nodes) is multiplied with a projection vector p to achieve the score vector y. Then, in this case, ($k = 2$) the top two scoring indices are chosen to select the most important nodes. Then in 2) the important nodes from X are matrix multiplied with the score vector to receive the new pooled graph.

$$i = top_k(\vec{y}) \tag{2.11}$$

where top_k is an operator selecting the indices i, corresponding to the k highest scores.

$$X' = X(i)\vec{y}(i) \tag{2.12}$$

then using i to select which nodes to keep, also a re-scaling of the selected nodes are done with the score vector.
In Fig. 2.2 a toy-example of TopK-pooling is shown, where a graph with three nodes and each node has four features is pooled with $k = 2$. The two high scoring nodes are kept and the node features are updated depending on the scoring vector.

One theoretical disadvantage of TopK-pooling is that the connectivity of the graph is ignored. To include some more complexity to the determination of kept nodes, a layer named *Self-Attention Graph pooling* (SAG)[16] can be introduced. The principle of SAG-pooling is the same as TopK but the calculation of projection scores now depend on a convolutional layer. Replacing (2.10) with

$$\vec{y} = \mathrm{GNN}(X, A) \tag{2.13}$$

where GNN is the chosen graph convolutional layer, e.g using the GCN in node wise representation:

$$y_i = \sigma\Big(\sum_{j \in \mathcal{N} \cup i} \frac{e_{j,i}}{\sqrt{d_j d_i}} \vec{x}_j \vec{a}\Big) \tag{2.14}$$

where a is an attention score. Notice that y_i is just a scalar. The rest of the procedure is identical to TopK-pooling.

3
Quantum Error Correction

The fundamental formalism of quantum bits (qubits) and the most simple representation of errors that act on them are discussed. The general way of writing the state of a qubit is:

$$|\Psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{3.1}$$

where α and β are complex numbers and need to fullfill $|\alpha|^2 + |\beta|^2 = 1$. It can be seen from this equation that the qubit is a two-level system where the ground state is in $|0\rangle$ and the excited state in $|1\rangle$. If a measurement of the qubit is done the probability of it being 0 or 1 is $|\alpha|^2$ respective $|\beta|^2$. What the two-level also implies is that the information no longer is binary like a classical bit but a superposition, the superposition of states is the first indication of a quantum advantage. To visualize a single qubit the use of the Bloch-sphere is standard. In Fig. 3.1 the ground state (0) and the excited state (1) can be seen as the north- and south pole of the sphere respectively. When the Hadamard gate, H, which is used to create superposition, acts on 0 we will get the $|+\rangle$-state and is analogue for the 1 state we get the $|-\rangle$-state[18].

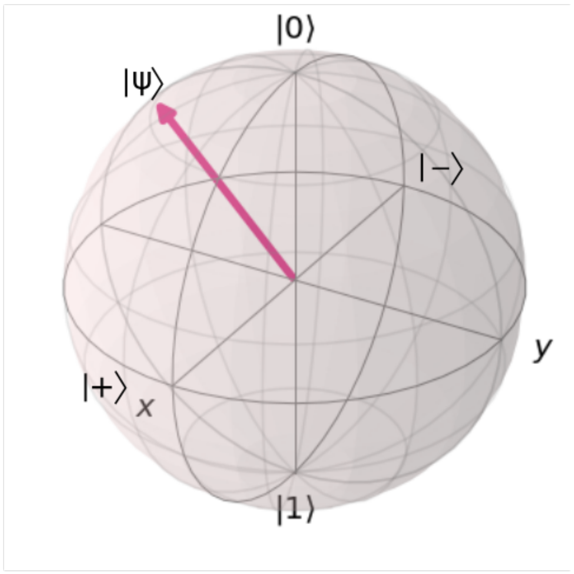


Figure 3.1: Qubit state viewed in the Bloch-sphere representation. The groundstate, $|0\rangle$ can be seen in the north pole and the excited state $|1\rangle$ in the south pole. The $|+\rangle$ is the received superposition state when acting with Hadamard on the groundstate and similar for $|-\rangle$ with the excited state.

The simplest representation of errors that can occur to single qubits are the quantum equivalent of the classical bit-flip. This bit-flip is represented as the Pauli-x operator(X) and the second error is the phase-shift, which is represented by the Pauli-z operator(Z). An operator Y is also an error and this arises from the product of X and Z (neglecting the overall phase) meaning both a bit-flip and a phase-shift error on the same qubit.

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \tag{3.2}$$

Acting with X on the $|0\rangle$ flips it into the $|1\rangle$ ($X|0\rangle = |1\rangle$). Acting with the phase shift it can be seen that the complex amplitude for the "one state" flips sign but otherwise stays the same as if we act on the general arbitrary state mentioned earlier.

