Learning to Do Renormalization Decoding for Toric Code with Neural Nets.

Xiaotong Ni

(Dated: August 14, 2018)

Mimic soft decision renormalization decoder to get a faster and more flexible neural net decoder.

There is a ongoing search of decoders for toric code to satisfy the needs of different experiment setups. Recently, a few neural network based decoders have been studied.

Last but not least, we describe and analyze a few failed approaches, which 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, which derived from the toric code, is a popular choice 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 point where small size surface codes 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 neural network decoders. One question has not been tackled so far is whether neural networks can also be used for decoding 2D toric code on a large lattice with good performance.

To design a neural decoder for large toric codes, a natural first step is to use convolutional neural networks (CNN), as the toric code and CNN are both translational-invariant on a 2D-lattice. In most case, including the CNN used in this paper, the number of parameters in the CNN only grows logarithmically w.r.t the lattice size. This gives a theoretical 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).

Interestingly, the renormalization decoder [2, 3] for toric code already has a structure very similar to the CNNs used in image classification. Both of them do "real-space renormalization" until the lattice is reduced to a very small size, and each renormalization stage consists of several rounds of local computation and a coarse-grain step. This similarity means that we should aim to achieve better or similar performance with the neural net decoder compared to the renormalization one. And in case of bad performance, we can "teach" the neural net decoder to use a similar strategy as the renormalization decoder. This is indeed how we get the good performance decoder in the end. Conceptually, this is similar to imitation learning (see [1] for an overview). Even though we initialize the neural decoder by mimicking the renormalization one, it can have following advantages:

- It can achieve a better performance than the renormalization decoder, as the latter one contains some heuristic steps and thus is likely not a local minimum (in some continuous manifold of algorithms). On the other hand, the neural decoder can be optimized to be a local minimum (strictly speaking, at least the gradient is very small.)

- It offer an additional way to adapt to experimental noise, which is simply doing end-to-end training on experimental data.

- The neural decoder can be run on dedicated hardware, thus easily parallelized and can decode multiple syndrome input at a high efficiency. (In theory, it is possible to convert renormalization decoder to some tensor computation, but at that point might as well do some end-to-end training and call it a neural decoder.)

It is tricky to evaluate the performance of neural net decoders. As it stands, we need to train the neural nets for different lattice sizes individually, 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 NN decoders are, we still make a "threshold plot" under a well-studied noise model. As long as an NN decoder does not rely on the noise model (e.g. explicitly taught to do MWPM), and the noise model do not change too drastically (e.g. become non-local), we can hope the decoder will adapt to the noise and perform similarly well.

The focus of this paper is not on how to obtain an optimized NN decoder. Indeed, there are a lot hyper-parameter optimization can be done to further improve the performance. Instead, we describe the key ideas that allow us to reliably obtain decent NN decoders for toric code. The knowledge we gained can help us design NN decoder for other large codes.

II. INTRODUCTION TO TORIC CODE AND THE RENORMALIZATION DECODER

A. Toric Code

First we give a brief introduction of toric code. Consider a 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

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

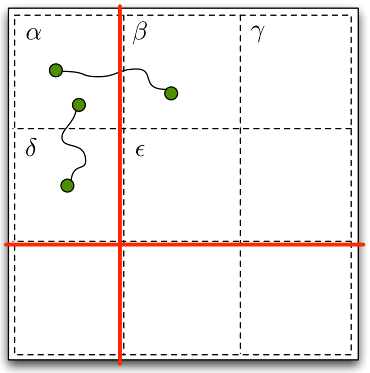
where s and p is any site and plaquette respectively, and $n(\cdot)$ consists of the 4 qubits neighboring s or p. The logical-Z operators have the form

$$\bar{Z}_i = \bigotimes_{q\in l_i} Z_q, \tag{2}$$

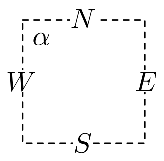
where $l_{1,2}$ are two non-contractible loop

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 $+1$.

B. Renormalization Decoder



(a)



(b)

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 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 corresponding to the support of logical operators Z_i. This allows us to correct the value of the corresponding logical operator.

