

# Scalable Neural Network Decoders for Higher Dimensional Quantum Codes

N. P. Breuckmann<sup>1</sup> and X. Ni<sup>1,2</sup>

<sup>1</sup>Institute for Quantum Information, RWTH Aachen University, Germany

<sup>2</sup>Max Planck Institute of Quantum Optics, Germany

May 16, 2018

Machine learning has the potential to become an important tool in quantum error correction as it allows the decoder to adapt to the error distribution of a quantum chip. An additional motivation for using neural networks is the fact that they can be evaluated by dedicated hardware which is very fast and consumes little power. Machine learning has been previously applied to decode the surface code. However, these approaches are not scalable as the training has to be redone for every system size which becomes increasingly difficult. In this work the existence of local decoders for higher dimensional codes leads us to use a low-depth convolutional neural network to locally assign a likelihood of error on each qubit. For noiseless syndrome measurements, numerical simulations show that the decoder has a threshold of around 7.1% when applied to the 4D toric code. When the syndrome measurements are noisy, the decoder performs better for larger code sizes when the error probability is low. We also give theoretical and numerical analysis to show how a convolutional neural network is different from the 1-nearest neighbor algorithm, which is a baseline machine learning method.

## 1 Introduction

A full-featured quantum computer will rely on some form of error correction as physical qubits are prone to the effects of environmental noise.

The authors contribute equally to this work.

When using error correcting codes, decoders play a large role in the performance of the fault-tolerant protocols. Using neural networks to decode the surface code has been suggested in earlier works [3, 21, 36, 37]. The primary motivation inspiring these works to use neural networks is their ability to adapt to the error model. However, an issue that has not been addressed so far is scalability: in previous approaches, the neural networks have to be re-trained for every system size, despite the fact that in general machine learning methods become problematic when the input space becomes too large. Thus, it is interesting to ask whether there exists a family of quantum error correcting codes, which we can decode using neural networks in a clearly scalable way.

To provide a positive answer to this question, we introduce a decoder for the four-dimensional version of the toric code based on neural networks for which the training only has to be performed once on a small system size. Afterwards the decoder can be scaled up to arbitrary system sizes without re-training. The layout of the network is a *convolutional neural network* which are widely used as a building block in image recognition. Furthermore, the neural network is constant depth with respect to the scaling of the system size. Our approach is informed by the existence of local decoders for the 4D toric code [6, 7, 11, 28]. A drawback of those decoders is their low threshold of less than 2% against independent noise (assuming perfect syndrome measurements). The (global) minimum-weight decoder, on the other hand, has a threshold of 10.1% [33] but it is not known to be computationally efficient. Our decoder shown a threshold of 7.1% while having a complexity close to local decoders. This result is very close to the 7.3%

arXiv:1710.09489v3 [quant-ph] 14 May 2018

reported in [14] which uses a renormalization approach. When the syndrome measurements are noisy, our decoder still performs better for larger lattice when error probability is low, see Figure 10 for the plot. More numerical simulation needs to be done to determine the threshold.

This paper is structured as follows. In section 2 we discuss previous work in which machine learning was applied to decode quantum codes. In section 3 we give a review of the 4D toric code and its properties. In section 4, we provide a general introduction to machine learning. In section 5 we describe the application of convolutional neural networks as a subroutine in our decoding algorithm. In section 6 we discuss the results of Monte-Carlo simulations. In Appendix A, we give a short introduction to the backpropagation algorithm which we use to train the neural network. In Appendix D, we give a toy example which shows how a baseline machine learning model can fail at a task similar to decoding when the system size becomes large. This highlights the importance of choosing a translational invariant model as we do. In Appendix E, we observed and analyzed one property of multi-layer convolutional neural networks, and show it fits nicely to the task of assigning likelihood of qubit errors for high dimensional toric code.

## 2 Previous Work

It is known for a while in the classical error correction community that the errors and syndromes of a parity checking code can be put together into a probabilistic graphical model, which is a major tool for machine learning. Probabilistic graphical models include for example Bayesian networks and the Ising model (see [39] for an introduction to probabilistic graphical model and connection to error correcting codes). In [15] (and references therein), it is shown that by using belief propagation, the decoder can achieve very good performance for LDPC and turbo code that are close to the Shannon limit. We want to note that while there is no formal connection (to our knowledge) between neural networks and graphical models, a graphical model of a problem is often helpful for designing the architecture of the neural network. For example, in [25] the structure of the Tanner graph is used to construct a neural network, in order to improve belief propagation for (classical)

codes that have high density checks.

In [29], the authors discussed applying belief propagation to decoding of quantum stabilizer codes. While stabilizer codes and classical parity checking codes share many similarities, the authors highlighted several issues that might cause belief propagation to work much worse for stabilizer codes (even with low density checks). In [12, 13], the authors used belief propagation as a subroutine in the renormalization decoder for topological codes, with the purpose of synchronizing marginal probabilities across neighbouring cells.

Neural network decoder for quantum stabilizer codes has been studied in [3, 21, 36, 37] (more precisely in [36] hidden Boltzmann machines are used instead of neural networks). Our work will use a similar scheme as [21, 36], where the neural networks predict which qubits have undergone an error. In [3, 37], the neural networks output 1 or 2 bits which will correct the final measurement of the logical operators. The main difference of our work is that our machine learning decoder is naturally scalable through the use of convolutional neural networks.

## 3 The 4D Toric Code

### 3.1 Definition

The *4D toric code* is a stabilizer quantum code defined on a four-dimensional (4D) hypercubic lattice which was first considered in [11]. We will consider periodic boundary conditions so that topologically the lattice is a 4D torus. The faces of the lattice are squares and they are identified with the qubits. From now on we will use the words *face* and *qubit* interchangeably. The number of faces in a lattice of side-length  $L$  is  $\binom{4}{2}L^4 = 6L^4$ . This follows from the fact that every face is uniquely determined by a vertex  $v$  and two coordinates  $i, j \in \{x, y, z, w\}$ , so that the face with base-point  $v$  lies in the  $i$ - $j$ -plane. More generally, the number of  $k$ -dimensional objects in the lattice (1-dimensional objects would be edges or 3-dimensional objects would be cubes) is given by  $\binom{4}{k}L^4$ . There are stabilizer checks for every edge and cube in the lattice. A stabilizer check associated with a particular edge acts as Pauli- $X$  on all faces which are incident to it (see Figure 1 (left)), whereas a stabilizer check associated with a particular cube acts as Pauli- $Z$  on all

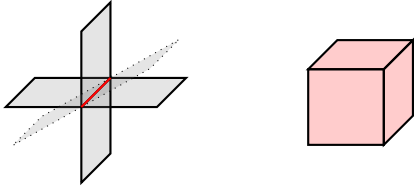


Figure 1: The stabilizer checks of the 4D toric code correspond to edges which act as Pauli- $X$  on all qubits incident to an edge (left) and cubes which act as Pauli- $Z$  on all qubits incident to an edge (right).

faces incident to this particular cube (see Figure 1 (right)). All stabilizer checks act on 6 qubits and each qubit is acted upon by 8 stabilizer checks.

Consider an operator  $E$  acting as Pauli- $Z$  on some subset of qubits. For  $E$  to commute with an  $X$ -check their overlap has to be even. Hence, to commute with all  $X$ -checks  $E$  can not contain any connected components that have a boundary where a single face is incident to an edge. Geometrically this means that  $E$  has to be a collection of boundaryless (closed) surfaces. If  $E$  itself is the boundary of a 3D volume inside the hypercubic lattice then it is the product of all  $Z$ -stabilizer elements inside this volume. But there are other examples of surfaces which do not have a boundary: Since the topology of the lattice is non-trivial we can consider a sheet extending over the whole  $x$ - $y$ -plane. Due to the periodicity of the lattice this sheet has no boundary, but is not itself the boundary of any 3D volume and thus it is not the product of any  $Z$ -stabilizers. It is therefore a logical operator. There is one such operator for every plane in the lattice. Each plane is labelled by two coordinates ( $x$ - $y$ -plane,  $x$ - $z$ -plane, ...) so that the 4D toric code encodes  $\binom{4}{2} = 6$  logical qubits. A sheet extending through the whole lattice consists of at least  $L^2$  faces which means that the 4D toric code has distance growing quadratically with the number of physical qubits. The parameters of the code are  $[[n = 6L^4, k = 6, d = L^2]]$ .

In [33] the authors argue that the threshold of the 4D toric code against independent bit-flip and phase-flip errors is approximately  $p_c \approx 0.11$  under minimum-weight decoding. This is the same threshold as for the 2D toric code and the surface code.

### 3.2 Syndromes

A feature of the 4D toric code is that the set of violated stabilizer checks form extended objects: since we pick up the boundary of an error (which is a collection of surfaces), the syndrome will always form closed loops as opposed to a set of points for the 2D toric code or surface code. A minimum-weight decoding, similar to minimum-weight perfect matching for the 2D toric code, is finding a minimum-area surface among all surfaces that have the syndrome as its boundary. It is known that the problem of finding such a minimum-area surface can be solved efficiently when the lattice is three-dimensional [32], but it is open whether it can be solved efficiently for a 4D lattice. However, there are several decoders with worse error-correction performance than the minimum weight decoder but which are computationally efficient [1, 7, 11, 18, 28]. The common way these decoders operate is by iteratively shortening the syndrome loops by flipping nearby faces.

The fact that the syndromes form closed loops can be understood in terms of local linear dependencies between the stabilizer elements: Taking all  $X$ -checks (edges) which are incident to a particular vertex gives the identity operator. This is the case because for every triple of vertex, edge and face which are pairwise incident to one another we can always find one more edge which is incident to the same vertex and face (see Figure 2). Similarly, taking all  $Z$ -checks (cubes) of one hypercube gives the identity since there are two cubes are overlapping on every face of the hypercube.

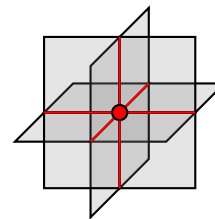


Figure 2: The dependency of the edge stabilizers which are incident to a common vertex. Taking the product of all edge-stabilizers (red) incident to a common vertex (red) gives the identity.

Each of those local linear dependencies can be interpreted as a (classical) parity check on the syndrome. More explicitly: Both the  $X$ -

syndrome and the  $Z$ -syndrome are encoded by a classical linear code. The code words of this code are the *valid syndromes*. What are the parameters of this code? - There is a bit for every  $X$ -check (edge) in the lattice, so the block size of the code is  $4L^4$ . There is a local dependency for every vertex. However, the local dependencies themselves are not independent. Taking the product over all vertices and all edge-checks incident to this vertex gives the identity since every edge is incident to two vertices. Hence, the number of independent checks in the classical linear code encoding the syndrome information is  $L^4 - 1$ . The encoded syndrome information therefore contains  $4L^4 - (L^4 - 1) = 3L^4 + 1$  bits. The distance of the classical code is 4 since adding the boundary of a face takes us from one valid syndrome to another valid syndrome.

Since the syndrome of the 4D toric code is encoded it has some build-in robustness against syndrome errors. In comparison, it is known that the 2D toric code does not have a decoding threshold in the presence of syndrome noise. The parity measurements have to be repeated and the record of the repeated syndrome measurements is decoded. This essentially implements a repetition code in time. Repeating measurements is not necessary for the 4D toric code to have increased error suppression capabilities with larger system size. The fact that measurements are not repeated is referred to as *single-shot measurements*. It has been shown analytically in [5] that a single-shot decoder can have a threshold and single-shot decoders for the 4D toric code have analyzed numerically in [2, 7, 14].

