Neural Network Based Decoder for Topological Color Codes Using Inverse Parity Check Matrix

A Project Report

submitted by

CHAITANYA CHINNI

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY AND MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

June 2018

THESIS CERTIFICATE

This is to certify that the thesis titled **Neural Network Based Decoder for Topological Color Codes Using Inverse Parity Check Matrix**, submitted by **Chaitanya Chinni**, to the Indian Institute of Technology, Madras, for the award of the dual degree of **Bachelor of Technology and Master of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Pradeep Kiran Sarvepalli
Research Guide
Assistant Professor
Dept. of Electrical Engineering
IIT Madras, 600 036

Dr. Kaushik Mitra
Research Guide
Assistant Professor
Dept. of Electrical Engineering
IIT Madras, 600 036

Place: Chennai

Date: 18th June 2018

ACKNOWLEDGEMENTS

I am deeply grateful to my guides Dr. Pradeep Kiran Sarvepalli and Dr. Kaushik Mitra for their constant support and encouragement during my research. Their immense knowledge and dedication towards research not only motivated me but also helped me in maturing as a researcher and a person.

I am deeply thankful to my faculty advisor Dr. Andrew Thangaraj who was always there for me whenever I needed an advice, both personally and professionally. I had a wonderful opportunity to be a TA for Dr. David Koilpillai. His towering figure yet down-to-earth personality is very inspiring and all my interactions with him have been extremely gratifying. I would also like to thank Abhishek and Dheeraj without whose efforts, this work would have been incomplete.

My five years of stay in IIT Madras is the most amazing part of my life. Surrounded by beautiful minds all around, IIT Madras helped me grow as a person, making me strong and confident. I would like to convey my heartfelt gratitude to my closest friends Swarup, Phani, Deepak, Susmitha, Manideep, and many others for making my stay here such an incredible experience. I would like to thank my seniors Abdeali, Shahidh and Shivaprasad, whom I have always looked up to for their knowledge and maturity and without them, I wouldn't have been the person I am today. I would also like to thank my lab mates Sharath, Anil, Prasan, Lokesh, Honey, Renju, Arun for all the encouragement and discussions, both technical and otherwise. Surrounded by all of them, I can proudly say IIT Madras has been my home.

Finally, I would like to thank my parents, my sister and my brother-in-law for their constant support, love and unwavering belief in me.

ABSTRACT

KEYWORDS: Quantum Error Correction, Neural Networks, Deep Learning, Surface Codes, Stabilizer Codes, Color Codes

Qubits are highly susceptible to noise. In order to build a reliable quantum system, active quantum-error-correction is required. These error correction algorithms should be of low time complexity and guarantee a good threshold. With the recent advancements of hardware and software in Deep Learning, both these requirements can be met. We propose a neural decoder using inverse parity-check matrix for the same and 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% and outperforms the state-of-the-art non-neural decoders proposed by Sarvepalli and Raussendorf (2012); Delfosse (2014). Our result is comparable to the recent work on neural decoder for quantum error correction by Maskara *et al.* (2018). We also conclude that our approach can be extended to arbitrary dimension and to non-CSS codes easily.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS **i**

ABSTRACT **ii**

LIST OF TABLES **v**

LIST OF FIGURES **vii**

ABBREVIATIONS **viii**

NOTATION **ix**

1 Introduction **1**

1.1 Neural network based decoders 1

1.2 Contributions of the Thesis 2

2 Quantum Error Correcting Codes **3**

2.1 Stabilizer Codes 3

2.2 Quantum Error Correction 4

2.2.1 QEC as a classification problem 4

2.3 Color codes 4

3 Machine Learning and Deep Learning **8**

3.1 An overview of Machine Learning 8

3.1.1 Classification problem 9

3.2 An overview of Deep Learning 10

3.2.1 Neuron and Activation functions 10

3.2.2 Architectures 10

3.2.3 Loss functions 12

3.2.4 Training 14

3.2.5 Weight initialization and back-propagation 14

3.2.6 Optimizers 15
3.2.7 Hyper-parameters 15
3.2.8 Process flow 16

4 Error Correction for Periodic Color Code using Neural Networks 17
4.1 Introduction 17
4.2 Problem Modeling 18
4.3 Architecture 20
4.4 Training Procedure 21
4.5 Results 24
4.6 Insights 25

5 Conclusion 27

LIST OF TABLES

4.1 The values of the hyper-parameters used in the neural decoder for bit-/phase-flip noise. 21

LIST OF FIGURES

2.1 Periodic color code on a hexagonal lattice illustrated with a face and a stabilizer. 5

2.2 The periodic hexagonal color code lattice with a phase-flip error E on the left. The corresponding syndromes are indicated in the right. The X stabilizers on the faces f and f' anti-commute and commute with the error E respectively. 6

2.3 The periodic hexagonal color code lattice with a different phase-flip error E' on the left where $E' = ES$. Both E and E' leave the same syndrome and have the same effect on the codespace. 6

2.4 The periodic hexagonal color code lattice with a different phase-flip error E' on the left where $E' = EL$. Both E and E' leave the same syndrome. Applying E' on to E will result in a logically different codespace than the original one. Some qubits are indexed to show that the lattice is periodic. Qubits with the same index are the same. . . . 7

3.1 Simple classification between domestic cats and dogs depending on weight and height using dummy data. Estimating the parameters of the boundary is solving the classification problem. 9

3.2 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 $b \in \mathbb{R}$. . . . 10

3.3 Various activation functions used commonly in DL. Note that `ReLU` does not saturate for high inputs. 11

3.4 A sample fully-connected architecture with one hidden layer. Each neuron in every layer is connected to every other neuron in the adjacent layers. The size of the input vector is m and the size of the output vector is o. There are n hidden nodes in the hidden layer. The parameters $\mathbf{w}$ represent the weights of the network. 13

3.5 The process flow of any deep learning network. The NN represents any neural network either FC, CNN, RNN etc. The NN takes input $\mathbf{x}$ and makes a prediction $\hat{\mathbf{y}}$. The loss is calculated between the ground truth $\mathbf{y}$ and the prediction $\hat{\mathbf{y}}$. The optimizer updates the weights of the network according to the update rule. 16

4.1 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 $\hat{\mathbf{e}}$ and in the second step, we predict the correction homology $L^{\star}$ using our trained NN. Our final error correction is $L^{\star}\hat{E}$. Eqs. (4.5), (4.7), and (4.8). 20

4.2 Plots of the various hyper-parameters used in our work with the distance d of the code. We can see that the curves are almost linear and is an advantage when we scale to higher lengths. 22

4.3 Performance of our $\mathbf{H}$-inverse $(\mathbf{G})$ decoder in step-one. Note that is 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. 24

4.4 Performance of our neural decoder. The threshold achieved is 10%. . 25

4.5 Performance of our neural decoder without the progressive training procedure. The threshold achieved is just about 7.2%. 26

ABBREVIATIONS

DL	Deep Learning
ML	Machine Learning
CSS codes	Calderbank-Shor-Steane codes
QEC	Quantum Error Correction
QECC	Quantum Error Correcting Codes
FC	Fully Connected
CNN	Convolutional Neural Networks
RNN	Recurrent Neural Networks
NN	Neural Network

NOTATION

$\mathbf{A}$	Bold face capital letters denote matrices
$\mathbf{x}$	Bold face small letters denote row vectors
x_n	n^{th} element of $\mathbf{x}$
$\lvert x \rvert$	absolute value of x
$\mathbf{I}_N$	$N \times N$ identity matrix

CHAPTER 1

Introduction

In order to build a quantum computer, we need to store data on quantum particles or qubits. Since these qubits are very sensitive to the external environment, the data will get corrupted easily. Hence the information is encoded in order to protect it. One such class of Quantum Error Correcting Codes (QECC) are the Topological Color Codes and we study the decoding problem for it.

