5-2024

Improving the Scalability of Neural Network Surface Code Decoders

Kevin Yipu Wu
William & Mary

Improving the Scalability of Neural Network Surface Code Decoders

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science with Honors in
Computer Science from the College of William and Mary in Virginia,

By

Kevin Y. Wu

Accepted for: Honors

Professor Qun Li

Professor Chi-Kwong Li

Professor Weizhen Mao

Williamsburg, Virginia
May 6, 2024

Decoding the Surface Code with Attention-based Neural Networks

by

Kevin Y. Wu

Submitted to the Department of Computer Science
on April 19, 2024 in partial fulfillment of the requirements for the degree of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

ABSTRACT

Quantum computers have recently gained significant recognition due to their ability to solve problems intractable to classical computers. However, due to difficulties in building actual quantum computers, they have large error rates. Thus, advancements in quantum error correction are urgently needed to improve both their reliability and scalability. Here, we first present a type of topological quantum error correction code called the *surface code*, and we discuss recent developments and challenges of creating neural network decoders for surface codes. In particular, the amount of training data needed to reach the performance of algorithmic decoders grows exponentially with the size of the quantum code, greatly limiting the applicability of this type of decoder. Here, we propose two approaches to this problem: the convolutional decoder and the transformer decoder. Using dimension reduction techniques, we can decrease the dependency on large data sets and accelerate the training process.

Our results demonstrate that the compression performed by the convolutional decoder may lessen data dependence and that the compression performed by the transformer decoder may accelerate training, which leads to promising directions for future work.

Thesis supervisor: Qun Li
Title: Professor of Computer Science

Acknowledgments

I would like to express my deepest gratitude to my thesis advisor, Professor Qun Li, for his guidance, patience, and support throughout the course of this research. His profound insights, constructive feedback, and unwavering belief in my abilities have been instrumental in shaping this work and fostering my growth as a researcher.

I am deeply indebted to my family and friends for their constant love, encouragement, and understanding during the challenging times of this journey. Their reassurance and moral support have been a source of strength, enabling me to persevere and overcome obstacles. I would also like to extend my heartfelt appreciation to Cordelia Li for her invaluable assistance in proofreading and providing insightful suggestions. Her support throughout this period has been invaluable to me.

Thank you all for your invaluable contributions to my journey.

Finally, I would like to acknowledge William & Mary Research Computing for providing computational resources and technical support that have contributed to the results reported within this thesis.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Quantum computing is an alternative computing paradigm that leverages quantum superposition and entanglement to accelerate computation. Quantum computers have the potential solve problems which are intractable for classical computers, providing quadratic to exponential speedups over classical computers for certain problems. Thus, quantum computers have the potential to revolutionize applications of computing in various fields, including cryptography, optimization, financial modeling, and machine learning. However, current quantum hardware is extremely limited, both in terms of noise and scalability. Quantum computers experience errors at a rate of about 10^{-3} per operation for superconducting quantum computers [1], and about 10^{-5} per operation for neutral atom quantum computers [2]. Since many of these quantum algorithms include millions of operations, the error rates in these quantum machines are far too high to be useful in many of these applications. These high error rates remain a prominent reason why quantum systems are not widely used in the present to accelerate computation.

To tackle this problem, we may apply *quantum error correction* techniques. Under the framework of quantum error correction, measurements of a *quantum error correction code* can used to deduce if errors occured in the system. This process of deduction is called *decoding*. In a physical quantum computer, a *decoder* usually takes the form of an algorithm running on a classical coprocessor, which reads in measurement results and outputs an error diagnosis. Afterwards, corrections may be applied to the quantum system or kept track of in classical control software. Such a decoder may take many different forms, including combinatorial algorithm or neural network, and exist in various computational mediums. Most importantly, the quality of a quantum computer's decoder will directly determine the system's capacity for fault tolerance.

Notably, an ideal decoder must have several properties:

1. **High Accuracy**: As the fault-tolerance of the quantum computer hinges on the accuracy of the decoder, an ideal decoder must be highly accurate.

2. **Low Latency**: Decoders operate in the liminal time between quantum operations. In current superconducting quantum architectures, this timeframe can be as short as 1μs.

If the decoder does not finish decoding before the next round of quantum operations, errors will accumulate and the computation will be ruined. Reducing the decoder's latency will allow it to keep pace with quantum operations.

3. **Multimodality**: Current quantum computers provide many readouts, all of which can be leveraged to perform error correction. An ideal decoder should be able to utilize all of this information to provide better decoding.

4. **Scalability**: An ideal decoder's accuracy should not suffer as the size of the quantum system and the density of physical errors increase.

Creating a decoder that meets these criteria remains an active research area. This work will utilize the product of recent advancements in deep learning to construct a robust decoder for the surface code that satisfies many of these requirements, while attempting to solve scalability issues with the neural network. The remainder of this chapter will provide a brief overview of the fundamentals of quantum information theory useful for quantum error correction, and how they correlate to quantum mechanics. Chapter 2 will give an overview of quantum error correction and the surface code, and Chapter 3 will give an overview of deep learning. Chapter 4 will discuss surface code decoders in the literature, along with the architecture and training for our proposed deep learning decoder. Finally, Chapter 5 will discuss the results.

1.2 Quantum Computing Basics

Qubits

In comparison to bits in classical computing, quantum computers use *qubits*, or quantum bits. Where the state of a bit represents high or low voltages in classical circuits, the state of qubits represent high and low energy states in systems with quantum properties. As we will discuss later, qubits have special properties, superposition and entanglement, that classical qubits do not possess. We may mathematically represent the state of a qubit as a complex unit vector $|\psi\rangle \in \mathbb{C}^n$, where $\mathbb{C}^n$ is a linear space of complex unit vectors, and $\mathbb{C}$ the complex numbers. Here, I use Dirac bracket notation to represent qubit states. I refer to $|\psi\rangle$ as a *ket vector*, and $\langle\psi|$ as a *bra vector*. They represent the following:

$$|\psi\rangle = \begin{pmatrix} \psi_1 \\ \vdots \\ \psi_n \end{pmatrix}, \langle\psi| = \begin{pmatrix} \psi_1 & \cdots & \psi_n \end{pmatrix}.$$

This ket vector representation corresponds to *pure states* in quantum mechanics.

Superposition

In 1804, Thomas Young conducted the double-slit experiment, where he showed that photons exhibit both wave-like and particle-like properties [3]. Through later experimentation, this

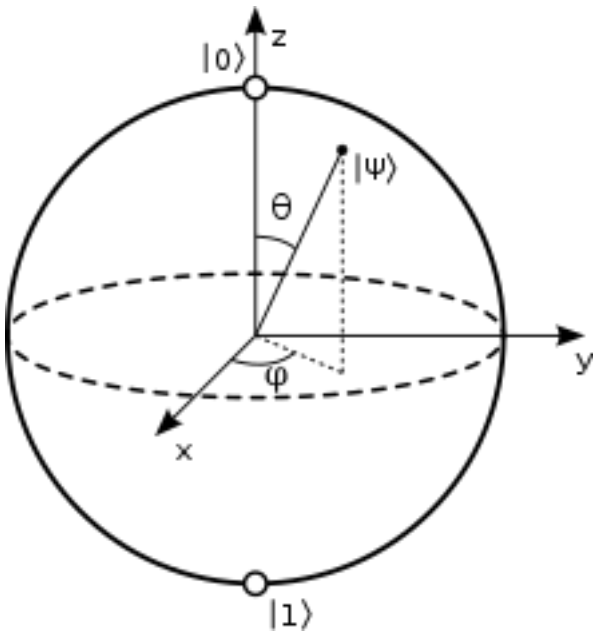


Figure 1.1: The Bloch Sphere

phenomenon was also shown to exist for atoms and electrons as well. Returning to quantum mechanics, these experiments highlight an important concept — superposition. This quality of quantum particles fundamentally distinguishes quantum physics from classical physics. In fact, it gives rise to the property of quantum particles being able exist in multiple states simultaneously. Here, in our mathematical framework, the state of a particle in superposition can be expressed as a convex combination of basis states, or vectors. We may define two basis states as follows:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Then, the state of a qubit in superposition, $|\psi\rangle$, can be expressed as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, for $\alpha, \beta \in \mathbb{C}$, with $\alpha^2 + \beta^2 = 1$. These complex α and β values are called *probability amplitudes*. Such a state $|\psi\rangle$ has a α^2 chance of being in state $|0\rangle$, and a β^2 chance of being in state $|1\rangle$.

Thenm the state of a qubit can be expressed as a point on the *Bloch Sphere*, displayed in Figure 1.1. The Bloch Sphere is a unit sphere, with each point on its surface representing the possible state of a qubit. The north and south poles of the sphere correspond to the basis states $|0\rangle$ and $|1\rangle$, respectively. Any other point on the sphere represents a superposition of these basis states. The position of a point on the sphere is defined by two angles: θ and ϕ. These angles give us the state of the qubit on the sphere as:

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi}\sin\left(\frac{\theta}{2}\right)|1\rangle,$$

where $\theta \in [0, \pi]$ is the polar angle measured from the z-axis, and $\phi \in [0, 2\pi]$ is the azimuthal angle in the x-y plane from the x-axis. As quantum states are indistiguishable under rotations corresponding to the roots of unity, we may assume that the probabililty amplitude for $|0\rangle$, α,

is real and nonnegative. Afterwards, there remain three degrees of freedom, which correspond to the three dimensions in the Bloch sphere. A simple parameterization yields the two angles.

The Bloch Sphere representation is useful because it provides an intuitive way to visualize the state of a qubit, and as we will see later, the operations (quantum gates) that change these states.

Entanglement and Multi-Qubit Systems

One of the main draws of quantum computing is the quantum parallelism that results from way quantum information is stored and processed across multiple qubits. The *tensor product*, denoted $\otimes$, is central to mathematically representing such quantum states. For two qubits, each represented by a Hilbert space $\mathcal{H}$, the combined system is described by the tensor product space $\mathcal{H} \otimes \mathcal{H}$. If a qubit is in state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ and a second qubit in state $|\phi\rangle = \gamma|0\rangle + \delta|1\rangle$, the joint system state $|\psi\rangle \otimes |\phi\rangle$ is given by:

$$|\psi\rangle \otimes |\phi\rangle = (\alpha|0\rangle + \beta|1\rangle) \otimes (\gamma|0\rangle + \delta|1\rangle) = \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle.$$

Such a composite system is *separable*, since it may be expressed as the tensor product of two single-qubit states. However, not all quantum states on multiple qubits have this property. Consider the Bell state, given below:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).$$

From the perspective of quantum information science, we may describe this state as *entangled*. Multiple qubits may be entangled together, producing a state which cannot be expressed as separate states. Instead, such a state must be described for the system as a whole. Entanglement is often produced by quantum operations that act on multiple qubits at once. Such a quantum operation effectively acts on a vector space of exponential size, $\mathbb{C}^{2^n}$. This greatly increases the computational power of a quantum computer when compared to a classical computer.

Quantum Operators

Quantum states may be manipulated and controlled via various quantum operations. Mathematically, these quantum operators may be expressed as unitary operators on a Hilbert space. Such a unitary operator, or matrix U preserves the inner product, i.e. $U^\dagger U = UU^\dagger = I$, where $U^\dagger$ is the conjugate transpose of U, and I is the identity matrix. An application of a quantum operator to a qubit is then matrix multiplication. Several key properties follow from this definition:

1. Reversibility: Since $U^\dagger U = I$, unitary matrices have an inverse, their conjugate transpose. This allows quantum operators to be undone by an application of their conjugate transpose.

2. Conservation of Quantum Information: Unitary matrices are Euclidean isometries, meaning that these quantum operators are transforms on qubits that preserve the

Frobenius norm. This means that the total probability in the quantum state remains 1 before and after application of such a unitary operator, reflecting conservation of quantum information.

These quantum operators serve as the mathematical foundation for *quantum gates*. By applying sequences of these unitary operations, we may maniplate qubits to perform more complex tasks.

The Pauli Group

The Pauli matrices are quantum operators that are used extensively in quantum computing. Each of these matrices represent a fundamental quantum gate that operates on a single qubit. They represent rotations of the qubit's state on the aforementioned Bloch sphere.

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

The X gate is analogous to the classical NOT operation. It flips the state of a qubit, performing a π-rotation about the x-axis of the Bloch sphere.The Y gate represents a phase shift of the qubit. Although it also flips the state of the qubit, it instead performs a π-rotation about the y-axis of the Bloch sphere. The Z gate affects a qubit's phase without changing its amplitude. It corresponds to a π-rotation about the z-axis of the Bloch sphere.

The Pauli group, denoted as $\mathcal{P}_n$ for a system of n qubits, is a group consisting of all possible tensor products of Pauli matrices, along with the identity matrix I. The elements of the Pauli group anticommute. For $\sigma_a, \sigma_b \in \mathcal{P}_n$, these elements have the anticommutation relation $\{\sigma_a, \sigma_b\} = \sigma_a\sigma_b + \sigma_b\sigma_a$. The Pauli group is involutory, each element is its own inverse.

Other Gates

Aside from the Pauli rotation gates, there exist some other widely used gates.

1. **Hadamard Gate**: The Hadamard gate is a single qubit gate denoted by H, creating superposition states from basis states. Its matrix representation is given below:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

   When applied to the state $|0\rangle$ or $|1\rangle$, it produces the states $(|0\rangle + |1\rangle)/\sqrt{2}$ and $(|0\rangle - |1\rangle)/\sqrt{2}$, respectively. Notably, conjugation of the X gate via the Hadamard gate produces the Z gate; $Z = HXH$.

2. **Phase-Shift Gate**: The phase-shift gate applies a phase shift ϕ to a qubit. It applies this phase ϕ to the $|1\rangle$ component of a quantum state, leaving the $|0\rangle$ component unchanged. Its matrix representation can be given below:

$$R_\phi = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}.$$

Some commonly used phase-shift gates include the $Z = R_\pi$ gate, the $S = R_{\pi/2}$ gate, and the $T = R_{\pi/4}$ gate.

3. **CNOT Gate**: The CNOT, or controlled-not gate applies the XOR operation conditionally to a qubit based on the state of another qubit. It is a two qubit gate with the following matrix representation:

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

The CNOT is very important, as it can create entanglement between two qubits.

Notably, the Pauli gates, along with the phase-shift gate and the CNOT gate form a viable set of universal quantum gates. This means that any quantum process can be approximated via a sequence of operations from this universal set. This gate set allows for the means to freely design quantum circuits.

Measurement

Quantum measurements follow the *Born rule*, which states that given a measurement operator M_m, the probability of obtaining some outcome m in a quantum state $|\psi\rangle$ can be computed by

$$p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle$$

Following the measurement, the state $|\psi\rangle$ projected to a new state $|m\rangle$, corresponding to the outcome m.

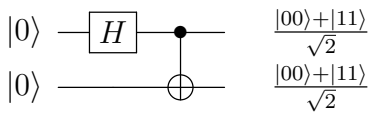
$$|m\rangle = \frac{M_m|\psi\rangle}{\sqrt{p(m)}}.$$

For example, take the state $|\gamma\rangle = \alpha|0\rangle + \beta|1\rangle$. When an identity measurement is performed on $|\gamma\rangle$, e.g. $|\gamma\rangle$ is measured under the measurement operator I, the result could either be 0 with probability α^2, resulting in a projection of $|\gamma\rangle$ to the $|0\rangle$ state, or 1 with probability β^2, resulting in a projection of $|\gamma\rangle$ to the $|1\rangle$ state.

Quantum Circuits

Under the gate-based quantum computing framework, the building blocks laid out in the previous subsections can be used together to create quantum circuits. The following is an example of a quantum circuit producing the previously mentioned Bell state. A register of qubits with states initialized to $|0\rangle$ are indicated on the left hand side of the circuit. Wires extend through the circuit. Gates are applied to these qubits, from left to right, with a gate drawn over a wire indicating an application of that gate to the corresponding qubit. Single qubit gates may be drawn as the single letter abbreviation for the gate, enclosed in a block.

CNOT gates are drawn with an $\oplus$ indicating the target bit, and a solid dot indicating the control bit. A measurement operation takes the form of a gauge drawn in a box.



1.3 Summary

In this chapter, we introduced some basics of quantum computing, setting the stage for understanding quantum error correction. We reviewed fundamental concepts such as qubits, superposition, entanglement, and quantum measurements, and discussed their mathematical representations. Additionally, we looked at the quantum circuit model, which is fundamental for constructing and understanding quantum algorithms and error correction schemes. Each of these elements plays an important role in quantum computing, and in the discussion of quantum error correction to come.