As 4D space is hard to visualize it is useful to consider the *3D toric code* to gain some geometric intuition. It is defined on a 3D cubic lattice with periodic boundaries. Similarly to the 4D toric code the qubits are identified with the faces of the lattice and the  $X$ -stabilizer checks act on all faces incident to an edge (weight 4) while the  $Z$ -stabilizer checks act on all faces incident to a cube (weight 6). A boundaryless sheet acting as Pauli- $Z$  commutes with all  $X$ -stabilizer checks just as for the 4D toric code. The  $X$ -stabilizer checks also satisfy the local linear dependencies shown in Figure 2. To commute with all  $Z$ -stabilizer checks it suffices to take a closed loop of Pauli- $X$  operators in the dual lattice. The existence of these string-like logical operators re-

sults in a threshold which is generally lower under minimum-weight decoding [33, 38].

## 4 Basics of Machine Learning

To have a better understanding of our decoding procedure, it is useful to first give a small overview of machine learning. This will provide insights to advantages and limitations of our procedure, and possible ways to improve it. For more extensive review on machine learning and in particular neural networks, we refer to [17, 22, 26].

Machine learning is often categorized into supervised learning, unsupervised learning and reinforcement learning, each has a different focus. However, all of them usually involve the use of some models. In this paper, the word “model” simply means a family of functions  $f_\alpha$  or subroutines  $s_\alpha$ , where the (most likely multidimensional) parameter  $\alpha$  needs to be trained by certain learning algorithms. Below we give a not very comprehensive introduction to these three kinds of learning

- *Supervised learning* is to find  $\alpha$  such that  $f_\alpha = f_{\text{hidden}}$ , where  $f_{\text{hidden}}$  is given implicitly by a dataset of input-output pairs  $(x_i, y_i = f_{\text{hidden}}(x_i))$ .
- *Unsupervised learning* refers to many different tasks which are related to finding structures in a dataset. In contrary to supervised learning, there is no desired output for each entry in the dataset. For example, considering tasks in error correction, inferring properties of the error model from a dataset of syndromes or using the dataset to generate new similar syndromes can be viewed as unsupervised learning, as well as some tasks studied in [9].
- *Reinforcement learning* is concerned with how agents ought to take actions in an environment so as to maximize some notion of cumulative reward. However, in this paper we will use the word “reinforcement learning” to denote the optimization of a subroutine  $s_\alpha$  of a program, so that at the end of the program a predefined score is maximized.

For complicated tasks, it is beneficial to combine these approaches. A notable example which



highlights this is the recent success of the AI *AlphaGo* [30, 31]. We give a short summary of this combined approach in Appendix C and discuss some ideas for applying it to the decoding problem of quantum codes.

One major difficulty encountered in machine learning is the overwhelmingly large input/state space. Here we will only give a short explanation of how this difficulty arises in supervised learning. Recall that we are given a dataset  $(x_i, y_i)$  with the goal being approximating  $f_{\text{hidden}}$ . If the input space is large, then for a random  $x$ , it is unlikely that  $x$  is “similar” to any of the  $x_i$  from the dataset. Therefore, it is hard to guess the value of  $f_{\text{hidden}}(x)$  and approximate it by  $f_{\alpha}(x)$ . This is why often the most important part of a machine learning project is to choose a suitable model, usually by encoding the prior knowledge of the task into it. The model we use in this paper is neural networks which we introduce in the following section.

#### 4.1 Basics of Neural Networks

A *neural network* is a directed, multipartite graph consisting of *layers*  $l = 0, \dots, L$ . The vertices in layer  $l$  are connected to vertices in the following layer  $l + 1$ .

The vertices of the network are called *neurons*. The main idea behind the neural network is the following: Each neuron computes a primitive non-linear function  $f_{w,b} : \mathbb{R}^q \rightarrow \mathbb{R}$ , for which the input values are given by neurons of the previous layer connected to it. The subscripts  $w \in \mathbb{R}^q$  and  $b \in \mathbb{R}$  are parameters which can differ for each neuron. Before we discuss the function  $f_{w,b}$  in more detail let us first understand how the network performs a computation. The neurons in the first layer do not have any predecessors and their output is simply set to be the input of the network, which is a bit string  $x \in \{0, 1\}^m$ . The values of the neurons in the last layer are interpreted as the output of the network. We see that the network describes a function  $F : \{0, 1\}^m \rightarrow [0, 1]^n$ . The first layer  $l = 0$  is called the *input layer* and the last layer  $l = L$  is called the *output layer*. All other layers  $l = 1, \dots, L-1$  are called *hidden layers* since they are considered to be internal to the network.

The parameters  $w$  and  $b$  are called *weights* and *biases*. They define a linear map  $w \cdot y + b$  where  $y$  is the input of the neuron. The function that

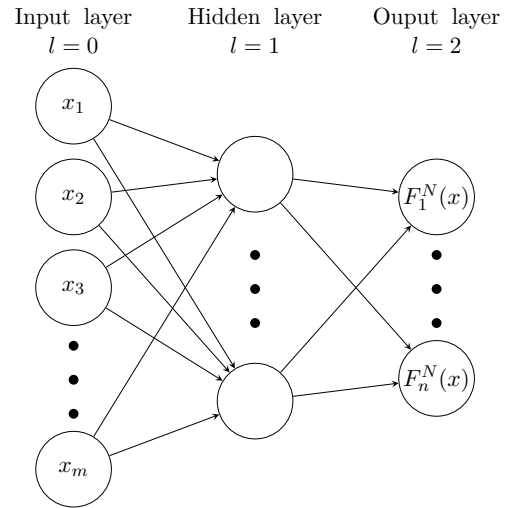


Figure 3: A neural network consisting of 3 layers. The network takes input  $x \in \{0, 1\}^m$  which is represented as the first layer of neurons. The values of neurons in the hidden layer and the output layer are given by the function  $f_{w,b} : \mathbb{R}^q \rightarrow [0, 1]$  evaluated on their input (indicated by  $q$  incoming arrows). The parameters  $w \in \mathbb{R}^q$  and  $b \in \mathbb{R}$ , called weights and bias, can be different for each neuron. The values of the neurons in the last layer are the output of the network  $F^N(x) \in \mathbb{R}^n$ .

each neuron computes has the form:

$$f_{w,b} : \mathbb{R}^q \rightarrow [0, 1], \quad y \mapsto \sigma(w \cdot y + b) \quad (1)$$

where in this paper  $\sigma$  is either tanh or the *sigmoid function*

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \quad (2)$$

which is plotted in Figure 4. Both non-linear functions can be thought of as a smoothed step function. The smoothness of  $\sigma$  is important to the training of the neural networks as it is based on optimizing certain objective function with gradient descent.

Let us have a look at a small example which gives some intuition why neural networks are able to perform interesting computations: Consider a single neuron which takes  $q = 2$  inputs and has weights  $w = (-12, -12)$  and bias  $b = 17$ . The neuron computes the following values for each input:

$$\begin{aligned} f(0, 0) &= \sigma(17) \approx 1 \\ f(1, 0) &= \sigma(5) \approx 1 \\ f(0, 1) &= \sigma(5) \approx 1 \\ f(1, 1) &= \sigma(-7) \approx 0. \end{aligned} \quad (3)$$

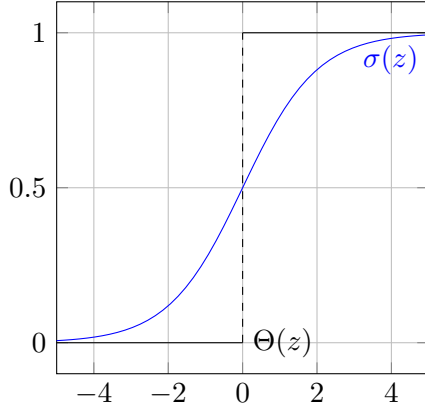


Figure 4: The sigmoid function  $\sigma$  is a smooth version of the Heaviside step function  $\Theta$ .

We observe that these are approximately the input/output relations of the NAND gate. The approximation can be made arbitrarily close by increasing the absolute values of the weights  $w$  and the bias  $b$ . Since any Boolean function  $F : \{0, 1\}^m \rightarrow \{0, 1\}^n$  can be computed by a network of NAND gates, there consequently also exists a representation of  $F$  as a neural network. However, in practice it is more efficient to adapt the connectivity of the network to the problem at hand.

The process of *training* the network, i.e. gradually adapting the values of the weights  $w$  and biases  $b$  to make the network a desired function, is described in [Appendix A](#).

## 4.2 Convolutional Neural Networks

Compared to fully connected neural networks, convolutional neural networks (CNN) require much fewer parameters to describe. We will start with the definition of one convolutional layer. The input to the layer resides on a  $D$ -dimensional lattice of size  $L^D$ . On each lattice site, there is a  $d$ -dimensional vector  $x_{\mathbf{u}} \in \mathbb{R}^d$ , where the subscript  $\mathbf{u} \in \mathbb{Z}_L^D$  (we use  $\mathbb{Z}_L$  to denote integer in range  $[0, L - 1]$ ). We define the kernel to be a vector  $K_{\mathbf{u}, i}$ , where  $\mathbf{u} \in \mathbb{Z}_n^D$  and  $i \in \mathbb{Z}_d$ . With a slight abuse of notation, we will say such a kernel has size  $n^D$ . The convolution is then

$$y_{\mathbf{v}} = \sum_{\mathbf{u} \in \mathbb{Z}_n^D} \sum_{i \in \mathbb{Z}_d} x_{v-\mathbf{u}, i} K_{\mathbf{u}, i}, \quad (4)$$

where  $x_{v-\mathbf{u}, i}$  is the  $i$ th element of  $x_{v-\mathbf{u}}$ , and each element of  $\mathbf{v}$  has range  $[n - 1, L - 1]$ . The index  $v$

of the output  $y$  can be shifted according to needs (e.g. to  $\mathbb{Z}_{L-n+1}^D$ ). In this paper, we solely work with input that has the periodic boundary condition. Thus, we can indeed treat the indices  $\mathbf{u}$  of input  $x_{\mathbf{u}}$  as elements of the cyclic group  $\mathbb{Z}_L^D$ . The convolution given in [Equation 4](#) can then be modified accordingly such that  $\mathbf{v} \in \mathbb{Z}_L^D$ . If there are  $r$  different kernels, we can apply [Equation 4](#) for each kernel individually and obtain  $y_{\mathbf{v}, i}$ , where  $i \in \mathbb{Z}_r$ . We will say the resulted output  $y_{\mathbf{v}, i}$  has  $r$  channels, and further convolution can be again applied on  $y_{\mathbf{v}, i}$  according to [Equation 4](#). Non-linear functions are usually applied after each convolution. In this paper, convolutional neural networks are neural networks with only convolutional layers (where in computer vision, the convolutional neural networks usually contains coarse-grain layers, etc).

The connectivity of a 1D convolutional neural networks is illustrated in [Figure 13](#). We can see that a convolutional layer is simply a neural network layer with local connectivity and the weights are translationally invariant.

## 5 Machine Learning for Decoding Quantum Codes

Decoding quantum error correcting codes can be considered both as a supervised and reinforcement learning problem.

**Supervised learning:** In most scenarios, we can generate pairs of (syndrome, correct decoding), either by simulation on classical computer or data from real experiments. Thus, we can use the pairs to train suitable models.

**Reinforcement learning:** Training certain types of decoders is more naturally considered as reinforcement learning problem. For example, we can leave some freedom in the cellular automaton decoder to be trained, with the goal of optimizing the memory time. In this setting, there is not a clear correct answer at each time step, and therefore we cannot directly train the cellular automaton with a input-output relation.