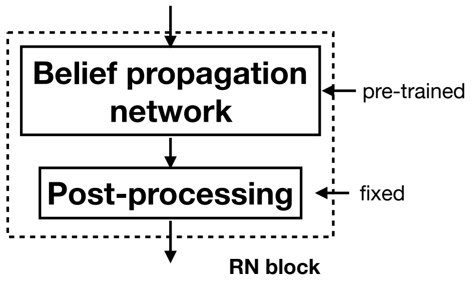
III. DESIGN AND TRAINING OF THE NEURAL NETS

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

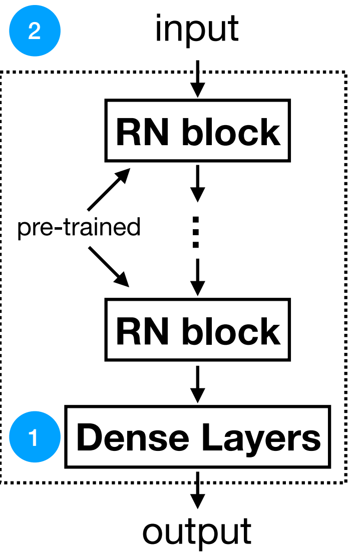
A. Design of the network

The network follow the same structure as the renormalization decoder. Most of the network is repetitively applying the renormalization block, which is depicted in Figure 1a. The belief propagation network, as its name suggested, is intended to approximate the belief propagation process (see Appendix A for an introduction). This means the inputs to the network are syndromes and error rates on each edge, and the outputs are supposed to approximate the error rates on coarse-grained edges. The post-processing has two steps. The first step is to remove the superficial entropy from the coarse-grained lattice. Whenever for a coarse-grained edge e has $p_e \equiv P(\text{X error}) > 0.5$, we apply an X on e and set $p_e \leftarrow 1 - p_e$. Although this step only changes the representation of the data (and in principle the neural net 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. For convenience, this is done by the first layer of every belief propagation network.

In more detail, each belief propagation network consists of 13 convolution and 3 batch normalization layers. The first layer reduce the lattice size L by half. The reasoning is that the belief propagation is done based on 2×2



(a) Structure of each Renormalization block (RN block).



(b) Structure of the entire network and the training order.

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 kernel 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 reduce the lattice to a small size (e.g. 2×2), we apply a few fully-connected layers. Note that the fully-connected 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)[1].

For $L = 64$, the total number of trainable layers in the network is around 60. It is a lot of layers compares to early deep neural networks [6]. 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 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. To do this, we implement a belief propagation 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 belief propagation algorithm, which will output an estimation of the error rates corresponding to the coarse-grained edges. We can subsequently train the belief propagation network with the same input-output relation.

Next, we load the pre-trained belief propagation network into the decoder network described in the previous subsection. We can then train the fully-connected layers, and lastly the whole network with input-output pairs (syndromes, logical correction). At this stage, the training data is measurable in experiments. For this paper, we train the neural nets with simulated bit-flip noise at error rate $p = 9\%$

We want to make two comments regarding the training process:

1. It explicitly guides the neural network to mimic the renormalization decoder. The advantage is that we have a clear path in how to design and train the network, as well as having some interpretability during training. The disadvantage is that it is unlikely to get a drastically different strategy compared to the renormalization decoder.

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

[1] For efficient training, an additional dimension called batch size will be added.

