Neural Decoder for Topological Codes using Pseudo-Inverse of Parity Check Matrix

Chaitanya Chinni,[1,2] Abhishek Kulkarni,[3] Dheeraj M. Pai,[3] Kaushik Mitra,[3] and Pradeep Kiran Sarvepalli[3]

[1]*FoodStreet.in, Chennai 600 042, India*
[2]*YNOS Venture Engine CC Pvt. Ltd., Chennai 600 113, India*
[3]*Department of Electrical Engineering, Indian Institute of Technology Madras, Chennai 600 036, India*

(Dated: January 21, 2019)

Recent developments in the field of deep learning have motivated many researchers to apply these methods to problems in quantum information. Torlai and Melko first proposed a decoder for surface codes based on neural networks. Since then, many other researchers have applied neural networks to study a variety of problems in the context of decoding. An important development in this regard was due to Varsamopoulos *et al.* who proposed a two-step decoder using neural networks. Subsequent work of Maskara *et al.* used the same concept for decoding for various noise models. We propose a similar two-step neural decoder using inverse parity-check matrix for topological color codes. We show that it outperforms the state-of-the-art performance of non-neural decoders for independent Pauli errors noise model on a 2D hexagonal color code. Our final decoder is independent of the noise model and achieves a threshold of 10%. Our result is comparable to the recent work on neural decoder for quantum error correction by Maskara *et al.*. It appears that our decoder has significant advantages with respect to training cost and complexity of the network for higher lengths when compared to that of Maskara *et al.*. Our proposed method can also be extended to arbitrary dimension and other stabilizer codes.

PACS numbers: 03.67.Pp
Keywords: Quantum Error Correction, Neural Networks, Deep Learning, Topological Codes, Surface Codes, Stabilizer Codes, Color Codes.

I. INTRODUCTION

In quantum computers basic unit of information is a qubit. Qubits are highly susceptible to noise. Hence to protect the information, we use quantum codes. A very popular class of quantum codes for protecting information are topological quantum codes. In this paper we focus on a subclass of topological codes in two spatial dimensions called color codes [1]. To correct the impact of noise on the encoded information we would need a decoder. Novel decoding algorithms for 2D color codes have been proposed earlier in [2–5]. However, these are not optimal and do not meet the theoretical bounds for performance. Furthermore, designing decoders for non-Pauli noise is a challenging problem.

Recent developments in the fields of machine learning (ML) and deep learning (DL) have motivated many researchers to apply these methods to decoding quantum codes. Torlai and Melko were the first to propose a decoder for surface codes based on neural networks [6]. Since then, many other researchers have applied neural networks to study a variety of problems in the context of decoding [6–15].

In this paper we only focus on decoding of color codes using neural networks. Early work based on neural networks attempted to the solve the problem using using neural networks entirely. These did not beat the non-neural methods. An important development in this context was due to [7] who proposed a combination of neural networks and non-neural decoders. More precisely, they have a two-step decoder where in the first-step, they estimate an pure-error and in the second-step, they use a neural network which estimates the logical. In their recent work [16], they mention that any simple decoder can be used in the first-step. The authors of [11] claim that the work of [7] is a special case of their generalized framework of building neural networks for decoding stabilizer codes. The works of [9, 10] attempt to use neural networks for fault-tolerant setting. The most relevant work to ours is [15] in which a similar combination of two decoders is employed to conclusively demonstrate the usefulness of neural decoders. They proposed a neural decoder with progressive training procedure that outperformed previously known decoders for 2D color codes.

In this work, we propose a similar two-step neural decoder for color codes and study its performance for the hexagonal color code on the torus. We propose two variations, one which achieves a threshold of 10% and another with an important modification that achieves a near optimal threshold for independent bit-flip/phase-flip noise model. This modification can be incorporated in other neural network based decoders and could be of potentially larger importance. The main challenge involved with neural networks is determining the correct architecture in order to improve the overall threshold. We model our non-neural decoder in a simple way and show the advantages of doing so with the improvement in performance of the neural decoder, the reduction in cost of training and scaling associated with the distance of the code. Our main contributions are,

1) We propose a two-step neural decoder with a simple decoding procedure in the first-step, applicable for all stabilizer codes.
2) We suggest an alternative approach on combining the non-neural and the neural decoder which can be incorporated in other neural network based decoders.
3) Our proposed approaches seem to have significant advantages with respect to training cost and complexity of the network for higher lengths when compared to the previous work of Maskara *et al.* [15].

The paper is organized as follows. We review the necessary background on Quantum Error Correction (QEC), ML and DL in Section II. We then describe our approach, the neural ar-

chitecture used in detail and compare it with related work in Section III. In Section IV, we point out valuable insights from our work and conclude in Section V.

II. BACKGROUND

In this section, we summarize the necessary background on Quantum Error Correcting Codes (QECC). In Section II A, we briefly review stabilizer codes. In this paper we focus on color codes which are introduced in Section II B. Lastly, in the Section II C we describe basics of ML and DL with emphasis on deep learning by discussing the various components in a neural network which can be changed depending on the problem to be solved.

A. Stabilizer codes

In this section, we briefly review stabilizer codes. Recall, that the Pauli group on a single qubit is generated by the Pauli matrices $\{\pm iI, X, Y, Z\}$. The group $\mathcal{P}_n$ consists of tensor products on n single qubit Pauli operators, $P_1 \otimes P_2 \otimes ... \otimes P_n$. A stabilizer code is defined by an abelian subgroup $\mathcal{S} \subset \mathcal{P}_n$, such that $-I \notin \mathcal{S}$. The codespace $\mathcal{Q}$, is joint +1-eigenspace of $\mathcal{S}$.

$$\mathcal{Q} = \{\, |\psi\rangle \in (\mathbb{C}^2)^{\otimes n} \mid S|\psi\rangle = |\psi\rangle \text{ for all } S \in \mathcal{S} \,\}$$

An $[[n, k]]$ stabilizer code encodes k logical qubits into n physical qubits and its stabilizer $\mathcal{S}$ will have $n-k$ independent generators. We assume that $\mathcal{S}$ is generated by $\mathcal{S}_g = \{S_1, \ldots, S_m\}$, where $m \geq n-k$ and $S_1, \ldots, S_{n-k}$ are linearly independent.

Let $\mathcal{C}(S)$ be the centralizer of $\mathcal{S}$ i.e. the set of all Pauli operators that commute with all the elements of $\mathcal{S}$. Let $\mathcal{L}_g = \{\overline{X}_i, \overline{Z}_i\}_{i=1}^{k}$, where $\overline{X}_i$ and $\overline{Z}_j$ denote the logical X and Z operators of the code. Also, $\overline{X}_i, \overline{Z}_j$ commute if $i \neq j$ and anti-commute if $i = j$. Let $\mathcal{L} = \langle \overline{X}_1, \ldots, \overline{X}_k, \overline{Z}_1, \ldots, \overline{Z}_k \rangle$.

We define another set of operators $\mathcal{T}_g = \{T_1, T_2, \ldots T_{n-k}\}$ called the pure errors, such that T_i and S_j commute if $i \neq j$ and anti-commute if $i = j$. The pure errors commute with each other and also with the logical operators. Let $\mathcal{T} = \langle T_1, \ldots, T_{n-k} \rangle$. Note that $\{\mathcal{S}_g, \mathcal{L}_g, \mathcal{T}_g\}$ together form a generating set for $\mathcal{P}_n$.

An error operator, $E \notin \mathcal{C}(S)$ will anti-commute with at least one stabilizer operator in group $\mathcal{S}$. If E anti-commutes with the i^{th} stabilizer $S_i \in \mathcal{S}$, the i^{th} syndrome bit s_i is one and zero otherwise. By calculating the syndrome values for all the stabilizer generators, the syndrome vector can be written as, $\mathbf{s} = (s_1, s_2, ..., s_m)$ where $m \geq n-k$.

