Quantum Computing

Data-driven decoding of Surface codes using GNNs

Trimigno Giuseppe
giuseppe.trimigno@studenti.unipr.it

October 2023

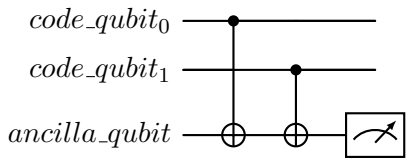
1 Quantum Error Correction

As we now know, Quantum Computers need to encode information in qubits. Most of quantum algorithms developed until now assume that qubits are perfect (i.e. they are *logical qubits*, that can be prepared in any state and be manipulated with complete precision).
In the last decade there were big progress in searching physical systems that act as qubits but, despite the improvement of designed qubits' quality, the design of custom algorithms and the usage of error mitigation effects, we can not entirely remove imperfections, so we refer to them as *phisical qubits*.
So, for the future era of fault-tolerance, we must find ways to build logical qubits from physical qubits. This will be done through the process of *Quantum Error Correction*, in which logical qubits are encoded in a large number of physical qubits, and the encoding is maintained by putting the physical qubits through a *highly entangling circuit*, in order to serve the purpose of error detection and correction.
Typically we consider two type of errors that can occur: (1) A *gate error*, i.e. an imperfection in any operation we perform (e.g. modeled as *depolarizing noise*), which happens with probability p_{gate}; (2) A *measurement error*, which flips between a 0 to a 1 or viceversa, with probability p_{meas}.

1.1 Storing Qubits

It is really important to consider that, in reality, there can be a significant delay between encoding and decoding (this means that there can be a significant amount of time between initializing the circuit, and making the final measurements).
As an obvious example, one may wish to encode a quantum state and store it for a long time, like a quantum hard drive. A less obvious but much more important example is performing fault-tolerant quantum computation itself.
For this, we need to store quantum states and preserve their integrity during the computation. This must also be done in a way that allows us to manipulate the stored information in any way we need, and which corrects any errors we may introduce when performing the manipulations.
In all cases, we need to know that errors do not only occur when something happens (like a gate or measurement), they also occur when the qubits are idle, because they interact with each other and with the environment. It is obvious that, the longer we leave our qubits idle for, the greater the effects of this noise becomes, i.e. p_{meas} increases.
The solution is to keep measuring, so that no qubit is left idle for too long, but this is easy for classical information (we just need to constantly measuring the value of each qubit, and by

keeping track of when the values change due to noise, we can easily deduce a history of when errors occurred), not for quantum information.
Suppose we have an encoding such that $|0\rangle \rightarrow |000\rangle$ and $|1\rangle \rightarrow |111\rangle$. We want to encode the logical $|+\rangle$, so it will be $|+\rangle \rightarrow \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$. Performing a Z measurement (which distinguish between $|0\rangle$ and $|1\rangle$) of the logical qubit consists of doing it for each of physical qubits, such that the final result for the logical measurement is simply decoded by looking which output is the majority.
So we can keep track of errors on logical qubits that are stored for a long time by constantly performing Z measurements of the physical qubits, which is fine if we are simply storing 0 and 1, not if we are storing a *superposition*, because we are not preserving it (it isn't an error caused by imperfections in devices). The solution to this problem is given by the following quantum circuit:

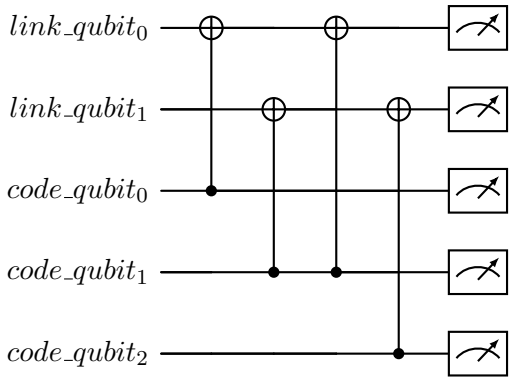


where the *ancilla_qubit* is always $|0\rangle$, while *code_qubits* can be initialized in different states (e.g. $|11\rangle$ if we add an X gate at the beginning of each *code_qubit* line). By making experimental measurements, it is possible to see that, for $|00\rangle$, $|11\rangle$ and any superposition, the result will be always 0, while for $|01\rangle$, $|10\rangle$ and any superposition, it will be 1.
This measurement is therefore telling us a property of multiple qubits: it checks if the two code qubits are in the same state (returns 0) or not (returns 1). Now, if we apply such a "*syndrome measurement*" on all pairs of physical qubits in our repetition code, we will get a binary list. Given it, we will know that our states are indeed encoded in the repeated states that we want them to be, and can deduce that no errors have occurred (if we get $[0, ..., 0]$). If some syndrome measurements return 1, however, there is an error.

1.2 Repetition code

The basic ideas behind *Quantum Repetition Code* are the same as for classical repetition code, so we can start considering a very simple example: if we are talking on the phone, and someone asks us a binary question, our answer will mainly depend on two factors: (1) How important is it that we understood correctly? (2) How good is your connection?
We can parameterize them as probabilities, e.g. let's call P_a the maximum acceptable probability of being misunderstood, and p the probability that our answer is garbled by the communication channel (e.g. a '*yes*' answer sounds like a '*no*' one or viceversa).
Tipically, a good connection will result in $p < P_a$ (in this case, we can just answer with a single '*yes*' or '*no*'), while a poor connection will result in $p > P_a$. In the last case instead, a single '*yes*' or '*no*' is not enough, because the probability of being misunderstood is too high, so we need to encode the answer in a more complex structure, allowing the receiver to decode our meaning even if the message is corrupted.
The simplest way to do it is repeating the answer many times, e.g. by sending '*yes yes yes*' instead of '*yes*', so that the receiver can conclude that (most likely) the answer is '*yes*' even if he receives '*yes no yes*' because the second bit has been corrupted. Of course, if Alice sends '*yes yes yes*' and Bob receives '*no no yes*', he will decode the wrong answer, but this is less likely to happen compared to an error on a single bit. Indeed, if $p = 0.01$ is the probability of a single error, the probability of at least 2 errors is $P = 3p^2(1-p) + p^3 = 0.0003$.
So, if $P < P_a$, the problem is solved, else we can simply add more repetitions. It does not matter

how bad the connection is, the problem can be solved by adding repetitions, since P will decrease exponentially.
Obviously, '*yes*' and '*no*' can be substituted by $|0\rangle$ and $|1\rangle$, obtaining the quantum repetition code, which is the simplest, but one of most inefficient way to implement the quantum error correction. We are free to choose how many physical qubits we want the logical qubit to be encoded in, and how many times the syndrome measurements will be applied while we store our logical qubit, before the final readout measurement. Considering the smallest non-trivial case, with three repetitions and one syndrome measurement round, we have the following quantum circuit for the logical 0 (we can simply apply an X gate on each code qubit for the logical 1):

