Neural Network Decoders for Large-Distance 2D Toric Codes

Xiaotong Ni

QuTech, Delft University of Technology, P.O.Box 5046, 2600 GA Delft, The Netherlands.[*]

(Dated: September 19, 2018)

We still do not have the perfect decoders for topological codes that can satisfy all needs of different experimental setups. Recently, a few neural network based decoders have been studied, with the motivation that they can adapt to a wide range of noise models, and can easily run on dedicated chips without a full-fledged computer. The later feature might lead to fast speed and the ability to operate in low temperature. However, a question which has not been addressed in previous works is whether neural network decoders can handle 2D topological codes with large distances. In this work, we provide a positive answer for the toric code [1]. The structure of our neural network decoder is inspired by the renormalization group decoder [2, 3]. With a fairly strict policy on training time, when the bit-flip error rate is lower than 9%, the neural network decoder performs better when code distance increases. With a less strict policy, we find it is not hard for the neural decoder to achieve a performance close to the minimum-weight perfect matching algorithm. The numerical simulation is done up to code distance $d = 64$. Last but not least, we describe and analyze a few failed approaches. They guide us to the final design of our neural decoder, but also serve as a caution when we gauge the versatility of stock deep neural networks.

I. INTRODUCTION

Before we can make the components of quantum computers as reliable as those of classical computers, we will need quantum error correction so that we can scale up the computation. The surface code and other topological codes are popular choices for several qubit architectures because of its high threshold and low requirement on connectivity between qubits. However, several good performing decoders have trouble to do real-time decoding for qubits with fast error-correction cycles, such as superconducting qubits. Moreover, as we are getting closer to the point where small size surface code can be implemented in the lab, it is desirable that the decoders can adapt to the noise models from the experimental setups. These considerations motivate the study of decoders based on neural networks, which we will refer to as neural decoders, for surface code and other topological codes [4–11]. One question has not been addressed so far is whether neural networks can also be used for decoding 2D topological codes on a large lattice with good performance. In this work, we will focus on answering this question for the toric code. While it is the simplest topological code, it shares many common features with others, which makes it a good test platform.

To design a neural decoder for large toric codes, a natural first step is to use convolutional neural networks (CNNs) [12, 13], as the toric code and CNNs are both translational-invariant on a 2D-lattice. Compared to normal neural networks, the number of parameters in CNNs only scale with the depth of networks. This gives an intuition that the training of the CNNs should remain feasible for the lattice size of concern in the near future. We want the decoder to be able to adapt to experimental noise, which we should assume to be constantly changing and thus the data for calibration is limited. The structure of CNNs allow us to have a great control of how many parameters to be re-trained during calibration so that we can avoid over-fitting (see Appendix E for an example).

Interestingly, the renormalization group (RG) decoder [2, 3] for toric code already has a structure very similar to the CNNs used in image classification. Both of them try to keep the information needed for the output intact while reducing the size of the lattice, by alternating between local computation and coarse-grain steps. This similarity means that we should aim to achieve better or similar performance with the neural decoder compared to the RG one. And in case of bad performance, we can "teach" the neural decoder to use a similar strategy as the RG decoder. This is indeed how we get a good performance in the end. Conceptually, this is similar to imitation learning (see [14] for an overview). Even though we initialize the neural decoder by mimicking the RG one, it can have the following advantages:

- It can achieve a better performance than the RG decoder, as the latter one contains some heuristic steps. On the other hand, the neural decoder can be optimized to be a local minimum with respect to the parameters of the neural network (strictly speaking, at least the gradient is very small). The idea of improving belief propagation with neural networks is also used for decoding classical linear codes [15].

- It offers an additional way to adapt to experimental noise models, which is simply training on experimental data.

- The neural decoder can be run on dedicated hardware, thus easily parallelized and can decode multiple toric code syndromes simultaneously at a high
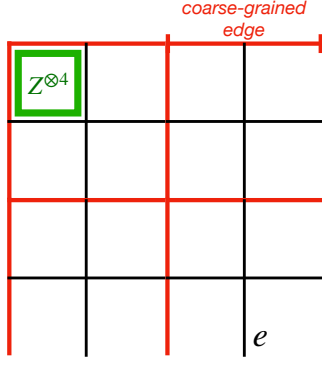
Figure 1: An illustration of the lattice and a B_p stabilizer check. The 2×2 unit cells are marked by the red color. We also give an example of a coarse-grained edge, which locates at the top-right and contains two edges of the original lattice.

efficiency.[1] In section V we will discuss more about this aspect of advantage.

It is tricky to evaluate the performance of neural decoders. As it stands, we need to train the neural nets for different lattice sizes separately, and the training process is not deterministic. Thus, we cannot define a threshold for the decoder. This is fine if our main goal is to have an adaptable decoder for near-future quantum devices. However, in order to know how optimal the neural decoders are, we still make a "threshold plot" under a well-studied noise model in Figure 4. Roughly speaking, the threshold benchmark is a good indicator of how good the decoder can process syndrome information. At the same time, we compare our neural decoder to the minimum-weight perfect matching algorithm in Figure 5, and show in Appendix E that our neural decoder can improve itself when trained on different error model. We hope these pieces of information together can give a first impression of neural decoders on toric code.