We will now discuss the basics of error correction, beginning by defining some gates

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}. \tag{3.3}$$

These gates are control gates, where action to a second qubit is governed by the first qubit(control-qubit). The CNOT-gate will create an entanglement between two qubits, since entanglement is the closest thing we can do to clone a state. This is used for encoding our states.
We define an encoded state as:

$$|\Psi_L\rangle = CNOT(|\Psi\rangle \otimes |0\rangle) = CNOT((\alpha|0\rangle + \beta|1\rangle) \otimes |0\rangle) = \alpha|00\rangle + \beta|11\rangle \tag{3.4}$$

where the L stands for logical. Let us now introduce a bit-flip error on the first qubit, $|\widetilde{\Psi}_L\rangle = \alpha|10\rangle + \beta|01\rangle$. To extract this error from the code an ancillary qubit entangled with a parity operator Z_1Z_2 is applied to the entangled state as can be seen in Fig 3.2. The Z_1Z_2 operator allows us to measure the states and see whether an error has occurred. This is because it acts as a stabilizer operator meaning that it does not perturb the superposition but only adds a global phase factor

$$Z_1Z_2|\Psi_L\rangle = |\Psi_L\rangle, \quad Z_1Z_2\left|\widetilde{\Psi}_L\right\rangle = -\left|\widetilde{\Psi}_L\right\rangle. \tag{3.5}$$

The results of the measurement will show whether a bit-flip has occurred or not, however with this small encoding we can not conclude which qubit flipped.

With this knowledge we can go ahead and look at the so called surface code four-cycle [17]

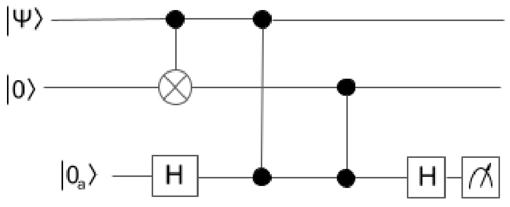


Figure 3.2: The encoded state shown in the circuit representation. First the CNOT gate is used to entangle the two states, afterwards a X-error occurs on one of the qubits. Then a Z_1Z_2 operator entangled with an introduced ancillary qubit, $|0_a\rangle$ is used to measure the parity. Note that for the CZ-gate it does not matter which qubit is the control one.

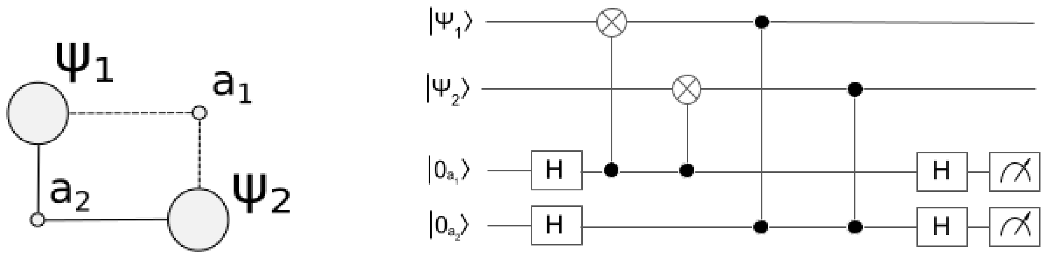


Figure 3.3: The first figure illustrates the representation of the smallest version of the surface code. Here $|\Psi_{1,2}\rangle$ are the states of the data qubit, which we do not want to disturb. $a_{1,2}$ are the ancillary qubits used to measure the parity. Note that the dashed line represents the CNOT gate and the solid line is the CZ gate. The second figure shows the corresponding surface code using the quantum circuit representation.

Looking at Fig. 3.3 we can see that the ancillary qubits a_1, a_2 are connected to both data qubits, $|\Psi_{1,2}\rangle$. This connection is a CNOT for the dashed line and a CZ for the solid line. This indicates that a_1 measures the stabilizer X_1X_2 and a_2 measures the stabilizers Z_1Z_2. However, this small code meets the same restrictions as the earlier encoded state, where we are only able to determine if an error occurred but not to which qubit, but what one can see is that this code is actually very scaleable. By Making a grid of four four-cycle surface codes we obtain the so called [[5,1,2]] code. This is often referred to as the perfect code since this is the least amount of physical qubits needed to encode a logical qubit that is protected from single-qubit errors. The numbers stand for the number of physical qubits that the enconding needs(5), the number of encoded (logical) qubits (1) and the Hamming distance(2). The Hamming distance is the distance between the logical qubit states. With the use of different sizes of the surface code it is sufficient to have an equation that

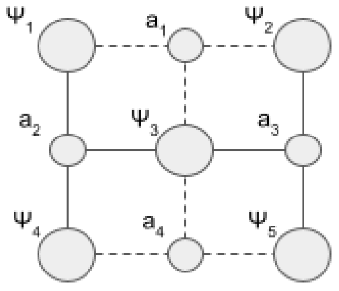


Figure 3.4: A surface code containing five physical qubits and four ancillary qubits, this is the least amount of qubits needed to make a surface code that is protected from single qubit errors.

states the different parameters:

$$[[p = d^2 + (d-1)^2, l = 1, d]] \tag{3.6}$$

The [[5,1,2]] code is visualized in Fig. 3.4, where we can see that the every ancillary qubit is connected to three data qubits. Stabilizers for these ancillary qubits are then:

$$X_1X_2X_3, \quad Z_1Z_3Z_4, \quad Z_2Z_3Z_5, \quad X_3X_4X_5 \tag{3.7}$$

To iterate why this code is protected from single qubit errors is that with three stabilizers for each ancillary qubit, we are now able to do measurements for every pair of qubits and based on outcome pin-point which ancillary qubit that has violated stabilizers. Ancillary qubits with violated stabilizers are called defects in the thesis. Where ancillary qubits with violated X-stabilizers are defined as vertex defects and ancillary qubits with violated Z-stabilizers are plaquette defects. A collection of defects is called a syndrome.
One set-back about the perfect code is that the logical errors are only of length 2. The logical operators are identified as operators that commute with the stabilizers. In this case $X_L = X_1X_4$ and $Z_L = Z_1Z_2$. These also anticommute with each other thus acting as Pauli operators on the code space. Meaning that the if multiple errors occurs on the code, there is a big chance of it being a logical error. Thus, larger code size is necessary to avoid the otherwise common occurrence of logical errors.