We can write the error operator $E = TLS$ up to a phase as proposed in [17]. Here $T \in \mathcal{T}$, $S \in \mathcal{S}$ and $L \in \mathcal{L}$. Note that the operators T, L, S depend on the error E. The effect of S is trivial, implying two error patterns E and $E' = \mathcal{S}E$ will have same effect on codespace. $\mathcal{S}$ introduces an equivalence relation in error operators and hence finding S is of little interest. Also, given syndrome $(\mathbf{s})$, we can uniquely identify T but identifying L is a difficult task. The problem of error correction for stabilizer codes is finding the most likely L given the syndrome vector $\mathbf{s}$. Mathematically, we can write this as,

$$\widehat{L} = \underset{\gamma \in \mathcal{L}}{\operatorname{argmax}}\, Pr\left(\gamma \mid \mathbf{s}\right) = \underset{\gamma \in \mathcal{L}}{\operatorname{argmax}} \sum_{\delta \in \mathcal{S}} Pr\left(\gamma\delta \mid \mathbf{s}\right) \quad (1)$$

Decoding can be thought of as a classification problem. We have 4^k classes, which is exponential in k and this reformulation of the decoding problem as a classification is not much help for large k. Fortunately, surface codes and color codes have fixed number of logical operators for any length and this reformulation can be taken advantage of. However, this is not sufficient, note that the computation of the probabilities in Eq. (1), requires the summation over 2^{n-k} terms which is of exponential complexity. So the reformulation of the decoding as a classification is not adequate, but further work is required to fully exploit this perspective.

B. Color codes

Topological codes are a class of stabilizer codes where the stabilizer generators are spatially local. Popular examples of topological codes are Toric codes [18] and Color codes [1]. Color codes are defined using a lattice embedded on a surface. Every vertex is trivalent and faces are 3-colorable.

Qubits are placed on the vertices of the lattice and for each face f, we define an X and Z type operators called the face operators. We define the the stabilizers as,

$$Z^{(f)} = \prod_{v \in f} Z_v, \qquad X^{(f)} = \prod_{v \in f} X_v$$

All X and Z type operators corresponding to every face generate the stabilizers of the color code. The color code on a hexagonal lattice with periodic boundary is shown in the Fig. 1. It encodes four logical qubits [1].

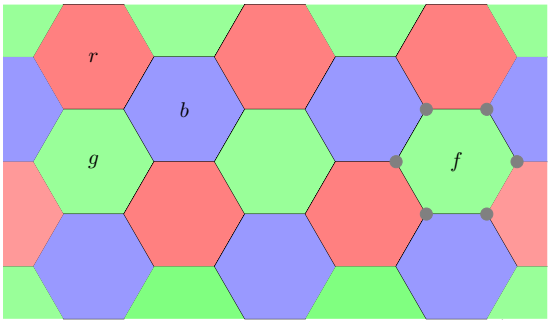


Figure 1: Periodic color code on a hexagonal lattice illustrated with a face and a stabilizer.

C. Machine Learning and Deep Learning

1. An overview of Machine Learning

In traditional computing, algorithms are sets of explicitly programmed instructions which perform a specific task as to give out correct output for the given input. ML is a concept to learn patterns from data through statistical analysis and make predictions without those rules being programmed explicitly. These ML algorithms are therefore data driven methods and the process of learning these rules or patterns is called training of the ML model. Training is essentially an optimization process minimizing an objective function called the loss function. This loss function plays an important role in the algorithm learning these patterns and making good predictions.

There are many such algorithms for solving problems of classification, regression etc and some of them are mentioned in [19, 20]. Any function can be used as a loss function but they need not necessarily help the algorithm learn. There exist specific loss functions which are mathematically proven to be apt for solving each of the above mentioned tasks. Mathematically, the core of any ML algorithm is to estimate the parameters of a function or set of functions which solve the given task.

Training can be classified into two types, supervised learning and the unsupervised learning. The requirement for supervised learning is labeled dataset of inputs $(\mathbf{x})$ and the corresponding true outputs $(\mathbf{y})$. These true outputs are sometimes referred to as ground truth. The ML algorithm will learn the patterns in the data by this information of input and correct output during training and tries to predict $(\widehat{\mathbf{y}})$, the correct prediction during testing. Eg. Classification, Regression. In unsupervised learning, we still have input data but the corresponding ground truth information is not present. The ML algorithm is required to learn the patterns from the input data alone without the information of the ground truth. Eg. Clustering.

2. An overview of Deep Learning

Neuron and Activation functions: A *neuron* is an element which takes an input $\mathbf{x}$ and performs the operation $f\left(\mathbf{w}^\top \mathbf{x} + b\right)$ as shown in the Fig. 2. The parameters $\mathbf{w}$ are called weights and the parameter b is called the bias. Each element of these vectors $\mathbf{x}$, $\mathbf{w}$ and b are real numbers. The function f is a non-linear function and is called the *activation function*. Some common activation functions include `Sigmoid`, `TanH`, `ReLU` (Rectified Linear Unit) etc as shown in the Fig. 3 and are exhaustively discussed in [21].

Deep Learning (DL) is a method in ML to estimate the parameters of a function using a combinations of this basic element neuron. It is common to address the combined set of parameters in $\mathbf{w}$ and b as weights or parameters and we follow this same convention in our subsequent discussion. The activation function plays a very crucial role in DL since without that, a neuron just performs a linear operation.

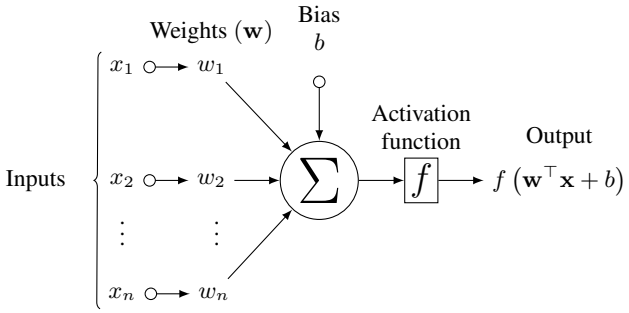


Figure 2: A single neuron which accepts input $\mathbf{x}$ and outputs $f\left(\mathbf{w}^\top \mathbf{x} + b\right)$ where f is an activation function. The vectors $\mathbf{x}, \mathbf{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$.

Architectures: Different combinations of these basic neurons result in different architectures. Some of such famous architectures are Fully-Connected Networks (FC), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN) etc. All these architectures comprise of layers which are again a combination of neurons. Essentially, these architectures can be characterized by these layers.

Fully-connected Network: We briefly describe the FC architecture which we use in our work as shown in Fig. 4. Any FC network has an input layer, an output layer and hidden layers. Each layer comprises of neurons and each neuron is connected to every other neuron in the adjacent layers. Connectedness implies that each neuron receives the output of the neurons it is connected to in the previous layer and it passes the output of itself to all the connected neurons in the next layer. All the neurons in every layer follow this rule except that the neurons in the input layer take the input from the data and the neurons in the output layer give us the final prediction. The input data and the output prediction varies from problem to problem. In a simple image classification task, the input data is the image and the output is the class label. As mentioned before, the non-linear function plays a crucial role in the success of DL in estimating complicated functions efficiently, making DL a very powerful tool.

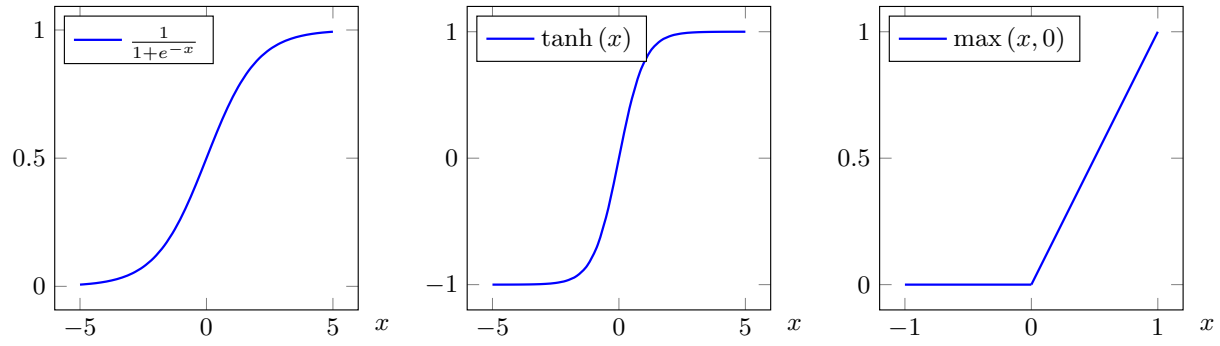
Loss functions: The loss function plays a prominent role in the performance of any DL model. It is calculated between the true label $(\mathbf{y})$ or the ground truth and the prediction made by the network $(\widehat{\mathbf{y}})$. The training procedure as described next ensures that the predictions made by the network get closer to the ground truth by minimizing the loss function as the training progresses. For regression problem, commonly used loss functions are are ℓ_2 and ℓ_1 norms as defined below.

$$\ell_2\left(\mathbf{y}, \widehat{\mathbf{y}}\right) = \left\|\mathbf{y} - \widehat{\mathbf{y}}\right\|_2 = \sum_i \left(y_i - \widehat{y_i}\right)^2$$

$$\ell_1\left(\mathbf{y}, \widehat{\mathbf{y}}\right) = \left\|\mathbf{y} - \widehat{\mathbf{y}}\right\|_1 = \sum_i \left|y_i - \widehat{y_i}\right|$$

For classification problems, *cross-entropy* (ℓ_{CE}) is used as the loss function which is defined in the following equation.

$$\ell_{CE}\left(\mathbf{y}, \widehat{\mathbf{y}}\right) = -\sum_i y_i \log\left(\widehat{y_i}\right)$$



(a) `Sigmoid` function (b) `TanH` function (c) `ReLU` function

Figure 3: Various activation functions used commonly in DL. Note that `ReLU` does not saturate for high inputs.

