3-2022

Evaluating Neural Network Decoder Performance for Quantum Error Correction Using Various Data Generation Models

Brett M. Martin

EVALUATING NEURAL NETWORK DECODER PERFORMANCE FOR QUANTUM ERROR CORRECTION USING VARIOUS DATA GENERATION MODELS

THESIS

Brett M. Martin, 2d Lt, USAF

AFIT-ENG-MS-22-M-044

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-22-M-044

EVALUATING NEURAL NETWORK DECODER PERFORMANCE FOR QUANTUM ERROR CORRECTION USING VARIOUS DATA GENERATION MODELS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Brett M. Martin, B.S.

2d Lt, USAF

March 24, 2021

AFIT-ENG-MS-22-M-044

EVALUATING NEURAL NETWORK DECODER PERFORMANCE FOR QUANTUM ERROR CORRECTION USING VARIOUS DATA GENERATION MODELS

THESIS

Brett M. Martin, B.S.
2d Lt, USAF

Committee Membership:

Laurence D. Merkle, Ph.D
Chair

David E. Weeks, Ph.D
Member

Douglas D. Hodson, Ph.D
Member

AFIT-ENG-MS-22-M-044

Abstract

Neural networks have been shown in the past to perform quantum error correction (QEC) decoding with greater accuracy and efficiency than algorithmic decoders. Because the qubits in a quantum computer are volatile and only usable on the order of milliseconds before they decohere, a means of fast quantum error correction is necessary in order to correct data qubit errors within the time budget of a quantum algorithm. Algorithmic decoders are good at resolving errors on logical qubits with only a few data qubits, but are less efficient in systems containing more data qubits. With neural network decoders, practical quantum computation becomes much more realizable since the error corrective operations are calculated much faster than with the MWPM or partial lookup table implementations. This research is aimed at furthering neural network QEC decoder research by generating exhaustive and randomly sampled data sets using high-performance computing algorithms to evaluate the effect of data set generation methods on the effectiveness of these neural networks compared to similar models. The results of this work show that different data sets affect various performance metrics including accuracy, F1 score, area under the receiver operating characteristic curve, and QEC cycles.

Acknowledgements

To my parents—thank you for the unconditional love and support. My whole life, you've repeatedly demonstrated for me the value of hard work and integrity. If not for the example you set for me, I would not be where I am today.

To my pastor—thank you for being such a phenomenal mentor. Nobody has expressed interest in my research quite the way you have. I appreciate all of the conversations we've had about not just the fine details of my thesis, but about all of the related experiences from your tenure at AT&T Bell Laboratories, as well.

To my advisor, Dr. Merkle—thank you for all of the help and assurance you've given me throughout my graduate school experience. From all of your classes I attended to my actual thesis research, you always reassured me whenever I was unsure about my work.

To Dr. Weeks—thank you for all of the instruction both in and out of the classroom. Your ability to describe complicated subject matter in a way that is easy to understand is truly unparalleled.

To Dr. Hodson—thank you for the instruction and mentorship. I never expected to be able to travel to present my research at a conference, but you made it happen. It was an experience that I'll never forget.

Lastly, to my kind, supportive, and all-around amazing wife—thank you for your patience and understanding throughout my graduate studies. I thank the Lord my God every day for bringing you into my life.

Brett M. Martin

Table of Contents

Page

List of Figures

Figure Page

List of Tables

Table Page

EVALUATING NEURAL NETWORK DECODER PERFORMANCE FOR QUANTUM ERROR CORRECTION USING VARIOUS DATA GENERATION MODELS

I. Introduction

1.1 Motivation

In 1994, Bell Labs researcher Peter Shor developed an algorithm that was intended to run on a then-theoretical quantum computer. The algorithm, aptly named Shor's algorithm, was developed as a means of factoring large integers into their prime factors in polynomial time—a feat otherwise impossible to achieve even by today's classical computers, which are only capable of prime factorization in non-deterministic polynomial time [26]. At the time of it's formulation, the algorithm existed solely as a theoretical procedure as the hardware necessary to produce an experimental demonstration did not yet exist. In fact, the first experimental demonstration of the algorithm was not accomplished until 2001 when NASA successfully exhibited the factorization of the number 15 using the nuclei in a molecule as qubits, or quantum binary digits [29]. Although there is no practical use for a polynomial time speedup of the factoring of small numbers, the implications of this experiment suggest that it is possible to achieve prime factorization for large numbers used in most public-key cryptography.

The measures that keep virtually every aspect of digital commerce secure from cyber threats rely on the difficulty of prime factorization of large numbers. Like most modern public-key cryptography, the RSA algorithm—named after Ronald Rivest, Adi Shamir, and Leonard Adleman, the scientists who created it—uses a class of mathematics called modular arithmetic to produce public and private keys that are used to encrypt and decrypt information. The prime numbers used to generate these keys are too large to factor in any reasonable amount of time with any modern hardware. Perhaps one of the most significant implications of a technology capable of factoring the integers used in public-key cryptosystems like RSA is the clear national security risk that such a technological advantage would pose for any adversary. When cryptosystems are susceptible to exploit in any way, the security of any electronic communication is effectively worthless.

The clear national security risks notwithstanding, quantum computers offer the potential for a means of cryptographic data transmission impervious to any exploitation. In a 2020 interview, Peter Shor suggested that such a cryptosystem is absolutely necessary to maintain internet security once quantum computers are capable of breaking RSA [9]. It is for these aforementioned reasons that the United States, along with our peer adversaries Russia and China, have placed such a high priority on quantum computer research and development.

The general consensus among agencies and departments within the U.S. government is that there is a significant potential in the area of quantum information science. The National Strategic Overview for Quantum Information Science, published by the National Science and Technology Council, recognizes that quantum information science research and development shows great promise in areas of defense infrastructure, national defense, and drive international cooperation [23]. The National Quantum Initiative, a federal program signed into law by the 2018 National Quantum Initiative Act, was created by congress in order to provide awareness of quantum information science and coordinate research and development opportunities in the field. The initiative ensures the continuous advancement of this novel technology for the purpose of U.S. national and economic security [5].

Unfortunately, the road to quantum computer innovation is beset by a variety of barriers. By nature of the quantum-mechanical components of these machines, data is only usable for short periods of time and virtually every aspect of the system from the environment to the qubits are subject to a variety of errors. This age of volatile quantum computer systems is known colloquially as the NISQ (noisy intermediate-scale quantum) era [24]. This research aims to provide insight into the use of a sub-discipline of artificial intelligence, neural networks, as a means of correcting the abundance of errors in quantum computers.

1.2 Problem Background

Quantum computers today are constructed using a wide range of methods. In fact, a qubit can be represented by anything with a binary quantum-mechanical state. Today, some of the most widely-used quantum computers include ion traps, which use electron spin states of trapped ions to represent qubits; photonic processors, which use the phase of photons as qubit values; and superconducting quantum computers, which store qubit data as a charge in a more complex adaptation of a harmonic oscillator circuit called a transmon (transmission line shunted plasma oscillation qubit) [8]. While some platforms have advantages over others in terms of qubit resiliency, scalability, and fabrication costs, they are all considered noisy systems that produce large quantities of error. In order to reduce the prevalence of error in any quantum computer, quantum error correction must be implemented in some form.

The quantum computer stack consists of several layers of abstraction that represent high- to low-level components within a quantum computer and encompasses everything from the quantum algorithm to the actual hardware [13]. Within this stack, it is possible to implement quantum error correction on different layers, where corrections can be applied to individual physical qubits, which is considered high-level decoding, as well as on logical qubits that consist of multiple physical qubits, which is referred to as low-level decoding [30]. Regardless of the approach, the task of performing algorithmic quantum error correction is computationally complex and requires more time to accomplish as the number of qubits in the system increase. Past a certain threshold, it is not feasible to implement such error correction methods without exceeding the time budget, or the amount of time that can elapse before the qubits in the system begin to decohere and become unusable. It is possible to remedy the time complexity problem by implementing neural networks, which are capable of operating much faster than algorithmic error correction methods. Although neural networks have been implemented in past research as both high- and low-level decoders for quantum error correction [30][31][21][19], this research seeks to provide a better understanding of low-level neural network decoders by extending and elaborating on previous designs [7].

1.3 Research Objectives

This research is aimed at testing the following hypotheses in support of a broader research question, which is as follows:

Research Question:

How does the performance of neural network-based decoders for the surface code with various data set generation techniques compare to traditional algorithmic error correction decoders?

Hypotheses:

1. Using exhaustive training data sets produce a neural network decoder that performs more effectively and efficiently than decoders used in previous research.

2. Training a neural network on data sets that are randomly-sampled without replacement yield greater effectiveness and efficiency than those trained on randomized data sets with duplicate samples.

3. Training a neural network on larger randomly sampled data sets produce neural networks that have greater effectiveness and efficiency than those trained on small data sets.

The research question serves as the basis for the work documented in this thesis, which aims to provide a more comprehensive understanding of the effect of data set generation method on the performance of neural network decoders. Previous work has identified various effective high- and low-level decoding methods for quantum error correction using neural networks, but there has been little discourse related to the qualities and diversity of the data sets used in the training of those networks.

1.4 Scope

In the research conducted by Badger [7], which serves as the precursor to the research performed in this thesis, the noise model was assumed to follow the depolarizing noise model, where the error rate for the various types of error for all qubits in a system can happen with equal probability. Although this is not typical of a legitimate quantum computer, it simplifies the problem by making the error probability uniform across the circuit. The other limitation described in previous research involved limiting the scope to surface codes of depths three, five, and seven, since training a neural network on surface codes of higher depths would require too much time to accomplish within the time frame of this research. This research maintains these assumptions on the same grounds.

In this thesis, the scope of the data set is limited to three primary groups: exhaustive data sets and randomly sampled data sets with and without duplicate sample values. Although data sets containing all possible error configurations for a surface code would be ideal for all three surface code depths explored in this research, hardware limitations prevent the collection of exhaustive data sets for surface codes of depths larger than three. For that purpose, randomly sampled data sets of various sizes and composition are also obtained and used in the implementation of neural network decoder models.

1.5 Document Overview

This paper is organized as follows. Chapter II outlines the prerequisite knowledge related to this research, as well as a literature review of previous related work. Chapter III contains details on the implementation of a neural network quantum error correction decoder. Chapter IV lists the findings of the aforementioned neural network decoder implementation. Lastly, Chapter V provides a brief synopsis of the findings of this research as well as a section containing possible directions for future work.

II. Background and Literature Review

This chapter outlines the prerequisite knowledge pertaining to parallel algorithms and neural network-based decoders for quantum error correction. Section 2.1 covers high-performance computing and the nuances surrounding parallel algorithm design and implementation. In section 2.2, the quantum computing stack is broken down into its primary components, starting with the fundamentals of basic quantum physics, the differences between classical and quantum computers, and the qubit. The section further elaborates on these concepts by explaining the function of quantum gates, quantum algorithm or circuit design, and quantum error correction. This section also includes the topic of decoders and the various ways in which they can be implemented for the purpose of quantum error correction. Such decoding methods outlined in this section include look-up tables, algorithmic decoding schemes, and neural network-assisted decoding. In section 2.3, the topic of neural networks is explained, from the basic elements to the many different ways in which they can be implemented. Lastly, section 2.4 provides a summary of the previous research that has already been conducted in the area of neural network decoders for the purpose of quantum error correction.

2.1 High-Performance Computers

High-performance computers—which are often colloquially referred to as super-computers—are computer systems that are capable of performing computationally complex calculations and processing large volumes of data, often by means of parallel processing. High-performance computing is a crucial tool in many areas of research including aeronautical engineering, artificial intelligence, and physics, to name a few. This section describes all of the qualities that distinguish a high-performance computer from computers that are ubiquitous in our everyday lives.

2.1.1 High-Performance Computer Architectures

High-performance computer architectures offer a number of advantages over traditional architectures. Such advantages not only include improved spatial capacity such as disk and memory space, but also a multiplicity of resources such as processor cores and data paths [14]. Although most modern low-performance computer processors are capable of instruction-level parallelism and multi-threading, most high-performance computers add another level of abstraction by grouping one or more set of resources into a set of nodes, or a cluster [6]. Clusters are also capable of communicating among each other and writing data to shared memory spaces, if available, across one or more data paths belonging to an overarching interconnection network.

2.1.1.1 Control Structure

The control structure of a high-performance computer determines in what way a program is executed. The two primary control structures are SIMD (single instruction stream, multiple data stream) and MIMD (multiple instruction stream, multiple data stream). In a SIMD control structure, the same sequential lines of code are executed on each process simultaneously, with a single control unit responsible for iterating over the instructions in a program. With MIMD control structures, each processing element has it's own control unit, which allows for code one one processor to execute independently from another. A specific, and often used type of MIMD architecture is the SPMD (single program, multiple data stream) control structure. With a SPMD control structure, the same program is executed on all processes, which requires software-level logic to restrict which processes are able to execute certain components of the program [14]. Figure 1 shows a graphical representation of both a SIMD and MIMD control structure.

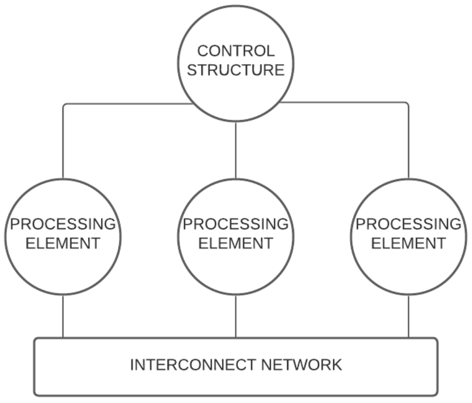
2.1.1.2 Memory Structure

The classification of memory structure in a high-performance computer can be categorized into one of two access types: uniform and non-uniform. With uniform memory access, or UMA, the amount of time it takes for one process to access information in memory is always the same for any other process. Conversely, non-uniform memory accesses, which are referred to as NUMA, vary in duration depending on which process is trying to access the information and from which memory location it is requesting the information. The distinction between UMA and NUMA are illustrated in figure 2.

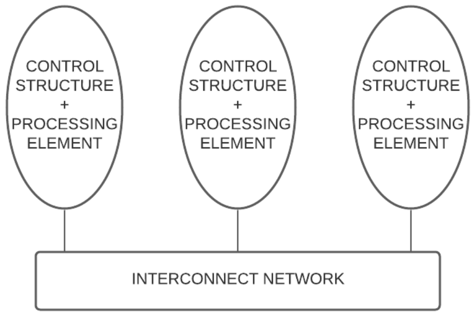
High-performance computers will sometimes use a distributed memory scheme where the same memory space is accessible from all nodes. Other architectures use node-specific memory spaces that are accessible only from a single process node. Hybrid memory models make use of both approaches by allowing all processes to access a common memory space, but also restrict access to certain other memory stores to the processes within that node.

2.1.1.3 Communications

It is often times necessary in the design of high-performance computing algorithms for data to be shared between nodes across the architecture. Data from a process in one node, for example, might need to be sent to another in order to be written out to a file. This message-passing is done across the aforementioned interconnection network, which connects nodes to one another in a variety of topologies ranging from fully-connected star networks to arrays with connections only between consecutive nodes. Arrays and meshes, which connect neighboring nodes, can also implement a

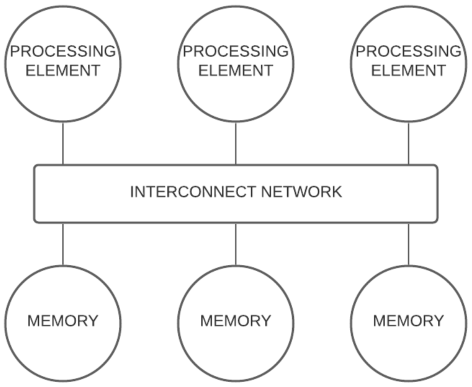


(a) SIMD control structure

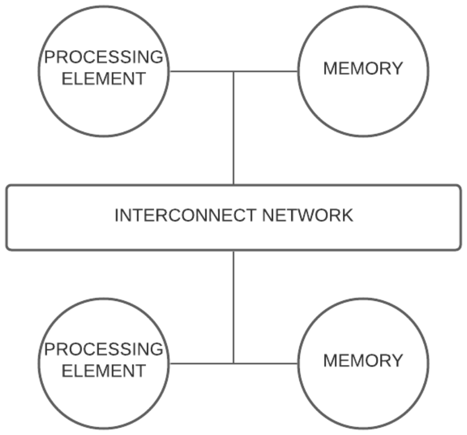

CONTROL STRUCTURE + PROCESSING ELEMENT
CONTROL STRUCTURE + PROCESSING ELEMENT
CONTROL STRUCTURE + PROCESSING ELEMENT
INTERCONNECT NETWORK

(b) MIMD control structure

Figure 1: Graphical representation of the SIMD and MIMD control structures, modified from [14]



(a) Uniform memory access



(b) Non-uniform memory access

Figure 2: Graphical representation of UMA and NUMA structures, modified from [14]

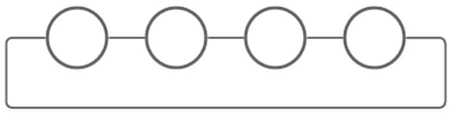
means of connection where edges along like axes are connected in a wrap-around way. The array and mesh topologies are shown in figure 3.

Hypercubes, or topologies that can exist in various dimensions, are special versions of mesh networks. Hypercubes are usually referred to as *k-d* mesh networks, where *k* is the number of nodes in a single dimension, *d*. Additionally, a hypercube network will have *logp* dimensions and, conversely, the number of nodes in a *d-dimensional* hypercube is equal to 2^d. Several hypercube networks with varying dimensionalities are shown in figure 4.

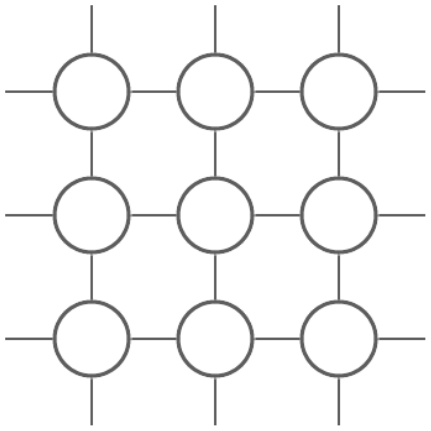
The tree is another common network topology in high-performance computing. In a tree network, each node on a tree structure refers to either a process node, which contains the actual processors used in executing programs, or a switching element for message-passing. In a fat tree, like the one pictured in figure 5, the communication bottlenecks that occur as information is passed from various nodes up the interconnect network simultaneously are eliminated by increasing the bandwidth of the data path inversely proportionate to the level of the tree. Such an architecture is referred to as non-blocking, since it reduces the blocking that would ordinarily occur with a normal tree with the same bandwith between nodes at each level of the network.

2.1.2 Parallel Algorithm Design

In the design of parallel algorithms, there are a number of factors that must be considered. Perhaps the most important first step in creating a parallel algorithm is to define the problem. For instance, some variation of a merge sort algorithm would require a divide-and-conquer approach, while an algorithm responsible for summing the contents of the rows in a two-dimensional array can be approached in a number of different ways. The decomposition of an algorithm into various processes is entirely dependent on how components of an algorithm that are capable of operating con-

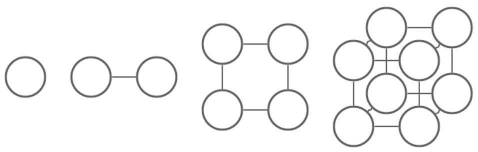


(a) Array

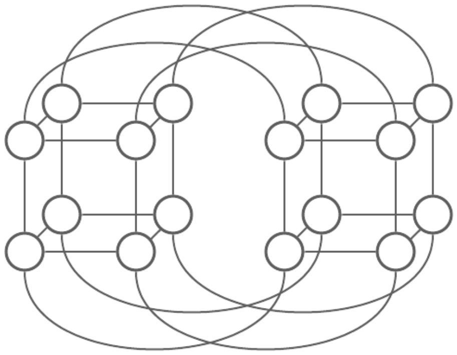


(b) Mesh

Figure 3: Array and mesh topologies, modified from [14]



(a) 0-D, 1-D, 2-D, and 3-D hypercubes



(b) 4-D hypercube network

Figure 4: Hypercube networks of varying dimensions, modified from [14]

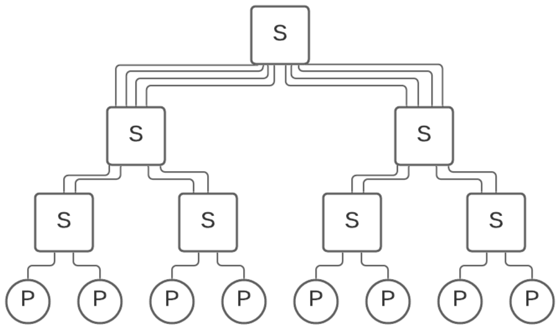


Figure 5: Non-blocking fat tree network, where p are individual nodes and s are switching nodes

currently are identified. In the aforementioned merge sort example, processes should be decomposed into blocks of processes in order to leverage the spatial locality of the data in memory such that each division of the data set is assigned to a block of concurrent processes. Another approach to the decomposition is the interleaving of processes for each iteration of a parallelizable task. Using the previous example of summing rows in a two-dimensional array, each row in the array can be assigned to a concurrent process. Unlike the merge sort algorithm, where data from one block of the divided data structure is required in other processes for the purpose of sorting, summing the contents of the values in one row of an array doesn't require interaction between other array elements, thus there would be no benefit to decomposing processes into blocks over interleaving them. Additionally, some algorithmic components cannot be decomposed. Figure 6 illustrates the difference between interleaving and block decomposition. The number of tasks in a problem that can be decomposed is referred to as the *degree* of concurrency and the size of each decomposed block is referred to as *granularity*.

It is also important when writing a parallel algorithm to determine how input, output, and intermediate data is managed throughout the life cycle of a program. For instance, components of an algorithm that are assigned to one process block

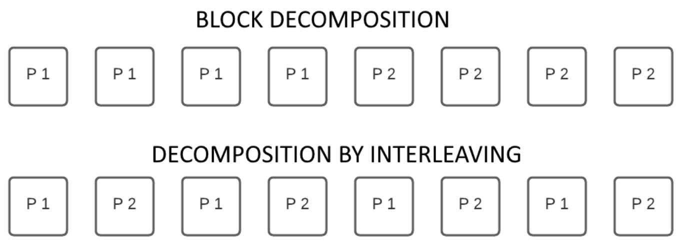


Figure 6: Block decomposition assigns one contiguous section of memory to a block of processes, while interleaving tasks assigns one concurrent task to the next iteration of a component of data or instructions

might require information that is calculated on another process block. This requires some level of communication between processes. Additionally the input or output data itself is sometimes decomposed among several processes.

Parallel algorithms can usually be classified as one of several algorithm models. According to the literature, the most common of these models are the data-parallel, task graph, work pool, manager-worker, and pipeline models, which are sometimes combined for use in hybrid models [14]. In the data-parallel model, processes are assigned tasks that operate mostly the same, but on different sets of partitions of data. In the task graph model, the dependency of some tasks on others is compiled into a graph called a *dependency graph* which is used to assign tasks in such a way that communications between tasks that are most dependent on one another occurs on the same or local processes. The work pool model assigns tasks dynamically to processes in such a way that when one process starts to become underutilized, more tasks will be assigned to it. In the manager-worker model, one process takes on the role as the manager while all other processes act as workers, operating on work that is assigned by the manager process. Lastly, the pipeline model, as the name suggests, consists of a pipeline of processes that each perform their own unique set of operations on data as it passes from process to process.

