Qudit Decoders Based on Neural Networks

Tobias Weber
tm.weber@stud.unibas.ch
20-052-908

Final Project for the
Quantum Computation and Error Correction
Lecture by Dr. James Wootton

January 2024

1 Introduction

Various different methods exist for decoding error syndrome graphs in the context of quantum error correction. They primarily focus on decoding problems for qubits ($k = 2$ states). A well-established decoder for this is based on finding a minimum weight perfect matching (MWPM decoder). Several papers have also employed feed-forward [1] or convolutional neural networks [2] for this decoding problem. The results are quite promising: The achieved logical error rates rival those of MWPM decodes while using less computational resources.

Instead of limiting ourselves to quantum systems with just 2 states, we can also consider $k > 2$ level quantum systems. The unit of quantum information for a general k is called qudit. Designing efficient decoders for topological quantum error correction codes using qudits can be quite challenging, but promise to be more resilient to noise [3].

The goal of this project is to explore the feasibility of neural network based decoders for qudit quantum error correction. We achieve this by evaluating the performance of a particular family of convolutional neural networks.

2 Training

The task for our model is to predict the correct boundary correction for decoding problems given by the *Decodoku* game [4]. The input given to the model is a decoding graph (representing the syndrome). This is a rectangular lattice with $d + 1$ rows and $d - 1$ columns, where d is the code distance. We are interested in the value of each node, representing the error. Instead of using an explicit graph, we can thus represent it instead as a $(d + 1) \times (d - 1)$ matrix.

Given such a matrix, the model should predict the boundary correction. It is sufficient to calculate the correction for just one of the two boundaries. The other one can be inferred from the first one. For evaluation, it is thus sufficient to only check one boundary.

2.1 Data Generation

A big advantage of the problem setting is that we can generate an arbitrary amount of training data on the fly and are not limited to using only existing data sets. This has the additional benefit of eliminating the problem of overfitting our model. We do not have to make a distinction between training data and data for testing.

For generating the syndromes and corresponding boundary corrections, we adapt the code used by the qiskit-qec Decodoku implementation. The same error model is used, where on each edge (including those to boundary nodes) with probability p an error can occur. Some changes have been made to increase performance, allowing us to generate millions of samples for training:

- Errors can not appear between two boundary nodes (they would be discarded anyways).
- Instead of generating random pairs of nodes, we choose a random subset of edges. It is thus not possible to have an error appear "twice" (For $k = 2$ this always results in their annihilation and thus effectively a lower value of p.).
- The expected number of errors is given by p times the number of edges ($p \cdot 2 \cdot d^2$). This is slightly lower than $p \cdot 2 \cdot (d+1)^2$ used by Decodoku. Decodoku thus uses an effectively higher value for p than our implementation (even when factoring in that errors between boundary nodes are discarded by Decodoku).

The effect of these differences can also be seen between Figure 6 and Figure 7.

Exploiting Symmetries To further increase performance when creating large batches of training data, we also use the fact that a horizontally or vertically mirrored decoding graph is still valid (swapping the boundary errors if required). From each generated sample, we thus get three additional samples for free. This also helps the model take advantage of the symmetric structure in the problem. For testing, this is not used, as we want to evaluate the model on more diverse data.

2.2 Network Architecture

Initial experiments have shown that the convolutional networks presented in [2] are superior to the feed-forward networks from [1]. We thus use a slightly modified version of their CNN-based architecture. It consists of:

- an input layer for a $(d+1) \times (d-1)$ matrix
- 6 hidden convolutional layers, each with 64 kernels of size 3
- a hidden fully connected layer with 512 nodes
- an output layer with k nodes, indicating the boundary correction

As an activation function after each hidden layer the ReLU function is used.

The idea behind using convolutional layers is that we want to find local patterns in the syndrome graph. For this, convolutions are the ideal choice while also reducing model complexity. By concatenating multiple such layers, patterns over longer distances can be detected. A sweet spot between training time and decoding performance was found with six such layers (increasing the number of kernels or their size did not significantly improve performance). The final fully connected layers can then summarize the detected patterns into the final boundary correction prediction.

