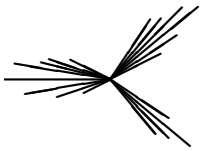


Instituto de Física Teórica
Universidade Estadual Paulista

DISSERTAÇÃO DE MESTRADO IFT–D.009/18

Machine Learning Quantum Error Correction Codes: Learning the Toric Code

Carlos Gustavo Rodriguez Fernandez

Orientador

Leandro Aolita

Dezembro de 2018

Rodriguez Fernandez, Carlos Gustavo
R696m Machine learning quantum error correction codes: learning the toric code / Carlos Gustavo Rodriguez Fernandez. – São Paulo, 2018
43 f. : il.

Dissertação (mestrado) - Universidade Estadual Paulista (Unesp), Instituto de Física Teórica (IFT), São Paulo
Orientador: Leandro Aolita

1. Aprendizado do computador. 2. Códigos de controle de erros(Teoria da informação) 3. Informação quântica. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca do Instituto de Física Teórica (IFT), São Paulo. Dados fornecidos pelo autor(a).

Acknowledgements

I'd like to thank Giacomo Torlai for painstakingly explaining to me David Poulin's methods of simulating the toric code. I'd also like to thank Giacomo, Juan Carrasquilla and Lauren Haywards for sharing some very useful code I used for the simulation and learning of the toric code. I'd like to thank Gwyneth Allwright for her useful comments on a draft of this essay. I'd also like to thank Leandro Aolita for his patience in reading some of the final drafts of this work.

Of course, I'd like to thank the admin staff in Perimeter Institute – in partiuclar, without Debbie not of this would have been possible. I'm also deeply indebted to Roger Melko, who supervised this work in Perimeter Institute (where a version of this work was presented as my PSI Essay). I thank the taxpayers of Ontario and CNPq for funding.

Resumo

Usamos métodos de aprendizagem supervisionada para estudar a decodificação de erros em códigos tóricos de diferentes tamanhos. Estudamos múltiplos modelos de erro, e obtemos figuras da eficácia de decodificação como uma função da taxa de erro de um único qubit. Também comentamos como o tamanho das redes neurais decodificadoras e seu tempo de treinamento aumentam com o tamanho do código tórico.

Palavras Chaves: Código Tórico; Correção de Erros Quãnticos; Aprendizado de Máquina

Áreas do conhecimento: Informação Quãntica ; Aprendizado de Máquina

Abstract

We use supervised learning methods to study the error decoding in toric codes of different sizes. We study multiple error models, and obtain figures of the decoding efficacy as a function of the single qubit error rate. We also comment on how the size of the decoding neural networks and their training time scales with the size of the toric code.

Keywords: Toric Code; Quantum Error Correction ; Machine Learning
Area of knowledge: Quantum Information; Machine Learning

Contents

Chapter 1

Introduction

The machine learning genie is out of the bottle[1]. These methods are useful for a diverse range of tasks: whether it's beating the top Go player in the world or automatically tagging gigabytes of videos every hour [1, 2], machine learning algorithms are fit for the job. Of course, many recent developments have conspired to the proliferation of machine learning in our daily life. Several new algorithmic techniques – many of them due to Geoffrey Hinton[2] – and technological developments have happened on par. More importantly, the sheer amount of data available nowadays have allowed for the training of machine learning models of scales never seen before. Of course, unless you are working for Google or Facebook, you might not have access to such data caches.

As physicists, we try to use every tool available to us to solve problems. And machine learning seems like the most beautiful and perfect hammer – at least, as long as you have sufficient data to train your modes! Well, data is the name of the game when you're doing measurements or simulations in many-body physics, and since Melko's 2016 paper [5], a lot of condensed matter and quantum physicists have jumped on the machine learning bandwagon. These last two years we've seen everything from learning to predict phase boundaries, to using neural network ansätze states, to turning full circle and using machine learning methods to make faster algorithms for Monte Carlo sampling [6, 7, 8].

Physicists have also studied machine learning techniques for decoding surface codes, a family of quantum error correction codes. These have been studied with both unsupervised [9] and supervised learning [10, 11, 12]. Quantum error correction codes are becoming

[1] As remarked by Roger Melko.

[2] Deep convolutional neural networks and restricted Boltzmann machines, just to name two, would be impractical without his contributions [3, 4].

more relevant as some near-term quantum devices are in development from many different companies. Moreover, these neural network decoders have a good flexibility to model many different kinds of errors – including correlated errors. This might might turn out crucial when the complete characterization of the error rate of the physical qubits is not feasible. In the present work, we expand on some of the results of [11] and [10] , utilizing architectures and methods inspired by [12].

Chapter 2

Brief review of supervised learning

In this essay we will focus on supervised learning, a subset of machine learning. The content of this section is mainly based on [13]. In supervised learning, we give the machine some training data, or a training set $\mathcal{D} = \{\{\mathbf{x}_i, \mathbf{y}_i\}\}_{i=1}^{|\mathcal{D}|}$. Each pair $\{\mathbf{x}_i, \mathbf{y}_i\}$ consists of an input configuration, $\mathbf{x}_i \in \mathcal{X}$, and its corresponding label, $\mathbf{y}_i \in \mathcal{Y}$. For example, each $\mathbf{x}_i$ could be (the RGB pixels of) a picture of a pet, and the $\mathbf{y}_i$ could be the species of the pet: $\mathbf{y}_i \in \{\text{cat}, \text{dog}, \text{pangolin}\}$. The training set is used to train the supervised learning algorithm to obtain good estimates of the labels, given the input data. That is, a supervised learning algorithm tries to obtain a function f that maps input configurations $\mathbf{x_i}$ into estimates of their labels, $\hat{\mathbf{y}}_i$:

$$f : \mathcal{X} \to \mathcal{Y}, \qquad \text{such that} \quad f(\mathbf{x}_i) = \hat{\mathbf{y}}_i \approx \mathbf{y}_i \quad \text{for all } i = 1, ..., |\mathcal{D}|, \tag{2.1}$$

This equation holds for both regression and classification problems, but it is only schematic. We need to do two things to properly define this learning problem. First, we'll give some structure to our label estimating function f. We will parametrize f by a set of parameters θ:

$$f(\mathbf{x}_i; \theta) = \hat{\mathbf{y}}_i. \tag{2.2}$$

The learning problem will then become finding the proper parameters θ that makes this labeling function f work the best. Now, we will also need to define some metric-like function acting on the label space, D, that satisfies:

$$D : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R},$$

$$D(\mathbf{y}, \mathbf{y}') \geq 0 \text{ for all } \mathbf{y}, \mathbf{y}' \in \mathcal{Y}, \text{ and}$$

$$D(\mathbf{y}, \mathbf{y}') = 0 \text{ if and only if } \mathbf{y} = \mathbf{y}'.$$

Note that we don't demand D to be symmetric nor to obey the triangle inequality. Our metric-like function D allows us to define a *cost function*, C:

$$C\left(\theta; \mathcal{D}\right) = \sum_{i=1}^{|\mathcal{D}|} D\left(\hat{\mathbf{y}}_i, \mathbf{y}_i\right). \tag{2.3}$$

An example of such a cost function is the cross entropy, which is used in classification problems.

With all these definitions in mind, the " $\approx$ " of Equation (2.1) refers to finding a set of parameters θ that minimize this cost function. When we do this minimization, we say we *train* the parameters θ :

$$\theta_{\text{trained}} = \arg\min_{\theta} C\left(\theta; \mathcal{D}\right). \tag{2.4}$$

However, this is not the whole story, as we've only seen what we do with the training set, $\mathcal{D}$. We also need a testing set, $\mathcal{D}_{test}$, to verify that we are not overfitting the data in the training set. The theory we've described works for both discrete labels (classification problems: is this image a cat or a dog?) or continuous labels (regression problems: what is the cost of this house?). For classification problems, we can readily define the training and testing accuracy of our classifying function f:

$$\textbf{Accuracy} = \frac{\#\text{ of correct classifications}}{\#\text{ of total classifications}}. \tag{2.5}$$

No matter how good the training accuracy is, we only really care about high testing accurazy, as that's the only way of knowing that a network has really "learned" the actual features of the data instead of just what each label of the training set is. The process of avoiding this overfitting – regularization – can be done in multiple ways. A prime example is adding a term to the cost function proportional to the sum of squares of some of the parameters of the network:

$$C(\theta, \mathcal{D}) = \sum_{i=1}^{|\mathcal{D}|} D\left(\hat{\mathbf{y}}_i, \mathbf{y}_i\right) + \alpha \sum_j \theta_j^2.$$

The art and craft of supervised machine learning is to find f, C and a method to train the parameters θ such that one can learn to label the data points while avoiding overfitting. We will see in the next section one of the most important parametrizations of f: the feed-forward neural network.

2.1 (Almost) the simplest neural networks:feed-forward neural networks for classification problems

To understand neural networks, we need to define their building blocks: layers, neurons, weights, biases and activation functions.

First, we will start with the very basic units of our neural networks, the *neurons*. Neurons map vectors $\mathbf{x} \in \mathbb{R}^n$, to real numbers:

$$a(\mathbf{x}; \mathbf{w}, b, g) = g\left(\mathbf{w}^T\mathbf{x} + b\right), \tag{2.6}$$

where $\mathbf{w} \in R^n$ and b are called the weights and biases of a neuron, respectively, and are the equivalent of the parameters θ that we mentioned in the previous section .g, the *activation function*, is some nonlinear real-valued function[1]. This single neuron is actually good enough of a classifier! If we choose $g(z) = \Theta(z)$, the Heaviside step function, this single neuron can actually work already as a fully-fledged classifying function $f(\mathbf{x}, \theta)$ to classify between two different labels. Such classifier is called a *perceptron*. Nowadays, people would use other activation functions, such as $tanh(z)$, the sigmoid function, given by

$$g(z) = \frac{1}{1 + \exp(z)},$$

or the rectified linear unit:

$$g(z) = ReLU(z) = \max(0, z).$$

These activation functions differ from $\Theta(z)$ in that they have non-vanishing derivatives (almost) everywhere, and allow for easier learning through gradient descent[2]. At the output layer of the network, it's common to use a *softmax* nonlinearity, defined as:

[1] We need nonlinearities because we will later on use outputs of previous neurons as inputs of new neurons. Without nonlinearities, all of this would just be a linear transformation of the input data.

[2] Gradient descent is a minimization algorithm for the cost function. We'll see more on this topic in Section 2.3.

$$g(z_i) = \frac{\exp(z_i)}{\sum_i \exp(z_i)}.$$

This softmax nonlinearity is used in classification problems, to have an output normalized to 1.

A single neuron is good enough to be a classifier by itself, so surely multiple neurons can do better![3] A neural network is a classifier composed of many of these neurons stacked together. Among such neural networks, we'll describe the ones in which the flow of information is in one way only, and in which the neurons themselves are neatly stacked in *layers*.

We will consider neural networks made out of L layers of neurons. The $l-$th layer will have n_l neurons, which we'll denote $\left\{a_i^{(l)}\right\}_{i=1}^{n_l}$, or more compactly in vector notation as $\mathbf{a}^{(l)}$. Each neuron has a properly-sized vector of weights and a bias. Thus, the layer of neurons $\mathbf{a}^{(l)}$ has a weight matrix $\mathrm{W}^{(l)}$ and a vector bias $\mathbf{b}^{(l)}$. If we furthermore define a $g^{(l)}(z)$ that acts component-wise on vectors[4],

$$\begin{aligned} &g^{(l)} : \mathbb{R}^n \to \mathbb{R}^n, \text{ where} \\ &g^{(l)}\left(\{x_1, x_2, ..., x_n\}\right) = \left\{g^{(l)}(x_1), g^{(l)}(x_2), ..., g^{(l)}(x_n)\right\}, \end{aligned}$$

we can define the output of each layer of neurons as follows:

$$\mathbf{a}^{(l)} = g^{(l)}\left(\mathrm{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\right), \text{ for } l = 1, 2, ..., L, \tag{2.7}$$

and where we have used the convention that $\mathbf{a}^{(0)} := \mathbf{x}_i$, the 0-th layer of our neural network will be the input data, $\mathbf{x}_i$.