The focus of this paper is not on how to obtain an optimal neural decoder. Indeed, a lot of hyper-parameter optimizations can be done to further improve the performance or reduce the amount of data needed for training. Instead, we describe the key ideas that allow us to reliably obtain decent neural decoders for the toric code. The knowledge we gained can help us design neural decoders for other large codes.

----

[1] In theory, it is possible to convert RG decoder to some tensor computation, but at that point might as well do some end-to-end training and call it a neural decoder.

II. INTRODUCTION TO TORIC CODE AND THE RENORMALIZATION GROUP DECODER

A. Toric Code

First we give a brief introduction of toric code. Consider a $L \times L$ square lattice with periodic boundary condition, where a qubit lives on each edge. The stabilizer group of toric code is generated by two types of operator A_s and B_p

$$A_s = \bigotimes_{q \in n(s)} X_q, \quad B_p = \bigotimes_{q \in n(p)} Z_q, \qquad (1)$$

where s and p is any site and plaquette respectively, and $n(\cdot)$ consists of the 4 qubits neighboring s or p. The logical-Z operators have the form

$$\bar{Z}_i = \bigotimes_{q \in l_i} Z_q, \qquad (2)$$

where $l_{1,2}$ are two shortest inequivalent non-contractible loop. The toric code has a distance $d = L$.

In this paper, we will focus on the bit-flip noise model, i.e. only X errors can happen. We will also assume perfect measurements. Under this restriction, the quantum states will stay in the $+1$ eigenspace of A_s. Therefore, we only need to consider the expectation values of B_p and $\bar{Z}_i$. For simplicity, let us suppose in the beginning $\langle Z_i \rangle = +1$. And then a set of X errors happened, which leads to the syndrome $s = \{\langle B_p \rangle\}$. The goal of a decoder is to apply X to the qubits, such that $\langle B_p \rangle$ and $\langle Z_i \rangle$ return to $+1$. Without going to detail, we claim it is enough to know the parity of the number of X errors happened on the loops $l_{1,2}$. These two parities will be the final training target for our neural decoder. We will refer to the two parities as logical correction.

B. Renormalization Group Decoder

Let us first set up some notation. We will use e to denote an edge. When we say e is a coarse-grained edge, we mean e is an edge of a unit cell which consists of two (or more) edges of the original lattice. We use $x(e) = 1$ to denote an X error happened on edge e, and otherwise $x(e) = 0$. When e is a coarse-grained edge consists of edges $\{e_i\}$, $x(e) = \sum_i x(e_i) \mod 2$. Lastly, the marginal probability distribution of error on edge e is denoted by p_e. For example, the error rate of edge e is $p_e(1)$, and $p_e(0) = 1 - p_e(1)$.

One renormalization stage consists of the following:

1. Divide the lattice into $m \times m'$ unit cells, where in this work $m = m' = 2$.

2. The outputs of the renormalization step are $\{p_e\}$ for each coarse-grained edge e that is a border of

a unit cell. They are computed by belief propagation, which is a heuristic procedure for computing marginal probabilities (see Appendix A). These $\{p_e\}$ are treated as the error rate of the coarse-grained edge e for the next renormalization stage.

At the end of renormalization process, we obtain p_e for e being either of the two non-contractible loops. For simplicity, we assume the two non-contractible loops are $l_{1,2}$. Thus, we get an approximation of the marginal probability distribution for logical correction.

III. DESIGN AND TRAINING OF THE NEURAL NETS

At a first glance, to build a neural decoder, we can simply train a convolutional neural net with input-output pairs (syndrome, logical correction). However, in practice, this does not allow us to get a good enough performance. A detailed description of some simpler approaches and discussion will be presented in Appendix C. Those failures eventually motivate us to design and train the neural decoder in the following way.

A. Design of the network

The network follows the same structure as the renormalization decoder. Most of the network is repetitively applying the renormalization block, which is depicted in Figure 2. The belief propagation (BP) network, as its name suggested, is intended to approximate the BP process (see Appendix A for an introduction). More concretely, the first step of the training process is to train the BP network with the data generated by a handcrafted BP algorithm. This means initially the inputs to the BP network are syndromes and error rates on each edge, and the outputs are supposed to approximate the error rates on coarse-grained edges. However, later in the training process (i.e. global training mentioned in section III B), the BP network can deviate from this initial behavior. The post-processing has two steps. The first step is to remove the superficial complexity from the coarse-grained lattice. Whenever for a coarse-grained edge e has $p_e(1) > 0.5$, we apply an X on e and switch $p_e(1) \leftrightarrow p_e(0)$. If e is on either of the two non-contractible loops $l_{1,2}$, then the desired logical correction will be updated accordingly. Although this step only changes the representation of the data, and in principle, neural nets can learn to do the same thing, it is a quite costly step for neural nets as it can call the parity function multiple times. The second step is coarse-graining. We need to reduce the lattice size by half, and for convenience, this is done by the first layer of every belief propagation network. We also compute the parity of four $\langle B_p \rangle$ in each unit cell and feed these parities to the next BP network as the effective syndrome of the coarse-grained lattice.
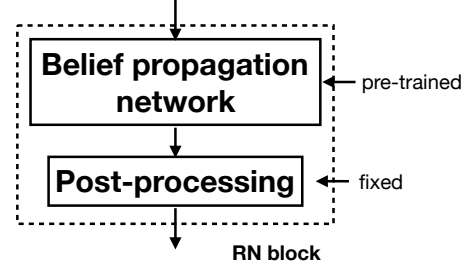

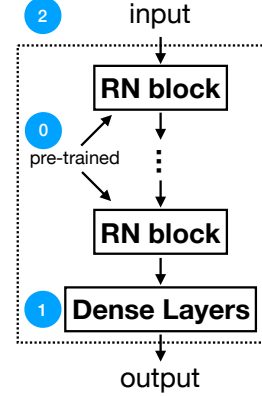Figure 2: Structure of each renormalization block (RN block).