1.1 Neural network based decoders

In order to do Quantum Error Correction (QEC) in real-time for a fault-tolerant system, the decoding algorithm for these codes should be of very low complexity and at the same time, accurate. If the decoder takes more time to correct, the whole system can result in an erroneous state since multiple errors can occur during the time the decoder takes to decode. For topological color codes, there exist some non-neural decoders which achieve threshold of 7.8% Ref. Sarvepalli and Raussendorf (2012) and 8.7% Ref. Delfosse (2014) for independent bit-flip/phase-flip error model which has the theoretical threshold of 10.97% as shown in Katzgraber *et al.* (2009).

Recently, data-driven based neural network decoders have been proposed for quantum-error-correction for various codes and noise models by Torlai and Melko (2017); Varsamopoulos *et al.* (2017); Krastanov and Jiang (2017); Baireuther *et al.* (2018*a*); Chamberland and Ronagh (2018); Davaasuren *et al.* (2018); Jia *et al.* (2018); Breuckmann and Ni (2018); Baireuther *et al.* (2018*b*); Maskara *et al.* (2018). Among them, Maskara *et al.* (2018) have outperformed the traditional decoders in terms of performance for various noise models on triangular color codes and achieved a threshold of 10% for independent bit-flip/phase-flip error model. The works by Varsamopoulos *et al.* (2017); Chamberland and Ronagh (2018); Maskara *et al.* (2018) have a two step decoder where in the first-step, they estimate an error and in the second-step, they use a neural network which improves this estimate.

1.2 Contributions of the Thesis

In this work, we study the decoding problem for topological color codes. We propose a neural decoder which achieves a threshold of 10% for independent bit-flip/phase-flip noise model. The main challenge involved with neural networks is determining the correct architecture in order to improve the overall threshold. We model a part of our decoder in a deterministic 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 it.

The thesis is organized as follows,

Chapter 2 introduces QEC and stabilizer codes with emphasis on color codes. We introduce the equivalence class and how QEC differs from classical error correction.

Chapter 3 broadly introduces Machine Learning (ML) and Deep Learning (DL). We discuss the various components in a neural network which can be changed depending on the problem to be solved.

Chapter 4 discusses our contributions of using neural decoder for QEC. We describe the problem formulation and reduce the decoding problem to a classification problem which can be solved efficiently by a neural network. We describe the architecture and the hyper-parameters and loss function used. We discuss the progressive training procedure employed and show the results through numerical simulations. We also discuss the insights we have concluded from our work. We show the importance of the progressive training procedure with results and describe how our problem modeling simplifies the decoding problem when compared to other works and show the impact in training and scaling cost as we move to higher lengths.

Chapter 5 gives conclusions from our work.

CHAPTER 2

Quantum Error Correcting Codes

In this chapter, we summarize the necessary background on QECC. In section, 2.1 we introduce Stabilizer codes and it's formalism. In Section 2.2 we describe the error correction of stabilizer codes. In this thesis we focus on color codes which are introduced in Section 2.3.

2.1 Stabilizer Codes

In this section we will briefly review stabilizer codes. Stabilizer codes are special class of quantum codes. Let $\mathcal{P}_n$ be the group of Pauli operators acting on n qubits. Stabilizer codes are defined by an abelian group $\mathcal{S} \subset \mathcal{P}$. The group $\mathcal{S}$ consists of Pauli operators, $P_1 \otimes P_2 \otimes ... \otimes P_n$. Stabilizer codes encodes information in +1 eigenspace of Pauli operators in $\mathcal{S}$, i.e codespace, $\mathcal{C}$, is +1 eigenspace of $\mathcal{S}$. Stabilizer operators act trivially on codespace.

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

Stabilizer codes acting on n qubits and encoding k logical qubits will have $n-k$ independent generators. Let $\mathcal{N}(S)$ be the centralizer of group $\mathcal{S}$. Elements of set $\mathcal{N}(S)\backslash\mathcal{S}$ are called logical operators, $\mathcal{L}$, and let $\mathcal{L}_g$ be it's generating set. $\mathcal{L}$ maps $\mathcal{C}$ to itself but map is not trivial as in case of $\mathcal{S}$. Hence, logical operators are said to act non-trivially on codespace.

$\mathcal{L}_g$ has $2k$ generators and $\overline{X}_i, \overline{Z}_i$ for $1 \leq i \leq 2k$. Also, $\{\overline{X}_i\overline{Z}_j\}$ commute if $i \neq j$ and anti-commute if $i = j$.

Once group $\mathcal{S}$ is defined with $\mathcal{S}_g$ as generator set, we implicitly define another set called pure errors, T with generator set $\mathcal{T}_g$ such that,

$$\mathcal{T}_g = \{ \, \forall t \in \mathcal{T}_g \, \exists \, s \in S_g | \, ts = -ts, \forall s' \in S_g \backslash s, \, ts' = s't \, \}$$

Pure errors also commute with each other and logical operators. We should also note that $\{\mathcal{S}, \mathcal{L}, \mathcal{T}\}$ together form the generating set of $\mathcal{P}_n$.

2.2 Quantum Error Correction

In this section we will discuss basics of error correction. Most of the error correction procedures discretize the errors, which collapses any error into Pauli operator, $E \in \mathcal{P}_n$.

Error operator, $E \notin \mathcal{N}(S)$ will anti-commute with at least one stabilizer operator in group, $\mathcal{S}$. If E anti-commutes with the i^{th} stabilizer $S_i \in \mathcal{S}$ then, s_i syndrome bit is one and if it is commuting then s_i is zero. As $n-k$ stabilizers are independent, syndromes vector can be written as, s $= (s_1, s_2, ..., s_{n-k})$. Remaining components of the vector can be obtained by linear combination of $n-k$ components. This s vector is binarized form and it lies in $\mathbb{GF}_2^n$.

2.2.1 QEC as a classification problem

As we already know $\{\mathcal{S}_g, \mathcal{L}_g, \mathcal{T}_g\}$ form a generating set of $\mathcal{P}_n$ we can write $E = TLS$. Here $T \in \mathcal{T}$, $S \in \mathcal{S}$ and $L \in \mathcal{L}$. All the T, L, S are a function of 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. Therefore, $\mathcal{S}$ introduces an equivalence relation in error operators. Hence finding S is of little interest. Also, given syndrome vector we can uniquely identify T but identifying L is a very difficult task. The problem of error correction for stabilizer codes is finding the most likely L given the syndrome vector, s. Surface codes have fixed number of logical operators for any length and hence, decoding can be thought of as a classification problem.

2.3 Color codes

Topological codes are special class of stabilizer codes where Pauli operators are spatially local. Popular examples of topological codes are Toric codes and Color codes.

In this section we briefly introduce the 2D color code whose lattice is shown in the

Fig. 2.1. The code lattice shown is periodic and is embedded on a torus. Every vertex is trivalent and faces are 3-colorable.

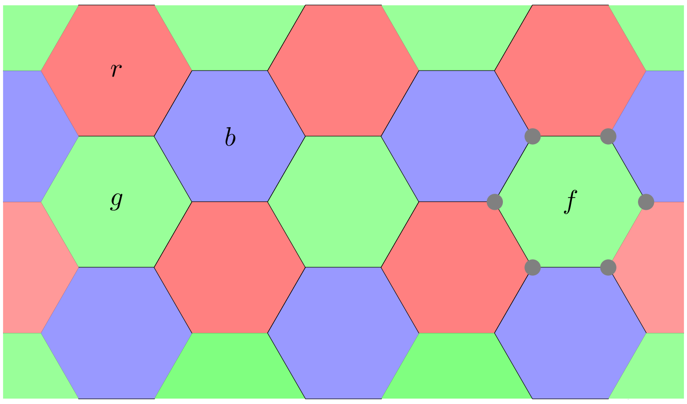


Figure 2.1: Periodic color code on a hexagonal lattice illustrated with a face and a stabilizer.