Thus, we have defined our feedforward, fully-connected neural network[5], in which the classification function $f(\mathbf{x}, \theta)$ is given by the output of the last layer, and by Equation (2.7):

$$f(\mathbf{x}, \theta) = \mathbf{a}^{(L)} = \hat{\mathbf{y}}, \tag{2.8}$$

[3] We say this only partly in jest.

[4] We are assuming here that all the neurons in the same same layer use the same activation function. In practice, this is usually the case, and it certainly makes writing these equation an easier task.

[5] Feedforward because the "information" flows only in one direction, as the outputs of one layer become the inputs of the next layer. Fully connected because the weight matrices relate all the neurons of one layer to the next one (i.e. the weight matrices are not sparse by construction).

and the parameters θ are given by the set of weights and biases:

$$\theta = \left\{ \left\{ \mathrm{W}^{(l)} \right\}_{l=1}^{L}, \left\{ \mathbf{b}^{(l)} \right\}_{l=1}^{L} \right\}.$$

This is shown diagramatically in Figure 2.1. We now note that the minimization (or learning) procedure of Equation (2.4) will only optimize the parameters θ of the above equation. There are of course some other parameters that we've used to define our neural network: the number of layers L, the number of nodes per layer $\{n_l\}_{l=1}^{L}$, and the activation functions, $\{g^{(l)}\}_{l=1}^{L}$. This is the first example of the so-called *hyperparameters*, which are parameters that certainly define our network, but which are not minimized directly in the learning procedure described by Equation (2.4).

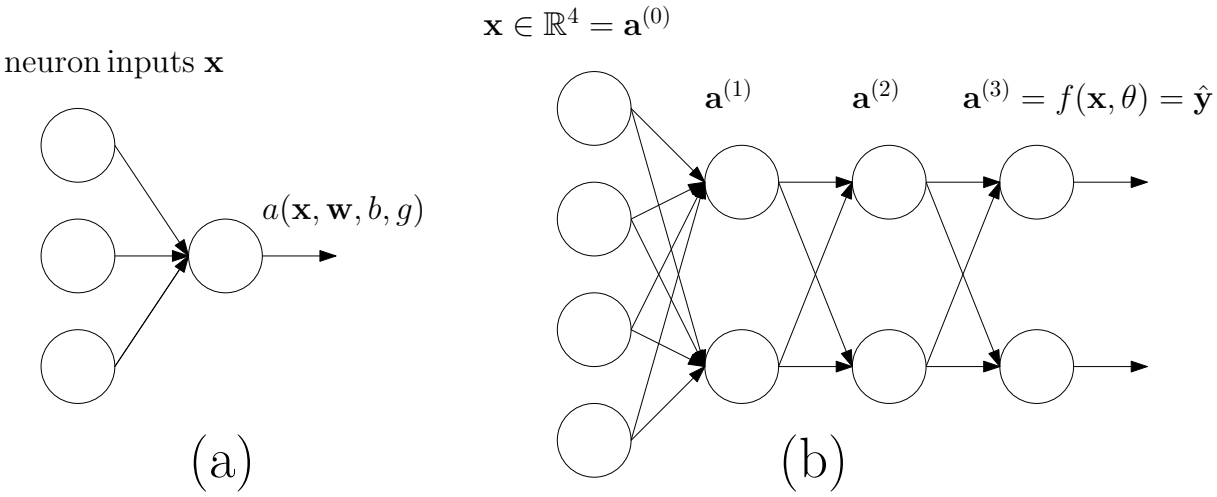


Figure 2.1: Diagrammatic notation for neurons and feed-forward neural networks. (a) A single neuron accepts inputs $\mathbf{x}$ and outputs a value $a(\mathbf{x}, \mathbf{w}, b, g)$. (b) Several neurons stacked together for a classifying function, $f(\mathbf{x}, \theta)$.

2.2 Neural networks that exploit locality: Convolutional neural networks

Fully-connected neural networks seem a bit wasteful, especially if the input data is from pixels or spins in a lattice. If you want your network to recognize faces (or learn a local Hamiltonian) then it doesn't make much sense to consider pixels from different corners

of an image (or spins that are not neighboring each other) to be input of a single neuron. If we want to exploit this locality (say, in a system of spins on a $2d$ lattice), it helps to represent the layers of the network to be themselves $2-$dimensional, and for the neurons in one layer to only obtain as inputs the layers or previous neurons. These *local weights* are the first ingredients of convolutional neural networks.

Convolutional neural networks go a step further, with the concept of *shared local weights*. Thus, a small block of local weights, say, a 3×3 matrix of weights $\mathrm{W}_{\mathrm{conv}}$, is used by all the neurons in a convolutional layer to obtain the input of the previous layer. It turns out that this is the discretized version of a convolution, which is where the name of the convolutional layers and convolutional networks come from. The details of how each convolution layer is created depeds heavily on the dimensions of these shared local weight matrices (also called the kernels or filters of the convolution). See Figure 2.2 for an example of a convolution layer employing a 3×3 filter.

Computing the convolutions with bigger filters also allows for *strides* – skipping over the input layer to obtain a convolution layer *smaller* than the input layer. Finally, we notice that one such filter $\mathrm{W}_{\mathrm{conv}}$ outputs a whole new layer of comparable size to the input layer. In practice, multiple such filters are used for one convolutional layer, so complete convolutional layer preserves both the (say) $2d$ structure of the input data, but now also has several *channels* of data in each lattice point - one for each filter. Suddenly the amount of data we have explodes! To handle this, there is a last ingredient of some convolutional neural networks: *pooling*. Pooling keeps the increasing amount of data of the convolution layers in control, by reducing the size of the layers. For example, max-pooling applied on a 10×10 layer might return a 5×5 layer, where every pixel of the pooled layer is the maximum of a 2×2 block of pixels of the input layer.

Convolutional neural networks usually have two parts: a series of convolution layers, each possibly followed by a pooling layer, and some fully-connected layers at the end. Introducing the convolution layers means that now there are new parameters to be trained: the convolution filters $\mathrm{T}_{\mathrm{conv}}$. The usual expectation is that training these filters will allow the network to do *feature detection* on the input data. For example, there are some image convolutions that allow for edge detection on pictures. Note, however, that employing convolutional layers also brings yet more hyperparameters to take into account: number of filters, their size, their strides, among others.

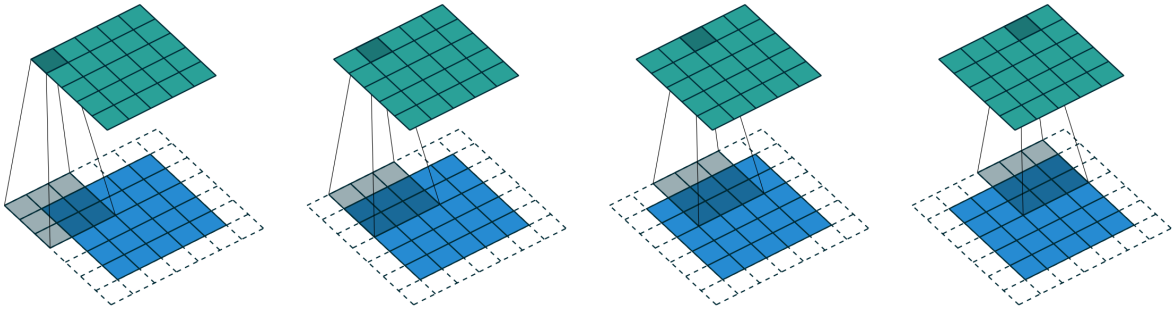


Figure 2.2: Input layer (blue) for a convolutional layer (green). Inputs and outputs are represented as squares. Each neuron of the convolutional layer is computed from acting a filter on 3×3 patches of the input layer. The dashed lines represent *padding* – data points added on the borders of the input layer to preserve the size of the layer after the convolution. In this essay, we will use padding that obeys periodic boundary conditions. Image taken from Figure 2.3 of [14].

2.3 Learning revisited: Training the networks and training the trainers

We have seen many ways to parametrize $f(\mathbf{x}, \theta)$ using neural networks. Now we should go back to Equation (2.4), which defines how we train our neural network via the minimization of a cost function, $\mathcal{C}(\theta, \mathcal{D})$ with respect to the parameters θ. The simplest way to do this would be to use *gradient descent*. This is an iterative method that requires an initial set of parameters, θ_0, and tries to find a local minimum of $\mathcal{C}(\theta, \mathcal{D})$. Because the gradient of a scalar function points towards the direction in which the function increases the fastest, the gradient descent method proposes the following updates to the parameters θ:

$$\theta_{i+1} = \theta_i - \alpha \nabla_\theta C(\theta, \mathcal{D}), \tag{2.9}$$

where θ_i is the set of hyperparameters in the i-th step of the algorithm, and α is the *learning rate*. Too small of a training rate and the training will be too slow, too fast of a training rate and the algorithm will jump around minima without converging. This learning rate is yet another example of a hyperparameter[6]. Algorithms like the Adam

[6]We'll find ways of dealing with this and other hyperparameters in the last part of this section.

optimizer [15] use versions of gradient descent with extra terms added to take into account moving-averages of gradients (i.e. a notion of momentum in the minimization). Of course, another good reason to use methods based on gradient descent is that our parametrization of $f(\mathbf{x}, \theta)$ allows for an efficient, analytical computation of the gradient.

We are now going to address again the problem of overfitting. We should intuitively expect that having all the θ parameters in our neural networks will make them very prone to overfit data. We have more tools to prevent overfitting than simply adding terms into the cost function. We will consider two stochastic methods to deal with overfitting, in a sort of *implicit regularization*: *stochastic gradient descent* (SGD) and *dropout*[7]. In SGD, every training step we randomly partition our training set into *batches*:

$$\mathcal{D} = \bigcup_{i=1}^{n_{\text{batches}}} \mathcal{D}_i. \tag{2.10}$$

Then, for every single step of gradient descent (2.9) we iterate over all batches:

$$\begin{aligned}
\theta_{i+1}^{(1)} &= \theta_i - \alpha \nabla_\theta C(\theta_i, \mathcal{D}_1), \\
\theta_{i+1}^{(2)} &= \theta_{i+1}^{(1)} - \alpha \nabla_\theta C(\theta_{i+1}^{(1)}, \mathcal{D}_2), \\
&\vdots \\
\theta_{i+1} &= \theta_{i+1}^{(n_{\text{batches}}-1)} - \alpha \nabla_\theta C(\theta_{i+1}^{(n_{\text{batches}}-1)}, \mathcal{D}_{n_{\text{batches}}}).
\end{aligned}$$

Dropout is even simpler to describe than SGD. When using dropout, for every batch in the training procedure, we define $f(\mathbf{x}, \theta)$ as before, but we randomly set to some neurons in the fully-connected layers to output zero (with probability p_{dropout}). This deceptively simple procedure, proposed by Hinton in 2014 [4], made the training of convolutional neural networks with several fully connected layers viable in the first place.