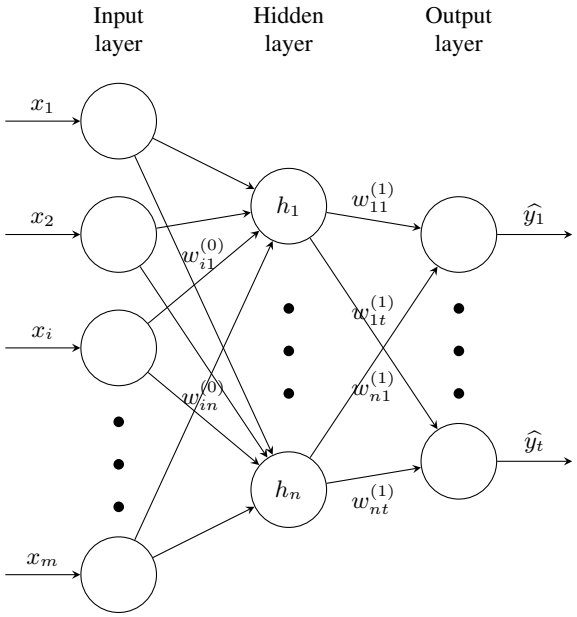


Figure 4: A sample fully-connected architecture with one hidden layer. Each neuron in every layer is connected to every other neuron in the adjacent layers. In this example, the size of the input vector is m and the size of the output vector is t. There are n hidden nodes in the hidden layer. The parameters $\mathbf{w}$ represent the weights of the network.

We use this cross-entropy loss in our work since QEC can be viewed as a classification problem as described in Section III A. We discuss the reasons for using this loss in Section III C.

Training: Training is nothing but estimating the values of the weights of the network which minimizes the chosen loss function for the given training data or the input-output pairs. One of the traditional method of updating the weights to minimize a function is *Gradient Descent* (GD) algorithm. It is an iterative algorithm which tries to optimize the objective function and in our case minimize the loss function (ℓ) through updating the weights $(\mathbf{w})$ of the network in each iteration by following the update rule defined below, as discussed in [21].

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} \ell\left(\mathbf{y}, \mathbf{x}, \mathbf{w}_t\right)$$

Here, $\mathbf{w}_i$ are the weights of the network at the i^{th} iteration. The weights $\mathbf{w}_0$ are initialized randomly. There are many methods to initialize these weights and we mention about them shortly. The parameter α is called the *learning-rate* and is a *hyper-parameter*. There are many such hyper-parameters and we also discuss them later in this section. The speed with which and the optima to which the model converges to, depends on α.

The gradient descent algorithm requires us to train on the entire training dataset at once, i.e calculate the average loss for all the inputs in the dataset and update the weights. Since that is not usually computationally feasible, a popular variant of it called the *Stochastic Gradient Descent* (SGD) is employed. Instead of training on the entire dataset at once, the model is trained on small batches of data until all the training data is exhausted which completes one *epoch*. The size of this batch is called the *batch-size* as mentioned in [21]. For example, if the entire dataset contains 1000 data points, then GD requires us to calculate the average loss on all the 1000 inputs and then update the weights in one iteration. In SGD, say we choose the batch-size to be 50, then 50 data points are chosen randomly from the entire dataset of 1000. The average loss is calculated for that batch of 50 and the weights are updated. This completes one iteration. In the second iteration, another set of 50 data points are chosen randomly from the remaining 950 data points and the rest of the procedure follows. In this example, a total of 20 iterations are required to exhaust the entire dataset which completes an epoch.

One of the major limitation of gradient descent and its variants is that it does not guarantee convergence to global optima. Since the loss is calculated between the true label $(\mathbf{y})$ and the prediction of the network $(\widehat{\mathbf{y}})$, it is indirectly a function of the weights of the network $\mathbf{w}$, since $\widehat{\mathbf{y}}$ is a function of $\mathbf{w}$ and $\mathbf{x}$.

Weight initialization and back-propagation: Before training, the weights of the NN, $\mathbf{w}$ are randomly initialized. Weight initialization plays a crucial role in training and performance of the NN. There are many weight initialization methods but

the popular ones are proposed by [22] and [23]. These methods have been shown to perform well in solving classification problems. Training neural networks can be incredibly costly with GD or SGD but with the use of a dynamic programming based algorithm called the *back-propagation* algorithm, the cost of training reduces significantly as discussed in [21]. The back-propagation algorithm also uses gradient-descent but stores the values of the gradients to the current layer in order to calculate the gradients to the weights of the previous layer.

Optimizers: There are many variants of the SGD algorithm described above like RMSProp, AdaGrad as mentioned in [21] which have a modified update rule. All these rules are commonly called *optimizers* since they optimize the weights of our network in order to minimize the loss function. We use *Adam* optimizer, proposed by [24] because of the significant improvements it offers during training and also in the performance of deep neural networks.

Hyper-parameters: As we can see, numerous design decisions are required to build a neural network like the architecture, the loss function, activation function, weight initialization, optimizer etc. Once those are selected, we have few more parameters to experiment with, listed as follows,

i) The number of hidden layers
ii) The learning rate
iii) The number of neurons in each layer
iv) The batch-size

These parameters are called *hyper-parameters* of the network. Choosing the right set of hyper-parameters for a give problem is one of the biggest challenges of DL. These parameters play a crucial role in both training and performance of the networks because the training procedure does not guarantee convergence to global minima of the loss function, as mentioned previously.

3. *Process flow of a common DL architecture*

The process flow of any DL architecture can be modeled as shown in Fig. 5. The NN can be any neural network as described previously. The NN takes an input $\mathbf{x}$ from the training data and makes a prediction $\widehat{\mathbf{y}}$. The loss is calculated between the ground truth $\mathbf{y}$ and the prediction $\widehat{\mathbf{y}}$. The optimizer then updates the weights of the NN according to the update rule. This whole process completes one iteration during training. We repeat this process until the loss value between $\mathbf{y}$ and $\widehat{\mathbf{y}}$ saturates over multiple iterations.

4. *Classification problem*

In machine learning and statistics, classification is the problem of identifying to which of a set of categories or classes a new observation belongs to. This relation is statistically obtained from training data. A classification algorithm will predict the confidence score or the probability of the new observation belonging to a particular class. This can be illustrated in a dummy example of classification between domestic cats and dogs with the knowledge of their weight and length as shown in Fig. 6. The weight and height are called the *features* since the algorithm classifies with that information. Estimating the parameters of the line is solving the classification problem. In general the boundary could be a complicated curve and there could be multiple classes with multiple features. Commonly, these features might not be available and we have to devise algorithms to extract them from the input.