2.3 Training Process

While the basic architecture is fixed, it still depends on specific values for the number of states k and the code distance d. This implies that for each combination of k and d, a separate model has to be trained. Instead of a one-size-fits-all approach, this has the advantage that each model is trained specifically for the given problem instances. This would also be the case for a real-world scenario, where the architecture of the quantum computer determines these values in advance.

To get a good understanding of the performance of the models, we trained models for $k = 2, 3, 4, 6, 10$. For d we used values $5, 7, 9, 11$. Every combination of k and d was trained, resulting in 20 individual models.

Hyperparameters Each model was trained using the Adam optimizer in combination with a cross-entropy loss. An epoch consisted of $2^{21} = 2097152$ training samples with a batch size of 4096. It typically took around 50 epochs for the loss to converge. The physical error rate p for the training samples was set equal to 0.1. As discussed in [1], training the network at a specific error rate optimizes the performance at that same rate. While it could be interesting to investigate the effect of changing it, this fixed rate proved to be a good choice.

3 Results

In this section we provide logical error plots for all of our models, with the goal of approximating the threshold. This threshold is significant as it determines up to which physical error probability the decoder can still be effective (by choosing a large enough d).

3.1 Method

We use the same data generation process as for training (without the mirrored samples). To determine the logical error probability, each model has to predict the boundary corrections for 50000 samples. This is done for 13 linearly spaced probabilities between 0.01 and 0.25.

We also evaluated the performance of the MWPM decoder discussed in the lecture. This was done once on data with errors generated by Decodoku itself, and once using our modified error generation method. As the decoder had a significantly higher runtime, the number of samples was reduced to 20000.

3.2 Threshold Plots

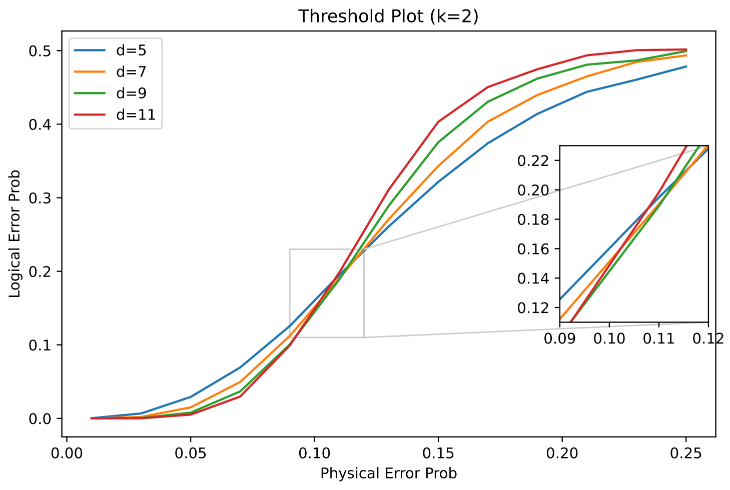

Figure 1: Logical error probabilities for the CNN-based decoder with $k = 2$.

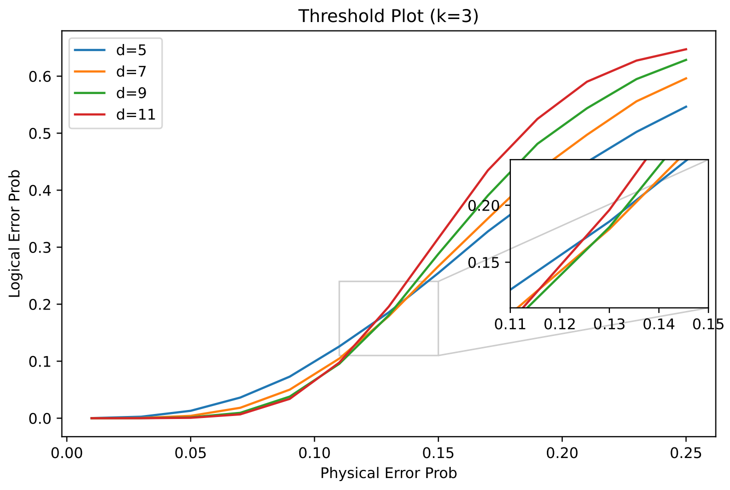

Figure 2: Logical error probabilities for the CNN-based decoder with $k = 3$.