Now that we know most of the secrets of training a model, there's only one thing we have avoided talking about: choosing the optimal hyperparameters. The hyperparameters (as opposed to the parameters, θ) are not directly optimized by all the routines based on gradient descent we have talked about, because they either define the model itself, or the optimization procedure we use. For example, when using gradient descent, we need a

[7] Consensus on SGD being a form of implicit regularization is still uncertain. See [16, 17] for a discussion of SGD as implicit regularization. However, SGD has a very important role other than implicit regularization: dealing with training sets that are too big to be loaded on memory.

hyperparameter, α, that itself isn't chaned while the gradient descent routine runs. We can still, however, try to create an even "higher-order" cost function that depends on the hyperparameters $\Theta = \{p_{\text{dropout}}, \alpha, ...\}$, that we'll try to optimize to find a good set of hyperparameters. This cost function will simply be the testing accuracy of our trained $f(\mathrm{x}, \theta_{\text{trained}})$, which we introduced in Equation (2.5). That is, we measure the **Accuracy** in our testing set, $\mathcal{D}_{\text{test}}$. The process of obtaining optimal hyperparameters can then be defined as follows:

$$\Theta_{\text{optimal}} = \arg\min_{\Theta} \left[\textbf{Testing accuracy} \left(\Theta \right) \right]. \tag{2.11}$$

This optimization problem is quite different from the one we described in Equation (2.4), for two main reasons: There is no efficient way of computing gradients (and they wouldn't mean much, because many of the hyperparameters are integer-valued), and the function we are trying to minimize, $\textbf{Test accuracy}\left(\Theta\right)$, takes an enormous amount of time to evaluate compared to $\mathcal{C}(\mathbf{x}, \mathcal{D})$. It turns out that for optimization problems like this, where the target function is costly to evaluate, and when there is no practical way to compute gradients, we can still use a method called *Bayesian optimization* [18].

The main idea of Bayesian optimization is to relate the evaluations of the costly function as data points taken from a gaussian process model with parameters given by the hyperparametrs, Θ. A fitting of the gaussian process to all the data points seen so far is performed in every step of Bayesian optimiza tion, and the current best-fit of the gaussian process is then used to select a new data point, according to an acquisition function. With a gaussian optimization routine, we can do the minimzation of Equation (2.11), although, ironically, the Bayesian optimization algorithms themselves have their own hyperparameters. We will ignore the precise optimization of these.

Chapter 3

Brief review of quantum error correction

The following discussion is based on [19], although the same information can be found in other sources, such as [20]. Before talking about quantum error correction, let's review one of the simplest classical error correction codes: repetition codes. If we want to send a string of (classical) bits through a noisy classical channel that causes a bit-flip error with probability p, a simple way to handle the error is by repeting the information. Instead of sending the bit 1, send 111. Instead of sending the bit 0, send 000. That's the encoding – to decode back to the original bit string, we just take a majority vote. Thus, for example, if the encoded string 111 got one bit-flip while transported through the channel, it changes to 110, but the a're mostly 1, so they are decoded back to 1. By implementing this repetition protocol (where error decoding is done through majority), then the effective error probability of the channel becomes $O(p^2)$ instead of $O(p)$, for $p < 1/2$.

Quantum information is weird, fragile and hard to correct. Consider a two-level quantum system (say, a spin$-1/2$ particle), or qubit:

$$|\Psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,.$$

This qubit has two complex numbers encoded in it, α and β, which are encoded in the overall phase of the qubit and the relative phase between the basis vectors. A quantum error correction algorithm should protect this quantum information against the environment. This becomes extra hard when we also consider that making projective measurements on the qubit (to see if the environment has changed it) also destroys the

quantum information in it. Moreover, quantum information is special in that it can not simply be copied (remember the no-cloning theorem!) – so classical quantum error correction codes like repetition codes don't seem to be straightforwardly applicable here either.

The secret to making quantum error correction codes lies in encoding the information of the qubits you want to protect in a subspace of a higher-dimensional Hilbert space. In a way analogous to the repetition code above, it turns out that there exists a quantum error correction code appropriately called the 3-qubit code, that encodes the quantum information with an unitary operation defined as:

$$|0\rangle \to |0\rangle \otimes |0\rangle \otimes |0\rangle \quad \text{and}$$
$$|1\rangle \to |1\rangle \otimes |1\rangle \otimes |1\rangle \,.$$

Just like the bit-flip code, this code can correct a single σ_x error in the encoded qubits[1]. As we will be using Pauli operators a lot in this essay, we'll condense the notation:

$$\sigma_x \to X,\ \sigma_y \to Y,\ \sigma_z \to Z.$$

Also, as exemplified by this 3-qubit code, any (classical or quantum) error correction code is designed with a specific set of errors in mind. An example that we'll make use of in the following section is the spin-flip error: with probability p, apply an X operators to the state, and with probability $(1-p)$, do nothing. It is described by the following CPTP map:

$$\mathcal{C}(\hat{\rho}) = (1-p)I\hat{\rho}I + pX\hat{\rho}X \tag{3.1}$$

In this essay we will consider only error models with Pauli errors, that is, errors described by CPTP maps in which the Kraus operators are Pauli operators. A more precise definition of Pauli operators will be given in the next section.

[1] Using the notation and formalism of the next section, the stabilizer generators for this code are Z_1Z_2 and Z_2Z_3.

3.1 Stabilizer code formalism

So far, we have mentioned that the main idea behind quantum error correction consistits of encoding quantum information into the subspace of large Hilbert spaces. The qubits belonging to this large Hilbert space are called the *physical qubits*. Meanwhile, the qubits encoded in the subspaces of this Hilbert space (and which form the quantum information we want to transport across a noisy quantum channel) are called the *logical qubits*. We will denote the basis elements of the logical qubits by overbars. Moreover, e.g. $\left|\overline{0}\right\rangle$ and $\left|\overline{1}\right\rangle$. Moreover, we will call the Hilbert space spanned by these logical qubits, the *code space*. The quantum states in this code space will be called *codewords*.

A systematic way of encoding logical qubits into a larger set of physical qubits is through the stabilizer code formalism. The stabilizer codes are a family of quantum error correction codes that we can define on a Hilbert space of n physical qubits, $C^{2\otimes n}$. To better understand what we will do on this larger Hilbert space, we will first consider the Pauli group of one qubit, P_1, which is a group of operators (under multiplication) generated by $\{X, Y, Z\}$. From these, we can form P_n, the Pauli group for n qubits, by looking at the group of operators that can be obtained by tensor products of n copies of P_1. The operators in these groups will be called "Pauli operators" or simply "Paulis" in the rest of this work.

Now, let us take some commuting operators $\{s_i\}_{i=1}^{r}$ from the Pauli group of n qubits, and generate from them an abelian subgroup of P_n, $\mathcal{S}$. Furthermore, we are going to do this so that the abelian group $\mathcal{S}$ does *not* contain the operator $-I$. If we have done everything correctly so far, we should end up with a *stabilizer group* $\mathcal{S}$. We will call the s_i the generators of the stabilizer group, or simply generators if the meaning is clear enough from the context. Of course, to make things interesting we will demand $1 < r < n$, and that each generator of the stabilizer group has non-trivial support on at least 2 qubits (if they act on only one qubit, then that qubit will not contribute to error correction at all).

We mentioned before that quantum error correction works by encoding the information of some amount of *logical qubits* into subspaces of the physical qubits. If we go back to the definition of our stabilizer group, $\mathcal{S}$, we will notice something pretty interesting: all the generators square to the identity, and have eigenvalues ± 1. Our subspace of interest will then be the space in which all the generators of the stabilizer group have eigenvalue

+1. Demanding this from each of the independent[2] generators s_i halves the size of the Hilbert space, and thus this subspace will be isomorphic to $C^{2\otimes(n-r)}$. In other words, we will be able to encode a total of $l = n - r$ logical qubits in our n physical qubits, when using r independent generators for the stabilizer code. Moreover, we can find *logical Pauli operators* acting on these logical qubits – these are operators in P_n that commute with the generators, but are not themselves in $\mathcal{S}$. As an example, we can consider the 3-qubit code, which encodes one qubit into the subspace generated by Z_1Z_2 and Z_2Z_3. For this quantum error correction code there is just one encoded qubit, with logical Pauli operators $\overline{Z} = Z_1Z_2Z_3$ and $\overline{X} = X_1X_2X_3$. With this selection of logical operators, then we would define the logical qubits, of the 3−qubit code as:

$$\left|\overline{0}\right\rangle = \left|0\right\rangle \otimes \left|0\right\rangle \otimes \left|0\right\rangle ,$$
$$\left|\overline{1}\right\rangle = \left|1\right\rangle \otimes \left|1\right\rangle \otimes \left|1\right\rangle .$$

A systematic way of finding the logical qubits is straightforward: just start from an orthornormal basis in $C^{\otimes n}$ and project this basis over the +1 eigenvalue subspace of the stabilizer generators with projectors of the form $(1 + s_i)/2$. After this is done, some of the elements of the basis will be projected to zero, so we just take the ones that survive this projection to span our code space.

Now that we have the logical qubits, we can look at the codewords spanned by them, $|\Psi\rangle$. Because each logical qubit lies in the $+1-$ eigenspace of the $\{s_i\}_{i=1}^{r}$, then the following relation holds for all codewords:

$$s_i \left|\Psi\right\rangle = + \left|\Psi\right\rangle , \quad i = 1, ..., r. \tag{3.2}$$

Equation (3.2) encompasses the idea behind the use stabilizer codes for quantum error correction. If we encode information in a codeword $|\Psi\rangle$ and let some noisy quantum channel act upon it, we can use the stabilizer operators, $\{s_i\}_{i=1}^{r}$, to *check* whether we are still in the +1 eigenspace of these. Detectable quantum errors will be those that moves our original state $|\Psi\rangle$ outside of the code space. That is, there will be a set of s_i that, upon checking, will return a -1 eigenvalue.

[2]We want the generators s_i to be independent in the sense that each generator s_i can not be written as the product of some other generators.

Of course, not all operators (or errors!) acting on $|\Psi\rangle$ will take it away from the code space. There are two kinds of operators that will not take $|\Psi\rangle$ away from the code space: operators belonging to the stabilizer group, $\mathcal{S}$ (i.e. products of the generators s_i), and operators that *commute* with the generators s_i, but don't themselves belong to $\mathcal{S}$. This second kind of operators, in the language of group theory, belong to the quotient group $N(\mathcal{S})/\mathcal{S}$. This second kind of operators act non-trivially on the codewords themselves, while keeping them inside the wordspace. Thus, they furnish the *logical operators*.

There is another property of P_n we have not explicitly mentioned yet, but that will prove to be indispensable: elements of the Pauli group either commute or anticommute. Thus, we can readily check that if we start from a codeword $|\Psi\rangle$, and a Pauli error E that anticommutes with at least one s_i, we have:

$$s_i E |\Psi\rangle = -E s_i |\Psi\rangle = -E |\Psi\rangle . \tag{3.3}$$

This means that for any codeword Ψ, the Pauli errors E just defined will be *detectable* by looking at the eigenvalues that $E|\Psi\rangle$ has with the stabilizer generators. In order to detect the errors, we look at the *eigenvalues of the stabilizer operators*. We will call these values the *error syndromes*. The (classical) information of the syndromes can be then used to correct the errors in the encoded qubit, as long as we can detect the error that caused the specific syndromes. It turns out that if we want our stabilizer code defined by $\mathcal{S}$ to correct a set of errors $\mathcal{E}$, then we should demand that

$$E^\dagger F \notin N(\mathcal{S})/\mathcal{S} \text{ for } \text{ all } \ E, F \in \mathcal{E}, \tag{3.4}$$

where we disregard any overall phases. The appearance of two errors in Equation (3.4) suggests that we need not only to detect our error, but to reverse it in a non-ambiguous way. In spoken English, Equation (3.4) states that the quantum error, followed by its correction procedure, should not apply a logical operation on the original codeword.

Before ending the discussion on stabilizer codes, we will need to define the *weight* of a Pauli error $E \in P_n$: the number of qubits E acts non-trivially on. We can define the *distance* d of a quantum error correction code as the minimum weight of a Pauli error that causes Equation (3.4) to fail. In terms of usefulness, this means that a distance d code can perfectly correct any $\lfloor \frac{d-1}{2} \rfloor$ single-qubit Pauli errors. Conversely, to correct any t single-qubit Pauli errors, we need a code of distance $d = 2t + 1$. Finally, stabilizer codes usually use a compact notation, $[[n, k, d]]$. In this notation, n is the number of

physical qubits the code employs, k is the number of logical qubits the code has,and d is the distance of the code.