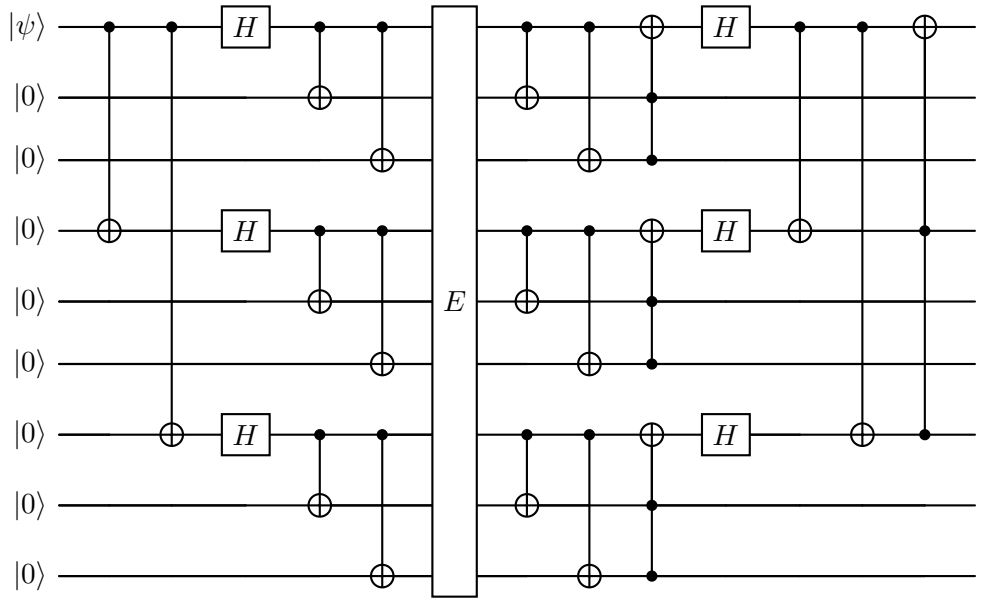


where *code qubits* encode the logical state, while *link qubits* serve as the ancilla qubits for the *syndrome measurements*. So in this case a single round of syndrome measurements consists of just two syndrome measurements (one which compares code qubits 0 and 1, the other which compares code qubits 1 and 2); we can generalize this concept for n qubits, which require $n-1$ syndrome measurements of neighbouring pairs of qubits.
Now, suppose we run this experiment with some noise, and we get '000 00' or '111 00', it's obvious that 0 and 1 are encoded, respectively; similarly, if we see '001 00', we can decode it as a logical 0 (that 1 in the output is probably an error). This tactic can employed to decode all other outcomes, forming the so called *lookup table decoding*, where every possible outcome is analyzed, and the most likely value to decode it as is determined.
For many qubits, this quickly becomes intractable (the table grows exponentially large as code size increases), as the number of possible outcomes becomes so large. To overcome this problem, decoding is typically done in a more algorithmic manner that takes into account the structure of the code and its resulting syndromes. Specifically, MWPM (*Minimal Weight Perfect Matching*) algorithm is used, which is discussed in chapter 3.

1.2.1 Shor's code

Even if it's possible to create different quantum circuits to correct *phase errors* or *bit flip errors*, there is a quantum circuit which can correct both *phase flips* as well as *bit flip* errors, the *Shor code*. It is a 9 qubit circuit that requires 8 ancillary qubits to correct 1 qubit (the main qubit).
The Shor code works by first taking the computational state of the main qubit and transferring it to the 3^{rd} and 6^{th} qubit. These qubits are used for correcting *phase errors*. After this these qubits are put in to superposition using a *Hadamard gate*. Next the states of the main qubit as well as the 3^{rd}, and 6^{th} qubits use *CNOT gates* to transfer their states to ancillary qubits responsible for correcting *bit flips*. More specifically, the main qubit transfers its state to the 1^{st} and 2^{nd} ancillary qubit. The 3^{rd} transfers it state to the 4^{th} and 5^{th}. The 6^{th} transfer its state to the 7^{th} and 8^{th}

qubit.
Below, we can see the quantum circuit of Shor code, where E gate indicates the introduction of an error:



1.3 Surface code

Generally speaking, a *quantum code* is a linear subspace $\mathcal{L} \subset (\mathcal{C}^2)^{\bigotimes n}$. The simplest one in the *Surface code* family is the *Toric code*, defined on $n = 2L^2$ physical qubits, arranged on a square grid of dimension $L \times L$.
Relying on figure 1, physical qubits are on black marks (edges), and they are defined with two main *Stabilizer operators* (defined as *Pauli operators* product), i.e. $A_s = \prod_{j \in \text{star(s)}} \sigma_j^x$ (associated to vertexes, green in the figure) and $B_p = \prod_{j \in \text{boundary(p)}} \sigma_j^z$ (associated to a square of the lattice, red in the figure).
By definition, we have that $|\xi\rangle \in \mathcal{L} \iff A_s |\xi\rangle = |\xi\rangle, B_p |\xi\rangle = |\xi\rangle \, \forall s, p$, i.e. $|\xi\rangle$ is a *code vector* if and only if it is a *joint eigenvector* related to eigenvalue 1 of all Stabilizer operators. In order to make this working, A_s and B_p must commute, which is obvious in the left image, since they act on disjoint qubit subsets, but it is also valid for the right image, because there are two shared qubits, so two negative signs that elide themselves.

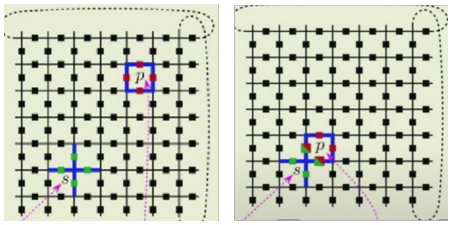


Figure 1: General lattice-structure of Surface code, with A_s and B_p Stabilizer operators.

Focusing on errors, each error can be decomposed as *Pauli operators* in the form $E = \sigma^x(g^{(x)})\sigma^z(g^{(z)})$, where $g^{(x)}$ and $g^{(z)}$ are qubit sets.