Chapter 2

Noise and Quantum Error Correction

As mentioned previously, qubits experience noise at a much higher rate than their classical counterparts. Therefore, to achieve useful, scalable quantum computation, these qubits have to be error corrected. This chapter lays out the fundamentals of quantum error correction requisite for understanding the surface code.

2.1 Mixed States

In many cases, mathematically expressing quantum states is not as simple as just assigning a single ket vector. Here, we may consider the use of a *density matrix*. For a pure state $|\psi\rangle$, its corresponding density matrix ρ is given by

$$\rho = |\psi\rangle\langle\psi|.$$

Such a density matrix is positive semi-definite, with its trace $\mathrm{Tr}(\rho)$ equal to 1. A *mixed state* then corresponds to a probabilistic mixture of these density matrices. Consider the following example:

2.2 Quantum Channels

Just as the pure state was inadequate for fully representing a more general quantum state, necessitating the introduction of the mixed state, the previously introduced unitary operator representation is also inadequate for representing quantum operations, especially in the presence of noise. Noise in a quantum operation may be throught of as a probabilistic deviation from the expected action. The resulting state could vary between applications of the same noisy operation. Motivated by this phenomenon, we introduce the *quantum channel*, which acts on mixed states, not just pure states.

Note that in quantum mechanics, the time evolution of a pure state in a closed, isolated quantum system under a time-independent Hamiltonian H is governed by Schrödinger's equation:

$$i\hbar\frac{\partial}{\partial t}|\psi(t)\rangle = H|\psi(t)\rangle \Rightarrow |\psi(t)\rangle = e^{-iHt/\hbar}|\psi(0)\rangle.$$

Here, the Hamiltonian is an operator corresponding to the total energy in the system. The time evolution of density matrices are similarly governed by the Liouville-von Neumann equation:

$$i\hbar\frac{\partial\rho}{\partial t} = [H, \rho].$$

Quantum channels, formally known as Completely Positive Trace-Preserving (CPTP) maps, are transformations that describe the evolution of the state of a quantum system, incorporating the effects of environmental interactions. These maps are defined to be completely positive, ensuring that they do not introduce any physical impossibilities into the state dynamics, and trace-preserving, which guarantees that the total probability distribution of the state remains constant. Importantly for quantum error correction, these CPTP maps capture the effects of various noise types and interactions that are not accounted for by simple unitary evolution.

The Bit-flip Channel

The quantum bit-flip channel is analogous to the classical bit flip in digital communications where a bit 0 might randomly flip to 1, and vice versa. In quantum computing, the bit-flip channel affects qubits by flipping their state from $|0\rangle$ to $|1\rangle$ and vice versa with a certain probability p.

The mathematical description of the bit-flip channel $\mathcal{E}$ applied to a quantum state represented by a density matrix ρ can be given by:

$$\mathcal{E}(\rho) = (1-p)\rho + pX\rho X$$

where X is the Pauli-X matrix. As mentioned previously, the X matrix swaps the amplitudes of the states $|0\rangle$ and $|1\rangle$, effectively flipping the state of the qubit. The probability p represents the likelihood of the bit-flip error occurring, while $1-p$ represents the probability that the state remains unchanged. Note that we may also define the phase-flip and bit-phase-flip channels accordingly, for the Z and Y gates replacing X, respectively.

The Depolarizing Channel

The depolarizing channel represents a type of noise that uniformly randomizes the state of a qubit, driving it towards a completely mixed state. This channel is characterized by its simplicity and symmetry, making it widely used in theoretical studies of quantum communication and computation.

For a single qubit, the depolarizing channel can be described by the equation:

$$\mathcal{E}(\rho) = (1-p)\rho + \frac{p}{3}(X\rho X + Y\rho Y + Z\rho Z)$$

where p is the probability of depolarization occurring within a given time interval, and the X, Y, and Z Pauli matrices. The effect of this channel is to replace the state ρ with the maximally mixed state $\frac{I}{2}$ with probability p, while the system remains in its original state with probability $1-p$. The addition of $\frac{p}{3}$ to each Pauli operation ensures that the trace of ρ

is preserved, adhering to the properties of a completely positive and trace-preserving map. This channel thereby introduces isotropic noise to the system, reducing the qubit's purity regardless of its initial state.

2.3 Stabilizer Codes

Detecting quantum errors is difficult. There are several properties of quantum states which complicate the process, preventing a direct application of classical error correction techniques.

1. The No Cloning Theorem — General quantum states cannot be copied through a single general unitary process, due to a result called the No-Cloning Theorem [4]. This precludes the classical technique of redundancy, as you cannot copy the quantum states holding data. Thus, any effective quantum error correction process must be able to detect and correct errors in a quantum system without copying any of the constituent quantum states.

2. Projective quantum measurements — Quantum measurements "destroy" the quantum state, producing a basis state as the measurement result, and projecting the measured quantum state into that basis state. This rules out directly measuring qubits, as it would destroy the data they hold. Therefore, any effective quantum error correction process cannot directly measure qubits holding data as you would do to their classical counterparts in classical error correction.

A stabilizer code [5] is a type of quantum error correction code that successfully circumvents these obstacles. Stabilizer codes consist of an arrangement of physical qubits holding data, along with extra physical *ancilla qubits*. These physical qubits, data and ancillae, are entangled together according to the design of the code. The purpose of these ancilla qubits are to give a feasible way to detect errors without measuring any qubits holding data. In a stabilizer code, measurement of these ancilla qubits produce a *syndrome measurement*, the parity of which can be used by a decoder to produce error corrections [6].

Stabilizers

Central to stabilizer codes is the *stabilizer*. I begin our discussion of stabilizers with an illustrative example, first presented in [6]. Consider a two qubit system with qubits a and b. For convenience, I define the following operations:

$$Z_a = Z \otimes I, Z_b = I \otimes Z,$$
$$X_a = X \otimes I, X_b = I \otimes X.$$

First we note that operations on different qubits commute. For X_aX_b and Z_aZ_b, we have:

$$\begin{aligned}[X_aX_b, Z_aZ_b] &= (X_aX_b)(Z_aZ_b) - (Z_aZ_b)(X_aX_b) = X_aZ_aX_bZ_b - Z_aX_aZ_bX_b \\ &= (-Z_aX_a)(-Z_bX_b) - Z_aX_aZ_bZ_a = 0.\end{aligned}$$

Since commuting matrices share an eigenvector, there must exist a simultaneous eigenstate $|\psi_{ab}\rangle$ for both X_aX_b and Z_aZ_b. Under these two operator products, this simultaneous eigenstate is unique. This implies that repeatedly measuring $|\psi_{ab}\rangle$ under both the X_aX_b and Z_aZ_b operators will not change the state. Without loss of generality, and for simplicity, let $|\psi_{ab}\rangle$ be the $(+1,+1)$ eigenstate for X_aX_b and Z_aZ_b. That is, $|\psi_{ab}\rangle$ has eigenvalues of $+1$ for both X_aX_b and Z_aZ_b. Now, the action of a depolarizing channel applying an extraneous X_a gate to $|\psi_{ab}\rangle$. Then, we have the following:

$$\begin{aligned} X_aX_bX_a|\psi_{ab}\rangle &= X_aX_aX_b|\psi_{ab}\rangle = X_a|\psi_{ab}\rangle, \\ Z_aZ_bX_a|\psi_{ab}\rangle &= -X_aZ_aZ_b|\psi_{ab}\rangle = -X_a|\psi_{ab}\rangle. \end{aligned}$$

Notice that this external X error changes $|\psi_{ab}\rangle$ from a $(+1,+1)$ eigenstate to a $(+1,-1)$ eigenstate of X_aX_b and Z_aZ_b, flipping the eigenvalue for Z_aZ_b. We may also consider the same example for a Z error applying an extraneous Z_b gate to $|\psi_{ab}\rangle$. As before, we have the following:

$$\begin{aligned} X_aX_bZ_b|\psi_{ab}\rangle &= -Z_bX_aX_b|\psi_{ab}\rangle = -Z_b|\psi_{ab}\rangle, \\ Z_aZ_bZ_b|\psi_{ab}\rangle &= Z_bZ_aZ_b|\psi_{ab}\rangle = Z_b|\psi_{ab}\rangle. \end{aligned}$$

In both of these cases, this induced change resulting from the external bit-flip flips the sign of one of the two eigenvalues of $|\psi_{ab}\rangle$, modifying the final measurement result. Repeatedly measuring a complete set of these operator products will allow us to determine whether or not an error has occured on these two qubits. This is quite useful. However, if the set of operator products is not complete, the simultaneous eigenstate of these operator products is not guaranteed to be unique.

In the context of quantum error correction, these X_aX_b and Z_aZ_b operator products are known as *stabilizers* for qubits a and b. As we can see, applications of these stabilizers have the potential to facilitate error correction, but there are some caveats. Note the effect of a depolarizing X error applying a X_b:

$$\begin{aligned} X_aX_bX_b|\psi_{ab}\rangle &= X_bX_aX_b|\psi_{ab}\rangle = X_b|\psi_{ab}\rangle, \\ Z_aZ_bX_b|\psi_{ab}\rangle &= -X_bZ_aZ_b|\psi_{ab}\rangle = -X_b|\psi_{ab}\rangle. \end{aligned}$$

This maps our eigenvector from a $(+1,+1)$ eigenstate to a $(+1,-1)$, which is exactly the same outcome as the previous example applying an erroneous X_a! This implies that our setup of stabilizers is not enough to completely determine the errors that aflict this system.

In this example, even though the stabilizers, specific operator products, allow us to detect some errors, the simple setup involved does not allow us to correct them. True quantum error correction will require a more complex setup.

Stabilizer Formalism

We may now formally define stabilizers and stabilizer codes. I will use the definition given by Gottesman [7]. A stabilizer code $[n,k]$ encodes k logical qubits into n physical qubits. The stabilizers of such a code form an Abelian subgroup $\mathcal{S}$ of the Pauli group $\mathcal{P}_n =< I, X, Z >$ on n qubits. Recall from the example in the previous section that the commutation of the

stabilizers allows for simulatenous eigenstates, which is useful for error detection. Therefore, $\mathcal{S}$ should be Abelian. The codeword $|\psi\rangle$ for such a stabilizer code, residing in a Hilbert space $\mathcal{H}_n$ of size n, is chosen to be a $+1$ simultaneous eigenstate of the stabillizers of the code.

Such a stabilizer code with set of stabilizers S will detect errors that are either in S or anticommute with an element of S. Under the influence of depolarizing noise, when an error $E \in \mathcal{P}_n$ occurs on a data qubit, it introduces an extraneous Pauli operator that anticommutes with an element of the stabilizer group. This action flips the eigenvalue of the codeword $|\psi\rangle$ with respect to the anticommuting stabilizer. The measurement of the stabilizers on the ancilla qubits allows for the detection of changes in eigenvalue, which are indicative of errors on the physical qubits. The result of the measurements of the stabilizers at any one time is called the *syndrome* of the code. We can detect and potentially correct errors through a careful analysis of the syndrome via a decoding algorithm.

In fact, there exists a decomposition of the Pauli group $\mathcal{P}_n = \mathcal{E} \otimes \mathcal{L} \otimes \mathcal{S}$ [7], [8], where $\mathcal{L}$ is the logical operator group generated by logical operators on the stabilizer code, and $\mathcal{E}$ is the Abelian subgroup of the Paulis that anticommute with the stabilizers. A *recovery operator* would be a combination of a pure error $E \in \mathcal{E}$ and logical operator $L \in \mathcal{L}$ that matches the syndrome.

2.4 The Surface Code

Surface codes [6] are a family of stabilizer codes with a high error threshold, making them a suitable candidate for fault-tolerant quantum computation. Recent research demonstrates the experimental viability of this family of codes. Surface codes encode logical qubits into a two-dimensional lattice of physical data qubits arranged in a grid. Ancilla qubits are evenly interspersed throughout the lattice to aid in error correction. The stabilizers of the surface code are generated by the tensor product of the X or Z operators corresponding to *vertices* and *plaquettes* on the lattice of physical qubits. Then, for X_q denoting an application of X to qubit q, we have $\hat{X}_s = \bigotimes_{q \in v} X_q$ denoting the star stabilizer generator acting on the qubits at the star site v. The plaquette stabilizer generator is similarly defined, for $\hat{Z}_p = \bigotimes_{q \in p} Z_q$ acting on the qubits at plaquette site p. The stabilizer group is defined as follows:

$$S =< \hat{X}_v, \hat{Z}_p > \subseteq \mathcal{P}_n.$$

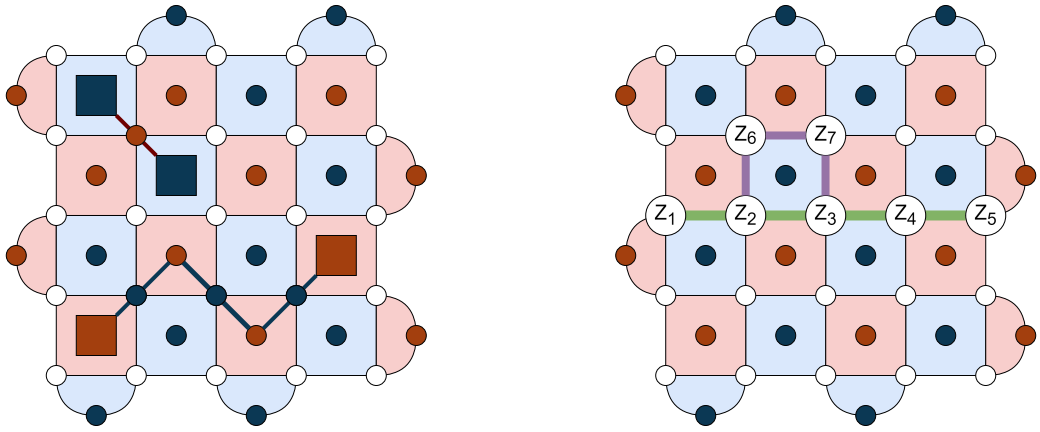
A codeword $|\psi\rangle$ encoded into the surface code and stabilized by S is a simultaneous eigenstate of these stabilizers. The codespace is then defined as follows:

$$T = \{|\psi\rangle \in (\mathbb{C}^2)^{\otimes n} : \hat{X}_v|\psi\rangle = \hat{Z}_p|\psi\rangle = |\psi\rangle, \forall v, p\}.$$

Note that the surface code is *degenerate*, there exists multiple recovery operators for any given syndrome. Notable examples of codes in this family are the toric code, rotated planar code, and XZZX code.

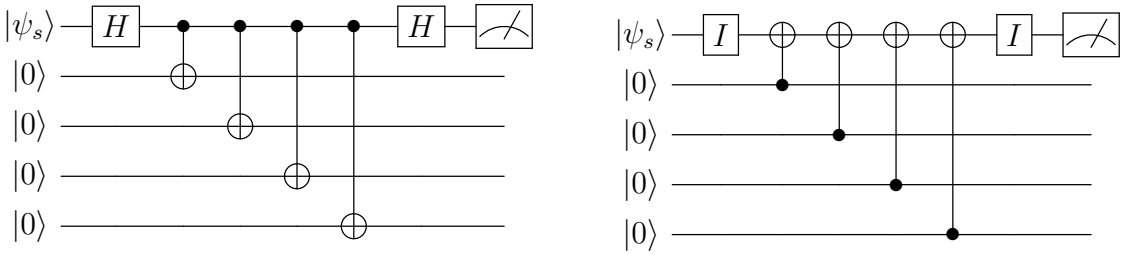
The Rotated Planar Code

This work will focus on the rotated planar code. The rotated planar code is constructed by rotating the standard square lattice of a planar code by 45 degrees. This rotation changes



(a) A $d = 5$ Code (b) Equivalent Chains

Figure 2.1: The Rotated Surface Code



(a) X-stabilizer Measurement Circuit (b) Z-stabilizer Measurement Circuit

Figure 2.2: X and Z Measurement Circuits for the Rotated Planar Code

the lattice geometry to a diamond shape, effectively reducing the boundary conditions and the number of qubits needed to construct the code, when compared to a planar code. See Figure 2.1a for an example of a rotated planar code. Each colored square in the rotated planar code represents a single stabilizer qubit, with red and blue denoting X and Z-type stabilizers, respectively. The data qubits lay at the intersection of the grid lines, denoted by white dots. The stabilizer squares containing circles denote $+1$ measured eigenvalues, and similarly, the stabilizer squares containing squares denote -1 measured eigenvalues. Data qubits which have sustained an error are colored red or blue, depending on the type of error. The rotated planar code is operated by measuring all of the stabilizer qubits in lockstep. See Figure 2.2 for the different types of stabilizer measurement circuits.