Now let us see an example of quantum error correction. We'll consider the 3-qubit code, in which a codeword goes through a spin-flip channel. A single X_1 error will make the syndrome data read $\{-1, +1\}$. A single X_2 error will instead make the syndrome data read $\{-1, -1\}$, and a single X_3 error will make the syndrome data read $\{+1, -1\}$. Thus, as long as we are dealing with single bit-flip errors, the 3-qubit code can recover the information from the syndrome data: simply reapply the X_i operator that caused the error syndromes.

3.2 Local stabilizers on a torus: Introducing the toric code

Some quantum error correction codes include stabilizers operators that act on many qubits at once, and these qubits might be physically apart. However, there is a family of stabilizer codes that employ stabilizers that only act on a small, local set of qubits. A particular kind of these codes are the so-called surface codes, in which the logical qubit are encoded in some non-trivial topological quantity. The first surface code ever proposed is the toric code, which was discovered by Kitaev [21].

Consider an $L \times L$ square lattice of dimensions with periodic boundary conditions. Now place qubits in every *edge* of the lattice, for a total of $N = 2L^2$ qubits. You can see this configuration in Figure 3.1 for a 4×4 toric code. We can define a stabilizer code on this arrangement of qubits with two different types of stabilizer generators for the stabilizer group. First, we will have the plaquette operators $\{B_p\}_{p=1}^{L^2}$, which will consist of products of the Pauli operators Z applied on the 4 qubits around each plaquette, or face, of the lattice:

$$B_p = \prod_{e \in \text{ plaquette } p} Z_e. \tag{3.5}$$

There will also be node operators, $\{A_n\}_{n=1}^{L^2}$, which will be given by the products of the Pauli operators X applied to each qubit adjacent to each node, or lattice point:

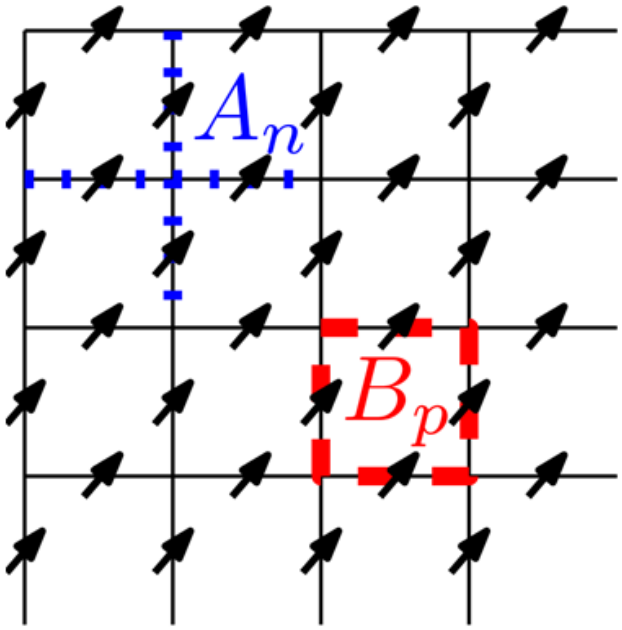


Figure 3.1: Spins on a 4×4 toric code. Notice that in an $L \times L$ toric code there are $2L^2$ spins sitting on the edges. There are two kinds of stabilzer operators on a toric code. The node operators, A_n, are the product of the X_e Paulis taken along the 4 edges around the node n. The plaquette operators, B_p, are the product of the Z_e Paulis taken along the 4 edges surrounding the plaquette p.

$$A_n = \prod_{e \in \text{ node } n} X_e. \tag{3.6}$$

These two kinds of stabilizer operators can be seen on Figure 3.1. One can easily check that the two different kinds of stabilizers commute with each other. Now, we have been a bit too generous with the stabilizers listed here. Because $Z^2 = X^2 = I$, we have the following relations:

$$\prod_{n=1}^{L^2} A_n = \prod_{p=1}^{L^2} B_p = I. \tag{3.7}$$

This last equation makes the total number of independent[3] stabilizer generators equal

[3]Remember, having a set of independent generators means that no generator can be written as a

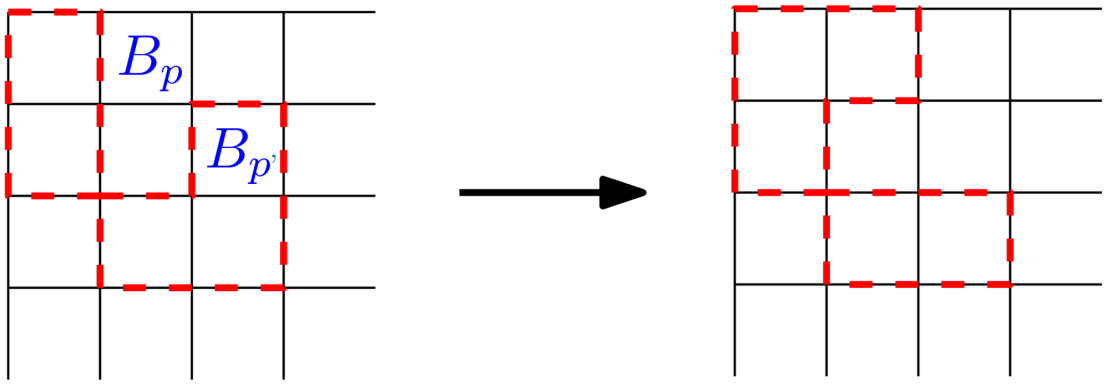


Figure 3.2: Action of B_p and $B_{p'}$ on an existing $Z-$loop. The action of B_p expands the uppermost loop. The action of $B_{p'}$ contracts the lowermost loop. Thus, the action of general plaquette operators contract or expand $Z-loops$. Note that the single B_p operators themselves are small $Z-$loops.

to $2L^2-2$, so out of the original $2L^2$ qubits, we are left with 2 logical qubits!

Before we go to describe the logical Pauli operators, consider the product of two adjacent plaquette operators: each B_p is like a small "loop" of Z Paulis acting on individual spins. Their product, however, is just a bigger loop, as $Z^2=1$. Thus, all such closed loops of Z operators are part of the stabilizer code, and bigger loops can be acted by the B_p operators until they contract and, eventually, disappear. An example of this is shown in Figure 3.2. This means that the products of B_p operators are (possibly disjoint) contractible loops of Z Paulis. However, the torus is famous for allowing for non-contractible loops! These correspond precisely to a couple of logical Pauli operators, $\overline{Z}_1$ and $\overline{Z}_2$. An analogous argument work for A_n operators, except that the natural chains of Xs looks different than the natural chains of Zs. [4]

The logical Pauli operators for the toric code correspond to the non-contractible loops of individual Pauli Xs and Pauli Zs on the torus. We can define logical Pauli operators $\overline{X}_1$ and $\overline{Z}_1$ acting on the first logical qubit, and Pauli operators $\overline{X}_2$ and $\overline{Z}_2$ acting on the second logical qubit. In Figure 3.3 we can see the logical Pauli operators on the toric code. Now, let's look a bit into the logical codewords of the toric code.

product of the others.

[4]To understand why this is the case, we should check the dual lattice, obtained by doing $A_n \leftrightarrow B_p$.

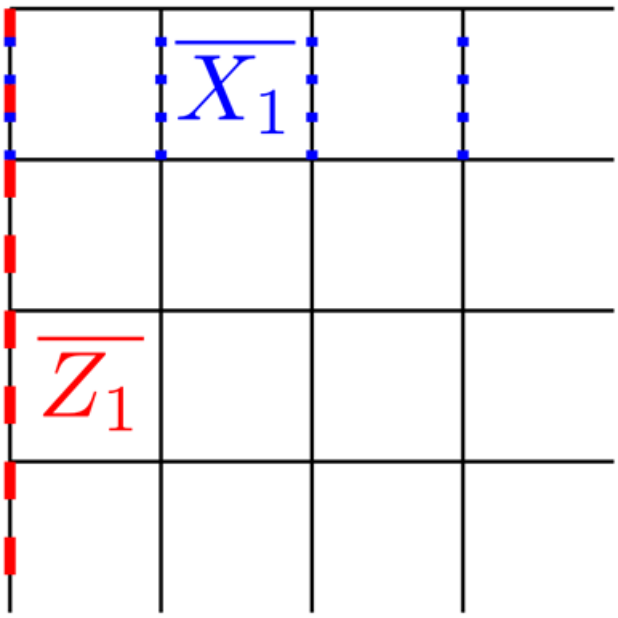


Figure 3.3: Logical operators on the toric code. $\overline{Z_1}$ is a vertical non-contractible loop of Z Paulis, and $\overline{X_1}$ is a horizontal non-contractible loop of X Paulis. This pair of operators act on the same logical qubit (as can be seen from the fact that they don't commute). Of course, we can easily define the corresponding $\overline{Z_2}$ and $\overline{X_2}$ operators.

3.3 Logical codewords of the toric code

We remember that the codewords of the toric code live in the subspace that has $+1$ eigenvalues with the generators of the stabilizer code. It will suffice to find one member of the basis of the logical qubits, as we can obtain the rest from logical Pauli operators. For this section only, we will use the X basis:

$$X_i \left|\rightarrow\right\rangle = \left|\rightarrow\right\rangle_i, \text{ and}$$
$$X_i \left|\leftarrow\right\rangle = -\left|\leftarrow\right\rangle_i.$$

So let us find this subspace! First, we define the following state:

$$\left|\rightarrow\right\rangle = \bigotimes_{i=1}^{L^2} \left|\rightarrow\right\rangle_i.$$

By construction, we have that

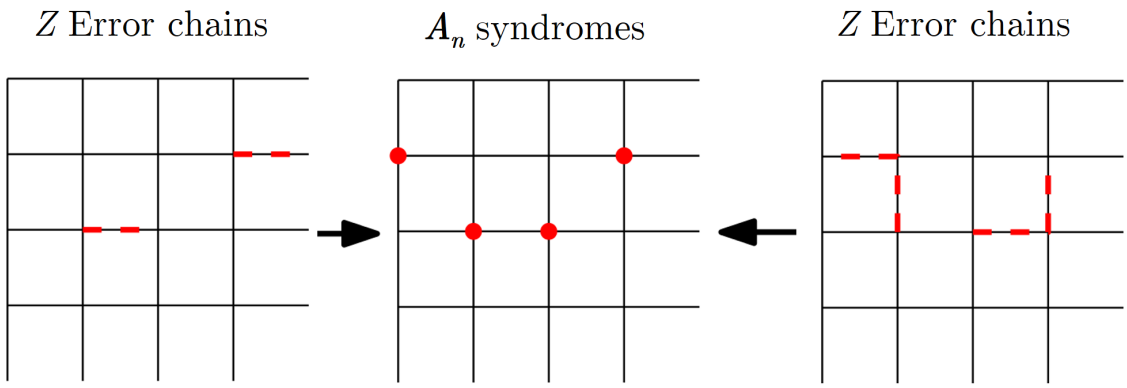


Figure 3.4: A_n error syndromes mark the end-points of Z error chains. This is a many-to-one map, so multiple inequivalent error chains can have the same error syndromes.

$$A_n \left|\rightarrow\right\rangle = \left|\rightarrow\right\rangle, \ \forall n. \tag{3.8}$$

For the next step, we just note that the operator $(B_p + 1)/2$ projects into the $+1$ subspace of B_p. Thus, we can obtain, modulo normalization, our $\overline{|0,0\rangle}$ logical qubit by first projecting our previous state into the code space (i.e., into the subspace which has $+1$ eigenvalues for all the generators of the stabilizer code):

$$\overline{|0,0\rangle} \propto \overline{P}_{0,0} \prod_{p=1}^{L^2-1} (1 + B_p) \left|\rightarrow\right\rangle = \overline{P}_{0,0} \left(1 + \sum_p B_p + \sum_{p \neq p'} B_p B_{p'} + ... \right) \left|\rightarrow\right\rangle . \tag{3.9}$$

In the equation above, after projecting our state into the code space, we apply the further apply the projector $\overline{P}_{0,0} = \frac{1}{4}(1 + \overline{Z}_1)(1 + \overline{Z}_2)$. This is the logical operator that projects states of the code space into the $\overline{|0,0\rangle}$ state. There is something very interesting going on in that last term. It turns out that the logical qubit states form an entangled state that is the sum of states with all the possible loops of Z operators acting on $\left|\rightarrow\right\rangle$.