In Fig.3.5 the logical-X and logical-Z operators are shown for a surface code with size $d = 5$, where the row of X-operators is X_L and the column of Z-operators is Z_L. The X-stabilizers surrounding a vertex and Z-stabilizers surrounding a plaquette are also illustrated in the figure. The different error chains that occur on the surface code is categorised into equivalence classes. There are four equivalence classes for the planar surface code used in this thesis. The equivalence class of a error chain depends on whether it commutes or not with the logical operators, all chains that anticommutes with Z_L belong to class one, similar for chains anticommuting with

X_L belong to class two. Combining the X_L and Z_L we get Y_L, chains anticommuting with Y_L belong to class three. Class zero is then the chains that commutes with the logical operators.

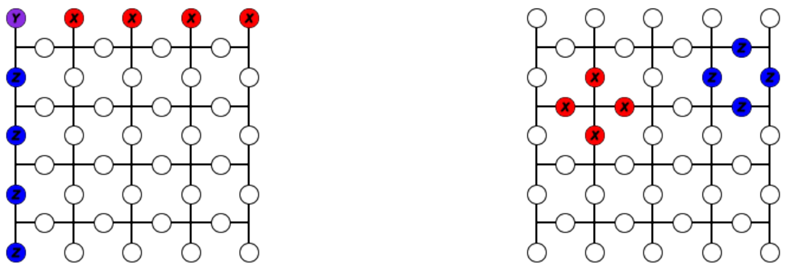


Figure 3.5: In the left figure a column of X-operators (red dots) define the logical-X operator and a row of Z-operators (blue dots) define the logical-Z. In the right figure the X-stabilizers surrounding a vertex and Z-stabilizers surrounding a plaquette are shown.

One challenge that comes with large surface codes is that some clever algorithm needs to decode the most probable error-chain. This is not an easy task since, with the knowledge of how stabilizers work, multiple error chains can correct the same syndrome and the amount of possible syndromes on the surface code increases exponentially with the size of the code. In Fig. 3.6, the same syndrome, containing two vertex defects is error corrected by three different error chains.

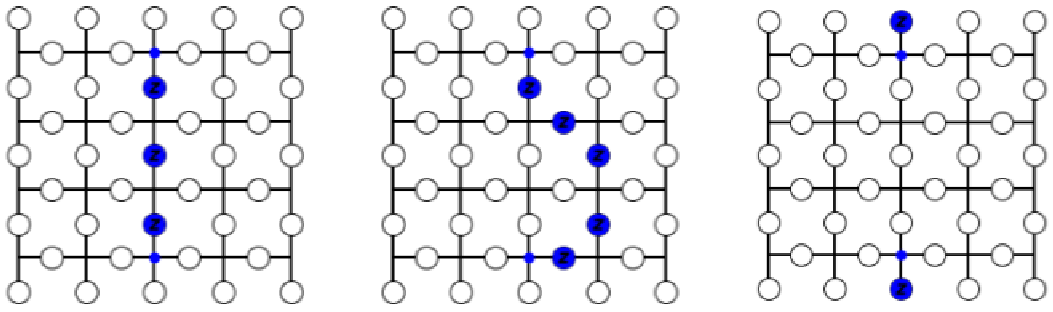


Figure 3.6: A two vertex defect syndrome on the planar code and three possible error-chains. Since there is no distinctive error-chain that corresponds to a syndrome the definition of equivalence classes is essential since the first two chains belong to class 0 while the last corresponds to class 2.

The standard algorithm used to decode syndromes is called *Minimum Weight Perfect Matching* (MWPM)[19]. MWPM is a rather simple algorithm that tries to match all the defects pair-wise and using the least amount of errors-corrections needed to eliminate the defects. MWPM is approximate, assuming Y-errors only occur as independent X- and Z-errors on the qubit. This a crucial set-back especially when it

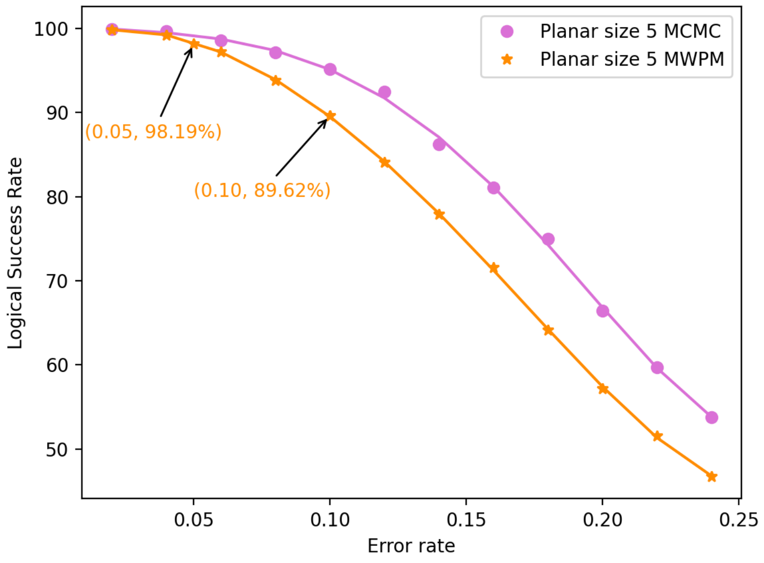


Figure 3.7: The success ratio of error correction versus the error probability (p) for planar code of size $d = 5$ with the MWPM algorithm and the MCMC-decoder. Highlighted success rates for MWPM shows that for $p = 0.05$ MCMC and MWPM are almost equal but for $p = 0.1$ the success ratio decreases significantly for MWPM. For depolarizing noise, $p_x = p_y = p_z = p/3$.

