5-2023

Improved Grover's Implementation of Quantum Binary Neural Networks

Brody A. Wrighter
baw9895@rit.edu

Improved Grover's Implementation of Quantum Binary Neural Networks

Brody A. Wrighter

Improved Grover's Implementation of Quantum Binary Neural Networks

Brody A. Wrighter

May 2023

A Thesis Submitted
in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in
Computer Engineering

R·I·T | Kate Gleason
College of ENGINEERING

Department of Computer Engineering

Improved Grover's Implementation of Quantum Binary Neural Networks

Brody A. Wrighter

Committee Approval:

Dr. Sonia Lopez Alarcon *Advisor* Date
Computer Engineering

Dr. Cory Merkel Date
Computer Engineering

Dr. Nathan Cahill Date
School of Mathematical Sciences

Acknowledgments

I would like to thank my research colleagues for all the help and support that was given during my graduate career. I would like to thank Harrison for general support, help with research computing, and being a good friend. Thank you Tony for research computing help as well. I would like to thank Aden for Quantum Fourier Transform discussions that led me to my Quantum Phase Estimation Simplification Training. I would like to acknowledge Dr. Cory Merkel for all of his knowledge and assistance with machine learning. Finally, I would like to thank Dr. Sonia Lopez Alarcon for her help, support, and being a great advisor throughout the last couple years.

Dedicated to my family: The Wrighters, Mancinis, Ackermans, and Horns

Abstract

Binary Neural Networks (BNNs) are the result of a simplification of network parameters in Artificial Neural Networks (ANNs). The computational complexity of training ANNs increases significantly as the size of the network increases. This complexity can be greatly reduced if the parameters of the network are binarized. Binarization, which is a one bit quantization, can also come with complications including quantization error and information loss.

The implementation of BNNs on quantum hardware could potentially provide a computational advantage over its classical counterpart. This is due to the fact that binarized parameters fit nicely to the nature of quantum hardware. Quantum superposition allows the network to be trained more efficiently, without using back propagation techniques, with the application of Grover's Algorithm for the training process. This thesis presents two BNN designs that utilize only quantum hardware and provides practical implementations for both of them. Looking into their scalability, improvements on the design are proposed to reduce complexity even further.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Artificial Neural Networks (ANNs) are used in Artificial Intelligence (AI) to provide a model for computers to predict outputs based on a set of inputs. ANNs can provide computers with the ability to learn, often called Machine Learning (ML), by using a set of tunable parameters (weights) in the network. The act of adjusting the network parameters comes with growing computational cost as the network or input dataset grows. The storage required for a large number of network paramters can also be large. Both the computation cost and memory of networks can be reduced by utilizing a Binary Neural Network (BNN) to restrict the weights of the network to binary values. While BNNs can reduce the complexity of Neural Networks (NNs), various approaches are being explored to attempt to reduce complexity or computation time further. One such method, explored in this thesis, is the use of Quantum Hardware to implement BNNs. Properties of Quantum Computing (QC) such as superposition can be used to improve the training of BNNs to perform better than its classical counterpart. Development and refinement of BNNs on QC is still required to improve the performance of these networks for use with larger, more complex networks and problems. This thesis explores practical implementations of the binary neural network, utilizing the design proposed in [1]. It will evaluate the scalability of proposed

Quantum Binary Neural Networks (QBNNs) and will improve upon the design of the Networks.

1.2 Objective

The objective of this thesis is to explore current implementations of QBNNs and provide a practical implementation utilizing the two distinct designs found in [1]. While both designs implement the Grover's Search Algorithm, one design uses Phase Estimation while the other uses Register Counting to train the neural network. This work, [1], discusses designs of the Phase Estimation circuit, but it does not give an implementation of the Register Counting circuit. The implementations of the Phase Estimation circuit only cover a small set of neural networks and datasets, the largest network being a three-layer Neural Network with eight weights, three inputs and one ouput. The implementations are made using Huawei's HiQ quantum computing cloud platform and contain errors that may cause confusion or incorrect results if they were to be implemented. The small set of neural networks that [1] tested their circuits on may be caused by the large exponential growth in circuit depth as the neural network gets larger. To combat these shortcomings, an implementation of the Register Counting method will be developed, and a larger neural network will be tested with more datasets and practical implementations. The discrepancies in the design by [1] includes a misrepresentation of weight applications, and a non-standard implementation of the Quantum Phase Estimation algorithm. These issues will be discussed and corrected in this thesis. The thesis explores the scalability of the design when using quantum hardware and provides formulas detailing the circuit scaling given a problem and network size. It will also examine aspects of the Grover's Algorithm Quantum Binary Neural Network (QBNN) and attempt to improve upon its implementation to reduce the number of qubits, depth, or computation time. Other related work will be researched and compared against the algorithms in this

thesis.