Figure 3: Structure of the entire network, and the training order labeled by the blue circles. After loading the pre-trained belief propagation network into RN block, the first step is to train the dense layers, and the second step is to train all the layers together. We will call the second step global training.

In more detail, the input to the BP network can be packed in a tensor I with shape $(l, l, 3)$, where l is the initial lattice size or the output size of the precedent BP network. For example, we can set $I(i, j, 0)$ to be $\langle B_p \rangle$ on plaquette (i, j), and $I(i, j, 1), I(i, j, 2)$ to be the error rates corresponding to the top and left qubits of the plaquette. Each BP network consists of 13 convolution and 3 batch normalization layers. The definition of convolution layers can be found in Appendix B, and batch normalization is introduced in [16]. The first layer reduces the lattice size L by half. The reasoning is that the belief propagation is done based on 2×2 unit cells. The remaining layers keep the lattice size unchanged. Among them, only four involve communication between unit cells, i.e. the kernels of these four convolution layers have size 3×3. They spread evenly in the 13-layer network. Other layers only have kernels of size 1×1, which can then be viewed as computation inside unit cells. The rationale behind this is that the messages likely need to be processed before the next round of communication. The batch normalization layers also spread evenly, with the hope that they can make the training more stable.

After the renormalization process reduces the lattice

to a size of 2×2, we apply 4 dense layers (a.k.a fully-connected layers). Note that the dense layers conveniently break the translational symmetry imposed by the convolution layers. In the end, we have a neural network with input shape $(L, L, 3)$ and output shape $(2)^2$. The input shape is $(L, L, 3)$ because this is the input shape of BP networks. For $L = 64$, the total number of trainable layers in the network is around 60, which is very large compared to early deep neural networks [13]. However, most of the computation cost and the trainable parameters are concentrated in the 16 convolution layers with kernel size 3×3. Combining this and the careful training strategy we describe below, we find that the training can be done very efficiently.

B. Training

In general, training neural networks becomes harder when the number of layers increases. This is often attributed to the instability of gradient backpropagation. Considering we have a very deep neural network, we should find a way to train parts of the network first. The training is divided into two stages. First, we train the belief propagation network to indeed do belief propagation (BP). This corresponding the blue circle with 0 in Figure 3. To do this, we implement a BP algorithm and use it to generate training data for the network. More concretely, we first assign a random error rate e^{-k} to each edge, where $k \in [0.7, 7]$ from a uniform distribution. The choice of the distribution is quite arbitrary. Then we sample error on each edge according to its error rate and compute the syndrome. After that, we feed both the error rates and syndrome into our handcrafted BP algorithm, which will output an estimation of the error rates p_e corresponding to the coarse-grained edges. We can subsequently train the BP network with the same input-output relation. An important detail is that we transform the error rates $p_e(1)$ in both input and output to $r_e = \log \left(p_e(1)/p_e(0) \right)$. The reason behind this is described in Appendix C.

Next, we load the pre-trained belief propagation network into the decoder network described in the previous subsection. To ensure r_e stay bounded, we perform a rescale $r_e \rightarrow 7 r_e / \max_e |r_e|$ before feed it into next RN block (the choice of 7 here is arbitrary). We can then train the dense layers and afterward the whole network with input-output pairs (syndrome, logical correction). These two trainings correspond to the blue circle 1 and 2 in Figure 3, respectively. The training data is measurable in experiments in these two training.

We train the decoders for different lattice sizes L separately. Although this makes the concept of threshold
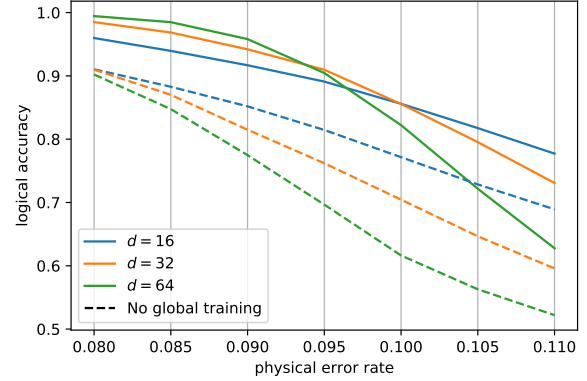

Figure 4: Logical accuracy versus physical error rate. The neural decoders are trained at physical error rate 9%. For the three solid lines, the decoder has been trained globally, while the dashed lines it has not. The colors of the dashed lines indicate the code distance they are evaluated on. The vertical grid indicates the physical error rates for which we evaluate the logical accuracy.

pointless, it is still useful to estimate the "threshold" so that we can have a rough comparison of the neural decoder with the existing ones. For this, we train the decoder for different L with the same amount of stochastic gradient steps, which also implies the optimizer sees the same amount of training data for each L. In addition, the training for each L is done under 1 hour (on the year 2016 personal computer with 1 GPU). We consider this to be a fairly strict policy. The result is plotted in Figure 4. We can also forgo this strict policy and spend more time in training the neural decoder for $d = 64$ toric code, which gives rise to Figure 5. The training time is still under 2 hours. More details about training can be found in Appendix D, and more discussion about the numerical results can be found in the following section.