The code's stabilizers are products of Pauli operators that, involving either four or three qubits in the interior or at the boundaries, respectively. Typically, one logical qubit is encoded per lattice, with logical operators spanning from one edge of the lattice to the opposite edge along the shortest path. This is a product of Pauli operators that commutes with the stabilizers, and manipulates the two degrees of freedom in the code. Under the presence of error, a logical error consists of a chain of extraneous noise operators that take the form of a logical operation; the boundary-to-boundary chain of operators. Note that

homologically equivalent chains, ones that have undergone a trivial deformation by adding stabilizer operators to the chain of products, have the same effect on this code. Therefore, any deformation of the original logical operator by adding stabilizer operators will have the same effect on the code. See Figure 2.1b for an example. In the diagram, $Z_1Z_2Z_3Z_4Z_5$ is logically equivalent to $Z_1Z_2Z_6Z_7Z_3Z_4Z_5$.

2.5 Decoding the Surface Code

Decoding is the process of using the syndrome produced by the surface code to correct errors. A *decoder* is a decoding algorithm that runs on a classical coprocessor. Its task is to find the most likely error configuration consistent with the observed syndrome. Let us assume that the surface code is subject to depolarizing noise. As this noise model consists of X, Y, and Z Pauli operators, an application of the same operators on the same qubits will undo the error. In this case, a decoder could take the syndrome as input, and output the most likely error configuration, which is therefore the same as the recovery operator. One way of doing this is solving an instance of the minimum-weight perfect matching problem on a graph, which can be solved efficiently with Edmond's blossom algorithm or other decoders tailored to the surface code geometry. See Section 4.2 for more information about this type of decoder.

Hence, the decoder's performance is directly correlated with the fault-tolerance of the code. The *threshold* of a quantum error correction code and a corresponding decoder algorithm refers to the rate of noise at which scaling the code no longer improves the fault-tolerance of the quantum system. The aforementioned decoder based on minimum-weight perfect matching is relatively efficient and achieves a reasonable threshold, but more advanced decoders can boost the threshold of the code further. Designing effective decoders remains an active area of research in quantum error correction, and is the issue that this work aims to tackle.

2.6 Quantum Computation Using the Surface Code

What is a good way to leverage surface codes for fault-tolerant quantum computation? The answer to this question will provide useful details and requirements for constructing an effective decoder for the surface code.

First, recall that a surface code can encode logical qubits. To simplify our model, and to match the rotated planar code that we focus on, we may assume that only one logical qubit is encoded per surface code. The X and Z logical operations would be boundary-to-boundary chains of X and Z operations on individual qubits. To perform logical measurements in the logical X and Z bases, we may measure the tensor product of the physical X and Z operations on data qubits that a valid logical X and Z operation consists of. Note that such an operator is Hermitian, making it a valid measurement observable. To perform operations on two logical qubits, like a logical CNOT, we may use lattice surgery [9], [10].

Since a logical qubit can be operated in a similar fashion to a physical qubit with some caveats, we may then treat these surface code logical qubits as the "smallest" building blocks of a quantum computer. As mentioned in [11], a functional quantum computer at the logical

level would then consist of data blocks, which store logical quantum information in multiple logical qubits, and distillation blocks, which use multiple logical qubits for creating arbitrary logical quantum states. A classical coprocessor would run control software to control the quantum computer by scheduling operations and reading in measurement data, and could even run a decoder for the surface code.

This has some implications for a decoder for the surface code. Consider a syndrome S for the surface code. Recall from Section 2.4 that due to the degeneracy of the surface code, there exist multiple valid recovery operators for S. Because logical qubits form the "smallest" building blocks of such a quantum computer, it does not matter what recovery operator is applied in response to S, as long as the logical state of the surface code is maintained by the recovery operator. Therefore, it suffices for a surface code decoder to just indicate whether the logical qubit has experienced a logical flip or not. Note that for a decoder that directly outputs a proper recovery operator that fits the syndrome, applying this recovery operator to the code also adds uncessary overhead in the form of noisy gates. Instead, such a decoder producing a recovery operator would have to postprocess this recovery operator to produce this aforementioned indication of a logical error. Then, the classical control software for the quantum computer may appropriately flip the result of the next measurement of the logical qubit based on this binary indication.

2.7 Superconducting Quantum Systems

There are many types of qubits, including transmon, photonic, and trapped ion. Among the various types of qubits, superconducting qubits stand out due to their scalability and the advanced level of control achieved in recent experiments. As such, this work will just focus on the superconducting variety. Superconducting quantum computers use Josephson junctions, thin insulating barriers that, with connected inductance and capacitance and a nearby microwave resonator, begins to function as a quasi-particle at superconducting temperatures [12]. These quasiparticles exhibit quantized energy levels, allowing for the discrete quantum states necessary for quantum computing. Therefore, such a Josephson junction can then be treated as a single qubit. Recently, IBM has released a superconducting quantum processor with 133 qubits [13], and Google Quantum AI has published promising results utilizing the superconducting modality [1].

2.8 Summary

In this chapter, we introduced noise in quantum computers, and ways to manage it using quantum error correction. We introduced mixed states and quantum channels, the mathematical language for expressing quantum noise. We then discussed stabilizer codes and in specific, the surface code. In particular, we highlighted the surface code as a particularly robust family of stabilizer codes, known for its high error threshold and suitability for fault-tolerant quantum computation. Finally, we describe what a surface code decoder is, and situate the surface code as the theoretical building block for a future fault-tolerant quantum computer.

Chapter 3

Deep Learning

Deep learning, a subset of machine learning, has revolutionized fields ranging from natural language processing to image recognition. This chapter will serve as a primer for concepts in deep learning which are foundational to the content in the following chapters. Each section of this chapter aims to build understanding that will support the more complex applications discussed later in this work. We will start by defining what neural networks are, discuss how they are trained, cover some common components, and explore specific architectures that will be utilized in subsequent chapters.

At a very high level, neural networks are large parameterized black-box functions, whose parameters may be updated through gradient descent. The architecture of these networks typically includes input, hidden, and output layers that facilitate data processing and learning. There are various types of network architectures, notably *feedforward*, where data moves in one direction, and *feedback*, where outputs are looped back into the system as inputs. Training neural networks involves learning from data, which is achieved through defining loss functions and applying optimization algorithms like backpropagation and gradient descent. These methods adjust the weights and biases to minimize errors in predictions. Neural networks have found widespread applications across diverse fields such as image recognition and natural language processing.

3.1 Backpropagation and Gradient Descent

Backpropgation is an algorithm for updating the weights of a neural network. First, a forward pass is computed, with input data passed through the network through any hidden layers all the way to the output layers. Once the network produces an output, a loss function can be used to calculate the distance of the prediction from the correct target values. Some examples of these loss functions include *mean squared error* for regression tasks, and *cross-entropy loss* for classification tasks.

After the loss is computed, a backwards pass takes place, where backpropagation is used to calculate the gradient of the loss function with respect to each weight and bias in the network. This algorithm begins from the output layer and moves backwards through the network, using the chain rule to iteratively calculate the gradient for each layer.

Using these gradients, the weights and biases in the network can be updated using gra-

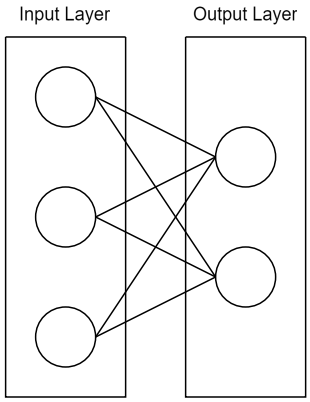


Figure 3.1: Linear Layer

dient descent. In other words, the weights and biases are repeatedly adjusted in a direction that decreases the loss the most. To train a neural network, this process of forward pass, loss calculation, backpropagation, and weight update is repeated across multiple iterations. Each iteration processes the entire training data set or subsets of it. Over time, this iterative process is designed to minimize the loss function, ideally converging to a state where the network makes accurate predictions or classifications.

3.2 Fully Connected Layers

A linear layer, also known as a fully connected, dense layer, or affine layer, is a fundamental component of many neural networks. It is a linear transformation, where each input is multiplied by a weight and then a bias is added. See Figure 3.1 for a diagram of this layer. In the figure, the layer has three input neurons and two output neurons. Each input neuron is connected to every neuron in the output layer, represented by lines. Each line has an associated weight. When data enters the input layer, each input neuron passes its value through these connections to the output neurons. The values are multiplied by the weights of the connections they travel along. Then, the results are summed up at each output neuron. Often, a bias term is added to this sum, and an activation function is applied to introduce non-linearity into the model. Mathematically, this can represented as $y = Wx + b$, where x is the input vector, W is the weight matrix, b is the bias vector, and y is the output vector.

Nonlinearities and Activation Functions

Neural networks rely on nonlinearities to learn complex patterns. Without nonlinearities, a neural network, regardless of how many layers it contains, would still perform no better than a linear model, as each layer would effectively just be a linear transformation of the previous layer. Nonlinearities allow neural networks to approximate non-linear functions, which is essential for tasks where the decision boundary is not a straight line. Nonlinearities, introduced through *activation functions*, allow neural networks to capture complex relationships in the data by adding depth to the decision-making process. They enable the network to aggregate

input in complex patterns, thereby enhancing its capacity to learn varied phenomena from data. There are two main activation functions used in this work:

1. **Sigmoid:** The sigmoid activation function outputs values between 0 and 1, making it useful as the final step before output for models designed to predict probabilities. The formula is given by:
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

2. **Rectified Linear Unit (ReLU):** ReLU has become very popular because it allows models to converge faster and reduces the likelihood of encountering the vanishing gradient problem introduced later in this work. ReLU is defined as:
$$ReLU(x) = \max(0, x)$$
It activates a neuron only if the input is above zero, which introduces non-linearity while keeping computational simplicity.

In a fully connected layer, an activation function resides in each neuron.

3.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of deep neural networks that are particularly powerful for processing data with a grid-like topology, such as images. CNNs are distinguished by their use of convolutional layers, which adaptively learn spatial hierarchies of features. This makes them efficient for tasks like image recognition, video processing, and medical image analysis.

The primary feature of CNNs is the convolutional layer, which applies a convolution operation to the input, passing the result to the next layer. This operation captures the spatial dependencies in the input data by learning from small regions of the input, known as receptive fields. Unlike fully connected networks that learn patterns at a global scale, CNNs focus on local patterns, making them more efficient and requiring fewer parameters.

A typical CNN architecture consists of several layers that transform the input volume into an output volume (e.g., class scores) through differentiable functions. These include the aforementioned convolutional layers, which learn local feature representations, pooling layers, which help to reduce the dimensionality of each feature map, and fully connected layers, which compute class scores and output the prediction of the network. Each layer of neurons applies different filters, typically learned during the training process, and downsampling techniques like max pooling to reduce the spatial size of the representation.

Training deep CNNs comes with some challenges. Due to the high capacity of some of these models, they are prone to overfitting, particularly when trained on small datasets. Additionally, CNNs require significant computational resources, especially for training on large datasets with high-resolution images.

3.4 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of neural networks designed specifically for processing sequential data. Unlike feedforward neural networks, RNNs have the distinctive feature of maintaining internal memory or state that captures information about the sequence they have processed so far. This makes RNNs ideal for applications where context and time are crucial, such as language modeling and speech recognition.

Whereas traditional neural networks take the form of directed acyclic graphs, RNNs are cyclic by design giving them the ability to process data through loops. This allows information to persist in the state of the RNN, enabling it to make decisions based on a history of inputs up to the current step, unlike traditional neural networks which process each input in isolation.

An RNN typically consists of a layer of neurons with a self-loop that represents the time dimension, enabling a form of memory. The architecture can be broken down into an input layer, which receives sequences of data; a hidden layer, which processes inputs using weights that are shared across time steps, maintaining a state or memory of past inputs; and an output layer, which produces the output for each time step, which can depend on the current input and the current state of the hidden layer.

The key equation that defines the basic operation of an RNN is:

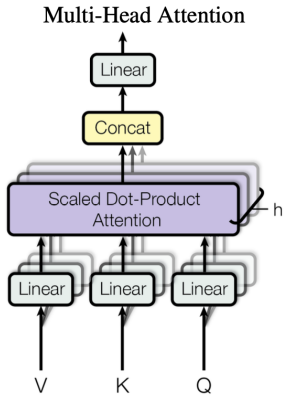
$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

where h_t is the hidden state at time t, x_t is the input at time t, W_{xh} are the weights from the input layer to the hidden layer, W_{hh} are the weights from the hidden layer to itself, b_h is the bias, and σ is the activation function, typically a sigmoid or tanh function.

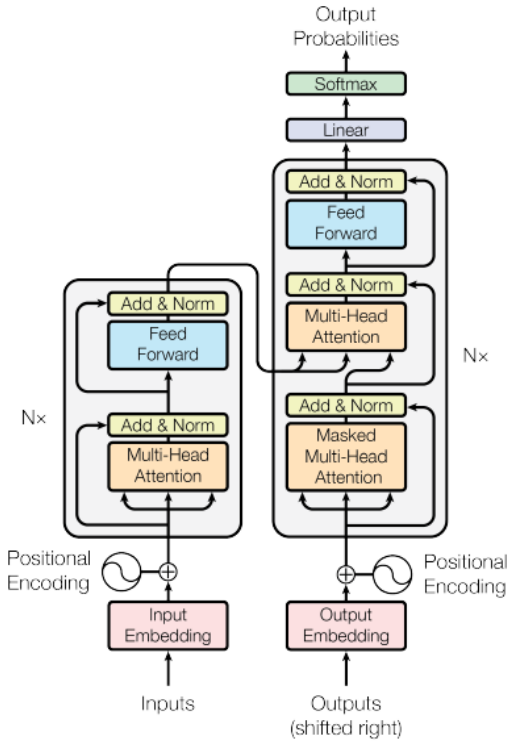
Training RNNs involves backpropagation through time (BPTT), a variant of the standard backpropagation algorithm adapted for sequential data. This method unfolds the RNN through time and then applies backpropagation. However, BPTT suffers from the vanishing gradient problem and the exploding gradient problem, where the gradients of the loss function can become very small or very large, making the weight update either negligible or very large. In these cases, the network is either unable to learn long-range dependencies, or has unstable convergence, potentially leading to convergence at a suboptimal solution.

To address the aforementioned vanishing and exploding gradient problem, several variants have been developed. This includes the Long Short-Term Memory (LSTM) network, which incorporates gates that control the flow of information. This effectively allows the network to learn when to "remember" and when to "forget" states, thereby mitigating the vanishing gradient problem. Another popular type of RNN is the Gated Recurrent Unit, which is a simpler alternative to LSTMs that use a similar gating mechanism, but with fewer parameters.

RNNs and their variants can be utilized in a range of applications that require an understanding of sequential patterns including machine translation, text generation, sentiment analysis, speech recognition, and time series prediction.



(a) Multi-Head Attention Mechanism



(b) Transformer Architecture

3.5 Transformer Networks

Transformers [14] have revolutionized the field of deep learning, particularly in tasks involving natural language processing (NLP) and image processing. Developed as an alternative to RNNs, transformers address several shortcomings of RNNs, including difficulty in parallelizing operations and the vanishing gradient problem. Key to the transformer's success is its use of the self-attention mechanism, which allows it to weigh the importance of different elements within a sequence, irrespective of their positional distances.

The Multi-Head Attention Mechanism

The Multi-Head Attention module allows the model to jointly attend to information from different representation subspaces at different positions. By doing this, the transformer can integrate information more effectively across the sequence. By focusing this attention on the same sequence as the input, we get self-attention. The self-attention mechanism allows the model to focus on different parts of the input sequence as it processes each word. For each word, the transformer computes a score that signifies its importance relative to every other word. This approach enables the model to capture context more effectively and manage dependencies between words far apart in the input sequence. Since transformers lack recurrence, they use positional encodings to incorporate information about the position of words in the sequence. These encodings are added to the input embeddings at the base of the model and help preserve the order of the sequence. See Figure 3.2a for a diagram of

this mechanism.