1.3 Thesis Contribution

Machine Learning is a research topic that is gaining more and more contributions within various research fields. Not only can machine learning be used to improve upon artificial intelligence, it can even be used to assist researchers in non-engineering, or non-artificial intelligence related fields, such as biology or more specifically neuroanatomy. Within all of the various research areas of machine learning, a prominent topic of discussion is the training of neural networks. Improving the training speed or efficiency of neural networks allows the machine learning models to produce more accurate results after a much smaller period of time. This thesis tackles binary neural network training efficiency by utilizing the properties of Quantum Computing and Grover's Algorithm. The contribution of this thesis to machine learning is:

1. **We have fixed a couple of errors present in a training circuit design (QPE) in [1] to provide correct results with fewer qubits.**

2. **The fixed training circuit design in [1] was improved upon to reduce the depth of the circuit greatly (QPE Simplified).**

3. **A Register Counting training circuit was proposed in [1] but not implemented. This thesis implements that circuit.**

4. **The three circuits above are tested using a basic three input, one output neuron to ensure accuracy and correct outputs.**

5. **The three circuits are also tested using a 2x2 and 3x3 pixel edge detection model as the circuit improvements allow for larger tests to be conducted.**

6. **The scalability of all three circuits are calculated in detail to show the feasibility of the circuits with larger problems.**

All of the above are explained in detail in Chapter 3. These contributions will allow for faster integration of quantum computing into the machine learning field.

Chapter 2

Background and Related Work

2.1 Artificial Neural Networks

Artificial Neural Networks are networks made of a collection of elementary nodes called neurons. These neurons can have any number of inputs and an output that is generated based on the input(s). Each input of a neuron has an associated weight that is multiplied by the input. The resulting values from each weighted input are summed together. This process can be seen in the sum of products equation

$$s = b + \sum_{n=1}^{N} a_n * w_n \tag{2.1}$$

where a_n is an input, w_n represents the weight applied to the input, and b is some bias value added to the sum of products that also acts as a tunable parameter. The variable a_n corresponds to an input of the neuron with index n, and the variable w_n is its corresponding weight. The resulting value, s, is then passed into an activation function that determines the output of the neuron. This function can be realized by multitude of different functions. Some functions include the sigmoid function, or ReLU activation function. The full neuron model can be seen in Figure 2.1.

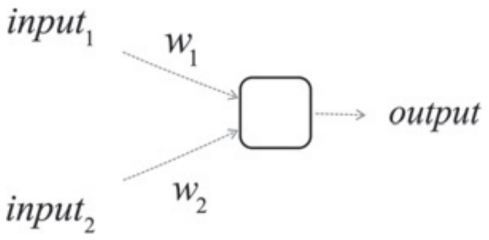


Figure 2.1: Two input neuron. [1]

Assembling these neurons together results in a neural network. This network can come in many shapes and sizes. The most common type of network that is used in Machine Learning (ML) is a feed-forward network and is a collection of neurons organized into multiple layers. The outputs of one layer serves as the input of the next layer. Even though a neuron only has one output, the output can be used by any number of other neurons as their inputs. The first layer of a NN serves as the input layer that takes in the initial inputs and the last layer results in the output of the full network. An example network can be seen in Figure 2.2.

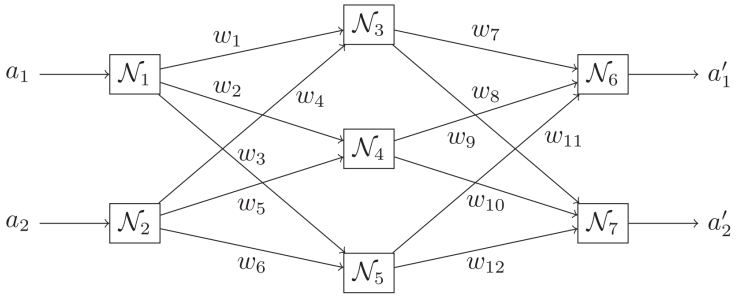


Figure 2.2: 2-3-2 neural network example. This network is a fully connected neural network since any particular node is connected to all nodes in the next layer. [1]

Binary Neural Networks (BNNs) are a subset of Artificial Neural Networks that binarize its network parameters. The parameters of a binary neural network are commonly constrained to either 0 and 1, or -1 and 1. The activation functions of BNNs are commonly threshold functions. This means that if the resulting sum of products is less than a threshold, then the neurons outputs a zero; otherwise the

output of the neurons is a one.