2.1.3 Message Passing Interface

For most high-performance computers, a C and Fortran library called MPI (message passing interface) exists to handle inter-process communications. This library includes over 125 functions that are responsible for everything from establishing communications channels and sending and receiving data between processes to determining process identification values. In a standard MPI program, setting up the interface and establishing communication between processes is simple. Listing II.1 demonstrates a simple MPI program where each process prints it's own process ID followed by the total number of processes.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    int iproc, nproc;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &iproc);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    printf("Process␣%d␣of␣%d\n", iproc, nproc);
    MPI_Finalize();
}
```

Listing II.1: Simple MPI program that prints each process ID

The MPI library contains a number of communication functions, as well. The **MPI_Send**, **MPI_Recv**, and **MPI_Sendrecv** functions are used to send and/or receive data from specific processes. The **MPI_Bcast** is used to perform an all-to-one broadcast operation, which sends the same data from one process to all other processes. Conversely, the **MPI_Reduce** function takes values from all other processes and sends them to the root process, which combines the values using any variety of logical, bit-wise, or arithmetic operators. When this reduced value is required by all processes, the **MPI_Allreduce** function is used instead. Much like **MPI_Bcast**, the **MPI_Scatter** function sends data from one process to all others, except that the original data is split up such that each receiving node will receive only a fraction of the data. **MPI_Gather** on the other hand works much like the reduce and all-reduce communications methods except that the data is concatenated when received by the root node as opposed to being reduced with some sort of operation. File I/O functions like **MPI_File_write**, **MPI_File_read**, and **MPI_File_set_view** are used to read and write to files, as well as restricting which section of a file each process is allowed to read from and write to [10]. The library also includes a number of MPI-specific data types, which are listed in table 1.

MPI Datatype	**Description**
MPI_CHAR	signed character
MPI_SHORT	signed short integer
MPI_INT	signed integer
MPI_LONG	signed long integer
MPI_UNSIGNED_CHAR	unsigned character
MPI_UNSIGNED_SHORT	unsigned short integer
MPI_UNSIGNED	unsigned integer
MPI_UNSIGNED_LONG	unsigned long integer
MPI_FLOAT	floating point
MPI_DOUBLE	double-precision floating point
MPI_LONG_DOUBLE	long double-precision floating point
MPI_BYTE	Byte
MPI_PACKED	Non-contiguous data

Table 1: MPI datatypes used in all MPI functions, list modified from [14]

2.2 Quantum Computing Stack

Quantum computers are different than their classical computer counterparts in a number of ways. Although the average person with little to no understanding of computer technology might think of a quantum computer as a faster, more efficient alternative to the personal computers we use regularly, there are few similarities in their function, architecture, and algorithm design. Quantum computing, in general, abides by an entirely different set of physical laws than classical computers. The benefits of quantum computation relies heavily on principles of quantum physics, such as superposition and entanglement, that are incapable of being leveraged by classical computers at all because of their deterministic nature. In order to understand the inner workings of a quantum computer, a basic knowledge of quantum mechanics is imperative.

2.2.1 Classical vs Quantum Computing

Quantum computers are quite unlike classical computers in a number of ways. While the computers that ordinary people are most familiar with operate in a deterministic way—that is, with a guaranteed certainty that a particular set of inputs across a circuit will result in one expected output—quantum computers behave non-deterministically, or probabilistically. When one queues a circuit to run on a quantum computer, there are a number of possible outcomes with varying possibilities. Although the most likely outcome of a single X-gate operation, for instance, would be a measurement of a qubit (quantum binary digit) in its excited state, quantum computers experience a preponderance of error from various sources and will not always produce the expected output. Anything from system noise to the act of measuring the qubit, itself, can result in the decoherence of the state of a qubit.

2.2.2 Qubit

A qubit is one of the most fundamental components of quantum information theory and is often visualized using the Bloch sphere, as shown in figure 7 where the basis states of the qubit are written as $|0\rangle$ or $|1\rangle$. While a classical computer uses regular binary digits, which can only have values of zero and one, the qubits in quantum computers can, under certain conditions, have a value that is a ratio of both $|0\rangle$ and $|1\rangle$, which is often visualized as a vector from the center to somewhere along the surface of the Bloch sphere. In a resting state, a qubit will always have a value of $|0\rangle$ when measured. Conversely, when the qubit is put into an excited state, it is highly probable that a subsequent measurement of the qubit will produce a value of $|1\rangle$. Due to the incredibly volatile nature of qubits, the state of the qubit is only capable of being held for a short period of time before the state of the qubit decoheres, or collapses, back into the $|0\rangle$ state. Decoherence times for even the most cutting edge quantum computers at the current time are somewhere on the order of milliseconds [18].

One unique quality of qubits is their ability to be put into a state of superposition of two basis states. This superposition is represented by the equation:

$$|\Psi\rangle = c_0\,|0\rangle + c_1\,|1\rangle$$

where c_0 and c_1 are some arbitrary values such that $|c_0|^2 + |c_1|^2 = 1$ [22]. $|c_0|^2$ is the probability that a measurement will result in a $|0\rangle$ value, while $|c_1|^2$ is the

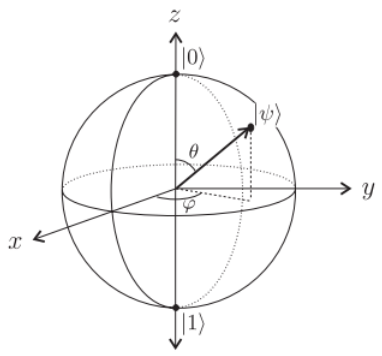


Figure 7: Bloch sphere representation of a qubit, taken from [17]

probability that a measurement will result in a $|1\rangle$. Changing the state of the qubit, including inducing quantum superposition or manipulating the phase, can be done by means of quantum gate operations.

2.2.3 Quantum Gates

Quantum gates can be applied to a qubit, or a number of qubits, by means of a quantum circuit notation. A typical quantum circuit consists of horizontal lines for each quantum and classical register being used in whatever algorithm or subroutine is being ran. A quantum register represents a single qubit while a classical register is used to read out values from the quantum registers as a binary value during a measurement operation. An example of a quantum circuit is shown in figure 8.

Quantum gate operations for each qubit occur concurrently from left to right. Unlike their classical counterparts, quantum gates must be reversible because of the unitary nature of all quantum operations, in general [27]. Single-qubit gates can be applied to one qubit at a time and consist of X-gates, which toggles the value of the qubit between $|0\rangle$ and $|1\rangle$, or Hadamard gates, which put the qubit into a superpositioned state, $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, to name a few. Other single-qubit gates include the Pauli gates (X, Y, and Z gates), phase gate (or P-gate), S-gate, T-gate, U-gate, and the identity gate (I-gate) [16].

Multiple-qubit gates, as the name suggests, operate on more than one qubit at a time. One of the most common of these is the controlled-NOT (CNOT) gate which is denoted by the blue gate in figure 8. When a CNOT gate is applied to two qubits, an X-gate operation is applied to the target qubit when the control qubit is $|1\rangle$, as shown in figure 8, and no change occurs when the control qubit is $|0\rangle$. If the control qubit is in a superpositioned state, $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, then the two qubits will become entangled, where the state of the qubits is $\frac{1}{\sqrt{2}}(|00\rangle+|11\rangle)$, which is known as a Bell state. Another

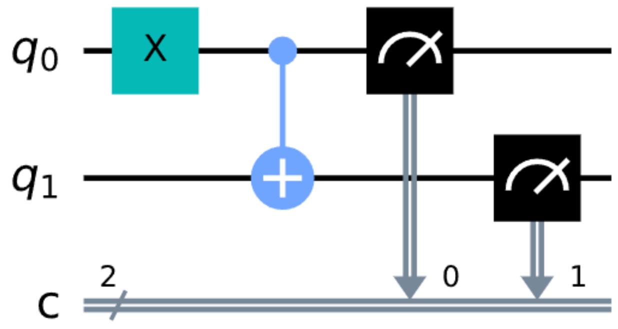


Figure 8: An example of the layout of quantum circuit that contains two qubits, two quantum gate operations, and two measurements, taken from [16]

multiple-qubit gate is the CCNOT-gate, otherwise known as the Toffoli gate. Much like the CNOT-gate, the CCNOT-gate functions the same way, but with two control qubits instead of one.

2.2.4 Quantum Algorithms

Section 2.2.3 introduces the idea of a quantum circuit, which when populated with various quantum gates is used to represent algorithms or critical subroutines. Some subroutines are absolutely essential for the execution of full-fledged quantum algorithms, like the quantum phase estimation (QPE) [16] in figure 9, which is a necessary component of Shor's prime factorization algorithm [26] and Grover's database search algorithm [15]. Another important subroutine includes the quantum Fourier transform (QFT), which can transform information back and forth between the computational basis, or the classically-interpreted binary values $|0\rangle$ and $|1\rangle$, and the Fourier basis, in which data is encoded along the equatorial axis of the Bloch sphere. The quantum Fourier transform is shown in figure 10.

All quantum algorithms have what is referred to as a time budget, which is the amount of time in which the qubits in a system will remain coherent. After the time

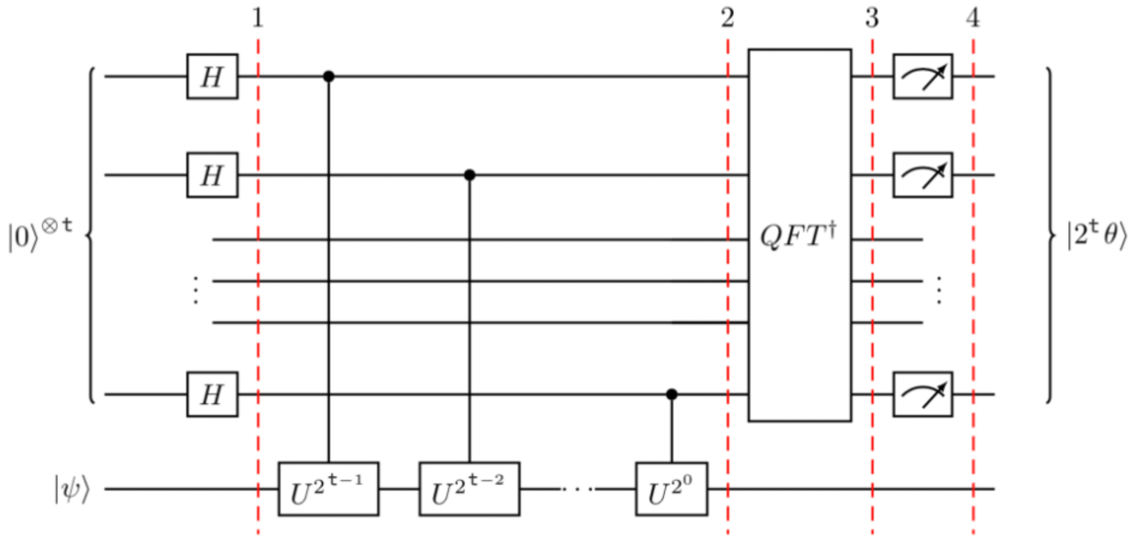


Figure 9: The circuit for the quantum phase estimation subroutine, taken from [16]

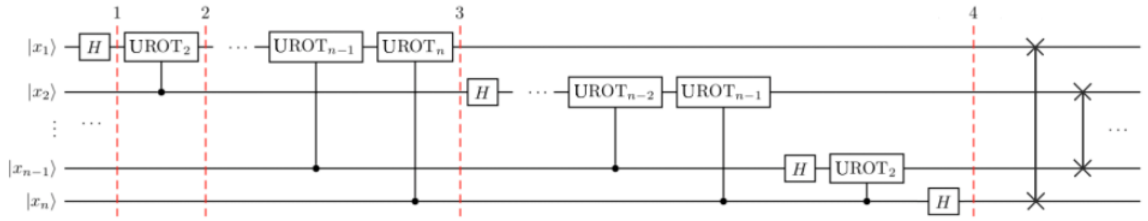


Figure 10: The circuit for the quantum Fourier transform subroutine, taken from [16]

budget has elapsed, any measurements taken as a part of the output of a quantum algorithm are not usable. The time budget of a quantum algorithm is largely dependent on the T_1 and T_2 times, or the amount of time until a qubit decoheres from an excited state and a superpositioned state, respectively, of the qubits in the architecture. The time budget for commercially available cloud-based quantum computers like IBM's Poughkeepsie is on the order of milliseconds and most quantum gates operate on the order of nanoseconds [18].

2.2.5 Quantum Error Correction

Though classical computers are also susceptible to error, it is exceptionally rare. Nonetheless, error correction protocols are still implemented in systems where errors are somewhat more frequent, like in communications. Error correcting code on classical computers, in the most simple case, usually consists of additional bits that reflect the value of the data bit. Further, in some cases, parity bits are also used as another layer of data surety. Quantum error correction follows a similar principle. In a quantum computer, there exist a number of interconnected qubits that can be used as either data qubits or ancilla qubits. Qubits that are used as a means of data encoding within a quantum circuit are known as data qubits. Alternatively, qubits that are used in support of another in some sort of critical subroutine or error correcting code are termed "ancilla" qubits. Together, these physical qubits form one logical qubit.

On a quantum computer, error correcting codes, or surface codes, are applied to a data qubit based on the values of surrounding ancilla qubits for errors involving bit-flips and phase-flips, or X and Z errors, respectively. X and Z errors are simply errors which can be shown to have propagated from operations involving either bit-flips or phase-flips. A surface code of depth three is shown in figure 11.

This paper assumes an architecture arranged in a lattice of $d \times d$ data qubits,

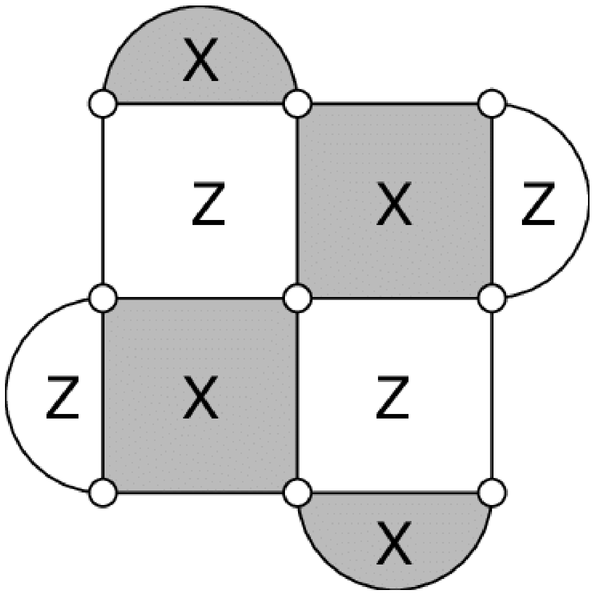


Figure 11: A depth three surface code where each white circle represents a data qubit, the white square and semicircular regions represent Z-ancilla qubits, and the grey square and semicircular regions represent X-ancilla qubits, taken from [12]

where d is the depth of the surface code, with ancilla qubits existing in the squares between these connected data qubits. Additionally, semi-circular ancilla qubits are located on the edges of the lattice. The proposed architecture in question provides an intuitive approach to quantum error correction. Each alternating ancilla qubit in this architecture is responsible for detecting X and Z errors. The idea of alternating the error type designation for these qubits provides each data qubit with a means of being detectable for both kinds of error. Additionally, each ancilla qubit is entangled with its four or two data qubits, depending on it's location on the circuit. By entangling these qubits, measurement of ancilla qubits can provide insight into the existence of error in a data qubit without collapsing said qubit's state. Multiple rounds of ancilla qubit measurement create a more effective means of parity checking. After this, a surface code can be applied to the quantum circuit in order to correct the error found in the error syndrome, which is determined by ancilla qubit measurements.

2.2.5.1 Decoders

Within the micro-architecture layer of the quantum computing stack, the quantum error decoder is responsible for determining the error syndrome, which essentially describes which errors exist and where they are located on a surface code. Decoding can be performed in a number of ways, from look-up tables to neural network assisted implementations. Furthermore, there exist matching algorithms such as Blossom's that are used to determine the correct response to the error syndrome of a surface code [30].

Look-up tables are perhaps the most inefficient means of decoding of all of the options available. They work by returning a unique output based on a particular input value. As the name suggests, this method works by storing responses to particular inputs in memory, like a table, and referencing the appropriate output when queried. Although this method might work well with negligible performance drawbacks for small tables, the efficiency deteriorates as the size of the table increases. Additionally, a table with a large number of entries will return the appropriate value after it references every single table entry, in the worst case.

Although many decoders scale poorly with an increase in the number of qubits in a system, neural network decoders work efficiently at constant time [30]. Although the time required to train neural networks on datasets scales with the number of samples in the set and the surface code depth, the neural network will operate at constant time once the model is exported.

2.3 Neural Networks

The field of artificial intelligence (AI) is the study of how machines are adapted to behave intelligently—that is, to react to various environment or input variables in order to achieve some desired outcome. From the algorithms responsible for generating computer player moves in a game of online chess, to the web application that translates text between languages in real-time, AI covers a broad range of problems in modern computer science. Machine learning, which is a more specific type of AI, takes one or more sets of data to "teach" a model to produce a certain output based on an input. At another level of abstraction, machine learning models that mimic the basic structure of the human brain are called neural networks. Neural Networks serve as the foundation of all deep learning applications. These networks consist of two or more layers of nodes called neurons that pass data along from input to output to produce some desired outcome. Additionally, the input and output data for a neural network is referred to as features and labels, respectively.

2.3.1 Neuron

The most basic component of a neural network is the neuron. Neurons, like the one shown in figure 12, resemble the characteristics and function of a biological neuron. A neuron is responsible for collecting inputs, summing them with their respective parameters, and outputting a value based on some activation function. The activation function for a neuron calculates an output value between zero and one. Two frequently-used activation functions, sigmoid and rectified linear unit, are shown in figure 13. With the sigmoid activation function, the neuron output can be any number between zero and one where the output value grows closer to one as the input value approaches infinity, and closer to zero as the input value approaches negative infinity.

2.3.2 Neural Network Structure

Every neural network has multiple layers of neurons with at least one input layer and one output layer. The input layer takes in data and has one neuron for each

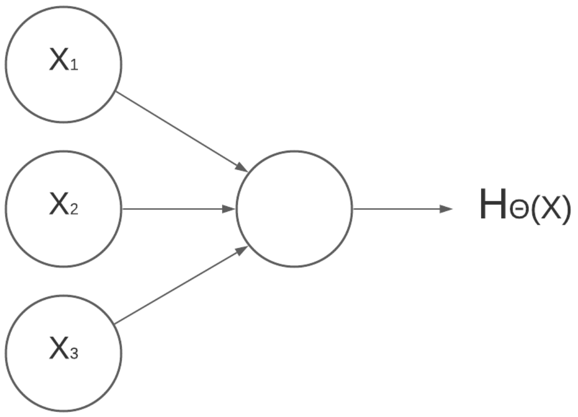
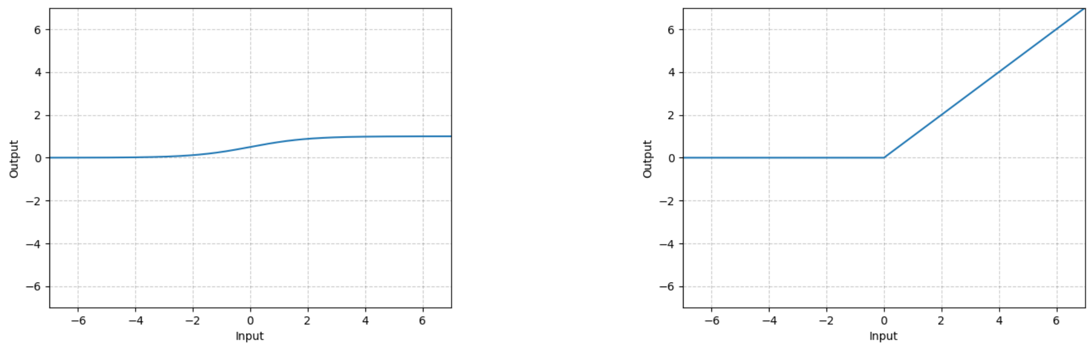


Figure 12: A neuron with three inputs, x_i, and output $H_\Theta(x)$



(a) Sigmoid

(b) Rectified linear unit (ReLU)

Figure 13: Neural network activation functions, plotted as input vs. output, modified from [3]

feature in the data set. Similarly, there is one neuron for each label in the output layer. Hidden layers can have any number of neurons per layer and will usually each have their own responsibility when solving an issue. For instance, a neural network used for facial recognition might use one hidden layer for edge detection and another for detecting facial features. The number of hidden layers and neurons per layer is different for every problem and requires experimentation in order to achieve the best results.

Each neuron in a neural network is connected to all other neurons in neighboring layers. Each of these connections also has a weight, or parameter associated with it that is used by the neuron to calculate the value that is passed into the activation function. These weights are represented as a matrix, Θ, which is used by the following equation in each neuron:

$$a_j = g\left(\left(\sum_{i=0}^{n} \Theta_i x_i\right) + b_j\right)$$

where x_i and Θ_i are the input and weight values for each connection from the previous layer, n is the number of inputs, b is the neuron bias, g is the activation function, and a is the output for neuron j in the current layer. Figure 14 shows a simple neural network with three inputs, one output, and one hidden layer containing three neurons. In a neural network like this, information is processed by means of vectorization, which consists of mathematical operations performed on vectors or matrices in a much more computationally efficient manner than by calculating the output value of each neuron individually. The expression denoting this operation for the hidden layer a_i in the neural network in figure 14, where i is the current layer, can be seen in the following equation:

$$a^i = g\left(\begin{bmatrix}\Theta_{1,1} & \Theta_{1,2} & \Theta_{1,3}\\ \Theta_{2,1} & \Theta_{2,2} & \Theta_{2,3}\\ \Theta_{3,1} & \Theta_{3,2} & \Theta_{3,3}\end{bmatrix}\begin{bmatrix}a_1^{i-1}\\ a_2^{i-1}\\ a_3^{i-1}\end{bmatrix} + \begin{bmatrix}b_1\\ b_2\\ b_3\end{bmatrix}\right)$$

where g is the activation function, Θ is the vector containing all weights for the connections between neurons in the previous layer and the current layer, and b is the vector of biases for each neuron in the current layer. Additionally, $\Theta_{j,k}$ denotes the weight from neuron a_k^{i-1} to neuron a_j^i where i is the current layer.

2.3.3 Neural Network Problems

Neural networks are often implemented a number of different ways in order to solve a broad diversity of problems. A short list of such problems include classification, regression, ranking, computer vision, and language processing. Table 2 shows these problems along with examples of each.

Classification problems deal with the categorization of input data into one or more classes. For each set of inputs, there is usually one correct output that should be produced by a classification network. In a multi-label classification problem, however, there are multiple outputs, each of which has a correct output that must be met for each set of given inputs. An example of a classification problem would be a neural network designed to classify which number is depicted in any image of a hand-written numeral. In this network, the inputs are broken down into an array containing pixel values of the image. Using these pixels, the neural network will detect patterns and edges in the image and make a guess on the number that is represented by these features.

Classification problems are evaluated on a number of metrics, including how often the correct output is produced over the total number of observations, or the accuracy;

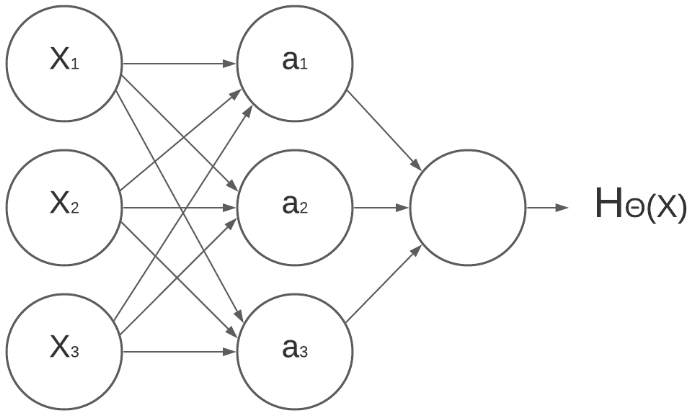


Figure 14: A neural network with inputs, x_i, a hidden layer with neurons, a_i, and output $H_\Theta(x)$

the precision, which shows how many times the output produced a true positive over the total number of times it predicted a positive outcome, positive and negative; and recall, which measures how well the neural network is able to correctly classify a sample that it has already seen.

Regression problems consist of predicting a single numerical value like a probability or a price. Like any traditional regression problem, these problems are statistical in nature and involve producing output values based on the relation between dependent and independent variables [4]. One such example would be a neural network that predicts the cost of a house based on relative population, crime rate, and geographic location. The network will train on a number of data sets that contain the independent variables (population, crime rate, and location) along with their respective dependent variables (housing costs) in order to create a network that will effectively predict the cost of a house given any values for inputs.

A neural network designed to handle a ranking problem will take a set of values and rank them according to the interest of whoever makes the query, usually based on some keyword or phrase [32]. One example of a neural network that implements a ranking problem would be a search engine that returns an aggregate of results based on a word or set of words. A good ranking neural network will provide a list of results with the most relevant items located at the top of the list.

Computer vision involves the use of digital imagery of footage to provide insight into some task. Computer vision is used today in most autonomous vehicle research and development. In fact, Tesla makes use of computer vision neural networks in the autopilot feature for their automobiles [1].

In language processing problems, a series of words or variables are analyzed within the context of each other. In the case of the auto-complete function native to most smartphones, the ability of the device keyboard to predict the probable next words in a dialogue is considered a language processing problem. Neural networks that solve these problems typically use recurrent neural networks that analyze sequences of text in the form of tokens that are converted from text to numeric values [33].

Problem	**Example**
Classification	Classifying written characters as alphanumeric values
Regression	Predicting housing costs
Ranking	Determining ranking of video search results
Computer vision	Facial recognition technology
Language Processing	Web applications for translating text between languages

Table 2: Short list of common neural network problems with examples, modified from [2]

2.3.4 Types of Neural Networks

There are a variety of neural network implementations that exist for different purposes, including artificial neural networks (ANN), which are often referred to as feed-forward neural networks (FFNN), convolutional neural networks (CNN), and recurrent neural networks (RNN). The most basic neural network is the feed-forward neural network, which processes all data in one direction from input to output layers. Feed-forward neural networks are capable of producing accurate results with incomplete data sets, but require a great amount of time to train. These models are often used for solving problems involving classification and regression.

Convolutional neural networks are similar to the feed-forward variety in that they contain several layers that process data in a unidirectional manner. As opposed to the feed-forward neural networks, convolutional networks make use of hidden layers that are implemented for performing specific, decomposed tasks related to the overall problem, called convolutional layers. These networks are also unique in that they often implement pooling layers that are responsible for compressing data from previous layers into a fewer number of neurons to produce lower training times for the overall network. Because convolutional neural networks are especially good at determining patterns for various components of an object or image, this model is often implemented in computer vision problems that seek to maximize the ability to correctly determine what an object or image is.

Recurrent neural networks, unlike feed-forward networks, are capable of processing data bi-directionally. In other words, data from the output of a layer can be fed backwards to a previous layer using this model. These neural networks can use this data from later layers to form early predictions for the output value of the network. Data that is fed back to previous layers is compounded in memory to create context for the overall problem, a unit of memory called long short-term memory (LSTM). Corrections to neuron output are calculated and assigned during backpropagation, which is when the error gradient re-evaluated across the network using a method called gradient descent.

2.4 Previous Work

In previous research, neural networks have been implemented in quantum error correction efforts for decoding purposes. The idea of using a neural network to handle the decoding process of a quantum circuit is appealing because of the clear benefits it provides in terms of execution time and performance, especially when compared to other methods of decoding like the Blossom decoding algorithm. Some implementations are even capable of using aspects of both neural networks and normal decoding algorithms, like in the case of [30], whose work suggests that such an implementation would also be effective. Unlike decoding algorithms such as Blossom, which scale either linearly or even in polynomial time in the case of the Minimum Weight Perfect Matching algorithm, neural networks are capable of executing in constant time. This is especially bad when applied to architectures of increasing qubit quantities. In order for quantum computers to become more scalable in terms of the number of qubits in a system, the means of applying error correction in a more effective way is absolutely necessary.

A means of error correction that can run fast and reliably is important in a quantum computer because of the unstable nature of qubits. The qubits within a quantum machine are constantly made subject to noise in the environment, which results in qubit decoherence over a long enough time scale. This time varies between platforms, such as superconductors and ion traps, and the type of decoherence from an excited state or from a state of quantum superposition, which result in T_1 and T_2 decoherence times, respectively. In order for any meaningful quantum computation to be performed, however, the time it takes for the computer to perform error correction cycles must be kept to a minimum. In other words, if an error correction cycle exceeds the time it takes for any of the qubits in a system to decohere, then the prospect of implementing error correction in that system to begin with is pointless. For a small number of qubits, like in an architecture similar to the rotated surface code of depth three, this should not be much of an issue. This becomes a problem, however, when the number of qubits in a system are increased, thus increasing the time budget for a cycle of error correction to run. This is important because, in order for quantum computers to have any sort of practical use, like for using Shor's Algorithm, we must be able to produce quantum computers with an exceptionally greater number of resources than are currently available in even the most advanced machines, like Google's Bristlecone or IBM's Eagle.

Several different implementations of neural network-based QEC decoders have been introduced over the past five years alone. One such implementation uses what is called a belief-propagation algorithm as the basis for the neural network [21]. The belief propagation algorithm is essentially a message-passing low-density parity-check (LDPC) decoder that is ordinarily unfit for quantum computing applications because of the degenerative nature of qubits. Researchers from the University of Sherbrooke and the Canadian Institute for Advanced Research, however, have coined the neural belief propagation (NBP) decoders to perform decoding on quantum LDPC codes in effort to provide an effective means of QEC with low execution time. The research proved that training a neural network with a belief-propagation algorithm can indeed provide a more fast and efficient means of decoding for quantum LDPC codes. Although for larger codes with a greater number of qubits still proves to be a challenge even for this particular decoder implementation, a significant increase in performance for smaller codes is noted on the toric, quantum bicycle, and hypergraph-product code.

Research conducted at Yale University has also shown success in using neural networks in QEC decoding applications [19]. Specifically, researchers developed a neural network decoder that at first seems more similar to a simple look-up table, but is in fact compressed by a few orders of magnitude. Since the ability of a normal decoder to deduce the stabilizer code of a circuit given its error syndrome is NP-hard, a reduction from time complexity $O(n^4)$ to $O(n^2L)$ is an enormous improvement. As opposed to the previous neural network decoder, however, this implementation opts out of a belief-propagation system altogether and instead implements a "hard-decision message passing algorithm." The neural network in this particular case learns about individual qubits and their error probabilities. This provides little in the way of correlating errors between a qubit and other qubits in the architecture. The authors do state, however, that a correlation can be made between multiple qubits, but that it would have to be implemented in the output layer of the neural network. Additionally, the authors boast that this particular implementation is the most effective means of decoding surface codes with architectures containing 200 or fewer qubits. In order for this method of decoding to remain relevant and effective, this quantity threshold must be improved.

In some cases, multiple error correction cycles might be necessary in neural networks, like in the presence of measurement error, the high- and low-level neural network decoders have been shown to perform considerably well under a variety of conditions and arrangements. Architectures also vary, though it seems that the rotated surface code and the toric code are the two most written-about in recent literature. In particular, the rotated surface code appears to be the most promising because of how scalable it can be when applied in a distributed manner.

Researchers at Delft University of Technology in the Netherlands have published multiple papers related to this particular topic. In 2020, they published a research paper comparing the Blossom decoding algorithm, which does not use neural networks, to the low- and high-level neural network decoders on various-sized rotated surface codes [31]. Like previous research, this paper realized the need for QEC in any form of quantum computing from data storage to critical subroutines in quantum algorithms. By providing a number of approaches to the integration of neural networks in the decoding of surface codes, some insight is given into the effectiveness of each and possible routes for future exploration or research. Though a number of implementations are detailed, they note that the best approach ought to have a balance between time and performance.

Another important factor in the research performed by Varsamopolous et al. is the training data set. For all of the various permutations of QEC arrangements, each of them either behaved according to the circuit noise model or the depolarizing error model. The process for collecting training data for the neural network requires the collection of the different error syndromes that might occur on a rotated surface code. The authors rightly point out that collecting every possible arrangement of errors across the code, to include Z and X errors, would prove to be an exercise in futility. The reason lies in the exponential scaling of errors as the code distance is increased. That is, the more qubits that are introduced into a surface code, the greater number of error combinations need to be recorded, making the gathering of training data physically impossible within a reasonable period of time. The approach the authors in this paper took to solving this issue relied on gathering incomplete, but highly generalized data sets that cover a wide diversity of error syndromes. Another approach to solving this issue in scalability is proposed by the authors and addressed in a later paper that implements a distributed decoding method [30]. Such a method would collect error syndrome training data for a small rotated surface code and apply it to various sectors of a larger surface code in a piece-wise fashion. Re-normalization Group decoding, as it is called in the paper, solves this problem since any error syndrome is able to be broken down into multiple depth three rotated surface codes.

From the research done at Delft University, it is apparent that high-level decoders trained with recurrent neural networks are the most efficient using both the circuit noise and depolarizing error models. Although low-level decoders actually perform the decoding by themselves, the high-level decoders simply use neural networks to assist in the decoding process. Thus, this research proves most useful in providing information for help in the configuration of future research in neural network decoding.

In previous research conducted at the Air Force Institute of Technology by Badger, low-level neural network decoders were created and tested against the minimum-weight perfect-match and partial lookup table decoders for quantum error correction [7]. Simple data sets containing the most frequent errors found on a rotated surface code were generated and used to train the neural network models for surface codes of depths three, five, and seven. These three decoders were tested on a toric circuit simulation in order to demonstrate their performance in correcting randomized error. This research serves as the starting point for the research in this paper and the three models used in Badger's work are used as the baseline models against which all other models created in this research are evaluated against.

2.5 Summary

This chapter covers all background information as it pertains to the subject matter in this research as well as a section covering related previous work in section 2.4. Section 2.1 describes high-performance computing and the design of parallel algorithms. Section 2.2 covers the basic principles of quantum computing and quantum error correction. Lastly, section 2.3 details the components of a neural network along with the various types of neural networks and how they are often used in practical applications.

III. Methodology

3.1 Overview

This chapter details the approach to solving the research question: How does the performance of neural network-based decoders for the surface code with various data set generation techniques compare to traditional algorithmic error correction decoders? Section 3.2 covers the design decisions for the construction of the various algorithmic simulations that are used to generate data sets of varying scopes for the neural network decoders used in later sections. These neural network decoders are adapted from those used in previous research and are trained on the aforementioned data sets so that they can be compared to the previously-implemented baseline models [7]. Section 3.3 describes in greater detail the parallel algorithms that are responsible for the generation of the data sets that are used by the neural network decoders. This section also explains how the static surface code simulation formats the data sets so that they can be used for training and testing the neural network models. Section 3.4 describes how previous neural network research is adapted for this research and provides insight into how the neural networks are trained, validated, and tested using the newly-acquired data sets from section 3.3. A dynamic surface code simulation, which measures the performance of each neural network decoder over multiple error correction cycles, is used to evaluate the neural network models created in this research against the baseline models created in previous work [7].

3.2 Design

The contents of this section describe the design considerations involving the simulation that will be used to generate data sets and the neural network that will be used as a decoder for quantum error correction.

3.2.1 Parallel Surface Code Simulation

The surface code simulation is a parallel algorithm designed to run on a high-performance computer for the purpose of generating training data sets for the neural network decoder. Parallel algorithms require the careful consideration of various factors including the algorithmic model, message passing methods, task mapping, and decomposition. The development of an algorithm is also dependent on many hardware factors as well, such as the computation model and architecture. This subsection contains details on which design options were chosen in the implementation of the surface code simulation. The high-performance computers used in this work include the Navy DoD Supercomputing Resource Center's (DSRC) Gaffney, the Air Force Research Laboratory's (AFRL) Mustang, and the commercial Sabalcore Red cluster.

HPC	**Compute Nodes**	**Peak PFLOPS**	**Parallel disk capacity**
Mustang	752	3.29	5.5 PB
Gaffney	1176	4.87	8.4 PB
Sabalcore	480	N/A	1 PB

Table 3: Specifications for each of the high-performance computers used in this research

Both Mustang and Gaffney contain two 24-core Intel Xeon Platinum 8168 processors per compute node, 192 GB memory for standard nodes, 768 GB memory for large-memory nodes, 384 GB memory for GPU accelerated nodes, and a parallel file system with RAID arrays. Mustang and Gaffney are both HPE SGI 8600 systems that use Intel Omni-path interconnect high-speed networks for message passing and file I/O operations. The Sabalcore Red cluster contains two 8-core Intel Xeon E5-2667v3 processors per compute node with 256 GB memory per node and 16 GB memory per core. The architecture of all systems are non-blocking fat trees. In addition, memory is shared on each respective node, but distributed across the entire tree. Common specifications for each of the three computers are listed in table 3. According to Sabalcore, peak floating point operations per second (FLOPS) are not published because it is subjective when compared to real-world applications.

3.2.2 Dataset Generation Method

The intended purpose of the HPC surface code simulation is to generate data sets for use in the deep neural network decoder. With that in mind, we must consider the scope of the data sets that will be used in the training, testing, and validation of the error correction model. Although the data sets can be produced in a number of ways, this research focuses mainly on collecting exhaustive and randomly-sampled data sets.

3.2.2.1 Exhaustive

In a surface code of depth three, there are only nine data qubits. For each data qubit, it is possible for there to be an X error, a Z error, both kinds of error, or no error whatsoever. Thus, the total number of combinations of error across an entire surface code comes out to 4^n where n is the total number of data qubits. For a surface code of depth three, this means that there are 262,144 possible error combinations that can occur. While this figure might seem feasible at this depth to iterate over within an algorithm, the number of combinations quickly exceeds the realm of possibility as the depth of the surface code scales up. Using the same formula, the number of unique error combinations for surface codes of depth five and seven come out to approximately 1.126×10^{15} and 3.169×10^{29}, respectively. Even if one were capable of storing the data for a single sample in a byte-long data structure, an exhaustive data set for a surface code of depth five would consume 1,024 Terabytes. For depth seven, that figure becomes nearly 2.815×10^{14} Petabytes. This figure would realistically be much larger since each sample is likely to require multiple bytes of space, depending on the surface code depth. It would appear that an exhaustive approach to data generation would only be feasible for a surface code of depth three. Even if such data sets cannot be accomplished for larger surface codes, the cross-validation results for the depth three surface code should provide insight into the effect of an exhaustive data set on the performance of the neural network decoder relative to other decoding methods, in general.

3.2.2.2 Random Sampling

The next-best alternative to generating exhaustive data sets for our error correction model is to randomly sample from the pool of possible combinations of error. By providing the algorithm with a predefined number of samples to collect, the size of the output for surface codes of depths greater than three will be more manageable. Additionally, since the probability distributions are uniform for combinations with the same number of X and Z errors, then the prospect of maintaining a data structure that contains all possible probabilities for use in random sampling becomes achievable.

In order to implement random sampling, the number of unique probabilities for the current surface code depth needs to be calculated and stored along with the respective number of both kinds of error and the number of times that each probability can occur. The latter of these entries, which turns out to be the binomial coefficients for each number of error, can be easily calculated by implementing Pascal's Triangle within a two-dimensional array. The elements of the probability list will then be sorted by ascending probability so that the cumulative probabilities can be calculated.

With the list of cumulative probabilities generated, the function used to generate randomly-sampling data will operate as follows. A random number between 0 and 1 is generated and the list of cumulative probabilities is iterated over until a cumulative probability greater than the random number is found. The respective quantity of X and Z errors for this probability is used in a Fisher-Yates Shuffle to determine which numbered data qubit on the surface code will contain error.

3.2.3 Algorithm Structure

The version-specific design considerations for various versions of the algorithm are outlined in the following section. The exhaustive version outputs the exhaustive data set outlined in Section 3.2.2.1 and the random-sampling version outputs the randomized data sets discussed in Section 3.2.2.2.

3.2.3.1 Exhaustive Version

Since the purpose of this algorithm is to generate training data that emulates the rotated surface code at various code depths, the algorithm will be expected to iterate over all data qubits once for each of the four potential error states of the respective qubit, but only for the generation of exhaustive data on the depth three code. This will involve 4^n iterations where n is equal to the number of data qubits. The number of data qubits in a rotated surface code is equal to the code depth squared.

Within each iteration of this first loop, two nested loops are implemented. The first loop will iterate over the data qubits in the surface code and record the X and Z error values for each respective qubit along with the probability that the errors will be arranged this particular way. The second loop will use these data qubit X and Z error values to determine the ancilla qubit values for each of the ancilla qubits. Finally, the algorithm will write the values for the ancilla qubits and the respective data qubit errors to a file. Algorithm 1 demonstrates the pseudo-code for the implementation of this surface code simulator. The time complexity of Algorithm 1 is $O(n4^n)$.

Algorithm 1 Parallel Surface Code Simulation (Exhaustive)

Inputs: *depth*, *p_x_error*, *p_z_error*