IV. NUMERICAL RESULTS

For the strict training policy, we plot the logical accuracies versus the physical error rates in Figure 4. Logical accuracy is simply $(1 - \text{logical error rate})$ and is averaged over the two logical qubits. For the solid lines, the decoders have been trained globally, i.e. have done both step 1 and 2 in Figure 3. For the dashed lines, the decoders only did the step 1, i.e. only the dense layers are trained. The colors of the dashed lines indicate the code distance they are evaluated on. The vertical grid indicates the physical error rates for which we evaluate the logical accuracy, where for each point we sample 10^4 (syndrome, logical correction) pairs. We can see that the solid lines cross around $p_{\text{physical}} = 0.095$, therefore we might say our neural decoder has an effective threshold

---

[2] For efficient training, an additional dimension called batch size will be added.
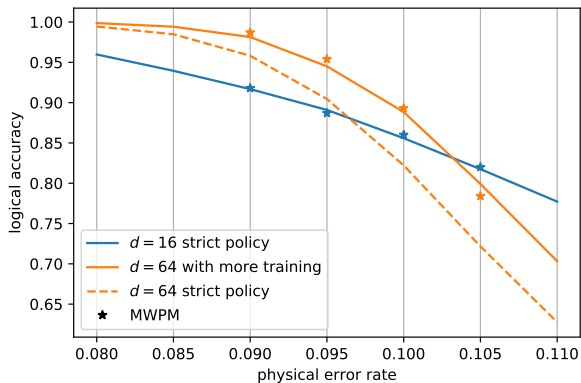
Figure 5: In this figure, we compare the performance of our neural decoders to the MWPM algorithm. The solid line for $d = 16$ and the dashed line for $d = 64$ decoder are using the strict training policy, while more training has been done on the $d = 64$ decoder corresponding to the solid line. The "star" points are the performance of the minimum-weight perfect matching algorithm. The colors of stars indicate the code distance they are evaluated on. The vertical grid indicates the physical error rates for which we evaluate the logical accuracy for the lines. We see the performance of neural decoders can be almost as good as MWPM for a decent range of physical error rates.

around 9.5%. It can be seen that the global training is crucial for getting a decent performance because without it the effective threshold will be below 8%.

We can also spend more time to train the $d = 64$ decoder, and then compare the performance of the neural decoders to the minimum-weight perfect matching algorithm (MWPM) in Figure 5. The "star" points are the logical accuracies of MWPM, where each one is evaluated by 3000 trials. The $d = 16$ decoder corresponding to the solid line and the $d = 64$ decoder corresponding to the dashed line are from the strict training policy. The $d = 64$ decoder corresponding to the solid line is obtained by doing more training while having the same network architecture. We see that without the strict training policy, the performance of the neural decoder is almost identical to MWPM for a decent range of physical error rates. We can also compare to the renormalization group (RG) decoder in [2], where the authors have shown a threshold of 8.2% when using 2×1 unit cell, and claim a threshold around 9.0% if using 2×2 unit cell. With the strict

training policy, our neural decoder is slightly better or at least comparable to the RG decoder, while without the policy our neural decoder is clearly better for $d \leq 64$.

V. DISCUSSION

One obvious question is whether we can get a good neural decoder for surface code or other topological codes on large lattices. In the case of surface code, the major difference compared to the toric code is the existence of boundaries. This means we have to inject some non-translational invariant components into the network. For example, we can have a constant tensor B with shape $(L, L, 2)$ marks the boundary, e.g. $B(x, y, i) = 1$ if (x, y) is at the smooth boundary and $i = 0$, or if (x, y) is at the rough boundary and $i = 1$; otherwise $B(x, y, i) = 0$. We then stack B with the old input tensor before feed into the neural decoder. More generally, if a renormalization group decoder exists for a topological code, we anticipate that a neural decoder can be trained to have similar or better performance. For example, neural decoders for surface code with measurement errors, for topological codes with abelian anyons can be trained following the same procedure described in this paper.

We want to discuss a bit more about running neural networks on specialized chips. It is straightforward to run our neural decoder on GPU or TPU [17] as they are supported by Tensorflow [18], the neural network library used in this work. There is software (e.g. OpenVINO) to compile common neural networks to run on commercially available field-programmable gate arrays (FPGAs), but we do not know how easy it is for our neural decoder[3]. Apart from power efficiency, there is a study about operating FPGAs at 4K temperature [19]. Overall, there is a possibility to run neural decoders at low temperature. Note that for running on FPGAs or benchmarking the speed, it is likely a good idea to first compress the neural networks, see [20].