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,

$$Z^{(f)} = \prod_{v \in f} Z_v, \qquad X^{(f)} = \prod_{v \in f} X_v \tag{2.1}$$

We use notation, $v \in f$ to denote that v is a vertex of f. All X and Z type operators corresponding to every face collectively form the stabilizer generators for the color code. The color code with periodic boundary encodes four logical qubits as shown in Bombin and Martin-Delgado (2006*a*).

Consider a periodic lattice as shown in Fig. 2.2. The black dots in the left indicate the phase-flip error E and they leave a trace which is the syndrome as indicated on the right. The X stabilizer on the face f anti-commutes with the error operator E and hence leaves a syndrome whereas the X stabilizer on the face f' commutes with the error E and hence does not leave a syndrome.

Consider the same lattice with a different error E' as shown in the Fig. 2.3. This error pattern also leaves the same syndrome as in Fig. 2.2. Here, E and E' differ by a stabilizer S i.e, $E' = ES$ and hence their effect is same on the codespace. This forms the equivalence class of errors. The problem of QEC is to find the most likely equivalence class given the syndrome. It is essentially estimating an error up to a stabilizer.

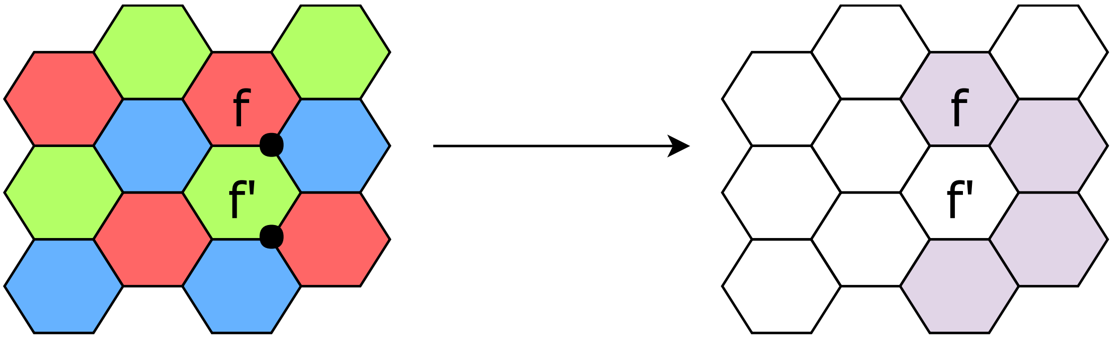


Figure 2.2: The periodic hexagonal color code lattice with a phase-flip error E on the left. The corresponding syndromes are indicated in the right. The X stabilizers on the faces f and f' anti-commute and commute with the error E respectively.

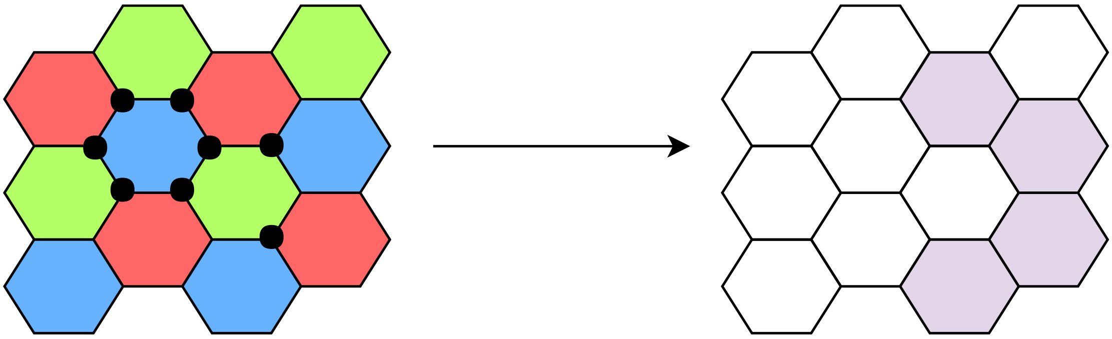


Figure 2.3: The periodic hexagonal color code lattice with a different phase-flip error E' on the left where $E' = ES$. Both E and E' leave the same syndrome and have the same effect on the codespace.

Consider the lattices with the corresponding errors in the Fig. 2.4. These error patterns also have the same resultant syndrome. These error patterns do not form the equivalence class with the error E since they no longer differ by a stabilizer but they differ by a logical operator. Hence, decoding to one of these errors will result in a logical error. This implies applying these errors on to E will result in a logically different codespace than the original one.

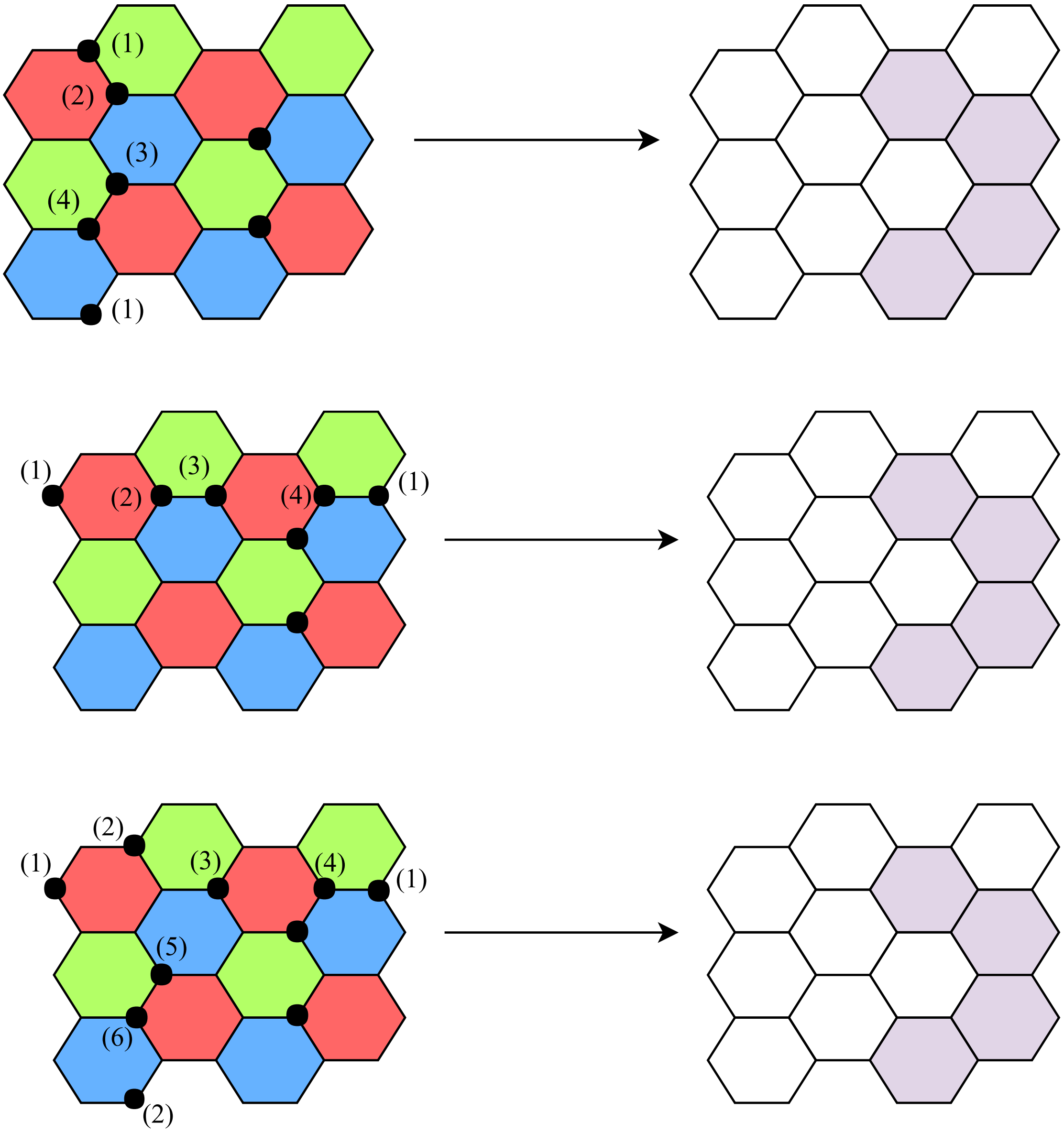


Figure 2.4: The periodic hexagonal color code lattice with a different phase-flip error E' on the left where $E' = EL$. Both E and E' leave the same syndrome. Applying E' on to E will result in a logically different codespace than the original one. Some qubits are indexed to show that the lattice is periodic. Qubits with the same index are the same.