comes to codes of greater size and also when the error probability (p) is larger for the physical qubits. The algorithm does not depend on the probability rate. There is also another algorithm [20][21] which makes use of the *Markov-Chain Monte Carlo* (MCMC) procedure to determine the most probable error-chain. This algorithm makes use of the actual error-rate of the physical qubits and performs better than the MWPM as exemplified in Fig. 3.7

The procedure starts with a syndrome and a corresponding error-chain, the syndrome is fed to the MCMC algorithm and many chains that solve the syndrome is generated using the Metropolis-Hastings algorithm. After the generation of chains is done, the distribution of equivalence classes among the chains is extracted. While determining the distribtuion of error chains in the MCMC-algorithm the physical error probability is incorporated, meaning that the MCMC-decoder is dependent on the error probability. This is a significant change from the MWPM. Note that the output from the MWPM-algorithm is binary since only one chain is generated compared to the distributions from MCMC.

4
Methods

This chapter will focus on the creation of the dataset and how the mapping from syndrome to graph is done and the architecture of the two networks used for classifying the graphs. The training procedure is also explained.

4.1 Dataset creation

When creating a dataset it is important to find a good representation. Since the graphs should be analogous to the syndrome on the surface code, the fundamental property of how many defects the syndrome consists of is mapped to the number of nodes. The fact that there are two different types of defects (vertex and plaquette) which determine the equivalence class indicates that the defect type is significant to include. Thus, the first node feature is the defect type. Another crucial attribute to transfer from the syndrome is the positions of the defects relative to the edge where the equivalence class is determined

As previously stated the data used to create the graphs are obtained from a decoder based on the MCMC-method. The output of the decoder is the syndrome given in the form of two matrices where one matrix contains the positions of the vertex defects and the other matrix the positions of the plaquette defects. The equivalence class distribution is also in the output of the decoder. From the two matrices all necessary data to create the mentioned features of the graphs are obtained.

The position of a defect in this thesis are chosen to be the distance to the closest edge. In Fig. 4.1 an illustrative example of the distance to the edge is shown for two defects, where the single plaquette defect is created by a X-error either left or right of the defect and the single vertex defect is created by a Z-error either above or below the defect. Depending on which type of defect it is the reference edge is different, this feature seemed important to include considering that the logical error depends on this distance, and while having the distance to both edges can be argued to hold more information the plaquette defect does not contribute to an logical Z-error and equivalent for vertex defects and logical X-errors. Note that larger distance in the figure is presented as -1, this is due to a normalization done to the distance. The reason for a normalization is that datasets with graphs created from different sized surfaces codes want to be possible. While the different sizes does not prevent such datasets to be done, without a normalization the distance from the edge could affect the results. An example of the mapping from "distance to edge" to node feature is shown in (4.1).

$$[0.5,\ 1.5,\ 2.5,\ 3.5] \longrightarrow [0.5,\ 1,\ -1,\ -0.5]. \tag{4.1}$$

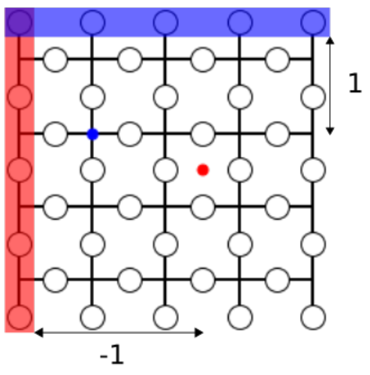


Figure 4.1: Two defects on a planar code with size $d = 5$. The edges for the specific type of defects are highlighted and the distance from the defects to its edge is shown in our normalized way.

Having determined the nodes and node features, the next step is the edges of the graph. Since all the defects depend on each other in the syndrome, edges between all nodes are included. The graph is thus a fully-connected graph. The Manhattan distances between all nodes are then calculated. However, the further away a defect is from another the less important it is, thus, the attribute distributed to the edge is the inverse Manhattan distance.

All components to create graphs have now been presented, but since the datasets are made for supervised learning, each graph needs to have an corresponding target label for training. The target label is the one hot encoded equivalence class obtained from the decoder.