One major difficulty of applying machine learning to decoding is the variable lattice size (or in

general, variable code size of some code family). Note that usually learning algorithms run on one code size at a time, and it is fair to assume that the training time will grow when we increase the code size, possibly exponentially. Thus, it is not trivial to train a machine learning decoder on a larger lattice to utilize the error suppression ability provided by topological codes. Moreover, if we restrict the machine learning models to neural networks, it is tempting to directly train a fully-connected neural nets with the pair (syndrome, correct decoding). While this is theoretically possible, in practice it is very hard to achieve for large lattices, due to the following:

1. Decoding quantum error correcting codes is a quite complicated task. In order for the fully-connected neural nets to approximate the input-output relation, they need to have large amount of parameters (either by having many layers or many neurons in each layer). This might already require too many computational resources to set up the nets or train them.
2. A neural net with large amount of parameters in turn requires large amount of training data, in order to not overfit.

See [Appendix D](#) for a relevant discussion on obstacle of applying a baseline machine learning algorithm to decoding problem on large lattice.

## 5.1 Architecture and Training

In this subsection, we will only concern about Pauli- $X$  stabilizer checks. We can pack the  $6L^4$  qubits into a multi-dimensional array of shape  $(L, L, L, L, 6)$ , and the Pauli- $X$  stabilizer checks into one of shape  $(L, L, L, L, 4)$ . We choose to use the following architecture for decoding:

**Architecture:** Given an  $X$ -syndrome with shape  $(L, L, L, L, 4)$ , a convolutional neural net (CNN) is applied, with an output of shape  $(L, L, L, L, 6)$ . Pauli- $X$  is applied to one or multiple qubits according to the output, and the error syndrome is updated accordingly. This process is repeated until no syndrome is left or certain condition for halting is met. For the particular training process we will describe shortly after, we choose to flip qubits corresponding to the largest values in the outputs from the CNN.

**Training:** It is not clear how the CNN should decide which qubits to flip, thus a natural approach would be optimizing the CNN to achieve the best memory time. However, we decide to explicitly train the CNN to estimate whether a bit-flip has occurred on each qubit. While not optimal, it allows a much faster training process and it is less prone to finite size effect potentially caused by reinforcement learning on a small lattice.

This architecture has a few desired properties. First, the convolutional neural net is a constant-depth (with respect to  $L$ ) and translational-invariant local circuit. As a result, each element of the  $(L, L, L, L, 6)$  output is determined by its surrounding region in the same way. Note that this also means that the training only needs to be done on a constant size region, which is a huge advantage for high dimensional lattices as the computational resource needed grows very fast with respect to  $L$ . Once we trained the CNN on a lattice with size  $L$ , we can naturally extend it to a larger size  $L' > L$ , therefore we can evaluate the performance of a fixed CNN on several different size  $L$ . This is crucial for discussing the error suppression ability by increasing the system size, or performing an estimation of the decoding threshold. We shall point out that the architecture together with training procedure is designed with 4D toric code in mind, where multiple local decoders have already shown a threshold behavior.

To some degree, the aim of our neural network decoder is to develop a local decoding strategy such as those analyzed in [7]. In order to see this, we will give a brief introduction to the DKLP rule and the Toom's rule. For the DKLP rule, a qubit is flipped if 3 or 4 adjacent edge stabilizer checks are violated, and is flipped with probability  $1/2$  if only two are violated. This update rule should not apply simultaneously to faces that share an edge. Toom's rule is defined for a 2D lattice, where each face is associated with a degree of freedom which can take the values  $+1$  and  $-1$ . In each step of the Toom's rule, the value of a face will be flipped if it is different from both its 'north' and 'east' faces' values. It can be applied to the 4D toric code by applying it to every 2D plane of the 4D hypercubic lattice (e.g. applying to all  $x$ - $y$ -planes and then applying to all  $x$ - $z$ -planes, etc). It is not hard to see that for

both rules, the criteria which decides whether a qubit should be flipped or not can be written as a neural network. After all, the neural networks can approximate any function on a finite input spaces. Intuitively, the DKLP rule tries to flip qubits that likely have an error on them, which our neural network decoder also intends to achieve due to the specific training procedure. On the other hand, Toom’s rule is not designed with this goal in mind. Therefore, it will not be obtained by our training procedure. This serves as a reminder that some good decoders require a different training procedure to find, e.g. some kind of reinforcement learning. It is worth noting that the schedule components of these rules (i.e. applying Toom’s rule to 2D planes sequentially and applying DKLP rule to non-adjacent faces) are not contained in the architecture of our decoder. If the goal is to accurately simulate the DKLP and Toom’s rules, we can introduce a clock variable as an additional input on each face.

One detail of our architecture is that, as we mentioned in the architecture paragraph, we flip qubits corresponding to the largest values in the outputs from the CNN. In order to reduce the computational time of decoding, we can flip multiple qubits based on a single evaluation of the CNN. In the numerical simulation described in [section 6](#), the number of qubits to be flipped is chosen to be  $\#(\text{current violated syndrome checks})/x$ , where  $x$  is around 50. We did not study the effect of different choice of  $x$ . It can be viewed as a tunable parameter that requires further optimization. Another reasonable way is to flip the qubits whose corresponding values from the CNN’s output are larger than a pre-determined threshold. If this approach is used, the decoder will be completely local with respect to the 4D lattice.

We also add an additional step in the architecture when decoding syndromes from perfect measurements of check operators, which is to apply the “parallel line decoder” introduced in [Appendix B](#). The motivation comes from the observation that without the parallel line decoder, a large percentage of the failures are resulted from the decoder getting stuck at syndromes that only contains a few pairs of parallel lines (e.g. [Figure 11](#)). Failures of this type are called “energy-barrier limited decoding” in [7]. As we do not

focus on building a completely local decoder, we choose to use this simple and fast subroutine to improve the performance.

## 6 Numerical Analysis of the Neural Network Decoder

### 6.1 Error Model and Setup

We perform a numerical analysis of the neural network decoder by Monte Carlo simulations. We consider two error models: (a) In the first error model we assume that the measurements of the check operators can be done perfectly. We consider the independent  $X$ - $Z$ -error model in which a Pauli- $X$  and a Pauli- $Z$  are applied independently to each qubit with probability  $p$ . (b) In the second error model we take measurement errors into account. The errors on the qubits are modeled as in (a). Additionally each outcome of a syndrome measurement is flipped with probability  $q$ . The error correction procedure of the Pauli- $X$  errors and the Pauli- $Z$  errors can be done independently as each parity-check can only detect either type and all parity-checks are pairwise commuting.

For error model (a) where measurements are perfect we numerically estimate the *rate of logical errors*  $\bar{P}$  depending on the physical error rate  $p$ . The rate of logical errors  $\bar{P}$  is the probability of the neural network decoder to apply a non-trivial logical operator (or exceeding a time limit). We can estimate  $\bar{P}$  for a fixed  $p$  as follows: First, we sample an error  $E$  by applying a Pauli- $X$  or Pauli- $Z$  each with probability  $p$ . We then compute the result of the syndrome check measurements  $s$  and give it to the neural network decoder which determines a recovery operator  $R$ . After the application of  $R$  we are back in a code state. The decoder was successful if the application of  $E$  and  $R$  acts trivially on the code state. This can be checked efficiently since any operator which leaves the code space as a whole invariant acts non-trivially within the code space if and only if it commutes with all other logical operators.

For error model (b), which takes syndrome errors into account, the neural network decoder can in general not correct back to a code state since the decoder has access to the erroneous measurement result only. To obtain a threshold we estimate the *memory time*  $T$  which gives the av-



erage number of error correction rounds until a logical failure occurs. If the physical error rate is below threshold the memory time will diverge with increasing lattice size  $L$ . More concretely, in each error correction round, we first sample an error  $E$  by applying a Pauli- $X$  or Pauli- $Z$  each with probability  $p$  and update the (noiseless) syndrome check measurement  $s$  from the last round. We then flip each bit in  $s$  with probability  $q$  to obtain the faulty syndrome check measurement  $s'_{\text{in}}$ , and feed  $s'_{\text{in}}$  to the neural network decoder. The decoder will output a list of position where Pauli- $X$  or Pauli- $Z$  are applied, so we can keep track of the noiseless syndrome measurement  $s_{\text{out}}$  afterwards. To evaluate whether a logical failure has occurred, we input  $s_{\text{out}}$  to the decoder we use in the error model (a). Note that we do not have to repeat the syndrome measurements as for the 2D toric code.

The networks we consider consist of a input layer which receives the result of the parity check measurements, one convolutional layer with kernel size  $3^4$  (i.e.  $n = 3$  in Equation 4) and then two convolutional layer with kernel size  $1^4$ . The number of channels in the hidden layer is 15 for the noiseless syndrome measurement, and 20 for the noisy case. The choices of these numbers here are mostly aimed for a large neural network without getting too slow or require too much memory to train. After the linear map of the final convolutional layer, a softmax function is applied on all inputs:

$$x_i \rightarrow \frac{e^{x_i}}{\sum_i e^{x_i}} \quad (5)$$

It is a widely used in the scenario when we want the output to be a probability distribution (i.e. non-negative numbers that sum to 1). However, there is no obvious reason that the softmax layer is needed in our neural networks, as during the decoding process we only care about the relative order of the output numbers, which the softmax layer preserves. The rational behind this is that among the limited number of neural networks we trained, the ones with the softmax layer performs slightly better. Note that the softmax layer breaks the locality of the networks. Nevertheless, if preferred, we can view softmax together with the cross-entropy as the modified cost function, while the neural networks remains local.

A 1D slice of the neural network connectivity is shown in Figure 5. It is obvious that without the

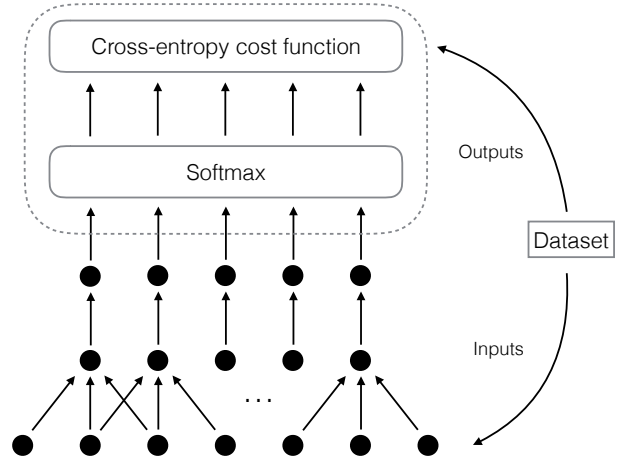


Figure 5: A 1D slice of the decoding neural network. The neurons in the first hidden layers depend on the neighbouring sites of the input layer, while each neuron in the later layers only depend on the neurons in the same sites in the previous layer. The input from the dataset is the error syndrome. The softmax layer does introduce non-locality in the network. However, we can effectively view softmax together with the cross-entropy as the modified cost function.

softmax layer, each element in the output is determined only by a corresponding  $3^4$  local region on the input. The reason that we only have one convolutional layer with kernel size  $3^4$  is mainly due to the faster runtime and likely will cause less finite-size effect on the lattice sizes we are testing on. See Appendix E for more discussion on using deeper neural networks.