3.4 Error correction in the toric code

When considering error correction in the toric code the notion of chains of Z's and X's is crucial. These are just adjacent products of Z or X operators that, if continued along a closed cycle, then become the loops which we talked about before (of both the contractible

and non-contractible kind). With this in mind, we can organize Pauli errors as *chains of errors*. Then, it turns out that there's a very nice geometric meaning to the position of error syndromes of the toric code: the error syndromes mark the end-points of chains of errors. A_n error syndromes mark the end-pont of Z error chains, B_p error syndromes mark the end-point of X error chains. See figure 3.4 for two examples of this.

Small enough error chains can thus be corrected by applying some extra chains of X and Z operators to turn the error chains into loops of Z and X operators, which, we recall, are part of the stabilizer group of the toric code. Of course, experimentally we'd only have access to the error syndromes - which themselves can be moved[5] by applying the corresponding Pauli operators. Error correction then becomes a process of bringing error syndromes of the same class together, which annihilates them. An example of this can be seen in Figure 3.5.

Figure 3.5 also shows how decoding in the toric code can fail. When dealing with homogeneous error models, an effective rule of thumb for decoding is maximum likelihood decoding. In the case of the toric code, this means that the proposed correction most likely to be correct is the one that introduces the *least* amount of Z or X chains in order to connect the syndromes. Keeping this in mind, it's clear why the decoding on the right-hand side of Figure 3.5 failed: it involved twice as many Z operators than the option on the left.

3.5 When decoding fails: Distance and error threshold of the toric code

Let us now discuss what the distance of the toric code is. If we have one short chain of, say, Z errors, then they can be readily corrected as there is a well-defined shortest-distance chain that joins the error syndromes. However, once a straight chain of errors has a length of $\lfloor L/2 \rfloor$ qubits, there is a problem. In this scenario, there is no way of knowing for sure if the correction procedure would should change the logical qubit – see Figure 3.6. From this heuristic argument, we can see why the toric code is a distance L code. Thus, the toric code is a $[[2L^2, 2, L]]$ code: it is a quantum error correction code employs L^2 physical qubits to encode 2 logical qubits, and is a distance L code.. We remember that this means

[5]One can think of a syndrome as a particle, and applying the corresponding X or Z operator extends or contracts the error chain so that the syndrome – the end=point of the error chain – moves.

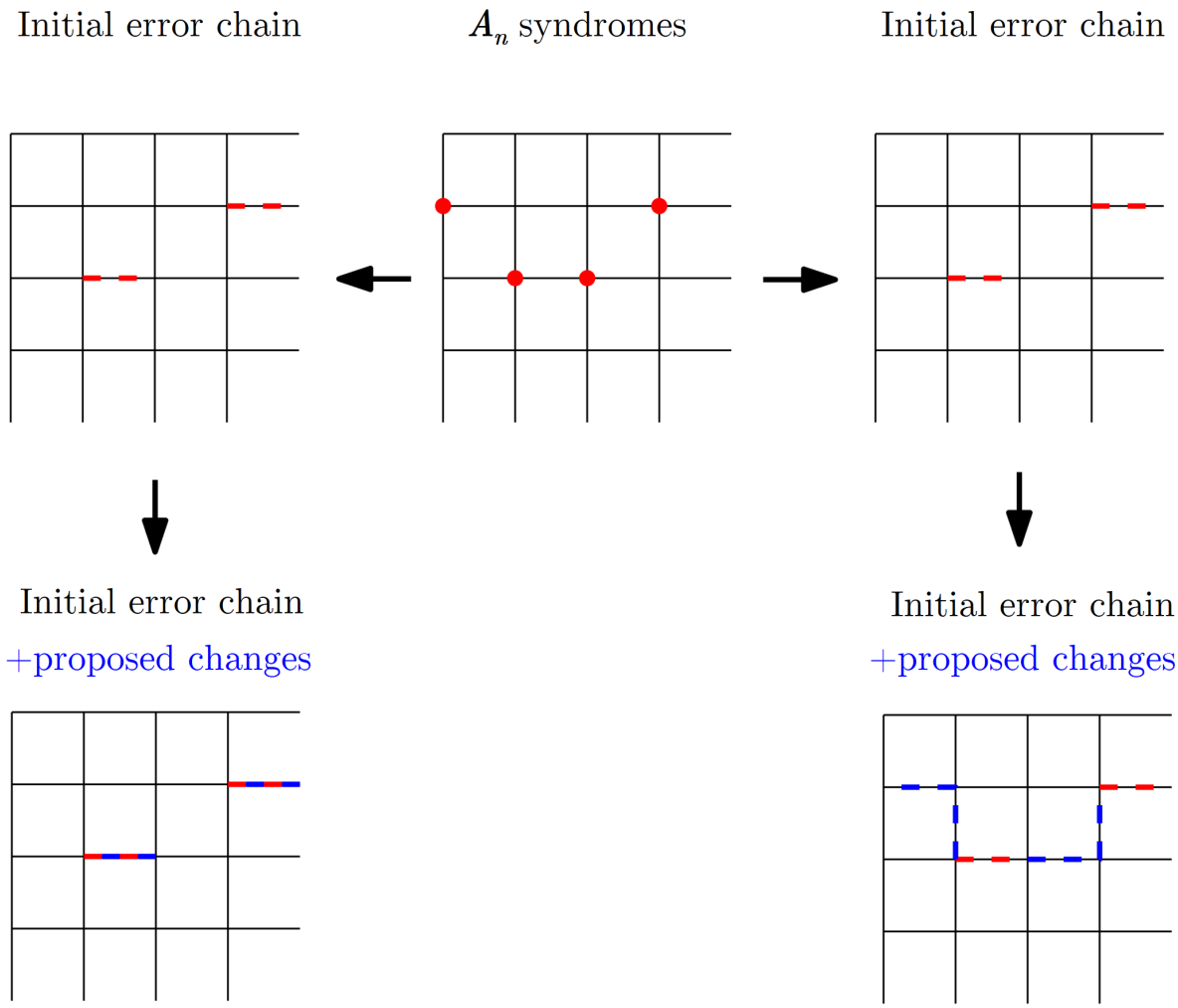


Figure 3.5: Error correction in a toric code. We show two ways of joining together the A_n syndromes. On the left, we see a correction that joins the syndromes using a total of 2 Z Paulis. On the right, we see a correction that joins the syndromes using a total of 4 Z Paulis. The correction on the left reverses the error correctly. The correction on the right introduces a logical operation on the original codeword.

that the toric code can decode arbitrary individual Pauli error affecting $\lfloor \frac{L-1}{2} \rfloor$ qubits.As a particular example, a 2×2 toric code is a terrible quantum error correction code: it can't correct any erorr – it can only detect them. Fortunately, the formation of such long error chains is exponentially supressed in local, homogeneous error models. The toric code is particularly good in those local, homogeneous erorr models: it turns out that if the error

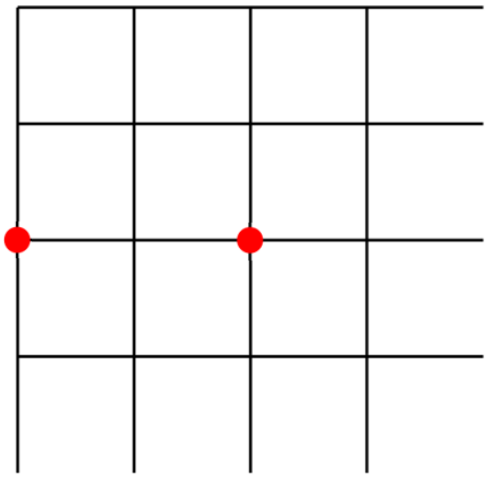


Figure 3.6: Simple example of error correction failing in a 4×4 toric code. The dots mark the position of A_n error syndromes. There is no way to decide among the 2 possible ways of joining the error syndromes with Z operators: inside the square or going around the periodic boundary. Thus, there is a 50% chance that decoding will fail against the error syndrome shown here.

rate is smaller than a certain threshold error rate, p_{th}, then a toric code can correct the typical errors as $L \to \infty$. This means that the toric code code is capable of correcting an infinite amount of errors, although while only encoding 2 logical qubits.

For some simple error models[6], this $p_{th} \approx 0.11$ can be obtained from the probability at the Nishimori point of the two-dimensional random-bond Ising model [22]. However, for more interesting[7] error models (and fault-tolerance scenarios - which we won't describe in this essay) the only way to estimate p_{th} is through direct simulation, which will be the topic of next section.

[6] In this case, perfect syndrome measurement and state preparation, and using the *spin-phase* error we'll describe in the next section.

[7] Any model that is not just a spin-phase error model. This includes some models studied in this work, but also might include some non-homogeneous error models.

Chapter 4

Simulating and learning error correction on the toric code

This section constitutes the main scientific contribution of the present work. To the author's knowledge, this is the first time that the simulation for the decoding of surface codes has been described in this level of detail in the existing literature. However, this is not new knowledge in the field – this is already part of the know-how of other workers in this field. The author is just writing down here what was, until now, mainly passed via word of mouth. The following simulation techniques were described to the author by [23]. These techniques have been alluded to before in [10, 12, 11], and are thus, not new in this field. In this work, we want to employ supervised learning on decoding errors with the toric code. First, we need to describe the labels we are going to use for the learning process. In order to do this, we will consider the following factorization of a Pauli error ε [10]:

$$\varepsilon = s\, c\, l, \tag{4.1}$$

where s is simply a member of the stabilizer group, c is a *pure error* – it's any operator that produces the measured syndrome –, and l is a logical Pauli operator. Similarly to Equation (4.1), we can use a factorization for the correction procedure we'd apply to the error: $\varepsilon' = s'\, c'\, l'$, in which the factorization is done in an analogous way. Of course, if we want this action on the physical qubits to apply a correciton, we should demand that $c' = c$, $l' = l$.