Training the network is the process that determines the weight values of the NN, and is the computationally intensive part of ANNs. This usually requires a large set of data containing inputs and expected outputs. The network works through the dataset, attempting to tune the network parameters to result in the highest possible accuracy compared to the data-set. In ANNs it is common to use back propagation to calculate the weights of the network. One way to perform back propagation is to compute gradient descent which can be calculated from the derivative of the cost function. The cost function calculates an error between the output of the NN and the expected output given by the dataset by utilizing the derivative of the activation function of the neurons. Propagating backwards and finding the derivative of the cost function is the computationally intensive part of training ANNs. Due to the use of threshold activation functions within BNNs, the derivative is often non-derivable and does not allow for back propagation to be used for training [5]. This is commonly fixed by utilizing a real-valued gradients of the weights to perform the training of the neural network [5]. The full process of training the BNN can be generalized by first completing a forward propagation of the NN with binarized weights, comparing the expected output to the calculated output of the NN, and then calculating the real-value gradient and updating the real-value weights. This process is done for each data point in the training dataset [5].

A common issue that arises when using the gradient descent for training is that gradient descent is designed to find the local minima of the hyperparameters. Gradient descent moves the hyperparameters toward values that generate less output error. However, if this set of hyperparameters is the lowest point locally, a more optimal set of hyperparameters will not be found using this method. A common way to find this globally optimal set of weights is to do a brute force search that tests the BNN with all $d \times N$ possibilities, where d is the number of training dataset entries, and N is the

number of possible weight strings where $N = 2^n$ and n is the number of weights.

2.2 Quantum Computing

Quantum Computing (QC) utilizes the nature of quantum physics to process and compute data. Utilizing quantum physics allows developers to perform computations of certain applications faster than classical computing. Currently, quantum computers are still in their early stages. Research is actively being done on many aspects of QC and new contributions are being generated rapidly.

One of the most important concepts that make up a quantum computer is the qubit. The qubit is used to encode information, similar to how bits are used to encode information in classical hardware. Qubits can only be measured into a binary set of states, 1 and 0. However, before measurement qubits can have a probability of being in a certain quantum state, as well as contain other quantum mechanical information such as global phase [6]. This phenomenon of quantum mechanics can be taken advantage of to process the data in new and useful ways.

The ability of qubits to be in more than one state at once with a certain probability of being in each state is known as superposition. Superposition of qubits is a very useful feature of quantum computing that allows for algorithms and computations which are not possible in classical computing. When a qubit or string of qubits are in superposition, any gate or algorithm that uses the qubits will act on all the possible states of the qubit. This feature will be used extensively in this research [6].

Multiple qubits can also become entangled with one another. Entangled qubits have a unique property wherein acting on one entangled qubit impacts the quantum state of other entangled qubits, even if the other qubits are not acted upon. This can occur even if the entangled qubits are far from one another. The quantum state of entangled qubits cannot be separated from one another, meaning that observing one of the qubits collapses the other qubit instantaneously (as the state of one qubit

means that the other must be in a certain state) [6].

2.2.1 Quantum Phase Estimation

Quantum Phase Estimation (QPE) [7] is an algorithm widely used in quantum computing. The main functionality of the QPE algorithm is to estimate the eigenvalue of a unitary operator U, where $U|\Psi\rangle = e^{2\pi i\Theta}|\Psi\rangle$ [7]. The controlled unitary operator is applied to a set of qubits called the counting register. It utilizes phase kickback that will rotate the state of the controlled bit of the unitary while keeping the target qubits unaffected. The kickback can be used for 2^t times, where $t = 0, 1, 2, ..., n$ is the index of the counting register and n is the number of counting registers. A general circuit design of the QPE algorithm is shown in Figure 2.3.

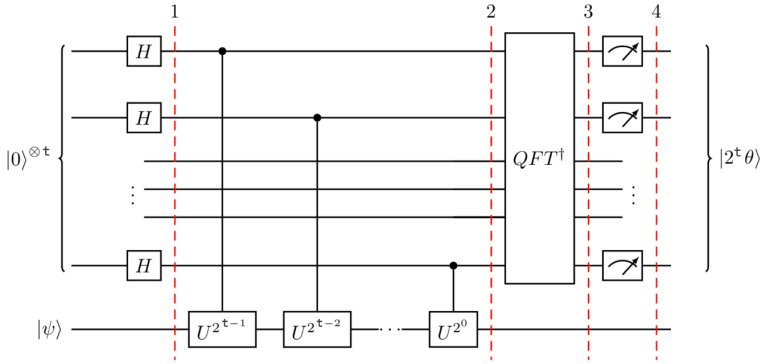


Figure 2.3: Quantum phase estimation circuit. The controlled unitary that applies an phase is represented as 'U' and is repeated a power of two times, dependent on which qubit acts as the control. [2]