Now, if we consider a Pauli operator E on a code vector $|\xi\rangle$ of a code defined by Stabilizer operators $S_1, ..., S_l$, we have that E and S_j can *commute* or *anti-commute*.
To track this event, a *syndrome bit* μ_j is used, whose value will be 0 if E and S_j commute, 1 otherwise, so that $S_j E = (-1)^{\mu_j} E S_j$. By constructing the binary vector $\mu = [\mu_1, ..., \mu_l]$, we obtain the *syndrome* of the error E.
This allows us to classify errors in three classes:

1. **Trivial** errors: if $E = \pm \prod_j S_j^{\gamma_j}$ (i.e. $\mu = \vec{0}$) with γ_j binaries, the error acts trivially on the code, preserving the code space and each code vector individually, so it is an harmless error;
2. **Detectable** errors: if $\mu \neq \vec{0}$, the error is detectable, since it does not completely preserve the code, but it transfer it into an orthogonal subspace;
3. **Non-Trivial** errors: if $\mu = \vec{0}$ but $E \neq \pm \prod_j S_j^{\gamma_j}$, the error preserves the code, but not code vectors individually, so it is a non-trivial logical operator. So, these operators are as the ones we know in quantum computing and that we apply for the computation, but when they occur spontaneously, as E, they can corrupt the computation.

Focusing on *Toric code*, and considering Z errors, we have that errors can be represented as a line, or a set of lines on the lattice. As we can see from figure 2:

- In case of *trivial errors*, we will have a boundary on a chain or on a bi-dimensional region;
- In case of *detectable errors*, we will have a boundary of non-null dimension covering more qubits, which can be detected by measuring stabilizers and checking for syndrome bits equals to 1;
- In case of *non-trivial errors*, we will have a line that cover the full lattice from left to right (and since the left-most and right-most points on a lattice's line are the same, we have a closed line, i.e. a mono-dimensional chain).

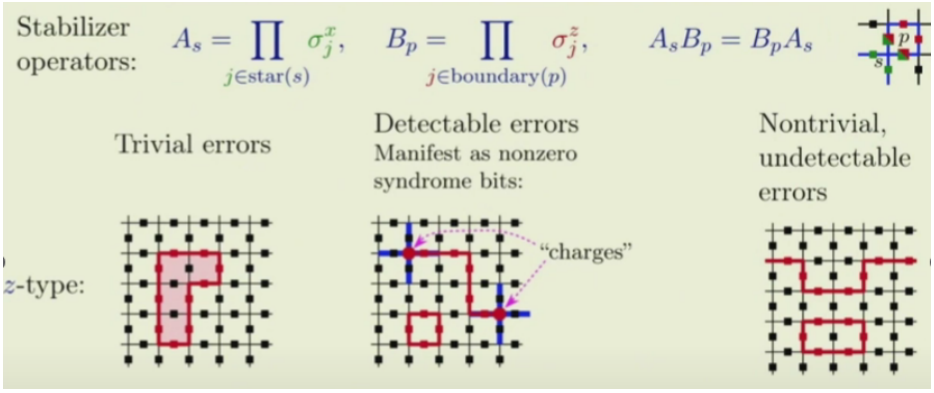


Figure 2: Summary of Z-type errors on Toric code

2 Minimum Weight Perfect Matching

In *graph theory*, a *matching* (or *indipendent edge set*) M in an undirected graph $G = (V, E)$ is a set of pairwise non-adjacent edges, none of which are loops; that is, no two edges share common

vertices. In other words, a subset of the edges is a matching if each vertex appears in at most one edge of that matching (if the graph is *bipartite*, then finding a matching is equivalent to the *network flow* problem).
A *perfect matching* is a matching that matches all vertices of the graph, i.e. every vertex of the graph is incident to an edge of the matching. So it will be $|E| = \frac{|V|}{2}$.
So, the *minimum weight perfect matching* problem consists in finding, between all perfect matching, the one with minimum weight, where the total weight is indeed the sum of all matching edges' weights.
Note that *MWPM* is an approximation, while *Maximum likelihood decoding* is the correct algorithm, which has a big drawback, i.e. we must know the full underlying error model. Actually, it is based on mapping individual errors to topological classes, computing the total error probability of all errors in each class, and selecting the class with the highest probability.
This algorithm satisfy the threshold theorem, for which, if in every qubit a Z and an X error occur with a probability less than $p_c \approx 0.109$, we can correct them.

3 GNN based Data Driven Decoder

To leverage the full potential of quantum error-correcting stabilizer codes it is crucial to have an efficient and accurate decoder. Accurate, maximum likelihood, decoders are computationally very expensive whereas decoders based on more efficient algorithms give sub-optimal performance. In addition, the accuracy will depend on the quality of models and estimates of error rates for idling qubits, gates, measurements, and resets, and will typically assume symmetric error channels.
The idea, given by Moritz Lange et al, is to explore a model-free, data-driven, approach to decoding, using a GNN (*Graph Neural Network*), which is well suited for this type of data. Namely, a single data point consists of a set of "detectors", i.e., changes in stabilizer measurements from one cycle to the next, together with a label indicating the measured logical bit- or phase-flip error. This can be represented as a labeled graph with nodes that are annotated by the information on the type of stabilizer and the space-time position of the detector, as we can see from figure 3. The maximum degree of the graph can be capped based on removing edges between distant detectors, keeping only a fixed maximum number of neighboring nodes.
So, the decoding problem is formulated as a graph classification task in which a set of stabilizer

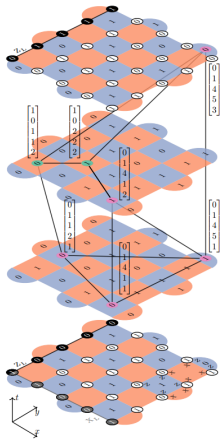


Figure 3: Example of labeled graph related to a single data-point

measurements is mapped to an annotated detector graph for which the neural network predicts the

most likely logical error class.
The paper aims to show that the GNN-based decoder can outperform a *matching decoder* for *circuit level noise* on the *surface code* given only simulated experimental data, even if the matching decoder is given full information of the underlying error model. Although training is computationally demanding, inference is fast and scales approximately linearly with the space-time volume of the code.
A schematical view of the GNN decoder is provided by Figure 4, which takes as input an annotated detector graph. Several layers of graph convolutional operations transform each node feature vector. Next, a mean-pooling operation averages all the node feature vectors into a single graph embedding, which is independent of the size of the graph. Finally, the latter is passed through two separate dense networks to give two binary class predictors, corresponding to the logical X and Z labels, respectively.