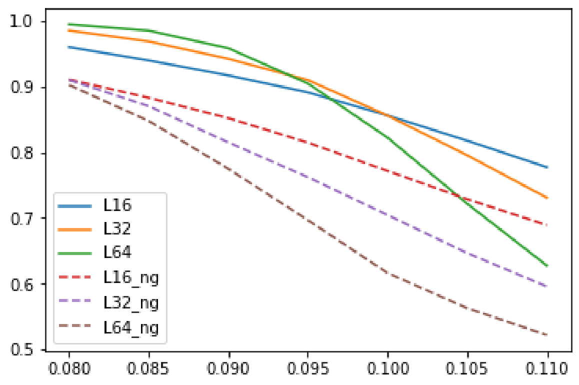
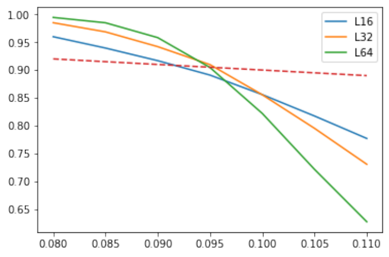


Figure 2: Logical error rate versus physical error rate. The neural decoders are trained at physical error rate 9%. For the solid lines, the decoder has been trained globally

(on a year 2016 personal computer with 1 GPU). More details about training can be found in Appendix D.

IV. NUMERICAL RESULTS



In fig [], we plot the logical error rates versus the physical error rates. The logical error rates are 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 1b. For the dashed lines, the decoders only did the step 1 of training. It can be seen that the global training is crucial for getting a decent performance. The solid lines and the $p_{\text{logical}} = p_{\text{physical}}$ all cross around $p_{\text{physical}=0.095}$, therefore we might say our neural decoder has an "effective threshold" around 9.5%.

V. FUTURE WORK

data processing inequality: If a decoder work well, then the hidden layers still contains enough information for decoding. Meaning we can reuse first few layers from smaller decoder?

find a way to quantify the variance in the noise model. (Auto-encoder style)

[1] Alexandre Attia and Sharone Dayan. Global overview of imitation learning.
[2] Guillaume Duclos-Cianci and David Poulin. Fast decoders for topological quantum codes. *Physical review letters*, 104 (5):050504, 2010.
[3] Guillaume Duclos-Cianci and David Poulin. Fault-tolerant renormalization group decoder for abelian topological codes. *Quantum Information & Computation*, 14(9-10): 721–740, 2014.
[4] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660, 2010.
[5] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *3rd International Conference for Learning Representations, San Diego*, 2015. URL `https://arxiv.org/abs/1412.6980`.
[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012. URL `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural pdf`.

Appendix A: Introduction to Belief Propagation

Belief propagation is a heuristic procedure for computing marginal probabilities of graphical models.

1. Implementation of the Belief Propagation

We choose to use a slightly different belief propagation implementation compared to [], as ours seems to be more natural for the bit-flip noise model. We divide the lattice into 2 × 2 unit cells. Consider a bipartite graph

G, where one part corresponding to unit cells, and another part corresponding to edges of the original lattice (i.e. not coarse-grained edges). Two vertices in G is connected when the edge is adjacent to the unit cell. The probability of an error configuration $\{x(e)\}$ (before normalization) can then be factorized according to G as

$$\prod_c f_c(\{x(e)\}_c), \tag{A1}$$

where the product is taken over all unit cells c and $\{x(e)\}_c$ are the set of $x(e)$ such that e is adjacent to c. We can then apply the standard belief propagation to the graph G. A unit cell sends to each of its adjacent cell 4 messages $\{m_e(x(e_1), x(e_2)\}$, where e is a coarse-grained edge consists of e_1, e_2. When the context is clear, we may use the simplified notation

$$p_e \equiv p_e(x(e)), \qquad m_e \equiv m_e(x(e_1), x(e_2)) \tag{A2}$$

To compute an out-going message, 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 which give the correct syndrome of the 4 plaquette stabilizer checks. More concretely, we have

$$m_e^{\text{out}}(x(e_1), x(e_2)) = \sum_{\vec{x}\in E} \prod m_{e_i}^{\text{in}} \prod p_{e_j}. \tag{A3}$$

In the above equation, E contains error configurations $\vec{x} \equiv \{x(e_i)\}$ which give the correct syndrome of the 4 plaquette stabilizer checks. In the end of the message passing, we can compute the marginal probability by

$$P(e_i, e_j) = m_{e_i,e_j} m'_{e_i,e_j} / \left(p_{e_i}(e_i) p_{e_j}(e_j)\right) \tag{A4}$$

where $e_{i,j}$ belong to the same coarse-grained edge, and m, m' are the messages from its two adjacent cells. 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)