Once the controlled unitaries are used on each counting register qubit, the phase of the unitary is encoded onto the counting registers in the Fourier domain. To convert from the Fourier domain to the computational basis in classical computing, an inverse Fourier transform is used. The same method is followed when working in the quantum computing realm. Here an inverse Quantum Fourier Transform (QFT) [8] is used to convert the phase to the computational basis. The phase can then be read as x by measuring the qubits after the inverse QFT. The binary output x can then be used to

find $\Theta = \frac{x}{2^n}$ where n is the number of counting registers [2]. It is important to note that as the number of counting registers are increased, the precision of Θ is increased.

The application of the QPE algorithm could be used for a variety of different problems and algorithms in quantum computing. Some applications include logical quantum arithemetic [9] and even the notorious Shor's algorithm [10] which can be used to find the prime factors of an integer and potentially break RSN encryption.

2.2.2 Grover's Algorithm

Grover's Algorithm [11] is a fundamental algorithm used in quantum computing. It is utilized in many diverse problems that require a search through a database and can identify a target element faster than classical computing search algorithms. Grover's Algorithm is able to achieve this speedup by utilizing quantum superposition of the database elements which allows the algorithm to search through the elements all at once. The algorithm is realized in two main parts: the Grover's oracle and the Grover's diffuser.

The task of the Grover's oracle is to perform the "search" of the database. The oracle is a quantum circuit that identifies the target element in a given "database", or more commonly, in the set of all possible states of the quantum system, placed in superposition. Once identified, the oracle applies a phase inversion onto the specific element, inverting its amplitude. This results in a superposition of all elements where the target elements have a negative magnitude and all others elements have a positive magnitude. An example of the oracle acting on the superposition of elements is shown in Figure 2.4.

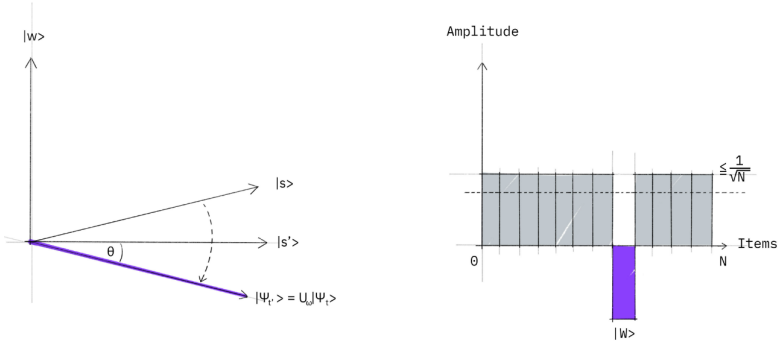


Figure 2.4: Grover's oracle vector and amplitude representation. [3]

In Grover's algorithm, the oracle is the most important component that defines the search. It can be designed to solve a number of problems related to data base searches, including some NP hard problems and optimization problems where the data base target is the solution to minimize a cost function. The definition and the implementation of a Grover's oracle is the most important part of the algorithm as well as the biggest challenge when implementing the algorithm.

The Grover's diffuser then inverts the amplitude of all elements about the mean of the amplitudes. This results in the target elements amplitude to increase while the non-target elements amplitudes decrease. This can be seen in Figure 2.5

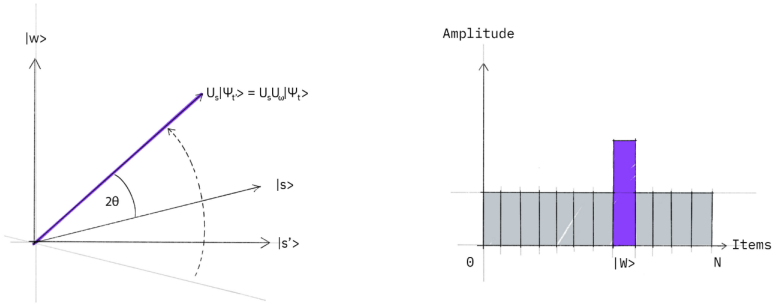


Figure 2.5: Grover's diffuser vector and amplitude representation. [3]

Combining the oracle and diffuser results in a single Grover's Algorithm cycle. The complete circuit is repeated in order to achieve an amplitude amplification in the target elements of at least 50%.