```
1: $num\_data\_qubits \leftarrow depth^2$
2: $data\_qubit\_x\_error \leftarrow [depth+2][depth+2]$
3: $data\_qubit\_z\_error \leftarrow [depth+2][depth+2]$
4: $ancilla\_qubit\_value \leftarrow [depth+2][depth+2]$
5: Initialize MPI with vars: $iproc$ and $nproc$
6: $total\_num\_iter \leftarrow 4^{num\_data\_qubits}$
7: $block\_size \leftarrow floor(total\_num\_iter/nproc)$
8: Add 1 to $block\_size$ if it doesn't divide evenly
9: $iter\_first \leftarrow iproc \times block\_size$
10: $iter\_last \leftarrow iter\_first + block\_size$
11: for $i$ in range($iter\_first, iter\_last$) do                    ▷ Outer for-loop
12:     $errors \leftarrow i$
13:     $probability \leftarrow 1.0$
14:     for $j$ in range($num\_data\_qubits$) do                  ▷ First inner for-loop
15:         $this\_error \leftarrow errors\%2$
16:         $data\_qubit\_x\_error[j/depth][j\%depth] \leftarrow this\_error$
17:         Multiply $probability$ by $p\_x\_error$ if $this\_error$ is 1. Otherwise, multiply
    by $(1.0 - p\_x\_error)$
18:         $errors \leftarrow errors/2$
19:         $this\_error \leftarrow errors\%2$
20:         $data\_qubit\_z\_error[j/depth][j\%depth] \leftarrow this\_error$
21:         Multiply $probability$ by $p\_z\_error$ if $this\_error$ is 1. Otherwise, multiply
    by $(1.0 - p\_z\_error)$
22:         $errors \leftarrow errors/2$
23:     end for
24:     for $j$ in range($(depth+1)^2 - 1$) do                    ▷ Second inner for-loop
25:         $ancilla\_qubit\_value[j/depth][j\%depth] \leftarrow data\_qubit\_x\_error[j/depth][j\%depth] \oplus$
    $data\_qubit\_x\_error[j/depth][(j\%depth)+1] \oplus$
    $data\_qubit\_x\_error[(j/depth)+1][j\%depth] \oplus$
    $data\_qubit\_x\_error[(j/depth)+1][(j\%depth)+1]$
26:         $j \leftarrow j+1$
27:         $ancilla\_qubit\_value[j/depth][j\%depth] \leftarrow data\_qubit\_x\_error[j/depth][j\%depth] \oplus$
    $data\_qubit\_x\_error[j/depth][(j\%depth)+1] \oplus$
    $data\_qubit\_x\_error[(j/depth)+1][j\%depth] \oplus$
    $data\_qubit\_x\_error[(j/depth)+1][(j\%depth)+1]$
28:     end for
29:     Write values for $data\_qubit\_x\_error$, $data\_qubit\_z\_error$, $ancilla\_qubit\_value$,
    and $probability$ to CSV file
30: end for
```

Since there are no data dependencies between each outer loop iteration of the algorithm, these iterations can be run independent of one another with no errors or race conditions. Both inner **for** loops on the other hand are not capable of leveraging parallelism because of the dependencies that occur when the second inner loop calculates ancilla qubit values from the data qubit error values from the previous inner loop. Additionally, the data produced in both of these **for** loops is needed in the last series of nested loops in order to write them to a file. Since the same computation is performed on different sets of data in each outer loop iteration, then the algorithm model can be described as a data-parallel model. The same computation is performed on separate data sets, reducing message-passing overhead. The only message-passing occurs when the data across various nodes needs to be written out to a file.

The algorithm uses block decomposition as a means of distributing the various tasks to processes. As opposed to interleaving tasks, which alternates which process each consecutive iteration is assigned to, a block decomposition method would be able to better leverage spatial locality, where subsequent memory accesses within one block of tasks reference neighboring locations in memory. The algorithm uses a dynamic generation model in which the block sizes are calculated at runtime. Block size can be denoted by $4^n/p$ where n is the number of data qubits and p is the number of processes that the algorithm is allocated. The resulting time complexity becomes $O(n4^n/p)$ for each process.

In a scenario where the number of outer loop iterations is equal to or less than the number of processes available, then each iteration of the outer loop can run on its own process, potentially reducing the time complexity of the to $O(n)$, best-case. Compared to the serial implementation of this simulation, as shown in Algorithm 2, the parallel implementation has the potential to run at a reduced time complexity. Both versions of the algorithm are similar except for the MPI routines and variables that are used in the parallel implementation. In addition, the outer **for** loop of the serial implementation iterates over the entire span of $4^{num\_data\_qubits}$ while the parallel implementation only iterates over the decomposed block size.

A scenario where the number of iterations in the outer **for** loop of the simulation and the number of available processes are equal is impossible because of the hardware limitations of the high-performance computers used in this research, even at a code depth of three. The total number of iterations for a depth three surface code is $4^{depth^2} = 4^{3^2} = 262,144$. According to the documentation for AFRL's Mustang, which contains the greater number of processing nodes of the three machines used in this work, the maximum number of cores that can be used for a standard job is 28,224, or just over a tenth of the number of cores needed in order to provide each outer loop iteration with it's own core.

3.2.3.2 Random Sampling Version

The random sampling version of the algorithm is executed one of two ways: either with or without replacement of selected samples. While a method involving replacement would allow for duplicate sample entries in the data set, sampling without replacement would ensure that the data set has only unique entries. Though the two versions of the random-sampling algorithm are similar, producing data sets with no duplicate samples requires additional logic that will affect the run time of the algorithm.

Unlike the exhaustive version of the algorithm which iterates over a total of $4^{num\_data\_qubits}$ outer loop iterations, the random sampling version will only iterate over the desired number of samples in the output data set. Depending on the number of samples, it would be possible under this regime to assign each outer loop iteration to it's own process on any of the high-performance computers used in this research.

Algorithm 2 Serial Surface Code Simulation (Exhaustive)

Inputs: $depth$, p_x_error, p_z_error

```
1: $num\_data\_qubits \leftarrow depth^2$
2: $data\_qubit\_x\_error \leftarrow [depth + 2][depth + 2]$
3: $data\_qubit\_z\_error \leftarrow [depth + 2][depth + 2]$
4: $ancilla\_qubit\_value \leftarrow [depth + 2][depth + 2]$
5: for $i$ in range($4^{num\_data\_qubits}$) do    ▷ Outer for-loop
6:     $errors \leftarrow i$
7:     $probability \leftarrow 1.0$
8:     for $j$ in range($num\_data\_qubits$) do    ▷ First inner for-loop
9:         $this\_error \leftarrow errors\%2$
10:        $data\_qubit\_x\_error[j/depth][j\%depth] \leftarrow this\_error$
11:        Multiply $probability$ by $p\_x\_error$ if $this\_error$ is 1. Otherwise, multiply
           by $(1.0 - p\_x\_error)$
12:        $errors \leftarrow errors/2$
13:        $this\_error \leftarrow errors\%2$
14:        $data\_qubit\_z\_error[j/depth][j\%depth] \leftarrow this\_error$
15:        Multiply $probability$ by $p\_z\_error$ if $this\_error$ is 1. Otherwise, multiply
           by $(1.0 - p\_z\_error)$
16:        $errors \leftarrow errors/2$
17:    end for
18:    for $j$ in range($(depth + 1)^2 - 1$) do    ▷ Second inner for-loop
19:        $ancilla\_qubit\_value[j/depth][j\%depth] \leftarrow data\_qubit\_x\_error[j/depth][j\%depth] \oplus$
           $data\_qubit\_x\_error[j/depth][(j\%depth) + 1] \oplus$
           $data\_qubit\_x\_error[(j/depth) + 1][j\%depth] \oplus$
           $data\_qubit\_x\_error[(j/depth) + 1][(j\%depth) + 1]$
20:        $j \leftarrow j + 1$
21:        $ancilla\_qubit\_value[j/depth][j\%depth] \leftarrow data\_qubit\_x\_error[j/depth][j\%depth] \oplus$
           $data\_qubit\_x\_error[j/depth][(j\%depth) + 1] \oplus$
           $data\_qubit\_x\_error[(j/depth) + 1][j\%depth] \oplus$
           $data\_qubit\_x\_error[(j/depth) + 1][(j\%depth) + 1]$
22:    end for
23:    Write values for $data\_qubit\_x\_error$, $data\_qubit\_z\_error$, $ancilla\_qubit\_value$,
       and $probability$ to CSV file
24: end for
```

Algorithm 3 demonstrates the initialization phase of the random sampling version of the surface code simulation. In this phase, MPI is initialized and Pascal's Triangle is implemented on each process. A data structure containing the probabilities for each combination of error, along with the number of times a sample of this probability occurs, is generated. The list is then sorted by ascending probabilities using quicksort so that the list of probabilities can be iterated over to calculate the cumulative probability at each index. An empty list is then allocated in memory of size *num_samples* for the purpose of storing samples that have already been collected. This last line is used exclusively in the version of the algorithm without replacement to check for duplicate samples.

After the initialization phase, the pseudo-code in Algorithm 4 performs the parallel phase of the algorithm. The outer loop will iterate over the block size that was decomposed in the previous phase. At the beginning of the loop, the arrays containing qubit values are set to zero and a random number between 0 and 1 is generated. The list of cumulative probabilities is iterated over until a value greater than the random number is found. This index will provide the rest of the algorithm with the number of X and Z error that need to be placed somewhere on the surface code.

With the number of data qubit errors determined, a Fisher-Yates Shuffle is performed in order to resolve the location of each error on the surface code. The corresponding location in the data qubit error arrays are written to with a value of 1, indicating an error present. After this, the ancilla qubit values are determined in the same way they were calculated in Algorithm 1 and the values are subsequently written out to a file.

While the version of the random sampling algorithm that involves replacement of already-written error syndrome configurations is similar to the version without replacement of these values, the latter requires a great deal more logic to ensure

Algorithm 3 Parallel Surface Code Simulation (Random-Sampling, Initialization Phase)

Inputs: $depth$, $num\_samples$, p_x_error, p_z_error

```
1: $num\_data\_qubits \leftarrow depth^2$
2: $data\_qubit\_x\_error \leftarrow [depth + 2][depth + 2]$
3: $data\_qubit\_z\_error \leftarrow [depth + 2][depth + 2]$
4: $ancilla\_qubit\_value \leftarrow [depth + 2][depth + 2]$
5: Initialize MPI with vars: $iproc$ and $nproc$
6: $block\_size \leftarrow floor(total\_num\_iter/nproc)$    ▷ Add 1 to $block\_size$ if it doesn't divide evenly
7: $iter\_first \leftarrow iproc \times block\_size$
8: $iter\_last \leftarrow iter\_first + block\_size$
9: Initialize Pascal's Triangle as 50x50 array
10: for $i$ in range($num\_data\_qubits + 1$) do    ▷ Initialize prob_values struct
11:     for $j$ in range($num\_data\_qubits + 1$) do
12:         Set $n\_x\_error$ to $i$ and $n\_z\_error$ to j
13:         $Prob \leftarrow (1 - p\_x\_err)^{n-i} \times (1 - p\_z\_err)^{n-j} \times p\_x\_err^i \times p\_z\_err^j$
14:         $Count \leftarrow pascal\_triangle[n][i] \times pascal\_triangle[n][j]$
15:     end for
16: end for
17: Perform Quicksort on prob_values struct
18: for $i$ in range($num\_data\_qubits + 1$) do    ▷ Increment counter each inner loop
19:     for $j$ in range($num\_data\_qubits + 1$) do
20:         $Cumulative \leftarrow Cumulative + prob[counter] \times count[counter]$
21:         $Cum\_prob[counter] \leftarrow cumulative$
22:     end for
23: end for
24: $combinations \leftarrow [num\_samples]$    ▷ Error Syndrome struct
```

Algorithm 4 Parallel Surface Code Simulation (Random-Sampled, Parallel Phase)