## 6.2 Training

We trained two neural networks for error model (a) and (b) described in section 6.1 respectively. However, the same neural network will be used for decoding syndromes generated according to different error probabilities. Roughly speaking, the training of the neural networks is done using gradient descent (see section 4.1). The inputs are error syndromes of shape  $(L, L, L, L, 4)$ . The outputs have shape  $(L, L, L, L, 6)$ , where an element is equal to  $1/N$  if an error happened on the corresponding qubit and equal to 0 otherwise, with  $N$  being the total number of errors. The normalization is done to match the normalization done by the softmax layer. We use a variation of the gradient descent algorithm called ADAM [20], which is included in the TENSORFLOW library. The cost function we use is cross-entropy. We also manually lower the learning rate of ADAM when the

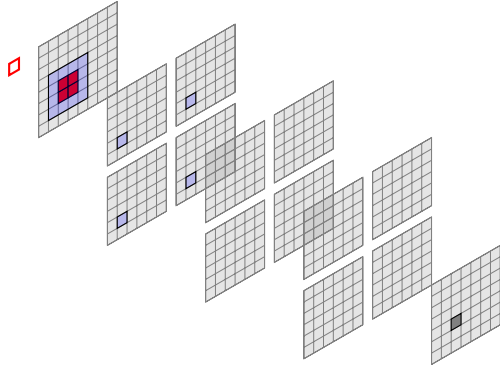
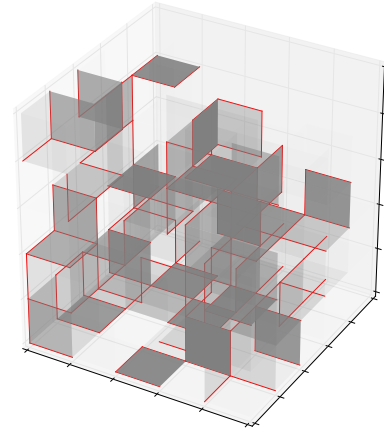


Figure 6: Illustration of the neural network. Each array of neurons is shown as a 2D square grid but in fact has the same dimensionality as the lattice (3D or 4D). The network consists of a single input layer (leftmost array) which receives the measurement results. The input layer is followed by three hidden layers. The number of channels in each hidden layer is 4 in this illustration. The final layer returns the probability distribution as output. A single convolution between the input layer and the first hidden layer is indicated in blue.

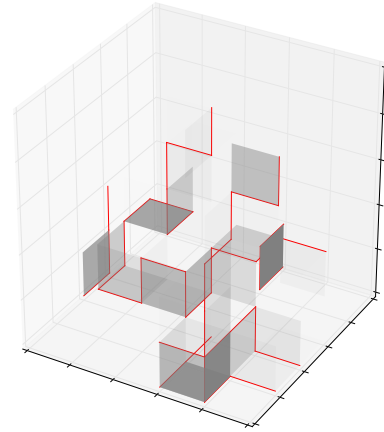
decrease of cost function slows down. For error model (a), we train the network with syndromes corresponding to error rate  $p$  uniformly distributed from 3% to 7%. For error model (b), we train the network with error rate  $p$  uniformly distributed from 2% to 3%, and  $q$  being a constant 2.5%. These values were determined to be the approximate locations of the thresholds in trial runs.

We also applied the neural network decoder to the 3D toric code under error model (a). The convolutions chosen to be three dimensional in this case, but the structure of the neural network is identical otherwise. The network is trained for  $p$  at 17% which is again just below the value of the threshold.

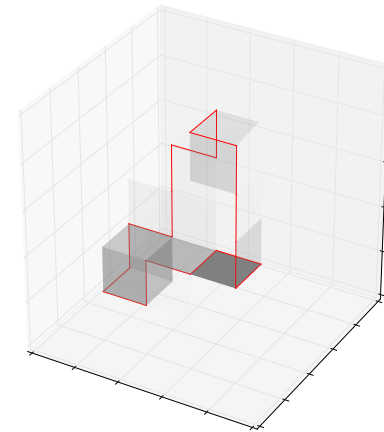
We do want to note that the details of the training are not very important (e.g. the variation of gradient descent we use, the concrete parameters we used for gradient descent, etc). Indeed, the neural nets in this paper are fairly shallow compared to the neural networks used by the machine learning community at the present time. Based on experience, any refined gradient optimizer should be able to train the networks decently well without fine-tuning the training parameters. Additionally, we almost did not do any post-selection on the training of neural networks, and in general the decoder works reasonably well



(a) 0 steps



(b) 15 steps



(c) 30 steps

Figure 7: Applying the neural network decoder to the 3D toric code with  $L = 5$ . The syndrome is highlighted in red. The neural network outputs a probability distribution over the faces indicating where it believes an error to be present. The probability of each face is indicated by its opaqueness. Each figure shows the current syndrome and the output of the network during the decoding. In each step the decoder flips the face with the highest probability.

with a trained neural network. We believe the performance of decoder observed in this paper is not a rare event.

### 6.3 Performance

To evaluate the performance, we follow the procedures described in [section 6.1](#). We will first discuss the results for error model (a) where we assume perfect stabilizer measurements. As we mentioned in [section 5.1](#), the parallel line decoder is applied after the neural network decoder when obtaining these results. In [Figure 8](#) and [Figure 9](#), we plot the logical error rates versus the physical error rates and compare them with the minimum-weight decoder. The minimum-weight decoder is implemented by mapping the problem of finding a minimum surface with the syndrome as boundary to a linear integer programming problem.

We assume a scaling behavior in the variable

$$x = (p - p_c)L^{1/\nu} \quad (6)$$

to determine the critical error probability  $p_c$ . We expand the logical error probability  $\bar{P}$  for small  $x$  (around  $p = p_c$  where the dependence on the system size  $L$  is small):

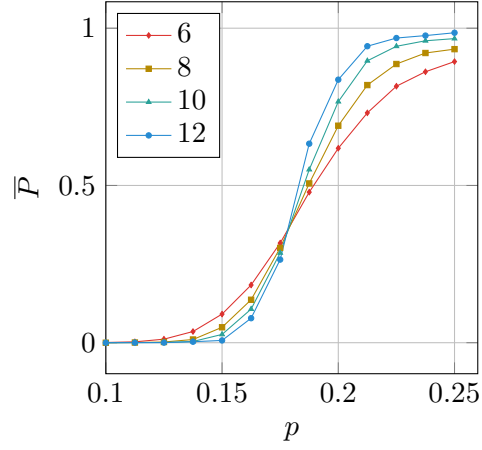
$$\bar{P}(p, L) = A + Bx + Cx^2 \quad (7)$$

For the 4D toric code we obtain by fitting [Equation 7](#) to the data for  $p = 0.066, 0.068, 0.072, 0.074$  (see [Figure 9](#)). The fitting parameters  $p_c$  and  $\nu$  were determined by a non-linear fit:

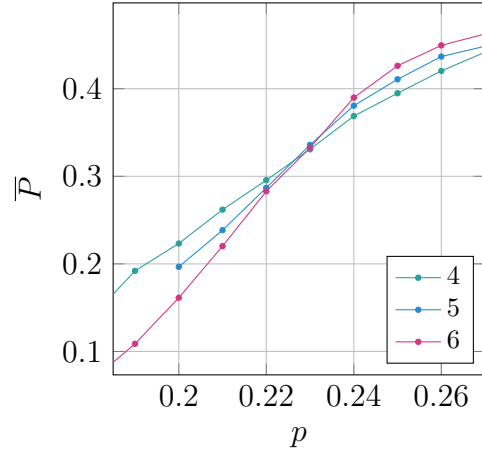
$$p_c = 0.071 \pm 0.003, \quad \nu = 0.65 \pm 0.02 \quad (8)$$

This is comparable to the performance of the 4D renormalization group decoder described in [\[14\]](#) which achieves a threshold of  $p_c = 0.073 \pm 0.001$  for the same error model.

Below we will discuss error model (b) with measurement errors. Here we will always set  $p = q$ , where  $p$  is the error rate of the physical qubits, and  $q$  is the error rate of each stabilizer parity measurement. For clarification, we do not use the parallel line decoder for this error model. In [Figure 10](#), we plot the average memory time  $T$  versus the error rate  $p$ . Due to a time restriction on the computing cluster some runs were forced to halt where the simulated system was still in a correctable state. In the worst case (large system

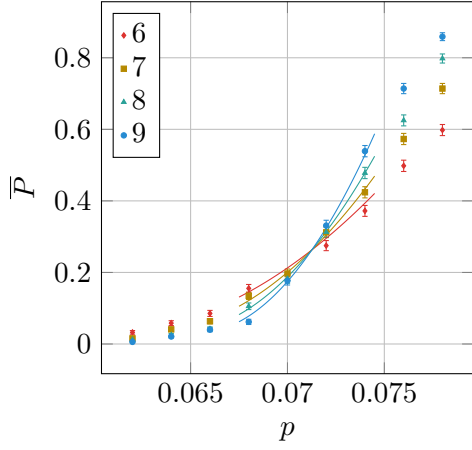


(a) Neural network decoder

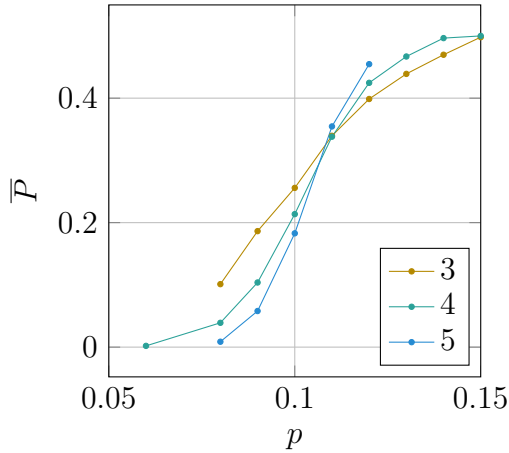


(b) Minimum-weight decoder

Figure 8: (a) The results of the numerical simulation for the 3D toric code for  $Z$ -errors only, assuming perfect measurements. We considered system sizes  $L = 6, 8, 10, 12$ . The lines cross at around 17.5%. (b) Results for the minimum-weight decoder which has exponential run-time. The lines cross at around 23%. Note that the threshold of the line-like logical operator will be significantly lower (see [\[33\]](#)).



(a) Neural network decoder



(b) Minimum-weight decoder

Figure 9: (a) The results of the numerical simulation for the 4D toric code assuming perfect measurements. We considered system sizes  $L = 5, 6, 7, 8$ . The solid lines are given by the values of Equation 7. (b) Numerical simulation for the minimum-weight decoder assuming perfect measurements. The lines cross at around 11% which is in agreement with the numerical results of [2].

size and low physical error rate) only around 10% of the runs finished. In order to obtain meaningful statistics from the data, we make the assumption that for a fixed  $L$  and  $p$ , the memory time follows an exponential distribution, and all unfinished runs have longer memory time than finished runs (which is called type II censoring on the right in the statistics literature). Under these assumption, the mean (and its confidence interval) can be estimated according to [10].

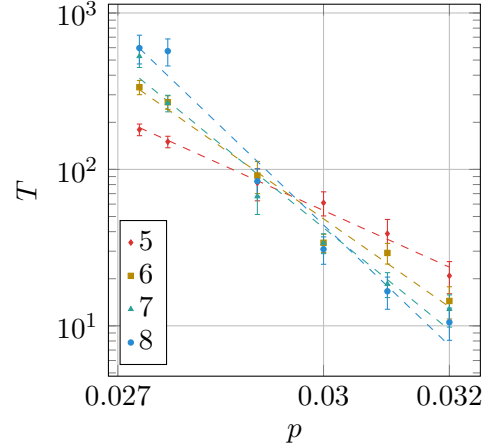


Figure 10: The results of the numerical simulation for the 4D toric code assuming noisy measurements. We considered system sizes  $L = 5, 6, 7, 8$ . The error bars show the 80% confidence interval. The dashed lines indicate a linear fit in the log-log-plot. The memory time is expected to decay exponentially with  $p$  and diverge exponentially with increasing  $L$  for any fixed  $p < p_c$ . The data does not allow a precise estimation of the threshold  $p_c$ .