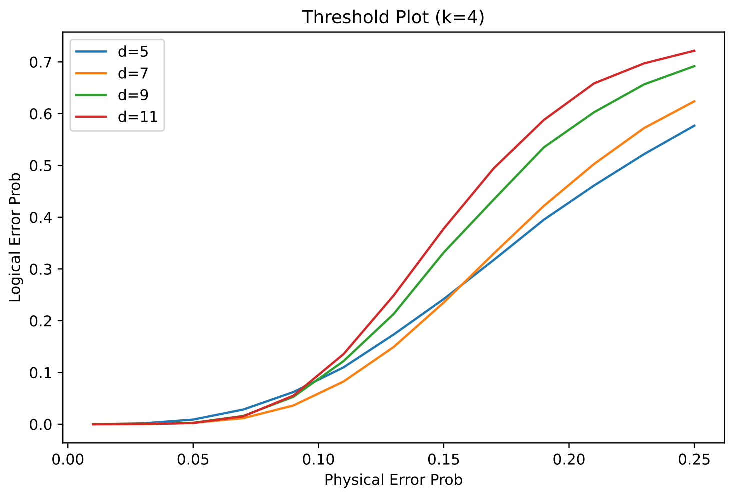

Figure 3: Logical error probabilities for the CNN-based decoder with $k = 4$.

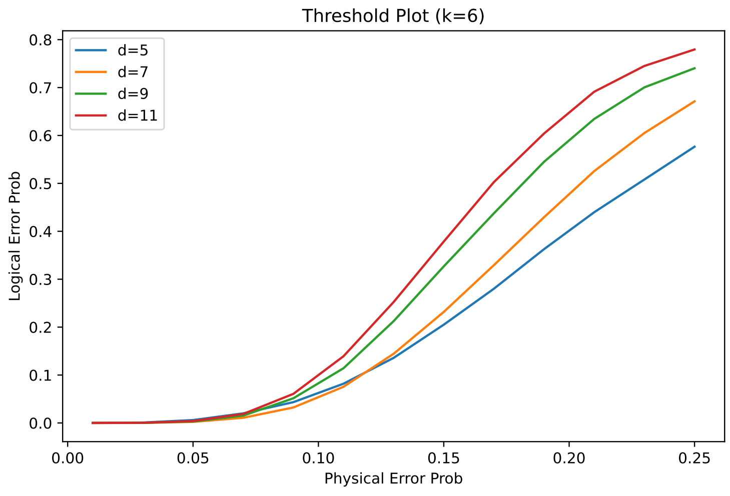

Figure 4: Logical error probabilities for the CNN-based decoder with $k = 6$.

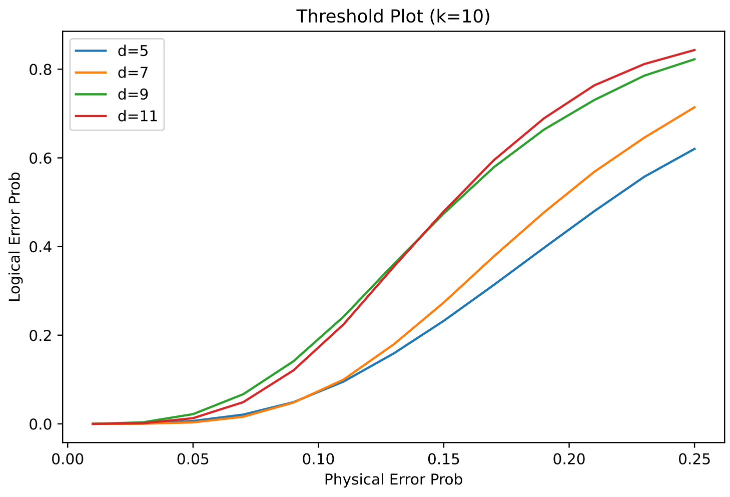

Figure 5: Logical error probabilities for the CNN-based decoder with $k = 10$.