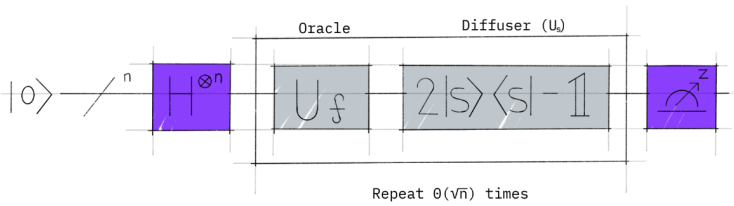


Figure 2.6: Complete Grover's algorithm circuit. The oracle and diffuser pair is to be repeated $\sqrt{N/M}$ times where N is the number of elements and M is the number of targets in the solution. [3]

The complete Grover's algorithm is seen in Figure 2.6. It can be seen that a set of Hadammard gates are placed on each qubit to put all of the qubits in superposition. The Grover's oracle and diffuser are repeated until one or more outcomes have a probability greater than 50% and are considered detected as the target outcome. The Grover's cycle should be repeated around $\sqrt{N/M}$ times where N is the number of elements and M is the number of target elements in the solution [11].

Grover's Algorithm [11] is also used in a wide array of quantum applications. An example of a problem includes solving the maximum clique problem [12][13]. This algorithm is able to correctly identify the maximum fully connected sub-graph (clique) of a larger graph. In this [12] solution, the Grover's oracle flips a target qubit if the state represents a subgraph that is larger than the minimum acceptable number of vertices for the clique. The diffuser can then identify cliques over the minimum acceptable size. Another use of Grover's algoritm is the solving of boolean satisfiability problems [14][15]. This determines if a set of binary clauses are satisfiable. The satifiability problem has been seen to be NP-complete [16]. It has also been observed that Grover's Algorithm provides speedup for NP-complete problems [17], which can allow quantum computing to speed up this algorithm.

2.3 Quantum Machine Learning and Related Work

Quantum Machine Learning is the use of quantum computers and quantum physics to attempt to improve various aspects of machine learning. This research area has been an attraction for a number of academic contributions that attempt to improve speedup, training, or simply implement a full neural network in quantum hardware [18]. The implementations of neural networks on quantum hardware either aim to model biological processes, or attempt to emulate classical algorithms in quantum hardware [18]. While some designs and models work better than others, this field is still in its infancy and more research is needed to integrate quantum machine learning into practice.

2.3.1 Quantum-Classical Hybrid Machine Learning

While there are many different approaches to improve machine learning, something that is often researched is the use of quantum computing in tandem with classical computing, to improve upon already existing classical machine learning algorithms. Due to the nature of quantum computing and quantum mechanics, some shortcomings of strictly classical implementations can be improved with the integration of quantum computation. An example of this utilizes quantum computing to speedup data storage and assignment in quantum machine learning applications [19]. Here, the use of quantum memory allows for the assignment of N-dimensional vectors to one of several clusters of M states [19]. It is even possible to reduce the storage space of N-dimensional complex vectors to $log_2 N$ qubits using qRAM [19].

In addition to improvements of the limitations found in classical machine learning applications, some algorithms exist that allow for the use of quantum computing to simplify the computation required for classical algorithms. An example of this is the acceleration of single-layer binary neural networks using the Harrow-Hassidim-Lloyd

(HHL) algorithm [20]. The HHL algorithm is used to provide a solution to the linear regression problem in training NNs and can be used to reduce the complexity on the classical side. This algorithm reduces the hyperspace of all possible weights to a surface of a hyper-sphere whose radius is defined by the SWAP test of a test state to a reference state [20]. This gives an advantage to the classical computation of the optimal weights, which resulted in a lower number of iterations required for the constrained/assisted training.

2.3.2 Quantum Neural Networks

Another method of introducing quantum computing into the field of machine learning is the attempt to implement the network models within the quantum circuitry itself. Some advantages can be seen by keeping information strictly in the quantum space rather than switching between both the quantum space and classical space. This includes the ability to keep qubits in superposition and in some cases allow qubits to remain entangled with one another. An example of a quantum neural network implemented in quantum circuitry is designed by [21]. This implementation utilizes a custom unitary gate that is applied to the input qubits. The output of the gate is the resulting output of the network. The weights are encoded into the matrix representation of the gate.

$$U_{nand} = \begin{bmatrix} \frac{1}{\sqrt{6}} & 0 & \frac{1}{\sqrt{6}} & \frac{-2}{\sqrt{6}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{3}} & 0 \\ \frac{1}{3\sqrt{2}} & \frac{2}{\sqrt{3}} & \frac{1}{\sqrt{2}} & \frac{\sqrt{2}}{3} \\ \frac{2}{3} & -\frac{2}{3} & 0 & \frac{2}{6} \end{bmatrix} \tag{2.2}$$

Equation 2.2 shows an example of a unitary function used in [21] to represent the application of weights onto a two qubit input state. After the application of the unitary, a gate $D(m, \delta)$ is used to essentially serve as the activation function of the

neuron.