The Transformer Architecture

Transformers utilize an encoder-decoder structure. The encoder multiple identical layers, each with two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. Layer normalization and residual connections are employed around each of the sub-layers. Similarly, the decoder also consists of a series of identical layers, but with an additional third sub-layer that performs multi-head attention over the output of the encoder stack. Similar to the encoder, the decoder employs residual connections and layer normalization. See Figure 3.2b for a diagram of this architecture.

3.6 State Space Models (SSMs)

SSMs can be thought of as a hybrid between a RNN and a CNN. Typically, an SSM operates by mapping an input sequence through a series of state transformations. These transformations are defined mathematically by a set of parameters that determine how the input data is processed to produce the output. For the S4 model [15], a specific type of SSM, these parameters are matrices referred to as A, B, and C, where

- A represents the state transition matrix that describes how the hidden state evolves from one step to the next.
- B is the control matrix that maps the influence of the input sequence on the state.
- C is the output matrix that maps the hidden state to the output.

These parameters are used in tandem to both update the current state h_{t-1} to h_t, and output a vector y, given an input x. These two actions define a sequence-to-sequence transformation. The update equations are given as follows:

$$h^{'}(t) = Ah(t) + Bx(t).$$

$$y(t) = Ch(t).$$

Mamba (S6)

Mamba, or S6, [16] is a recent improvement of the S4 model. In comparison with previous SSMs, Mamba allows its parameters to vary based on the input. This innovation allows the model to selectively remember or ignore information along a sequence, thus optimizing the handling of data through the sequence. However, the parameters to vary based on the input like this prevents the model from training in its faster convolutional mode. This can overcome through hardware-aware optimizations, intelligently using the faster SRAM (static RAM) and the slower HBM (high-bandwidth memory). This approach not only facilitates linear-time scaling in sequence length but also maximizes computational efficiency

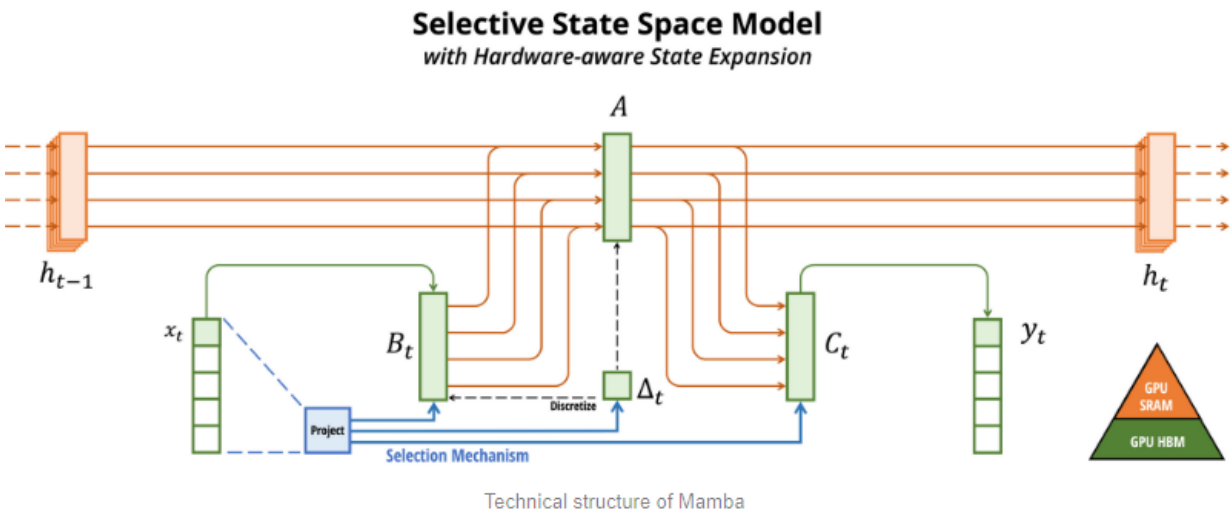


Figure 3.3: Mamba Architecture

by avoiding the full materialization of expanded states across different GPU memory levels. Empirical evaluations demonstrate Mamba's effectiveness across different tasks, showcasing its ability to handle sequences up to a million elements long with significant improvements in throughput and performance compared to existing Transformer models at this scale. See Figure 3.3 for a diagram of the Mamba Architecture.

3.7 Summary

In this chapter, we cover the fundamentals for deep learning. We discussed the neural network's basic structure — consisting of input, hidden, and output layers — and how data flows through these networks. We also explored the training process, including gradient descent and backpropagation. Finally, we cover network architectures, convolutional neural networks, recurrent neural networks, transformers, and state space models. Each of these architectures is suited to specific types of data and problems. This discussion sets the stage for the neural network decoders discussed in the next chapter, and the main ideas presented in this work.

Chapter 4

Neural Network Decoder

In this chapter, we introduce neural network decoder and our approach to improve its scalability. We first review some relevant decoders from the literature, then provide a description of our neural network decoder. Recall from Section 2.5 that the performance of the surface code decoder is directly correlated with the fault-tolerance of the surface code. As such, it is an important component to improve. There are many different approaches to decoding, and many different decoders. This chapter will provide a review of several categories of relevant decoders, as well as qualifying why we believe neural network decoders will lead to the best performance. We will delve into each category's underlying principles, their advancements, and the specific challenges they face in practical applications. This discussion will set the stage for introducing our methods to address the scalability and efficiency issues currently limiting neural network decoder performance in larger quantum systems.

4.1 Problem Statement and Proposed Approach

As mentioned in Section 2.5, a *decoder* is an algorithm that finds a most likely error configuration consistent with an observed syndrome in the surface code. However, as mentioned in Section 2.6, this is logically equivalent to just outputting an indication whether the logical observable in the code has flipped or not. Therefore, a neural network decoder, a decoder that is a neural network, would perform binary classification on the input syndrome, classifying the surface code as having experienced a logical flip or not. In an X-type surface code memory experiment, this takes the form of predicting a logical X-flip, and in a Z-type surface code memory experiment, this takes the form of predicting a logical Z-flip. Such a neural network would be trained on syndrome data labelled with the correct logical observable outcomes.

Although this type of neural network decoder has been demonstrated in the literature (See Section 4.2) to have a high threshold and low evaluation time, such a decoder needs to be trained on an exponential amount of data. In the remainder of this work, we propose two approaches to reduce the neural network decoder's need for training data. Both of these approaches involve dimension reduction, to reduce the size of the problem space such a decoder has to explore during training. We show that reducing the dimensionality of the problem could lead to less data dependence and faster training. We will refer to these two

approaches as the convolutional decoder and the transformer decoder.

4.2 Overview of Surface Code Decoders

Here, we introduce three relevant categories of decoder, each differentiated by the formulation of the decoding problem they solve: maximum probability decoders, maximum likelihood decoders, and machine learning decoders.

Maximum Probability Decoders

Maximum probability decoders find a recovery operator E which is maximally probable based on the observed syndrome. The main example of such a decoder is the minimum weight perfect matching (MWPM) algorithm. This *matching decoder* solves the MPWM problem on a *decoder graph* G consisting of ancilla qubits of one type as vertices, and edges weighted by a metric, such as the Manhattan distance between such ancilla qubits in the surface code lattice. For example, the X-type stabilizers for the surface code may be chosen as the vertices in G. The decoder will also solve the same problem on the graph consisting of Z-type stabilizers. The edge weights of such a G may be appropriately modified to reflect the probability distribution of errors afflicting the code.

Maximum probability decoders experience two main challenges, computational cost and decoder accuracy, both of which have been addressed in the literature. By principle, matching decoders solve a difficult combinatorial optimization problem twice to produce a diagnosis from the syndrome. Even so, there exist efficient implementations of these decoders, leveraging various characteristics of the decoding problem to greatly accelerate decoding [17], [18]. By adapting the algorithm to specifically solve the decoding problem, they greatly reduce the computational complexity of the algorithm to near linear. There also exists fast hardware implementations of maximum probability decoders in literature [19], [20]. However, due to the decoder graph having a node for every detection event in the syndrome, this decoder struggles to decode high density and distance syndromes.

The other remaining problem is accuracy. First, the formulation of the problem makes assumptions that the two stabilizer types have separate probability distributions, when they are actually correlated through Y-type errors. There exists a matching decoder that addresses this problem by running belief propagation as a subroutine to reconcile the divide between the two types of stabilizers [21]. Second, the formulation of the maximum probability decoding problem is inherently suboptimal. Recall that under our scheme for logical quantum computation, it does not matter what specific recovery operator is applied to the code, only that the state of the logical qubit is preserved. Since the surface code is degenerate, that many recovery operators have the same action at the logical level, it no longer makes sense to consider the probability of each recovery operator separately, but equivalence classes of recovery operators. Here, we turn to the maximum likelihood decoder.

Although some implementations of the maximum probability decoder reach the speeds necessary for decoding superconducting surface codes, this doesn't necessarily hold for high density and distance syndromes. We also note that the relatively lower accuracy compared to other types of decoders is another drawback of using this category of decoder.

Maximum Likelihood Decoders

In comparison to the maximum probability decoder, maximum likelihood decoders search for the maximally likely *equivalence class* of errors based on the observed syndrome. Under our current understanding of the decoding problem, it is a better formulation to solve, in terms of optimality. However, it is fundamentally more difficult than the maximum probability formulation. A naive, brute force maximum likelihood decoder searches for the most likely equivalence class of errors by enumerating all of the recovery operators in each equivalence class, and comparing the sums of the probabilities for each class.

There do exist decoders in the literature that circumvent the speed problem via approximations. The maximum likelihood problem can be equivalently expressed using a statistical mechanical model and reduced to calculating partition functions for this model, which further reduces to the contraction of tensor networks [22]. Such a tensor network decoder approximates the decoding problem by compressing the state of the tensor network appropriately to maintain a high level of accuracy [23], [24].

Another type of maximum likelihood decoder is the renormalization group decoder [25], [26], which more aggresively approximates the maximum likelihood decoding problem by treating the surface code as a concatenated code. This means that we partition the code in a way such that solving the subproblem in each partition produces a decoding problem on a smaller surface code approximately equivalent to the decoding problem on the original surface code [25], [27]. We call this process of scaling the code down *renormalization*. We introduce this decoder, as it bears similarity to our proposed method.

To summarize, the maximum likelihood solves an optimal formulation of the decoding problem, but due to the difficulty of the problem, it either aggresively approximates it, or struggles to solve it in a timely manner.

Neural Network Decoders

The last category of decoders which we introduce are the neural network decoders. These decoders do not directly solve any formulation of the surface code decoding problem, but instead directly read in the syndrome and predict either a recovery operator or a logical correction through application of a neural network. Recall that in practice, a decoder only needs to output an indication of whether the logical observable has flipped or not. In terms of a neural network, this would take the form of outputting a binary prediction, 0 or 1, of whether or not the logical observable has flipped. In machine learning terminology, this is a binary classification problem — a neural network built for this problem assigns a binary label to the input data.

There are many examples of machine learning decoders in the literature. In [28] the authors decode small distance surface codes and achieve similar performance to a vanilla matching decoder. We see that neural network decoders enjoy high accuracy and low latency [29]. In fact, [30] shows that a version of their neural network decoder outperforms tensor network decoding in terms of accuracy. [30], [31] showcase that through embeddings, the decoders may receive a rich set of inputs, including in-phase/quadrature (I/Q) readouts, which enhances the accuracy of their decoding. If designed well, the runtime of neural network decoders will not scale with the density of syndromes as the matching decoder does,

but just with the distance of the code [30]. Additionally, they may be trained to specifically target a specific quantum computer, further enhancing their accuracy [32].

We see that neural network decoders have a lot of potential, with lots of positive aspects. However, one of the primary problems with neural network decoders is exponential scaling of the training data with the distance of the surface code [28]–[30], [33]. This is an issue for two reasons:

1. **Data Sparsity:** Although training data can be generated very cheaply through simulators, real quantum experimental data is scarce. Training a neural network decoder that functions well for a specific quantum computer and noise model will require fine tuning the decoder for that specific system, using data generated from that system.

2. **Training Time:** [30] mentions that based on their dependence on data, it may take ten to a hundred trillion examples for their neural decoder to reach the performance of the correlated matching decoder on distance 25 codes. For much larger codes, it becomes infeasible to train a neural network decoder.

This problem is addressed in the literature. [34] applies the concept of renormalization through repeated convolutions, scaling down a surface code afflicted with bit-flip noise before decoding it through a feed-forward network. However, this work does not consider more complex noise models and how the model will deal with those issues. [35] applies the renormalization group decoder before decoding with a surface code. [36] scales the neural network to high distances by predicting a partial recovery operator with a fully convolutional network, before cleaning up the remaining sparse syndrome with a fast high-level decoder. Note that all of these works do not consider using embeddings to include more information in the input of the network through embeddings.

4.3 Convolutional Decoder

The convolutional decoder treats the surface code as an "image", which it downscales via repeated applications of a convolutional autoencoder before further processing. This way, the neural network can theoretically be fine tuned to decode at arbitrary code-distances, without needing to pretrain a model from scratch. The model can simply take a large code, and compress the state repeatedly until it reaches the distance that it was trained to decode at. In principle, this convolutional autoencoder functions in a manner similar to the aforementioned renormalization group decoder, where the decoding problem for the surface code can be scaled down using a combination of maximum likelihood decoding and belief propagation. In comparison with [34], [35], this is instead done on the model's hidden latent state, after embedding the input data.

Architecture

For the convolutional decoder, we train a recurrent neural network to decode surface code memory experiments, with architecture inspired by [30]. It consists of several smaller networks, including an embedding model, a recurrent core, and a readout network, which all

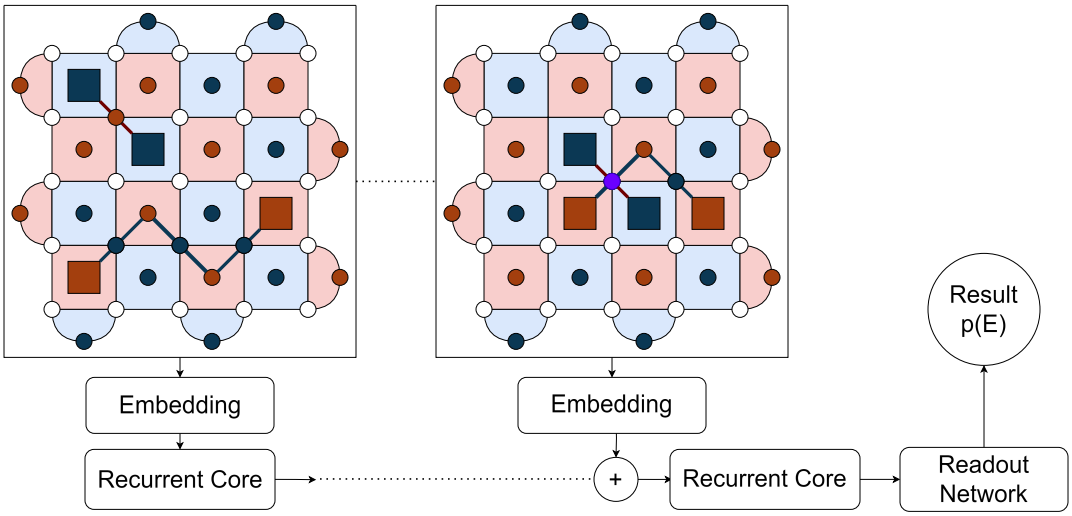


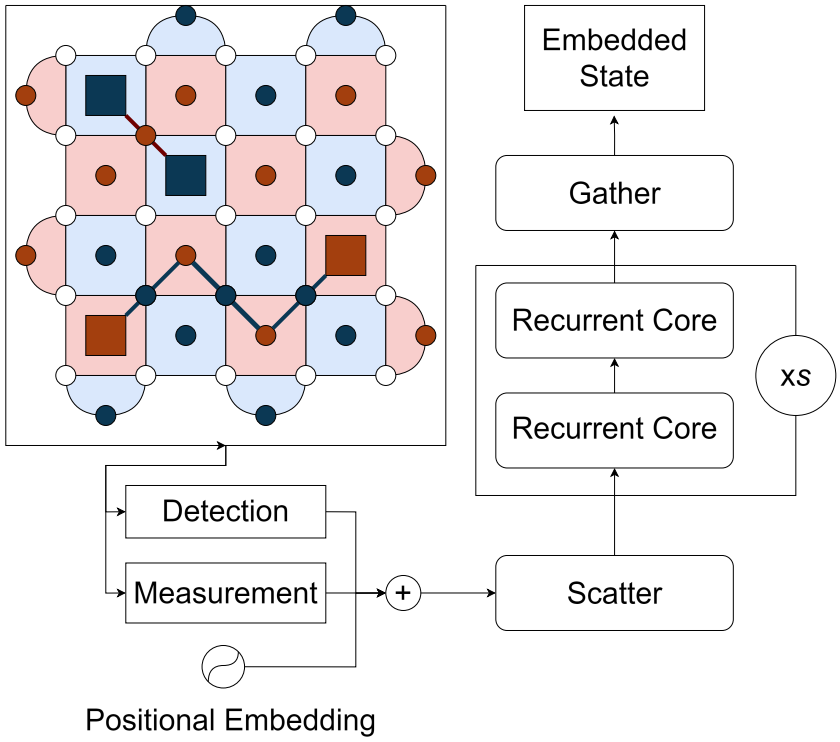
Figure 4.1: Mamba Decoder Model

work together in tandem to decode the surface code. Given the history of syndrome measurement data, it will produce a probability $p \in [0, 1]$ that the logical observable has flipped in the duration of the experiment.