In this thesis seven datasets are created, five with $p = 0.05$ and two with $p = 0.1$. With the lower error probability rate three datasets are created with the separate sizes, $d = 5, 7, 9$ and then two datasets containing a combination of sizes, 5 and 7 and finally 5,7 and 9 are created. For $p = 0.1$ there is one dataset for $d = 5$ and one for $d = 7$. To optimally train neural networks datasets containing evenly distributed target labels are desired. However with low error probabilities the rarity of syndromes belonging to equivalence class three is much higher than for the other classes, which leads to problems to achieve perfectly even distributions of classes and this also makes it difficult to create large datasets. The datasets created with $p = 0.05$ contains between 5000 and 6500 graphs, while datasets with $p = 0.1$ contains 15000 to 16000 graphs. Besides the evenly distributed target labels, all the graphs in the datasets are unique. This is done to improve the training of the network, since when the dataset is split into a training- and a test-dataset we do not

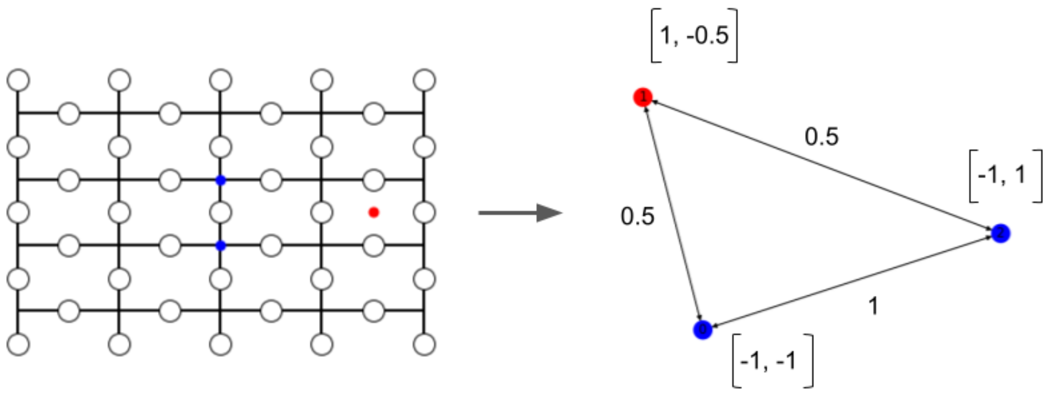


Figure 4.2: Schematic figure of the mapping from syndrome on the surface code to graph. The red node corresponds to the plaquette defect and blue nodes to the vertex defects. The node features and the edge attributes are visualized.

want to have graphs that appear in both subsets.
Important to note is that syndromes only containing one defect are neglected, since practical obstacles arise when graphs with no edges are included. Nevertheless, these are trivial to decode using only the type and distance to relevant edge.

Since the data is generated from the MCMC-decoder the best performance of the graph neural network is limited by the performance of this decoder. While the decoder has a much higher complexity, and is more computationally demanding than the MWPM algorithm the results are more accurate. This means that if our network shows a good accuracy on the MCMC-decoder data, the network could potentially outperform MWPM.

4.2 Network architecture

The network structures are somewhat different when working with graph networks compared to standard convolutional networks, mainly due to the problem of over-smoothing as mentioned earlier, which arises with the usage of stereotypical deep networks. There are two different network architectures tested in this thesis and they are quite similar. Both networks have the same core-structure which are two graph convolutional layers to create node embeddings.
In the first convolutional layer we want to increase the node feature dimension, while in the second layer the two models differ, the first model, "Model 1" also increases the dimension in the second layer while Model 2 maintains the dimension.
Posterior to the convolutional layer the node embeddings are pooled in two separate heads, the pooling layer in these models are TopK-pooling. Considering that the number of nodes in the graphs may vary immensely from one graph to another, pooling with a large k-value could lead to much of the data not being pooled at all

(k-value larger than number of nodes). To deal with this dilemma, after the TopK-pooling all graphs that contain less than k nodes get the remaining node number as empty nodes (nodes with zeros as node features). This way all graphs have the same size after pooling, thereafter the node feature vectors are squeezed such that the dimension $(1, kd)$ is achieved in each head, where d are node features.
This flattened vector is then fed to a Multilayered percepton (MLP). The MLP consists of three layers and between each layer is *tanh*-activation function. The final linear output dimension of the MLP is just a single number, which is activated by a Sigmoid function. This results in the final output of the head being a value between zero and one. The argument for using two different heads now becomes clear, since the equivalence classes 0,1,2 and 3 can be written in binary form as $0 = 00$, $1 = 01$, $2 = 10$, $3 = 11$ and then pointing out the first binary number in the sequence to represent a logical X-error or not and the second number for logical Z-errors. Then using these binary numbers as target labels for each head we can combine two binary classifiers to fit our problem.

To calculate the loss of the prediction of the network the *Binary-Cross Entropy* (BCE) loss-function is used, which is defined as

$$L_i = -(y_i \log x_i + \log(1 - x_i)(1 - y_i)) \tag{4.2}$$

where y is the target label and x is output of the network. Note that each head have a separate loss-function attached to them.

The second model while still having the same structure, also pool the information from the first convolutions layer and then concatenate the pooled graphs with the pooled graphs from the second layer, this way we include more node features and have a larger vector to process in the MLP. This type of method is inspired by [22], however here we do not include the global pooling of each batch of graphs. Again, the reason to include information from earlier layers is that in the event of over-smoothing, information from previous layers contributes to the output.

Various different graph convolutional layers could be used in these models but in this thesis the convolutional layers are chosen to be `Pytorch geometric`'s GraphConv. Model 1 has also been tested with the GAT-layer. Although not shown, other layers have also been tested, but with under performing results no further analysis was done with them.

For each dataset different parameters are used. The hidden channels parameter which governs the output size from the convolutional layer is adapted to the size of the surface code, since with a larger surface code more probable positions are introduced. Likewise for the k-value, where syndromes generated from larger codes or higher error rates includes more nodes.