VI. ACKNOWLEDGEMENT

---

[3] The only uncommon component of our neural decoder is element-wise parity function.

[1] A. Kitaev, Annals of Physics **303**, 2 (2003).
[2] G. Duclos-Cianci and D. Poulin, Physical review letters **104**, 050504 (2010).

[3] G. Duclos-Cianci and D. Poulin, Quantum Information & Computation **14**, 721 (2014).

[4] P. Baireuther, T. E. O'Brien, B. Tarasinski, and C. W. J. Beenakker, Quantum **2**, 48 (2018).

[5] S. Varsamopoulos, B. Criger, and K. Bertels, Quantum Science and Technology **3**, 015004 (2017).

[6] G. Torlai and R. G. Melko, Physical Review Letters **119**, 030501 (2017).

[7] N. P. Breuckmann and X. Ni, Quantum **2**, 68 (2018).

[8] P. Baireuther, M. D. Caio, B. Criger, C. W. J. Beenakker, and T. E. O'Brien, "Neural network decoder for topological color codes with circuit level noise," (2018), arXiv:1804.02926v1.

[9] S. Krastanov and L. Jiang, Scientific Reports **7** (2017), 10.1038/s41598-017-11266-1.

[10] N. Maskara, A. Kubica, and T. Jochym-O'Connor, "Advantages of versatile neural-network decoding for topological codes," (2018), arXiv:1802.08680v1.

[11] C. Chamberland and P. Ronagh, Quantum Science and Technology **3**, 044002 (2018).

[12] Y. LeCun *et al.*, Connectionism in perspective , 143 (1989).

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, in *Advances in Neural Information Processing Systems 25*, edited by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Curran Associates, Inc., 2012) pp. 1097–1105.

[14] A. Attia and S. Dayan, "Global overview of imitation learning," (2018), arXiv:1801.06503v1.

[15] E. Nachmani, Y. Be'ery, and D. Burshtein, in *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)* (IEEE, 2016).

[16] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," (2015), arXiv:1502.03167v3.

[17] N. P. Jouppi, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, C. Young, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, N. Patil, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Patterson, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, G. Agrawal, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, R. Bajwa, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, S. Bates, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, D. H. Yoon, S. Bhatia, and N. Boden, in *Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA '17* (ACM Press, 2017).

[18] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," (2015), software available from tensorflow.org.

[19] I. D. C. Lamb, J. I. Colless, J. M. Hornibrook, S. J. Pauka, S. J. Waddy, M. K. Frechtling, and D. J. Reilly, Review of Scientific Instruments **87**, 014701 (2016).

[20] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, IEEE Signal Processing Magazine **35**, 126 (2018).

[21] V. Kolmogorov, Mathematical Programming Computation **1**, 43 (2009).

[22] A. Hagberg, D. Schult, P. Swart, *et al.*, "Networkx," (2004–).

[23] G. Goh, Distill (2017), 10.23915/distill.00006.

[24] Y. Bengio and Y. LeCun, Large-scale kernel machines **34**, 1 (2007).

[25] D. Erhan, Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio, Journal of Machine Learning Research **11**, 625 (2010).

[26] D. Kingma and J. Ba, 3rd International Conference for Learning Representations, San Diego (2015).

Appendix A: Implementation of Belief Propagation Algorithm

Belief propagation is a heuristic procedure for computing marginal probabilities of graphical models. We choose to use a slightly different belief propagation implementation compared to [2], as ours seems to be more natural for the bit-flip noise model. We divide the lattice into 2×2 unit cells. Let G be a bipartite graph, where one part corresponds to unit cells, and another part to coarse-grained edges. Two vertices in G is connected when the coarse-grained edge is adjacent to the unit cell. This later decides how the messages flow in the graph. However, we assign two variables $\{x(e_i), x(e_j)\}$ to each vertex corresponding a coarse-grained edge which e_i and e_j form. In this section, the symbol e or e_i will denote original edges of the lattice (i.e. not coarse-grained). We define E to be the set of all edges e, $E_{cg} \subset E$ to be the set of e which are components of coarse-grained edges (i.e. red edges in Figure 1), and $\bar{E}_{cg} = E \setminus E_{cg}$. Given a syndrome S, the unnormalized probability of an error configuration $\vec{x} \equiv \{x(e)\}_{e \in E}$ can be written as

$$p(\vec{x}) = g(S, \vec{x}) \prod_{e \in E} p_e(x(e)), \qquad (A1)$$

where $g(S, \vec{x}) = 1$ if $\vec{x}$ has syndrome S, and otherwise $g(S, \vec{x}) = 0$. It is obvious $g(S, \vec{x})$ can be factorized to local terms. Thus, the marginal distribution for $e \in E_{cg}$ can then be factorized according to G as

$$\sum_{\{x(e), e \in \bar{E}_{cg}\}} p(\vec{x}) = \prod_c f_c(\{x(e)\}_c), \qquad (A2)$$

where the product is taken over all unit cells c, and $\{x(e)\}_c$ are the set of $x(e)$ such that $e \in E_{cg}$ is adjacent to c. We can then apply the standard belief propagation to the graph G. Without further explanation, we choose to use the following rule. A unit cell c_k sends to each of its adjacent cell c_n a message containing 4 real numbers