```
1: for $i$ in range($iter\_first$ to $iter\_last$) do    ▷ Outer for-loop
2:     Clear all values in arrays $data\_qubit\_x\_error$, $data\_qubit\_z\_error$, and
   $ancilla\_qubit\_value$
3:     $rand\_num \leftarrow$ random number between 0 and 1
4:     for $j$ in range($(num\_data\_qubits + 1)^2$) do
5:         Iterate over $cum\_prob$ until value at index $j$ is greater than $rand\_num$
6:         $sample\_idx \leftarrow j$
7:     end for
8:     $rand\_array \leftarrow [num\_data\_qubits]$    ▷ Values are equal to indices
9:     Perform Fisher-Yates Shuffle over $n\_x\_error$ iterations
10:    $qubit\_idx \leftarrow rand\_array[0]$
11:    $data\_qubit\_x\_error[qubit\_idx/depth + 1][qubit\_idx\%depth + 1] \leftarrow 1$
12:    Repeat previous three lines for $data\_qubit\_z\_error$
13:    Repeat lines 9-12 if $combinations$ already contains this combination of errors
14:    for $j$ in range($(depth + 1)^2 - 1$) do    ▷ Second inner for-loop
15:        $ancilla\_qubit\_value[j/depth][j\%depth] \leftarrow data\_qubit\_x\_error[j/depth][j\%depth] \oplus$
   $data\_qubit\_x\_error[j/depth][(j\%depth) + 1] \oplus$
   $data\_qubit\_x\_error[(j/depth) + 1][j\%depth] \oplus$
   $data\_qubit\_x\_error[(j/depth) + 1][(j\%depth) + 1]$
16:        $j \leftarrow j + 1$
17:        $ancilla\_qubit\_value[j/depth][j\%depth] \leftarrow data\_qubit\_x\_error[j/depth][j\%depth] \oplus$
   $data\_qubit\_x\_error[j/depth][(j\%depth) + 1] \oplus$
   $data\_qubit\_x\_error[(j/depth) + 1][j\%depth] \oplus$
   $data\_qubit\_x\_error[(j/depth) + 1][(j\%depth) + 1]$
18:    end for
19:    Write values for $data\_qubit\_x\_error$, $data\_qubit\_z\_error$, $ancilla\_qubit\_value$,
   and $probability$ to CSV file
20: end for
```

the same sample is not collected twice. In particular, lines 3-6 of Algorithm 4 are iterated over until a cumulative probability pool is found where the number of samples already collected from this pool does not exceed the number of possible arrangements as determined by the binomial coefficients calculated in line 14 of the initialization phase of Algorithm 3. Similarly, lines 9-12 are repeated, as indicated by line 13, if the arrangement of X and Z errors already exists in the list of combinations that have already been written out to a file.

3.2.4 Error Correction Model

The error correction model used in this research is based on the implementation used in previous work by Badger [7]. A neural network is trained using the data collected from the algorithms described in section 3.2.1 and section 3.2.3. The models created from these neural networks are tested using a simulated toric circuit that operates by generating a random error, placing it somewhere on a surface code, and passing the error syndrome created by taking ancilla qubit measurements into the neural network model. The neural network then returns corrective actions to the toric circuit to implement. This process is iterated over 1,000 times and each test implements multiple rounds of error correction in order to eliminate error corrective operations that incorrectly or insufficiently manipulate the data qubits. Since this design is meant to test the accuracy of the neural network trained in section 3.2.5, no changes to this design are necessary for this research.

3.2.5 Neural Network Model

The deep neural network used in this research is assigned the task of receiving an error syndrome and determining which data qubits on a surface code contain errors based on these ancilla qubit values. For this task, examples are fed into the neural network in the form of error syndrome data, or ancilla qubit values, and labels containing the appropriate response to the given arrangement of ancilla qubits in the syndrome. Each feature x_i in an example $x \in R^n$ is an ancilla qubit value of either 1 or 0. The value only appears as 0 if an error is present on that particular qubit. Additionally, the ancilla qubits are arranged in the examples by each semicircular or square region of the surface code from from left to right and top to bottom, where the X and Z ancilla qubits alternate on each row. The labels for each example are formatted as lists containing strings of names for all errors present within the error syndrome. The format follows the same convention used in previous work done by Badger [7].

Since the purpose of the neural network is to determine which data qubit, or which set of data qubits, must be corrected, then the task is a supervised multilabel classifier-type. The neural network is considered supervised because the training and testing data also contains the labels which are meant to be predicted by the network, itself. The performance measures most applicable to a classification problem such as this include accuracy, precision, recall, F1 score, and the area under the curve (AUC) of the receiver operating characteristic (ROC) curve. Additionally the effective capacity must be optimized such that both the training and test data sets behave on the neural network with high accuracy.

Other factors are also considered in the design of the neural network. When we look at the task of our neural network, we consider the input layer, or the binary representation of ancilla qubit values; the hidden layers; and the output layer, which consists of labels containing the data qubits that require corrective operations. If this output layer is capable of producing every possible output for a given surface code, then the number of output nodes will equate to the number of every possible combination of error for the surface code depth in which the neural network is designed for. In addition, the number of hidden layers is determined by the capacity of the neural network. In training the network, a greater or fewer number of hidden layers are introduced to the model to increase or decrease the capacity with respect to the generalization and training error. Additional considerations include the activation function and density for each layer in the model, the loss function, the optimizer, the learning rate, and the number of epochs. Previous work has shown that these values are best determined using SKLearn's GridSearchCV function.

For this research, we take the deep learning model developed by Badger and train the model using more complete sets of data produced by the high-performance computing (HPC) algorithm from Section 3.2.1 and section 3.3[7]. With new datasets for surface code depths of three, five, and seven, the neural network is reevaluated and trained to achieve greater performance. Neural network hyperparameters are then adjusted in order to maximize performance according to the aforementioned performance metrics.

3.3 Generating Data from Parallel Surface Code Simulation

This section covers the specific design methodology related to the parallel surface code simulator and the data that it produces. The simulator is examined and compared to the serial implementation of the algorithm. The data-parsing process is also explored in detail.

3.3.1 Surface Code Simulation Algorithm

This section contains the details regarding the operation of the several versions of the parallel surface code simulation.

3.3.1.1 Exhaustive Version

Algorithm 1 in Section 3.2.3 describes the structure of the algorithm used to generate the data used in the neural network decoder. The algorithm is designed to accept three command line arguments: the surface code depth, X error probability, and Z error probability. The output consists of a single CSV file where each row contains the ancilla qubit values associated with an error syndrome along with the data qubits where the respective errors are present.

At the start of algorithm execution, the start time is recorded for later use in determining the execution time at the end of the algorithm. MPI is then initialized with *nproc* processes where each individual process is denoted by *iproc*. After all variables, including the block decomposition variables, are instantiated, the file is created or opened with the MPI_File_open method. Following this, the $MPI\_File\_set\_view$ method is called to set the view of the file byte stream such that each block of processes is only capable of writing to it's own respective offset in the file.

The CSV file headers are then created on the root process and written out using the logic outlined in algorithm 5. The headers and the subsequent samples are written out to the file using the MPI_File_write method, which automatically writes to the offset position in the file based on the current process once $MPI\_File\_set\_view$ is called.

After MPI has been initialized, everything in the simulation will execute simultaneously on a number of processes to speed up the generation of this training data. Following this initialization, the algorithm determines which process block the current process will belong to. The outer loop on line 11 of algorithm 1 then iterates from the first process in the block to the last process. Logic that excludes processes outside of this range will automatically exit the loop if the process ID is not within this range.

Algorithm 5 Excerpt: Formatting CSV Headers

```
$ancilla \leftarrow 0$
for $i$ in range($1 \longrightarrow depth$) do
    if $i$ is even then                                  ▷ Check for upper edge Z ancilla
        Write "$Z + ancilla$" to file
        $ancilla{+}{+}$
    end if
    for $j$ in range($depth - 1 \longrightarrow 0$) do
        if $i == 1$ and $j$ is even then                 ▷ Check for left edge X ancilla
            Write "$X + ancilla$" to file
            $ancilla{+}{+}$
        else
            if $i == depth - 1$ and $j$ is odd then      ▷ Check for right edge X ancilla
                Write "$X + ancilla$" to file
                $ancilla{+}{+}$
            end if
        end if
        if $i + j$ is even then                          ▷ Check for middle X ancilla
            Write "$X + ancilla$" to file
            $ancilla{+}{+}$
        else                                             ▷ Check for middle Z ancilla
            Write "$Z + ancilla$" to file
            $ancilla{+}{+}$
        end if
    end for
    if $i$ is odd then                                   ▷ Check for lower edge Z ancilla
        Write "$Z + ancilla$" to file
        $ancilla{+}{+}$
    end if
end for
```

Inside this outer loop, *probability* is initialized to a floating-point value of 1 and the value for $errors$ is set to the value of the current process ID such that the bit string representation of $errors$ denotes the locations of X and Z errors for each alternating bit. The 2-dimensional arrays containing the data and ancilla qubit values are then initialized to zero using the **memset** function, which removes the need to add nested for-loops which would only increase the overall time complexity of the algorithm.

The first nested loop iterates through all of the data qubits in the circuit and is responsible for setting the values of each data qubit with respect to the presence of error. The *data_qubit_x_error* variable, for example, is essentially a matrix representation of the surface code where each index represents the coordinates of a data qubit on the surface code and the value represents the state of the qubit: '1' if there is an error present and '0' for no error. This is determined by checking the least significant bit of $errors$, which is stored in the variable *this_error* prior to shifting the value of $errors$ one place to the right before the next error is read out. Additionally, the probability value is calculated based on the number of errors present within a circuit. Ternary operators are used to multiply the probability by the *p_x_error* and *p_z_error* values whenever an error at the current coordinate is present, and by $1 - p\_x\_error$ and $1 - p\_z\_error$ when no error is present. The probability of a given circuit arrangement occurring is given by the equation $P = (p\_x\_error * p\_z\_error)^m * ((1 - p\_x\_error) * (1 - p\_z\_error))^{n-m}$ where n is the number of data qubits and m is the number of errors. This entire process is outlined in lines 11-23 algorithm 1.

The second nested loop on lines 24-28 of algorithm 1 assigns the ancilla qubit value by taking the bitwise XOR of each of the ancilla qubits' four associated data qubits. This serves as a simple emulation of the CNOT gate arrangement that sets the value of an ancilla qubit with respect to its four associated data qubits in a quantum circuit. If an odd number of errors are present in the four adjacent data qubits, then the value of the ancilla qubit will reflect that with a 1. Otherwise, the value will be set to 0. This loop executes over $(depth + 1)^2 - 1$ iterations.

Finally, three sets of doubly-nested loops are implemented to write the contents of the data qubit X and Z error arrays and the ancilla qubit value arrays to a CSV file. Following this, the root process calculates and prints the execution time, the file is closed with MPI_File_close, and MPI is completed by calling $MPI\_Finalize$.

3.3.1.2 Random Sampling Version With Replacement

Although the randomly-sampled versions of the surface code simulation are similar to the exhaustive version in many ways, they have their fair share of differences. The random sampling versions contain two unique data structures called *s_prob*, which are used in calculating and sorting all possible probabilities along with the number of samples per probability and the respective number of X and Z errors, and $error\_syndrome$, which are used by the version of the algorithm without replacement to store combinations of error that have already been collected.

Three additional functions are used for the quicksort and Fisher-Yates algorithms that are implemented later on in the code. algorithm 6 serves as the helper function passed into the quicksort algorithm to sort the list of *s_prob* structs by ascending probability later on in the simulation. algorithm 7 function is used for randomizing a given array of any size. algorithm 8 is the helper function used in the Fisher-Yates Shuffle to swap two values in an array.

At the start of the main function in the algorithm, everything is initialized much the same way as the exhaustive version, except that the block size is calculated based on the number of samples, *num_samples*, instead of the previous total number of outer loop iterations, or $4^{num\_data\_qubits}$.

Algorithm 6 Quicksort comparator function, q_comp

Inputs: $*p1,\ *p2$

```
1: $a \leftarrow *p1.probability$
2: $b \leftarrow *p2.probability$
3: if $a < b$ then
4:     Return $-1$
5: end if
6: if $a > b$ then
7:     Return $1$
8: end if
9: Return $0$
```

Algorithm 7 Fisher-Yates Shuffle function

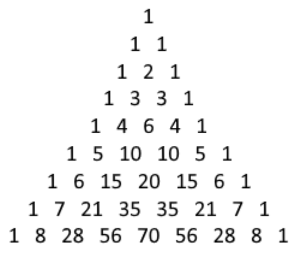
Inputs: $array[],\ size$

```
1: for $i$ in range($size - 1 \longrightarrow 0$) do
2:     $rand\_num \leftarrow rand()\%(i + 1)$
3:     $Swap(\&array[i], \&array[rand\_num])$
4: end for
```

Algorithm 8 Swap helper function

Inputs: $*first,\ *second$

```
1: $buffer \leftarrow *first$
2: $*first \leftarrow *second$
3: $*second \leftarrow buffer$
```



```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
    1   6  15  20  15   6   1
  1   7  21  35  35  21   7   1
1   8  28  56  70  56  28   8   1
```

Figure 15: Pascal's Triangle of binomial coefficients where each number in the figure is the sum of the two numbers diagonally above

After the initialization, Pascal's Triangle is generated in a 2-dimensional array based on Figure 15 where each tile is the sum of the two tiles above it in the triangle. This is used in calculating binomial coefficients in a later code segment. An array of size $(num\_data\_qubits + 1)^2$ is then initialized and cleared with *memset*. Nested for-loops iterate over $num\_data\_qubits + 1$ iterations each and assign the number of X and Z errors based on the current iteration of each loop. For each combination or error, the probability is calculated with eq. (1), where X and Z are equal to the number of X and Z error, and the number of samples that can be generated per probability by using the previously-instantiated Pascal's Triangle array. The number of unique samples per probability is described in eq. (2) where n, x, and z are equal to the number of data qubits, number of X errors, and number of Z errors, respectively. Once these values are assigned, the struct is written to *s_prob*.

$$P = (1-p\_x\_error)^{num\_data\_qubits-x} * (1-p\_z\_error)^{num\_data\_qubits-z} * p\_x\_error^{x} * p\_z\_error^{z} \tag{1}$$

$$Count = \binom{n}{x}\binom{n}{z} = \frac{n!}{x!(n-x)!}\frac{n!}{z!(n-z)!} \tag{2}$$

Once *s_prob* has been populated, it is sorted by ascending probability using quicksort. This allows the cumulative probabilities for each entry in *s_prob* to be calculated and written back to each entry in the struct.

Once inside the main loop, which iterates over the set of processes within each block, the values for the arrays containing the ancilla qubits and the data qubit X and Z errors are set to zero with *memset*. A random number between 0 and 1 is then generated. In a for-loop that iterates over $(num\_data\_qubits + 1)^2$ values in the *s_prob* struct, the values for the cumulative probabilies are compared to the random number until a probability is found that is larger than the random number. When this happens, the index at which the probability was found is set as the *sample_index* and will be used to access the number of X and Z errors for this probability in the next loop.

An array of length *num_data_qubits* is initialized where each element in the array is equal to its index. *num_x_error* and *num_z_error* iterations of the Fisher-Yates Shuffle are performed on the array, where the selected value will end up at index zero of the array and written to *qubit_index*. This value determines the location of the data qubit on which the error will be placed. If the position at $data\_qubit\_x\_error[(qubit\_index/depth)+1][(qubit\_index\%depth)+1]$ is already equal to 1, then the process restarts beginning at the Fisher-Yates Shuffle, otherwise the value will be set to zero. This process repeats for the Z errors. After this, the rest of the algorithm follows the same structure as the exhaustive version.

3.3.1.3 Random Sampling Version Without Replacement

Much of the code for the version of this algorithm that operates without replacement of samples is the same as the previous version. There are, however, a few slight differences. First, an array of type *error_syndrome* and of size *num_samples* called *combinations* is used to store the bit string representations of the error syndromes that have already been written out to a file. This requires an additional write operation with MPI after each sample is written to a file so that other process blocks will not write the same sample as another.

The portion of the algorithm responsible for determining the *sample_index* value is also made more thorough in order to ensure that the number of samples for a given probability have not already been depleted prior to proceeding. If the number of samples collected has, in fact, been depleted, then the algorithm breaks from the for-loop and resumes back at the line in which the random number is generated.

Similarly, the process involving the Fisher-Yates Shuffle in which the data qubit errors are written the data qubit error arrays is contained within a while-loop that will only exit if the chosen permutation of X and Z error has not already been written to *combinations* in the form of two bit strings of type *uint64_t* that are part of the struct *error_syndromes*. If a match for the current configuration is found, then the error syndrome bit strings are appended to *combinations*, MPI updates *combinations* on all process blocks, and the loop exits. The rest of the algorithm executes the same as the previous version.

3.3.2 Parsing Simulation Output

The data collected from the parallel surface code simulation must be parsed and arranged in such a manner that it can be used to train and test the deep neural network. Previous low-level neural network decoder work using a neural network similar to ours arranges data in the following manner. Headers consist of all of the ancilla qubits in a given depth of surface code followed by the labels. The values for each of the ancilla qubits are 1 for a corresponding data qubit without error or -1 when the current round of error correction detects an error. The labels contain the location of the data qubit in which the error is present along with the type of error present. For instance, an entry showing a surface code of depth three with an X error on data qubit 0 is depicted in Table 4.

X0	Z1	X2	Z3	Z4	X5	X6	Z7	Labels
1	1	1	1	1	1	1	1	“[]”
-1	1	1	1	1	1	1	-1	“[‘X00’,‘Z22’]”
-1	1	1	1	1	1	1	1	“[‘X00’]”

Table 4: Example CSV formatting for sample data for surface code of depth three.

3.4 Neural Network

This section covers the reconfiguration, training, and testing of the neural network model that was adapted from Badger's model [7]. The research for the previous model outlined three goals: to demonstrate the operation and effectiveness of a neural network decoder for the surface code, compare the neural network decoder to the MWPM and PLUT decoders, and implement the neural network decoder on a simulated toric circuit to demonstrate the performance of the decoder in the presence of measurement error.

The goal of this research remains the same, but with the added goal of creating a more effective model in terms of accuracy and F1 by means of implementing and testing various and more comprehensive data sets. Because much of the machine learning pipeline remains unchanged from the previous pipeline, this section will focus mainly on the changes that were made.

3.4.1 Training and Validation

The primary change in this iteration of the neural network decoder is the implementation of various data sets. While the previous data sets were relatively small, as noted in Table 5, the data set sizes that result from the random sampling algorithms are variable in size and can be generated relatively fast, even while using the serial implementations of these algorithms on machines less capable than the HPCs used in this research.

One of the obstacles from the previous research was the fact that the data set for the depth three surface code was too small and thus was unable to be used in any sort of meaningful training and testing regime. Figures 16 and 17 show the accuracy and loss plots and the AUROC curve for the depth three neural network model using the data set from previous work. The goal in implementing larger data sets, especially

Data Set	Depth 3	Depth 5	Depth 7
Baseline	28	2776	508180
Exhaustive	262144	∅	∅
Rand (w/ Repl)	Variable	Variable	Variable
Rand (w/o Repl)	Variable	Variable	Variable

Table 5: Data set size by depth and algorithm version vs. data set size from previous work.

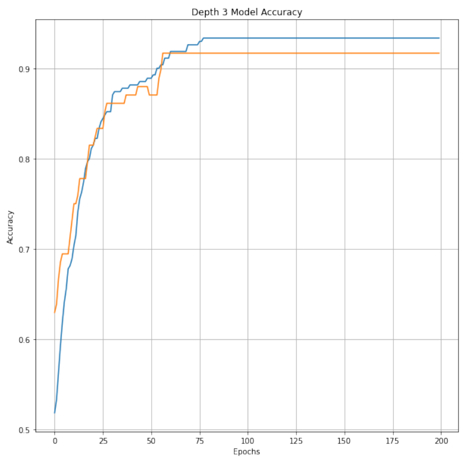
for depth three, is to provide the neural network models with the data necessary to perform better than the other decoding methods previously mentioned.

The training hyper-parameters for the neural network are also re-evaluated and adjusted in order to maximize accuracy and F1 for each model. Like in previous research, the *GridSearchCV* function from the SKLearn library is used to determine the appropriate number of hidden layers, hidden layer nodes, learning rate, activation function, loss function, and optimizer for each surface code depth. Similarly, the neural network is evaluated at various threshold values to determine the best threshold at which to round the output activation function's value. The selected values are shown in Table 6. This process is repeated for the various data sets outlined in section 3.2.2 in order to determine how these models perform against one another.

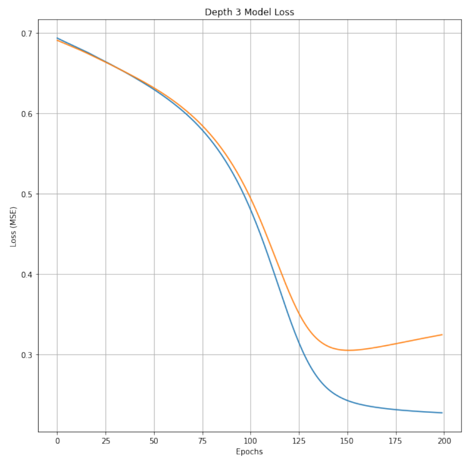
Once the parameters are determined, the neural network model is trained and validated, which provides insight into the overall accuracy and loss. The model is then exported for use in the aforementioned toric circuit simulation to test effectiveness

	Depth 3	Depth 5	Depth 7
Hidden Layers	4	4	4
Hidden Layer Nodes	32	250	400
Learning Rate	5%	5%	5%
Activation Function	ReLU	ReLU	ReLU
Loss Function	Log Loss	Log Loss	Log Loss
Optimizer	SGD	SGD	SGD

Table 6: Parameters used for each neural network model. Adapted from [7]



(a) *Accuracy*



(b) *Loss*

Figure 16: Training and validation accuracy and loss for surface code of depth three

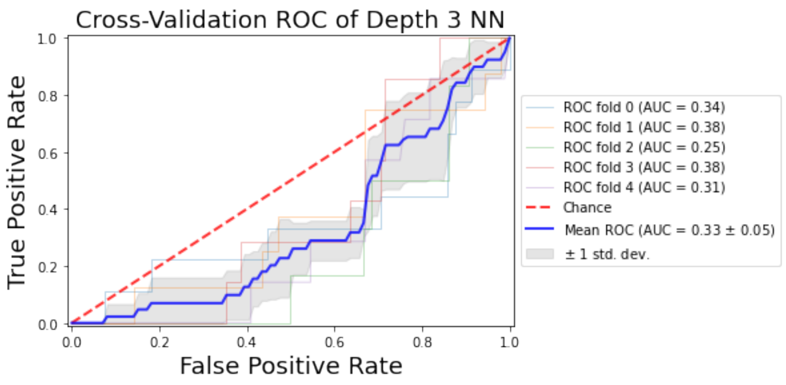


Figure 17: Training and validation accuracy for surface code of depth three

in a simulated noisy environment. In section 3.4.2, the network is further evaluated against the MWPM and PLUT decoders.

3.4.2 Decoder Performance Evaluation

In order to measure the performance of the neural network decoder, it must be evaluated against the MWPM and PLUT decoders. Do do this, a means of evaluating and comparing performance metrics is implemented in the form of K-fold cross validation. The input data is split into training and testing sets where the training set has the greater number of samples. The training set is used in training and validating the model while the test set is used later on in evaluating the decoders. All three decoders accept the same input samples.

The MWPM algorithm used in this research is borrowed from IBM's QISQIT library and remains unchanged from its implementation in the previous iteration of this training model. Since the algorithm performs all necessary calculations at run time to produce an output, this decoder does not require training of any kind. The PLUT decoder, on the other hand, simply instantiates a look-up table with the aforementioned training data and produces a best-guess response to a given input on testing.

Cross validation is performed in K-folds to reduce the amount of variance in performance that we would see from the results of the training and validation alone. The decoders are evaluated under the same parameter settings listed in Table 6 and the ROC curve is generated using the average values for each label. Performance metrics used to indicate performance of the decoders are the same as those used in the previous implementation: accuracy, F1 score, and execution time, along with X^2 for use in later statistical analysis using McNemar's Test. In addition to these previously-implemented metrics, the confusion matrix is produced on the test set for the neural network decoders to demonstrate the true vs. predicted values that the decoder produced, along with the average F1 score and accuracy of the decoder. The ROC curve is also generated after cross validation to demonstrate the precision and recall metrics in the form of an F1 score.

After determining the parameter values that produce the most accurate decoder, the neural network model is saved and written to a file for use in a toric circuit simulation so that it can be evaluated in the presence of measurement error. Neural network decoder model performance is then evaluated against other models in terms of error correction cycles and is noted in chapter IV accompanied by respective parameter values and data set attributes.

3.4.3 Testing

The neural network decoder models are tested using the toric circuit simulator written by Badger [7]. At the start of the simulation, a surface code of the current depth is instantiated of class $ToricCircuit$, which contains lists of ancilla and data qubit classes. This structure is used in the simulation to keep track of data and ancilla qubit values and includes a means of introducing any type of error into the system, checking for error, manipulating error values, and reading ancilla values as an error syndrome. Once the neural network model is loaded, the simulation iterates over 1000 tests where each test consists of the following:

- Random error introduced to data qubit at random position
- The error syndrome of the circuit is retrieved and the ancilla values are formatted for use in the neural network
- The physical error values are printed for reference

- The model is used to predict the corrective operation to be performed based on the error syndrome
- The threshold value determined in training is used to determine whether a correction should occur at a particular predicted value
- The inverse transform of the predictions produce the corrective operations used by the simulation
- The corrections are applied to the data qubits and incorrect operations are reported and printed
- The process repeats if the number of errors on the circuit is greater than the depth of the circuit
- Errors are cleared from the circuit and the test repeats after appending the number of error correction cycles

After testing the neural network on the simulation, the distribution of error correction cycles is plotted by cycles vs. frequency. As noted in previous work, each test will continuously perform error corrective operations until the number of errors on the circuit exceeds the depth of the surface code. Any number of errors greater than this will produce an error in the overall logical qubit.

The Kruskal-Willis H-test is then performed between each neural network decoder model and the baseline model using the aforementioned error correction cycle values. This test, which is a non-parametric version of the analysis of variance (ANOVA) test on normally-distributed data, is used to test the statistical significance between non-parametric data of two or more groups. This will prove whether or not there is a difference in performance between any of the decoders created for this research and the baseline models from previous work [7].

3.5 Summary

This chapter described the methodology in which the tests and experiments for this research were conducted. In section 3.2, we summarized all of the design decisions that were considered in the implementation of both the HPC simulation and the neural network. Section section 3.3 described the operation and use of the Message-Passing Interface (MPI) in the HPC algorithm and how the sample data for the neural network decoder was obtained. Lastly, the process by which the previous implementation of the neural network decoder was adapted, trained, validated, and tested was outlined in detail in section 3.4.

IV. Results and Analysis

4.1 Overview

This chapter presents the results and analyses of the research outlined in chapter III. In section 4.2, the performance and complexity of the high-performance computing algorithms are analyzed. The serial and parallel implementations of the algorithms, along with the various data sets they produce, are investigated. The performance and results of experiments involving the neural network decoder are examined in section 4.3. This section evaluates the training and testing of the neural network decoder as well as the implementation of the decoder in the surface code simulation.

4.2 High-Performance Computing Algorithms

This section covers the in-depth analysis of how well each of the high-performance computing algorithms performs in terms of time complexity, space complexity, and run time.

4.2.1 Serial Algorithm Performance

Each of the algorithms described in chapter III have a serial and a parallel implementation. All of the algorithms used in this research are capable of being executed serially on an ordinary desktop computer. Although the parallel implementations are necessary in order to produce large data sets for more hardware-intensive algorithms, such as the version that implements random sampling without replacement, most of the data collected in this research is obtainable by means of the serial versions. The metrics used in this research for the analysis of these serial algorithms include time complexity, space complexity, and run time.

4.2.1.1 Asymptotic Analysis and Time Complexity

Perhaps the most straightforward version of the surface code simulation algorithm is the exhaustive version. In algorithm 2, there exists a primary outer **for** loop with two primary inner for-loops that are necessary for performing calculations. These begin at lines 5, 8, and 18 in the pseudocode, respectively. In addition to these, there three sets of nested **for** loops that occur prior to the start of the outer **for** loop and after the execution of the second inner **for** loop that are responsible for formatting and writing to the CSV file that contains the sample values.

A simplified version of this algorithm is depicted in algorithm 9 which depicts all of the loops, functions, and other statements within the algorithm that have a time complexity greater than constant time. These values are used in determining the overall time complexity of the algorithm.

In algorithm 9, the sum of all consecutive time complexities of the various components of the algorithm can be described in terms of surface code depth, d, as:

$$T(d) = O(d^2 + d^2 + 4^{d^2} \times (d^2 + d^2 + d^2 + d \times d + d \times d)) = O(2d^2 + 4^{d^2} \times 5d^2)$$

As a general rule, when calculating the time complexity of an algorithm, the complexity of the term with the greatest order will always outweigh those of lower order when summed. As a result, the $2d^2$ can be removed from the equation since the main outer **for** loop, 4^{d^2}, has an exponential time complexity. Additionally, constant values, like the one in $5d^2$ can be removed. The resulting time complexity of the algorithm then becomes:

$$T(d) = O(4^{d^2} d^2)$$

When written in terms of the number of data qubits in the surface code, the time

Algorithm 9 Simplified Serial Surface Code Simulation (Exhaustive)

Inputs: $depth$, p_x_error, p_z_error