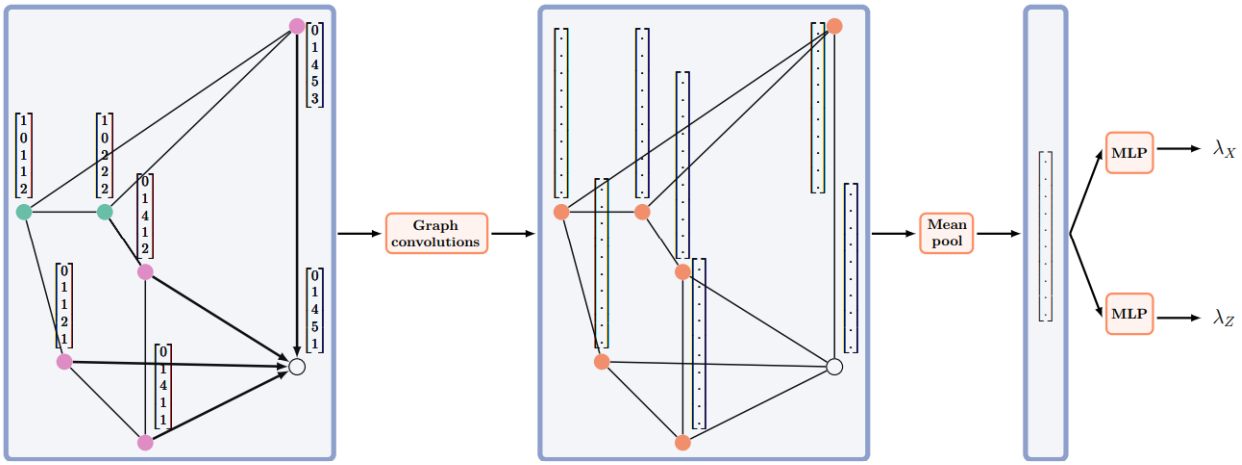


Figure 4: Schematic view of GNN decoder

3.1 Data structure

In order to train the decoder with both *circuit-level noise* and *perfect stabilizers*, the decision has been to generate *synthetic data* using python library, *Stim*. Simulated circuits use standard settings for the surface code, containing Hadamard single qubit gates, controlled-Z (CZ) entangling gates, measure and reset operations. All of these operations, and the idling, contain Pauli noise, scaled by an overall error rate p.
Specifically, for the number of data points, the following settings were used:

- **Circuit-level noise**: error rate p varying in the range $[0.001, 0.002, 0.003, 0.004, 0.005]$, generating a training set which contains 10^6 samples for each value of p, for a total of $5 \cdot 10^6$ training samples. Code distances 3 and 5 were considered, each one with 3, 5, 7, 9 and 11 round of stabilizer measurements.

- **Perfect stabilizers**: error rate p varying in the range $[0.01, 0.05, 0.10, 0.20]$, generating a training set which contains 10^6 samples for each value of p, for a total of $4 \cdot 10^6$ training samples. Code distances 5, 7 and 9 were considered, each one with 1 round of stabilizer measurements.

To avoid overfitting, after every epoch, the left-most (i.e. the oldest) 25% of the training data is replaced with new data, generated within the training cycle. As it is shown in Figure 5, there is a really big advantage with this method, in terms of both accuracy and loss.

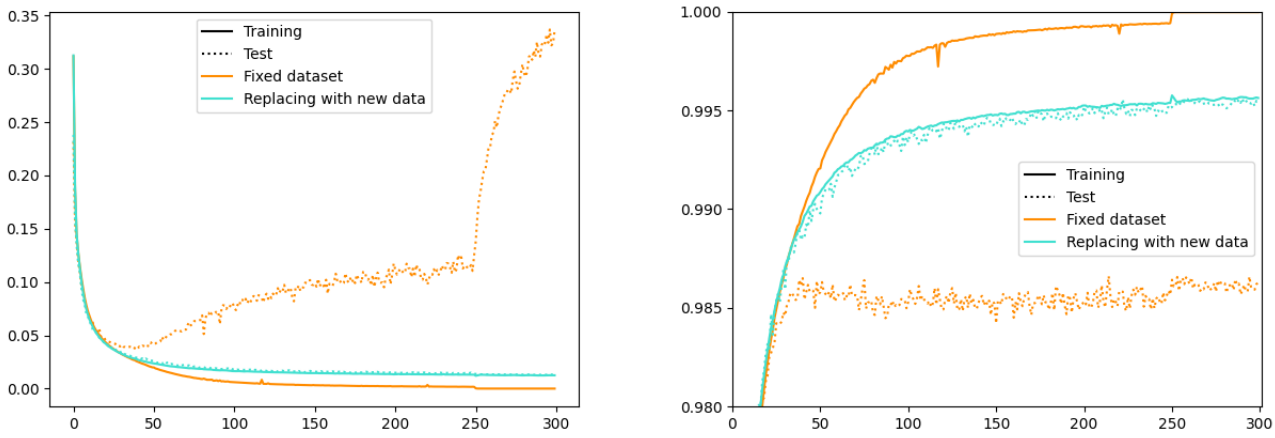


Figure 5: Loss (left) and accuracy (right) training history for surface code of size 5 and with 5 measurement rounds, in cases of both data replacement of 25% or not after every epoch.

3.2 Graph Neural Networks

A *Graph Neural Network* (**GNN**) is a class of artificial neural networks for processing data that can be represented as graphs. It can be viewed as a trainable message passing algorithm, where information is passed between the nodes through the edges of the graph and processed through a neural network.

The data is in the form of a graph $G = (V, E)$, with a set of nodes $V = \{i | i = 1, ..., N\}$, and edges $E = \{(i, j) | i \neq j \in V\}$ which is annotated by n-dimensional node feature vectors $\vec{X}_i$ and edge weights (or vectors) e_{ij}.

The basic building blocks are the message passing graph convolutional layers, which take a graph as input and outputs an isomorphic graph with transformed feature vectors. Specifically, a standard graph convolution have been used, where for each node the d_{in}-dimensional feature vector $\vec{X}_i$ is transformed to new feature vector $\vec{X}'_i$ with dimension d_{out} according to the following equation:

$$\vec{X}'_i = \sigma\left(W_1\vec{X}_i + \sum_j e_{ij} W_2 \vec{X}_i\right)$$

where we have $e_{ij} = 0$ for non-existent edges, W_1 and W_2 are trainable weight matrices ($d_{out} \times d_{in}$ dimensional), and σ is an element-wise non-linear activation function which includes a d_{out}-dimensional trainable bias vector.

Graph convolution layers are followed by a *pooling layer* that contracts the information to a single vector, a graph embedding, which is independent of the dimension of the graph. In this case, a simple *mean-pooling* layer $\vec{X}' = N^{-1} \sum_i \vec{X}_i$.

3.3 Network's architecture and Hyperparameters

Figure 6 displays the achitecture of the GNN decoder, which is used for all experiments, with details of layers and input/output (d_{in}/d_{out}) dimensions, shown in Table 1.