The trick for simulating the correction of the toric code relies on using the freedom of

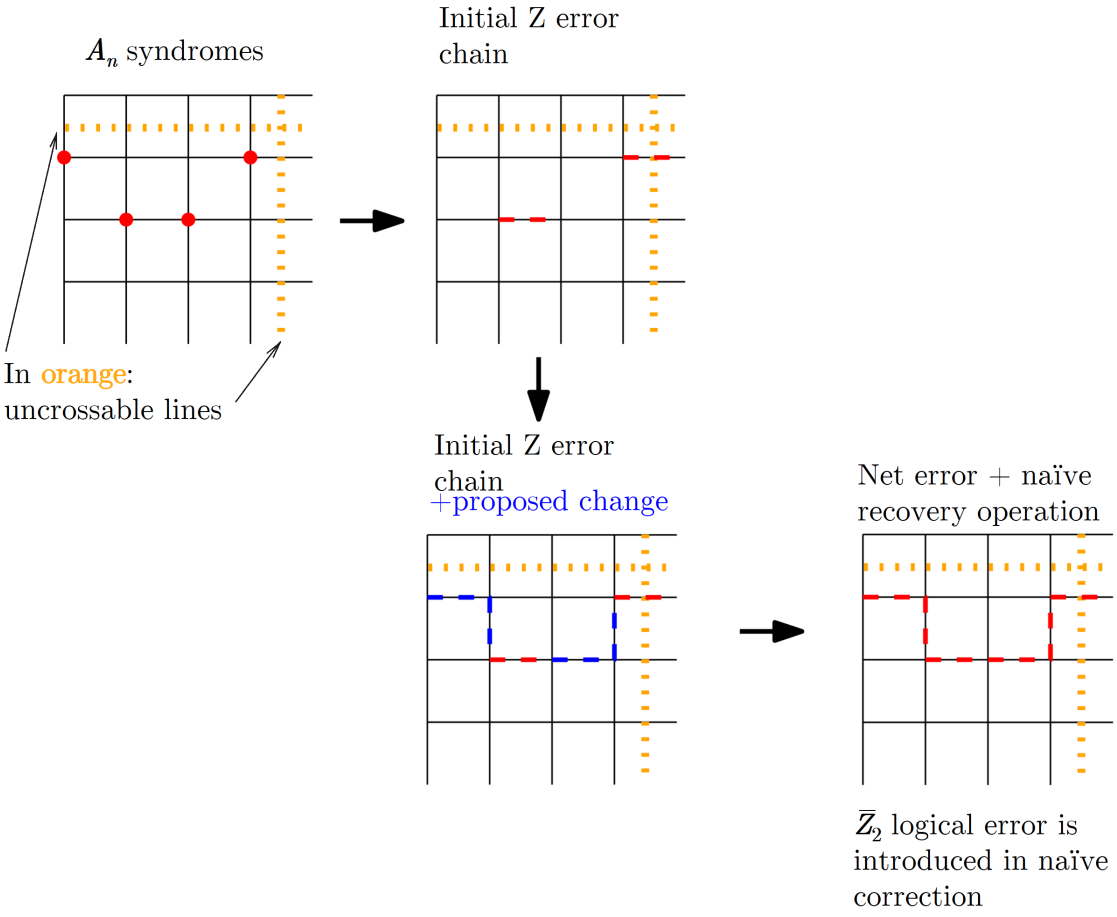


Figure 4.1: First step of the decoding: naive decoding under with uncrossable lines. In this example, the naive error correction creates an $\overline{Z_2}$ logical error on the codeword, when the initial Z error chain is considered. The next step in the decoding is continued in Figure 4.2.

choosing the s and s'. We use this "gauge freedom" to fix the position of the logical Paulis to fixed lines, which we'll call *gauge lines*, or uncrossable lines. Moreover, the algorithmic recovery of the errors will be done in two steps (shown in Figures 4.1 and 4.1):

1. Bring together all pairs of error syndromes at a single fixed point, while avoiding touching the corresponding gauge lines. First, we annihilate the A_n error syndromes, employing X Paulis to move them. Then, we annihilate the B_p error syndromes, employing Z Paulis to move them. An example of this step is shown in Figure 4.1.

2. Apply the necessary l logical operator to correct the error. First we apply the

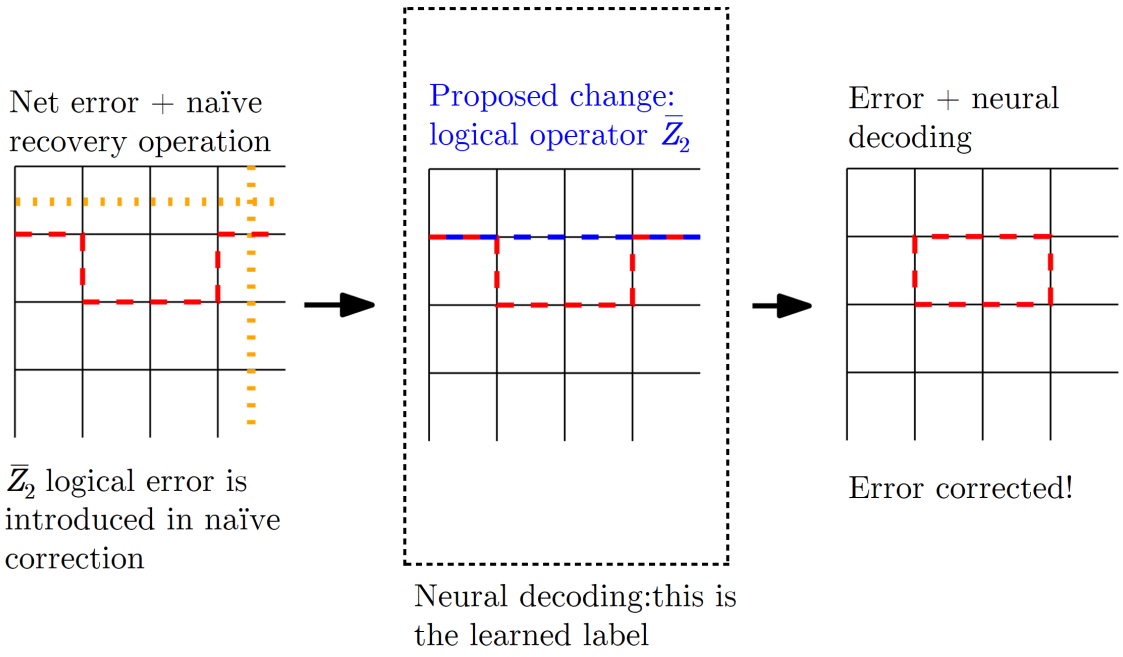


Figure 4.2: Second step of the decoding: neural decoding. In this step, we correct any logical operation introduced during the naive decoding. The choice of logical operation, l, needed to succesful do the error decoding is the label that is learned. In this example the final result includes a closed Z loop, which does not affect the original codeword.

neccesary Z_1 and/or Z_2 logical Paulis. Then we apply the neccesary X_1 and/or X_2 logical Paulis. An example of this step is shown in Figure 4.1.

The first of these steps requires no special computation, and while it might seem especially wasteful in the number of single-qubit quantum gates used ($\sim L$ per pair of syndromes), because $X^2 = Z^2 = I$, one can do a simple classical computation to count how many quantum gates are actually needed. It is a bit wasteful in single-qubit gates, but that's the price to pay for using our gauge-fixing. It is the second of these steps, the determination of l, that will be the focus of our study. This will be the label that we will use in our supervised learning. Our input data will be the syndromes of the errors, and the label will be one of the $4 \times 4 = 16$ possible values of l: the possible logical Paulis acting on the codewords.

4.1 Simulating the toric code: Conventions and tricks for efficient error simulation

Simulating error correction on the toric code turns out to be a simple task once we have fixed a convention for error recovery. So far, it seems that we still have a quantum system that we'd need to simulate, and that sounds awfully hard, especially after looking at the form of the logical qubit states in Equation(3.9). However, it turns out that we can just simulate everything as if we were dealing with a classical spin system. How so? Well, we only need to keep track of what syndromes the errors will change. Moreover, we will ignore the total phase of the qubits, so we will do all our multiplications modulo factors of -1 or i.

Let us consider the A_n generators. The corresponding syndromes only change sign under the effect of an Z or Y Pauli acting on any one of the 4 qubits of A_n. Similarly, the syndromes corresponding to the B_p stabilizers only change sign if one of the 4 qubits of B_p is acted on by a X or Y Pauli. Thus, noting that $Y = XZ$ in our convention, we only need to keep track of how many times X and Z operators have acted on the qubits starting from an error-free configuration in the wordspace. In practice, we can even *pretend that the individual qubits in a codeword have simulatenous eigenvalues* $+1$ *for both* X *and* Z operators, and according to this make all the corresponding calculations[1].

With this in mind, we will simulate our toric code as just a set of classical spins on the edges of a lattice with periodic boundary conditions, but each spin will have two numbers assigned to them, s_x and s_z, both of which can take values in $\{+1, -1\}$. With this in mind, we could readily assign eigenvalues to all the single-qubit Pauli operators, while simultaneously describing their action on flipping the spins: X flips s_z, Y flips both s_x and s_z and Z flips s_x. However, we can make our lives easier by only considering the spin-flipping aspects of these operators when simulating errors, and only considering the "having eigenvalues" aspect of them when obtaining the syndromes and when measuring the change of topological qubits on the gauge lines.

Thus, we can do our error simulations as follows:

1. We initialize all s_x and s_y to $+1$.

[1]We are going to simulate clasically the CPTP maps of the form of Equation (3.1), like coin flips, instead of quantum-mechanically, which would require keeping an overhead of some very big density matrices.

2. We simulate the Pauli errors, flipping the values of s_x and s_z.

3. After the errors have been simulated, we obtain the syndrome measurements by reading off the values of the stabilizer generators, treating s_x and s_y as eigenvalues of X and Y, respectively, but not flipping any spins with them (flipping would make no difference because of a gauge invariance that exists). We store these values of the syndromes for both A_n and B_p. The syndromes stored here are equivalent to the big dots in Figure 4.1 (big dots = -1 syndromes).

4. We measure the values of s_x and s_z along the logical operators in the gauge lines, without flipping any spins, and only reading the eigenvalues. From this, we can readily recover l, the logical error of Equation (4.1). It is a bit non-trivial, but the correspondence of this step with Figures 4.1 and 4.2 is that the initial error chains in the example shown the gauge lines an odd number of times. Thus, the naive decoding protocol *has* to introduce a logical $\overline{Z_2}$ error. That is, there is a direct correspodence between what is measured here and l.

Thus, at the end of the i-th classical simulation, we would obtain a pair of data points $\{\mathbf{x_i}, \mathbf{y_i}\}$ for our supervised learning algorithm, where $\mathbf{x_i}$ is the syndrome data, and $\mathbf{y_i}$ is the logical Pauli l which corresponds to the error that causes the syndrome. We have thus defined our supervised learning problem. In the following sections we explain the different error models we have used, as well as the specific neural network architectures we have employed.

4.2 Error models in the simulation

In the model we will study, we will assume that there are no errors due to measurement or state preparation. That is, all errors happen during the storage of the quantum information, and are caused by the environment[2]. The simplest example of such error is applying independent spin-flip errors of probability p on each spin. We described this error for a single spin in Equation (3.1). The classical version of this CPTP map[3] is the

[2]Errors in the preparation and in measurement can also be described, but we would need to look at the quantum circuits that define the projective measurements to describe "realistic" measurement error models.

[3]Meaning: instead of keeping track of the whole density matrix ρ after applying the CPTP map, we just flip coins according to the probabilities p.

simplest imaginable: we just move on every qubit in the simulation and act with X with probability p on each of them. We can similarly define a phase-flip error, which is like the spin-flip error, but using Z instead. Following the convention of [11], we can define a spin-/phase-flip error by combining these two error models - every qubit is affected by X with probability p, and independently by Z with probability p, following the CPTP map:

$$\mathcal{C}_{\text{spin-phase}}(\rho) = (1-p)^2 I\rho I + p(1-p)X\rho X + p^2 Y\rho Y + p(1-p)Z\rho Z. \tag{4.2}$$

We can also define a depolarizing error model, which with probability p applies one of the 3 non-trivial Pauli operators to a qubit, each selected with same probability:

$$\mathcal{C}_{\text{depolarizing}}(\rho) = (1-p)I\rho I + \frac{p}{3}X\rho X + \frac{p}{3}\rho Y + \frac{p}{3}Z\rho Z. \tag{4.3}$$

A final error model we will consider is the nearest-neighbor depolarizing error. In thhis error model, every neighboring pairs of qubits will have an error with probability p, given by one of the 15 non-trivial product of Paulis:

$$\begin{aligned}\mathcal{C}_{\text{NN-depolarizing}}(\rho) = (1-p)\,(I_1 \otimes I_2)\,\rho\,(I_1 \otimes I_2) + \frac{p}{15}\,(I_1 \otimes X_2)\,\rho\,(I_1 \otimes X_2) \\ + \frac{p}{15}\,(I_1 \otimes Y_2)\,\rho\,(I_1 \otimes Y_2) + \frac{p}{15}\,(I_1 \otimes Z_2)\,\rho\,(I_1 \otimes Z_2) + ...\end{aligned} \tag{4.4}$$

We will compare all these error models in this essay (extending the work of [11]), but before doing that, we need to standardize a definition of an effective error rate per qubit, p_{eff}. In short, $p_{eff}(p)$ will be the probability that a single qubit will change when the error model is used, given that the error model is defined by p as above. We have:

$$\begin{aligned}&p_{eff}^{\text{depolarizing}}(p) = p, \\ &p_{eff}^{\text{spin-phase}}(p) = 2p - p^2, \text{ and} \\ &p_{eff}^{\text{NN-dep}}(p) = \frac{2^4}{5}p - \frac{2^7}{5^2}p^2 + \frac{2^{12}}{3^2 5^2}p^3 - \frac{2^{14}}{3^3 5^4}p^4.\end{aligned}$$

$p_{eff}^{\text{NN-dep}}(p)$ is non-trivial: it is obtained from the recursion relation in equation (8) of [11][4].