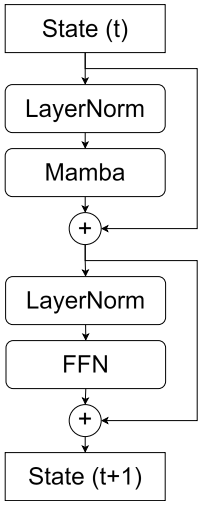
Embedding

An embedding model can be thought of as a translator, translating the input data into a format that is comprehensible to the processing modules in the neural network. In the memory experiment, each round of measurements, or time step t, produces syndrome measurement outcomes with a binary 1 or 0 for each stabilizer in the surface code. The embedding model receives syndrome measurement and detection data, which are embedded into vectors, one for each stabilizer. Detection data refers to a XOR between the stabilizer measurement results in the current round and the last round. Measurement data refers to the stabilizer measurement results in the current round. The two dimensional spatial coordinates of each stabilizer in the surface code are also passed through a 2D sinusoidal positional embedding and added to the embedding vectors. This embedding process produces a vector in a high-dimensional latent space for each stabilizer.

These stabilizer embeddings are then scattered to a 2-dimensional grid of size $(d_e + 1) \times (d_e + 1)$, where d_e is the size of the experiment surface code. These stabilizers are placed in the same positions they would occupy on the surfac code. Then, this grid is passed into a fully convolutional autoencoder, consisting of a convolution that reduces the effective "distance" of 2-dimensional grid by two, and dilated convolutions. The number of times this autoencoder is run is based on the size of the surface code that is being decoded, and the "native distance" of the surface code that is being decoded. Since the autoencoder reduces the size of the grid by two each time, it is run $(d_e - d_n)/2$ times, for d_n denoting the model's native distance. Through applications of this autoencoder, this grid-format latent space representation is scaled down to the RNN's "native" code distance where it is then gathered back into a sequence of updated stabilizer embeddings. This stabilizer embedding is added to the model's tracked hidden state.



(a) Embedding Model



(b) Recurrent Block

Recurrent Core

The convolutional decoder tracks a hidden state concurrently with the progress of the memory experiment. This hidden state can be thought of as the model's "understanding" of the current state of the surface code at that timestep. Once the stabilizer embeddings are created for any one timestep, it is multiplied by a scalar quantity less than 1, and added to the model's tracked hidden state. The *recurrent block* resembles the encoder stack of the Transformer model. It conists of a gated Mamba model with layer norm, followed by a gated feedforward network with layer norm. Each time new stabilizers are encoded through the embedding model, the sum of the previous hidden state and the new embedding is passed through three recurrent blocks.

Readout Network

The readout network processes the model's hidden state to produce a prediction of whether the logical observable has flipped or not. First, the hidden state is scattered to a 2-dimensional grid of size $(d_n + 1) \times (d_n + 1)$, where d_n is the native distance of the decoder. Again, as in the embedding model prior to autoencoding, the stabilizers in the model's hidden state are scattered to the positions that they would normally occupy on the surface code. A 2D convolution layer convolves the stabilizer representation to instead represent data qubits, on a $d_n \times d_n$ grid. Then, this representation is passed through an average adaptive pooling layer, pooling in the direction of logical operations. This state is now in the form of d_n vectors, which is passed through a gated feedforward network, and pooled again

to form a single vector. Finally, a linear layer with an attached sigmoid function outputs the probability of a flip in the logical observable at that timestep.

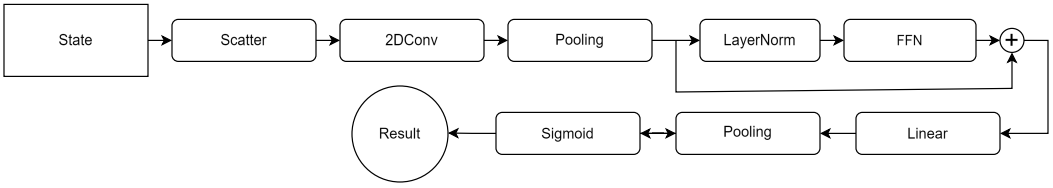


Figure 4.3: Readout Network

4.4 Transformer Decoder

The transformer decoder instead approaches the surface code decoding problem from the perspective of natural language processing. The same recurrent framework remains, but the rationalization differs. Each stabilizer is treated as a "word", and the syndrome readout from each round is treated as a "sentence". Then, in our natural language processing analogy, decoding the experiment is simply solving a classification problem on a string of text. We may apply an encoder network to produce a "sentence" embedding for each round of measurement, which is added to the model's hidden state.

Architecture

The architecture of the transformer encoding model is very similar to the previous architecture, but with some simplifications and modifications. See Figure 4.4 for a diagram of the architecture for this model. As with the prior model, s embeddings are generated for the s stabilizers, consisting of the syndrome measurement and detection data for that round, along with a sinusoidal embedding encoding the stabilizers' spatial positioning in the code. In addition, we also embed a binary flag indicating whether or not it is the final round of decoding. As the physical mechanism is different in the final round, making this distinction may be helpful for improving the quality of the model's predictions. This produces the "word"-level embeddings, one for each stabilizer.

Instead of compressing the hidden state via convolutions, a multi-head attention mechanism instead maps these stabilizer embedding vectors to a small number of vectors, which are then scaled up to a high dimension via a feed forward network. The goal is for this high dimensional representation to fully encapsulate the state of the the surface code at that time step. See Figure 4.5 for a diagram of this encoder mechanism. These high-dimensional vectors can be thought of as the "sentence"-level embedding.

Finally, this high-dimensional representation can be added to the hidden state of the RNN. Here, the recurrent core that acts on the hidden state after modification is replaced by a feed-forward network. The readout network is also simplified. A pooling layer consolidates the information from the high-dimensional vectors into a single vector, which is then processed by a fully connected layer to produce a probability output via a sigmoid function.

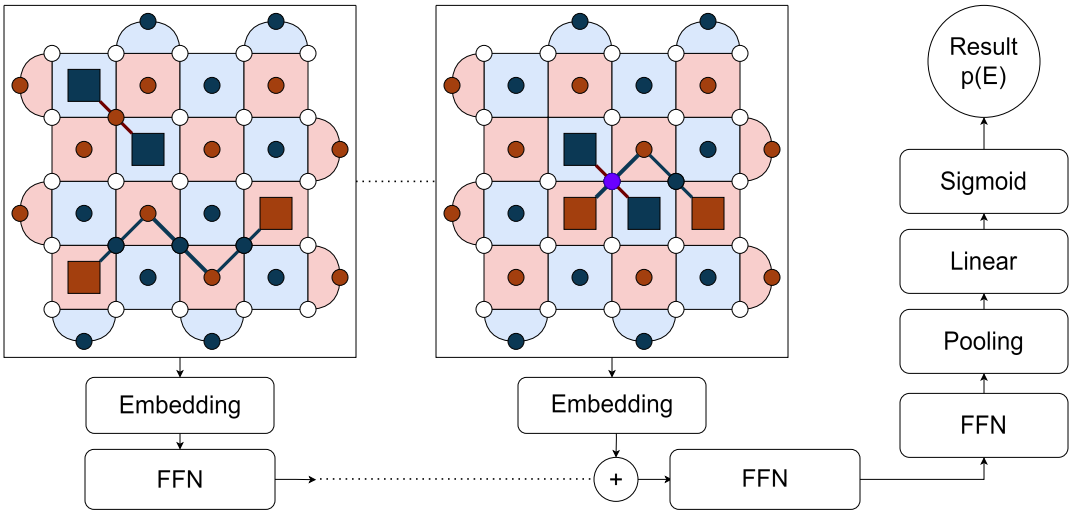


Figure 4.4: Transformer Decoder Model Architecture

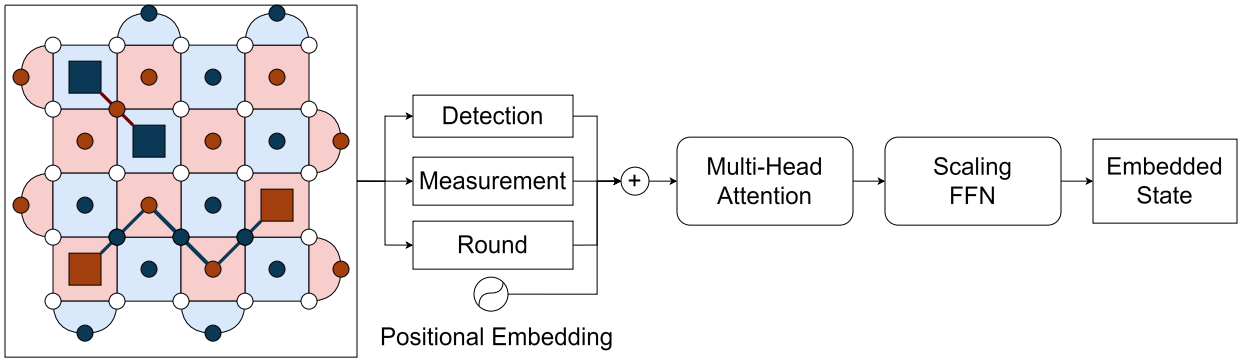


Figure 4.5: Encoding with Attention

4.5 Implementation and Training

Both networks were implemented using Pytorch, a Python library commonly used for training neural networks. We use the Mamba module provided on Github by the authors. Since the training data was fully synthetic and could be generated at nominal cost, the training dataset size is effectively infinite. Therefore, overfitting is not a concern in this case for both models; each model only sees each example once. However, to compare between various iterations of the model being trained, a validation dataset was generated using a fixed seed. The random number generator used to seed the training datasets was modified such that it will not generate the seed for the validation dataset. For training details, including hyperparmeters and loss curves, see Appendix A.

Training the Convolutional Decoder

The convolutional decoder was trained to decode the distance 11 surface code, afflicted with depolarizing noise. The base error probability was set to 0.1%, matching the base error probability of state-of-the-art superconducting hardware presented in [1] and neutral atom

hardware presented in [37]. The model was trained for 100 epochs, with $51,200$ Z-type memory experiments lasting 25 rounds each for all epochs.

Training the Transformer Decoder

Four separate transformer decoders were trained to decode the distance $d = 3, 5, 7, 9$ surface codes, afflicted with superconducting noise. All decoders were set to compress the number of stabilizer embeddings down to 8. The base error probability was set to 0.1%. The models were trained with $409,600$ Z-type memory experiments lasting 25 rounds each for all epochs. This greatly increased epoch size is due to how lightweight the decoder is. Due to time constraints, early stopping was employed. Since training takes longer for higher distance codes, the each model was trained on a different number of epochs. The higher distance decoders received less training than the lower distance decoders. See Appendix A for a more detailed overview.

Stim

For training data, surface code memory experiments are generated using Stim [38] at nominal computational cost. Stim is a Python library that simulates quantum stabilizer circuits at high efficiency. Stim reads in and executes circuit objects, which contain definitions of a stabilizer circuit with Pauli noise annotations. We modify an existing script to generate data according to a superconducting (SI1000) noise model, defined in this paper [39]. Aside from plain depolarizing noise applied to qubits as a result of Clifford operations, it also applies depolarizing noise due to idling and resonator idling, mimicking the behavior of superconducting qubits subject to noise.

Efficient Training

As noted in [30], due to the similarities in architecture, the readout network can actually be called at any timestep during the memory experiment. If we generate data using the same psuedorandom seed once for each round of the memory experiment, we can collect final round measurement outputs and the logical observable for each round of the experiment. For a memory experiment of r rounds, training in a naive manner will require $r + 1$ calls to the stabilizer embedding model and recurrent core for a single output. Using this method, the same memory experiment will require $2r$ calls to the embedding model, recurrent core, and readout network to produce r outputs. Since the majority of computation is concentrated in the embedding model and recurrent core, this greatly reduces the amount of computation needed to train the model.

4.6 Evaluation

We evaluate the model's performance at various distances for both decoders. These codes were simulated in Stim, subject to superconducting noise at varying base probabilities: 0.05%, 0.1%, 0.15%, and 0.2%, for 25 rounds each. These testing datasets consist of 1024

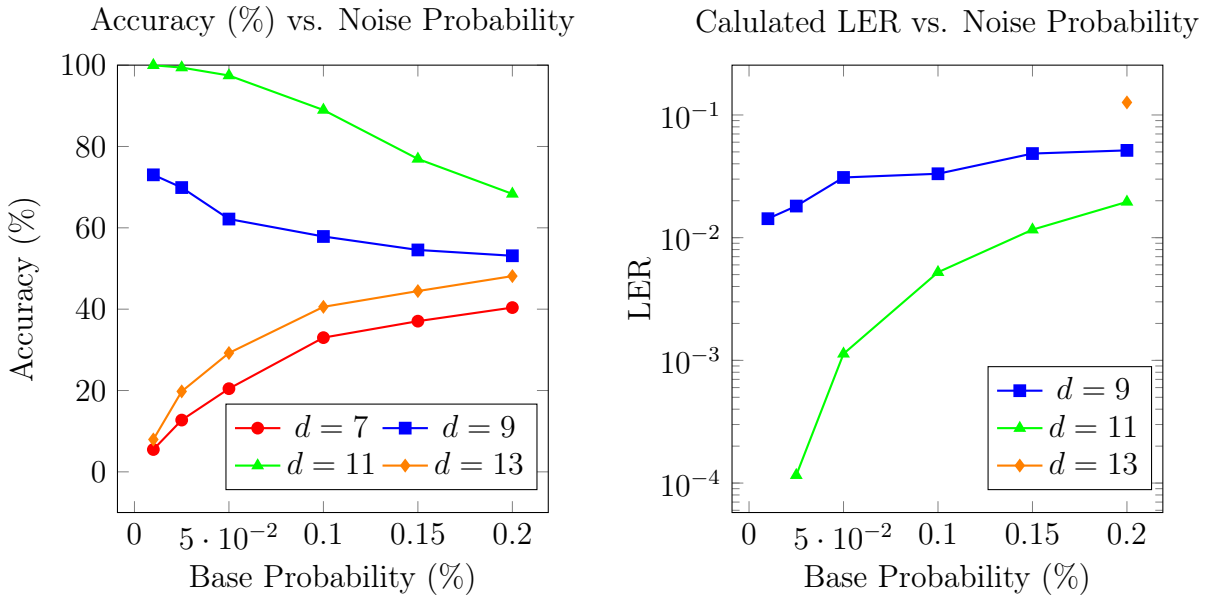


Figure 4.6: Accuracy and LER of the Convolutional Decoder, $r = 25$

shots of these simulations, with a set seed to ensure the same generated testing data for both codes. Note that both models were configured to perform readouts after every round save the first two, meaning that the effective dataset size for each probability setting increased to $23,552$ trials.

We calculate the logical error rate per round (LER) of the decoders at each setting using the following fixed formula, given in the Supplementary Material of [1]:

$$L = \frac{1}{2}[1 - (1 - 2E)^{1/r}]$$

Here, we denote L as the LER of the decoder, E as the error rate (1 - accuracy), and r as the number of rounds in the experiment.