The *Stochastic Gradient Descent* (SGD) optimizer was used. The momentum hyperparameter was 0.9 and the learning rate was set to 0.025 for both models. A learning rate scheduler was also implemented with the `Pytorch` library. Learning rate schedulers adjust the learning rate depending on some condition. The condition we use is that if the training loss does not decrease over an interval of 10 epochs the learning rate was lowered with 60%. This way the training could be more efficient because it is easier to descent into a narrow local minimum. For each training procedure 250 epochs were used, in some cases if the network had not yet fully converged by 250 epochs, it would be increased to 400 epochs. The dataset are split into to 80% training data and 20% test data.

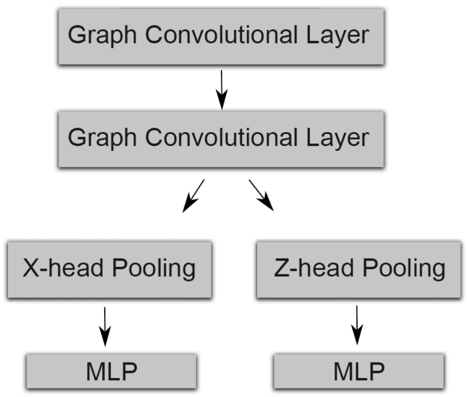


Figure 4.3: Schematic architecture of Model 1 where after two convolutional layers the data is fed to two separate heads. Each head contains a pooling layer with a multilayer-percepton afterwards.

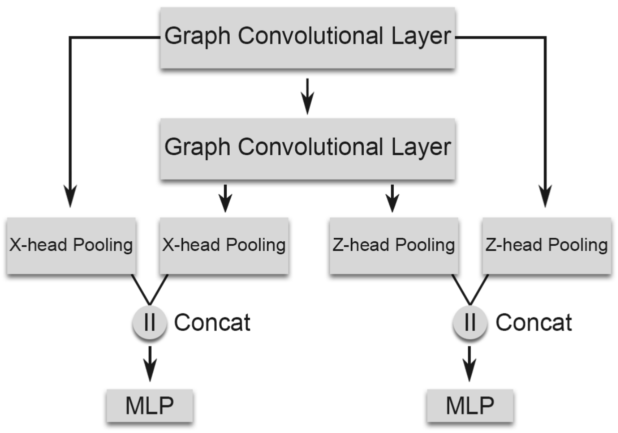


Figure 4.4: Network architecture for Model 2. Here one can see that after the first convolutional layer the information is sent parallel to a pooling layer and the next convolutional layer. The pooled graphs from both the first and second convolutional layer is then concatenated in separate heads and is later processed in multilayer-percepton.

5
Results

The results of classification accuracies for error probability $p = 0.05$ is presented in Tab. 5.1. For $p = 0.05$ Model 1 outperforms Model 2 and GAT for all datasets except for the combination dataset of $d = 5, 7, 9$ ($d5/d7/d9$). The highest accuracy reached was for the dataset $d7$. These results are obtained with k-values 8, 12, 14 for $d5, d7, d9$ respectively. Datasets containing multiple sizes use the same k-value as the largest individual size included. Hidden features remain the same for all datasets and are 16 for Model 1, 12 for Model 2 and 8 for GAT. The GAT network also have 6 attention heads.

While the classification accuracy for $p = 0.1$ is significantly lower than $p = 0.05$, Model 1 still exceed Model 2. The best precision was achieve for the dataset $d5$, where $k = 12$.

Table 5.1: Test accuracy for classification of syndromes generated with error probability $p = 0.05$ for five different datasets. The datasets are named such that the number following d is the size of the surface code which the syndromes were generated from. The two last datasets are combination of different code sizes.

	Datasets				
Model	*d5*	*d7*	*d9*	*d5/d7*	*d5/d7/d9*
Model 1	93%	97%	96.8%	96%	95.4%
Model 2	92.4%	94.5%	96%	95.5%	96.4%
GAT	86%	82%	83%	84%	-

Table 5.2: Test accuracy for datasets containing syndromes generated with $p = 0.1$. Comparison of two models are shown, for Model 1 k was 12 for both datasets, while k was 14 for Model 2.

	Datasets	
Model	*d5p1*	*d7p1*
Model 1	81.4%	77%
Model 2	79.5%	75%

Utilizing Model 1 trained with the $d5/d7$ dataset as decoder for quantum error correction with error probabilities between 0.02 and 0.05 and $d = 5$ is done and the results are shown in Fig. 5.1. The performance of the GNN is compared with the MWPM-algorithm and the MCMC-method. The successful correction ratio is decreasing as the error probability increases. In the GNN case all zero defect and one defect syndromes are determined to be solved trivially, thus decoded with a 100% accuracy.