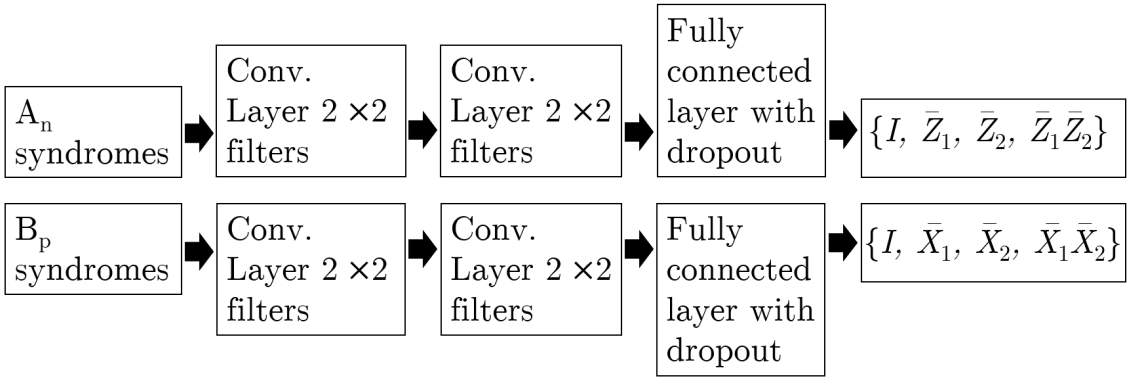


Figure 4.3: Architecture for the double convolutional neural network. ReLU nonlinearities are utilized after each convolution. *Softmax* nonlinearities are used in the output layers.The output from the last convolutional layer is the one that experieces dropout. Each of the two networks have its own cost function, and the total cost functio is the sum of the two cost functions. The **Testing Accuracy** of the network is measured in an all-or-nothing basis.

4.3 Convolutional neural networks for the toric code

A very important property of the toric code is that all the stabilizers are local. We exploit this locality by making manifest use of it in a convolutional neural network. The architecture of the convolutional neural network we'll use is the following: we'll use two different convolutional neural networks, one for each kind of stabilizer code. Thus, the A_n CNN determines the Z_1 and Z_2 part of the logical error, l, and the B_p CNN determines the X_1 and X_2 part of the logical error, l. Previous studies have used separations like these for the error syndromes [12] The architecture of this double CNN is shown in Figure 4.3.

[4]This recursion relation is obtained by taking into account how adding new nearest-neighbor might correct an existing error, or add a non-existing error.

Hyperparameter	**Min. value**	**Max. value**
Learning rate, α	10^{-7}	10^{-1}
Dropout probability	0.01	0.99
Number of neurons in fully connected layer	5	100
Number of filters in convolutional layers	5	50

Table 4.1: List of parameters that were optimized in the Bayesian optimization procedure. Optimization was done for each L separately, with the Depolarizing noise model, using $p_{eff} = 0.10$, 1.1 million data points. Adam optimizer was used, with default parameters except for the learning rate. A mini-batch size of 2000 was utilized throughout.

4.4 One trick for faster training and hyperparameter search

Because the toric code is self-dual (symmetric under $A_n \leftrightarrow B_p$), we can expect the weights in the A_n and the B_p networks to be interchangeable in some way[5]. By itself, this hints at a possible method for faster training of the CNNs: train only on one group of syndromes first for some iterations, copy the weights to the other network, and then train the joint network. We won't end up implementing this trick, but we'll do something similar for a really expensive aspect of training: hyperparameter search. The hyperparameter search we'll do will work on only one set of syndromes, and the optimized hyperparameters will be used on both the A_n and the B_p networks.

For hyperparameter search, we use the skopt package from the Scikit-Optimize Python library. This package employs Bayesian optimization for the hyperparameter search. The hyperparameters that we optimize for are given in Table 4.1, with the range of values taken. The values of hyperparameter that were not optimized over were not seen to have a big impact in the **Testing Accuracy**.

4.5 Estimating p_{th} via finite scaling

In the present work, we want to characterize the performance of our neural decoding algorithm working for different lattice sizes and error models. In order to achieve this, we define $p_{logical}$, the probability that a (logical) quantum error still exists after the decoding

[5]Under two caveats: (1) care in the implementation numbering, and (2) uncorrelated X and Z errors independent in each qubit.

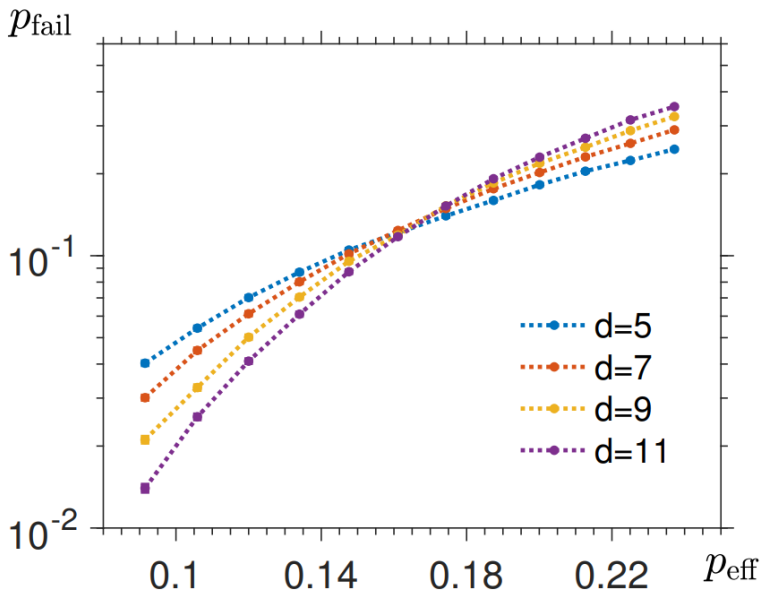


Figure 4.4: Expected finite scaling behavior of $p_{logical}(p_{eff})$. This is inset (c) of FIG 6. of [11]. In this figure, p_{fail} is what we denote as $p_{logical}$, and d is an increasing function of L. From this figure, we can estimate $p_{th} \approx 0.15$.

has been applied. We can define $p_{logical}$ directly from the **Testing Accuracy** of our trained neural networks:

$$p_{\text{logical}} = 1 - \textbf{Testing Accuracy}, \tag{4.5}$$

In our study, we train the neural networks from scratch, at values of $p_{eff} = \{0.01, 0.02, ..., 0.15\}$, and for each error model and lattice size ($L = 3, 4, ..., 8$), a total of 3 times each. We use training sets of 1000000 data points, and Testing sets of 100000 testing points. After training the networks, we want to obtain plots of $p_{logical}$ as a function of p_{eff} in order to estimate p_{th}. We remember now that the threshold probability, p_{th}, has the following definition: in the limit $L \to \infty$, the folowing relation holds[6] :

$$p_{logical}|_{L\to\infty}(p_{eff}) = \Theta(p_{eff} - p_{th}), \tag{4.6}$$

where Θ is the Heaviside step function. Of course, we won't have access to trained neural networks at large values of L that this limit becomes apparent. Instead, we will plot,

[6]Assuming that this limit exists, of course.

on the same graph, curves $p_{logical}(p_{eff})$ for different values of L, and use the value of p_{eff} at which the curves intersect. Additionally, for increasing values of L, the curves $p_{logical}(p_{eff})$ should be approaching the Heaviside step function. In particular, this means that for $p_{eff} < p_{th}$, we have that $p^{L}_{logical}(p_{eff}) < p^{L'}_{logical}(p_{eff})$ for $L > L'$. Alternatively, for $p_{eff} > p_{th}$ we have that $p^{L}_{logical}(p_{eff}) > p^{L'}_{logical}(p_{eff})$ for $L > L'$. An example of this expected finite scaling behavior can be seen in Figure 4.4, reproduced from [11].

Chapter 5

Results and Discussion

We used our neural decoding on toric codes of different lattice sizes ($L = 3, 4, ..., 8$) and with different error models. The plots of the probability of an uncorrected logical error as a function of the effective single-qubit error, $p_{logical}(p_{eff})$, are shown in Figures 5.1 to 5.3. To understand the use of these plots, the reader should check Section 4.5. On the captions of these figures, we write the estimates of p_{th} obtained from the intersection of the $p_{logical}(p_{eff})$ curves. Of course, if the limit indeed exists, this value should be independent of the parity of L. However, there might be a difference in the finite scaling of odd vs even lattice size L. This is why, for example, the authors of [9] use only even values of L for the unsupervised neural decoding of the toric code[1]. In all these Figures, we see an expected power law behavior for small p_{eff}[2], and we get an estimate of $p_{th} \sim 0.10$.

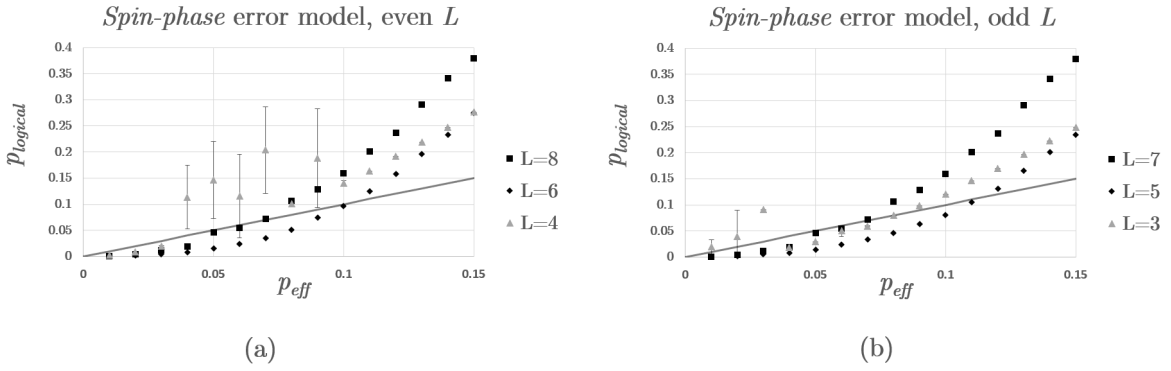
Two features that can be clearly seen from Figures 5.1 to 5.3 are: the very noisy behavior of the neural decoding (i.e. of the training of the neural networks) on small lattice sizes ($L = 3, 4$), and the suboptimal $p_{logical}(p_{eff})$ achieved for large lattice sizes ($L = 7, 8$). The first of this problems happens around low values of p_{eff}, and does not prevent us from estimating p_{th} from the intersection of $p_{logical}(p_{eff})$ curves with the two smallest lattice sizes. However, the suboptimal estimation of $p_{logical}(p_{eff})$ for large lattice sizes means that these curves are useless for the estimation of p_{eff}. We expect the $p_{logical}(p_{eff})$ curves with the largest lattice size to be consistently below the other curves, when $p_{eff} < p_{th}$, but this $p_{logical}^{L=7,8}(p_{eff})$ curve is consistently above the other curves after $p_{eff} \approx 0.05$. This can be explained by the use of a dataset not large enough to ensure the

[1] This statement comes from personal communication with the authors of [9]

[2] This is expected! For small p, quantum error correction tries to make the error probability go from $O(p)$ to $O(p^2)$.

proper learning of the neural network employed. More concretely, the **Testing Accuracy** achieved in the training of the neural decoders for $L = 7, 8$ is not sufficiently high to work as good neural decoders.