CHAPTER 3

Machine Learning and Deep Learning

3.1 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. Machine Learning (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 Kotsiantis (2007); Domingos (2012). 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.

This 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 $(\hat{\mathbf{y}})$, 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.

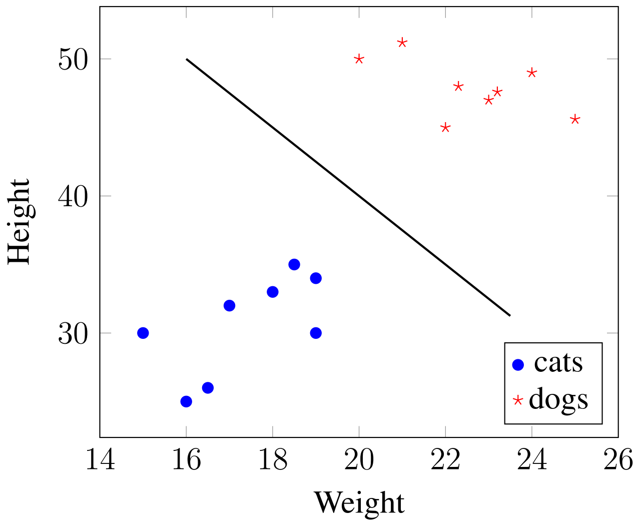


Figure 3.1: Simple classification between domestic cats and dogs depending on weight and height using dummy data. Estimating the parameters of the boundary is solving the classification problem.

3.1.1 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. 3.1. 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.

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 $\hat{y}$ is the most likely class that x belongs to as defined in Eq. (3.1).

$$\hat{y} = \underset{c \in \mathcal{C}}{\operatorname{argmax}} \, Pr\left(x \in c \,|\, \mathbf{f}\right) \tag{3.1}$$

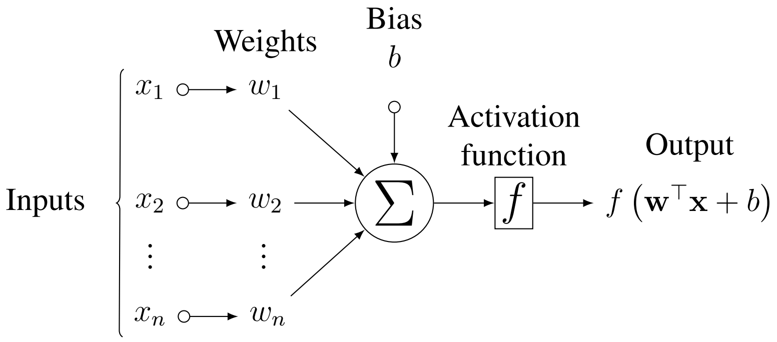


Figure 3.2: A single neuron which accepts input $\mathbf{x}$ and outputs $f\left(\mathbf{w}^\top\mathbf{x}+b\right)$ where f is an activation function. The vectors $\mathbf{x}, \mathbf{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$.

3.2 An overview of Deep Learning

3.2.1 Neuron and Activation functions

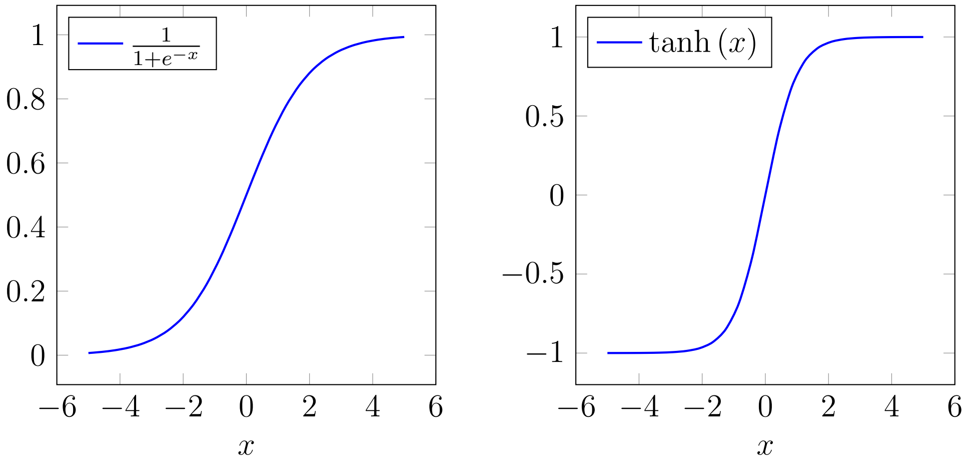
A *neuron* is an element which takes an input $\mathbf{x}$ and performs the operation in Eq. (3.2) where the parameters $\mathbf{w}$ are called weights and the parameter b is called the bias. Each element of these vectors $\mathbf{x}, \mathbf{w}$ and b are real numbers. The function f is a non-linear function and is called the *activation function*. Some common activation functions include `Sigmoid`, `TanH`, `ReLU` (Rectified Linear Unit) etc as shown in the Fig. 3.3 and are exhaustively discussed in Goodfellow *et al.* (2016).

$$y = f\left(\mathbf{w}^\top\mathbf{x}+b\right) \tag{3.2}$$

Deep Learning (DL) is a method in ML to estimate the parameters of a function using a combinations of this basic element neuron, as shown in the Fig. 3.2. 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 important role in DL since without that, a neuron just performs a linear operation.

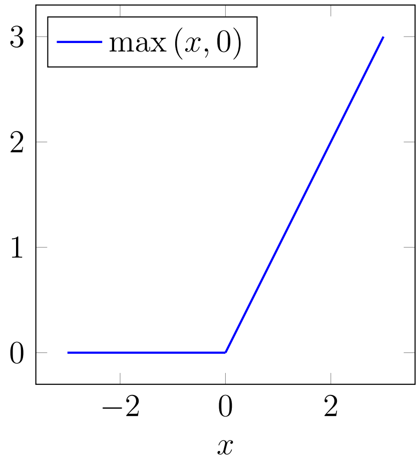
3.2.2 Architectures

Different combinations of these basic neurons result in different architectures. Some of such famous architectures are Fully-Connected Networks, Convolutional Neural Networks, Recurrent Neural Networks etc. All these architectures comprise of layers which



(a) `Sigmoid` function

(b) `TanH` function



(c) `ReLU` function

Figure 3.3: Various activation functions used commonly in DL. Note that `ReLU` does not saturate for high inputs.

are again a combination of neurons. Essentially, these architectures can be characterized by these layers.

In FC architecture, every neuron is connected to every other neuron. Whereas in a CNN architecture, the weights are posed as filters and these filters convolve with the input and produce the output as the convolution operation. This forces the weights in the filter to take a fixed context in the input and produce output from them. CNNs are particularly useful when the input has a local structure. RNNs mainly deal well with sequences, where the current input or output is dependent on the previously seen data. RNNs maintain a state vector which is carried to the next time step which encompasses all the data it has seen till that time-step in a compact way. It is to note that the network in each time-step of RNN can be thought of as a FC or a CNN. The state vector is the most essential part of an RNN. Combinations of FC and CNN can also be used as modeled in Krizhevsky *et al.* (2012).