2.4 Quantum Binary Neural Networks

Quantum Binary Neural Networks (QBNNs) are a realization of BNNs in Quantum Computing. QBNNs can use a mix of classical computing and quantum computing, or can use solely quantum computing. In general, the goal of utilizing quantum hardware in BNNs is to provide either raw speedup or information that will reduce the problem complexity in the classical space. Implementing the QBNN using only quantum hardware allows the network to maintain its information in the quantum space. One such implementation is explored in this thesis [1]. This article introduces a quantum circuit and algorithm that implements and trains a QBNN on strictly quantum hardware.

The paper details two possible implementations of the QBNN, both of which implement Grover's Algorithm to detect and amplify the ideal set of weights that should be used. Both implementations work by essentially counting the number of times that each possible set of network weight strings give the correct output compared to a set of training data. This allows the quantum training algorithm to find the globally optimal set of weights for a given neural network and dataset. The first of the implementations use Quantum Phase Estimation(QPE) to measure an accumulation of phase on weight strings that provide correct network outputs compared to the dataset. The second implementation increases a register of qubits each time a set of weights gives the correct output. Both implementations provide speedup compared to a classical implementation, as the quantum circuit is able to observe all possible combinations of weight strings at once by placing them in superposition [1].

2.4.1 Quantum Binary Neural Network Circuit Implementation

The first step in implementing the QBNN in quantum hardware is creating the network itself. Observing the classical implementation of a BNN, the weights are multiplied to their corresponding inputs. However, in QBNNs, the inputs and weights implement a XOR operation between each weight-input pair. This is done because the behavior relates directly to the quantum Controlled NOT (CNOT) gate. The CNOT gate utilizes one control qubit and one target qubit. If the control qubit is one, then a NOT gate is performed on the target qubit, otherwise, if the control qubit is zero, then the target qubit is kept the same. The next step in implementing the QBNN is to sum the weight-input pairs and apply the activation function threshold. This can be done by implementing a unitary function. The unitary will be able to observe the output of the weight-input pairs and apply a threshold as an activation function [1]. An example of the implementation of the QBNN structure is shown in Figure 2.7.

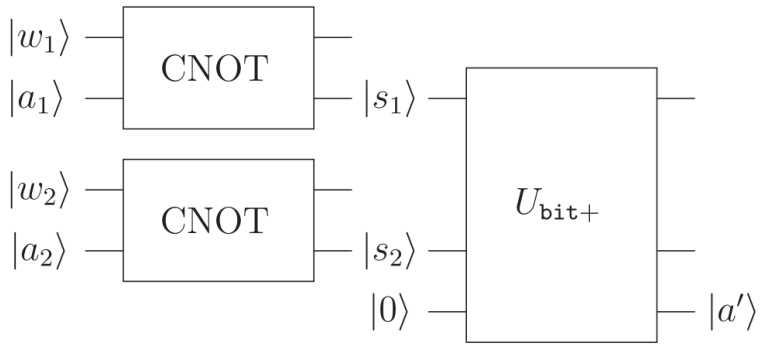


Figure 2.7: QBNN structure. Weight applications onto inputs are represented by a CNOT gate that leaves the result s, on the input qubit a. A circuit is then used to act as an activation function and generate an output onto a new qubit. This figure is an example of a single neuron with two inputs, two weights, and one output. [1]

The structure shown in Figure 2.7 shows an example of a single neuron implemented as a quantum circuit. The neuron has two inputs, two corresponding weights, and one output. The output is determined based on the unitary activation function

that implements a threshold. If the number of weight-input pairs, s, is greater than or equal to the threshold, the function will set the output qubit to state $|1\rangle$, otherwise it will remain at $|0\rangle$. The output of this neuron can then be used as an input to other neurons and a full network can be composed of any size or shape. Utilizing a unitary function and unitary gates during the creation of the BNN is required as it allows the circuit to be uncomputed without disrupting the quantum superposition or entanglement of qubits. This is useful for repeating this circuit multiple times on different data inputs, which is vital for training BNNs [1].