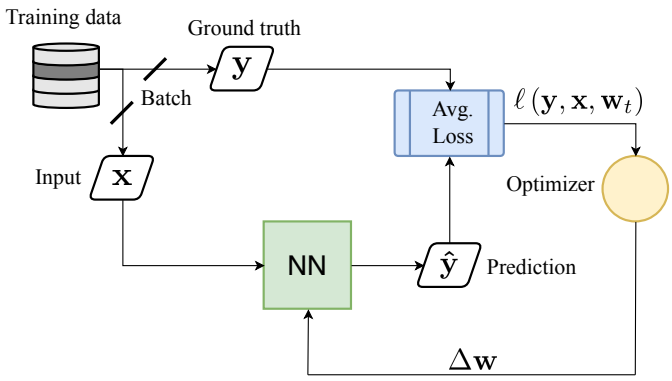


Figure 5: The process flow of any deep learning network. The NN represents any neural network either FC, CNN, RNN etc. It takes input $\mathbf{x}$ and makes the prediction $\widehat{\mathbf{y}}$. The loss is calculated between the ground truth $\mathbf{y}$ and the prediction $\widehat{\mathbf{y}}$ using the weights during the iteration t. The optimizer calculates the updates $\Delta\mathbf{w}$ according to the update rule and modifies the weights of the network for the $(t+1)^{th}$ iteration.

Mathematically, if we assume the feature vector to be $\mathbf{f}$ for an observation x and the total classes are the set $\mathcal{C}$, then the prediction $\widehat{y}$ is the most likely class that x belongs to as defined in the following equation.

$$\widehat{y} = \underset{c \in \mathcal{C}}{\operatorname{argmax}}\, Pr\left(x \in c \mid \mathbf{f}\right)$$

Generally, traditional ML algorithms requires us to extract these features $(\mathbf{f})$ from the input $(\mathbf{x})$ using some rules where as neural networks are known to extract them by themselves from the input directly, for example as shown in [25]. This helps immensely in the success of DL since the network learns to extract the important features for solving the problem, instead of us using hand coded rules to extract what we think are important features.

III. DECODING COLOR CODES USING NEURAL NETWORKS

In this section, we describe our problem formulation for correction of phase errors and how the decoding can be modeled as a classification problem. For any stabilizer code, every error E can be uniquely decomposed to the pure error T, logical error L and a stabilizer S as mentioned in the Section II A.

$$E = TLS$$

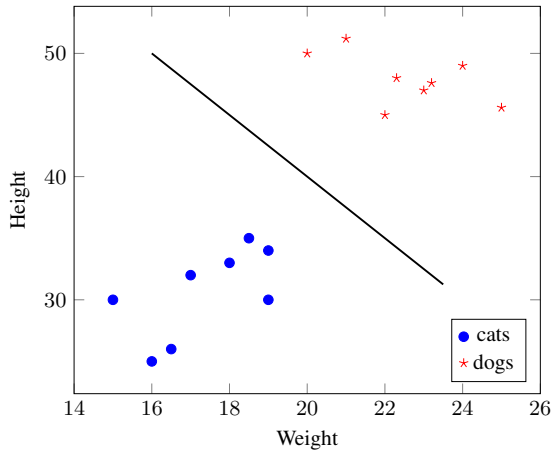


Figure 6: Simple classification between domestic cats and dogs depending on weight and height using dummy data. Estimating the parameters of the boundary is solves the classification problem.

Given the syndrome $\mathbf{s}$, we can uniquely identify T. Since the stabilizers S form the equivalence class, the decoding problem comes down to correctly estimating L given $\mathbf{s}$. In this work, we study CSS codes which have two types of stabilizers, X and Z. They can be written in the matrix form as,

$$\mathbf{S} = \begin{bmatrix} \mathbf{H}_X & 0 \\ 0 & \mathbf{H}_Z \end{bmatrix}$$

Phase errors create X non-zero syndromes and hence we consider only X stabilizers from now on. The matrix $\mathbf{H}_X$ represents the X stabilizers and $\mathbf{H}_Z$ represents the Z stabilizers. For 2D color codes, $\mathbf{H}_X = \mathbf{H}_Z$ and in the subsequent equations, we use $\mathbf{H}$ instead of $\mathbf{H}_X$ for simplicity. Denote the binary representation of E as $\mathbf{e} \in \mathbb{F}_2^n$. Then we can calculate the corresponding syndrome as,

$$\mathbf{s}^\top = \mathbf{H}\mathbf{e}^\top \tag{2}$$

The matrix $\mathbf{H}$ is not full rank. In color code, X stabilizers corresponding to faces have two dependencies as mentioned in [26]. We remove those two dependent stabilizers from the $\mathbf{H}$ matrix, one stabilizer each corresponding to two different colors and denote it as $\mathbf{H}_f$ which is full rank. We calculate the right pseudo-inverse of $\mathbf{H}_f$ and denote it as $\mathbf{H}_f^\dagger$.

$$\mathbf{H}_f\mathbf{H}_f^\dagger = \mathbf{I} \tag{3}$$

The resultant syndrome which does not list the syndromes calculated by the removed dependent stabilizers is denoted by $\mathbf{s}_f$ as shown below.

$$\mathbf{s}_f^\top = \mathbf{H}_f\mathbf{e}^\top \tag{4}$$

A. QEC as a classification problem

Researchers have previously studied the perspective of quantum error correction as a classification problem using neural networks [7, 10, 15]. As mentioned before, we model our decoder as a two-step process. The first-step is a simple inversion where we calculate an estimate $\widehat{E}$ of the actual error E which has occurred. We first calculate the syndrome from Eq. (4) and then estimate $\widehat{\mathbf{e}} \in \mathbb{F}_2^n$, the binary representation of the operator $\widehat{E}$ as follows,

$$\widehat{\mathbf{e}}^\top = \mathbf{H}_f^\dagger\mathbf{s}_f^\top \tag{5}$$

Note that the syndrome of the estimate $\widehat{\mathbf{e}}$ will be same as the syndrome of $\mathbf{e}$. Hence, they have the same pure error T.

$$\begin{aligned} \mathbf{H}_f\widehat{\mathbf{e}}^\top &= \mathbf{H}_f\mathbf{H}_f^\dagger\mathbf{s}_f^\top = \mathbf{s}_f^\top \\ \implies \mathbf{H}\widehat{\mathbf{e}}^\top &= \mathbf{H}\mathbf{e}^\top = \mathbf{s}^\top \end{aligned} \tag{6}$$

This estimate $\widehat{\mathbf{e}}$ computed using Eq. (5) need not be same as $\mathbf{e}$. This is because there exist multiple errors with the same syndrome. We have chosen one solution by fixing $\mathbf{H}_f^\dagger$ which is calculated only once. This makes the first-step of the decoder simple. From Eq. (6), we can conclude that the pure error is same in both E and $\widehat{E}$ and we denote it by T. Applying this initial estimate $\widehat{E}$ onto the system might result in logical error. This can be concluded through the following equations.

$$\begin{aligned} E = TLS \quad \text{and} \quad \widehat{E} &= T\widehat{L}\widehat{S} \\ \implies \widehat{E}E = T\widehat{L}\widehat{S}\,TLS &= (\pm)\,L\widehat{L}S\widehat{S} \\ \implies \widehat{E}E &= (\pm)\,\widetilde{L}\widetilde{S} \end{aligned} \tag{7}$$

Here $\widetilde{L} = L\widehat{L}$ and $\widetilde{S} = S\widehat{S}$. The reason for occurrence of $(\pm)$ in Eq. (7) is because the Pauli operators T, $\widehat{S}$ might commute or anti-commute. This is of little interest to us because we estimate the error up to a global phase.