Fully-connected Network

We briefly describe the fully-connected (FC) architecture which we use in our work as shown in Fig. 3.4. 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 above, the non-linear function plays a crucial role in the success of DL in estimating complicated functions efficiently, making DL a very powerful tool.

3.2.3 Loss functions

The loss function plays an important 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

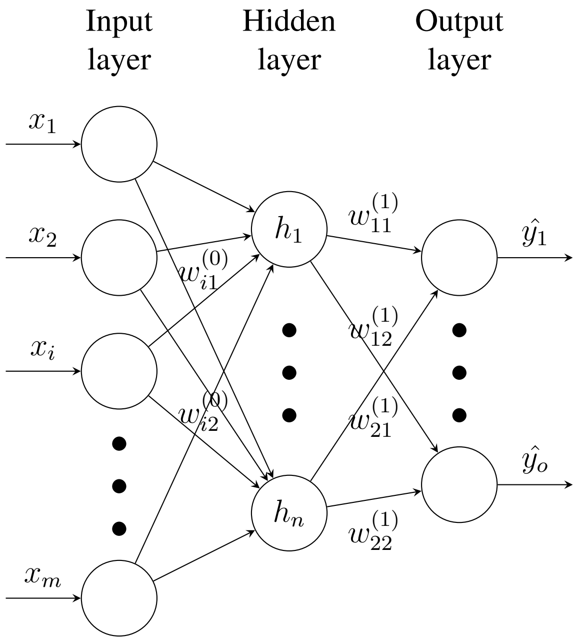


Figure 3.4: A sample fully-connected architecture with one hidden layer. Each neuron in every layer is connected to every other neuron in the adjacent layers. The size of the input vector is m and the size of the output vector is o. There are n hidden nodes in the hidden layer. The parameters $\mathbf{w}$ represent the weights of the network.

the network $(\hat{\mathbf{y}})$. 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 in Eqs. (3.3), (3.4).

$$\ell_2 (\mathbf{y}, \hat{\mathbf{y}}) = \|\mathbf{y} - \hat{\mathbf{y}}\|_2 = \sum_i (y_i - \hat{y_i})^2 \tag{3.3}$$

$$\ell_1 (\mathbf{y}, \hat{\mathbf{y}}) = \|\mathbf{y} - \hat{\mathbf{y}}\|_1 = \sum_i |y_i - \hat{y_i}| \tag{3.4}$$

For classification problems, *cross-entropy* is used as the loss function Eq. (3.5). We use the same loss since QEC can be viewed as a classification problem in Section 2.2.1 and we discuss the reasons for using this loss in Section 4.4.

$$\ell_{CE} (\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \log (\hat{y_i}) \tag{3.5}$$

3.2.4 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* algorithm. It is an iterative algorithm which tries to optimize the objective function and in our case, minimize the loss function (ℓ) by updating the weights $(\mathbf{w})$ of the network in each iteration, by following the rule in Eq. (3.6) as discussed in Goodfellow *et al.* (2016). The 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 perform the update rule. Since that is not usually computationally feasible, a popular variant of it called the *stochastic gradient descent* 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. The size of this batch is called the *batch-size* as mentioned in Goodfellow *et al.* (2016).

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 $(\hat{\mathbf{y}})$, it is indirectly a function of the weights of the network $\mathbf{w}$, since $\hat{\mathbf{y}}$ is a function of $\mathbf{w}$ and $\mathbf{x}$.

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla_{\mathbf{w}} \ell\left(\mathbf{y}, \mathbf{x}, \mathbf{w}_t\right) \tag{3.6}$$

Here, $\mathbf{w}_i$ are the weights of the network at the i^{th} iteration. The parameter α is called the *learning-rate* and is a *hyper-parameter*. There are many such hyper-parameters and they are described in Section 3.2.7. The speed with which and the optima to which the model converges to, depends on α.

3.2.5 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 He *et al.* (2015) and Glorot and Bengio (2010). These methods have been shown to perform well in classification.

Training neural networks can be incredibly costly with gradient descent but with the use of a dynamic programming based algorithm called the *back-propagation* algorithm, the cost of training reduces significantly as discussed in Goodfellow *et al.* (2016). 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.

3.2.6 Optimizers

There are many variants of the gradient-descent algorithm described above like RMSProp, AdaGrad as mentioned in Goodfellow *et al.* (2016) 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 Kingma and Ba (2014) because of the significant improvements it offers during training and also in the performance of deep neural networks.

3.2.7 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 in Section 3.2.4.

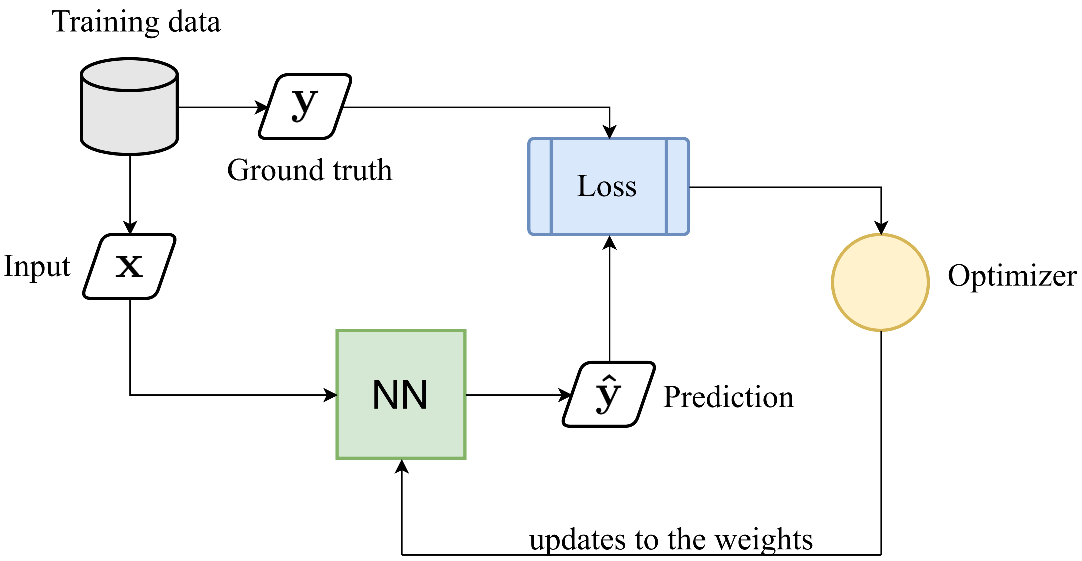


Figure 3.5: The process flow of any deep learning network. The NN represents any neural network either FC, CNN, RNN etc. The NN takes input $\mathbf{x}$ and makes a prediction $\hat{\mathbf{y}}$. The loss is calculated between the ground truth $\mathbf{y}$ and the prediction $\hat{\mathbf{y}}$. The optimizer updates the weights of the network according to the update rule.

3.2.8 Process flow

The process flow of any DL architecture can be modeled as shown in Fig. 3.5. The NN can be any neural network as described previously in Section 3.2.2. The NN takes an input $\mathbf{x}$ from the training data and makes a prediction $\hat{\mathbf{y}}$. The loss is calculated between the ground truth $\mathbf{y}$ and the prediction $\hat{\mathbf{y}}$. 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 $\hat{\mathbf{y}}$ saturates over multiple iterations.