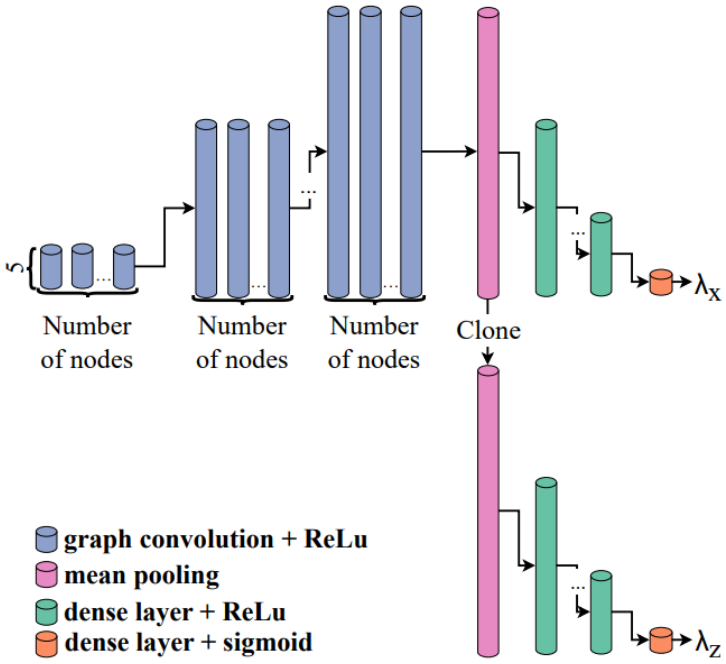


Figure 6: Schematic of the GNN architecture, with details in Table 1

Layer	d_{in}	d_{out}
GraphConv_1	5	32
GraphConv_2	32	128
GraphConv_3	128	256
GraphConv_4	256	512
GraphConv_5	512	512
GraphConv_6	512	256
GraphConv_7	256	256
$\text{Dense}_1 X$	256	128
$\text{Dense}_2 X$	128	181
$\text{Dense}_3 X$	128	32
$\text{Dense}_4 X$	32	1
$\text{Dense}_1 Z$	256	128
$\text{Dense}_2 Z$	128	181
$\text{Dense}_3 Z$	128	32
$\text{Dense}_4 Z$	32	1

Table 1: Overview over the input and output dimension of the graph convolutional and dense layers of the GNN decoder.

For each layer, the activation function is a *ReLU* (Rectified Linear Unit), which corresponds to chopping negative values. After the graph convolutional layers, the node features from all nodes

are pooled into one high-dimensional vector by computing the mean across all nodes (mean pool layer). This vector is then cloned and sent to two identical fully connected neural networks. Both heads consist of 4 dense layers which map the pooled node feature vector down to one real-valued number which is output in the range 0 to 1 through a sigmoid function (in fact, each feed-forward neural network end with a singe node that acts as a binary classifier).
As loss function, *BCEWithLogitsLoss* (*Binary Cross-Entropy*) from PyTorch was used, with *Adam* as optimizer, and a learning rate of 10^{-4}, manually decreased to 10^{-5} after epoch 250 is reached.

3.4 Results

Considering surface code with circuit-level noise and size 3, which has been trained with 3, 5, 7, 9 and 11 measurement rounds for 100 epochs each (for a total of 600 epochs, which required a total of 6 days for training on a Nvidia V100 40gb GPU and 56gb of CPU RAM), we can see, from Figure 7 a good training curve, with a loss between 0.05 (with 3 measurement rounds) and 0.07 (with 11 measurement rounds), really close to 0, and an accuracy score between 0.985 (with 3 measurement rounds) and 0.975 (with 11 measurement rounds), evidence of a really nice result.

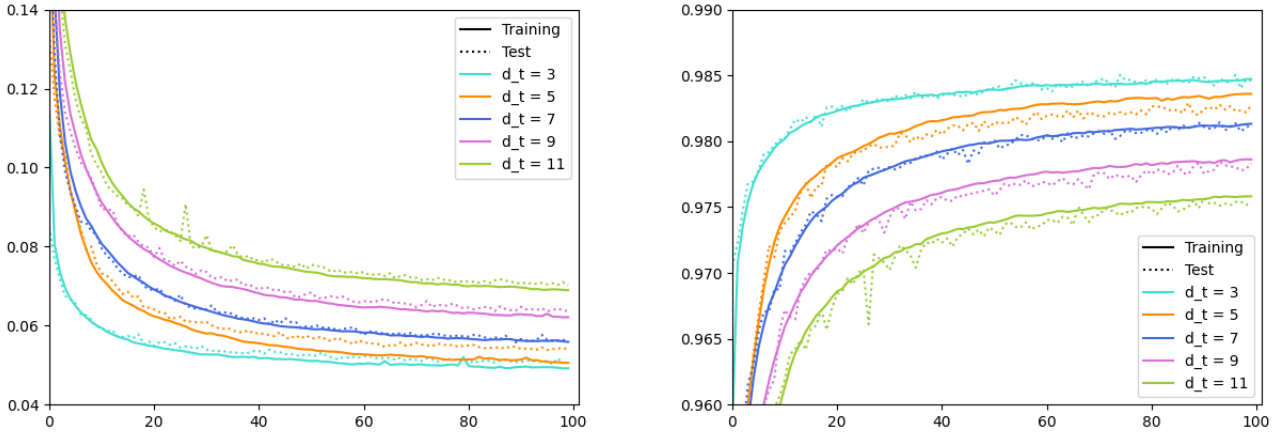


Figure 7: Loss (left) and accuracy (right) history extracted from training surface code with circuit-level noise at distance 3 with 3, 5, 7, 9 and 11 measurement rounds.

Considering surface code with circuit-level noise and size 5, which has been trained with 3, 5, 7, 9 and 11 measurement rounds for 600 epochs each (for a total of 3.000 epochs, which required a total of 36 days for training on a Nvidia A100 80gb GPU and 168gb of CPU RAM), we can see, from Figure 8 a good training curve, with a loss between 0.05 (with 3 measurement rounds) and 0.07 (with 11 measurement rounds), really close to 0, and an accuracy score between 0.985 (with 3 measurement rounds) and 0.975 (with 11 measurement rounds), evidence of a really nice result.