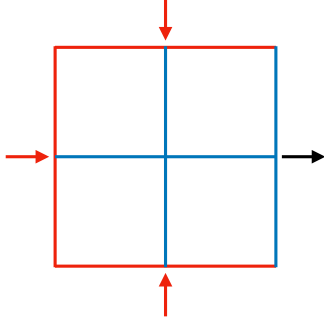$$\{m_{c_k, c_n}(x(e_i), x(e_j))\} \quad \text{for } x(e_i), x(e_j) = 0, 1, \qquad (A3)$$

Figure 6: An illustration of message passing for a unit cell.

where e_i and e_j form the coarse-grained edge between c_k and c_n. When we already fix an error configuration $\vec{x}$, we may use the simplified notations

$$p_e \equiv p_e(x(e)), \qquad m_{c_k,c_n} \equiv m_{c_k,c_n}(x(e_i), x(e_j)). \quad \text{(A4)}$$

To compute an out-going message from cell c, we take messages from the other three directions, and consider them to be the probability of error configuration on respective edges. We then sum over all error configurations in the cell which give the correct syndrome of the 4 plaquette stabilizer checks. More concretely, we define $\vec{x}_c$ to be $\vec{x}$ restricted to edges in c (i.e. all edges in Figure 6), and $g'(S, \vec{x}_c)$ checks whether $\vec{x}_c$ is compatible with S similar to g. We assume we want to send the messages from c to c_n, while the incoming messages are from cells $I = \{c_k\}$. Then we have

$$m_{c,c_n} = \sum_{\vec{x}_c} g'(S, \vec{x}_c) \prod_{c_k \in I} m_{c_k,c} \prod p_{e_i}. \quad \text{(A5)}$$

For the last term $\prod p_{e_i}$, the product is taken over the blue edges in Figure 6, assuming c_n is on the right of c.

In the end of the message passing, we can compute the marginal probability by

$$P(x(e_i), x(e_j)) = m_{c_k,c_n} m_{c_n,c_k} / \left(p_{e_i} p_{e_j} \right), \quad \text{(A6)}$$

where e_i and e_j are the edges between c_n and c_k. From the joint distribution $P(x(e_i), x(e_j))$ we can compute the distribution $P\left(x(e_i) + x(e_j) \mod 2\right)$. It is not hard to see that the above message passing rules will lead to the correct marginal probability when the underlying graph is a tree. (Note this is not the case for the square lattices we are considering)

The key differences between our implementation and the one in [3] is

- Ours utilizes all four stabilizer checks in each unit cell while in [3] only three are used.

- Each message contains 4 real numbers in our implementation while only 1 in [3].

Appendix B: Introduction to Neural Networks

A neural network, at the highest abstraction, can just be viewed as a black-box function $f_{\text{nn}}(x, \vec{w})$ with many parameters $\vec{w}$ to be tuned. We want f to describe the input-output relation presented in a dataset $D = \{(x_i, y_i)\}$. To do this[4], we choose a (smooth) loss function $\mathcal{L}$, and then we do the minimization

$$\min_{\vec{w}} \sum_i \mathcal{L}(f(x_i, \vec{w}), y_i). \quad \text{(B1)}$$

One important requirement is that f is (almost-everywhere) differentiable with respect to $\vec{w}$. This allows us to train the network with gradient descent, for which a good introduction can be found in [23]. In general, we can expect gradient descent will take us to a local minimum or some region with very small gradients. This is the advantage of "end-to-end" training compared to human-written heuristic algorithms, as the latter are unlikely to be a local optimum (assuming we can add real number parameters to those heuristic algorithms). A common loss function for classification problems is cross-entropy. Assume we have a dataset $D = \{(x_i, y_i)\}$ where $y_i \in \{0, 1\}$, and the neural network output y'_i which tries to approximate $\text{Prob}(y_i = 1)$, the cross-entropy loss function is then calculated as following:

$$-\sum_i \left(y_i \log y'_i + (1 - y_i) \log \left(1 - y'_i\right) \right). \quad \text{(B2)}$$

Note that when we use the notation $\text{Prob}(y_i = 1)$, we implicitly assume D is obtained by sampling from an underlying probability distribution.

More concretely, most neural networks consist of many layers. In this paper, the two relevant types of layers are the dense and convolution layer. Dense layers (a.k.a fully-connected layers) have the form $g(A\vec{x} + \vec{b})$, where g is some non-linear function applied entrywise, and the matrix A, vector $\vec{b}$ are the trainable parameters. Assuming A has a shape of $n \times m$, we will say the output dimension of the dense layer is n. One convolution layer, as the name suggests, contains a collection of discrete convolutions. For this paper, the input to the layer resides on a 2-dimensional lattice of size l^2 with periodic boundary condition. On each lattice site, there is a d-dimensional input vector $x_{\vec{u}} \in \mathbb{R}^d$, where the subscript $\vec{u} \in \mathbb{Z}_l^2$ (we use $\mathbb{Z}_l$ to denote integer in range $[0, l-1]$). We define the kernel to be a tensor $K_{\vec{u},i}$, where $\vec{u} \in \mathbb{Z}_n^2$ and $i \in \mathbb{Z}_d$. With a slight abuse of notation, we will say such a kernel has size n^2. The convolution is then

$$y_{\vec{v}} = \sum_{\vec{u} \in \mathbb{Z}_n^2} \sum_{i \in \mathbb{Z}_d} x_{\vec{v}-\vec{u},i} K_{\vec{u},i}, \quad \text{(B3)}$$