## 7 Discussion

We have shown how convolutional neural networks can be utilized to decode higher dimensional topological quantum codes, leading to a scalable architecture. The neural network is trained once and can then be scaled up to arbitrary system sizes. However, our approach is only one way of utilizing neural networks to decode high-dimensional quantum codes. Given the versatility of neural networks, it is clear that there will be other approaches to the problem, which are worth exploring. As we mentioned in section 5, ideally we need to perform reinforcement learning (i.e. optimize the parameters of the neural networks with the objective being lowest logical error rate or longest memory time).



There is another important reason that we choose neural networks instead of other machine learning models for decoding. As discussed in [4], good learning algorithms should be efficient in terms of human involvement. For example, it is highly undesirable if small changes in the quantum error correcting code and the experimental hardware require a large amount of human effort to rewrite the learning algorithms. These environmental changes are certainly going to happen very frequently before a large-scale quantum computer is built. On the other hand, if a certain class of learning algorithms are fast to implement, then with the same amount of man-hour we can test more different algorithms for the problem. Therefore, less human involvement often translates to better performance. At the moment of writing, neural networks have some of the most flexible and automated software packages among machine learning models.

We would also like to highlight that using neural networks for decoding may have advantages from a practical perspective. Many specialized neural network chips have been manufactured [19, 23, 24] which promise much lower power consumption. If the decoding has to take place inside a fridge such dedicated hardware could help to keep down thermal noise. For example, in [23] the authors report on an integrated circuit implementing a network of 1 million neurons on a  $240\mu\text{m} \times 390\mu\text{m}$  CMOS chip. The chip draws  $20\text{mW}/\text{cm}^2$  as compared to  $50\text{W}/\text{cm}^2$ - $100\text{W}/\text{cm}^2$  for a modern CPU or  $30\text{W}/\text{cm}^2$  for an FPGA [34], thereby reducing potential thermal noise by orders of magnitude.

A natural question is whether we can build a similar convolutional neural network decoder for 2D toric code. As the architecture we proposed heavily based on the fact that 4D toric code can be decoded in a local single-shot manner, it cannot be directly applied to 2D toric code. However, we foresee that a decent convolutional neural network decoder for 2D toric code exists, and it will likely share many similarities to the renormalization group decoder in [13].

### Acknowledgments

We would like to thank Christophe Vuillot and Kasper Duivenvoorden for interesting discussions and Barbara Terhal for feedback on our manuscript.

### References

- [1] Charlene Sonja Ahn. *Extending quantum error correction: new continuous measurement protocols and improved fault-tolerant overhead*. PhD thesis, California Institute of Technology, 2004.
- [2] Gaku Arakawa, Ikuo Ichinose, Tetsuo Matsui, and Koujin Takeda. Self-duality and phase structure of the 4d random-plaquette  $z_2$  gauge model. *Nuclear Physics B*, 709(1): 296–306, 2005.
- [3] Paul Baireuther, Thomas E. O'Brien, Brian Tarasinski, and Carlo W. J. Beenakker. Machine-learning-assisted correction of correlated qubit errors in a topological code. *Quantum*, 2:48, jan 2018. DOI: [10.22331/q-2018-01-29-48](https://doi.org/10.22331/q-2018-01-29-48).
- [4] Yoshua Bengio, Yann LeCun, et al. Scaling learning algorithms towards ai. *Large-scale kernel machines*, 34(5):1–41, 2007.
- [5] Héctor Bombín. Single-shot fault-tolerant quantum error correction. *Physical Review X*, 5(3):031043, 2015. DOI: [10.1103/PhysRevX.5.031043](https://doi.org/10.1103/PhysRevX.5.031043).
- [6] Nikolas P. Breuckmann. *Homological quantum codes beyond the toric code*. PhD thesis, RWTH Aachen University, 2017. URL <https://doi.org/10.18154/rwth-2018-01100>.
- [7] Nikolas P Breuckmann, Kasper Duivenvoorden, Dominik Michels, and Barbara M Terhal. Local decoders for the 2d and 4d toric code. *Quantum Information and Computation*, 17(3 and 4):0181–0208, 2017. DOI: [10.26421/QIC17.3-4](https://doi.org/10.26421/QIC17.3-4).
- [8] Christopher Clark and Amos Storkey. Training deep convolutional neural networks to play go. In *International Conference on Machine Learning*, pages 1766–1774, 2015.
- [9] Joshua Combes, Christopher Ferrie, Chris Cesare, Markus Tiersch, GJ Milburn, Hans J Briegel, and Carlton M Caves. In-situ characterization of quantum devices with error correction. 2014. URL <https://arxiv.org/abs/1405.5656>.
- [10] H.A. David and H.N. Nagaraja. *Order Statistics*. Wiley Series in Probability and Statistics. Wiley, 2004. ISBN 9780471654018.
- [11] Eric Dennis, Alexei Kitaev, Andrew Landahl, and John Preskill. Topological quan-

- tum memory. *Journal of Mathematical Physics*, 43(9):4452–4505, 2002.
- [12] Guillaume Duclos-Cianci and David Poulin. Fast decoders for topological quantum codes. *Physical review letters*, 104(5):050504, 2010. DOI: [10.1103/PhysRevLett.104.050504](https://doi.org/10.1103/PhysRevLett.104.050504).
- [13] Guillaume Duclos-Cianci and David Poulin. Fault-tolerant renormalization group decoder for abelian topological codes. *Quantum Information & Computation*, 14(9-10):721–740, 2014.
- [14] Kasper Duivenvoorden, Nikolas P Breuckmann, and Barbara M Terhal. Renormalization group decoder for a four-dimensional toric code. *arXiv preprint arXiv:1708.09286*, 2017.
- [15] Brendan J Frey and David JC MacKay. A revolution: Belief propagation in graphs with cycles. *Advances in neural information processing systems*, pages 479–485, 1998.
- [16] Gabriel Goh. Why momentum really works. *Distill*, 2017. DOI: [10.23915/distill.00006](https://doi.org/10.23915/distill.00006).
- [17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
- [18] Matthew B Hastings. Decoding in hyperbolic spaces: Ldpc codes with linear rate and efficient error correction. *Quantum Information and Computation*, 14, 2014.
- [19] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. *44th International Symposium on Computer Architecture*, 2017. DOI: [10.1145/3079856.3080246](https://doi.org/10.1145/3079856.3080246).
- [20] 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>.
- [21] Stefan Krastanov and Liang Jiang. Deep neural network probabilistic decoder for stabilizer codes. *Scientific Reports*, 7(1), sep 2017. DOI: [10.1038/s41598-017-11266-1](https://doi.org/10.1038/s41598-017-11266-1).
- [22] Stephen Marsland. *Machine Learning: An Algorithmic Perspective, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014. ISBN 1466583282, 9781466583283.
- [23] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014. DOI: [10.1126/science.1254642](https://doi.org/10.1126/science.1254642).
- [24] Janardan Misra and Indranil Saha. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, 74(1–3):239 – 255, 2010. ISSN 0925-2312. DOI: [10.1016/j.neucom.2010.03.021](https://doi.org/10.1016/j.neucom.2010.03.021).
- [25] Eliya Nachmani, Yair Be'ery, and David Burshtein. Learning to decode linear codes using deep learning. In *2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, sep 2016. DOI: [10.1109/allerton.2016.7852251](https://doi.org/10.1109/allerton.2016.7852251).
- [26] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.
- [27] Genevieve B Orr and Klaus-Robert Müller. *Neural networks: tricks of the trade*. Springer, 2003.
- [28] Fernando Pastawski. *Quantum memory: design and applications*. PhD thesis, LMU Munich, 2012. URL <https://edoc.uni-muenchen.de/14703/>.
- [29] David Poulin and Yeojin Chung. On the iterative decoding of sparse quantum codes. *Quantum Information and Computation*, 8(10):0987–1000, 2008.
- [30] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. DOI: [10.1038/nature16961](https://doi.org/10.1038/nature16961).
- [31] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, Oct 2017. ISSN 0028-0836. DOI: [10.1038/nature24270](https://doi.org/10.1038/nature24270).
- [32] John M Sullivan. *A crystalline approxima-*

- tion theorem for hypersurfaces*. PhD thesis, Princeton University, 1990.
- [33] Koujin Takeda and Hidetoshi Nishimori. Self-dual random-plaquette gauge model and the quantum toric code. *Nuclear Physics B*, 686(3):377–396, 2004. ISSN 0550-3213. DOI: [10.1016/j.nuclphysb.2004.03.006](https://doi.org/10.1016/j.nuclphysb.2004.03.006).
- [34] David Barrie Thomas, Lee Howes, and Wayne Luk. A comparison of cpus, gpus, fp-gas, and massively parallel processor arrays for random number generation. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '09*, pages 63–72, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-410-2. DOI: [10.1145/1508128.1508139](https://doi.org/10.1145/1508128.1508139).
- [35] Yu Tomita and Krysta M Svore. Low-distance surface codes under realistic quantum noise. *Physical Review A*, 90(6):062320, 2014. DOI: [10.1103/PhysRevA.90.062320](https://doi.org/10.1103/PhysRevA.90.062320).
- [36] Giacomo Torlai and Roger G Melko. A neural decoder for topological codes. *Physical Review Letters*, 119(3):030501, 2017. DOI: [10.1103/PhysRevLett.119.030501](https://doi.org/10.1103/PhysRevLett.119.030501).
- [37] Savvas Varsamopoulos, Ben Criger, and Koen Bertels. Decoding small surface codes with feedforward neural networks. *Quantum Science and Technology*, 3(1):015004, nov 2017. DOI: [10.1088/2058-9565/aa955a](https://doi.org/10.1088/2058-9565/aa955a).
- [38] Chenyang Wang, Jim Harrington, and John Preskill. Confinement-higgs transition in a disordered gauge theory and the accuracy threshold for quantum memory. *Annals of Physics*, 303(1):31–58, 2003.
- [39] Jonathan S Yedidia, William T Freeman, and Yair Weiss. Understanding belief propagation and its generalizations. *Exploring artificial intelligence in the new millennium*, 8:236–239, 2003.
- [40] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *5th International Conference on Learning Representations*, 2016.

## A Training of Neural networks

We have mentioned in the main text that neural networks are a powerful ansatz to model func-

tions  $F : \{0, 1\}^m \rightarrow [0, 1]^n$ . The question is how to choose the individual weights and biases of the neurons to make the network compute  $F$ , or at least give a good approximation. This task can be formulated in terms of an optimization problem where pairs of input and desired output  $(x, F(x))$  are used to find the right weights and biases. In our setup we assume that the inputs of  $F$  are weighed by some probability distribution  $P : \{0, 1\}^m \rightarrow [0, 1]$ . The distribution  $P$  prioritizes certain inputs over others and effectively reduces the dimensionality of the input space. In principle we would want to optimize over all possible pairs of inputs and outputs of  $F$  (while taking  $P$  into account). However, this is generally not practicable so that we restrict ourselves to optimize over some subset  $D \subset \{(x, F(x)) \mid x \in \{0, 1\}^m\}$ . The set  $D$  is sampled according to this distribution  $P$ . The optimization of the network is called *training* and the sample  $D$  is called the *training data* or *training set*.