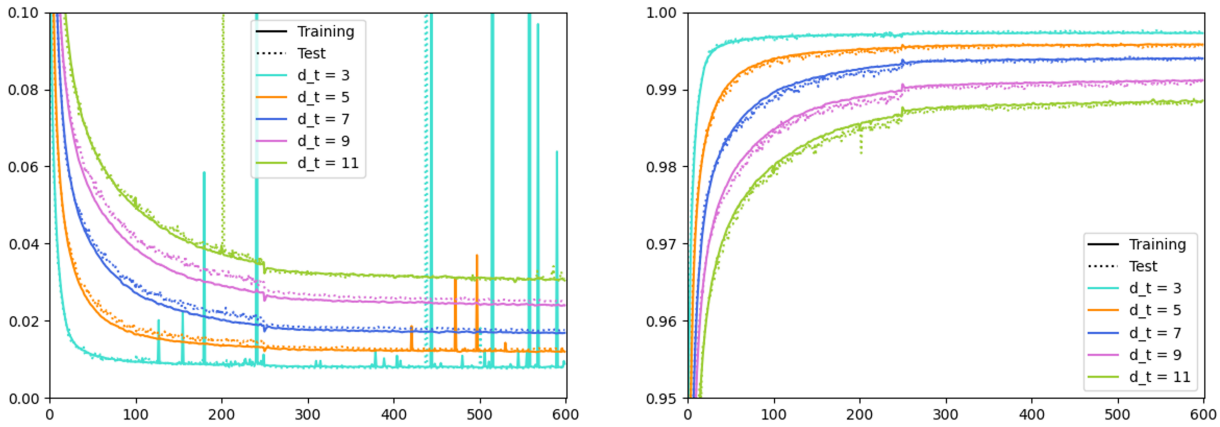


Figure 8: Loss (left) and accuracy (right) history extracted from training surface code with circuit-level noise at distance 5 with 3, 5, 7, 9 and 11 measurement rounds.

Considering surface code with perfect stabilizers (i.e. a single measurement round) and size 5, which has been trained for 600 epochs (which required a total of 5 days for training on a Nvidia A100 80gb GPU and 128gb of CPU RAM), we can see, from Figure 9 a good training curve, with a loss between 0.05 (with 3 measurement rounds) and 0.07 (with 11 measurement rounds), really close to 0, and an accuracy score between 0.985 (with 3 measurement rounds) and 0.975 (with 11 measurement rounds), evidence of a really nice result.

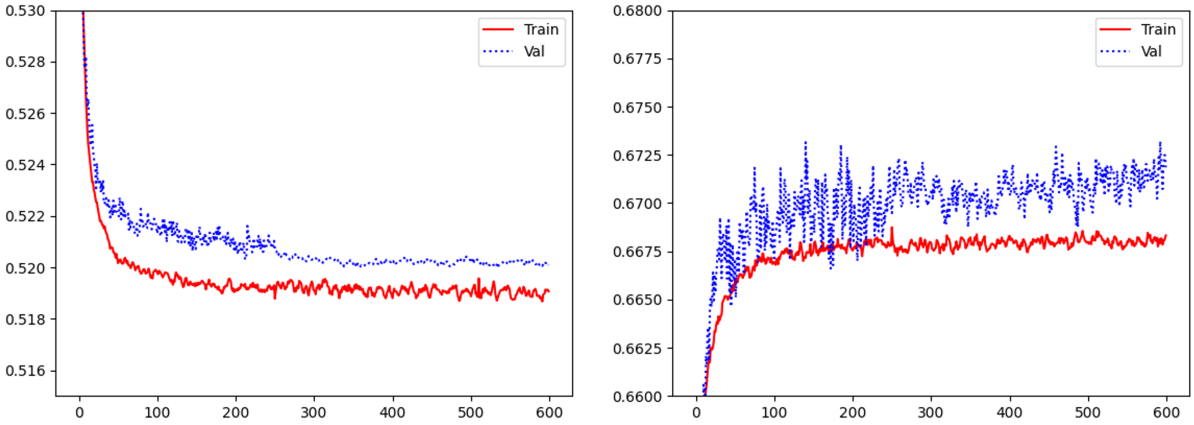


Figure 9: Loss (left) and accuracy (right) history extracted from training surface code with perfect stabilizers, i.e. with a single measurement round, at distance 5.

Considering surface code with perfect stabilizers (i.e. a single measurement round) and size 7, which has been trained for 600 epochs (which required a total of 6 days and half for training on a Nvidia A100 80gb GPU and 168gb of CPU RAM), despite the paper shows 700 epochs (as it is easy to see, performance already reached a plateau after 600 epochs), we can see, from Figure 10 a good training curve, with a loss between 0.05 (with 3 measurement rounds) and 0.07 (with 11

measurement rounds), really close to 0, and an accuracy score between 0.985 (with 3 measurement rounds) and 0.975 (with 11 measurement rounds), evidence of a really nice result.

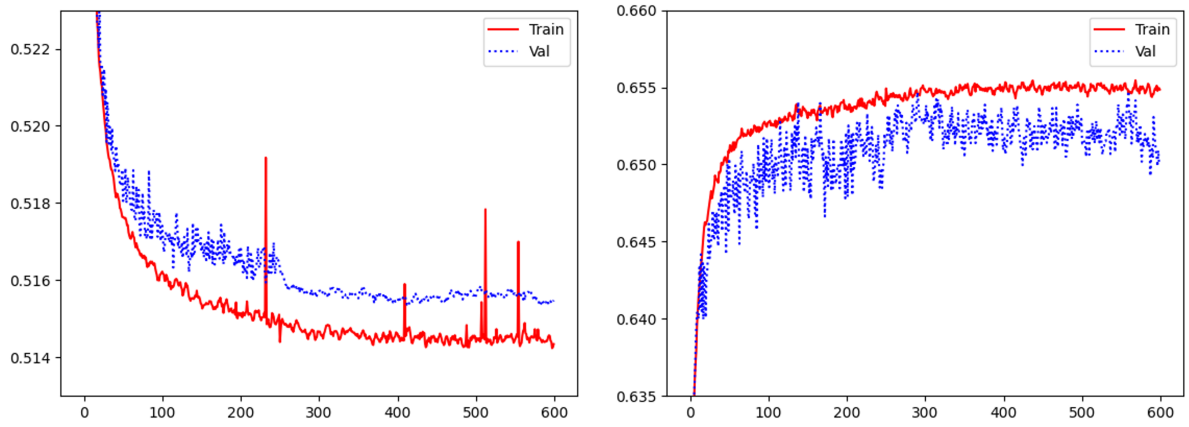


Figure 10: Loss (left) and accuracy (right) history extracted from training surface code with perfect stabilizers, i.e. with a single measurement round, at distance 7.

Considering surface code with perfect stabilizers (i.e. a single measurement round) and size 9, which has been trained for 600 epochs (which required a total of 6 days and half for training on a Nvidia A100 80gb GPU and 192gb of CPU RAM), despite the paper shows 800 epochs (as it is easy to see, performance already reached a plateau after 600 epochs), we can see, from Figure 11 a good training curve, with a loss between 0.05 (with 3 measurement rounds) and 0.07 (with 11 measurement rounds), really close to 0, and an accuracy score between 0.985 (with 3 measurement rounds) and 0.975 (with 11 measurement rounds), evidence of a really nice result.