2.4.2 Weight String Phase Accumulation

Once the structure for the QBNN is set up, the paper details the methods of training the network. The first method given is the Phase Estimation training method. In order to implement this algorithm, a Phase Accumulation cycle is introduced that uses the QBNN circuit shown in Figure 2.7. The procedure can be realized in the following steps:

1. Set all of the weight qubits into superposition, allowing them to be in all states at once.

2. Set the input and expected output qubits to match the corresponding values in the first entry of the training dataset.

3. Execute the QBNN circuit.

4. Use a phase accumulation circuit to increment the phase (by $\frac{\pi}{n}$, where n is the number of cases in the training datset) of the weight strings that produce an output equal to the expected output.

5. uncompute the QBNN by implementing the inverse of the circuit in step 3.

6. repeat from step 2, with the next entry of the training dataset until all dataset entries have been executed.

As explained in the steps shown above, the weight inputs are placed into superposition, which allows all possible combinations of weights to be used. This allows for every possible weight string to be tested all at once. The QBNN is then given inputs based on the training data-set and the QBNN is executed. This is represents steps 1-3 above and is implemented in Figure 2.8 below.

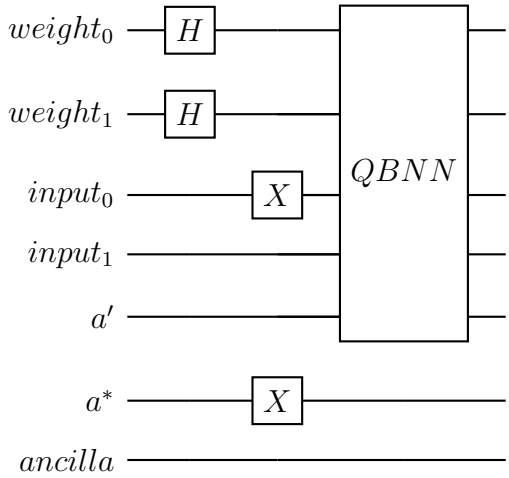


Figure 2.8: Steps 1-3 are shown in the circuit. The weights are first placed into superposition to allow the testing of all possible weight strings at once. The first entry in the training dataset is applied using X gates. In this case, the testing dataset entry used is $|1\rangle$ for $intput_0$, $|0\rangle$ for $intput_0$, and $|1\rangle$ for a^*, the expected output. The QBNN circuit is then executed.

The resulting output of the network, placed on the a' qubit, is then in a superposition corresponding to the output for each corresponding weight string. This means that the execution of the QBNN has 2^N possible weight strings, where N is the number of weights in the network, and 2^N outputs that correspond to each weight string.

The accumulation phase then implements an accumulation oracle that is used to compare each output for the given weight string to the desired output, a^*, given by the dataset. The oracle compares the output with the expected output and adds a

phase to the output of the network if they are equal to each other. The phase that is added to the output by the oracle is a phase of $\frac{\pi}{n}$ where n is the number of data entries in the training data-set.

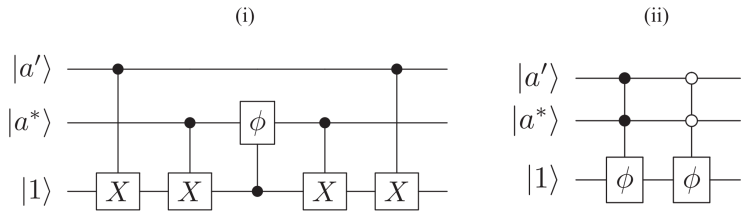


Figure 2.9: Accumulation oracle circuit. Applies a phase of ϕ to the weight string if a correct output is produced. a' represents the output from the QBNN and a^* represents the expected output from the training dataset. [1]

The accumulation oracle implemented in the accumulation cycle can be realized in multiple ways. The methods introduced by the paper uses an ancilla qubit that is set to a state of 1. The first method (i) changes the ancilla bit if the network output (a') and desired output (a^*) from the dataset are different from each other. That ancilla bit is then used as a control bit on a quantum controlled phase gate, which adds the phase of $\frac{\pi}{n}$ to the output [1]. The second design (ii), shown in Figure 2.9, only utilizes two doubly-controlled phase gates that have control bits of either both one or both zero [1]. If the output and expected output are the same, the phase gate is applied to the ancilla bit. The accumulation oracle will be represented as Λ for future circuits.

After the accumulation oracle is executed, the QBNN is un-computed. This allows the added phase to be decoupled from the output and applied to the weight string that gave the successful output [1]. The dataset entry is also undone in order to revert the inputs and expected output qubits to state $|0\rangle$. This circuit is shown in Figure 2.10 below.