To study this last claim, we also performed an explicit check on how **Testing Accuracy** scales with the size of the training set, $|\mathcal{D}_{training}|$. This relationship is shown in Figure 5.4. This makes sense, because when learning to decode the toric code both the possible error syndromes and the number of possible error chains they correspond to, scale exponentially with the number of qubits. As seen in Figure 5.4, there is indeed a regime of $|\mathcal{D}_{training}|$ in which there is a linear relation between **Testing Accuracy** and $\log/, |\mathcal{D}_{training}|$. In the particular case of $L = 8$, this regime seems to hold up to at least $|\mathcal{D}_{training}| \approx 500000$. Only when these curves flatten, can we be certain that the model is, in some qualitatively sense, optimal.



(a) (b)

Figure 5.1: $p_{logical}(p_{eff})$ using the spin-phase error model for (a) even lattice sizes and (b) odd lattice sizes. The curves corresponding to $L = 7, 8$ are suboptimal for estimating p_{th} – see discussion. From the intersection of the two curves remaining curves, we estimate (a) $p_{th} \approx 0.15$ and (b) $p_{th} \gtrapprox 0.15$. See the discussion to see why this estimates here are not neccesarily the same. Keep in mind that this is a very rough estimates, as we only employ two curves for the finite scaling estimation.

For surface codes in $2d$, the question of scaling of the neural network has been addressed before in the literature. The authors of [11] implied that in neural decoders of surface codes using fully-connected neural networks, the following quantities scales exponentially with L: the number of layers, the number of neurons per layers, and the number of data used to train the networks. In the present work, we tried to avoid the exponential

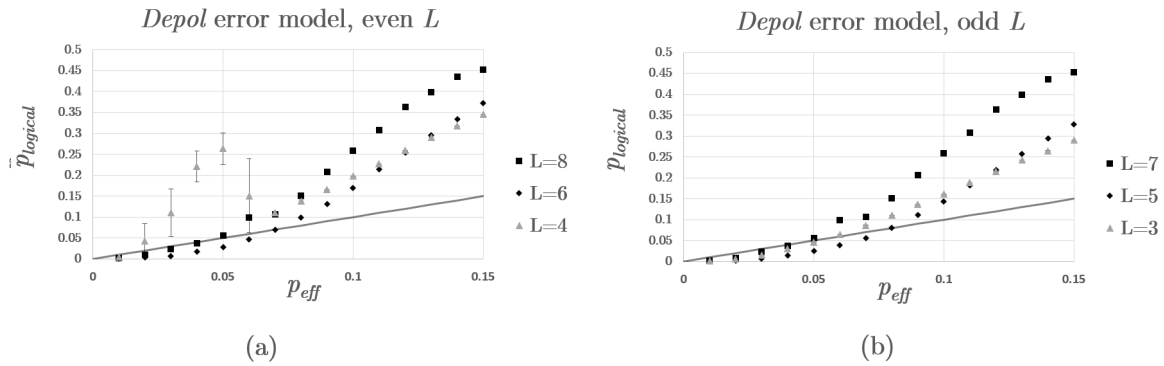
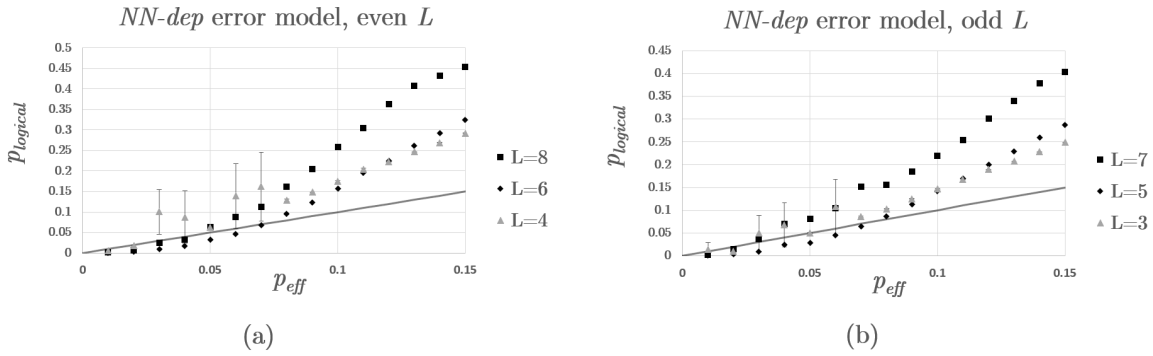


Figure 5.2: $p_{logical}(p_{eff})$ using the depolarizing error model for (a) even lattice sizes and (b) odd lattice sizes. The curves corresponding to $L = 7, 8$ are suboptimal for estimating p_{th} – see discussion. From the intersection of the two curves remaining curves, we estimate (a) $p_{th} \approx 0.12$ and (b) $p_{th} \approx 0.12$. Keep in mind that this is a very rough estimates, as we only employ two curves for the finite scaling estimation.

scaling of number of layers and number of neurons by using some convolutional layers. However, the neural networks still appear to need an amount of training data that grows exponentially with L, as can be seen from Figure 5.4. This means that neural decoding as described in this work would be extremely inefficient methods to decode $2d$ toric codes of large L. However, a key advantage of neural decoders lies in their generality with respect to error models. Had we used some position-dependent error models for the qubits, then neural decoding still works, while methods that assume homogeneity, such as MWPM [9] would not work. We expect that this versatility of neural decoders might transfer to some practical use when dealing with actual quantum devices, where variation in the fabrication of the qubits might lead to a wide variety of errors beyond the homogeneous ones.



(a) (b)

Figure 5.3: $p_{logical}(p_{eff})$ using the nearest-neighbor depolarizing error model for (a) even lattice sizes and (b) odd lattice sizes. The curves corresponding to $L = 7, 8$ are suboptimal for estimating p_{th} – see discussion. From the intersection of the two curves remaining curves, we estimate (a) $p_{th} \approx 0.12$ and (b) $p_{th} \approx 0.10$. See the discussion to see why this estimates here are not neccesarily the same. Keep in mind that this is a very rough estimates, as we only employ two curves for the finite scaling estimation.

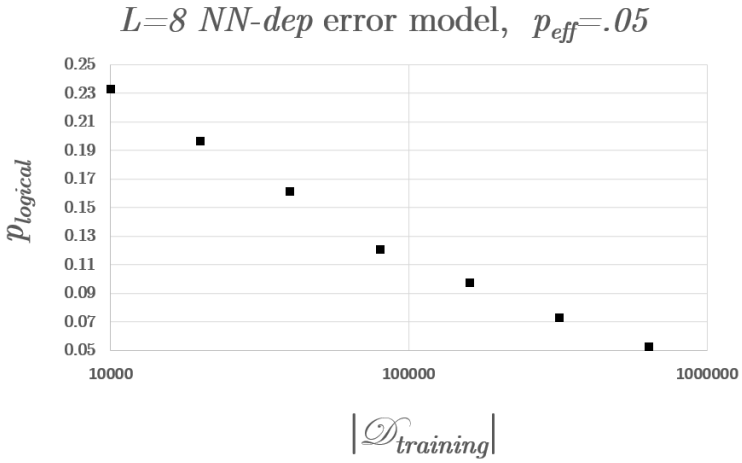


Figure 5.4: **Testing Accuracy** as a function the of size of the training set, $|\mathcal{D}_{training}|$, for neural decoding of nearest-neighbor depolarizing error model on an $L = 8$ toric code, for $p_{eff} = 0.05$. The main takeaway from this curve is that it has a nearly constant slope in a log-graph. More concretely, the **Testing Accuracy** increases linearly with $\log/, |\mathcal{D}_{training}|$. Of course, this does not go on forever, and this curve should flatten at some point. However, this flattening does not seem to be happening anywhere near 1000000 million data points, and a much bigger quantity should be necessary for an optimal training of the neural decoder.

Chapter 6

Conclusions

We have described the process of simulating errors in the toric code, and how to transform the decoding of those errors into a supervised learning problem. Because the toric code is self-dual and local, we were able to use a neural network that uses two convolutional neural networks to learn the decoding of the toric code – one for each syndrome type. Moreover, we use a clever error factorization technique to separate decoding into a naive step, which can be automated easily, and a step that requires learning on 16 different labels. After these simplificaitons, the decoding of the toric code is turned into a machine learning problem.

We managed to obtain an acceptable neural decoders for toric codes of length $L = 3$ to $L = 6$ for a first estimation p_{th}. We could not manage to obtain satisfactory neural decoders for $L = 7, 8$, as the training data sets used in this work were too small to ensure the required training accuracy. Having satisfactory neural decoders for these larger toric codes would allow us to have better finite-scaling estimates for p_{th}.

There are many topics which we could not cover in this work, but might be fruitful to consider in future work. For example, the features learned by the trained filters of the convolutional layers might give some interesting information on the process of decoding. In a surface code as symmetric as the toric code, maybe some of these trained weights might be reused for the training of larger toric codes. If that were the case, it could be an essential to study the neural decoding of larger surface codes, which was one of the biggest hurdles found in this work. Also, in a more forward-looking manner, it would be interesting to study the neural decoding of even more complex quantum error correction codes.

Bibliography

[1] C.-S. Lee *et al.*, "Human vs. Computer Go: Review and Prospect," ArXiv e-prints (2016).

[2] Pornhub, *Pornhub Launches AI-powered Model That Detects Over 10,000 Pornstars in Videos Using Computer Vision*, 2017-04-10, https://www.pornhub.com/press/show?id=1362.

[3] G. Hinton, "Training Products of Experts by Minimizing Contrastive Divergence," GCNU TR 2000-004 (2000).

[4] N. Srivastava *et al.*, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," Journal of Machine Learning Research **15** (June), 1929-1958 (2014).

[5] J. Carrasquilla and R. G. Melko, "Machine learning phases of matter," Nature Physics **13**, 431 (2017) [arXiv:1605.01735].

[6] Y.-H. Liu and E. P. L. van Nieuwenburg, "Discriminative Cooperative Networks for Detecting Phase Transitions," Phys. Rev. Lett. **120** 176401 (2018) [arXiv:1706.08111].

[7] Y. Huang and J. E. Moore, "Neural network representation of tensor network and chiral states," [arXiv:1701.06246v1].

[8] J. Liu *et al.*, "Self-learning Monte Carlo method," Physics Review B **95**, 041101 (2017) [arXiv:1610.03137].

[9] G. Torlai and R. G. Melko, "Neural Decoder for Topological Codes," Physical Review Letters **119**, 030501 (2017). [arXiv:1610.04238]

[10] S. Varsamopoulos, B. Criger, and K. Bertels, "Decoding small surface codes with feedforward neural networks," Quantum Science and Technology **3**, 015004 (2018). [arXiv:1705.00857]

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

[12] C. Chamberland and P. Ronagh, "Deep neural decoders for near term fault-tolerant experiments," Quantum Science and Technology, **4** 044002 (2018) [arXiv:1802.06441].

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

[14] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," [arXiv:1603.07285].

[15] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," [arXiv:1412.6980].

[16] R. Shwartz-Ziv and N. Tishby, "Opening the Black Box of Deep Neural Networks via Information," [arXiv:1703.00810].

[17] Z. Zhu *et al.*, "The Anisotropic Noise in Stochastic Gradient Descent: Its Behavior of Escaping from Minima and Regularization Effects,"[arXiv:1803.00195]

[18] E. Brochu, V. M. Cora, and N. de Freitas, "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning," [arXiv:1012.2599].

[19] D. Gottesman, *Surviving as a Quantum Computer in a Classical World* (Work in progress., 20XX).

[20] J. Preskill, "Lecture notes for ph219/cs219,", http://www.theory.caltech.edu/ people/preskill/ph229/.

[21] A. Y. Kitaev, "Fault-tolerant quantum computation by anyons," Annals of Physics **303**, 2 (2003) [arXiv:quant-ph/9707021].

[22] E. Dennis *et al.*, "Topological quantum memory," Journal of Mathematical Physics **43**, 4452 (2002). [arXiv:quant-ph/0110143]

[23] D. Poulin, Personal communication.