```
1: CSV Headers are written to file                                   ▷ $O(d^2)$
2: Data and ancilla qubit arrays are initialized                     ▷ $O(d^2)$
3: for $i$ in range($4^{num\_data\_qubits}$) do                      ▷ $O(4^{d^2})$
4:     Data and ancilla qubit arrays are set to 0                    ▷ $O(d^2)$
5:     for $j$ in range($num\_data\_qubits$) do                       ▷ $O(d^2)$
6:         Data qubit errors and probabilities are calculated
7:     end for
8:     for $j$ in range($(depth + 1) \times (depth + 1)$) do          ▷ $O(d^2)$
9:         Ancilla qubit values are calculated
10:    end for
11:    for $j$ in range($depth$) do                                   ▷ $O(d)$
12:        for $k$ in range($depth$) do                               ▷ $O(d)$
13:            Ancilla qubit values are written to CSV
14:        end for
15:    end for
16:    for $j$ in range($depth$) do                                   ▷ $O(d)$
17:        for $k$ in range($depth$) do                               ▷ $O(d)$
18:            Data qubit labels are written to CSV
19:        end for
20:    end for
21: end for
```

complexity is:

$$T(n) = O(n4^n)$$

All of the loops contained in this algorithm will execute over every iteration, thus there is no distinction between the best- and worst-case time complexity for this particular algorithm. Best, worst, and average time complexities for all serial algorithm versions are noted in table table 7.

Both versions of the algorithm that involve random sampling are much more complicated than the exhaustive version for two reasons. For one, the additional algorithmic components responsible for generating random samples and keeping track of which samples already exist in a set add a great deal of complexity to the problem. Additionally, some aspects of these algorithms—particularly in the version that produces data sets with no duplicate samples—are non-deterministic because of the use of random number generation in the selection of data qubit errors and sample indices. For that reason, it is assumed in algorithm 10 that any **while** loop that is implemented for the purpose of eliminating duplicate indices or samples will run at $O(1)$ time. Although this is the best-case time complexity for all such while-loops, the average case is non-deterministic.

The version of the algorithm that generates random samples with replacement is simplified in algorithm 10 to only include components of the algorithm that have a time complexity greater than constant time. Using the same method of summing complexities of individual algorithmic components as the previous algorithm, the time complexity before simplification is:

$$T(d+m) = O(2d^2+4d^4+d^2\times log\ d^2+d^4 log\ d^4+m\times(3d^2+log\ d^4+d^2\times d^2+2\times(d\times d)))$$

where d is the surface code depth and m is the desired number of samples. This equation, once simplified, comes out to:

$$T(d+m) = O(d^4 log\ d^4 + mlog\ d^4)$$

The quicksort algorithm on line 11 has an average time complexity of $O(nlogn)$, but can be $O(n^2)$ in the worst-case scenario. Similarly, the random index selection on line 15 will not always reach all d^4 iterations and may even exit on the first iteration in the best-case scenario, making the average case $log\ d^4$. Lastly, when the number of X and Z errors in the current sample are being iterated over, it is possible for there to be as few as one and as many as d^2 errors for each loop to iterate over. The best, worst, and average time complexities for the overall algorithm are listed in table table 7.

Much like the previous algorithm, the version that produces random samples without replacement follows much of the same flow of logic. The only difference, however, is that there must include data structures that keep track of samples that have already been collected in order to prevent duplicate values from populating the data set. Logic is therefore introduced to verify that the samples being generated are not in the set of already collected samples. These factors not only add to the time budget of the algorithm, but also to the spatial complexity, which is explored more in section 4.2.1.3. This algorithm is outlined in algorithm 10.

This implementation introduces a new data structure that contains the bit-string representations of X and Z error values. In the algorithm, we call the list of these structs *combinations*, and it's initialization has a time complexity of $O(m)$ since it will only contain sample data for the number of samples created, which must be less than or equal to the total number of samples, m. **While** loops are also implemented when randomly selecting the sample index that points to the number of X and Z errors in the current sample, as well as when assigning errors to data qubits in order

Algorithm 10 Simplified Serial Surface Code Simulation (Randomly Sampled with Replacement)

Inputs: $depth$, $num\_samples$, p_x_error, p_z_error