The homology of $\widehat{E}E$ is same as the homology of $\widetilde{L}$ since $\widetilde{S}$ has a trivial homology. If we can predict the resultant homology $\widetilde{L}$, we can get back to the trivial state and the decoding succeeds. Since the number of homologies are fixed, this is modeled in the second-step of our decoder as a classification problem using NN. The goal of the NN is to predict $\widetilde{L}$ given the syndrome $\mathbf{s}$. Our final error correction will be,

$$\widetilde{E} = \widetilde{L}\widehat{E} \tag{8}$$

If the NN properly predicts $\widetilde{L}$ this correction will restore the state up to a global phase which is evident through the following equations.

$$\begin{aligned} \widetilde{E}E &= \widetilde{L}\widehat{E}E \\ \implies \widetilde{E}E &= (\pm)\,\widetilde{L}\widetilde{L}\widetilde{S} \\ \implies \widetilde{E}E &= (\pm)\,\widetilde{S} \end{aligned}$$

The work by [15] used a naive decoder which removes syndromes by pushing errors to the boundary in the first-step. Their neural network tries to improve upon this estimate by predicting the correction homology. Mathematically, this

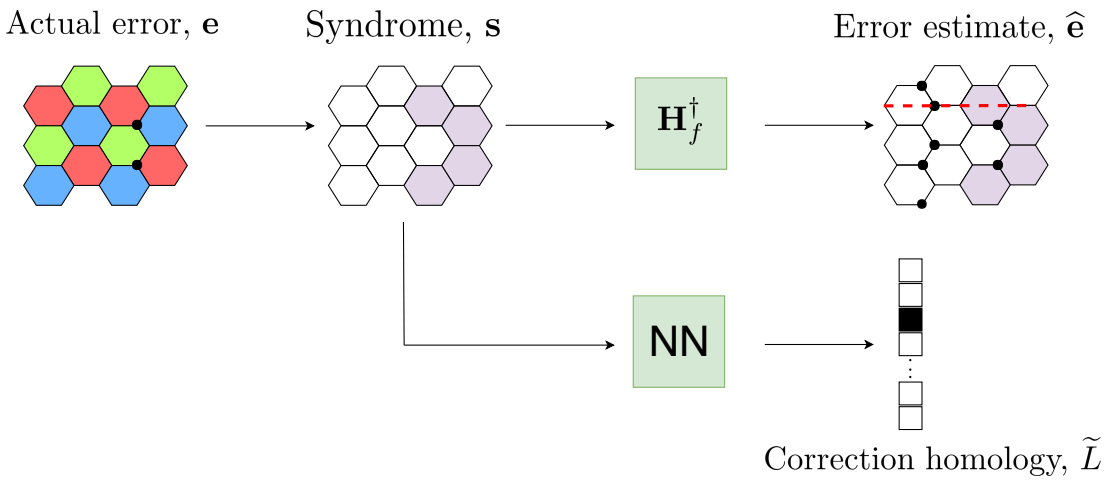


Figure 7: Flow diagram of our two-step decoder. The black dots represent error on the qubits and the marked regions represent the syndrome caused. In the first-step we get an estimate of the error $\widehat{\mathbf{e}}$ and in the second-step, we predict the correction homology $\widetilde{L}$ using our trained NN. Our final error correction is $\widetilde{L}\widehat{E}$. Refer Eqs. (5), (7), and (8). Note that the $\mathbf{H}$-inverse decoder in step-one need not always give us pure error. In this example, the error estimate operator $\widehat{E}$ anti-commutes with a logical operator (red dashed line) and hence cannot be a pure error.

means that their decoder could implement different inverse for a different syndrome. In our approach, we fix the inverse in the first-step, making our initial decoder much simpler. We discuss more on this in the Section IV. The first-step decoder in [7] is to estimate the pure-error which needs to satisfy many properties. We want to emphasize that our inverse matrix $\mathbf{H}_f^\dagger$ in step-one gives us an error estimate which need not always be pure error. It entirely depends on the construction of $\mathbf{H}_f^\dagger$. We used *SageMath* [27], an open-source mathematics software for calculating $\mathbf{H}_f^\dagger$ from Eq. (3).

B. Neural decoder

In this section, we describe our neural decoder in the second-step. As mentioned before, we have modeled our NN in two ways and in both of them we have used a fully-connected architecture where every neuron in one layer is connected to every other neuron in the adjacent layers. The output of the network is the homology vector where each element of it represents a homology class. Since this is a classification problem, we use cross-entropy as our loss function which needs to be minimized during training. We have used Adam optimizer proposed by [24] since it has been observed to perform better than the other optimizers in terms of convergence of the loss. We have also used 1D batch normalization layer after every layer in the network. It is proven to significantly boost the training speed as shown in [28]. The activation function used for every neuron is `ReLU` since it has shown to perform well when compared to other functions like `Sigmoid` or `TanH` by reducing the problem of vanishing gradients as the network goes deeper as shown in [29, 30].

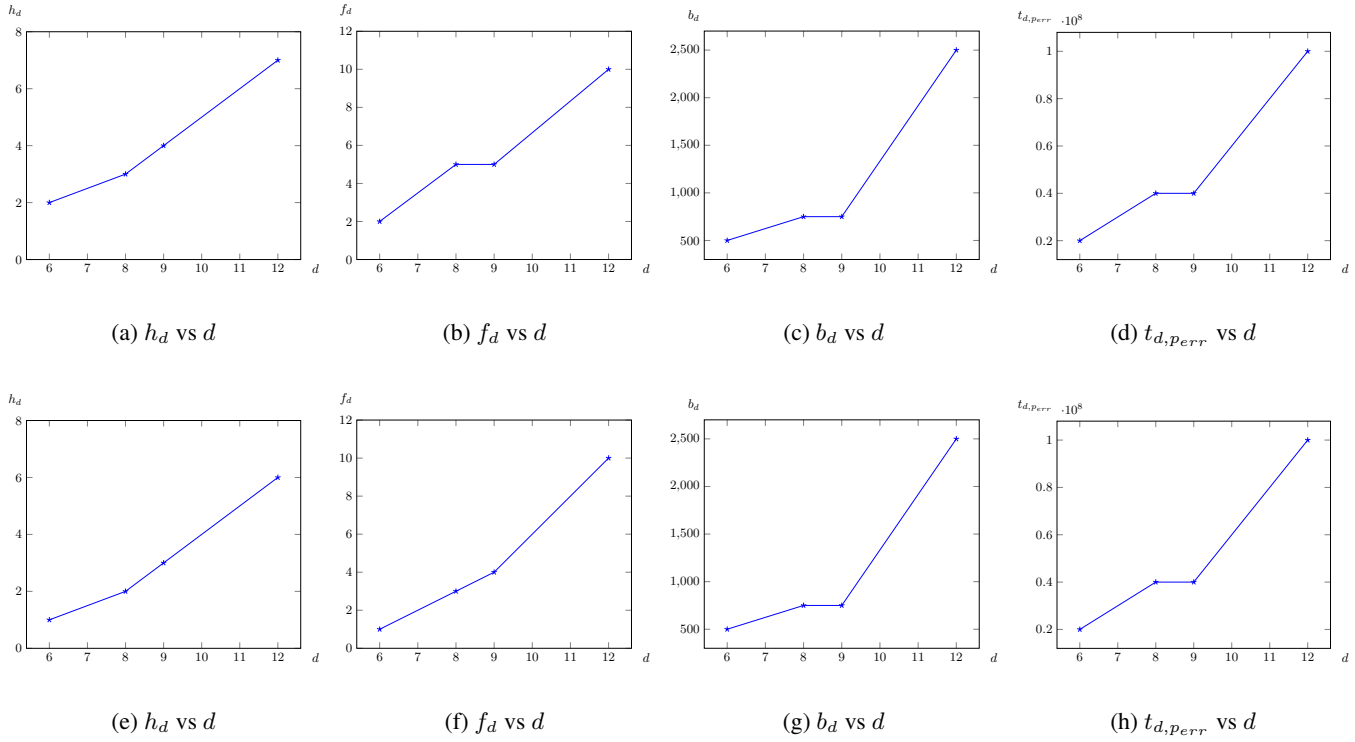
Table I: The values of the hyper-parameters used in the neural decoder in our first approach.

d [a] \ parameters	h_d [b]	f_d [c]	b_d [d]	α [e]	$t_{d,p_{err}}$ [f]	T_d [g]
6	2	2	500	0.001	2×10^7	1.4×10^8
8	3	5	750	0.001	4×10^7	2.8×10^8
9	4	5	750	0.001	4×10^7	2.8×10^8
12	7	10	2500	0.001	10×10^7	7×10^8

[a] Distance of the code
[b] Number of hidden layers
[c] Hidden dimension factor
[d] Batch size
[e] Learning rate
[f] Number of training samples per each p_{err}
[g] Total number of training samples for all p_{err} combined

C. Training procedure

For the network to decode correctly, it needs to be trained. We employ a supervised training procedure where we have labeled data of input (we generate $\mathbf{e}$ according to the noise and calculate syndromes $\mathbf{s}$ from Eq. (2)) and the corresponding output (homology $\widetilde{L}$). This output is the ground truth. Training is nothing but an optimization process where the weights of the network are optimized to minimize an objective function. This objective function is called loss function. The loss function plays a crucial role during training since certain loss functions are apt for certain problems. Since our NN needs to solve a classification problem, we use cross-entropy as our loss function. This is because given a syndrome $(\mathbf{s})$, the NN predicts a probability distribution over all the possible classes. If we assume input is $\mathbf{x}$, the output of the NN is a distribution



(a) h_d vs d (b) f_d vs d (c) b_d vs d (d) $t_{d,p_{err}}$ vs d

(e) h_d vs d (f) f_d vs d (g) b_d vs d (h) $t_{d,p_{err}}$ vs d

Figure 8: Plots of the various hyper-parameters of our neural networks with the distance d of the code. Figs. (a)-(d) are for the first-approach and the Figs. (e)-(h) are for the second-approach.