CHAPTER 4

Error Correction for Periodic Color Code using Neural Networks

4.1 Introduction

In this chapter, we describe our problem formulation for correction of phase errors and how the decoding can be modeled as a classification problem. For any surface code, every error E can be uniquely decomposed to the pure error T, logical error L and a stabilizer S as shown in Eq. (4.1) where T, L and S are a function of E. Given syndrome, s we can uniquely identify T. Since the stabilizers S form the equivalence class, the decoding problem comes down to correctly estimating L given the syndrome, s.

$$E = TLS \tag{4.1}$$

Since we are studying CSS codes, we have two type of stabilizers, X and Z. Stabilizers can be written in matrix form as,

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

Phase errors create X non-zero syndromes. Hence we consider only X stabilizers from now on. $\mathbf{H}$ represents X stabilizers. If E is equivalent to $\mathbf{e} \in \mathbb{GF}_2^n$, we can calculate the syndrome as,

$$\mathbf{s}^\top = \mathbf{H}\mathbf{e}^\top \tag{4.2}$$

Matrix $\mathbf{H}$ is not full rank. In color code X stabilizers corresponding to faces have two dependencies as mentioned in Bombin and Martin-Delgado (2006*b*). Hence, we remove the 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. This allows us to calculate the pseudo-inverse of it and we denote it as $\mathbf{G}$ in the Eq. (4.3). The resultant syndrome which does not list the syndromes calculated by the removed dependent stabilizers is denoted as s_f as shown in Eq. (4.4). Hence,

$$\mathbf{G}\mathbf{H}_f = \mathbf{I}_n \tag{4.3}$$

$$\mathbf{s}_f^\top = \mathbf{H}_f \mathbf{e}^\top \tag{4.4}$$

4.2 Problem Modeling

We model our decoder into a two step process. The first step is a simple inversion where we calculate an estimate $\left(\hat{E}\right)$ of the actual error (E) which has occurred. We first calculate the syndrome using the Eq. (4.4) and calculate $\hat{\mathrm{e}} \in \mathbb{GF}_2^n$, the binary representation of the operator $\hat{E}$ as follows,

$$\hat{\mathrm{e}}^\top = \mathbf{G}\mathbf{s}_f^\top \tag{4.5}$$

Note that the syndrome of the estimate $\hat{\mathrm{e}}$ will be same as the syndrome of $\mathbf{e}$. Hence the pure error T of both E and $\hat{E}$ will be the same.

$$\begin{aligned} \mathbf{H}_f\hat{\mathbf{e}}^\top &= \mathbf{H}_f\mathbf{e}^\top = \mathbf{s}_f^\top \\ \implies \mathbf{H}\hat{\mathbf{e}}^\top &= \mathbf{H}\mathbf{e}^\top = \mathbf{s}^\top \end{aligned} \tag{4.6}$$

This estimate $\hat{\mathrm{e}}$ computed using Eq. (4.5) need not be same as $\mathbf{e}$. This is because there exist multiple error patterns with the same syndrome. This aligns with the fact that $\mathbf{H}$ is not full rank and we have chosen one such solution by fixing $\mathbf{G}$ which is calculated only once. This process makes the first step of the decoder deterministic. From Eq. (4.6), we can conclude that the pure error is same in both E and $\hat{E}$ and we denote it by T. Applying this initial estimate $\hat{E}$ onto the system might result in logical errors. This can be concluded through the following equations,

$$
\begin{aligned}
E &= TLS \\
\hat{E} &= T\hat{L}\hat{S} \\
\hat{E}E &= T\hat{L}\hat{S}\,TLS \\
\implies \hat{E}E &= (\pm)\,\hat{L}L\hat{S}S \\
\implies \hat{E}E &= (\pm)\,L^{\star}S^{\star}
\end{aligned}
\tag{4.7}
$$

The reason for occurrence of $(\pm)$ in Eq. (4.7) is because the Pauli operators T, $\hat{S}$ 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 $\hat{E}E$ is same as the homology of $L^{\star}$ since $S^{\star}$ has a trivial homology. If we can predict the resultant homology $L^{\star}$, we can get back to the trivial state and the decoding succeeds. Since the number of homologies are fixed in number, this is modeled in the second step of our decoder as a classification problem using Neural Network (NN). The goal of the NN is to predict $L^{\star}$ given the syndrome s. The NN can learn to predict this because the first step is completely deterministic. Our two step decoder can be illustrated in Fig. 4.1. Our final error correction will be,

$$
E^{\star} = L^{\star}\hat{E} \tag{4.8}
$$

This is because it gets the system into the original state up to a stabilizer and a phase which is evident through these equations,

$$
\begin{aligned}
E^{\star}E &= L^{\star}\hat{E}E \\
\implies E^{\star}E &= (\pm)\,L^{\star}L^{\star}S^{\star} \\
\implies E^{\star}E &= (\pm)\,S^{\star}
\end{aligned}
$$

This idea of a two-step decoder was first started by Varsamopoulos *et al.* (2017) where they use a look-up table for calculating the pure error T and predict the homology L using a neural decoder. Subsequent work of Chamberland and Ronagh (2018) used the same concept in the context of fault-tolerant system. Recent work by Maskara *et al.*

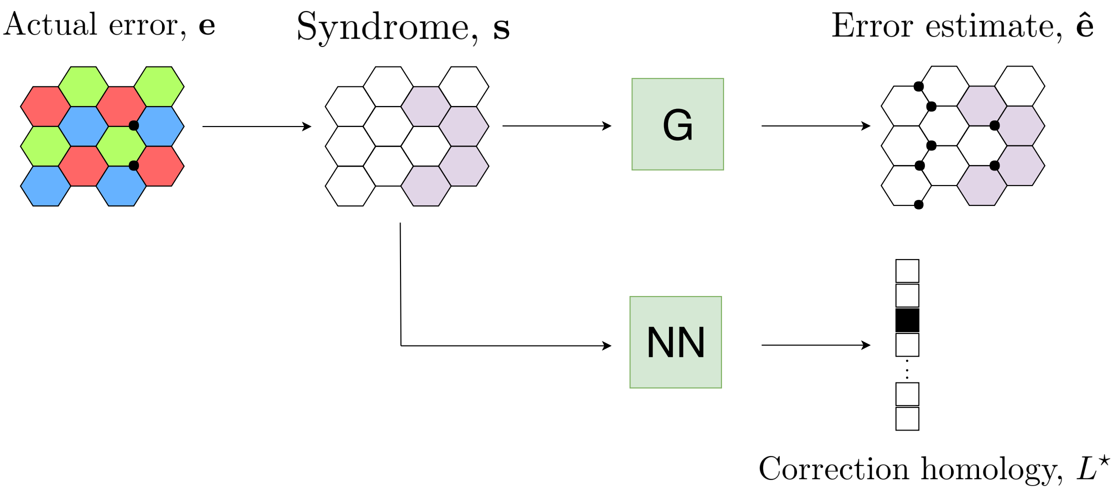

Actual error, $\mathbf{e}$
Syndrome, $\mathbf{s}$
Error estimate, $\hat{\mathbf{e}}$
G
NN
Correction homology, $L^{\star}$

Figure 4.1: 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 ê and in the second step, we predict the correction homology $L^{\star}$ using our trained NN. Our final error correction is $L^{\star}\hat{E}$. Eqs. (4.5), (4.7), and (4.8).

(2018) used a naive decoder which removes syndromes instead of estimating T in the first step. Their neural network tries to improve upon the estimate given by the initial naive decoder by predicting the correction homology. We want to emphasize that our $\mathbf{H}$-inverse in step-one gives us an error estimate which need not always be pure error. It entirely depends on the construction of the inverse matrix $\mathbf{G}$.

4.3 Architecture

In this section we describe our neural decoder in step-two. 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 Kingma and Ba (2014) 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 Ioffe and Szegedy (2015). 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 Karlik and Olgac; Glorot *et al.* (2011).