```
 1: CSV Headers are written to file                                      ▷ $O(d^2)$
 2: Data and ancilla qubit arrays are initialized                        ▷ $O(d^2)$
 3: Pascal's Triangle is initialized                                     ▷ $O(d^4)$
 4: for $i$ in range($depth^2 + 1$) do                                   ▷ $O(d^2)$
 5:     for $j$ in range($i$) do                                         ▷ $O(log\ d^2)$
 6:         Calculate Pascal's Triangle values
 7:     end for
 8: end for
 9: $prob\_values$ struct is initialized                                 ▷ $O(d^4)$
10: Calculate $prob\_values$ values                                      ▷ $O(d^4)$
11: Quicksort $prob\_values$ by ascending probability                   ▷ $O(d^4 log\ d^4)$
12: Calculate cumulative probabilities for $prob\_values$               ▷ $O(d^4)$
13: for $i$ in range($num\_samples$) do                                 ▷ $O(m)$
14:     Data and ancilla qubit arrays are set to 0                       ▷ $O(d^2)$
15:     Find random sample index                                         ▷ $O(log\ d^4)$
16:     Generate array of indices                                        ▷ $O(d^2)$
17:     for Number of x and z errors errors do                           ▷ $O(d^2)$
18:         Perform $depth^2$ iterations of Fisher-Yates Shuffle on random array ▷
    $O(d^2)$
19:     end for
20:     for $j$ in range(($depth + 1$) × ($depth + 1$)) do               ▷ $O(d^2)$
21:         Ancilla qubit values are calculated
22:     end for
23:     for $j$ in range($depth$) do                                     ▷ $O(d)$
24:         for $k$ in range($depth$) do                                 ▷ $O(d)$
25:             Ancilla qubit values are written to CSV
26:         end for
27:     end for
28:     for $j$ in range($depth$) do                                     ▷ $O(d)$
29:         for $k$ in range($depth$) do                                 ▷ $O(d)$
30:             Data qubit labels are written to CSV
31:         end for
32:     end for
33: end for
```

to prevent duplicates. While these loops don't add to the overall complexity of the algorithm in the best-case, the worst-case scenario would involve searching the entire list of values in *combinations* over *num_samples* iterations. The time complexity of this algorithm version is thus:

$$T(d+m) = O(d^4 log\ d^4 + m(log\ d^2 + m))$$

In both of the random sampling algorithms, the component with the highest order time complexity is the quicksort operation that takes place on line 11 of algorithm 10, with a time complexity of $O(n^2)$ in the best-case, $O(n^4)$ in the worst-case, and $O(n^2 log\ n^2)$ in the average-case scenario. Other bottlenecks include the Fisher-Yates shuffles on line 18 that occur for each number of data qubit errors in the current sample, which drives the time complexity to an added $O(n^2)$, $O(n\ log\ n)$, and $O(n)$ in the worst, average, and best-cases, respectively; and the logic that checks the current sample against the samples in *combinations*, which only adds an additonal $O(log\ m)$ in the average case and $O(m)$ in the worst case.

Although the time complexities for both random sampling algorithms are different only in terms of the number of samples, m, the version with replacement will most often outperform the version without replacement, especially for larger data sets. This is simply because of the aforementioned logic that is implemented in the version without replacement that will require an indeterminant number of cycles for the

Data Set	Worst Case	Average Case	Best Case
Exhaustive	$O(n4^n)$	$O(n4^n)$	$O(n4^n)$
Rand (w/ Repl)	$O(n^4 + mn^2)$	$O(n^2 log\ n^2 + mn\ log\ n)$	$O(n^2 + mn)$
Rand (w/o Repl)	$O(n^4 + m(n^2 + m))$	$O(n^2 log\ n^2 + m(n\ log\ n + log\ m))$	$O(n^2 + mn)$

Table 7: Best, worst, and average time complexities for all three serial surface code algorithms.

operations on lines 15 and 17 through 19 of algorithm 10. The **while** loops used in these sections will only exit once a permutation of errors is found that is not already contained in the list of samples that have already been written out to a file, thus the number of **while** loop iterations for these sections will inevitably increase as the pool of possible samples is depleted.

4.2.1.2 Run Time

Serial run time for each algorithm is measured between the very start of the main function and right before the main function's return statement. The average run time for each algorithm with each set of constraints is listed in table 8.

By taking the averages of the differences in time between versions of the algorithms with similar depths and sample sizes, it can be determined that the run time for the algorithm that generated random samples without replacement will tend to run longer than the version that produces duplicate samples. This is most likely due to the additional logic required to store previously-written samples and check against when writing new ones to a file, as noted in section 4.2.1.1. Average run time differences for both algorithms on depths three and five are shown in table 9. The values in this table suggest that the difference in run time between both versions of the random sampling algorithm scales with the number of samples. This is true for the exhaustive version, as well. Although this algorithm was not successfully run on surface codes of depths five and seven due to hardware constraints, attempts were made to execute the algorithm at both depths, which lasted over several hours in both instances before hard disk space was depleted and the program was forced to exit.

Algorithm	Depth	Samples	Run Time (sec)
Exhaustive	3	262,144	1.542
Rand (w/ Repl)	3	1,000	0.004
Rand (w/ Repl)	3	10,000	0.040
Rand (w/ Repl)	3	100,000	0.377
Rand (w/ Repl)	5	1,000	0.012
Rand (w/ Repl)	5	10,000	0.109
Rand (w/ Repl)	5	100,000	1.082
Rand (w/ Repl)	7	1,000	0.027
Rand (w/ Repl)	7	10,000	0.240
Rand (w/ Repl)	7	100,000	2.367
Rand (w/ Repl)	7	500,000	11.827
Rand (w/o Repl)	3	1,000	0.013
Rand (w/o Repl)	3	10,000	0.954
Rand (w/o Repl)	5	1,000	0.014
Rand (w/o Repl)	5	10,000	0.289
Rand (w/o Repl)	7	1,000	0.027

Table 8: Average run times of all algorithms for all depths and sample sizes.

Depth	1,000 Samples	10,000 Samples
3	0.009	0.914
5	0.002	0.180
7	0.0	N/A

Table 9: Average run time differences between both random sampling algorithms.

4.2.1.3 Hardware and Spatial Analysis

In addition to time complexity, each algorithm has a space complexity that describes the memory space required in order for an algorithm to run successfully, with respect to input values. Much like the time complexity, this value simply denotes the highest order in which a variable or data structure occupies memory with respect to input. The input values, much like with the time complexity, are d or n for the depth or number of data qubits where $n = d^2$, and m for the number of samples in the random sampling algorithms.

For the exhaustive algorithm, the number of samples produced is a function of the surface code depth. As such, the space complexity will only require one input variable, d or n. The variables that require the greatest order of memory space are the two-dimensional arrays that contain the data and ancilla qubit values. The space complexity for these values is $O(d^2) = O(n)$. Since the space complexity also accounts for the size of data written to disk, the complexity must also consider the 4^{d^2} or 4^n samples that are written out by the time the program exits. This value supercedes the previous $O(n)$ value, so the new space complexity of the algorithm becomes $O(4^{d^2})$ or $O(4^n)$.

Both of the random sampling algorithms are largely the same in terms of variables and data structures. Although they both contain the two-dimensional arrays of size d^2, these are by no means the highest order structure in these programs. The array of data structures that contains all of the probability values for every possible combination of error has a space complexity of $O(d^4)$ or $O(n^2)$. The algorithm that implements random sampling without replacement, however, implements another array of data structures that contains the combinations of error that have already been written out to a file. The space complexity of this array has the same space complexity of the files that both algorithms write to, $O(m)$, where m is the number of samples. The only other significant source of memory usage from these algorithms is the use of quicksort, which has a space complexity of $O(log\ d^4)$ or $O(log\ n^2)$[25]. As the surface code depth increases, however, this value is negligible compared to higher-order structures like the array probability values and is thus disregarded.

The space complexity of all three algorithms is noted in table 10. From this table, we can discern that the space complexity of the exhaustive algorithm is the greatest with an exponential increase in the required memory space as the surface code depth increases. As noted in section 3.2.2.1, the heap space required to run anything past depth 3 is infeasible by any HPC standard. The world's fastest supercomputer according to the TOP500 Project, Fujitsu's Fugaku, has over 4 petabytes of total memory available for use across 158,976 nodes [28][20]. Even with these hardware specifications, the memory overhead required to run the exhaustive algorithm with depth five or larger is simply too great for the algorithm to run to completion.

4.2.2 Parallel Algorithm Performance

This section details the results and analysis of the parallel implementations of the algorithms from the previous section. This includes not only an analysis of the time and space complexity, but also an examination of the parallel components of the algorithms, such as process decomposition and communications overhead.

Data Set	Space Complexity
Exhaustive	$O(4^n)$
Rand (w/ Repl)	$O(n^2 + m)$
Rand (w/o Repl)	$O(n^2 + m)$

Table 10: Space complexities for all three serial surface code algorithms.

4.2.2.1 Time and Space Complexity

The parallel time complexity of each of the three surface code simulation algorithms is expressed as a function of the surface code depth, number of processes, and—in the case of the random sampling algorithms—the number of samples. Although, when there is only one process, the time complexity is the same as that of the respective algorithm's serial implementation. The functions describing the time complexities of each algorithm are expressed in equations 3, 4, and 5.

When the number of processes for each algorithm exceeds one, the time complexity is simply divided by the number of processes in which the parallel-capable portions of the program are decomposed. In the exhaustive version, the entire outer loop runs in parallel by means of block decomposition, where each block of processes executes on it's own process. Since the sections of code that have the highest order time complexities are also capable of running in parallel, then the overall time complexity is split between the number of processes, p. Thus, the time complexity of the parallel exhaustive algorithm is $O(n4^n/p)$, as shown in eq. (3).

$$T(n,p)_{exhaustive} = \begin{cases} p = 1, & O(n4^n) \\ p > 1, & O(n4^n/p) \end{cases} \tag{3}$$

For the random sampling algorithms, the same principle of dividing the time complexities by the number of processes still applies. However, since the creation of the array of probability values is the segment in both algorithms with the highest-order, and since these code segments occur outside of the loop that is ran in parallel, it is not decomposed. Both algorithms are decomposed into process blocks based on the number of samples being generated, where each process executes a block of iterations of the loop starting on line 13 of algorithm 10. Therefore, only the time complexity of the code within this loop is divided by the number of processes, as shown in eq. (4) and eq. (5).

$$T(n,m,p)_{random,\ with\ replacement} = \begin{cases} p = 1, & O(n^2 log\ n^2 + mn\ log\ n) \\ p > 1, & O(n^2 log\ n^2 + mn\ log\ n/p) \end{cases} \tag{4}$$

$$T(n,m,p)_{random,\ without\ replacement} = \begin{cases} p = 1, & O(n^2 log\ n^2 + m(n\ log\ n + log\ m)) \\ p > 1, & O(n^2 log\ n^2 + m(n\ log\ n + log\ m)/p) \end{cases} \tag{5}$$

When evaluating the time complexities of all three parallel algorithms against each other, it is clear that the exhaustive algorithm benefits the most from a parallel model, since the entire time complexity is reduced proportional to the number of processes used to run the algorithm.

The space complexity of each parallel algorithm is the same as the space complexity of their respective serial implementations. In the serial version of the exhaustive algorithm, the space occupied by the output file is the data structure with the highest order. Since each process in the parallel algorithm writes to the same file, the number of processes will have no effect on the space that the file occupies. For the random sampling algorithms, there are two components of the time complexities in table 10 that must be considered: n^2 and m. The n^2 value refers to the data structures that are used to implement Pascal's triangle and the array containing the list of structures used to store probability values for all combinations of error. Since these data structures must be accessible by each process in order for the algorithm to function, these cannot be split across multiple processes. Similarly, m is used to reference the size of the output file that, like the exhaustive algorithm, cannot be reduced in size no matter the number of processes implemented. Thus, the space complexity values of the parallel algorithms are the same as the serial space complexity values in table 10.

4.2.2.2 Run Time and Speedup

The run times for each algorithm are evaluated at a varying number of processes for each depth. The results for depth three are shown in table 11, depth five in table 12, and depth seven in table 13.

The best-case speedup for each parallel algorithm over their serial implementations is described in eq. (6). These values are noted in table 14, table 15, and table 16. Based on the speedup values obtained in these tables, we can deduce that, for most data sets, the speedup from serial to parallel implementations tends to decrease as the number of processes is increased. For depth three, the exhaustive algorithm outperforms the serial implementation when executed on two processes. The random sampling algorithms without replacement for depths three and five also outperform the serial implementations for every number of processes on data sets of size 10,000. For depth seven, the random sampling algorithm with replacement shows superior performance over the serial version for all data sets with between 100,000 to 500,000 samples. It appears that the speedup increases as the number of samples is increased and as the number of processes is reduced. This is most likely due to the high communications overhead generated by message-passing operations used as part of the MPI-IO library to write from multiple processes to one file.

$$Speedup = \frac{Average\ serial\ run\ time}{Average\ parallel\ run\ time} \tag{6}$$

Algorithm	Samples	Processes	Run Time
Exhaustive	262,144	2	1.484
Exhaustive	262,144	4	3.188
Exhaustive	262,144	8	3.359
Exhaustive	262,144	12	4.906
Rand (w/ repl)	1,000	2	0.031
Rand (w/ repl)	1,000	4	0.016
Rand (w/ repl)	1,000	8	0.047
Rand (w/ repl)	1,000	12	0.063
Rand (w/ repl)	100,000	2	0.609
Rand (w/ repl)	100,000	4	1.203
Rand (w/ repl)	100,000	8	1.313
Rand (w/ repl)	100,000	12	2.250
Rand (w/o repl)	1,000	2	0.031
Rand (w/o repl)	1,000	4	0.016
Rand (w/o repl)	1,000	8	0.047
Rand (w/o repl)	1,000	12	0.031
Rand (w/o repl)	10,000	2	0.266
Rand (w/o repl)	10,000	4	0.109
Rand (w/o repl)	10,000	8	0.063
Rand (w/o repl)	10,000	12	0.063

Table 11: Average run times for depth three parallel algorithms.

Algorithm	Samples	Processes	Run Time
Rand (w/ repl)	1,000	2	0.016
Rand (w/ repl)	1,000	4	0.047
Rand (w/ repl)	1,000	8	0.047
Rand (w/ repl)	1,000	12	0.047
Rand (w/ repl)	100,000	2	0.797
Rand (w/ repl)	100,000	4	1.313
Rand (w/ repl)	100,000	8	1.438
Rand (w/ repl)	100,000	12	2.109
Rand (w/o repl)	1,000	2	0.016
Rand (w/o repl)	1,000	4	0.016
Rand (w/o repl)	1,000	8	0.016
Rand (w/o repl)	1,000	12	0.031
Rand (w/o repl)	10,000	2	0.172
Rand (w/o repl)	10,000	4	0.078
Rand (w/o repl)	10,000	8	0.047
Rand (w/o repl)	10,000	12	0.109

Table 12: Average run times for depth five parallel algorithms.

Algorithm	Samples	Processes	Run Time
Rand (w/ repl)	1,000	2	0.031
Rand (w/ repl)	1,000	4	0.047
Rand (w/ repl)	1,000	8	0.031
Rand (w/ repl)	1,000	12	0.031
Rand (w/ repl)	100,000	2	1.359
Rand (w/ repl)	100,000	4	1.594
Rand (w/ repl)	100,000	8	1.594
Rand (w/ repl)	100,000	12	2.156
Rand (w/ repl)	500,000	2	7.203
Rand (w/ repl)	500,000	4	7.984
Rand (w/ repl)	500,000	8	7.219
Rand (w/ repl)	500,000	12	8.156
Rand (w/o repl)	1,000	2	0.031
Rand (w/o repl)	1,000	4	0.031
Rand (w/o repl)	1,000	8	0.063
Rand (w/o repl)	1,000	12	0.031

Table 13: Average run times for depth seven parallel algorithms.

Algorithm	Samples	Processes	Speedup
Exhaustive	262,144	2	1.039
Exhaustive	262,144	4	0.484
Exhaustive	262,144	8	0.459
Exhaustive	262,144	12	0.314
Rand (w/ repl)	1,000	2	0.128
Rand (w/ repl)	1,000	4	0.128
Rand (w/ repl)	1,000	8	0.085
Rand (w/ repl)	1,000	12	0.064
Rand (w/ repl)	100,000	2	0.619
Rand (w/ repl)	100,000	4	0.313
Rand (w/ repl)	100,000	8	0.287
Rand (w/ repl)	100,000	12	0.168
Rand (w/o repl)	1,000	2	0.416
Rand (w/o repl)	1,000	4	0.831
Rand (w/o repl)	1,000	8	0.277
Rand (w/o repl)	1,000	12	0.416
Rand (w/o repl)	10,000	2	3.592
Rand (w/o repl)	10,000	4	8.720
Rand (w/o repl)	10,000	8	15.264
Rand (w/o repl)	10,000	12	15.264

Table 14: Speedup over serial algorithms for depth three parallel algorithms.

Algorithm	Samples	Processes	Speedup
Rand (w/ repl)	1,000	2	0.768
Rand (w/ repl)	1,000	4	0.256
Rand (w/ repl)	1,000	8	0.256
Rand (w/ repl)	1,000	12	0.256
Rand (w/ repl)	100,000	2	1.358
Rand (w/ repl)	100,000	4	0.824
Rand (w/ repl)	100,000	8	0.753
Rand (w/ repl)	100,000	12	0.513
Rand (w/o repl)	1,000	2	0.896
Rand (w/o repl)	1,000	4	0.896
Rand (w/o repl)	1,000	8	0.896
Rand (w/o repl)	1,000	12	0.488
Rand (w/o repl)	10,000	2	1.681
Rand (w/o repl)	10,000	4	3.699
Rand (w/o repl)	10,000	8	6.165
Rand (w/o repl)	10,000	12	2.642

Table 15: Speedup over serial algorithms for depth five parallel algorithms.

Algorithm	Samples	Processes	Speedup
Rand (w/ repl)	1,000	2	0.864
Rand (w/ repl)	1,000	4	0.576
Rand (w/ repl)	1,000	8	0.864
Rand (w/ repl)	1,000	12	0.864
Rand (w/ repl)	100,000	2	1.741
Rand (w/ repl)	100,000	4	1.485
Rand (w/ repl)	100,000	8	1.485
Rand (w/ repl)	100,000	12	1.098
Rand (w/ repl)	500,000	2	1.642
Rand (w/ repl)	500,000	4	1.481
Rand (w/ repl)	500,000	8	1.638
Rand (w/ repl)	500,000	12	1.450
Rand (w/o repl)	1,000	2	0.086
Rand (w/o repl)	1,000	4	0.086
Rand (w/o repl)	1,000	8	0.432
Rand (w/o repl)	1,000	12	0.086

Table 16: Speedup over serial algorithms for depth seven parallel algorithms.

4.3 Neural Network Decoder

This section details the training and performance results for the neural network decoders. The results listed in this section assume a 5% measurement error using all of the neural network parameters that were chosen in chapter III.

4.3.1 Training and Execution Times

The pipeline that is responsible for training the neural network decoders in this research is comprised of two primary components: a pre-processing phase and a training phase. The pre-processing phase, as explained in chapter III, simply takes the data from a CSV file that was produced using one of the HPC algorithms and formats it into arrays such that they can be used in the training of neural networks. This process takes a considerable amount of time for large data sets. Training the neural network decoders on each of these data sets is also a time-consuming process, especially for data sets with a large number of samples. The pre-processing and training times for each surface code depth and data set are listed in table 17, table 18, and table 19. Training times increase predictably with the number of samples and with the depth of the surface code.

The mean execution times for each decoder at each depth for all data sets are recorded in tables 20, 21, and 22 along with their respective standard deviation values.

Data Set	Samples	Pre-processing Time (sec)	Training Time (sec)
Baseline	28	0.1	1.4
Exhaustive	262,144	346.2	951.7
Rand (w/ repl)	1,000	1.5	5.3
Rand (w/ repl)	10,000	13.0	39.3
Rand (w/ repl)	100,000	131.2	351.7
Rand (w/o repl)	1,000	1.4	5.5
Rand (w/o repl)	10,000	13.9	40.1

Table 17: Pre-processing and training time for depth three data sets.

Data Set	Samples	Pre-processing Time (sec)	Training Time (sec)
Baseline	2,776	11.8	64.2
Rand (w/ repl)	1,000	3.2	19.3
Rand (w/ repl)	10,000	24.3	130.6
Rand (w/ repl)	100,000	169.3	861.1
Rand (w/o repl)	1,000	4.1	23.7
Rand (w/o repl)	10,000	40.1	205.5

Table 18: Pre-processing and training time for depth five data sets.

Data Set	Samples	Pre-processing Time (sec)	Training Time (sec)
Baseline	508,180	4,388.0	4460.7
Rand (w/ repl)	1,000	8.9	11.6
Rand (w/ repl)	10,000	82.8	103.0
Rand (w/ repl)	100,000	763.5	897.5
Rand (w/ repl)	500,000	3,590.6	4,030.9
Rand (w/o repl)	1,000	9.0	11.3

Table 19: Pre-processing and training time for depth seven data sets.

In all cases, the neural network decoder outperforms the MWPM decoder. From this data it is also clear that the execution times for the neural network decoders approach the execution times of the for the partial lookup table decoders as the number of samples in the data set increases. This is evident in the case of the exhaustive and 100,000-sample random data sets in table 20 and in the case of the baseline and 500,000-sample random data sets in table 22.

Data Set	**NN**	**MWPM**	**PLUT**
Baseline	0.045 (±0.002)	0.042 (±0.005)	0.000 (±0.000)
Exhaustive	0.831 (±0.073)	506.636 (±29.866)	12.321 (±0.236)
Rand (1k w/ repl)	0.042 (±0.002)	1.554 (±0.043)	0.002 (±0.000)
Rand (10k w/ repl)	0.056 (±0.002)	16.019 (±0.082)	0.028 (±0.002)
Rand (100k w/ repl)	0.186 (±0.007)	164.853 (±0.753)	0.403 (±0.092)
Rand (1k w/o repl)	0.045 (±0.003)	1.933 (±0.118)	0.003 (±0.000)
Rand (10k w/o repl)	0.056 (±0.000)	20.203 (±0.340)	0.040 (±0.001)

Table 20: Mean execution times (in seconds) and standard deviations for all depth three decoders on all data sets

Data Set	**NN**	**MWPM**	**PLUT**
Baseline	0.052 (±0.004)	25.662 (±0.331)	0.010 (±0.000)
Rand (1k w/ repl)	0.045 (±0.002)	7.637 (±0.120)	0.003 (±0.000)
Rand (10k w/ repl)	0.061 (±0.001)	61.898 (±0.890)	0.021 (±0.001)
Rand (100k w/ repl)	0.163 (±0.002)	443.826 (±1.950)	0.159 (±0.005)
Rand (1k w/o repl)	0.050 (±0.006)	9.652 (±0.131)	0.003 (±0.000)