—————

[4] In this work, we choose to not study the effect of overfitting, as we have the ability to generate infinite data.

where $x_{\vec{v}-\vec{u},i}$ is the ith element of $x_{\vec{v}-\vec{u}}$, and $\vec{v}-\vec{u}$ is calculated module l because of the periodic boundary condition. We will have a collection of kernels $\{K_{\vec{u},i,j}\}_j$ for one convolution layer, which means we also have a collection of outputs $\{y_{\vec{v},j}\}_j$. The cardinality of $\{K_{\vec{u},i,j}\}_j$ is conventionally called the number of filters. After Equation B3, we can also apply a non-linear function g entrywise if needed.

Before concluding this section, let us make one clarification. In this paper, sometimes we only train part of the network, e.g. for blue circle 1 in Figure 3 we only train the dense layers. Assuming $\vec{w}_1$ are the parameters in the part of the network we want to train and $\vec{w}_2$ are the rest, then we are doing the optimization

$$\min_{\vec{w}_1} \sum_i \mathcal{L}(f(x_i, \vec{w}_1, \vec{w}_2), y_i) \qquad \text{(B4)}$$

by using some gradient descent optimizer.

Appendix C: Comparison to Simpler Approachers

In this section, we will show the performance of the neural net decoders when trained with simpler approaches (more precisely, approaches with less human involvement and prior knowledge of toric code decoding), and provide some reasoning if possible. The neural nets will be the same as the ones we used in the main text, except that they do not contain the "removing complexity" step in the post-processing. We will see in general the performance gets much worse, especially when the lattice size grows large. However, this does not mean these simpler approaches will always fail. It just imply that a large amount of training time / human involvement is needed, which could make them impossible in practice.

The simplest approach is to train the whole network with input-output pairs (syndrome, logical correction). This does not work. A hand-waving explanation is the following. It is fair to assume a lot of parity functions need to be evaluated during the decoding process. It is known that the parity is not an easy function for neural nets to compute [24], and one good way to approximate it is to increase the depth of the network. So let us assume each renormalization stage need 5 layers. This means to decode $L = 32$ toric code, the network will have 25 layers, which exceeds the range where neural nets can be reliably trained.

The problem of too many layers can be alleviated if we can pre-train the earlier layers of the network. Similar strategy was used in training neural nets for computer vision problems [25]. For the bit-flip noise model, we can pre-train the earlier layers to mimic the renormalization group decoder. Recall in section II, we mentioned that for the renormalization group decoder, the output corresponding to a coarse-grained edge e is p_e. Since we generate syndromes by first sampling $x(e)$ for all (not

Block	RN1	RN2	RN3	RN4	Dense
Cross-entropy	0.16	0.22	0.28	0.4	0.5
Accuracy	0.93	0.90	0.86	0.79	0.58

Table I: Cross-entropy and accuracy after each renormalization block when we do the training block by block. It is done with a bit-flip error rate 0.08. Although the accuracy after the first RN block is comparable to the input error rate, it gradually decreases until barely above 0.5

coarse-grained) edges e, we also have the ability to generate pairs (syndrome, $\{x(e)\}$ for coarse-grained edges e) for training. Although $\{x(e)\}$ in the above pairs are binary numbers, with the cross-entropy as the cost function, in theory the output will converge to $p_e(1)$. The pre-training is done one renormalization block at a time. More concretely, with the network we are using in the main text, we will train the output of the 12th layer with the training target of first renormalization block, and the output of the 24th layer with the second block, etc. We can try this method on $L = 32$ toric code and bit-flip error rate 0.08. It does not work well, as we can see in Table I. These numbers are not very accurate and coming from a single training instance. However, the author has observed the same trend many times that the loss and accuracy slowly degrade in the process of renormalization, even though the error rate is way below the theoretical threshold. To diagnose the reason, we first notice that the first RN block actually performs reasonably well. This suggests that the optimizer is capable of training each RN block alone. Assume this is indeed the case, the degrading of performance is likely caused by the following two reasons.

First, later in the renormalization process, if we look at the coarse-grained syndrome or p_e alone, they behave more and more like white noise. While it is possible for the neural nets to do the same post-processing described in section III A, a few layers of the network will be occupied by this. Therefore, the natural solution is to implement the post-processing ourselves. By doing this, we suspect it is possible to reach a threshold of 8%, but apparently 8% is still not good enough.