We will now describe the training of neural networks based on gradient descent. We denote the weight vector of the  $i$ th neuron in layer  $l$  by  $w_i^l$  and the  $j$ th entry of this vector by  $w_{i,j}^l$ . Similarly, the bias of the  $i$ th neuron in the  $l$ th layer is labeled  $b_i^l$ . These are the parameters that we need to optimize. An essential ingredient for the optimization is a measure of how good a neural network performs on the training data  $D$ . This measure is called the *cost function*  $C_D(w_{i,j}^l, b_i^l)$  which usually maps the values of the weights  $w_{i,j}^l$  and biases  $b_i^l$  of the neural network into  $[0, \infty]$ . If the value of the cost function is small then this is an indicator that the network performs well. For reasons that will become apparent in the following discussion, we demand  $C_D$  to be differentiable. An obvious choice for the cost function is the average squared  $L^2$  norm  $\|\cdot\|^2$  of the difference of the networks output  $F_N(x, w_{i,j}^l, b_i^l)$ , which depends on the choice of the weights  $w_{i,j}^l$  and biases  $b_i^l$ , and the desired value  $F(x)$  over all elements of the training set  $D$ :

$$C_D(w_{i,j}^l, b_i^l) = \frac{1}{2|D|} \sum_{(x, F(x)) \in D} \|F^N(x, w_{i,j}^l, b_i^l) - F(x)\|^2 \quad (9)$$

To optimize the weights and biases we perform an iterative procedure called *gradient de-*

*scent*. A good introduction to gradient descent and its variants can be found in [16]. Generally, gradient descent is a tool to find a local minimum of a differentiable function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  which is close to some initial point  $x_0 \in \mathbb{R}^n$ . In the first step we evaluate the negative gradient  $-\nabla f$  at  $x_0$ . By following the negative gradient for a small enough distance we will obtain a point  $x_1 := x_0 - \eta_0 \nabla f(x_0)$  such that  $f(x_1) \leq f(x_0)$ . Iterating this process gives a sequence  $x_0, x_1, x_2, x_3, \dots$ , where

$$x_{i+1} := x_i - \eta_i \nabla f(x_i). \quad (10)$$

If the parameters  $\eta_i$  where chosen small enough we have that  $f(x_{i+1}) \leq f(x_i)$  so that the sequence  $x_i$  will converge towards the location of a local minimum. Clearly, we do not want to choose the  $\eta_i$  too small or otherwise the rate of convergence of the  $x_i$  will be slow. However, if we are choosing  $\eta_i$  too large it will make us overshoot the location of the local minimum. There are situations in which  $f$  satisfies conditions, i.e. if  $f$  is convex and smooth, in which there exists an explicit choice of  $\eta_i$  for which the convergence can be guaranteed. In the context of training neural networks the parameters  $\eta_i$  are collectively referred to as the *learning rate*. At the time of writing there is no developed theory on how to choose the learning rates optimally and we have to consider heuristics. An overview of several heuristics can be found in [27].

Let us now apply gradient descent to optimize neural networks. The setup is the following: We have some set of training data  $D$ , a neural network with some initial choice of weights and biases and a cost function  $C_D$ . The task is to find weights and biases which (locally) minimize the cost function  $C_D$ . This confronts us with the problem of how to compute the gradient  $\nabla C_D$ . This is solved by the second major ingredient of the training of neural networks: The *backpropagation algorithm*. The backpropagation algorithm consists of two steps: In the first step we compute the cost function  $C_D$  of the neural network (Eqn. 9) as well as record all values of the neurals in the network. To evaluate the output of the network  $F^N$  we evaluating the input  $x$  on the first hidden layer and then feed the output of the first hidden layer into the second hidden layer and so forth until we obtain the output of the network at the last layer. In the second step of the backpropagation algorithm we compute the derivative

of the cost function with respect to all weights and biases. The derivatives can be computed in linear time in the number of neurons. Obtaining the derivatives is a matter of applying the chain rule several times. To simplify notation we introduce the variable  $s_i^l = \sum_{k \in \text{pred}(i,l)} w_{i,k}^l f_k^{l-1} + b_i^l$ , where  $\text{pred}(i,l)$  and  $f_i^l$  are the predecessors and the value of the  $i$ th neuron in the  $l$ th layer respectively. Remember that the value of a neuron is  $f_i^l = \sigma(s_i^l)$ . The derivatives of the cost function  $C_D$  with respect to the weight  $w_{i,j}^l$  can be expanded as

$$\frac{\partial C_D}{\partial w_{i,j}^l} = \frac{\partial C_D}{\partial s_i^l} \frac{\partial s_i^l}{\partial w_{i,j}^l} \quad (11)$$

The second factor of Equation 11 is simply

$$\frac{\partial s_i^l}{\partial w_{i,j}^l} = f_j^{l-1}. \quad (12)$$

The form of the first term of Equation 11 depends on whether  $l$  is a hidden layer or the output layer. For  $l = L$  we expand over the values of the neurons in the output layer  $F_k^N = f_k^L$

$$\frac{\partial C_D}{\partial s_i^L} = \sum_{k=1}^n \frac{\partial C_D}{\partial f_k^L} \frac{\partial f_k^L}{\partial s_i^L} = \frac{\partial C_D}{\partial f_i^L} \sigma'(s_i^L) \quad (13)$$

For all hidden layers  $l < L$  we expand over the sums  $s_k^{l+1}$  of neurons which are in the next layer and connected to the  $i$ th neuron in layer  $l$

$$\begin{aligned} \frac{\partial C_D}{\partial s_i^l} &= \sum_{k \in \text{succ}(i,l)} \frac{\partial C_D}{\partial s_k^{l+1}} \frac{\partial s_k^{l+1}}{\partial s_i^l} \\ &= \sum_{k \in \text{succ}(i,l)} \frac{\partial C_D}{\partial s_k^{l+1}} w_{i,k}^{l+1} \sigma'(s_i^l) \end{aligned} \quad (14)$$

where  $\text{succ}(i,l)$  indicates the set of all neurons in the next layer connected to the  $i$ th neuron in layer  $l$ . The derivatives with respect to the biases  $b_i^l$  proceeds completely analogously, the only difference being that Equation 12 evaluates to 1.

Note that in order to compute Equation 14 for some layer  $l$  we need to have the result of layer  $l+1$ . Hence we first evaluate Equation 13 for the output layer and then go backwards through the layers of the network (hence the name backpropagation). Finally, having obtained all derivatives with respect to the weights and biases allows us to perform a single step in the gradient descent (see Equation 10).



In the discussion above we made two simplifications which we are going to address now: The first simplification was the choice of the cost function. The  $L^2$  norm is very intuitive but it leads to a very slow convergence of the training. The

reason for this can be seen in [Equation 13](#) and [Equation 14](#). The derivatives are proportional to the derivative of the sigmoid function  $\sigma'(z)$  which is close to 0 when  $|z|$  is sufficiently large. This is avoided by choosing the *cross-entropy* as cost-function:

$$C_D = -\frac{1}{|D|} \sum_{x \in D} \left( \sum_{k=1}^n \left[ F_k(x) \log(F_k^N(x)) + (1 - F_k(x)) \log(1 - F_k^N(x)) \right] \right) \quad (15)$$

The cross-entropy has a less intuitive form. However, since  $F(x) \in [0, 1]$  and  $0 < F^N(x) < 1$  one can see that the cross-entropy is (a) positive and (b) small when the output of the network is close to the desired output. The cross-entropy has the advantage that in [Eqn. 11](#) the derivatives of the sigmoid function cancel, see [\[17, 22, 26\]](#) for a derivation.

The second simplification was taking the average over the whole training set  $D$ . In practice, the training set is usually subdivided into several subsets called *batches* so that a batch of data can be loaded into memory. This is known as *stochastic gradient descent*. Furthermore, part of the available training data is kept aside and not used for the training of the network. This data set is called the *validation set*  $V$  and it is only used to evaluate the cost function after every step of the training. The reason to keep the validation set separate is to check whether the neural network performs well on data outside of the training data. In other words, this is the first measure against overfitting. To summarize the training procedure:

1. Initialize the weights and biases.
2. Pick a batch  $B \subset D$  and learning rate  $\eta$ .
3. Perform a single step of the gradient descent, using backpropagation to compute the gradient.
4. Compute the cost function  $C_V$  on the validation set.

As long as  $C_V$  keeps descending we repeat steps 2 - 4. The initial values in step 1 are usually chosen to be random.

## B Description of the parallel line decoder

In this section, we will describe in detail the parallel line decoder. As we mentioned in [section 5.1](#), its purpose to decode the syndromes that consist only a few parallel lines, and we designed it with easiness to code in mind. The steps are the following:

1. Make a list of current violated parity checks. Order the violated edges in the list by their direction.
2. For each edge  $e$  in the list: Assume  $e$  has coordinate  $(x_0, x_1, x_2, x_3, d)$ , we look for the closest edge  $e' = (y_0, y_1, y_2, y_3, d)$  that still has a violated parity check, such that  $x_d = y_d$ . We then change one of the  $x_i$  in  $e$  so that  $e$  and  $e'$  get closer by flipping the corresponding qubit. Update the syndrome.
3. Repeat Step 1, 2 until no parity check is violated or certain time limit is exceeded.

## C Brief summary of the AlphaGo architecture

In this section we will give a brief summary about the architecture of AlphaGo [\[30\]](#), which ends up being a strong AI at playing board game Go. It highlights the importance of combining different type of machine learning in solving complicated tasks.

1. Supervised learning is used to train a model to approximate human strategy, where the optimization is done to predict human move

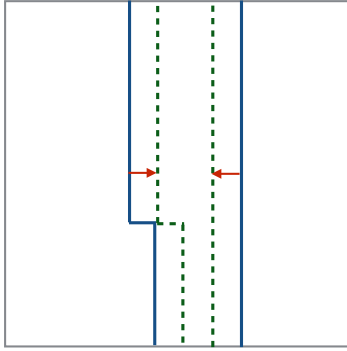


Figure 11: The effect of Step 2 illustrated in a 2D cross section. The solid blue lines are the initial violated parity checks, and the dotted green lines are the violated parity checks after a single execution of the Step 2. In the next loop, no violated parity check will be left.

with most accuracy. While solely mimicking human moves can already achieve non-trivial performance [8], it is almost surely not the best strategy. Therefore the following steps are needed.

2. Reinforcement learning is then used to further improve the model, with the goal of achieving best win rate against previous trained models. Roughly speaking, it is done by gradually changing parameters of the neural network towards the direction of winning more games, starting from the parameters obtained from the supervised learning above.
3. Monte-Carlo tree search is hand-picked as a subroutine, which reflects the 2-player turn by turn nature of the game Go. The details of Monte-Carlo tree search is not of concern to this paper. Here the point being a large and important fraction of the AlphaGo AI is pre-determined.

While it is possible with only the reinforcement learning in the step 2, the trained AI can still achieve a similar performance as the AlphaGo, it is a much riskier approach. This is because the reinforcement learning can get stuck at some local minimum. The goal of step 1 is to set the start point of optimization to be approximately the human strategy, so that we can hope the local minimum obtained by reinforcement learning is at least better.

In a later paper [31], the authors proposed a new training procedure which does not require

the dataset of human moves. With the nature of 2-player game in mind, they use the policy neural network which decides the next move to play out the following few turns of both players. By doing that, they are able to recognize the better next move, and use that as a training target for the policy neural network. Overall, this approach avoids the difficulty of training the policy neural network solely based on the win or loss of a match which typically consists of hundreds of moves. Similarly, in our paper we try to avoid the same difficulty by explicitly training the neural network to recognize qubits affected by errors from the syndromes, instead of training the neural network to achieve longer memory times. We envision the possibility of applying reinforcement learning after the supervised learning phase for our decoder, so that it can even find a slightly better strategy or adapt to not drastically different noise models.