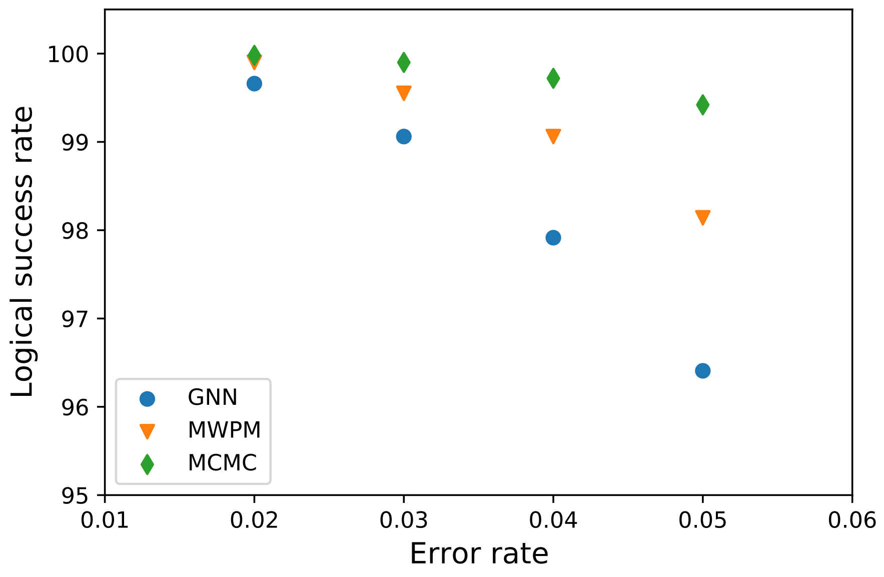


Figure 5.1: Here the error correction success rate is shown on the y-axis and the error probability on x-axis. The decoded syndromes are generated on $d = 5$ surface code. The GNN-model is "Model 1" and is trained with dataset $d5/d7$ and is clearly showing lower success rate than MWPM and MCMC.

6
Discussion and Analysis

From Tab. 5.1 it can be seen that the accuracy of the networks vary depending on the code size, while this is expected the deviation is not critical and therefore larger d could be investigated to see whether the behaviour maintains or if there is a threshold where the accuracy diverge. The precision for the datasets containing a mixture of sizes seems to decrease, however the decrease is not significant thus implying that a dataset consisting of various code sizes could result in a graph neural network decoder which could perform well independently of code size.

An analysis regarding which type of syndromes that are hardest to classify shows that syndromes belonging to class 3 are hardest. Considering that the majority of syndromes that the network fail to classify include larger number of nodes, and class 3 syndromes often contain many nodes indicates a correlation.
Viewing the classification accuracies for syndromes generated from $p = 0.1$, the accuracies are significantly lower. While it is expected to be harder to classify those syndromes the decrease is too substantial to consider it as a viable decoder at this time. However, we need to keep in mind that in the interval between $p = 0.05$ and $p = 0.1$ the performance of MWPM deviate notably from MCMC. Increasing the test accuracy to approximately 90% when training a network for $p = 0.1$ could potentially give a decoder that matches the performance of MWPM.

One important fact when comparing the accuracy when training a network and using it as decoder is that the occurrence of syndromes belonging to equivalence class zero is higher than the other classes (for lower p) and especially the rarity of class 3, since it is very high. For example the 10000 syndromes generated for $p = 0.2, 0.3, 0.4, 0.5$ contained 80, 165, 221 and 300 class 3 syndromes. This indicates that training a network to better classify syndromes giving arise to class 0 could be rewarding.

Practical modification could be done to both the way we structure the graphs and the network architecture. Having graphs containing a large number of nodes and also being fully connected could lead to some of the poor results, seen in the case of $p = 0.1$. One idea could be to introduce a cut-off, where edges outside a certain threshold gets neglected. However this could be problematic for graphs that have node clusters far away from each other in the graphs, since then the cut-off would neglect the edges connecting the two cluster, resulting in two graphs.

Theoretically a pooling layer like SAG could perform better considering that its projections scores are conditional on the connectivity of the graphs. While this is

not certain a deeper analysis of training with SAG-pooling would be appropriate.
In addition the k-value used when pooling is often larger than the mean node number of the dataset, meaning that we introduce more empty nodes than we actually down-sample the information. This indicates that comparable accuracy could perhaps be achieved without pooling.
We believe that the main obstacle to obtain better results while training the network is that with the current model the processing is only done on node features. A natural extension would be to develop a new custom-layer that operates on the edge weights as well.

7 Conclusion

In this thesis graph neural networks have been used to classify syndromes according to equivalence class on the surface code. From a *Markov Chain Monte Carlo* based decoder, probability distributions of error-chains corresponding to stochastic syndromes on the surface code for depolarizing noise are generated. The most likely equivalence class of error chains and the syndromes are used to create graph structured data. Datasets containing unique syndromes generated from different surface code sizes along with different error rates are used for training. The uncommon occurrence of syndromes corresponding to equivalence 3 makes it a challenge to create large class balanced datasets for low error probability. Several types of graph convolutional layers and pooling layers have been tested. The networks are trained in a supervised fashion using graphs representing syndrome as input and most likely equivalence class as target.
Examining the results it can be seen that the accuracies for $p = 0.05$ do not deviate significantly depending on the code size d, however the accuracy for $p = 0.1$ is considerably lower. Utilizing the best performing network as a decoder for low error probabilities shows that it underperforms compared to the reference algorithm *Minimum Weight Perfect Matching* (MWPM). Where MWPM achieves approximately 98.2 success ratio of error correction for $p = 0.05$, the GNN reaches 96.5.

Some ideas for future research on the subject are to compose a convolutional layer that incorporates a trainable parameter for edge attributes as well. Considering further analysis of other pooling-layers could also be of interest, as is other ways to represent the edge and node features of the graph representation of the syndrome.

Bibliography

[1] Wootter, W. K. and Zurek, W. H. "A single quantum cannot be cloned". *Nature*, vol. 299, pages 802-803, 1982.