Table 4.1: The values of the hyper-parameters used in the neural decoder for bit-/phase-flip noise.

parameters d[a]	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

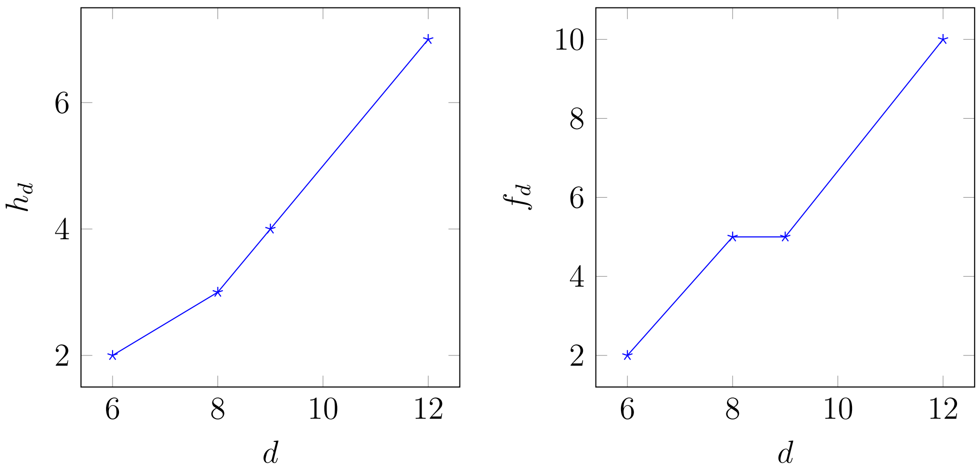
---

[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

4.4 Training Procedure

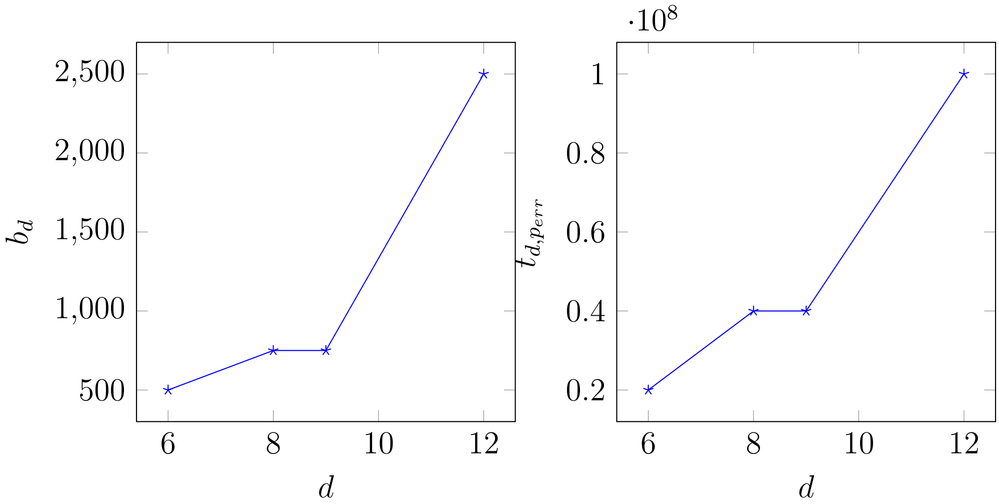
For the network to decode correctly, it needs to be trained. We employ a supervised training procedure where we have labeled data of input (syndromes $\mathbf{s}$ from Eq. (4.2)) and the corresponding output (homology $L^\star$). This output is called ground truth. Training is nothing but an optimization 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 function (ℓ_{CE}) as our loss function in Eq. (4.9). 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 $\mathbf{q}(\mathbf{x})$ and the true distribution is $\mathbf{p}(\mathbf{x})$, cross-entropy can be written as follows,

$$\ell_{CE}(\mathbf{p},\mathbf{q}) = -\sum_{x} \mathbf{p}(\mathbf{x}) \log \mathbf{q}(\mathbf{x}) \tag{4.9}$$



(a) h_d vs d

(b) f_d vs d



(c) b_d vs d

(d) $t_{d,p_{err}}$ vs d

Figure 4.2: Plots of the various hyper-parameters used in our work with the distance d of the code. We can see that the curves are almost linear and is an advantage when we scale to higher lengths.

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 minimizing the Eq. (4.9) 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 $L^\star$ 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 Maskara *et al.* (2018). 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. (4.9) 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 Chapter 4.6.

The hyper-parameters (as described in the Section 3.2.7) we have used for our networks are listed in the Table 4.1. The variation of some of them with the distance d are shown in the Fig. 4.2. The distance of the code is denoted by d and the number of hidden layers in our network is denoted 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 have used *PyTorch*[1], one of the popular deep learning framework for training our neural networks.

---

[1] `https://pytorch.org/`

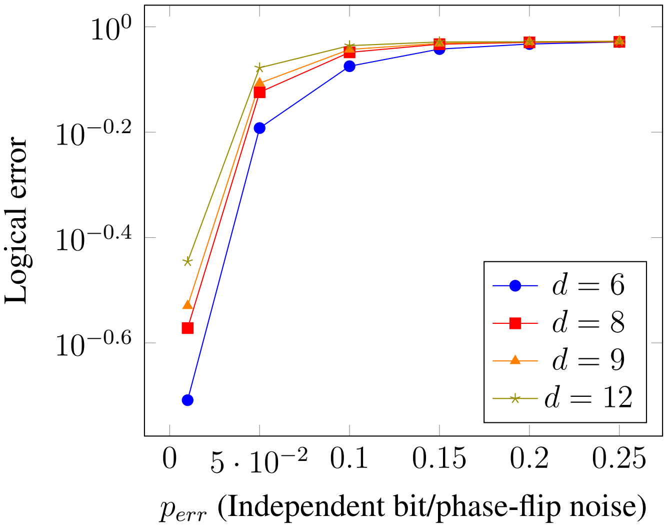


Figure 4.3: Performance of our $\mathbf{H}$-inverse $(\mathbf{G})$ decoder in step-one. Note that is is a very bad decoder by itself since for a fixed p_{err}, the logical error increases as the length of the code increases and this decoder on its own does not have a threshold.

4.5 Results

We describe our simulation results for bit-flip noise model in this section. As described earlier in the Section 4, our decoder is a two step decoder where we use a completely naive and deterministic $\mathbf{H}$-inverse $(\mathbf{G})$ 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. 4.3. 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%.

The performance of our fully-connected NN trained according to the training procedure mentioned in Section 4.4 is shown in the Fig. 4.4. The final trained NN model is independent of the p_{err} and the it outperforms the previous state-of-the art methods not based on neural networks by Sarvepalli and Raussendorf (2012); Delfosse (2014). We report the final threshold achieved by our NN is 10% and is comparable to the result mentioned in Maskara *et al.* (2018).

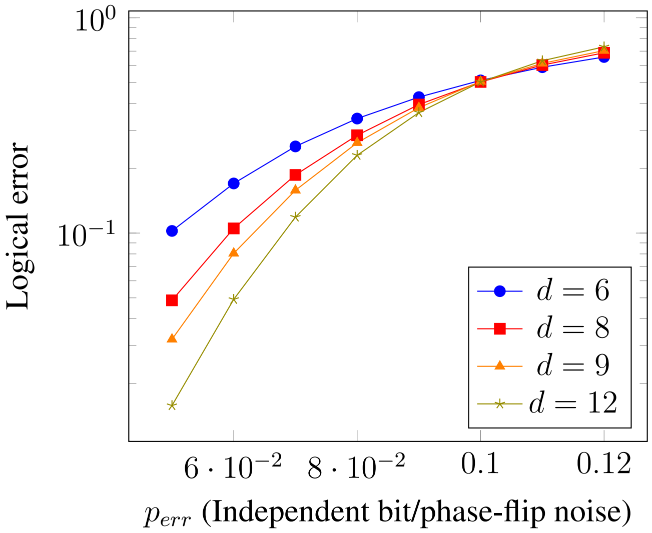


Figure 4.4: Performance of our neural decoder. The threshold achieved is 10%.