## D A toy example of error correction with 1-nearest neighbor algorithm

The goal of this section is to demonstrate a baseline machine learning algorithm can fail at a toy problem similar to decoding topological codes when the lattice gets large. This should serve as an alert when attempt to use neural networks for decoding, especially due to the lack of understanding of the generalization behavior of neural networks (see [40]). The simple algorithm we will be considering is the 1-nearest neighbor algorithm. It belongs to the family of  $k$ -nearest neighbor algorithms, which is used for classification and regression. The output of these algorithms only depend on the  $k$  entries in the dataset that are closest to the input, e.g. average over the  $k$  entries for regression. It can be considered as a natural extension to the lookup table when the input space is too large and we cannot pre-compute all possible inputs (lookup table has been used for decoding the surface code in [35]) The concrete procedure of using 1-nearest neighbor algorithm for our toy problem will be explained below.

Suppose we are given a square lattice with size  $L^2$ . Among all the plaquettes,  $\lfloor pL^2 \rfloor$  are flipped, where  $p < 0.1$  is some fixed small number. Our goal is to flip them back, but it is forbidden to look at the lattice directly, and we can only com-

pare it to entries of a database. The database contains  $N = \text{poly}(L)$  entries, and each one is generated by randomly flip  $\lfloor pL^2 \rfloor$  plaquettes. A natural way to use the database is to find the entry that has the most overlapping flipped plaquettes with our lattice, and then flip all the plaquettes in that entry. This approach can be considered as a 1-nearest neighbor algorithm. Intuitively, this strategy will perform poorly if  $L$  is large, as the entries of the database are too sparse compared to all possibilities. In more detail, define the random variable  $C$  to be the number of overlapping flipped plaquettes between two randomly generated lattice configuration described above. For large  $L$ ,  $C$  can be well approximated by  $\mathcal{N}(L^2 p^2, L^2 p \alpha)$ . Or equivalently,

$$\frac{C - L^2 p^2}{L \sqrt{p \alpha}} \approx \mathcal{N}(0, 1) \quad (16)$$

Since it is known that [10]

$$\mathbb{E} \left( \max_{i \leq N} (X_i) \right) \sim \sqrt{\log N} \quad (17)$$

$$\text{Var} \left( \max_{i \leq N} (X_i) \right) \sim 1/\log N \quad (18)$$

where  $X_i \sim \mathcal{N}(0, 1)$ . Thus, in the database, the most similar entry will typically have around  $L^2 p^2 + L \sqrt{p \log N} \alpha'$  overlapping plaquettes. This is much smaller compare to  $L^2 p$ . Therefore, if we flip plaquettes according the entry, the total number of flipped plaquettes will increase instead of decrease.

Instead of this toy problem, we can apply a similar 1-nearest neighbor algorithm to a dataset of (syndrome, position of flipped qubits). Let us consider the regime where the error rate is low. In this case, most flipped qubits are separated from each other, and the syndromes in the dataset are almost equivalent to a list of flipped qubits. Therefore, we might extrapolate from the toy problem discussed above that this particular 1-nearest neighbor algorithm will fail to decode topological codes when the lattice becomes large enough. Thus, if we want to have a machine learning decoder that is scalable, it should ideally distinguish from such an approach.

## E Analysis of the Convolutional Neural Network

In the paper, the convolutional neural networks we use only have one convolution layer with ker-

nel size  $3^4$  and others are  $1 \times 1 \times 1 \times 1$ . Therefore each element in the output (before applying softmax) is determined by its surrounding  $3^4$  region. This is mainly for the sake of fast training and evaluation. It is interesting to ask what will happen if we have a deeper neural network, especially considering the recent success of deep neural networks in image classification and processing. More concretely, we wish to know if we increase the number of convolution layers (and as a result increase the region each qubit can “look around” to decide its error probability), will the machine learning procedure described in our paper perform better? In the previous section we give an example that a nearest neighbor algorithm performs worse on a larger region. And in theory a deeper neural network will have a better ability to remember all the training data, thus has the capacity to be a nearest neighbor algorithm. Therefore, it is important to try to understand some difference between how neural nets and nearest neighbor algorithm generalize to unknown inputs. To this end, we tested a slightly larger neural network for our decoder.

### E.1 Behavior of a deeper neural network trained as decoder

In this subsection, We consider a neural network which has two convolution layer with kernel size  $3^4$ . Thus, each element in the output of the neural network can be influenced by a  $5^4$  region, or around 2500 input bits. Generally speaking, if we have large neural nets on this many input bits, overfitting is very likely to happen. For our particular setup of convolutional neural networks, it is much harder to estimate how many training data are needed to avoid overfitting. Nevertheless, it is still interesting to look at the following two facts:

- The trained larger network has a similar performance when decoding noiseless syndromes compared to the network we use in the main text, but is not tested as extensively.
- The larger network exhibits the decay of sensitivity, which means each element in the output will on average change more if an input bit closer to it is being changed. See [Figure 12](#) for the plot.

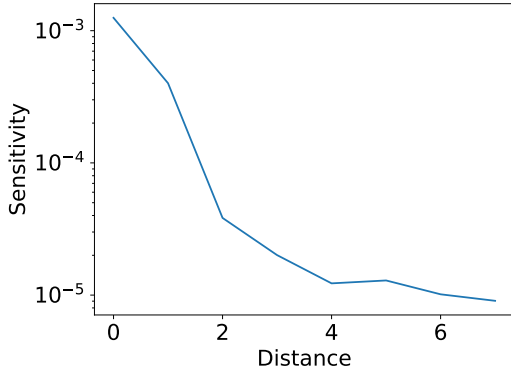


Figure 12: Sensitivity of the neural network output with respect to changing a single input bit. The neural network is trained to estimate the probability of an error happened on the center qubit. The X-axis is the distance between the single changed input bit and the center, induced by  $L^1$ -norm. For each distance, we pick a random input bit and evaluate the difference of output if we change that bit. This is repeated for 20 times and an average is computed.

Note that the second fact partially explained the first one: if the input bits outside the  $3^4$  region do not have much influence on the corresponding output bit, then the smaller neural network used in the paper can in principle approximate the same input-output relation. The decay of sensitivity is likely resulted from the following two reasons. First, the decoding task, and thus the dataset we train our neural networks on have this structure. In more detail, measurement outcome of a parity check far away from a qubit contains very little information about whether an error happened on the qubit. Secondly, as we will discuss in the next subsection, a convolutional neural network has some tendency of prioritizing local information. Intuitively, the alignment of the tendency of the convolutional neural network and the decoding task will lower the chance of overfitting. Therefore, we might think convolutional neural network is not only a convenient machine learning model for this task, but also a naturally suitable one.

## E.2 Tendency of using local information

As mentioned in the last subsection, here we will discuss one mechanism that make the convolutional neural network prioritize local information. The mechanism mainly involves the local connectivity, the initialization and training of the net-

works. We will explain the mechanism using a highly simplified setting. However, we believe it still has some effect for more general cases. We do not aim for rigorousness in this subsection.

In Figure 13 we draw a  $m = 4$  layer 1D CNN with  $n = 7$  input bits and 1 output bit. The variables  $x_{ij}$  are the values of the neurons (as well as inputs). We will first consider the following simplified situation:

1. There is no non-linear activation function applied in the middle of the network. The function  $\tanh$  is only applied after the final layer  $x_{m1}$  (In the figure this is  $x_{41}$ ). There is also no bias term in the network. So the last layer  $x_{m1}$  is a linear combination of the input bits  $\{x_{1i}\}$ . For the ease of notation, the number of channels for each layer is set to 1.
2. The neural network is not convolutional, but instead a normal NN with the same local connectivity as a CNN.
3. The input-output relation to be learned as  $y = x_{11} = x_{1k} = \pm 1$ , where  $k = \lfloor n/2 \rfloor$ ,  $y$  is the desired output, and the probabilities of being 1 and  $-1$  are both 0.5. Other  $x_{1i}$  are set to 0. The cost function is  $c = (y - \tanh x_{m1})^2$ .

It is obvious that the neural net can approximate the above input-output relation well by being a linear combination  $x_{m1} = a_1 x_{11} + a_k x_{1k}$  with the weights satisfy  $a_1 + a_k \gg 1$ . We will argue that with gradient descent as training method, in general we will have  $a_k > a_1$ . In other words, the output of the network  $y' = \tanh(x_{m1})$  will depends more heavily based on  $x_{1k}$  compared to  $x_{11}$ . Operationally, this can be checked by setting  $x_{11} \neq x_{1k}$ . As we never train the neural net with data that have  $x_{11} \neq x_{1k}$ , this can be considered as checking how the neural net generalize to unseen inputs.

We will assume that the weights  $\{w_i\}$  are initialized with the same normal distributions with mean 0 and standard deviation  $\sigma$ . Since there is no non-linear activation function, we have

$$x_{ij} = \sum_k a_k^{(ij)} x_{1k} \quad (19)$$

We will show that when the weights of the network are initialized,  $\text{Var}(a_k^{(ij)})$  is proportional to



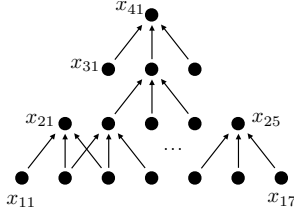


Figure 13: An illustration of the connectivity of a small 1D convolutional neural network. In this section, we set the number of channels to 1. Thus, each  $x_{ij}$  is a real number. The arrays represent dependence relation. For example,  $x_{21}$  is a function of  $x_{11}, x_{12}, x_{13}$ , and in this section the function is simply a linear sum.

the total number of paths from  $x_{1k}$  to  $x_{ij}$ . Let  $p$  be any path from  $x_{1k}$  to  $x_{ij}$ . We can think  $p$  as the set of weights  $w_i$  on the path. It is easy to show that

$$a_k^{(ij)} = \sum_p \prod_{w_i \in p} w_i \equiv \sum_p W_p \quad (20)$$

Note that for different path  $p_1, p_2$ , while  $W_{p_1}$  is not independent of  $W_{p_2}$ , we have  $\text{Cov}(W_{p_1}, W_{p_2}) = 0$ . Thus,

$$\text{Var}(a_k^{(ij)}) = \sum_p \text{Var}(W_p) \quad (21)$$

More generally, we can formulate the above argument as the following observation:

**Observation 1.** Define  $b_{(i_1 j_1)}^{(i_2 j_2)} = \sum_{p \in S} \prod_{w_k \in p} w_k$ , where  $S$  is the set of paths from  $x_{i_1 j_1}$  to  $x_{i_2 j_2}$ . When the weights of the network are initialized,  $\text{Var}(b_{(i_1 j_1)}^{(i_2 j_2)})$  is proportional to the total number of paths in  $S$ .

We want to use the variance to compare the magnitudes of  $b_{(i_1 j_1)}^{(i_2 j_2)}$ . In order for this to work, one condition is that the distributions of  $b_{(i_1 j_1)}^{(i_2 j_2)}$  should all have a ‘‘regular shape’’ (e.g. approximated by normal distribution). In Figure 15, we plotted the distribution of  $a_{[n/2]}^{(m1)}$ , which is defined in Equation 19. Again  $m$  is the layer of the network and  $n$  is the size of input. From this numerical experiment, it is likely that the probability distribution of  $b_{(i_1 j_1)}^{(i_2 j_2)}$  will have a bell shape when  $x_{i_1 j_1}$  and  $x_{i_2 j_2}$  are reasonably far away. Thus, the variances will provide us a very rough way to compare the magnitudes of  $b_{(i_1 j_1)}^{(i_2 j_2)}$ . For example, this implies that in Figure 13,  $x_{32}$  likely has a larger dependence on the  $x_{14}$  compared to  $x_{31}$

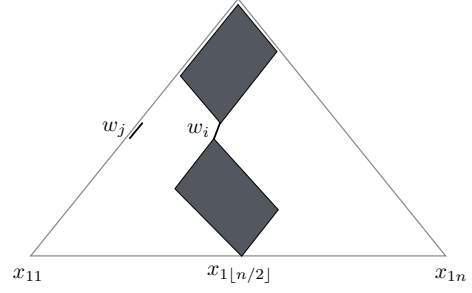


Figure 14: All paths through the grey area contributes to the gradient of  $w_i$ , while only the path along the edge contributes to  $w_j$  (assuming there is no path from  $x_{1[n/2]}$  to  $w_j$ ).