As discussed in [2], the idea of renormalization decoder is linked to viewing toric code approximately as a concatenated code. This link can help us to estimate the "threshold" of some work in progress neural decoder, as we are constantly switching design and training methods. Consider a genuine concatenated code $\{\mathcal{C}_n\}$ based on a code $\mathcal{C}$. Let us assume that $\mathcal{C}$ has a pseudo-threshold p' for bit-flip noise, by which we mean if the physical error rate $p_e < p'$, then the logical error rate $p_l < cp_e$ where $c < 1$. It is then obvious that the asymptotic threshold p of $\{\mathcal{C}_n\}$ satisfies $p > p'$. In terms of neural decoder for toric code, this translates to the intuition that if for physical error rate p_e, we can predict $x(e_1)$ for some coarse-grained edge with error rate lower than p_e, then the "threshold" of the neural decoder will likely be larger than p_e. The converse also holds: if the prediction for some $x(e_1)$ is worse than the physical error rate, then we are likely above the threshold. It is also interesting to observe in [] that the pseudo-threshold of each lattice size is quite close to where the curves cross.

Appendix B: Introduction to Neural Network

Neural nets, 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, we choose a (smooth) loss function L, and then we do the minimization

$$\min_{\vec{w}} \sum_i L(f(x_i, \vec{w}), y_i) \tag{B1}$$

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. 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 in those heuristic algorithms).

More concretely, most neural networks consist of many layers. In this paper, the two relevant types of layers are the fully-connected and convolution layer. Fully-connected layers have the form $g(A\vec{x}+\vec{b})$, where g is some non-linear function, and the matrix A, vector $\vec{b}$ are the trainable parameters. Convolution layers, as the name suggests, have the form of discrete convolution

$$f * x(\vec{u}) = \sum_{\vec{v}} f(\vec{v}) x(\vec{u} - \vec{v}). \tag{B2}$$

Here x is the N-dimensional input array, and f is the trainable kernel, i.e. a function with small support. $\vec{u}, \vec{v}$ are the coordinates of N-dimensional array. In this paper, since we have toric code syndromes as inputs, which have a periodic boundary condition, it is natural to perform convolution with the same boundary condition. More concretely, $\vec{u} - \vec{v}$ in the above equation is calculated module lattice size L.

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 entropy" steps. 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 (syndromes, logical correction).

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

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 a natural function for neural nets to compute, 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 [4]. For the bit-flip noise model, we can pre-train the earlier layers to mimic the renormalization decoder, by computing training target for the outputs of renormalization blocks. Recall in Appendix A, we mentioned that the output corresponding to a coarse-grained edge e is p_e, the marginal probability of a string crossing e. The training target, which is computed based on the complete information of errors, is of course binary. However, with the cross-entropy as the cost function, in theory the output will converge to p_e. 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 0.8. It does not work well, as we can see in Table **??**. 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 degrades later into the renormalization, even though the error rate is way below the theoretical threshold. This 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 not hard 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 a string cross e vs no string cross e corresponding to a weight 3 vs weight 1 local configuration, then we will have $p_e \approx p_0^2$, where p_0 is the initial error rate (for simplicity, we assume $p_0 = 0.1$ from now on). 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.

The optimizer we use is the ADAM [5]. The learning rate parameter of the optimizer is set to 7×10^{-4} for training belief propagation network, 7×10^{-5} for global training of $L = 16, 32$ lattice, and 7×10^{-6} for $L = 64$.

ADAM keeps track of t by saving $beta^t$. Batch normalization moving mean is trainable in Tensorflow.

Appendix E: Varying Error rates

A natural usage of the neural network decoder is to train it with experiment 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 non-translational invariant.

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

In this section, we will consider the error model that has a 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 a 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 learned 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 weights in the first renormalization block to the end of first stage training in section III B. In other words, the belief propagation network in the first renormalization block now again approximate belief propagation.

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

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 1b. This way, the first renormalization block will not learn to ignore the error rate inputs.