4.6 Insights

In this chapter, we mention the key insights we could conclude from this work. 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 Section 4.4, we mentioned the importance of the progressive training. We have run our simulations by training a new NN with Xavier normal and Gaussian normal initializations. The performance of that decoder with similar hyper-parameters as mentioned in the Table 4.1 is shown in the Fig. 4.5. This shows that without the progressive training, the threshold of the decoder drops to 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 Bengio *et al.* (2009). 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 not desirable as we have seen increments in the logical errors. This concept of **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 non-CSS codes.

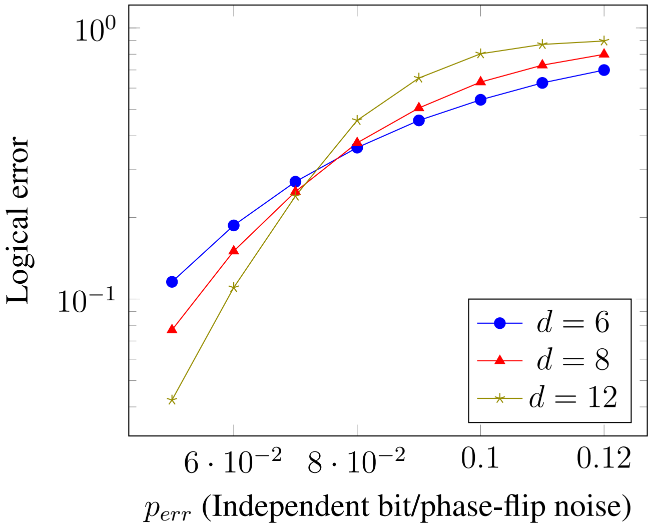


Figure 4.5: Performance of our neural decoder without the progressive training procedure. The threshold achieved is just about 7.2%.

Any decoder which does error correction essentially solves the equation $\mathbf{Hx}^\top = \mathrm{s}$ by removing the dependent stabilizers. To implement a good decoder, determining which dependent stabilizers to remove for a given syndrome is an important task. Till now, most of the approaches of neural decoders use heuristics based decoders like pushing the syndromes to boundaries or estimating the pure error etc in step-one. These heuristics take care of determining which dependent stabilizers to remove depending on the syndrome. This makes these step-one decoders not entirely deterministic and there is a lot more for the neural network to learn to do a good decoding. In our approach we fix the inverse $\mathbf{G}$ and make the step-one decoder completely deterministic. This allows the neural network to easily improve the initial estimate and hence results in superior performance with comparatively lesser training cost. This makes our approach applicable for any decoding problem where the equation, $\mathbf{Hx}^\top = \mathrm{s}$ needs to be solved.

CHAPTER 5

Conclusion

We have demonstrated that data-driven methods like neural networks can perform superior decoding when compared to the traditional rule based approaches. We have stated the importance of progressive training and come up with conditions on it, numerically showing the improvement in threshold because of it. The drawbacks of neural network 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. We believe our approach can be extended to other realistic noise models in building fault-tolerant systems and also to codes in higher dimensions or non-CSS codes.

REFERENCES

1. **Baireuther, P.**, **M. Caio**, **B. Criger**, **C. Beenakker**, and **T. O'Brien** (2018*a*). Neural network decoder for topological color codes with circuit level noise. *arXiv preprint arXiv:1804.02926*.

2. **Baireuther, P.**, **T. E. O'Brien**, **B. Tarasinski**, and **C. W. Beenakker** (2018*b*). Machine-learning-assisted correction of correlated qubit errors in a topological code. *Quantum*, **2**, 48.

3. **Bengio, Y.**, **J. Louradour**, **R. Collobert**, and **J. Weston**, Curriculum learning. *In Proceedings of the 26th annual international conference on machine learning*. ACM, 2009.

4. **Bombin, H.** and **M. A. Martin-Delgado** (2006*a*). Topological quantum distillation. *Phys. Rev. Lett.*, **97**, 180501. URL `https://link.aps.org/doi/10.1103/PhysRevLett.97.180501`.

5. **Bombin, H.** and **M. A. Martin-Delgado** (2006*b*). Topological quantum distillation. *Physical review letters*, **97**(18), 180501.

6. **Breuckmann, N. P.** and **X. Ni** (2018). Scalable neural network decoders for higher dimensional quantum codes. *Quantum*, **2**, 68.

7. **Chamberland, C.** and **P. Ronagh** (2018). Deep neural decoders for near term fault-tolerant experiments. *arXiv preprint arXiv:1802.06441*.

8. **Davaasuren, A.**, **Y. Suzuki**, **K. Fujii**, and **M. Koashi** (2018). General framework for constructing fast and near-optimal machine-learning-based decoder of the topological stabilizer codes. *arXiv preprint arXiv:1801.04377*.

9. **Delfosse, N.** (2014). Decoding color codes by projection onto surface codes. *Phys. Rev. A*, **89**, 012317. URL `https://link.aps.org/doi/10.1103/PhysRevA.89.012317`.

10. **Domingos, P.** (2012). A few useful things to know about machine learning. *Communications of the ACM*, **55**(10), 78–87.

11. **Glorot, X.** and **Y. Bengio**, Understanding the difficulty of training deep feedforward neural networks. *In Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010.

12. **Glorot, X.**, **A. Bordes**, and **Y. Bengio**, Deep sparse rectifier neural networks. *In Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011.

13. **Goodfellow, I.**, **Y. Bengio**, and **A. Courville**, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

14. **He, K.**, **X. Zhang**, **S. Ren**, and **J. Sun**, Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *In Proceedings of the IEEE international conference on computer vision*. 2015.

15. **Ioffe, S.** and **C. Szegedy** (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

16. **Jia, Z.-A.**, **Y.-H. Zhang**, **Y.-C. Wu**, **L. Kong**, **G.-C. Guo**, and **G.-P. Guo** (2018). Efficient machine learning representations of surface code with boundaries, defects, domain walls and twists. *arXiv preprint arXiv:1802.03738*.

17. **Karlik, B.** and **A. V. Olgac** (). Performance analysis of various activation functions in generalized mlp architectures of neural networks.

18. **Katzgraber, H. G.**, **H. Bombin**, and **M. A. Martin-Delgado** (2009). Error threshold for color codes and random three-body ising models. *Phys. Rev. Lett.*, **103**, 090501. URL https://link.aps.org/doi/10.1103/PhysRevLett.103.090501.

19. **Kingma, D. P.** and **J. Ba** (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

20. **Kotsiantis, S. B.** (2007). Supervised machine learning: A review of classification techniques.

21. **Krastanov, S.** and **L. Jiang** (2017). Deep neural network probabilistic decoder for stabilizer codes. *Scientific reports*, **7**(1), 11003.

22. **Krizhevsky, A.**, **I. Sutskever**, and **G. E. Hinton**, Imagenet classification with deep convolutional neural networks. *In Advances in neural information processing systems*. 2012.

23. **Maskara, N.**, **A. Kubica**, and **T. Jochym-O'Connor** (2018). Advantages of versatile neural-network decoding for topological codes. *arXiv preprint arXiv:1802.08680*.

24. **Sarvepalli, P.** and **R. Raussendorf** (2012). Efficient decoding of topological color codes. *Phys. Rev. A*, **85**, 022317. URL https://link.aps.org/doi/10.1103/PhysRevA.85.022317.

25. **Torlai, G.** and **R. G. Melko** (2017). Neural decoder for topological codes. *Physical review letters*, **119**(3), 030501.

26. **Varsamopoulos, S.**, **B. Criger**, and **K. Bertels** (2017). Decoding small surface codes with feedforward neural networks. *Quantum Science and Technology*, **3**(1), 015004.

CURRICULUM VITAE

1. Name : Chaitanya Chinni
2. Date of birth : 6th November 1995
3. E-Mail : chchaitanya95@gmail.com
5. Work Experience : PayPal India Development Center (2016)
   Software Development Engineer Intern