Convolutional Decoder

The aforementioned validation experiments were carried out for the convolutional decoder at distances $d = 7, 9, 11, 13$. See Figure 4.6 for a plot of the results. Note that for distances $d = 7$ and $d = 13$, accuracies below a certain threshold (around 50%) lead to complex valued results for the LER. As such, they are omitted from the plot for those data points. Here, it seems that the convolutional decoder is unable to effectively decode the experiments at distances other than the one it was trained at. This implies that the convolutional autoencoder layer did not correctly learn to compress the code for arbitrary distances.

To check that the decoder can scale to longer experiments with more rounds, the performance of the decoder was also checked at $r = 250$. As the convolutional decoder seemed underfit for distances outside of its training data, we did not deem it necessary to also benchmark these. See Figure 4.7 for a plot of the results. We see that although it is still able to decode with high accuracy at the low base noise probabilities, the accuracy falls off much

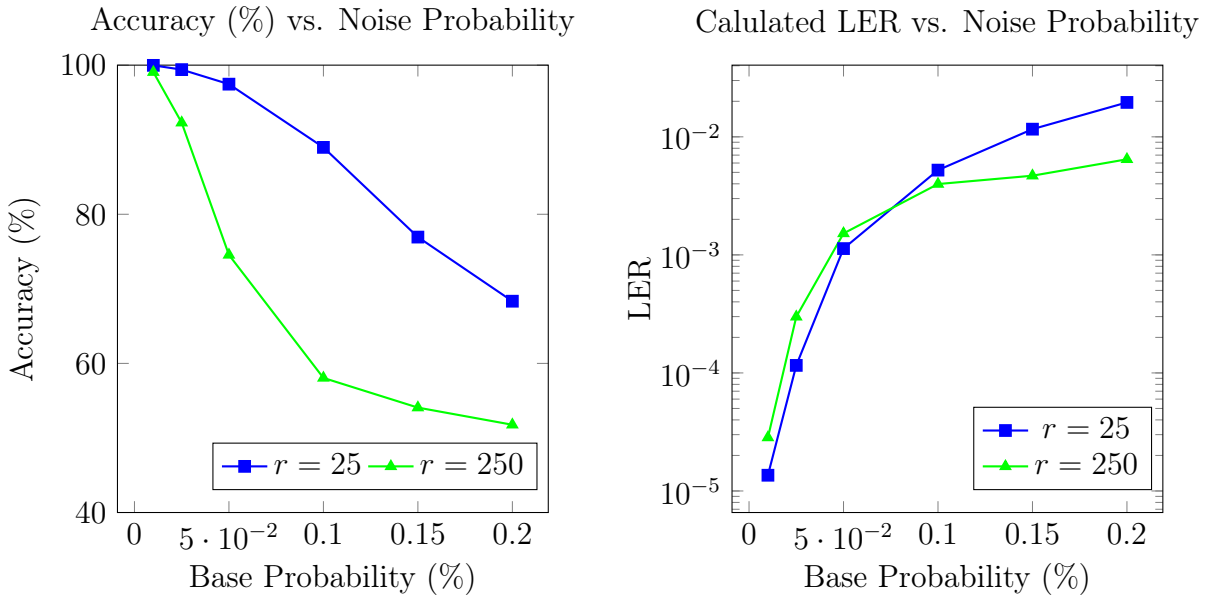


Figure 4.7: Accuracy and LER of the Conv. Decoder at $d = 11$, $r = 25$ vs $r = 250$

quicker with increasing base noise probability. We also note that although the accuracy for the decoder evaluated at $r = 250$ is lower, the LER is also lower due to how many more rounds it was evaluated on.

Transformer Decoder

The aforementioned validation experiments were carried out for the four transformer decoders at the distances they were trained at, $d = 3, 5, 7, 9$. See Figure 4.8 for a plot of the results. Note that there exists a tradeoff between the size of the code and the amount that each decoder was trained. We expect that a decoder performs better on higher distance codes, as the higher distance code should be innately more fault-tolerant at base error probabilities less than the threshold. However, since the higher distance decoders received much less training, they are underfit, and will have lower performance than expected.

To check that the decoder can scale to longer experiments with more rounds, the performance of the decoder was also checked at $r = 250$. See Figure 4.9 for a plot of the results. Similar to the case with the convolutional decoder, we see that it is able to decode with high accuracy at the low base noise probabilities, with the accuracy falling off quickly with increasing base noise probability. We also note that although the accuracy for the decoder evaluated at $r = 250$ is lower, the LER is about the same, due to how many more rounds it was evaluated on. Here, we also see the high fault-tolerance of the higher distance codes overpowering the weakness of the higher distance decoders at $r = 250$, since $r = 25$ is likely not a long enough experiment to fully accumulate errors at these base error probabilities.

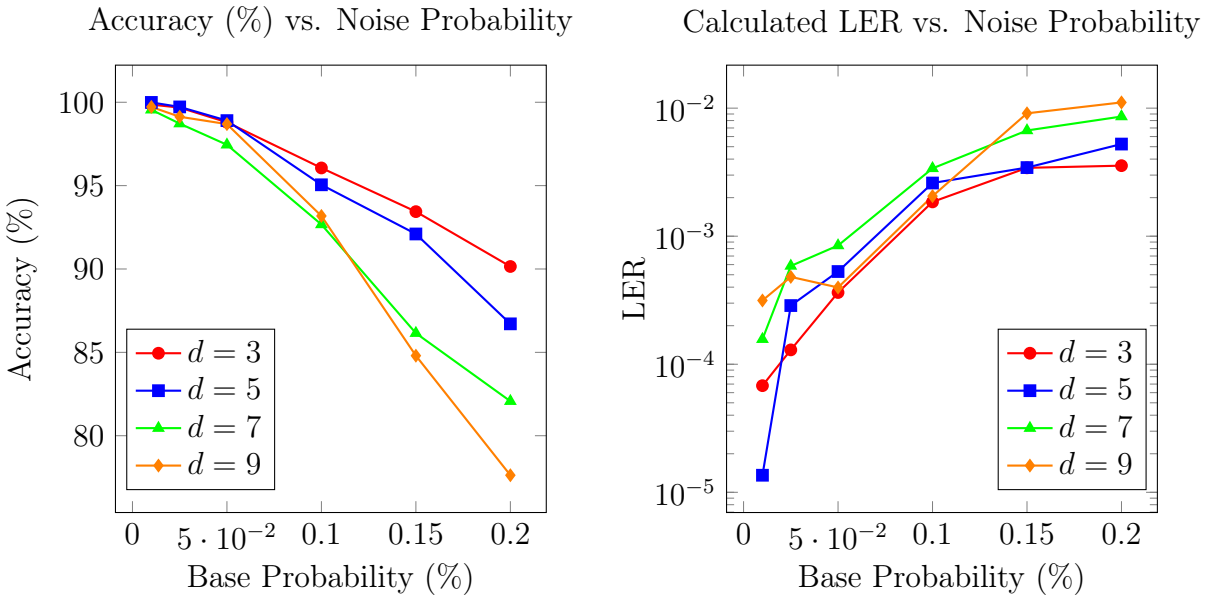


Figure 4.8: Accuracy and LER of the Transformer Decoder, $r = 25$

4.7 Summary

In this chapter, we introduce the neural network decoder, along with other types of decoders, like the maximum probability decoders and maximum likelihood decoders. Although neural network decoders are able to decode with high accuracy and low latency, it takes exponentially more training data to train such decoders for higher distance codes. We proposed two potential ways to improve the scalability of neural network surface code decoders. The first is the convolutional decoder, which repeatedly applies convolutions in the spirit of a renormalization decoder to reduce the size of the problem that the neural network has to solve. The second is the transformer decoder, which reduces the representation of the syndrome to just a few high dimensional vectors. By reducing the problem space that such a neural network decoder has to search, these techniques may reduce the dependency on vast training datasets. We also benchmark these proposed architectures and report the results.

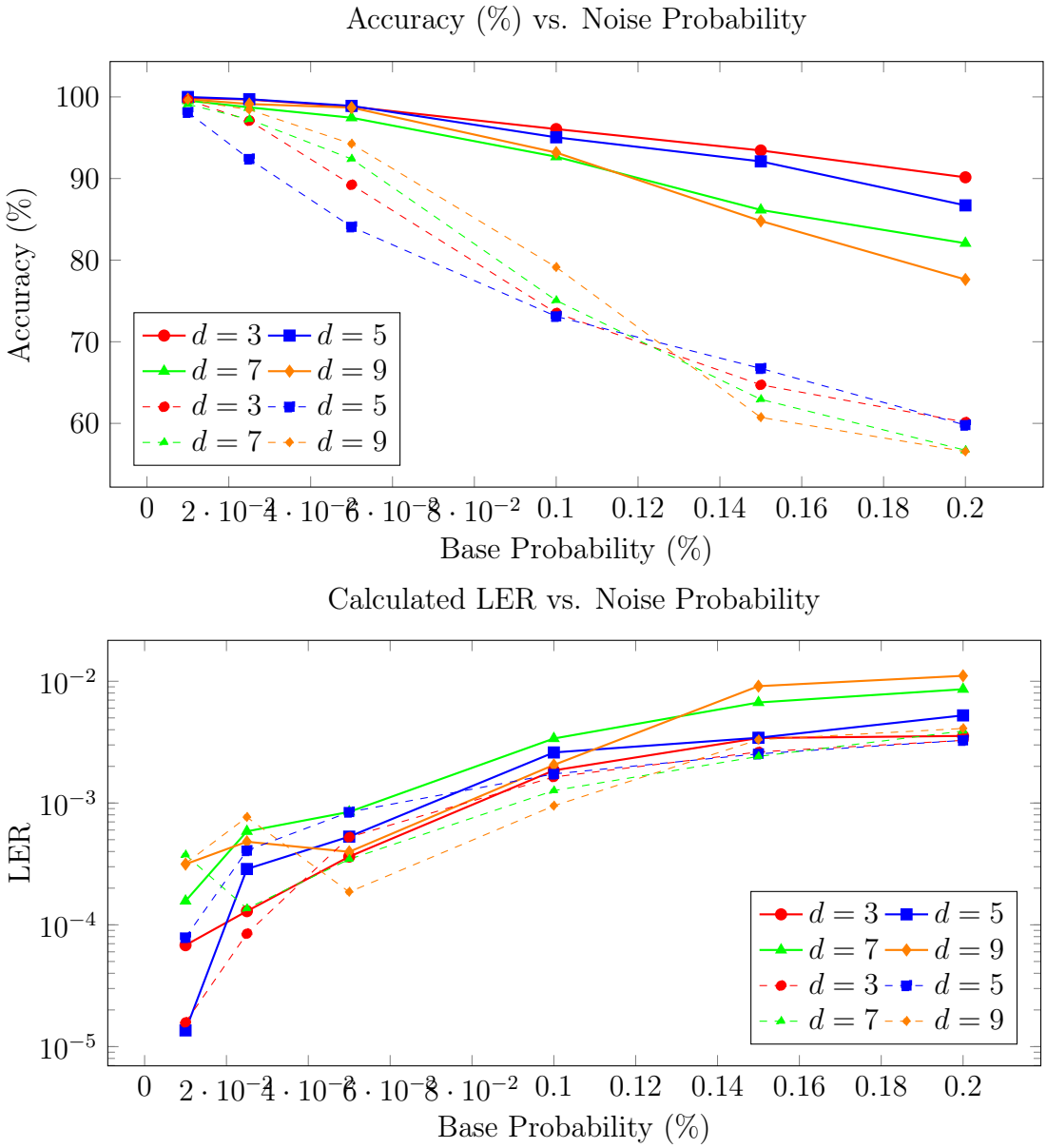


Figure 4.9: Accuracy and LER of the Transformer Decoder, $r = 25$ vs $r = 250$. Solid lines represent the tests done at $r = 25$, and dashed lines represent the tests done at $r = 250$.

Chapter 5

Conclusion

This section will include a discussion of the strengths and weaknesses of both decoders trained in the duration of this project. The threshold is a conventional metric for assessing decoder performance, where a decoder is benchmarked across various code distances and error rates. It represents the physical error rate at which scaling the code distance will not lead to better error correction. In the literature, this was usually done by training a strong model to decode each distance code. Unfortunately, the testing necessary to establish a reliable threshold was not achievable within the limited timeframe of this project, as it requires both decoders function well at various distances and noise probabilities. The convolutional decoder was unable to decode distances beyond what it was trained on, and both decoders are unable to scale well to high base noise probabilities. This aspect of decoder evaluation also remains an area for future investigation.

Ideally, an ablation study would also be performed, to study the contributions of various submodules to the overall effectiveness of both models. However, similar time constraints prevented such a study from ocurring.

Convolutional Decoder

As mentioned in Section 4.6, evaluating the convolutional at code distances different from the ones it was trained on revealed a tendency for the convolutional autoencoder in specific to overfit to the training data. This failure to generalize suggests that the autoencoder has not adequately learned to compress the syndrome information in general. One potential approach to reduce this overfitting could involve freezing the decoding layers of the network while further fine-tuning the convolutional layers. Alternatively, expanding the training dataset to include a variety of distance codes might help the network learn more generalizable features. Due to time constraints, it was not feasible to optimize the model for various distances during this study; thus, this task is recommended for future research.

We note that the convolutional decoder only saw about 110 million examples in total during training before convergence, an order of magnitude less examples than other comparable models in the literature [30], [31]. We may attribute this to two driving forces behind this behavior:

1. **Dimensionality Reduction:** The repeated applications of the convolutional autoen-

coder reduced the dimensionality of the problem, and also reduced the need for more training data.

2. **Fittedness of the Model:** The convolutional decoder could be underfit for the problem we are attempting to solve. However, this seems somewhat unlikely from viewing the loss curves from training (See Appendix A).

As the noise probability increased, there was pronounced performance degradation, meaning that the architecture is unable to handle high rates of noise, or more likely, that it is underfit for this level of noise. One possible avenue for improvement could be increasing the base error probability of the noise model as the neural network learns. This was tried briefly during training, but was applied too aggressively, leading the models to diverge.

We also note that one of the main strengths of Mamba is modeling very long sequences. In fact, its strength over transformers seems only to be shown at scale when processing long sequence data [16]. Therefore, in hindsight, it would not make much sense to use it over a transformer in the core of the convolutional decoder. A point of further investigation could be replacing the Mamba block with a transformer encoder layer.

Transformer Decoder

Although the transformer decoder achieved fairly low LER, it is clear that the decoders were held back by the amount of training that they received, especially the ones at high code distance. It also seems that the transformer encoder was underparameterized for its model type. Typically, a transformer needs a much wider and deeper architecture to be more effective than an LSTM or GRU. Therefore, it may be helpful in future works to add more transformer blocks, and widen them by increasing the embedding dimension, or consider simplifying the model to just use a LSTM or GRU.

We do note that this transformer decoder is one or two orders of magnitude faster than the convolutional decoder, and much more main memory friendly. This allowed the transformer decoder to be trained on batch sizes 8x larger than the convolutional decoder, effectively allowing it to see 8x as much data in the same amount of time. In fact, this decoder was able to see nearly 9 billion training examples in just two days. We may attribute this speed to two main factors:

1. **Dimensionality Reduction:** The reduction in dimensionality led to much faster training, since less gradients are accumulated during the forward pass, which speeds up the backwards pass.

2. **Simpler Operations:** In this architecture, we substitute computationally expensive spatially aware operations like the scatter, gather, and dilated convolutions with a simple 2D positional embedding. The custom scatter and gather operations are one of the main reasons why the convolutional decoder was so slow to train.

Further investigation could take the form of training a model on a lower distance code, then selectively fine-tuning the encoding module to utilize the already-trained feedforward network RNN core. We may also consider combining both convolutional and transformer decoding approaches.