3.3 MWPM Decoder

As expected, the two plots are quite different. The way Decodoku generates errors results in a threshold which is significantly higher (even higher than the theoretically optimal thresholds). To make any further analysis comparable, we will restrict ourselves to Figure 7, which uses the same data generation method as the other models.

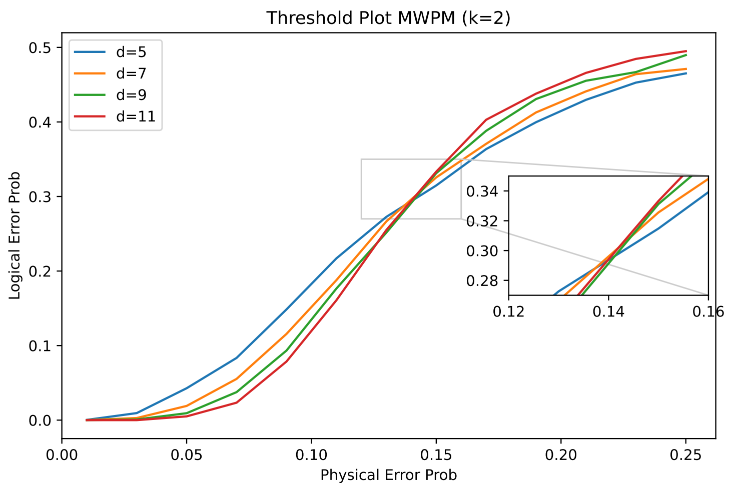

Figure 6: Logical error probabilities for the MWPM decoder from the lecture. Errors were generated by the *Decodoku* object itself.

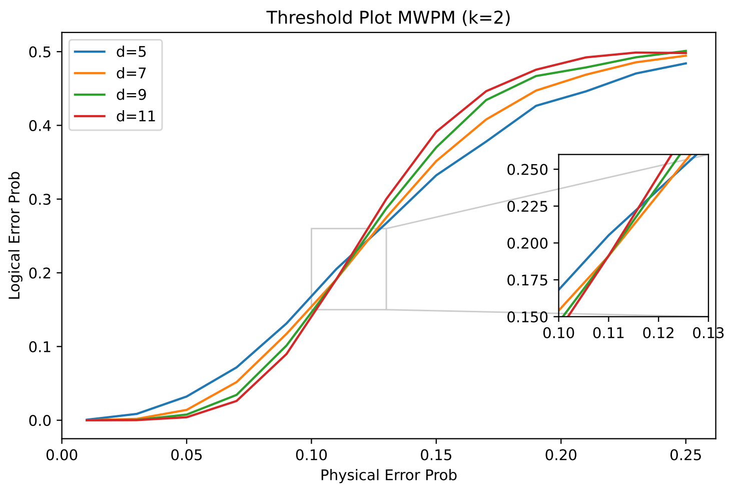

Figure 7: Logical error probabilities for the MWPM decoder from the lecture. Errors were generated with the method from Section 2.1.

4 Discussion

We summarize the observed thresholds in the following table:

Table 1: Error thresholds estimated from the diagrams before. We additionally state the estimations for the best possible thresholds by Andrist, Wootton, and Katzgraber [3].