[2] Steane. A. M. "A Tutorial on Quantum Error Correction", *Proceedings of the International School of Physics "Enrico Fermi"*, course CLXII, "Quantum Computers, Algorithms and Chaos", G. Casati, D. L. Shepelyansky and P. Zoller, eds., pages. 1-32 (IOS Press, Amsterdam 2006), 2006. url: `https://www2.physics.ox.ac.uk/sites/default/files/ErrorCorrectionSteane06.pdf`

[3] Shor, P. W. "Scheme for reducing decoherence in quantum computer memory". *Physical Review A*, vol. 52, issue 4, pages R2493–R2496. 1995. url:`https://link.aps.org/doi/10.1103/PhysRevA.52.R2493`

[4] Kitaev. A. "Fault-tolerant quantum computation by anyons", *Annals of Physics*, vol. 303, number 1, pages 2-30, Jan. 2003. url:`https://www.sciencedirect.com/science/article/pii/S0003491602000180`

[5] He. K, Zhang. X, Ren. S and Sun. J. "Deep Residual Learning for Image Recognition", *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2016.

[6] Stokes. J. M, Yang. K, Swanson. K, Jin. W, Cubillos-Ruiz. A, Donghia. N. M, MacNair. C. R, French. S, Carfrae. L. A, Bloom-Ackermann. Z, Tran. V. M, Chiappino-Pepe. A, Badran. A. H, Andrews. I. W, Chory. E. M, Church. G. M, Brown. E. D, Jaakkola. T. S, Barzilay. R and Collins J. J. "A Deep Learning Approach to Antibiotic Discovery", *Cell*, vol. 180, number 4, pages 688-702, 2020. url:`https://www.sciencedirect.com/science/article/pii/S0092867420301021`

[7] Murphy, K. P. *Machine Learning: A Probabilistic Perspective*, chapter 1, pages 1-24. MIT Press, 2012.

[8] Silver. D, Huang. A, Maddison. C. et al. "Mastering the game of Go with deep neural networks and tree search". *Nature*, **529**, pages 484-489. 2006. url: `https://doi.org/10.1038/nature16961`

[9] Andreasson. P, Johansson. J, Liljestrand. S and Granath. M. "Quantum error correction for the toric code using deep reinforcement learning", *Quantum*, vol. 3, page 183, 2019. url: `https://quantum-journal.org/papers/q-2019-09-02-183/pdf`

[10] Sweke. R, Kesselring. M. S, van Nieuwenburg. E. P. L and Eisert. J. "Reinforcement Learning Decoders for Fault-Tolerant Quantum Computation", *Machine Learning: Science and Technology* vol. 2, number 2, pages 025005, 2021. url:`https://iopscience.iop.org/article/10.1088/2632-2153/abc609/pdf`

[11] Wang. H, Xu. T, Liu. Q, Lian. D, Chen. E, Du. D. Wu. H and Su. W. "MCNE: An End-to-End Framework for Learning Multiple Conditional Network Representations of Social Network", *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1064–1072, 2019. url: `https://doi.org/10.1145/3292500.3330931`

[12] Zhang. M and Chen. Y. "Link Prediction Based on Graph Neural Networks", *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 5171–5181, 2018. url:`https://dl.acm.org/doi/pdf/10.5555/3327345.3327423`

[13] Hamilton, L. W. "Graph Representation Learning". *Synthesis Lectures on Artificial Intelligence and Machine Learning*, Vol. 14, No. 3, Pages 1-159. 2020

[14] Kipf, T. and Welling, M. "Semi-Supervised Classification with Graph Convolutional Networks", 2016. arXiv preprint arXiv:1609.02907 [cs.LG].

[15] Veličković. P, Cucurull. G, Casanova. A, Romero. A, and Liò. P and Bengio. Y. "Graph Attention Networks" *International Conference on Learning Representations*, 2018. url: `https://openreview.net/forum?id=rJXMpikCZ`

[16] Lee. J, Lee. I and Kang. J, "Self-Attention Graph Pooling", *Proceedings of the 36th International Conference on Machine Learning* (ICML 2019), 2019.

[17] Roffe. J. "Quantum Error Correction: An Introductory Guide", *Contemporary Physics*, vol. 60, number 3, pages 226-245, 2019. url: `https://doi.org/10.1080/00107514.2019.1667078`

[18] Ferrini. G, Kockum F. A, García-Alvarez. L and Vikstål. P. "Advanced Quantum Algorithms", lecture notes for course "Quantum Computation", chapter 3, pages 16-22, Chalmers/Gothenburg University, 2020.

[19] Kolmogorov. V. "Blossom V: a new implementation of a minimum cost perfect matching algorithm", *Mathematical Programming Computation 1.1*, pages

43-67, 2009.

[20] Hutter. A, Wootton. J. R and Loss. D. "Efficient Markov chain Monte Carlo algorithm for the surface code", *Phys. Rev.* A, vol. 89, 022326, Feb 2014. url:`https://link.aps.org/doi/10.1103/PhysRevA.89.022326`

[21] Hammar. C, Lindgren. W, Orekhov. A, Wallin Hybelius. P and Karariina Wisakanto. A. "Monte Carlo based reward scheme for deep Q-learning decoder for the toric code", Bachelor thesis, 2020.

[22] Cangea. C, Veličković. P, Jovanovicć. N, Kipf. T and Liò. P. "Towards Sparse Hierarchical Graph Classifiers", arXiv:1811.01287 [stat.ML], 2018.

[23] Liu. Y, Shi. X, Pirece. L and Ren. X. "Characterizing and Forecasting User Engagement with In-app Action Graph: A Case Study of Snapchat", *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2023–2031, 2019. url:`https://doi.org/10.1145/3292500.3330750`