Appendix A

Training Details

A.1 Hyperparameters

Hyperparameter	Value
Mamba State Dimension	16
Mamba Conv. Dimension	4
Mamba State Expansion	2
Number Mamba Blocks	3
Embedding Dimension	256
Model Dimension	256
Weight Decay	1e-9
Learning Rate	1e-6
Batch Size	256

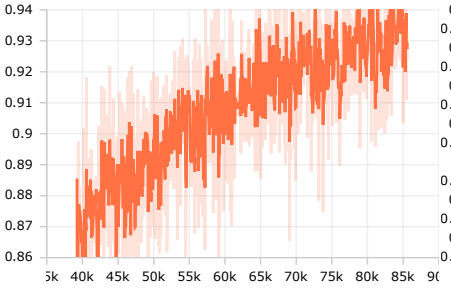
Figure A.1: Convolutional Hyperparameters

Hyperparameter	Value
Multi Head Attention Heads	4
Embedding Dimension	48
Model Dimension	256
Embedding Dimension	256
Weight Decay	0
Learning Rate	1e-6
Batch Size	2048

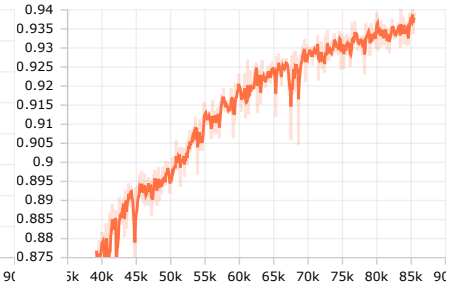
Figure A.2: Transformer Model Hyperparameters