$\mathbf{q}(\mathbf{x})$ and the true distribution is $\mathbf{p}(\mathbf{x})$, cross-entropy can be written as follows.

$$\ell_{CE}(\mathbf{p},\mathbf{q}) = -\sum_{x} \mathbf{p}(\mathbf{x}) \log \mathbf{q}(\mathbf{x}) \tag{9}$$

This is same as minimizing the Kullback-Liebler divergence (D_{KL}) between the distributions $\mathbf{p}(\mathbf{x})$ and $\mathbf{q}(\mathbf{x})$ up to a constant since $D_{KL}(\mathbf{p}\|\mathbf{q})$ can be written as,

$$D_{KL}(\mathbf{p}\|\mathbf{q}) = \ell_{CE}(\mathbf{p},\mathbf{q}) - \sum_{x} \mathbf{p}(\mathbf{x}) \log \mathbf{p}(\mathbf{x})$$

and the term $\sum_x \mathbf{p}(\mathbf{x}) \log \mathbf{p}(\mathbf{x})$ is a constant because it is completely determined by the true distribution $\mathbf{p}$. This implies that minimizing ℓ_{CE} in Eq. (9) gets the distribution learned by our NN i.e, $\mathbf{q}$ closer to the true distribution $\mathbf{p}$.

Given a syndrome vector $\mathbf{s}$, a trained NN should be able to correctly predict the correct correction homology class $\widetilde{L}$ for all error rates under the threshold. In order to train a NN which is independent of the error rate, we employ a progressive training procedure as described in [15]. We generate training samples at a fixed error rate p_{err} in each case and we train our NN for that noise until the loss function in Eq. (9) saturates. We then move on to a higher p_{err} and repeat the process for various error rates under the threshold. For our experiments (bit-flip noise), we have trained our NN for the error rates $\{0.05, 0.06, 0.07, 0.08, 0.09, 0.10, 0.11\}$. We use Xavier normal initialization for the parameters in fully-connected layers and Gaussian normal initialization for the parameters in batch-normalization layer before we start training. We do not reinitialize the weights during the progressive training while we train on the higher p_{err}. We discuss the importance of this progressive training with evidence in the Section IV. In our first approach, we use the syndrome $\mathbf{s}$ alone as the input to the network whereas in our second approach, we use the concatenated vector of both initial estimate $\widehat{\mathbf{e}}$ and the syndrome $\mathbf{s}$. In both cases, the network is trained to predict correction homology $\widetilde{L}$. Our $\mathbf{H}$-inverse decoder in step-one can be summarized in Alg. 1. The neural decoders can be summarized in Algs. 2, 3 for our first and second approaches respectively. The architectures for our decoders are illustrated in Figs. 7, 11 for first and second approaches respectively.

Algorithm 1 $\mathbf{H}$-inverse decoder (step-one)

Input: Syndrome vector $\mathbf{s}$ and requires pre-computed $\mathbf{H}_f^\dagger$ matrix

Output: Error estimate operator $\widehat{E}$

1: Compute $\mathbf{s}_f$ from $\mathbf{s}$ by removing the syndromes of the removed dependent stabilizers while computing the matrix $\mathbf{H}_f$

2: Compute $\widehat{\mathbf{e}}^\top = \mathbf{H}_f^\dagger \mathbf{s}_f^\top$ ▷ from Eq. (5)

3: Return $\widehat{E}$, the error operator of $\widehat{\mathbf{e}}$ as the initial error estimate

Algorithm 2 Neural decoder (step-two, first approach)

Input: Syndrome vector $\mathbf{s}$, requires the trained neural network to predict the correction homology $\widetilde{L}$ and the initial estimate $\widehat{E}$
Output: Final error correction operator $\widetilde{E}$
1: Using the trained neural network, predict the correction homology $\widetilde{L}$ by giving the syndrome vector $\mathbf{s}$ as the input
2: Compute $\widetilde{E} = \widetilde{L}\widehat{E}$ ▷ from Eq. (8)
3: Return $\widetilde{E}$ as the final error correction

Algorithm 3 Neural decoder (step-two, second approach)

Input: Syndrome vector $\mathbf{s}$ and the initial estimate $\widehat{E}$, requires the trained neural network to predict the correction homology $\widetilde{L}$
Output: Final error correction operator $\widetilde{E}$
1: Using the trained neural network, predict the correction homology $\widetilde{L}$ by giving the concatenated vector of initial estimate $\widehat{\mathbf{e}}$ and the syndrome $\mathbf{s}$ as the input
2: Compute $\widetilde{E} = \widetilde{L}\widehat{E}$ ▷ from Eq. (8)
3: Return $\widetilde{E}$ as the final error correction

D. Results

We describe our simulation results for bit-flip noise model in this section. As described earlier in the Section III, our decoder is a two-step decoder where we use a naive and deterministic $\mathbf{H}$-inverse $\left(\mathbf{H}_f^\dagger\right)$ decoder in step-one and then improve its performance in step-two using a NN. The performance of our $\mathbf{H}$-inverse decoder in the step-one by itself is shown in the Fig. 9. It shows that $\mathbf{H}$-inverse alone is a very bad decoder since the logical error increases as the length of the code increases for a fixed p_{err}. It is quite evident that this decoder does not have a threshold since the curves do not meet anywhere below the theoretical threshold of 10.97% [31].

The performance of our neural decoder in first approach (Fig. 7) trained according to the training procedure mentioned in Section III C is shown in the Fig. 10a. The fully trained NN model is independent of the p_{err} and the it outperforms the previous state-of-the art methods which are not based on neural networks by [3–5]. We report that our neural decoder achieves a threshold of 10% and is comparable to the result mentioned in [15].

In our second approach, we have given additional information of $\widehat{\mathbf{e}}$ along with the syndrome vector $\mathbf{s}$ (by concatenating them both) to our NN (Fig. 11) and saw a dramatic improvement in the threshold for small lengths, as well as a reduction in logical errors for each error rate as shown in the Fig. 10b. The training is exactly similar to the previous case. This shows that the NN is able to understand and learn the behaviour of the $\mathbf{H}$-inverse decoder much better with the additional knowledge of the initial estimate $\widehat{\mathbf{e}}$ and hence is able to perform better correction. This implies that the data driven methods and in particular neural networks' performance can be improved by providing all the information available to us relevant to the problem to be solved. This modification can be incorporated into other works of building two-step decoders using neural networks and improve the overall performance.

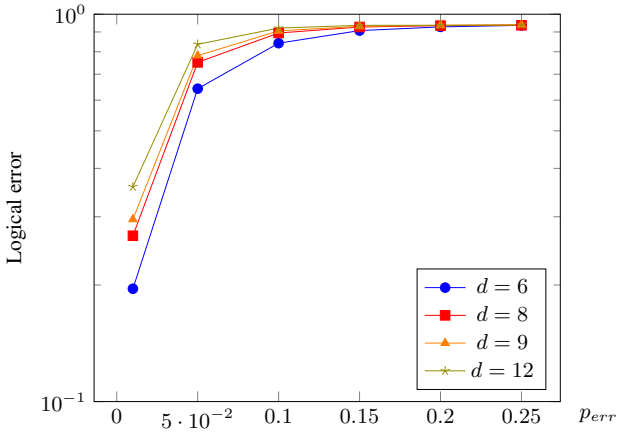


Figure 9: Performance of our $\mathbf{H}$-inverse $\left(\mathbf{H}_f^\dagger\right)$ decoder in step-one. Note that it is a very bad decoder by itself since for a fixed p_{err}, the logical error increases as the length of the code increases and this decoder on its own does not have a threshold.