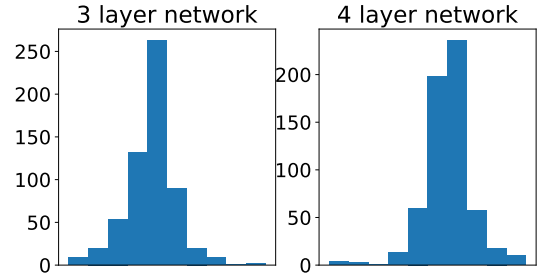


Figure 15: The distribution of the outputs from 3 and 4 layer neural network when the weights are initialized. 600 numerical trials are done in order to plot each figure. The networks have the connectivity shown in Figure 13. The only non-zero element we set in the input is the center element  $x_{1[n/2]} = 1$ . Thus, what we are showing here is the distribution of  $a_{[n/2]}^{(m1)}$ , where  $a_k^{(ij)}$  is defined in Equation 19 and  $x_{m1}$  is the output of the network. We see that the distribution can be described as bell shapes. Thus, the variances already contain a large amount of information of the distribution.

on  $x_{11}$ . Some other conditions will be needed if we want to make the above comparison rigorous. However, we will skip further discussion on this topic and be content with an incomplete analysis.

We can use this approach to compare the gradients of the cost function  $c$  with respect to different  $w_i$ . The gradient sum over the two possible inputs  $x_{11} = \pm 1$  is

$$\sum_{x_{11}=\pm 1} \frac{\partial c}{\partial w_i} = \sum_{x_{11}=\pm 1} \frac{\partial c}{\partial f} \frac{df}{dx_{m1}} \frac{\partial x_{m1}}{\partial w_i} \quad (22)$$

It is easy to see that

$$\left. \frac{\partial c}{\partial f} \frac{df}{dx_{m1}} \right|_{x_{11}=1} = - \left. \frac{\partial c}{\partial f} \frac{df}{dx_{m1}} \right|_{x_{11}=-1} \quad (23)$$

and

$$\frac{\partial x_{m1}}{\partial w_i} = \sum_{p \in S_i} \prod_{w_j \in p, j \neq i} w_j x_{11} \quad (24)$$

where  $S_i$  is the set of paths which go through  $w_i$  and one of  $x_{11}$  and  $x_{1\lfloor n/2 \rfloor}$ . Substitute these in, we have

$$\sum_{x_{11}=\pm 1} \frac{\partial c}{\partial w_i} = 2 \frac{\partial c}{\partial f} \frac{df}{dx_{m1}} \Big|_{x_{11}=1} \sum_{p \in S_i} \prod_{w_j \in p, j \neq i} w_j \quad (25)$$

Since the first two terms on the r.h.s are the same when we compute  $\frac{\partial c}{\partial w_i}$  for different  $w_i$  in the same network, to compare the gradients we only need to consider the last term. By using Observation 1, we know that just after the initialization (or when  $w_i$  have not been too correlated), the gradient of the weight that involved in more paths connecting  $x_{11}$  or  $x_{1\lfloor n/2 \rfloor}$  to  $x_{m1}$  has a larger variance, and obviously the mean of the gradients are 0. On a speculation level, this means most weights connecting  $x_{1\lfloor n/2 \rfloor}$  to  $x_{m1}$  changes faster compared to those connects  $x_{11}$  to  $x_{m1}$  in the same layer. Intuitively, this trends will continue, since the gradient w.r.t to a weight will be larger if other weights on the path have larger absolute values. Indeed, in Figure 16, we can see for one particular setting, the gradient with respect to a weight in the center region is in general larger compared to one in the corner throughout the whole training process.

There is another important factor that causes  $x_{1\lfloor n/2 \rfloor}$  to have a larger influence on the output.

**Observation 2.** *If we assume that at the end of the training, all the weights have the same value, e.g. 1. In this case,  $x_{1\lfloor n/2 \rfloor}$  will still have a larger coefficient in the expansion of the output, because there are more paths connecting from it.*

So the high chance of  $x_{1\lfloor n/2 \rfloor}$  having larger influence is likely a combined result of the Observation 2 and the gradient mechanism we discussed previously.

Now let us discuss the simplification we made above. If the network is convolutional, which means the weights are shared in the same layer, Observation 2 is still true. However, it is not clear whether the above argument about gradients is still relevant, as apparently the weights are changing at the same rates regardless of its

position in one layer. Below we will look at a very specific scenario, in which the magnitudes of the gradients with respect to different weights still play a role in the final outcome. First, note that Observation 1 still holds, because it remains true that  $\text{Cov}(W_{p_1}, W_{p_2}) = 0$  for any two different paths  $p_1$  and  $p_2$ . Assume in the Fig 14,  $w_i$  and  $w_j$  are shared weights and thus  $w_i = w_j$ . Let us consider the possible scenario that the gradient with respect to  $w_i$  and  $w_j$  have different signs, but the norm of the gradient w.r.t  $w_i$  is much larger. Then the update of  $w_i$  (and  $w_j$ ) will be according to the gradient of  $w_i$ , which increase the likelihood of the final decision  $x_{m1}$  to be based on  $x_{1\lfloor n/2 \rfloor}$ . Overall, more study needs to be done regarding the convolutional neural networks.

The effect of activation function in the middle of the network will depend on the choice of the particular function and the initial distribution of the weights. For example, if we initialize all weights with a small enough variance, then initially all  $x_{ij}$  are still in the linear regime of activation function such as sigmoid or tanh. For the popular RELU activation function (i.e.  $y = \max(x, 0)$ ), since it is linear whenever the input is larger than 0, we can expect observation 1 to still be true.

Lastly, we considered a very simple input probability distribution. In general, the inputs are likely to be noisy, and the input-output relation is more complicated than an identity function. To have an toy example of noisy inputs, later in the next subsection we are going to perform a numerical experiment where we replace the 0 input bits to a random  $\pm 1$  in the input distribution we considered above. Intuitively, we can still argue that for  $x_{ij}$  in the middle of a layer will have a larger signal-to-noise ratio compared to the ones on the edge, as a larger part of the variance come actually from the signal. This might suggest that the weights in the middle will change faster.

### E.3 Numerical results

We setup the network and the training data according to the description above, with the only change being that inputs are 2D with size  $7 \times 7$  and  $9 \times 9$ . In particular, the neural networks are not convolutional, but have the same local connectivity as a 2D convolutional one. The number of channels for each hidden layer is 10. We choose one of the corner input bits to be the same (or

reverse) of the center bit. The weights of the network are initialized with normal distribution that has mean 0 and standard variance 0.1. We use the vanilla gradient descent with some exponential decay of the learning rate. The final output of the tanh is rounded to integer and compared to the desired output. The training stops when the average of cost function  $c$  is smaller than 0.02 for a batch of 100 training data (the threshold of 0.02 is chosen so that when given the reversed input, the output of tanh will be rounded to  $\pm 1$  instead of 0). We then test the network on inputs where the chosen corner bit and the center bit have different signs. We repeat the above procedure for 20 times on both  $7 \times 7$  and  $9 \times 9$  inputs. In all experiments, the network always output the center input bit when the center and corner input bits have different sign. To verify the argument in the previous subsection that the gradients with respect to the variables in the center region are larger, we numerically compute the gradients at 10 selected time steps during the training process. The result can be found in Figure 16. Here we set the input size to be  $9 \times 9$ , and the number of channels for each hidden layer to be 1. We compute the gradient of the loss function with respect to two weights. Both weights are connecting from the first to the second hidden layers. One of the weights located in the center, while the other located in the corner corresponding to the non-zero element in the input. In the plot, each vertical segment represents the range of the gradients at a particular time step of the 30 training runs, while the width of transparent color patches represents the distribution. Indeed we can see the gradient of the weight in the center region is in general larger compared to the one in the corner.

We also do a test where the scenario is more complicated (and to some degree more practical). As discussed previously, we can add noise by changing all the 0 in the inputs to a random  $\pm 1$ . RELU activation function (i.e.  $y = \max(x, 0)$ ) is added to the middle layers of the network. The size of the input is  $9 \times 9$ , and accuracy is evaluated by comparing the signs of the final tanh to the desired outputs. Otherwise the setting is the same as in the previous experiment. In Figure 17, we plot the accuracy during the training process. We can see while the accuracy rises when given the aligned inputs (i.e. the particular corner bit

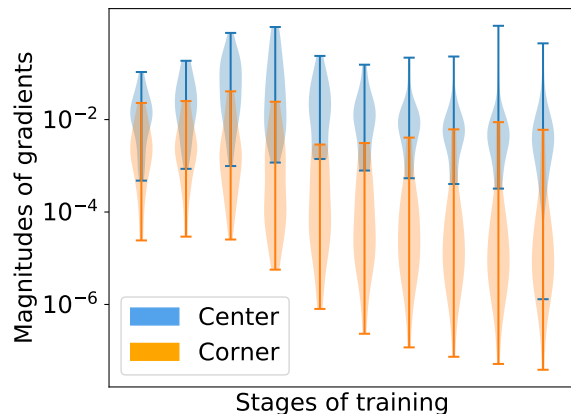


Figure 16: Absolute values of gradients with respect to two weights during different stages of the training. Both weights are connecting from the first to the second hidden layers. The data comes from running the same training process for 30 times, and we compute the gradient at 10 specific time steps during the training. In the plot, the vertical segments made of solid lines represent the ranges of the gradients, while the width of the transparent color patches represents the distribution. We can see the gradient of the weight in the center region is in general larger compared to the one in the corner.

and the center bit are equal), the network becomes more deterministically at outputting the center bit when given reversed inputs (i.e. the particular corner bit and the center bit have inverted sign). The test are repeated several times where similar results are observed.

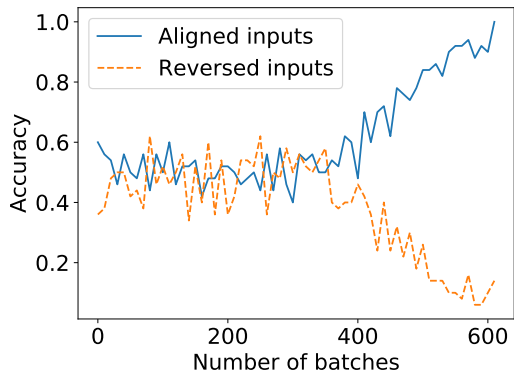


Figure 17: Accuracy of matching the center input bits during training. Inputs are noisy and there are non-linear activation functions for all layers. The blue solid line corresponds to the accuracy when given aligned inputs, while the orange dotted one corresponds to reversed inputs. Note that in order to differentiate the lines, the accuracy for reversed inputs is defined to be the percentage of output matching the corner bit. Thus towards the end of the training, the outputs almost always matching the center bits when given reversed input. The accuracy is evaluated on a batch of 50 inputs.