A.2 Loss Curves

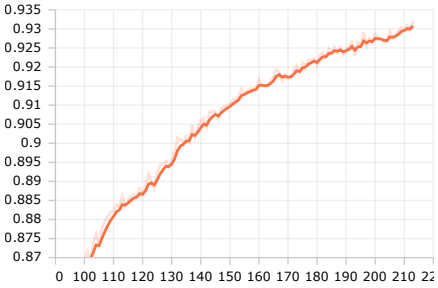
Convolutional Model



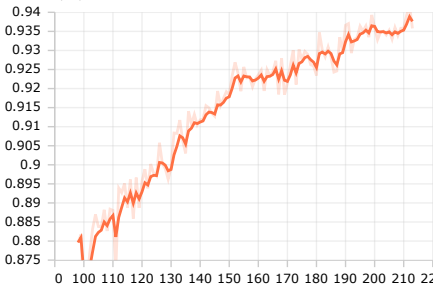
(a) Training Accuracy



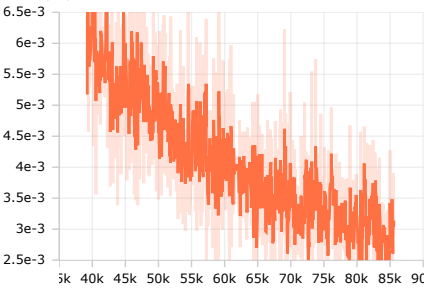
(b) Validation Accuracy



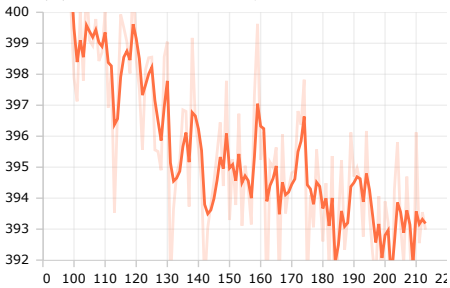
(c) Training Acc./Epoch



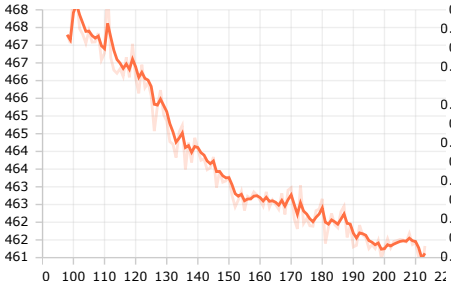
(d) Validation Acc./Epoch



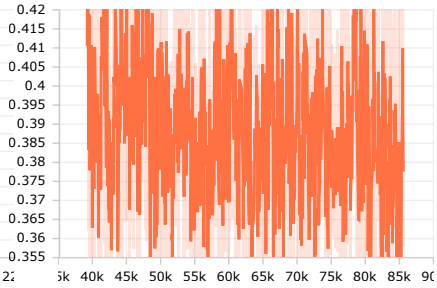
(e) Average LER



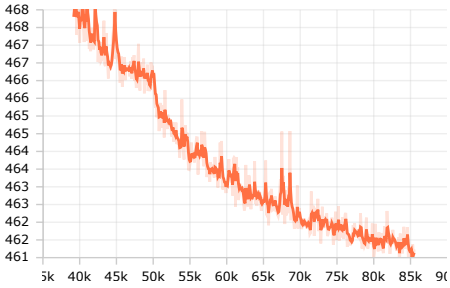
(f) Training Loss by Epoch



(g) Validation Loss by Epoch



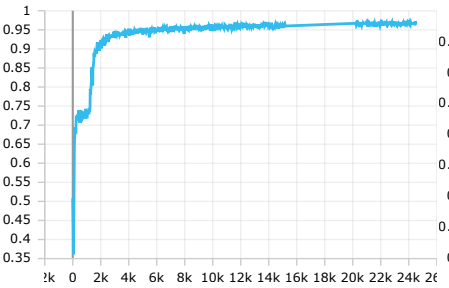
(h) Training Loss



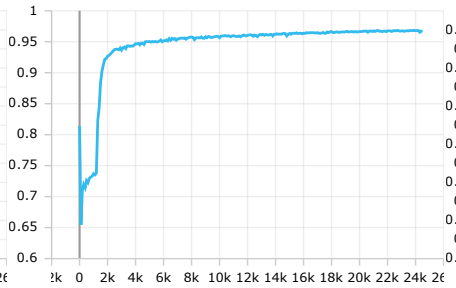
(i) Validation Loss

Transformer Model

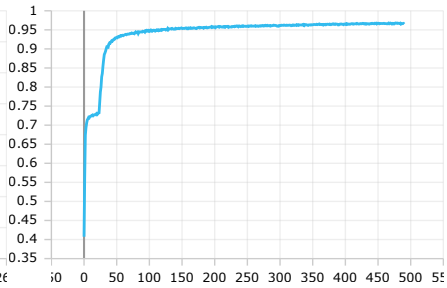
Distance 3 Model



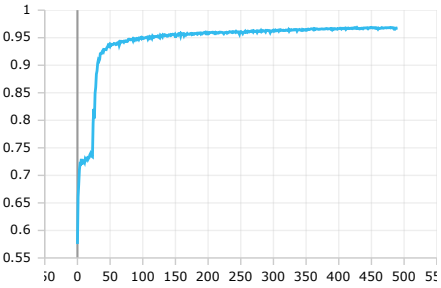
(a) Training Accuracy



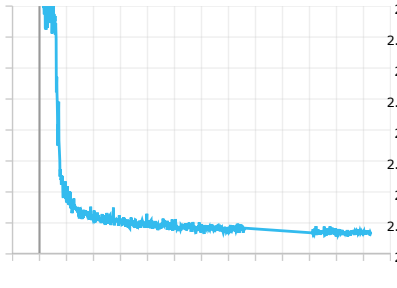
(b) Validation Accuracy



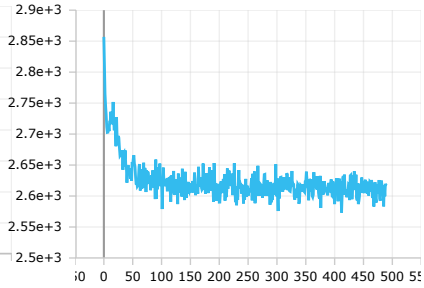
(c) Training Acc./Epoch



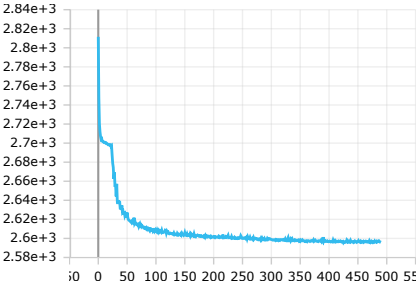
(d) Validation Acc./Epoch



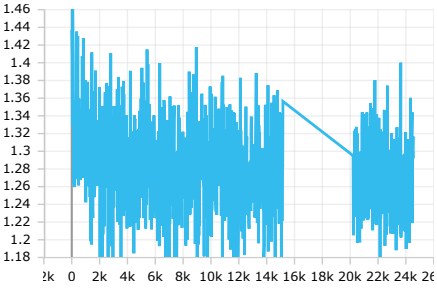
(e) Average LER



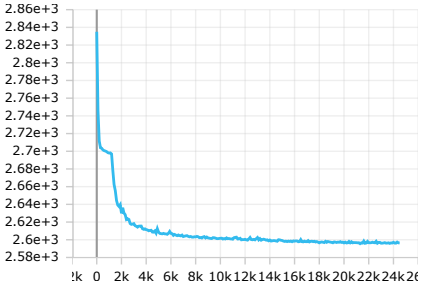
(f) Training Loss by Epoch



(g) Validation Loss by Epoch

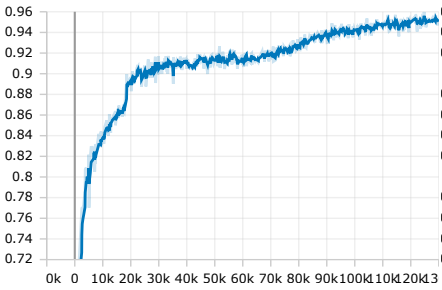


(h) Training Loss

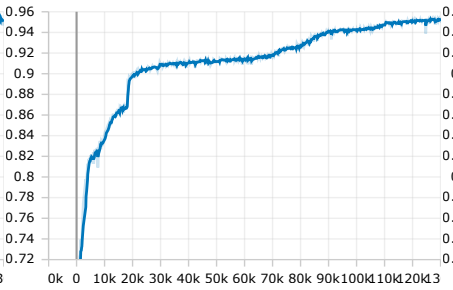


(i) Validation Loss

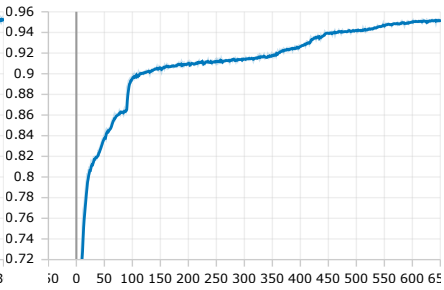
Distance 5 Model



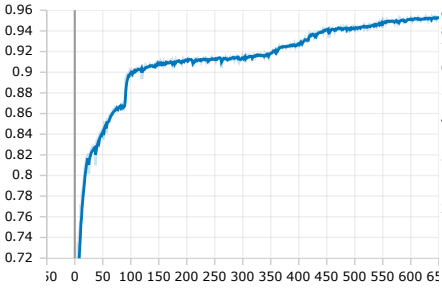
(a) Training Accuracy



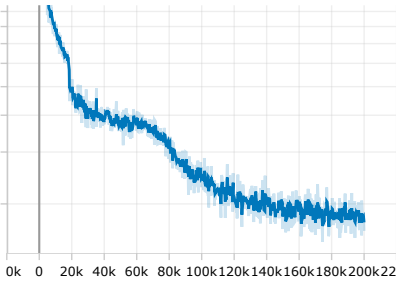
(b) Validation Accuracy



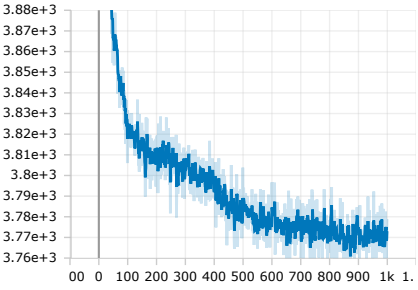
(c) Training Acc./Epoch



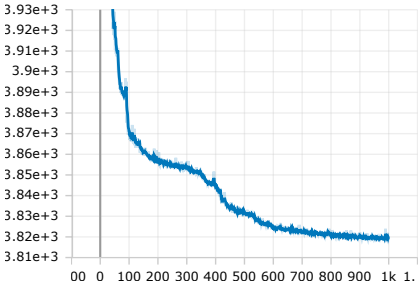
(d) Validation Acc./Epoch



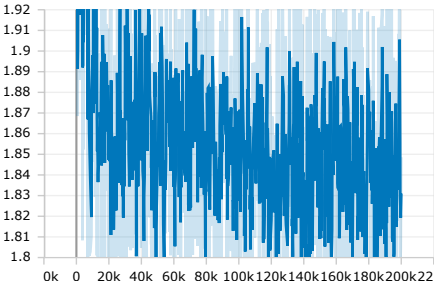
(e) Average LER



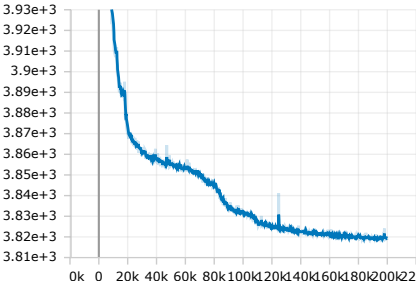
(f) Training Loss by Epoch



(g) Validation Loss by Epoch

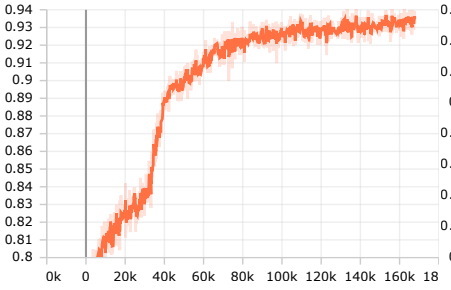


(h) Training Loss

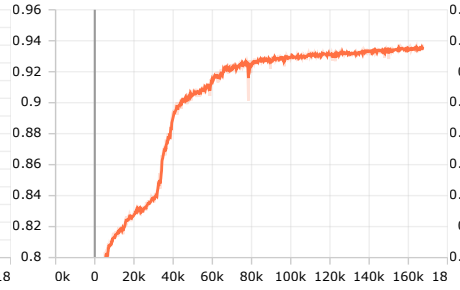


(i) Validation Loss

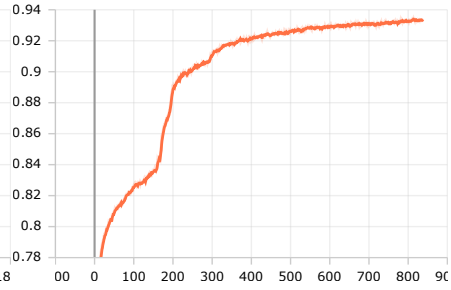
Distance 7 Model



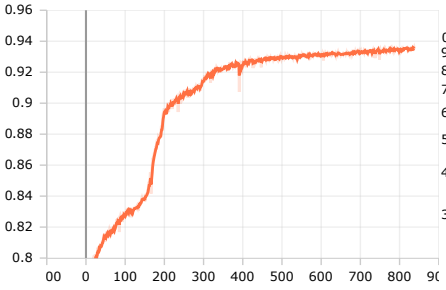
(a) Training Accuracy



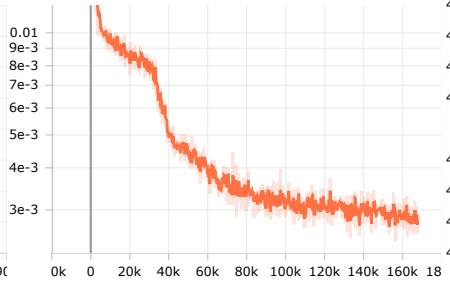
(b) Validation Accuracy



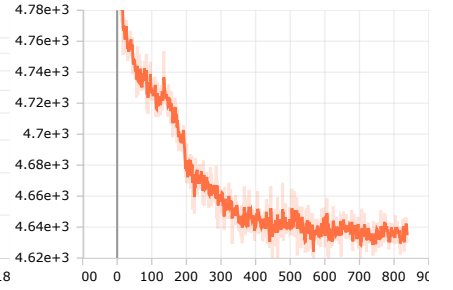
(c) Training Acc./Epoch



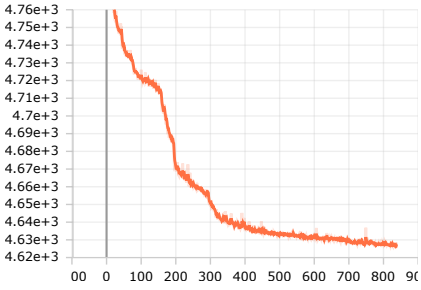
(d) Validation Acc./Epoch



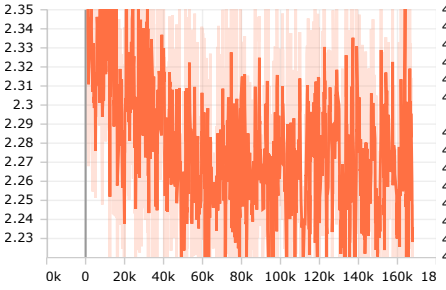
(e) Average LER



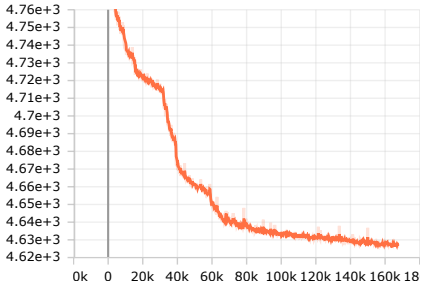
(f) Training Loss by Epoch



(g) Validation Loss by Epoch

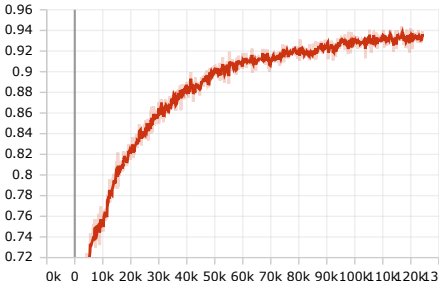


(h) Training Loss

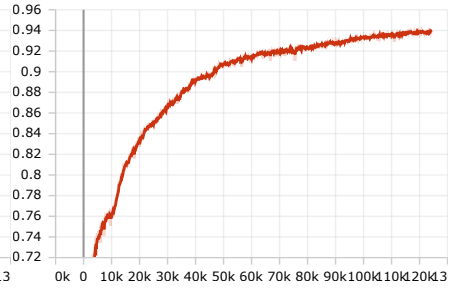


(i) Validation Loss

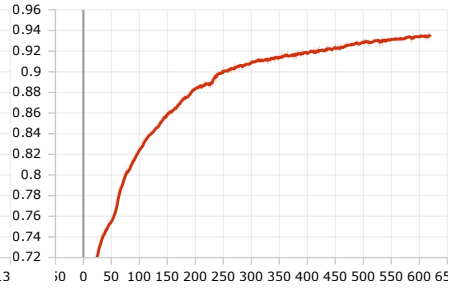
Distance 9 Model



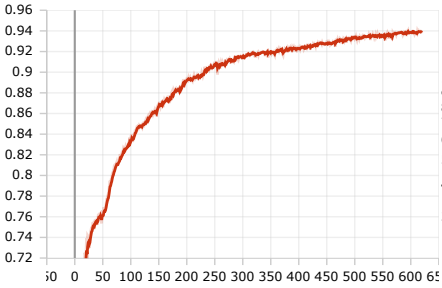
(a) Training Accuracy



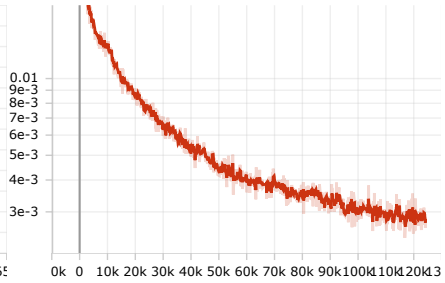
(b) Validation Accuracy



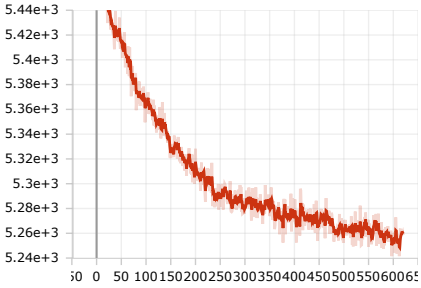
(c) Training Acc./Epoch



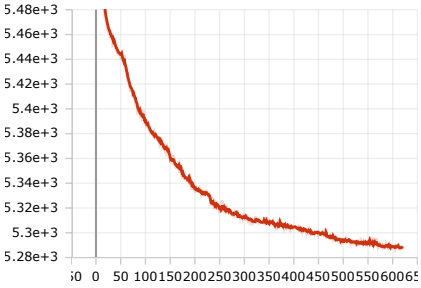
(d) Validation Acc./Epoch



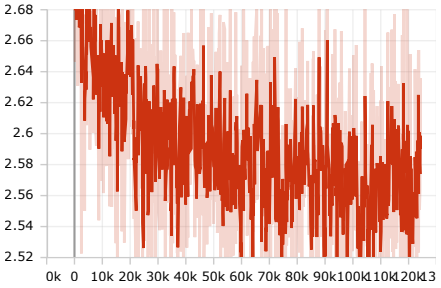
(e) Average LER



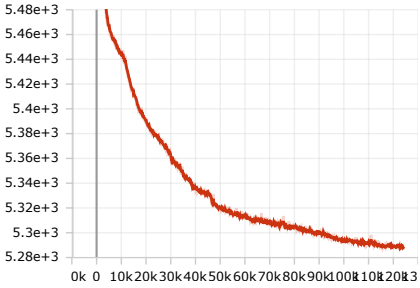
(f) Training Loss by Epoch



(g) Validation Loss by Epoch



(h) Training Loss



(i) Validation Loss

References

[1] R. Acharya, I. Aleiner, R. Allen, *et al.*, "Suppressing quantum errors by scaling a surface code logical qubit," *Nature*, vol. 614, no. 7949, pp. 676–681, Feb. 2023, ISSN: 1476-4687. DOI: 10.1038/s41586-022-05434-1. (visited on 05/05/2023).

[2] S. Ma, G. Liu, P. Peng, B. Zhang, S. Jandura, J. Claes, A. P. Burgers, G. Pupillo, S. Puri, and J. D. Thompson, "High-fidelity gates and mid-circuit erasure conversion in an atomic qubit," *Nature*, vol. 622, no. 7982, pp. 279–284, Oct. 2023, ISSN: 1476-4687. DOI: 10.1038/s41586-023-06438-1. (visited on 05/05/2024).

[3] T. Young, "I. The Bakerian Lecture. Experiments and calculations relative to physical optics," vol. 94, Dec. 1804, ISSN: 0261-0523. DOI: 10.1098/rstl.1804.0001. (visited on 03/30/2024).

[4] W. K. Wootters and W. H. Zurek, "A single quantum cannot be cloned," *Nature*, vol. 299, no. 5886, pp. 802–803, Oct. 1982, ISSN: 1476-4687. DOI: 10.1038/299802a0. (visited on 05/02/2024).

[5] A. M. Steane, "Error Correcting Codes in Quantum Theory," *Physical Review Letters*, vol. 77, no. 5, pp. 793–797, Jul. 1996. DOI: 10.1103/PhysRevLett.77.793. (visited on 05/05/2023).

[6] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, "Surface codes: Towards practical large-scale quantum computation," *Physical Review A*, vol. 86, no. 3, p. 032 324, Sep. 2012, ISSN: 1050-2947, 1094-1622. DOI: 10.1103/PhysRevA.86.032324. (visited on 05/05/2023).

[7] D. Gottesman, *Stabilizer Codes and Quantum Error Correction*, May 1997. DOI: 10.48550/arXiv.quant-ph/9705052. arXiv: quant-ph/9705052. (visited on 03/26/2024).

[8] H. Cao, F. Pan, Y. Wang, and P. Zhang, *qecGPT: Decoding Quantum Error-correcting Codes with Generative Pre-trained Transformers*, Jul. 2023. arXiv: 2307.09025 [cond-mat, physics:quant-ph, stat]. (visited on 02/22/2024).

[9] D. Horsman, A. G. Fowler, S. Devitt, and R. V. Meter, "Surface code quantum computing by lattice surgery," *New Journal of Physics*, vol. 14, no. 12, p. 123 011, Dec. 2012, ISSN: 1367-2630. DOI: 10.1088/1367-2630/14/12/123011. (visited on 11/14/2023).

[10] E. T. Campbell, B. M. Terhal, and C. Vuillot, "Roads towards fault-tolerant universal quantum computation," *Nature*, vol. 549, no. 7671, pp. 172–179, Sep. 2017, ISSN: 1476-4687. DOI: 10.1038/nature23460. (visited on 05/05/2024).

[11] D. Litinski, "A Game of Surface Codes: Large-Scale Quantum Computing with Lattice Surgery," *Quantum*, vol. 3, p. 128, Mar. 2019, ISSN: 2521-327X. DOI: 10.22331/q-2019-03-05-128. arXiv: 1808.02892 [cond-mat, physics:quant-ph]. (visited on 10/17/2023).

[12] O. Ezratty, "Perspective on superconducting qubit quantum computing," *The European Physical Journal A*, vol. 59, no. 5, p. 94, May 2023, ISSN: 1434-601X. DOI: 10.1140/epja/s10050-023-01006-7. (visited on 03/30/2024).

[13] *IBM Debuts Next-Generation Quantum Processor & IBM Quantum System Two, Extends Roadmap to Advance Era of Quantum Utility*, https://newsroom.ibm.com/2023-12-04-IBM-Debuts-Next-Generation-Quantum-Processor-IBM-Quantum-System-Two,-Extends-Roadmap-to-Advance-Era-of-Quantum-Utility. (visited on 05/09/2024).

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. ukasz Kaiser, and I. Polosukhin, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017. (visited on 05/03/2024).

[15] A. Gu, K. Goel, and C. Ré, *Efficiently Modeling Long Sequences with Structured State Spaces*, Aug. 2022. DOI: 10.48550/arXiv.2111.00396. arXiv: 2111.00396 [cs]. (visited on 05/05/2024).

[16] A. Gu and T. Dao, "Mamba: Linear-Time Sequence Modeling with Selective State Spaces,"

[17] O. Higgott and C. Gidney, *Sparse Blossom: Correcting a million errors per core second with minimum-weight matching*, Mar. 2023. DOI: 10.48550/arXiv.2303.15933. arXiv: 2303.15933 [quant-ph]. (visited on 07/14/2023).

[18] Y. Wu and L. Zhong, *Fusion Blossom: Fast MWPM Decoders for QEC*, May 2023. arXiv: 2305.08307 [quant-ph]. (visited on 07/11/2023).

[19] P. Das, A. Locharla, and C. Jones, "LILLIPUT: A lightweight low-latency lookup-table decoder for near-term Quantum error correction," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Lausanne Switzerland: ACM, Feb. 2022, pp. 541–553, ISBN: 978-1-4503-9205-1. DOI: 10.1145/3503222.3507707. (visited on 07/22/2023).

[20] N. Liyanage, Y. Wu, A. Deters, and L. Zhong, *Scalable Quantum Error Correction for Surface Codes using FPGA*, May 2023. DOI: 10.48550/arXiv.2301.08419. arXiv: 2301.08419 [quant-ph]. (visited on 07/11/2023).

[21] O. Higgott, T. C. Bohdanowicz, A. Kubica, S. T. Flammia, and E. T. Campbell, *Improved decoding of circuit noise and fragile boundaries of tailored surface codes*, Jul. 2023. arXiv: 2203.04948 [quant-ph]. (visited on 07/22/2023).

[22] C. T. Chubb and S. T. Flammia, "Statistical mechanical models for quantum codes with correlated noise," *Annales de l'Institut Henri Poincaré D*, vol. 8, no. 2, pp. 269–321, May 2021, ISSN: 2308-5827. DOI: 10.4171/AIHPD/105. arXiv: 1809.10704 [cond-mat, physics:quant-ph]. (visited on 10/15/2023).

[23] C. T. Chubb, *General tensor network decoding of 2D Pauli codes*, Oct. 2021. DOI: 10.48550/arXiv.2101.04125. arXiv: 2101.04125 [quant-ph]. (visited on 08/29/2023).

[24] C. Piveteau, C. T. Chubb, and J. M. Renes, *Tensor Network Decoding Beyond 2D*, Oct. 2023. arXiv: 2310.10722 [quant-ph]. (visited on 12/23/2023).

[25] G. Duclos-Cianci and D. Poulin, *A renormalization group decoding algorithm for topological quantum codes*, Jun. 2010. arXiv: 1006.1362 [quant-ph]. (visited on 08/28/2023).

[26] K. Duivenvoorden, N. P. Breuckmann, and B. M. Terhal, "Renormalization group decoder for a four-dimensional toric code," *IEEE Transactions on Information Theory*, vol. 65, no. 4, pp. 2545–2562, Apr. 2019, ISSN: 0018-9448, 1557-9654. DOI: 10.1109/TIT.2018.2879937. arXiv: 1708.09286 [quant-ph]. (visited on 04/23/2024).

[27] G. Duclos-Cianci and D. Poulin, "Fast Decoders for Topological Quantum Codes," *Physical Review Letters*, vol. 104, no. 5, p. 050 504, Feb. 2010, ISSN: 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.104.050504. arXiv: 0911.0581 [cond-mat, physics:hep-th, physics:quant-ph]. (visited on 04/22/2024).

[28] S. Varsamopoulos, B. Criger, and K. Bertels, "Decoding small surface codes with feed-forward neural networks," *Quantum Science and Technology*, vol. 3, no. 1, p. 015 004, Nov. 2017, ISSN: 2058-9565. DOI: 10.1088/2058-9565/aa955a. (visited on 05/21/2023).

[29] S. Varsamopoulos, K. Bertels, and C. G. Almudever, "Comparing Neural Network Based Decoders for the Surface Code," *IEEE Transactions on Computers*, vol. 69, no. 2, pp. 300–311, Feb. 2020, ISSN: 1557-9956. DOI: 10.1109/TC.2019.2948612.

[30] J. Bausch, A. W. Senior, F. J. H. Heras, *et al.*, *Learning to Decode the Surface Code with a Recurrent, Transformer-Based Neural Network*, Oct. 2023. arXiv: 2310.05900 [quant-ph]. (visited on 12/19/2023).

[31] H. Wang, P. Liu, K. Shao, D. Li, J. Gu, D. Z. Pan, Y. Ding, and S. Han, *Transformer-QEC: Quantum Error Correction Code Decoding with Transferable Transformers*, Nov. 2023. DOI: 10.48550/arXiv.2311.16082. arXiv: 2311.16082 [quant-ph]. (visited on 02/22/2024).

[32] M. Lange, P. Havström, B. Srivastava, V. Bergentall, K. Hammar, O. Heuts, E. van Nieuwenburg, and M. Granath, *Data-driven decoding of quantum error correcting codes using graph neural networks*, Jul. 2023. arXiv: 2307.01241 [quant-ph]. (visited on 09/25/2023).

[33] A. deMarti iOlius, P. Fuentes, R. Orús, P. M. Crespo, and J. E. Martinez, *Decoding algorithms for surface codes*, Jul. 2023. arXiv: 2307.14989 [quant-ph]. (visited on 08/28/2023).

[34] X. Ni, "Neural Network Decoders for Large-Distance 2D Toric Codes," *Quantum*, vol. 4, p. 310, Aug. 2020, ISSN: 2521-327X. DOI: 10.22331/q-2020-08-24-310. arXiv: 1809.06640 [quant-ph]. (visited on 07/03/2023).

[35] S. Varsamopoulos, K. Bertels, and C. G. Almudever, "Decoding surface code with a distributed neural network–based decoder," *Quantum Machine Intelligence*, vol. 2, no. 1, p. 3, Mar. 2020, ISSN: 2524-4914. DOI: 10.1007/s42484-020-00015-9. (visited on 04/24/2024).

[36] S. Gicev, L. C. L. Hollenberg, and M. Usman, *A scalable and fast artificial neural network syndrome decoder for surface codes*, Jun. 2023. DOI: 10.48550/arXiv.2110.05854. arXiv: 2110.05854 [quant-ph]. (visited on 07/03/2023).

[37] S. J. Evered, D. Bluvstein, M. Kalinowski, *et al.*, "High-fidelity parallel entangling gates on a neutral-atom quantum computer," *Nature*, vol. 622, no. 7982, pp. 268–272, Oct. 2023, ISSN: 1476-4687. DOI: 10.1038/s41586-023-06481-y. (visited on 05/04/2024).

[38] C. Gidney, "Stim: A fast stabilizer circuit simulator," *Quantum*, vol. 5, p. 497, Jul. 2021. DOI: 10.22331/q-2021-07-06-497. (visited on 06/24/2023).

[39] C. Gidney, M. Newman, A. Fowler, and M. Broughton, "A Fault-Tolerant Honeycomb Memory," *Quantum*, vol. 5, p. 605, Dec. 2021. DOI: 10.22331/q-2021-12-20-605. (visited on 01/24/2024).