Table II: The values of the hyper-parameters used in the neural decoder in our second approach.

d [a] \ parameters	h_d [b]	f_d [c]	b_d [d]	α [e]	$t_{d,p_{err}}$ [f]	T_d [g]
6	1	1	500	0.001	2×10^7	1.4×10^8
8	2	3	750	0.001	4×10^7	2.8×10^8
9	3	4	750	0.001	4×10^7	2.8×10^8
12	6	10	2500	0.001	10×10^7	7×10^8

[a] Distance of the code
[b] Number of hidden layers
[c] Hidden dimension factor
[d] Batch size
[e] Learning rate
[f] Number of training samples per each p_{err}
[g] Total number of training samples for all p_{err} combined

The hyper-parameters (as described in the Section II C) of our networks are listed in the Tables I, II for first and second approaches respectively. The variation of some of them with the distance d are shown in the Fig. 8 for both the approaches. The distance of the code is denoted by d and the number of hidden layers in our network is denoted by h_d. The batch size used for each length is denoted by b_d. The number of nodes in each hidden layer are characterized by the hidden dimension factor f_d which is equal to f_d multiplied by the dimension of the input syndrome vector $\mathbf{s}$. The parameter $t_{d,p_{err}}$ is the number of samples required for training for each p_{err} and T_d determines the total number of samples the final trained NN has seen entirely. The parameter α is the learning rate used for optimization. We used *PyTorch* [32], an open-source deep learning framework for training our neural networks.

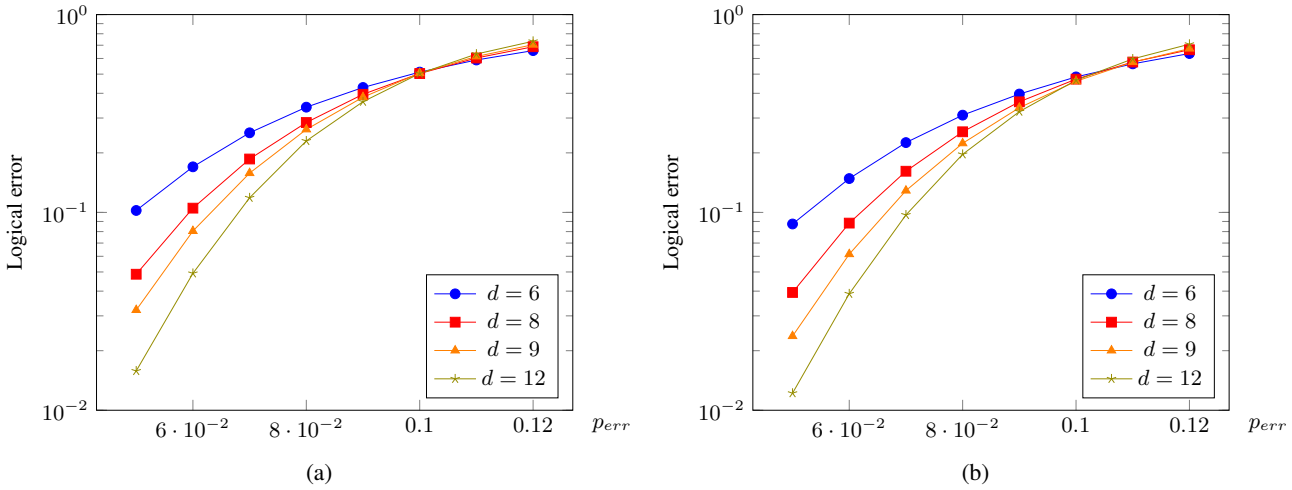


Figure 10: The performance of neural decoder in first approach, achieving a threshold of 10% is shown in (a). The performance of neural decoder in second approach, achieving a near optimal threshold is shown in (b). Note the reduction in logical error for decoder in second approach (b) when compared to that of first approach (a).

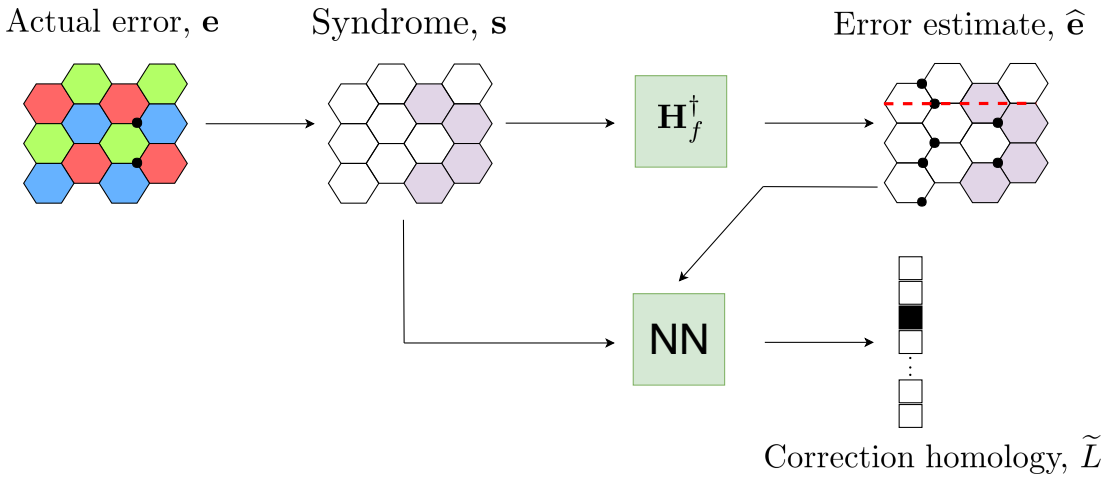


Figure 11: Flow diagram of our two-step decoder. The black dots represent error on the qubits and the marked regions represent the syndrome caused. In the first-step we get an estimate of the error $\widehat{\mathbf{e}}$ and in the second-step, we predict the correction homology $\widetilde{L}$ using our trained NN with the information of both $\widehat{\mathbf{e}}$ and $\mathbf{s}$. Our final error correction is same as $\widetilde{L}\widehat{E}$.

IV. REMARKS AND INSIGHTS

We clearly demonstrate the power of data-driven methods and in particular neural networks, through which we were able to improve the performance of a very bad decoder which does not even have a threshold. When compared to the previous state-of-the-art on neural decoders for color codes, our decoder requires significantly less training data for higher lengths like $d = 9, 12$. In addition to the gains in training cost, our decoder has less complexity with respect to the number of layers and number of nodes in each layer when compared to the previous work and still achieved a comparable threshold. In Section III C, we mentioned the importance of the progressive training. We ran our simulations by training a new NN with Xavier normal and Gaussian normal initializations for every p_{err}, without employing the progressive training. The performance of that decoder with similar hyper-parameters as mentioned in the Table I is shown in the Fig. 12. This shows that without the progressive training, the threshold of the decoder drops to about 7.2%. This is because as the p_{err} increases, it would be very likely that our optimizer converges to a bad local minima. This progressive training is similar to the common practice of *curriculum-learning* in neural networks so that the optimizer converges to a better local minima in the hyperspace of the network weights as proposed in [33]. We also report that this progressive training should be carried on till the p_{err} equals the theoretical threshold and we have observed constant decrement in logical errors at all error rates. Training the model with a p_{err} above the threshold is

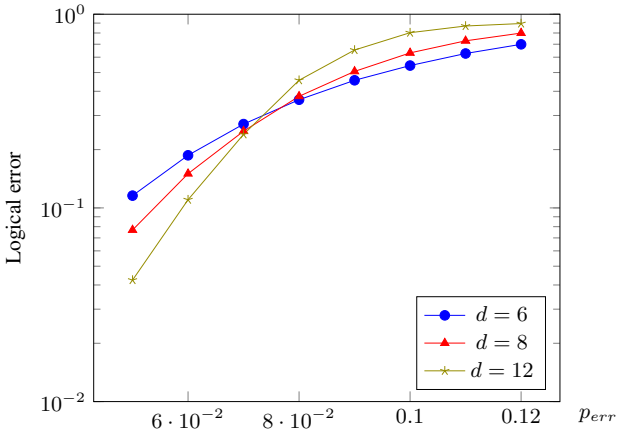


Figure 12: Performance of our neural decoder without the progressive training procedure. The threshold achieved is just about 7.2%.

not desirable as we have seen increments in the logical errors. This concept of $\mathbf{H}$-inverse as a base decoder improved with a neural decoder can be effectively extended to other noise models and also to codes in higher dimension including other stabilizer codes.