Rand (10k w/o repl)	0.077 (±0.002)	104.855 (±0.812)	0.035 (±0.001)

Table 21: Mean execution times (in seconds) and standard deviations for all depth five decoders on all data sets

Data Set	**NN**	**MWPM**	**PLUT**
Baseline	4.405 (±0.025)	40.772 (±0.888)	5.622 (±0.841)
Rand (1k w/ repl)	0.059 (±0.002)	41.933 (±0.370)	0.009 (±0.001)
Rand (10k w/ repl)	0.131 (±0.001)	43.510 (±0.637)	0.088 (±0.008)
Rand (100k w/ repl)	0.826 (±0.035)	43.585 (±0.764)	0.804 (±0.030)
Rand (500k w/ repl)	3.566 (±0.160)	43.894 (±1.197)	3.728 (±0.115)
Rand (1k w/o repl)	0.058 (±0.003)	41.796 (±1.155)	0.009 (±0.000)

Table 22: Mean execution times (in seconds) and standard deviations for all depth seven decoders on all data sets

4.3.2 Accuracy, F1 Score, and Confusion Matrix

4.3.2.1 Accuracy

In fig. 18, the accuracy values from the baseline data sets used in previous research is shown [7]. Figure 19 details the effect of depth, data set type, and data set size on the accuracy of all three decoders. Additionally, Figure 20 shows the accuracy of the decoders trained on the depth three exhaustive data set and the depth seven random sampled data set with 500,000 samples. These figures are shown separately because these data sets were not collected for the other surface code depths.

Unlike the analysis performed in the previous research, which measured accuracy with varying percentages of measurement error on the surface code, the assumption is made in this research that the measurement error will remain at 5%[7]. The baseline data set produced the results that will serve at the reference point for how the decoders

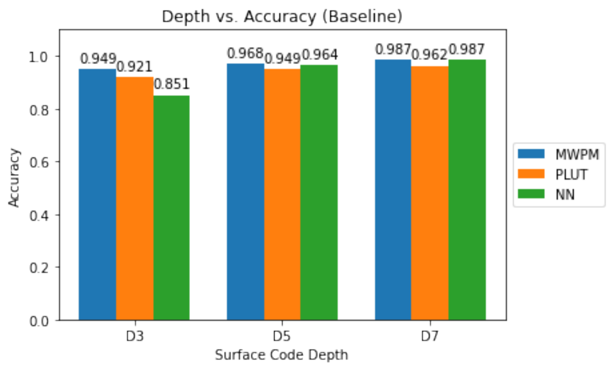
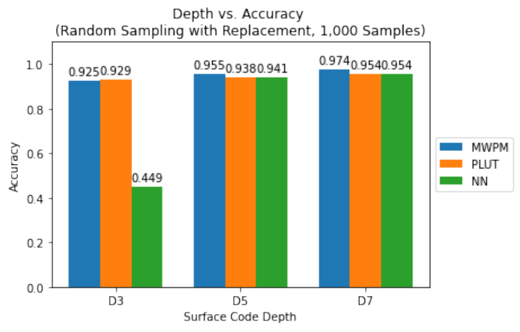
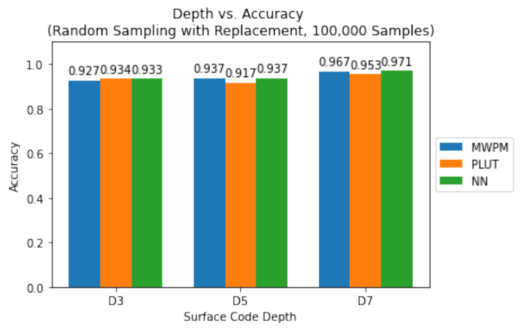


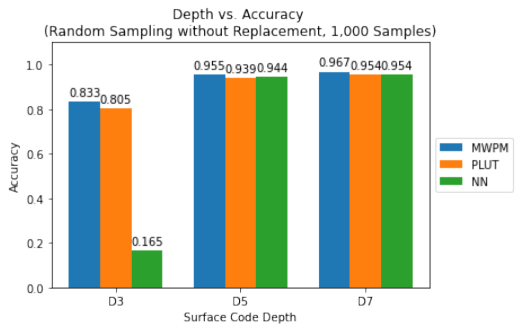
Figure 18: Depth vs. Accuracy for the decoders trained on the baseline data sets used in [7]



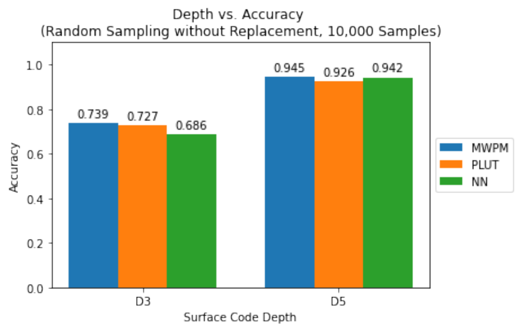
(a) 1,000 samples with replacement



(b) 100,000 samples with replacement

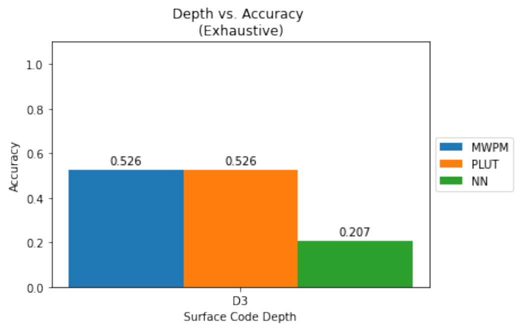


(c) 1,000 samples without replacement

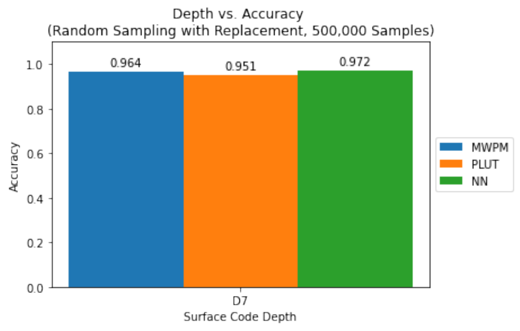


(d) 10,000 samples without replacement

Figure 19: Depth vs. Accuracy for the decoders trained on the random sampled data sets of various sample sizes



(a) Exhaustive depth three data set



(b) 500,000 samples with replacement

Figure 20: Depth vs. Accuracy for the decoders trained on the exhaustive data set (a) and the random sampled data set for depth seven (b)

trained on other data sets perform. Based on the results in fig. 18, the neural network decoder only outperforms the other decoders in the case of the depth seven model. While the accuracy of the neural network is comparable to the minimum weight perfect matching and partial lookup table decoders for depths five and seven, the depth three neural network decoder falls to almost 10% below the accuracy of the best-performing decoder.

The graphs in fig. 19 show that the accuracy of the neural network decoders increase with respect to the number of samples in the data set. For the data sets that contain 1000 samples that are sampled with replacement, the neural network decoder performs worse than the other decoders at depth three, with an accuracy of only 44.9%. The decoder performs much better on code depths five and seven with accuracy values of 94.1% and 95.4%, compared to the respective minimum weight perfect match accuracy values of 95.5% and 97.4%. When the number of samples in the data set is increased from 1,000 to 100,000, we see that the accuracy of the neural network decoder meets or exceeds the accuracy of the other decoders in all cases except for depth three, where the partial lookup table has a 0.1% higher accuracy than the neural network.

Although the algorithm responsible for generating random sampled data without replacement requires too much heap space to run on large sample quantities, we can see from the graphs in fig. 19 that accuracy tends to increase with the number of samples.

4.3.2.2 F1 Score

The data in fig. 21 shows the F1 score values using the data sets used in previous research [7]. In fig. 22, the F1 scores are listed for the decoders trained on the random sampled data sets for all surface code depths. The F1 scores for the depth three decoders trained using the exhaustive data set and the depth seven decoders trained using the random sampling data set with replacement are shown in fig. 23.

For both of the random sampled data sets, the F1 score decreases as the surface code depth increases for data sets with a small quantity of samples. This trend is reversed when the number of samples in the data set is increased. Although this increase in sample size allows for a higher F1 score, the scores for the decoders trained on the baseline data set are higher for depths five and seven. For depth three, the F1 scores for neural network decoders trained on all data set are higher than the score found in previous work.

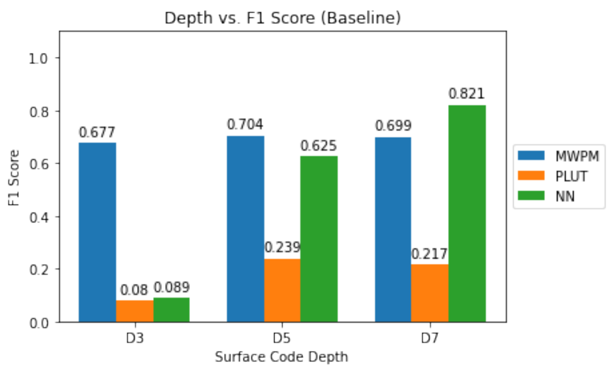
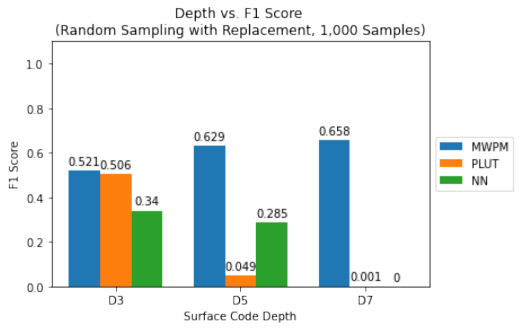
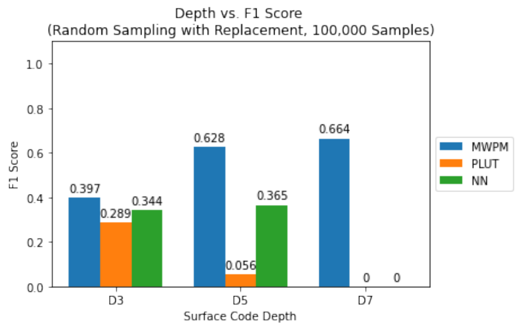


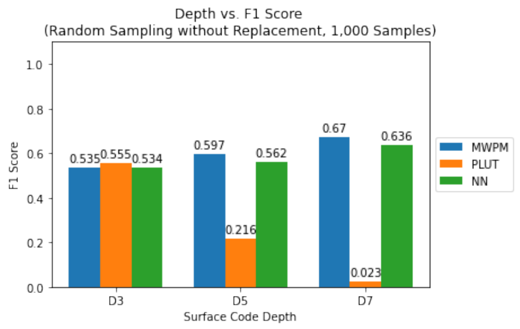
Figure 21: Depth vs. F1 Score for the decoders trained on the baseline data sets used in [7]



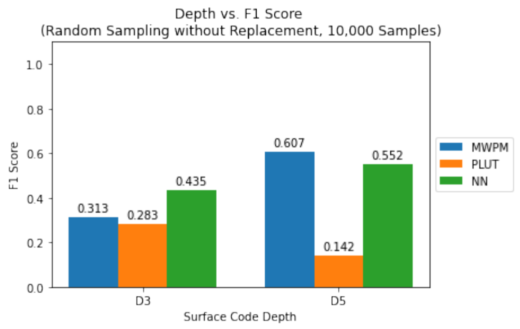
(a) 1,000 samples with replacement



(b) 100,000 samples with replacement

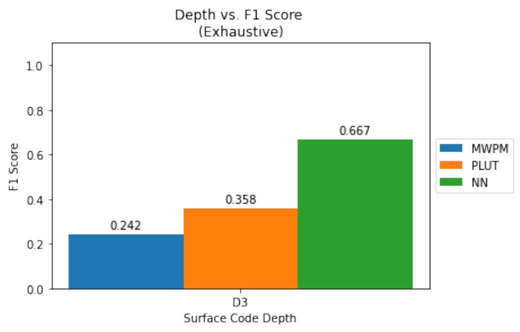


(c) 1,000 samples without replacement

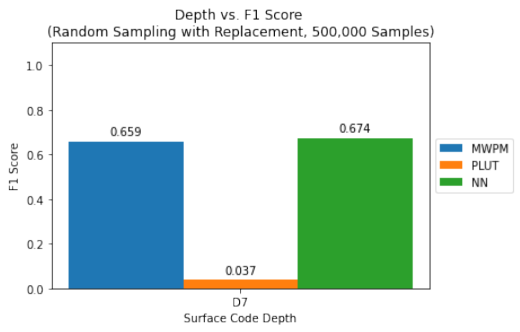


(d) 10,000 samples without replacement

Figure 22: Depth vs. F1 Score for the decoders trained on the random sampled data sets of various sample sizes



(a) Exhaustive depth three data set



(b) 500,000 samples with replacement

Figure 23: Depth vs. F1 Score for the decoders trained on the (a) exhaustive data set and the (b) random sampled data set for depth seven

4.3.2.3 Confusion Matrix

While the accuracy and F1 scores are intended to provide insight into the performance of a classifier, a confusion matrix provides a much more detailed analysis of what the classifier is actually doing with each sample. The confusion matrices in table 23, for instance, show that the neural network decoder trained on the baseline data set for depth three is only predicting no error, regardless of the ancilla qubit values, and that the baseline neural network decoders for depth five and depth seven are more likely to correctly discern whether or not an error exists on a surface code.

The confusion matrix values for the neural network decoders trained on the newly-implemented data sets are listed in tables 27 through 33 in appendix A. Of the decoders listed in these tables, the neural network decoders trained on smaller data sets are much less likely to predict true positives and true negatives than neural networks trained on similar data sets with a greater number of samples.

4.3.3 ROC Curve and AUC

The ROC curves in fig. 24 show the performance of the baseline neural network decoders [7]. ROC curves show the average true positive rate of the binary classifiers used in each output node for each fold during cross-validation. The area under the curve (AUC) will be closer to 1 the more effective the classifiers are. Anything below the 0.5 mark is considered poor performance. According to the ROC curves for the baseline neural networks, the depth three decoder performs poorly at a meager AUC of 0.33, while the depths five and seven decoders perform significantly better with AUC values of 0.96 and 0.99, respectively.

The results from the k-fold cross validation, which was performed on each data set on all surface code depths, produced the ROC curves in figures 29 through 44 in appendix A. All depth three decoders trained in this research perform better than

		Test	
		Positive	Negative
Predict	Positive	0	0
	Negative	6	96

(a) Depth three

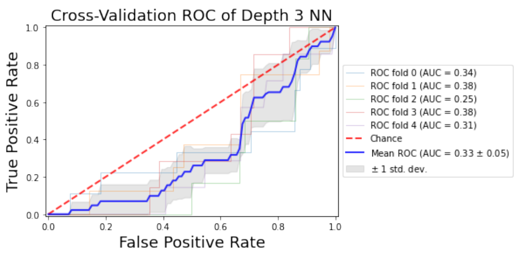
		Test	
		Positive	Negative
Predict	Positive	929	309
	Negative	840	32,572

(b) Depth five

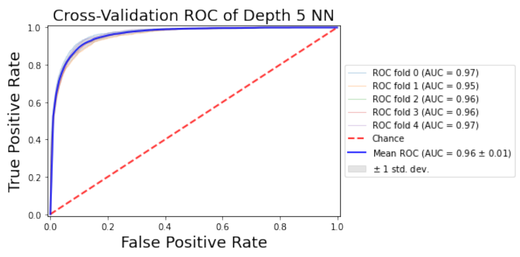
		Test	
		Positive	Negative
Predict	Positive	403,451	52,594
	Negative	96,031	11,898,236

(c) Depth seven

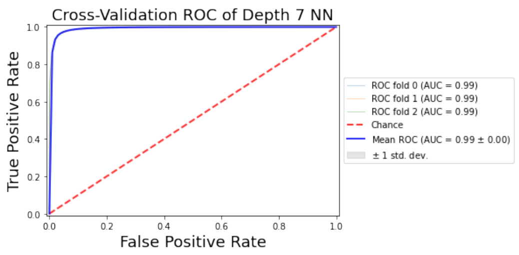
Table 23: Confusion matrices for neural network decoders trained on baseline data sets



(a) Depth three



(b) Depth five



(c) Depth seven

Figure 24: ROC Curves for the neural networks trained on the baseline data sets

the baseline decoders, with the lowest AUC of 0.55 belonging to the decoder that was trained on the exhaustive data set. The best performance is found in the depth three neural network trained on the random sampled data set with replacement, which has an AUC of 0.93.

For the depth five neural network decoders, the lower and upper bounds for AUC are 0.84, for the decoder trained on the random sampled (with replacement) data set with 1,000 samples, and 0.93, for the decoder trained on the random sampled (without replacement) data set with 10,000 samples (see figures 35 and 39 in appendix A).

For neural network decoders for the surface code of depth seven, we see that the decoder trained on the random sampled (with replacement) data set with 1,000 samples offers the worst performance with an AUC of 0.57, as shown in fig. 40. This ROC curve in fig. 43 shows us that the decoder trained on the random sampled (with replacement) data set with 500,000 samples, however, performs best for this depth with an AUC of 0.97.

For all code depths, the performance increases as the number of samples in the data set grows. This, however, is not true for the depth three neural network decoder for the exhaustive data set, which has a smaller AUC than all of the other depth three neural networks. This exhaustive model still performs better than the baseline depth three neural network in this regard, however.

4.3.4 Statistical Analysis

The results from McNemar's test are shown in figures 25, 26, and 27. During the k-fold cross-validation, each decoder is assigned a contingency table that stores the true vs. predicted values for the test data set. Using the data stored in these contingency tables, the x^2 is calculated by the formula [11]:

$$x^2 = \frac{(|b - c| - 1)^2}{b + c}$$

where b is the number of times the neural network incorrectly decoded the test set and c is the number of times that either the MWPM algorithm or partial lookup table provided the incorrect response. From this x^2 value, probability values are calculated and plotted. These probability values can be seen in figures 25, 26, and 27, which show the results for all data sets on depths three, five, and seven, respectively.

A dotted line is also added to figures 25—27 in order to illustrate the probability value past which the null hypothesis that the two decoders being compared perform the same is rejected. In fig. 25, it is most immediately clear that the neural network model trained on the exhaustive data set produces a mean probability value exceeding the 5% threshold, but this is not so clear for the neural networks trained on other data sets. Based on the data collected in tables 34—39, however, we know that the mean probability values for all models are greater than 5% when compared to the other decoders, except for three instances. Both of the depth three decoders trained with the random sampled data sets without replacement have mean probability values less than 5% when compared against the MWPM decoder. Additionally, the depth seven decoder trained with the random sampled data set without replacement has a probability value of 0.1% when compared to the partial lookup table. Thus, for these three instances, we reject the null hypothesis that the performance of each set of decoders is the same.

4.3.5 Implementation Results

In this section, the results from the dynamic surface code simulation, which demonstrate the effectiveness of each neural network model in a quantum circuit, are shown and analyzed.

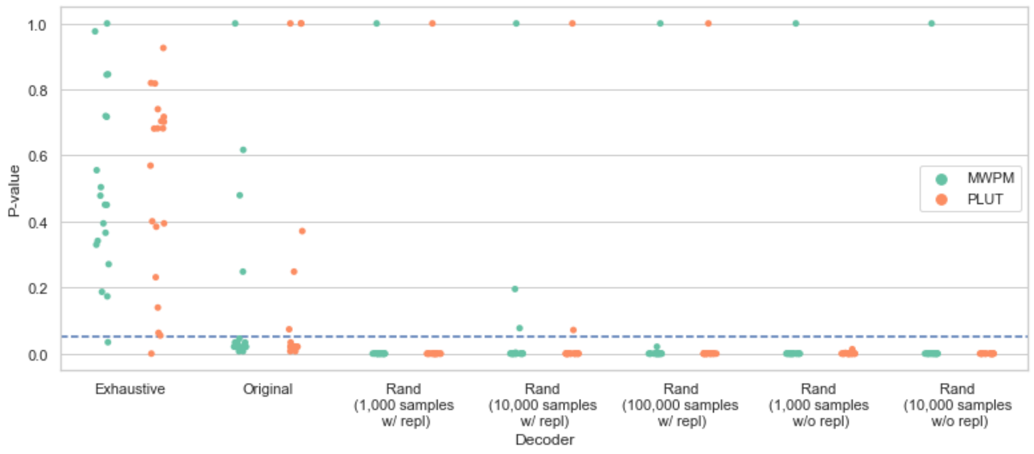


Figure 25: McNemar's Test results for depth three decoders for all data sets

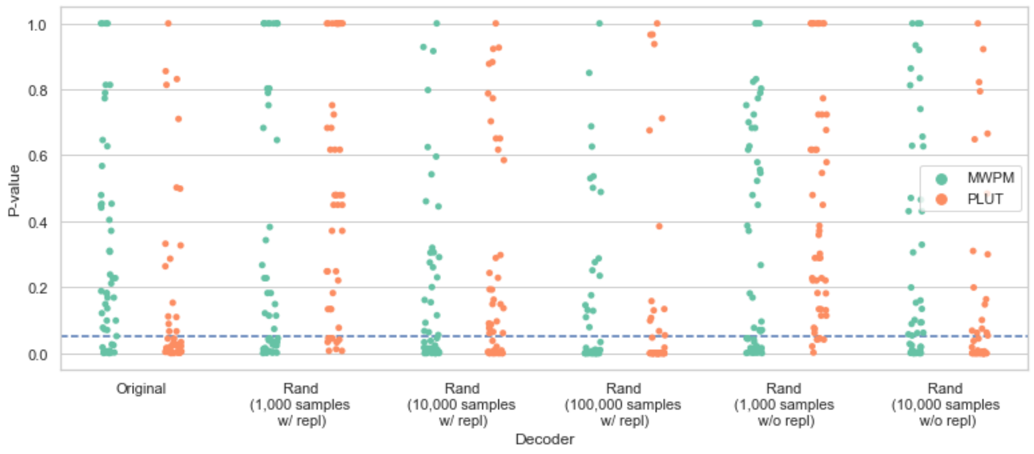


Figure 26: McNemar's Test results for depth five decoders for all data sets

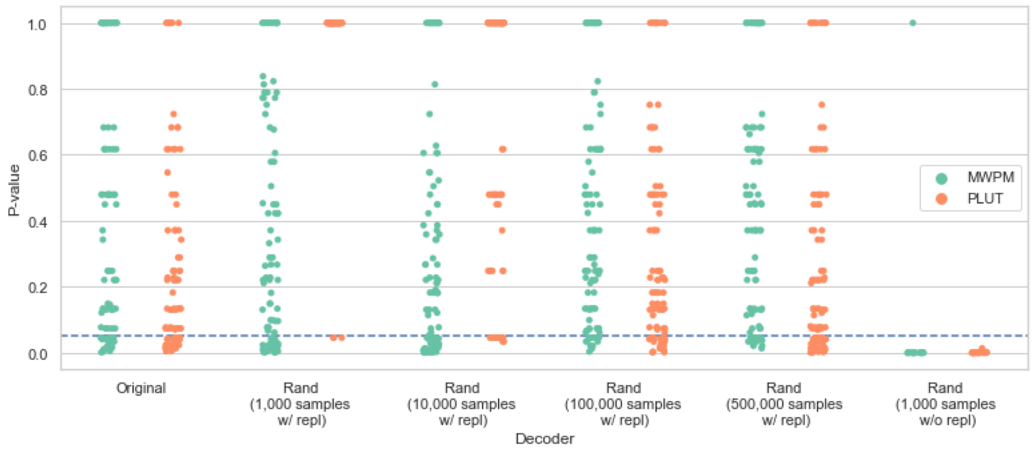


Figure 27: McNemar's Test results for depth seven decoders for all data sets

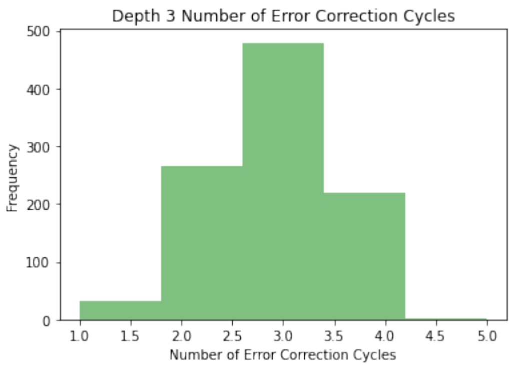
4.3.5.1 Simulation Performance

The error correction cycles for the implementation of the neural network decoders trained on the baseline data sets are shown in fig. 28. The mean and standard deviation error correction cycles for these data sets are also shown in figures 40, 41, and 42. In previous research, it was stated that if the mean error correction cycles exceeds the surface code depth for a particular decoder, than it is capable of performing better than chance [7]. Based on the values from these tables, we know that the depth three decoder does not perform better than chance, but the depths five and seven decoders do.

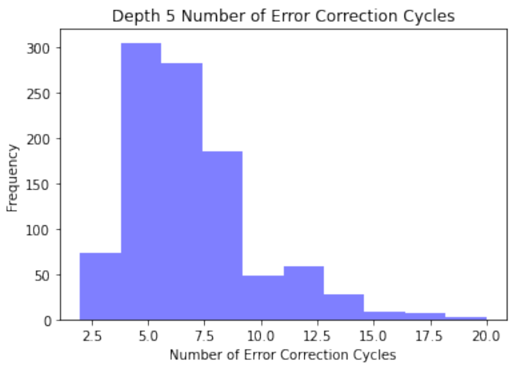
The decoders trained on the new data sets are shown in figures 45—51 in appendix A, with mean and standard deviation error correction cycles listed in tables 40, 41, and 42 in appendix A. While none of the depth three decoders produce results indicative of a well-performing decoder, with the highest mean error correction cycles value at 2.868 for the random sampled data set with replacement at 1,000 samples, some decoders at depths five and seven show promising results.

For depth five, all of the decoders except for the ones with quantities of 1,000 samples perform better than chance. In fact, the decoder that was trained on the random sampled data set without replacement with 10,000 samples performs the best with a mean error correction cycles value of 7.967. This shows a performance boost over the baseline depth five decoder, which had a mean number of error correction cycles of 6.917.

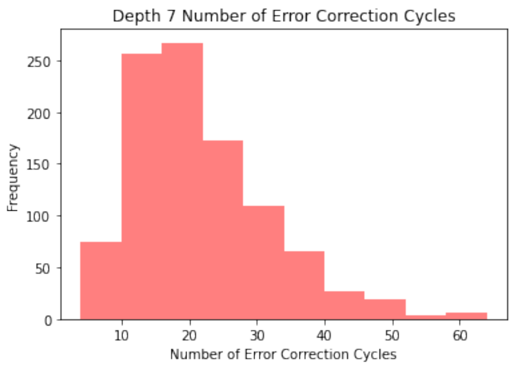
For the depth seven decoders, once again, all decoders have a mean error correction cycles value higher than the surface code depth except for those trained on the data sets containing 1,000 samples. This suggests that all other decoders perform better than chance. Unlike in the depth five experiments, however, the baseline depth seven decoder performs over twice as well as even the highest performing decoder trained



(a) Depth three



(b) Depth five



(c) Depth seven

Figure 28: Error correction cycles for the implementation of the neural networks trained on the baseline data sets

on newer data sets, with a mean error correction cycles value of 20.974.

4.3.5.2 Kruskal-Wallis H-Test

In order to determine whether any of the results in subsection 4.3.5.1 are statistically significant, statistical analysis of these results is necessary. Since the arrays containing error correction cycles for each model are non-parametric, or not normally-distributed, then the Kruskal-Wallis test is used to test the null hypothesis. For each neural network model, the null hypothesis is that there is no difference in performance, or mean error correction cycles, between the model under test and the baseline model used in previous research [7]. The results of these tests are shown in tables 24, 25, and 26.

Model	**Statistic**	**P-value**
Exhaustive	1,661.667	0.0
Rand (1,000 samples w/ repl)	359.728	3.226×10^{-80}
Rand (10,000 samples w/ repl)	6.594	0.010
Rand (100,000 samples w/ repl)	18.751	1.489×10^{-5}
Rand (1,000 samples w/o repl)	108.790	1.804×10^{-25}
Rand (10,000 samples w/o repl)	423.022	5.366×10^{-94}

Table 24: Krustkal-Wallis H-Test results for all depth three neural network models tested against the baseline model

Model	**Statistic**	**P-value**
Rand (1,000 samples w/ repl)	465.999	2.378×10^{-103}
Rand (10,000 samples w/ repl)	106.974	4.509×10^{-25}
Rand (100,000 samples w/ repl)	55.536	9.172×10^{-14}
Rand (1,000 samples w/o repl)	520.818	2.807×10^{-115}
Rand (10,000 samples w/o repl)	20.893	4.855×10^{-6}

Table 25: Krustkal-Wallis H-Test results for all depth five neural network models tested against the baseline model

Tables 24, 25, and 26 show that each neural network decoder model, when compared to the baseline models, has a p-value less than 0.05. This means that we reject

Model	**Statistic**	**P-value**
Rand (10,000 samples w/ repl)	1500.603	0.0
Rand (100,000 samples w/ repl)	1095.413	3.278×10^{-240}
Rand (500,000 samples w/ repl)	853.625	1.184×10^{-187}
Rand (1,000 samples w/o repl)	1445.819	0.0

Table 26: Krustkal-Wallis H-Test results for all depth seven neural network models tested against the baseline model

the null hypothesis that there is no difference in performance, or mean error correction cycles, between all models and the baseline models.

4.4 Summary

This chapter showed the results of the experiments and an analysis of the performance of both the high-performance computing algorithms and the neural network decoders. The serial and parallel implementations of the algorithms designed to produce data that is used in the neural network decoders are compared and evaluated in section 4.2. This section suggests various means of producing several kinds of data sets with different scopes. In section 4.3, the training, cross-validation, and testing of the neural network decoders are evaluated against the results from previous research [7] in terms of a number of performance metrics to include performance and statistical analysis. The results in this section indicate a significant performance boost for the depth three decoder with the introduction of new data sets, while the depths five and seven decoders see insignificant or marginal performance gains.

V. Conclusions

This section concludes the research performed into neural network decoders and data set generation methods for error correction on the rotated surface code. Section 5.1 covers a brief overview of the research performed, a reiteration of the research question, and the results of the hypotheses proposed at the beginning of this research. Section 5.2 states the contributions that this research provides to the research and design of data-generation algorithms and neural network decoders for the rotated surface code. In section 5.3, several ideas for future research endeavors related to this research are detailed. Lastly, 5.4 covers all concluding remarks.

5.1 Overview

Efficient quantum error correction is absolutely necessary for the scalability and performance of quantum computation. In order for the development of quantum computers to reach a point where they are able to serve any sort of practical purpose, like the implementation of Grover's or Shor's algorithms, a means of correcting error on individual qubits is absolutely necessary. With quantum error correction using neural network decoders that perform well under a reduced time budget, the prospect of creating scalable quantum computers becomes much more of a reality. In this research, neural network decoders are trained on various data sets in order to evaluate the effect of data set contents and volume on decoder performance relative to other, more traditional forms of decoding, such as the minimum weight perfect matching algorithm and partial lookup tables.

In section 1.3, the research question is proposed: How does the performance of neural network-based decoders for the surface code with various data set generation techniques compare to traditional algorithmic error correction decoders? The results of this research provide an answer to this question by answering the proposed hypotheses:

1. *Using exhaustive training data sets produce a neural network decoder that performs more effectively and efficiently than decoders used in previous research.*
    - This hypothesis is **true with respect to execution time, F1 score, and AUC for surface code of depth three** (See Sections 4.3.1, 4.3.2.2, and 4.3.3).
    - This hypothesis is **false with respect to accuracy, statistical analysis results, and mean error correction cycles** for surface code of depth three (See Sections 4.3.2.1, 4.3.4, 4.3.5.1).
2. *Training a neural network on data sets that are randomly-sampled without replacement yield greater effectiveness and efficiency than those trained on randomized data sets with duplicate samples.*
    - This hypothesis is **true with respect to accuracy for depth five decoders, F1 score for small data sets, AUROC for depth five decoders, and error correction cycles for depths three and seven decoders with small data sets and depth five decoders with large data sets** (See Sections 4.3.1 through 4.3.5).
    - This hypothesis is **false with respect to accuracy for depth three decoders, F1 score for small data sets, AUROC for depths three and seven decoders, and error correction cycles for depth three decoders with large data sets and depth five decoders with small data sets** (See Sections 4.3.2.1, 4.3.4, 4.3.5.1).
3. *Training a neural network on larger randomly sampled data sets produce neural*

networks that have greater effectiveness and efficiency than those trained on small data sets.

- This is **true with respect to accuracy for surface codes of depth three and seven, F1 score for random sampled data sets with replacement, and error correction cycles for depth three decoders trained on data sets with replacement and depth five decoders trained on data sets without replacement** (Sections 4.3.2.3 and 4.3.3).
- This is **false with respect to accuracy for surface codes of depth five, F1 score for random sampled data sets without replacement, and error correction cycles for depth five decoders trained on data sets with replacement and depth three decoders trained on data sets without replacement** (Sections 4.3.2.3 and 4.3.3).

These result provide a better understanding of which data generation methods are best for the formulation of data sets for use in the training of neural network decoders. They also highlight the pitfalls and bottlenecks of algorithm design for the purpose of creating such data sets and provide the opportunity for future work to revise and optimize these algorithms for the purpose of creating larger, more effective data sets.

5.2 Contributions

5.2.1 Analysis of Algorithmic Generation of Neural Network Training Data for the Rotated Surface Code