The second reason is related to the convergence of p_e. When below the threshold, we will encounter very often that p_e is very close to 0 or 1. For example, if $x(e) = 1$ vs $x(e) = 0$ corresponding to a weight 4 vs weight 1 local configuration, then we will have $p_e \approx p_0^3$, where p_0 is the initial error rate. This will become more prominent later in the renormalization process, as p_e from previous renormalization stage become the error rate in the next stage. It is important to know how close to 0 (or 1) p_e is on the logarithmic scale. Otherwise, in the later stage, the information will not be accurate enough to deduce configuration close to minimum weight of errors. This poses the following requirements:

- When we pass p_e to some intermediate layer of

a neural net, it should be able to distinguish between small p_e. However, recall that each layer does the computation $f(Ax + b)$, where f has a bounded derivative. Thus, to distinguish a set of small $\{p_e\}$, we need $\|A\| \sim 1/\min\{p_e\}$. This will either not achieved by training, or cause instability of the network. Another issue related to the minuscule nature of p_e is the cross-entropy loss function does not provide enough motivation for p_e to converge to the target value q in log-scale. More accurately, the derivative of the cross-entropy scale like $O(|p_e - q|)$ when $p_e \approx q$, which will be too small before the convergence in log-scale. A natural solution is we replace the appearance of p_e with $\log p_e - \log(1 - p_e)$.

- Even with a good representation of p_e, we shall still be very cautious about the training, as we are trying to estimate very small p_e from sampling. In the end, we decide to implement a belief propagation routine, and use the input-output pair from the routine to train the network. The advantage is that belief propagation directly output the probability, which should be reasonably accurate in the logarithmic scale. Therefore, we can get a much stable training process.

Appendix D: Technical Details

The objective of this section is to describe some technical details for people who do not plan to read the source code.

For the majority of network, we use leaky ReLUs as the activation function, which has the form

$$y = x \quad \text{if } x > 0; \quad y = 0.2x \quad \text{if } x \leq 0 \qquad \text{(D1)}$$

Apart from the last layer of each renormalization block and the last dense layer, the number of filters in each convolution layer is 200, and the output dimension of each dense layer is 50.

The optimizer we use is the ADAM [26]. The learning rate parameter of the optimizer is set to 7×10^{-4} for training belief propagation network, 10^{-3} for training dense layers, 7×10^{-5} for global training of $L = 16, 32$ lattice, and 7×10^{-6} for $L = 64$. The batch size for training is 50. The training of the dense layers uses around 1000 batches. For the strong policy, the global training uses 3000 batches regardless of the code distance. To see the potential of our neural decoder at $d = 64$, we also did a longer training and compared it to MWPM in Figure 5. In total, it is trained using 18000 batches. The first 12000 batches are trained on physical error rate $p = 0.09$, and the last 6000 batches are on $p = 0.095$. The reason of switching to a higher error rate for late stage training is that the accuracy at $p = 0.09$ is too close to 1 for effective training.

Appendix E: Spatially Varying Error Rates

A natural use of the neural decoder is to train it with experimental data. Naturally, the noise models in the experiments will not be translational invariant. There are two simple ways to reconcile this fact with the translational-invariance of convolutional neural nets:

1. Allow the first few layers of the network to be nontranslational invariant.

2. Introduce site-dependent trainable variables to the networks.

In this section, we will consider the error model that has varying bit-flip error rates across the lattice. For this, we can use the second approach, where the site-dependent variables can in principle represents the varying error rates. In fact, recall that the neural decoder has an input shape $(L, L, 3)$, which contains $2L^2$ numbers that are originally error rates feeding into belief propagation. Thus we can simply feed the site-dependent variables into the neural decoder and then train them.

However, there is still a complication regarding the starting point of this training. A natural choice is to start with the trained neural decoder for uniform error rate and only train the site-dependent variables. With this route, there is a risk that if previously we trained the neural decoder under uniform error rate for long enough, the neural decoder could learn to ignore the error rate inputs as they are constant. In this case, only training the site-dependent variables can fail.

Another route we can take is that we also reinitialize the first renormalization block. More accurately, we do the following:

1. We start with the trained neural decoder for uniform error rate.

2. We reverse the belief propagation network in the first renormalization block to the pre-trained weights. In other words, it now again approximates belief propagation.

3. We then train the site-dependent variables and the first renormalization block together.

We test this scheme with a distance 16 toric code. For each qubit, there is a 50% chance it has bit-flip rate 0.16, and a 50% chance of rate 0. In other words, only around half of the qubits are noisy. With this new error model, we train the site-dependent variables and the first renormalization block for 4500 batches, where each batch contains 50 training data. The learning rate of the ADAM optimizer is set to 2×10^{-4}. By performing this training, we increase the logical accuracy from 0.967 to 0.993, where 0.967 corresponding to the decoder trained on uniform bit-flip error model. The accuracy is each evaluated by using 10^5 (syndrome, logical correction) pair. To provide some comparison, we run the minimum-weight

perfect matching (MWPM) algorithm for the same error model. Without providing error rate information, the MWPM algorithm assigns an equal weight to each qubit, and have a logical accuracy of 0.975. If we provide the perfect information about the error rates, we can assign weight 1 to noisy qubits, and weight 100 to the noiseless ones. For each pair of violated parity checks, we can select the path with the minimum total weight between them, and use this weight for the MWPM algo-rithm. With this choice, the logical accuracy rises to 1, e.g. 100% success. These accuracies are each evaluated by 10^4 (syndrome, logical correction) pair.

Based on the thoughts above, it is likely better to not start with the neural decoder trained on uniform error rate. Instead, we can train a neural decoder with training data that has varying error rates, but otherwise the same procedure as depicted in Figure 3. This way, the first renormalization block will not learn to ignore the error rate inputs.