Any decoder which does error correction essentially solves the equation $\mathbf{H}\mathbf{x}^\top = \mathbf{s}$. Since there are many solutions, it implies there exist many pseudo-inverses to $\mathbf{H}$. To implement a good decoder, choosing the correct inverse for a given syndrome is an important task. Different inverses must be chosen for different syndrome patterns in order to have a threshold. The choice of decoder in the step-one can be anything as long as it clears the syndrome and good decoders which have a threshold can also be chosen. In such cases, these good decoders take care of selecting the inverse depending on the syndrome. This makes these step-one decoders not entirely simple and there is a lot more for the NN to learn to improve the initial estimate. This is because the inverse selected will be different for different syndromes. In our approach, we fix the inverse $\mathbf{H}_f^\dagger$ though it does not have a threshold and make the step-one decoder very simple. Our NN only has to understand on inverse which is $\mathbf{H}_f^\dagger$ to improve the initial estimate. Intuitively, this means that the learning should be easier for our NN which can be verified empirically through the superior performance with comparatively lesser training cost and complexity when compared to [15]. Our approach is applicable for any decoding problem where the equation, $\mathbf{H}\mathbf{x}^\top = \mathbf{s}$ needs to be solved.

V. CONCLUSION

We have demonstrated that data-driven methods like NN can perform superior decoding when compared to the traditional approaches. We propose a neural decoder with simplified non-neural part achieving a threshold of 10% for 2D color codes. We suggest an alternative approach to combine non-neural and neural decoders reducing the logical error which can be incorporated into other NN based decoders. The drawbacks of NN based decoders are figuring out the right set of hyper-parameters for each length and practical issues of convergence of the loss when the number of trainable parameters increase. Our approach can be extended to other realistic noise models and codes in higher dimensions or other stabilizer codes.

VI. ACKNOWLEDGEMENTS

The authors would like to thank Arun B. Aloshious for valuable discussions. During the preparation of this manuscript, five related preprints were made available [34–38], however their scope and emphasis are different from our work. This work was completed when CC was associated with Indian Institute of Technology Madras as a part of his Dual Degree thesis.

[1] H. Bombin and M. A. Martin-Delgado, "Topological quantum distillation," Phys. Rev. Lett. **97**, 180501 (2006).
[2] David S Wang, Austin G Fowler, Charles D Hill, and Lloyd Christopher L Hollenberg, "Graphical algorithms and threshold error rates for the 2d colour code," arXiv preprint arXiv:0907.1708 (2009).
[3] Pradeep Sarvepalli and Robert Raussendorf, "Efficient decoding of topological color codes," Phys. Rev. A **85**, 022317 (2012).
[4] Nicolas Delfosse, "Decoding color codes by projection onto surface codes," Phys. Rev. A **89**, 012317 (2014).
[5] Hector Bombin, Guillaume Duclos-Cianci, and David Poulin, "Universal topological phase of two-dimensional stabilizer codes," New Journal of Physics **14**, 073048 (2012).
[6] Giacomo Torlai and Roger G. Melko, "Neural decoder for topological codes," Phys. Rev. Lett. **119**, 030501 (2017).
[7] Savvas Varsamopoulos, Ben Criger, and Koen Bertels, "Decoding small surface codes with feedforward neural networks," Quantum Science and Technology **3**, 015004 (2018).
[8] Stefan Krastanov and Liang Jiang, "Deep Neural Network Probabilistic Decoder for Stabilizer Codes," Scientific reports **7**, 11003 (2017).
[9] P Baireuther, M D Caio, B Criger, C W J Beenakker, and T E O'Brien, "Neural network decoder for topological color codes with circuit level noise," New Journal of Physics **21**, 013003 (2019).
[10] Christopher Chamberland and Pooya Ronagh, "Deep neural decoders for near term fault-tolerant experiments," Quantum Science and Technology **3**, 044002 (2018).
[11] Amarsanaa Davaasuren, Yasunari Suzuki, Keisuke Fujii, and Masato Koashi, "General framework for constructing fast and near-optimal machine-learning-based decoder of the topological stabilizer codes," arXiv preprint arXiv:1801.04377 (2018).

[12] Zhih-Ahn Jia, Yuan-Hang Zhang, Yu-Chun Wu, Liang Kong, Guang-Can Guo, and Guo-Ping Guo, "Efficient machine-learning representations of a surface code with boundaries, defects, domain walls, and twists," Phys. Rev. A **99**, 012307 (2019).

[13] Nikolas P. Breuckmann and Xiaotong Ni, "Scalable Neural Network Decoders for Higher Dimensional Quantum Codes," Quantum **2**, 68 (2018).

[14] 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 (2018).

[15] Nishad Maskara, Aleksander Kubica, and Tomas Jochym-O'Connor, "Advantages of versatile neural-network decoding for topological codes," arXiv preprint arXiv:1802.08680 (2018).

[16] Savvas Varsamopoulos, Koen Bertels, and Carmen G Almudever, "Designing neural network based decoders for surface codes," arXiv preprint arXiv:1811.12456 (2018).

[17] G. Duclos-Cianci and D. Poulin, "A renormalization group decoding algorithm for topological quantum codes," in *2010 IEEE Information Theory Workshop* (2010) pp. 1–5.

[18] A.Yu. Kitaev, "Fault-tolerant quantum computation by anyons," Annals of Physics **303**, 2 – 30 (2003).

[19] S. B. Kotsiantis, "Supervised machine learning: A review of classification techniques," in *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies* (IOS Press, Amsterdam, The Netherlands, The Netherlands, 2007) pp. 3–24.

[20] Pedro Domingos, "A few useful things to know about machine learning," Commun. ACM **55**, 78–87 (2012).

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning* (MIT Press, 2016) http://www.deeplearningbook.org.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15 (IEEE Computer Society, Washington, DC, USA, 2015) pp. 1026–1034.

[23] Xavier Glorot and Yoshua Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Proceedings of Machine Learning Research, Vol. 9, edited by Yee Whye Teh and Mike Titterington (PMLR, Chia Laguna Resort, Sardinia, Italy, 2010) pp. 249–256.

[24] Diederik Kingma and Jimmy Ba, "Adam: A method for stochastic optimization," International Conference on Learning Representations (2014).

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems* (2012) pp. 1097–1105.

[26] H. Bombin and M. A. Martin-Delgado, "Topological quantum distillation," Phys. Rev. Lett. **97**, 180501 (2006).

[27] http://sagemath.org.

[28] Sergey Ioffe and Christian Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15 (JMLR.org, 2015) pp. 448–456.

[29] Bekir Karlik and A Vehbi Olgac, "Performance analysis of various activation functions in generalized mlp architectures of neural networks," in *International Journal of Artificial Intelligence and Expert Systems*, Vol. 1 (2011) pp. 111–122.

[30] Xavier Glorot, Antoine Bordes, and Yoshua Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, Proceedings of Machine Learning Research, Vol. 15, edited by Geoffrey Gordon, David Dunson, and Miroslav Dudík (PMLR, Fort Lauderdale, FL, USA, 2011) pp. 315–323.

[31] Helmut G. Katzgraber, H. Bombin, and M. A. Martin-Delgado, "Error threshold for color codes and random three-body ising models," Phys. Rev. Lett. **103**, 090501 (2009).

[32] https://pytorch.org/.

[33] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston, "Curriculum learning," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09 (ACM, New York, NY, USA, 2009) pp. 41–48.

[34] Xiaotong Ni, "Neural network decoders for large-distance 2d toric codes," arXiv preprint arXiv:1809.06640 (2018).

[35] Ryan Sweke, Markus S Kesselring, Evert PL van Nieuwenburg, and Jens Eisert, "Reinforcement learning decoders for fault-tolerant quantum computation," arXiv preprint arXiv:1810.07207 (2018).

[36] Ye-Hua Liu and David Poulin, "Neural belief-propagation decoders for quantum error-correcting codes," arXiv preprint arXiv:1811.07835 (2018).

[37] Philip Andreasson, Joel Johansson, Simon Liljestrand, and Mats Granath, "Quantum error correction for the toric code using deep reinforcement learning," arXiv preprint arXiv:1811.12338 (2018).

[38] Hendrik Poulsen Nautrup, Nicolas Delfosse, Vedran Dunjko, Hans J Briegel, and Nicolai Friis, "Optimizing quantum error correction codes with reinforcement learning," arXiv preprint arXiv:1812.08451 (2018).