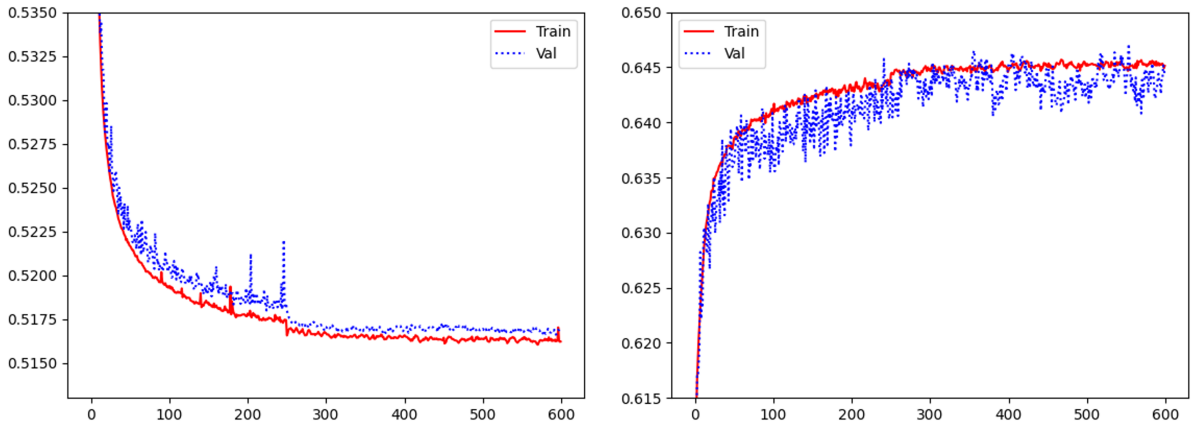


Figure 11: Loss (left) and accuracy (right) history extracted from training surface code with perfect stabilizers, i.e. with a single measurement round, at distance 9.

PS: Note that training days are related to a continuous training, but it was not our case, since we have 1-day limit of consecutive GPU usage.

4 MWPM vs GNNs

At this point, we want to compare how our *GNN decoder* works compared to *MWPM*, considering both *circuit-level noise* and *perfect stabilizers*, based on *accuracy*, *logical failure rate* and *time* required for decoding.
First of all, we consider *circuit level noise* at a reference error rate of 0.003, using $5 * 10^6$ testing samples for both distance 3 and 5 codes (considering, for each one, 3, 5, 7, 9 and 11 repetitions).
As we can see from table 3 and from figures 12, we can state that:

- in terms of time needed for decoding, except for some rare cases, *GNN decoder* is much faster than *MWPM*, which is really important in real-time applications;
- in terms of accuracy and logical failure rate, both models offer sub-optimal performances, close to the perfection, but looking into high precision, *GNN decoder* performs a little better.

Distance	Repetitions	Model	Accuracy	Logical failure rate	Mean time for sample (s)	Total time (s)
3	3	MWPM	0.9934994	0.0065006	$1.756499 * 10^{-6}$	8.782495
3	3	GNN	0.9941224	0.0058776	$2.6934926 * 10^{-6}$	13.467463
3	5	MWPM	0.9894224	0.0105776	$2.60999 * 10^{-6}$	13.04996
3	5	GNN	0.9911342	0.0088658	$1.29598 * 10^{-6}$	6.4799
3	7	MWPM	0.985516	0.014484	$3.26818 * 10^{-6}$	16.34091
3	7	GNN	0.988286	0.011714	$1.7872 * 10^{-6}$	8.93602
3	9	MWPM	0.9817616	0.0182384	$3.773677 * 10^{-6}$	18.86838
3	9	GNN	0.9853172	0.0146828	$2.27093 * 10^{-6}$	11.35465
3	11	MWPM	0.9777186	0.0222814	$4.193756 * 10^{-6}$	20.96878
3	11	GNN	0.9821406	0.0178594	$2.721868 * 10^{-6}$	13.60934
5	3	MWPM	0.99812	0.00188	$5.8444 * 10^{-6}$	29.222
5	3	GNN	0.99857	0.00143	$7.287 * 10^{-6}$	36.4351
5	5	MWPM	0.996628	0.003372	$7.6529 * 10^{-6}$	38.2645
5	5	GNN	0.99756	0.00244	$4.45529 * 10^{-6}$	22.276484
5	7	MWPM	0.9952662	0.0047338	$8.88143 * 10^{-6}$	44.40714
5	7	GNN	0.9965374	0.0034626	$6.15745 * 10^{-6}$	30.78725
5	9	MWPM	0.9938	0.00612	$9.92267 * 10^{-6}$	49.61337
5	9	GNN	0.994851	0.005149	$7.757456 * 10^{-6}$	38.787283
5	11	MWPM	0.9923444	0.0076556	$10.99565 * 10^{-6}$	54.97825
5	11	GNN	0.993426	0.006574	$9.35035 * 10^{-6}$	46.75176

Table 2: Testing results (accuracy, logical failure rate, mean time for sample and total time) for both GNN and MWPM models, evaluated over 5 milions of samples at a reference error rate of 0.003, with circuit-level noise codes with distances 3, 5 and repetitions 3, 5, 7, 9, 11.

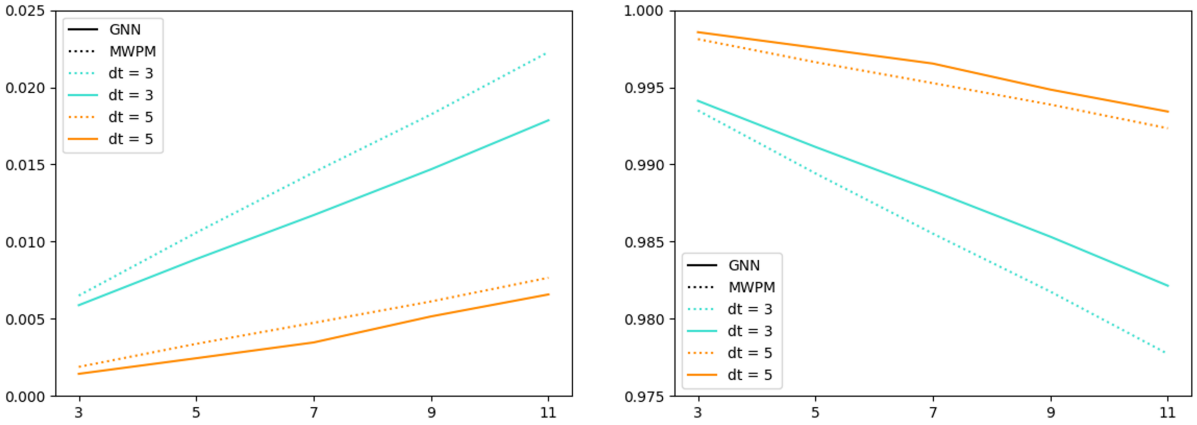


Figure 12: Logical failure rate (left) and accuracy (right) plotted over repetitions (3, 5, 7, 9, 11), extracted from testing surface code with distances 3 and 5.