The first contribution of this thesis research is the analysis of the various algorithmic approaches to generating data sets for neural network decoders. Various algorithms are written in this research for the purpose of generating sample data sets for use in training neural network decoders. These algorithms include a version that creates an exhaustive data set that writes every possible combination of error to a file along with the respective corrective operation, and two versions that produce random sampled data sets—one with replacement of collected samples and one without. Although the exhaustive algorithm is infeasible past a certain surface code depth, it at least demonstrates how such a data set is produced algorithmically and the memory and disk space required to create it. The data sets produced by the random sampling algorithms also require varying volumes of memory space during their collection. This research provides insight into the various components of these algorithms and their various space and time complexities.

5.2.2 Proof of Impracticality of Exhaustive Data Sets for Large Code Depths

The second contribution of this research shows the hardware limitations to creating exhaustive data sets for the purpose of training a neural network decoder. Although the algorithm responsible for creating exhaustive data sets requires very little disk or memory space to run for the depth three surface code, this research provides evidence that shows the infeasibility of running the algorithm with surface codes of five or greater. This proves that, while an exhaustive data set for surface code errors might seem desirable under certain circumstances, it is impossible by today's computing standards to achieve.

5.2.3 Analysis of the Effect of Data Set Generation Method on Decoder Performance

The third contribution this research makes to the area of neural network-based quantum error correction research is the effect of data set generation method on neural network decoders. This research shows that there are many differences between neural network decoders that are trained using varying data sets. While the performance of the baseline neural network decoders implemented in previous iterations of this research perform better in many cases than any of the decoders that were trained using these newly-created data sets, there are some instances, especially with respect to the depth three surface code, where the neural networks trained on the data sets produced in this research perform better in terms of accuracy and speed, among other performance metrics.

5.3 Future Work

The work done in this research leaves open a number of opportunities for continued research. The high-performance computing algorithms can be optimized to run more efficiently in order to create larger data sets. The algorithm used to generate random samples without replacement, for instance, requires a great deal of heap space in order to run on large surface code depths and for data sets with large volumes. In fact, for depth seven, the algorithm cannot collect more than 1,000 samples without either more powerful hardware, or a redesign or optimization of the code used to create these data sets. With the ability to create larger data sets without replacement, the decoders trained on data from both random sampling algorithms can be better compared against one another. This also opens up the opportunity to train the neural network decoders on larger data sets.

Another possible avenue for future work related to this research would be to evaluate the neural network decoders described in this research against other decoders, including other types of neural network decoders. Possible decoders include the distributed neural network decoders described in [30] and the belief-propagation decoders implemented in [21]. Providing a comparison between the three decoders implemented in this research and the various other decoders implemented in similar work would contribute a more comprehensive analysis of decoder performance for this research area.

5.4 Concluding Remarks

The *National Defense Strategy* lists a growing number of concerns for the United States and among them is the risk of malicious cyber attacks against our information and industrial control systems, both civilian and military [23]. With our peer-adversaries like China already taking steps towards the milestone of achieving quantum supremacy—which would render most modern asymmetric encryption useless—it is paramount that we achieve a means of scalable and fault-tolerant quantum computing to protect us against this growing cyber threat.

Quantum error correction seeks to reduce the frequency of error in an already unstable environment. Qubits are the most fundamental component of a quantum computer and are vulnerable to decoherence and a preponderance of errors that render entire circuits useless for any kind of meaningful use. Unfortunately, most algorithmic means of correcting these errors do not scale well with the introduction of more qubits to the architecture. Thus, training neural networks serves as one of the best alternatives to algorithmic quantum error correction because of the ability of such decoders to operate in constant time.

If these neural network decoders are optimized to a certain degree, it may be possible for the quantum computers of tomorrow that use this technology to outperform classical computing in some meaningful capacity. Producing the world's first quantum computer capable of performing Shor's algorithm to break public key cryptosystems, for example, would provide the U.S. with a significant competitive edge over our adversaries. Furthermore, achieving a means of quantum communication would reduce the threat of compromised communications over classical computing systems.

Appendix A. Tables and Figures

		Test	
		Positive	Negative
Predict	Positive	257,887	260,430
	Negative	299,440	296,338

Table 27: Confusion matrix for neural network decoder trained on depth three exhaustive data set

		Test	
		Positive	Negative
Predict	Positive	2	0
	Negative	332	3,916

(a) 1,000 samples

		Test	
		Positive	Negative
Predict	Positive	1,682	603
	Negative	1,874	38,341

(b) 10,000 samples

		Test	
		Positive	Negative
Predict	Positive	15,935	3,741
	Negative	19,488	385,836

(c) 100,000 samples

Table 28: Confusion matrices for neural network decoders trained on depth three random sampled (with replacement) data sets

		Test	
		Positive	Negative
Predict	Positive	2	3
	Negative	717	3,528

(a) 1,000 samples

		Test	
		Positive	Negative
Predict	Positive	1,507	1,549
	Negative	9,250	30,194

(b) 10,000 samples

Table 29: Confusion matrices for neural network decoders trained on depth three random sampled (without replacement) data sets

		Test	
		Positive	Negative
Predict	Positive	13	6
	Negative	600	9,331

(a) 1,000 samples

		Test	
		Positive	Negative
Predict	Positive	2,228	1,086
	Negative	3,033	69,203

(b) 10,000 samples

		Test	
		Positive	Negative
Predict	Positive	19,861	8,217
	Negative	20,645	463,227

(c) 100,000 samples

Table 30: Confusion matrices for neural network decoders trained on depth five random sampled (with replacement) data sets

		Test	
		Positive	Negative
Predict	Positive	17	14
	Negative	710	11,759

(a) 1,000 samples

		Test	
		Positive	Negative
Predict	Positive	4,065	1,702
	Negative	4,776	114,457

(b) 10,000 samples

Table 31: Confusion matrices for neural network decoders trained on depth five random sampled (without replacement) data sets

		Test	
		Positive	Negative
Predict	Positive	0	0
	Negative	1,133	23,171

(a) 1,000 samples

		Test	
		Positive	Negative
Predict	Positive	1,561	957
	Negative	9,519	226,103

(b) 10,000 samples

		Test	
		Positive	Negative
Predict	Positive	59,694	24,533
	Negative	46,862	2,090,669

(c) 100,000 samples

		Test	
		Positive	Negative
Predict	Positive	308,881	84,933
	Negative	204,581	9,718,555

(d) 500,000 samples

Table 32: Confusion matrices for neural network decoders trained on depth seven random sampled (with replacement) data sets

		Test	
		Positive	Negative
Predict	Positive	0	0
	Negative	1,085	23,415

Table 33: Confusion matrix for neural network decoder trained on depth seven random sampled (without replacement) data set with 1,000 samples

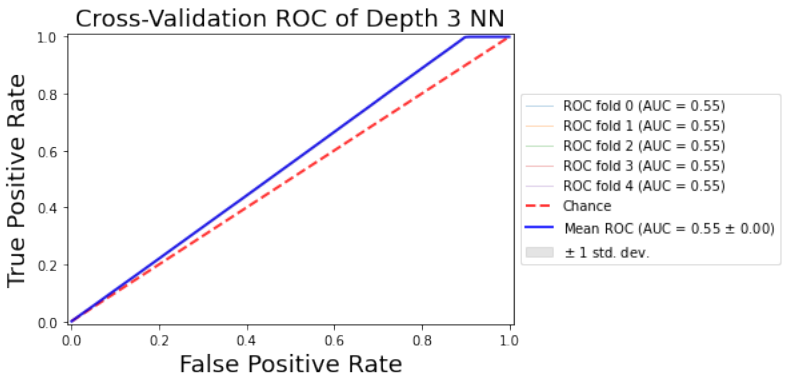


Figure 29: ROC curve for the depth three decoder trained on the exhaustive data set

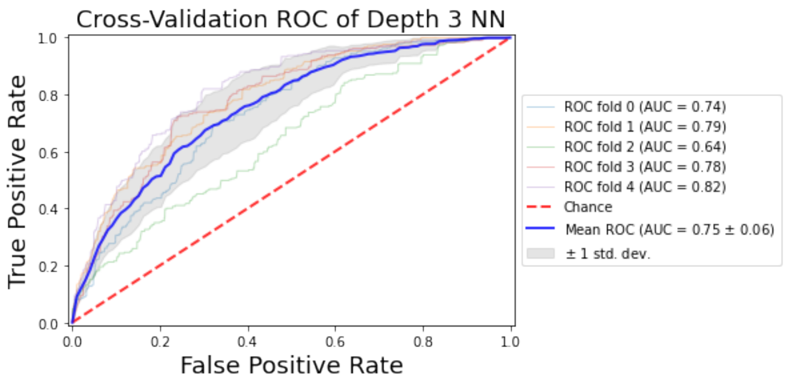


Figure 30: ROC curve for the depth three decoder trained on the random sampled (with replacement) data set with 1,000 samples

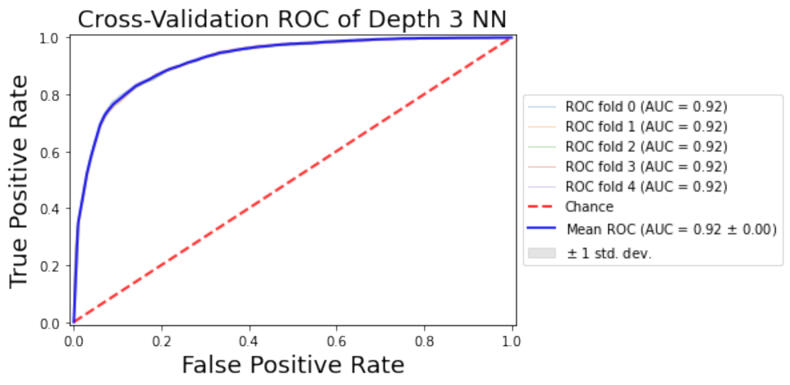


Figure 31: ROC curve for the depth three decoder trained on the random sampled (with replacement) data set with 10,000 samples

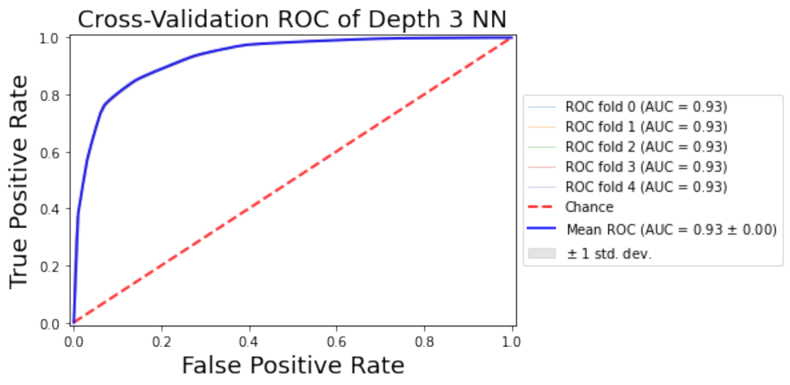


Figure 32: ROC curve for the depth three decoder trained on the random sampled (with replacement) data set with 100,000 samples

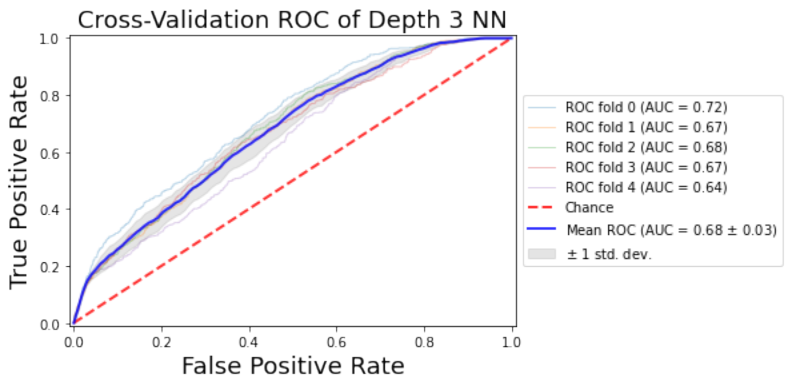


Figure 33: ROC curve for the depth three decoder trained on the random sampled (without replacement) data set with 1,000 samples

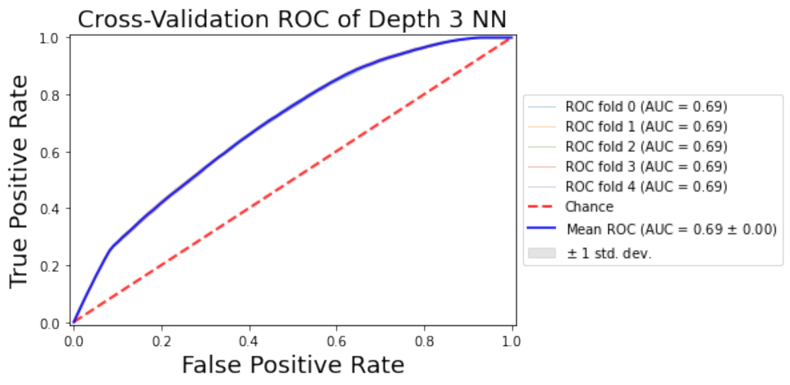


Figure 34: ROC curve for the depth three decoder trained on the random sampled (without replacement) data set with 10,000 samples

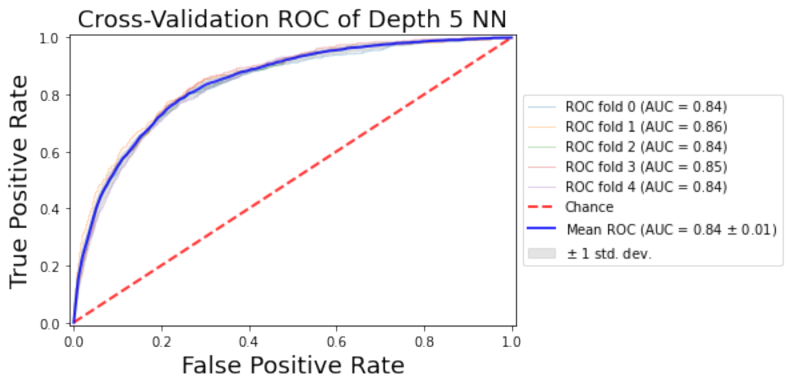


Figure 35: ROC curve for the depth five decoder trained on the random sampled (with replacement) data set with 1,000 samples

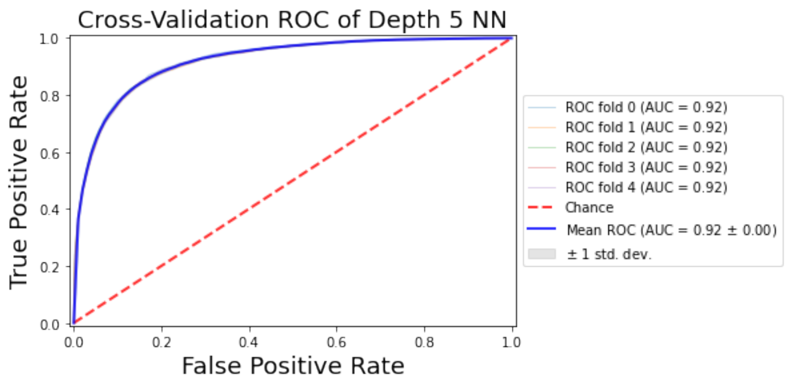


Figure 36: ROC curve for the depth five decoder trained on the random sampled (with replacement) data set with 10,000 samples

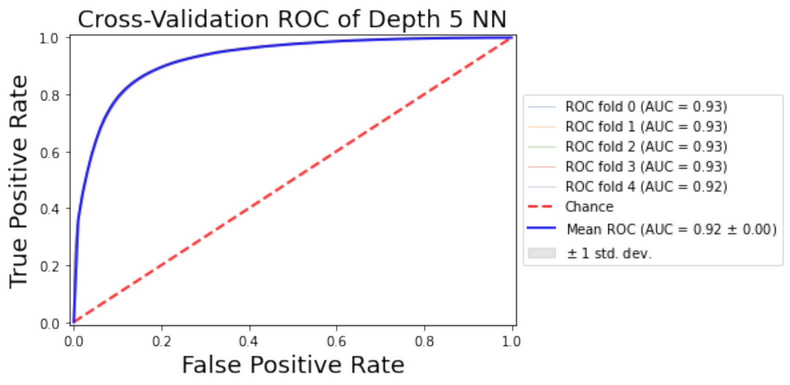


Figure 37: ROC curve for the depth five decoder trained on the random sampled (with replacement) data set with 100,000 samples

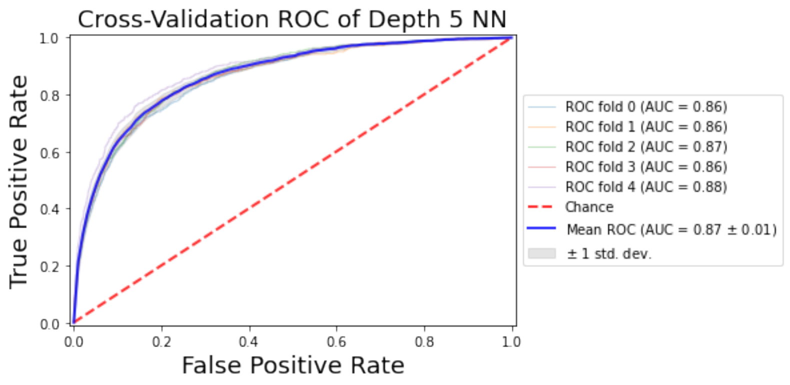


Figure 38: ROC curve for the depth five decoder trained on the random sampled (without replacement) data set with 1,000 samples

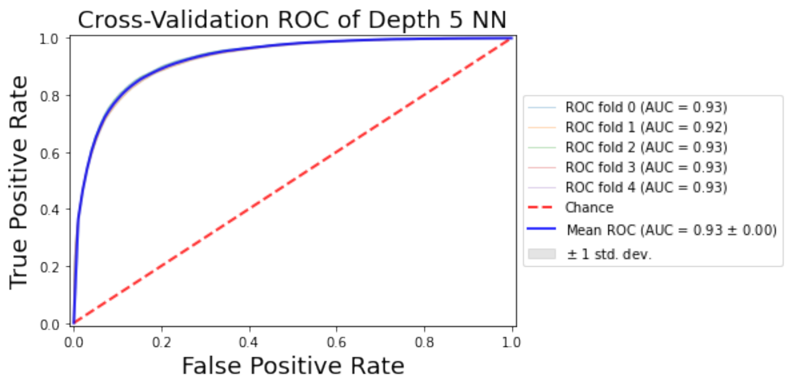


Figure 39: ROC curve for the depth five decoder trained on the random sampled (without replacement) data set with 10,000 samples

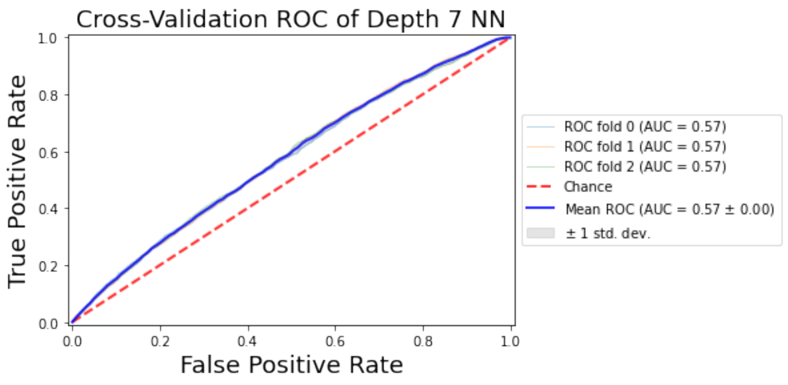


Figure 40: ROC curve for the depth seven decoder trained on the random sampled (with replacement) data set with 1,000 samples

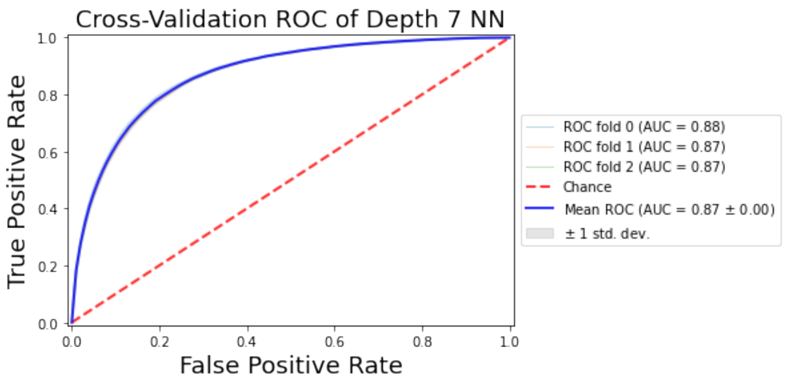


Figure 41: ROC curve for the depth seven decoder trained on the random sampled (with replacement) data set with 10,000 samples

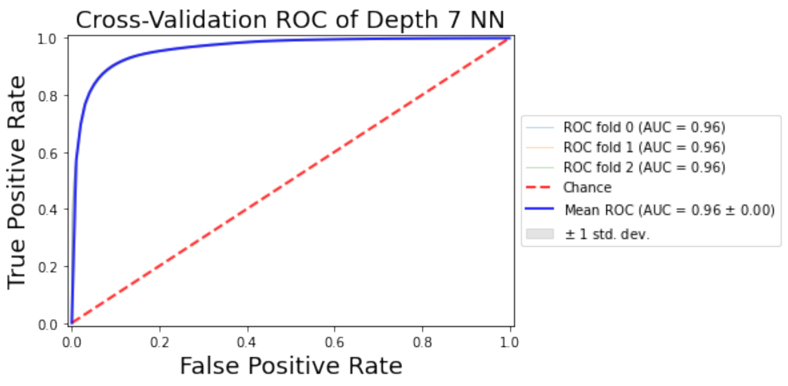


Figure 42: ROC curve for the depth seven decoder trained on the random sampled (with replacement) data set with 100,000 samples

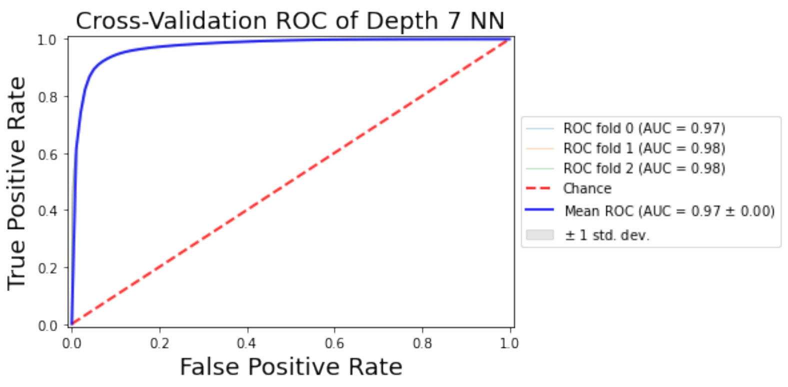


Figure 43: ROC curve for the depth seven decoder trained on the random sampled (with replacement) data set with 500,000 samples

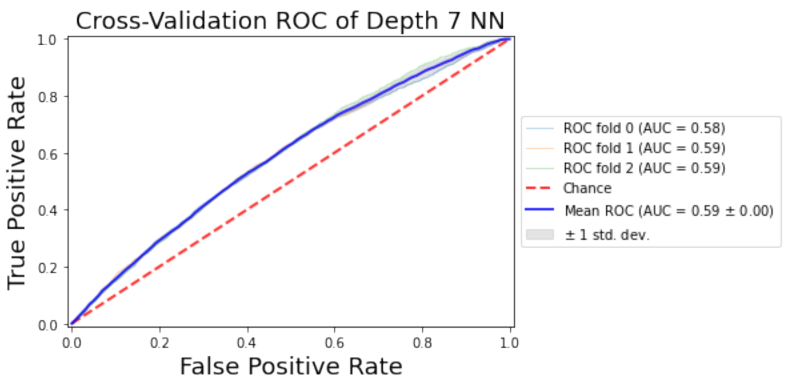


Figure 44: ROC curve for the depth seven decoder trained on the random sampled (without replacement) data set with 1,000 samples

Dataset	**Mean (MWPM)**	**Stdev (MWPM)**
Original	0.142	0.269
Exhaustive	0.507	0.276
Rand (1,000 samples w/ repl)	0.053	0.229
Rand (10,000 samples w/ repl)	0.067	0.231
Rand (100,000 samples w/ repl)	0.054	0.229
Rand (1,000 samples w/o repl)	0.053	0.229
Rand (10,000 samples w/o repl)	0.053	0.229

Table 34: McNemar's Test results between the depth three neural network decoder and the MPWM decoder for all data sets

Dataset	**Mean (PLUT)**	**Stdev (PLUT)**
Original	0.208	0.364
Exhaustive	0.511	0.293
Rand (1,000 samples w/ repl)	0.053	0.229
Rand (10,000 samples w/ repl)	0.056	0.229
Rand (100,000 samples w/ repl)	0.053	0.229
Rand (1,000 samples w/o repl)	0.001	0.003
Rand (10,000 samples w/o repl)	0.000	0.000

Table 35: McNemar's Test results between the depth three neural network decoder and the partial lookup table decoder for all data sets

Dataset	**Mean (MWPM)**	**Stdev (MWPM)**
Original	0.303	0.315
Rand (1,000 samples w/ repl)	0.336	0.356
Rand (10,000 samples w/ repl)	0.186	0.273
Rand (100,000 samples w/ repl)	0.140	0.246
Rand (1,000 samples w/o repl)	0.354	0.353
Rand (10,000 samples w/o repl)	0.271	0.346

Table 36: McNemar's Test results between the depth five neural network decoder and the MPWM decoder for all data sets

Dataset	**Mean (PLUT)**	**Stdev (PLUT)**
Original	0.148	0.262
Rand (1,000 samples w/ repl)	0.493	0.378
Rand (10,000 samples w/ repl)	0.238	0.321
Rand (100,000 samples w/ repl)	0.127	0.287
Rand (1,000 samples w/o repl)	0.450	0.348
Rand (10,000 samples w/o repl)	0.137	0.268

Table 37: McNemar's Test results between the depth five neural network decoder and the partial lookup table decoder for all data sets

Dataset	**Mean (MWPM)**	**Stdev (MWPM)**
Original	0.451	0.387
Rand (1,000 samples w/ repl)	0.353	0.367
Rand (10,000 samples w/ repl)	0.350	0.369
Rand (100,000 samples w/ repl)	0.452	0.354
Rand (500,000 samples w/ repl)	0.484	0.347
Rand (1,000 samples w/o repl)	0.053	0.229

Table 38: McNemar's Test results between the depth seven neural network decoder and the MPWM decoder for all data sets

Dataset	Mean (PLUT)	Stdev (PLUT)
Original	0.251	0.294
Rand (1,000 samples w/ repl)	0.971	0.164
Rand (10,000 samples w/ repl)	0.721	0.377
Rand (100,000 samples w/ repl)	0.345	0.328
Rand (500,000 samples w/ repl)	0.280	0.321
Rand (1,000 samples w/o repl)	0.001	0.003

Table 39: McNemar's Test results between the depth seven neural network decoder and the partial lookup table decoder for all data sets

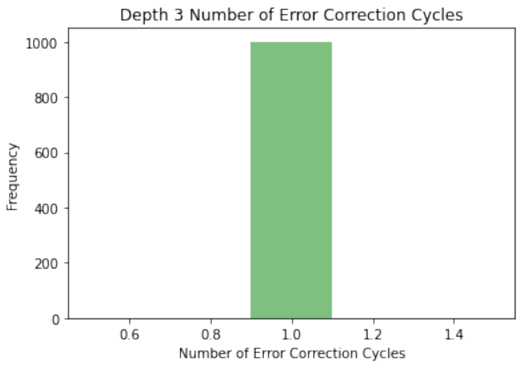


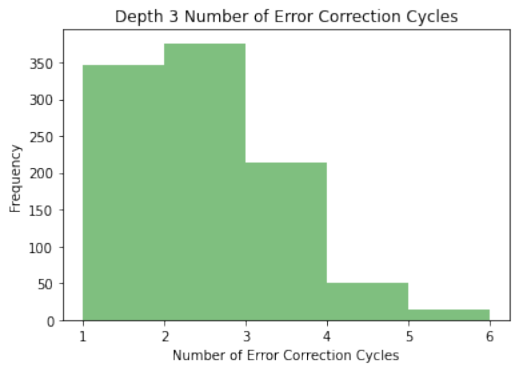
Figure 45: Error correction cycles for depth three neural network trained on exhaustive data set

Dataset	Mean	Stdev
Original	2.89	0.793
Exhaustive	1.000	0.000
Rand (1,000 samples w/ repl)	2.097	0.981
Rand (10,000 samples w/ repl)	2.868	1.130
Rand (100,000 samples w/ repl)	2.668	1.255
Rand (1,000 samples w/o repl)	2.479	0.948
Rand (10,000 samples w/o repl)	1.875	1.022

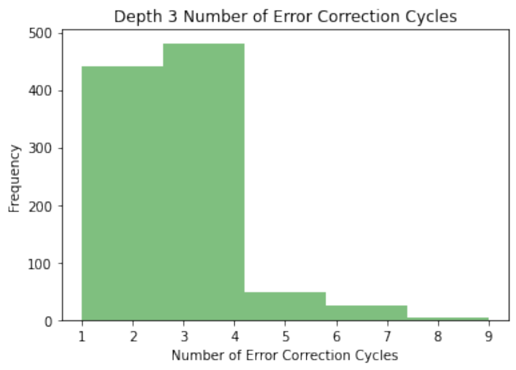
Table 40: Mean error correction cycles for all depth three neural networks

Dataset	Mean	Stdev
Original	6.917	2.981
Rand (1,000 samples w/ repl)	4.648	1.305
Rand (10,000 samples w/ repl)	5.916	2.620
Rand (100,000 samples w/ repl)	6.112	2.419
Rand (1,000 samples w/o repl)	4.524	1.384
Rand (10,000 samples w/o repl)	7.967	3.823

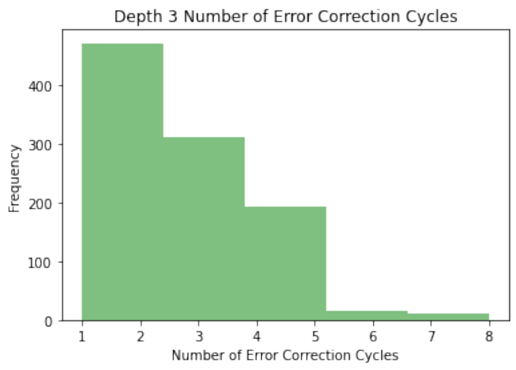
Table 41: Mean error correction cycles for all depth five neural networks



(a) 1,000 samples

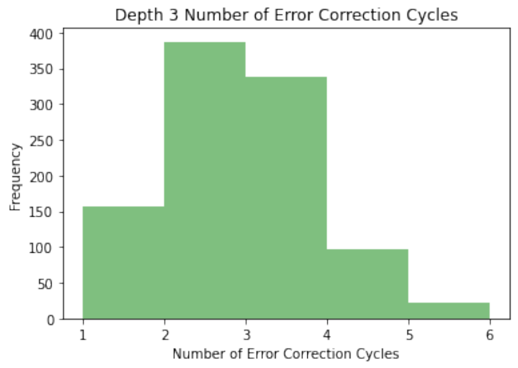


(b) 10,000 samples

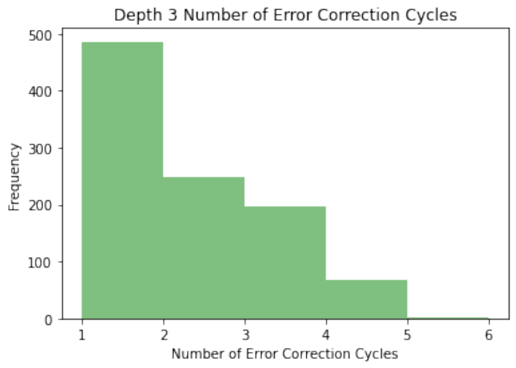


(c) 100,000 samples

Figure 46: Error correction cycles for depth three neural networks trained on random sampled data sets with replacement at various sample sizes

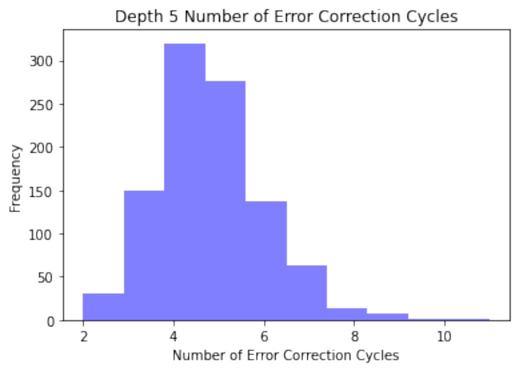


(a) 1,000 samples

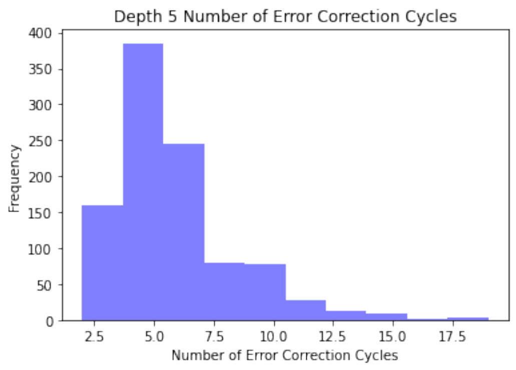


(b) 10,000 samples

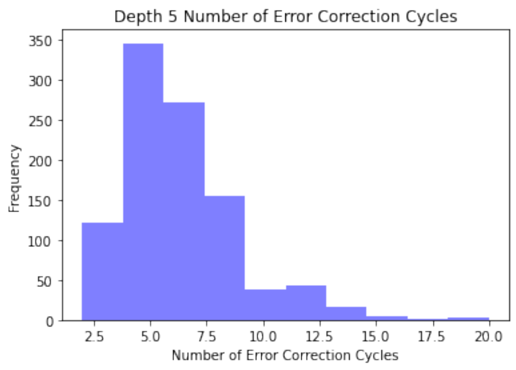
Figure 47: Error correction cycles for depth three neural networks trained on random sampled data sets without replacement at various sample sizes



(a) 1,000 samples

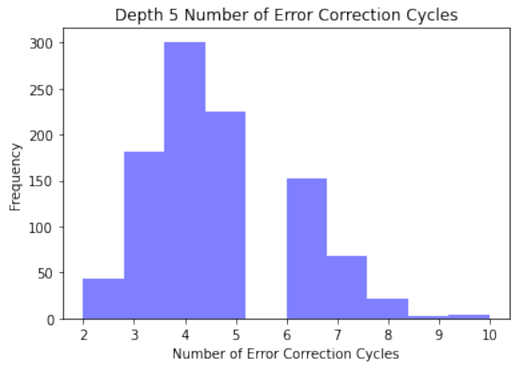


(b) 10,000 samples

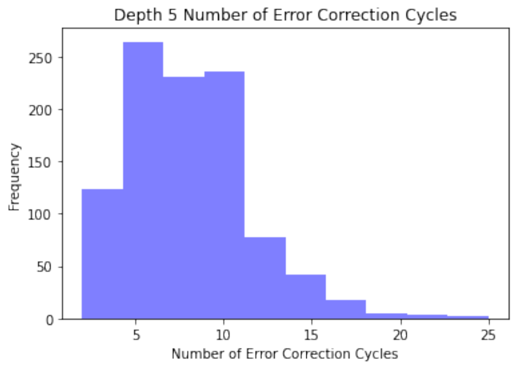


(c) 100,000 samples

Figure 48: Error correction cycles for depth five neural networks trained on random sampled data sets with replacement at various sample sizes

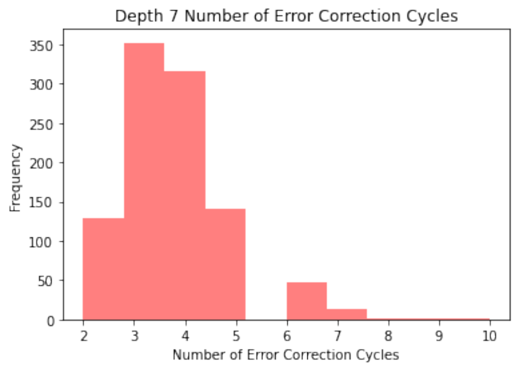


(a) 1,000 samples

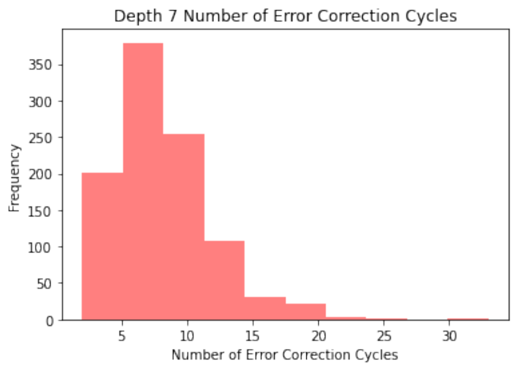


(b) 10,000 samples

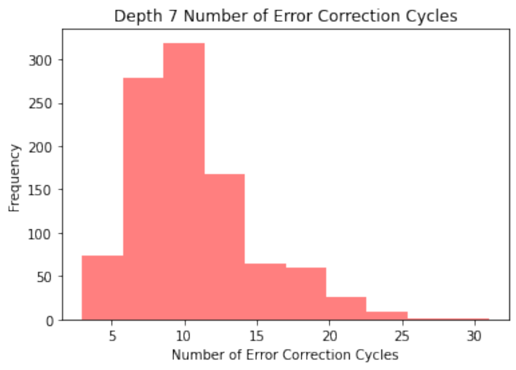
Figure 49: Error correction cycles for depth five neural networks trained on random sampled data sets without replacement at various sample sizes



(a) 10,000 samples



(b) 100,000 samples



(c) 500,000 samples

Figure 50: Error correction cycles for depth seven neural networks trained on random sampled data sets with replacement at various sample sizes

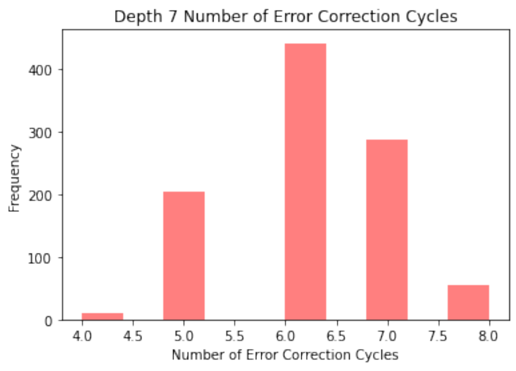


Figure 51: Error correction cycles for depth seven neural network trained on random sampled data set without replacement (1,000 samples)

Dataset	**Mean**	**Stdev**
Original	20.974	10.549
Rand (10,000 samples w/ repl)	3.645	1.186
Rand (100,000 samples w/ repl)	8.862	3.866
Rand (500,000 samples w/ repl)	10.426	4.216
Rand (1,000 samples w/o repl)	6.176	0.898

Table 42: Mean error correction cycles for all depth seven neural networks

Bibliography

1. "Artificial intelligence and autopilot." [Online]. Available: https://www.tesla.com/AI

2. "Machine learning crash course — google developers." [Online]. Available: https://developers.google.com/machine-learning/crash-course

3. "Pytorch." [Online]. Available: https://pytorch.org/

4. "Tensorflow." [Online]. Available: https://www.tensorflow.org/

5. "National quantum coordination office (nqco)," Apr 2021. [Online]. Available: https://www.quantum.gov/

6. S. Almeida, "An introduction to high performance computing," *International Journal of Modern Physics A*, vol. 28, pp. 40 021–, 09 2013.

7. C. E. Badger, "Performance of various low-level decoder for surface codes in the presence of measurement error," Ph.D. dissertation.

8. L. S. Bishop, "Circuit quantum electrodynamics," Ph.D. dissertation, 2010. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1007/1007.3520.pdf

9. D. Castelvecchi, "Quantum-computing pioneer warns of complacency over internet security," Oct 2020. [Online]. Available: https://www.nature.com/articles/d41586-020-03068-9

10. P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J. Prost, M. Snir, B. Traversat, and P. Wong, "Overview of the MPI-IO parallel I/O interface," 04 1995.

11. A. L. Edwards, "Note on the "correction for continuity" in testing the significance of the difference between correlated proportions," *Psychometrika*, vol. 13, no. 3, p. 185–187, 1948.

12. A. Fowler, D. Sank, J. Kelly, R. Barends, and J. Martinis, "Scalable extraction of error models from the output of error detection circuits," 05 2014.

13. X. Fu, L. Riesebos, L. Lao, C. G. Almudever, F. Sebastiano, R. Versluis, E. Charbon, and K. Bertels, "A heterogeneous quantum computer architecture," *Proceedings of the ACM International Conference on Computing Frontiers*, 2016.

14. A. Grama, *Introduction to parallel computing, Second edition.* Addison-Wesley, 2003.

15. L. K. Grover, "Quantum mechanics helps in searching for a needle in a haystack," *Physical Review Letters*, vol. 79, no. 2, p. 325–328, 1997.

16. IBM, "Qiskit 0.23.1 documentation," Accessed on: 10 February, 2021. [Online]. Available: https://qiskit.org/documentation/

17. P. Kaye, R. Laflamme, and M. Mosca, "An introduction to quantum computing," 2006.

18. D. Koch, B. Martin, S. Patel, L. Wessing, and P. M. Alsing, "Demonstrating nisq era challenges in algorithm design on ibm's 20 qubit quantum computer," *AIP Advances*, vol. 10, no. 9, p. 095101, 2020.

19. S. Krastanov and L. Jiang, "Deep neural network probabilistic decoder for stabilizer codes," *Scientific Reports*, vol. 7, no. 1, 2017.

20. F. Limited, "Japan's fugaku retains title as world's fastest supercomputer for fourth consecutive term." [Online]. Available: https://www.fujitsu.com/global/about/resources/news/press-releases/2021/1116-01.html

21. Y.-H. Liu and D. Poulin, "Neural belief-propagation decoders for quantum error-correcting codes," *Physical Review Letters*, vol. 122, no. 20, 2019.

22. M. A. Mannucci and N. S. Yanofsky, *Quantum computing for computer scientists.* Cambridge University Press, 2008.

23. J. Mattis, "Summary of the 2018 national defense strategy of the United States of America," Department of Defense Washington United States, Tech. Rep., 2018.

24. J. Preskill, "Quantum computing in the nisq era and beyond," *Quantum*, vol. 2, p. 79, Aug 2018. [Online]. Available: http://dx.doi.org/10.22331/q-2018-08-06-79

25. R. Sedgewick, *Algorithms In C: Fundamentals, Data Structures, Sorting, Searching, Parts 1-4, 3/E.* Pearson Education, 1998. [Online]. Available: https://books.google.com/books?id=ylAETlep0CwC

26. P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM Journal on Computing*, vol. 26, no. 5, p. 1484–1509, 1997.

27. K. Thapliyal and A. Pathak, "General structures of reversible and quantum gates," 2017.

28. TOP500, "The top500 project." [Online]. Available: https://www.top500.org

29. L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang, "Experimental realization of shor's quantum factoring

algorithm using nuclear magnetic resonance," *Nature*, vol. 414, no. 6866, p. 883–887, Dec 2001. [Online]. Available: http://dx.doi.org/10.1038/414883a

30. 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, 2020.

31. ——, "Comparing neural network based decoders for the surface code," *IEEE Transactions on Computers*, vol. 69, no. 2, p. 300–311, 2020.

32. T. Werner, "A review on ranking problems in statistical learning," 2020.

33. L. Wu, Y. Chen, K. Shen, X. Guo, H. Gao, S. Li, J. Pei, and B. Long, "Graph neural networks for natural language processing: A survey," 2021.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704–0188*

1. REPORT DATE *(DD–MM–YYYY)* 24–03–2022	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Jun 2020 — Mar 2022

4. TITLE AND SUBTITLE Evaluating Neural Network Decoder Performance for Quantum Error Correction Using Various Data Generation Models	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Martin, Brett M, 2d Lt	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, Graduate School of Engineering and Management (AFIT/EN), 2950 Hobson Way, WPAFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENG-MS-22-M-044
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/RITQ, Ms. Laura Wessing, 525 Brooks Road, Rome, NY 13441, COMM: 315-330-2937	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFR/RITQ
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Neural networks have been shown in the past to perform quantum error correction (QEC) decoding with greater accuracy and efficiency than algorithmic decoders. Because the qubits in a quantum computer are volatile and only usable on the order of milliseconds before they decohere, a means of fast quantum error correction is necessary in order to correct data qubit errors within the time budget of a quantum algorithm. Algorithmic decoders are good at resolving errors on logical qubits with only a few data qubits, but are less efficient in systems containing more data qubits. With neural network decoders, practical quantum computation becomes much more realizable since the error corrective operations are calculated much faster than with the MWPM or partial lookup table implementations. This research is aimed at furthering neural network QEC decoder research by generating exhaustive and randomly sampled data sets using high-performance computing algorithms to evaluate the effect of data set generation methods on the effectiveness of these neural networks compared to similar models. The results of this work show that different data sets affect various performance metrics including accuracy, F1 score, area under the receiver operating characteristic curve, and QEC cycles.

15. SUBJECT TERMS

Quantum computing, Quantum error correction, high-performance computing, neural networks

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Laurence D. Merkle, AFIT/ENG
U	U	U	UU	125	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255-6565; laurence.merkle@afit.edu