Decoder	Threshold	Comment	Estimated thresholds [3]
MWPM	0.11	(0.14 with Decodoku error generation)	0.11
k=2	0.11		0.11
k=3	0.135		0.158
k=4	0.16	only considering d=5,7	0.19
k=6	0.12	only considering d=5,7	0.22
k=10	-	no reasonable threshold visible	0.26

We observe that for small k the CNN-based decoder performs very well. For $k = 2$ the performance is very similar compared to the MWPM-decoder. For both decoders, the threshold coincides with the optimal thresholds. It follows that our decoder might be a suitable alternative to existing decoders for the qubit case. Altough as [1] have shown, there exist even simpler neural networks for this task that perform at least as well. In practice, our model would take more computational resources than theirs, as it consists of much more parameters.

For $k = 3$ our decoder performs significantly better compared to $k = 2$. This highlights the potential of qudits for quantum error correction, as they allow for larger physical error probabilities. The achieved threshold is slightly below the optimal threshold. Visually observing the graph, it seems that the performance of $d = 9$ and $d = 11$ might be lower than expected. This can be explained by the fact that a larger syndrome also allows for more diverse patterns. This in turn requires a more complex model to learn them all.

For $k = 4$ this effect becomes even more prominent. While $d = 5$ and $d = 7$ intersect at an even higher value (as expected from theory), the other lines no longer follow this pattern. It seems that further increasing d only leads to higher logical error probabilities.

For $k = 6$ the threshold value has decreased, contrary to the expected trend. It is evident that the model does not work with this or even higher values for k. Higher values of d only make the problem worse, which becomes evident with $k = 10$.

4.1 Conclusion

We can conclude that for $k = 2, 3$ the model yields very promising results. Especially for $k = 3$, CNN-based decoders could be an interesting alternative to existing HDRG-decoders. By learning common patterns in the syndrome, even non-obvious samples can be correctly resolved. In a practical setting, there is also an additional advantage not yet discussed: The model can take hardware-specific variations in physical error probabilities into account by training it on modified syndromes.

The main drawback of using this model family is that it restricts us to comparatively small code distances. Larger values for d would quickly result in much larger networks, which are slower in training, but also during inference. This does not mean that it is infeasible, but it would require to tailor the network architecture more to the specific values for k and d. This could also be the right approach for designing decoders for larger k, where our model suffers from underfitting.

Finally we have found evidence that the error generation used by Decodoku results in skewed threshold plots. It appears that the proposed error generation solves this issue and is computationally less demanding.

4.2 Future Work

The results of this project suggest that CNN-based decoders are promising not only for qubit error correction, but also for qudits. Different architectures should be further investigated and adapted for specific use cases. It would also be interesting to compare them directly with other existing qudit decoders. Finally, evaluating the performance in terms of execution time would be useful to determine whether these decoders are viable in real-world applications.

References

[1] Ramon WJ Overwater, Masoud Babaie, and Fabio Sebastiano. “Neural-network decoders for quantum error correction using surface codes: A space exploration of the hardware cost-performance tradeoffs”. In: *IEEE Transactions on Quantum Engineering* 3 (2022), pp. 1–19.

[2] Simone Bordoni and Stefano Giagu. “Convolutional neural network based decoders for surface codes”. In: *Quantum Information Processing* 22.3 (2023), p. 151.

[3] Ruben S Andrist, James R Wootton, and Helmut G Katzgraber. “Error thresholds for Abelian quantum double models: Increasing the bit-flip stability of topological quantum memory”. In: *Physical Review A* 91.4 (2015), p. 042331.

[4] James R Wootton. “Decodoku: Quantum error rorrection as a simple puzzle game”. In: *APS March Meeting Abstracts*. Vol. 2017. 2017, pp. V27–010.

Appendix

Provided Scripts & Models

- `data_generation.py`
  Python script containing functions for syndrome graph generation.
- `decoder_model.py`
  Python script defining the neural network architecture of the model.
- `training.ipynb`
  Notebook used for training and exporting the models using *PyTorch*.
- `decoder_evaluation.ipynb`
  Notebook for generating the threshold plots for the CNN-based decoders.
- `mwpm_evaluation.ipynb`
  Notebook for generating the threshold plots for the MWPM decoder.
- `compiled/`
  Folder containing compiled versions of all trained models.
- `outputs/`
  Contains PDFs of all threshold plots.