Now, let's consider *perfect stabilizers* with error rates of 0.01, 0.05, 0.10, 0.15 and 0.20, using 10^6 testing samples (for each error rate, totalling $5*10^6$ samples for each code distance) for distance 5, 7 and 9 codes (considering, for each one, a single measurement round, since we are dealing with perfect stabilizers).
As we can see from table 3 and from figures 13, we can state that:

- in terms of time needed for decoding, *GNN decoder* is much faster than *MWPM* in every case (quite the half of time needed for the other in most of them), which is really important in real-time applications;
- in terms of accuracy and logical failure rate, performances are quite identical for both models, and they are lower compared to *circuit-level* noise, which is mainly caused by the higher error rates.

Distance	Erorr rate	Model	Accuracy	Logical failure rate	Mean time for sample (s)	Total time (s)
5	0.01	MWPM	0.9857	0.0143	$3.78 * 10^{-6}$	3.78
5	0.01	GNN	0.985734	0.01426	$2.568 * 10^{-6}$	2.568
5	0.05	MWPM	0.6567	0.3433	$7.47 * 10^{-6}$	7.47
5	0.05	GNN	0.6713	0.3287	$4.62 * 10^{-6}$	4.62
5	0.10	MWPM	0.5123	0.4877	$8.9232 * 10^{-6}$	8.9232
5	0.10	GNN	0.51672	0.48328	$5.74886 * 10^{-6}$	5.74886
5	0.15	MWPM	0.50054	0.49946	$9.2869 * 10^{-6}$	9.2869
5	0.15	GNN	0.5005	0.4995	$6.143 * 10^{-6}$	6.143
5	0.20	MWPM	0.50069	0.4993	$9.3879 * 10^{-6}$	9.3879
5	0.20	GNN	0.501204	0.498796	$6.23 * 10^{-6}$	6.23
7	0.01	MWPM	0.9919	0.0081	$5.34 * 10^{-6}$	5.34
7	0.01	GNN	0.9916	0.0084	$3.4 * 10^{-6}$	3.4
7	0.05	MWPM	0.6151	0.3849	$12.82 * 10^{-6}$	12.82
7	0.05	GNN	0.6237	0.3763	$9.1566 * 10^{-6}$	9.1566
7	0.10	MWPM	0.5034	0.4966	$16.08 * 10^{-6}$	16.08
7	0.10	GNN	0.5035	0.4965	$11.197 * 10^{-6}$	11.197
7	0.15	MWPM	0.50065	0.49935	$16.988 * 10^{-6}$	16.988
7	0.15	GNN	0.50012	0.49988	$11.7 * 10^{-6}$	11.7
7	0.20	MWPM	0.500434	0.499566	$17.333 * 10^{-6}$	17.333
7	0.20	GNN	0.50052	0.49948	$11.82648 * 10^{-6}$	11.82648
9	0.01	MWPM	0.99585	0.00415	$6.7228 * 10^{-6}$	6.7228
9	0.01	GNN	0.995023	0.004977	$5.42 * 10^{-6}$	5.42
9	0.05	MWPM	0.5825	0.4175	$21.236 * 10^{-6}$	21.236
9	0.05	GNN	0.5834	0.4166	$14.8634 * 10^{-6}$	14.8634
9	0.10	MWPM	0.5005	0.4995	$27.24 * 10^{-6}$	27.24
9	0.10	GNN	0.501	0.499	$17.99 * 10^{-6}$	17.99
9	0.15	MWPM	0.499864	0.500136	$28.895 * 10^{-6}$	28.895
9	0.15	GNN	0.499613	0.500387	$18.8 * 10^{-6}$	18.8
9	0.20	MWPM	0.49981	0.50019	$29.09 * 10^{-6}$	29.09
9	0.20	GNN	0.50118	0.49882	$19.0357 * 10^{-6}$	19.0357

Table 3: Testing results (accuracy, logical failure rate, mean time for sample and total time) for both GNN and MWPM models, evaluated over 5 milions of samples at a reference error rate of 0.003, with circuit-level noise codes with distances 3, 5 and repetitions 3, 5, 7, 9, 11.

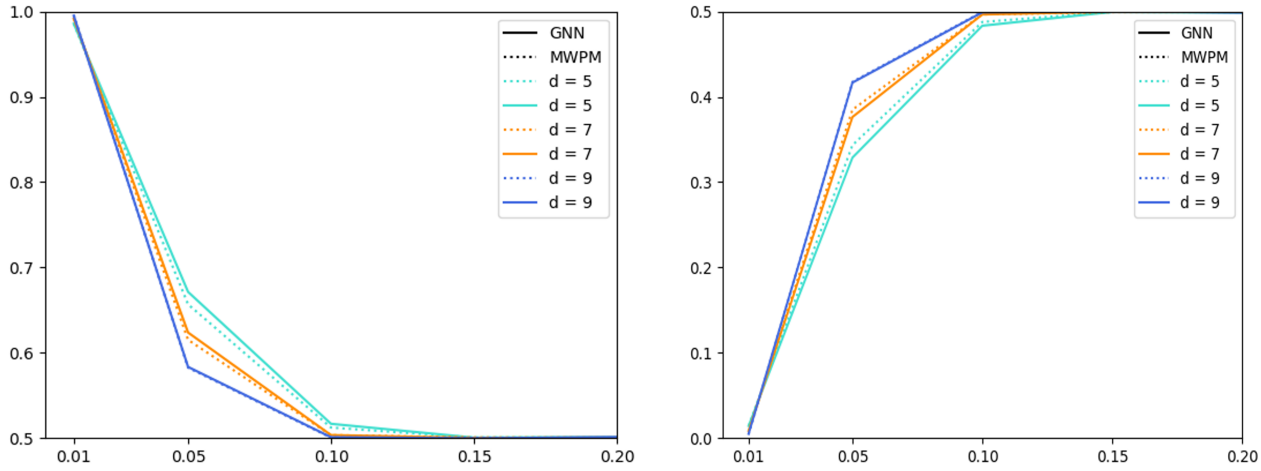


Figure 13: Logical failure rate (left) and accuracy (right) plotted over erorr rates (0.01, 0.05, 0.10, 0.15, 0.20), extracted from testing surface code with perfect stabilizers at distances 3, 5, 7 and 9.