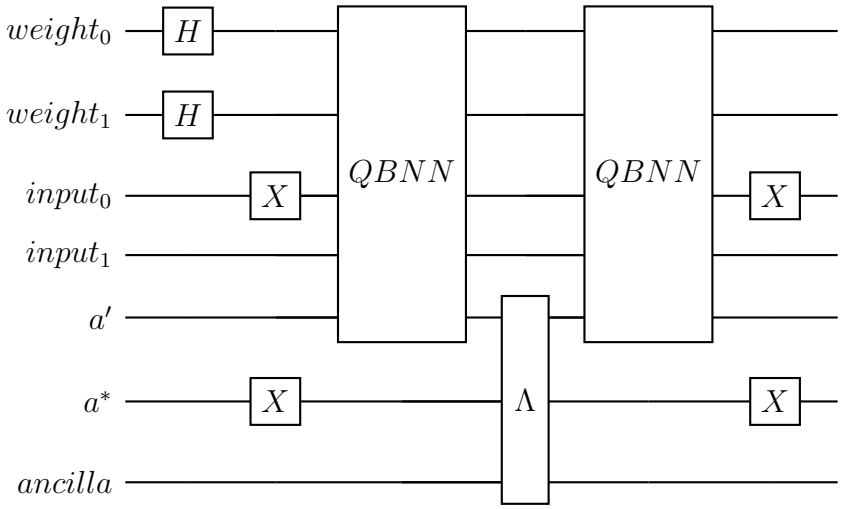


Figure 2.10: Accumulation cycle circuit. The circuit seen in Figure 2.7 is executed once and then uncomputed, with an accumulation oracle in between. The un-computation of the QBNN reverts the qubits to their original state, with a new phase applied to a correct output producing weight string.

The accumulation cycle, without the Hadamard gates, is then repeated n times for each entry in the training dataset. This will continue to add phases to the weight strings that result in accurate output compared to the training set. The results after repeating the circuit for each data point will give a set of weight strings in superposition with varying phase accumulations. The best weight string that gives the most accurate results will have the highest accumulated phase, with a maximum phase of $\frac{n\pi}{n}$. An example of the full phase accumulation is shown in Figure 2.11.

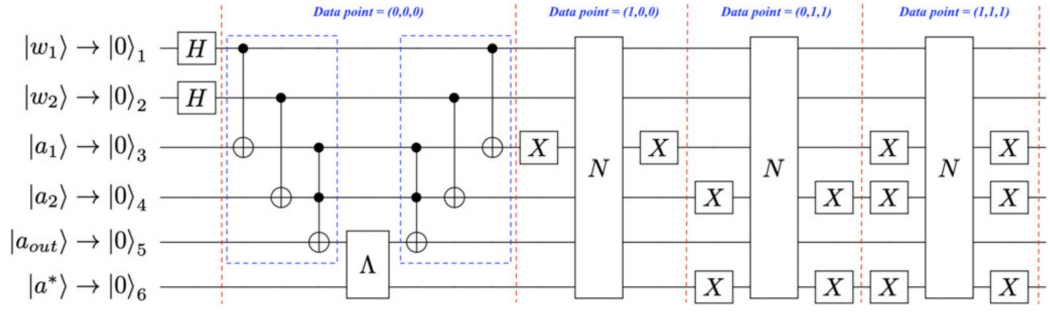


Figure 2.11: Full accumulation cycle. The QBNN circuit shown in Figure 2.10 is repeated for each data entry in the training dataset. The blue boxes in this figure represents the QBNN in this case. NOTE: a_{out} is equivalent to a', and the ancilla qubit used for the accumulation oracle is still used but not represented in the circuit. [1]

Figure 2.11 implements an example of four different training dataset entries. These entries are arbitrarily chosen as an example and can be changed.

2.4.3 Quantum Phase Estimation Training

Now that the accumulation phase is complete, each weight string contains a phase that represents the number of times they provided a correct output. The Quantum Phase Estimation algorithm must be used in order to read and utilize the phases. This is done by converting the complete circuit shown in Figure 2.11 to a controlled unitary gate. This controlled unitary can then implement phase kickback, as explained in Section 2.2.1, to encode a multiple of the phase onto a register of qubits. This then allows the use of the inverse QFT to be applied to the register, and the estimated phase will be encoded on the register as a binary representation.

An accuracy threshold is then called that reads the binary representation of each weight string, and flips the weight strings that are above a pre-defined phase threshold. This threshold is implemented using a multi-controlled CNOT gate. Finally the phase estimation circuit is un-computed to revert all the states to their original states, and a Grover's diffuser is applied to the set of weight strings, amplifying the weight strings that give the most accurate outputs [1]. The full training algorithm is shown in Figure 2.12.

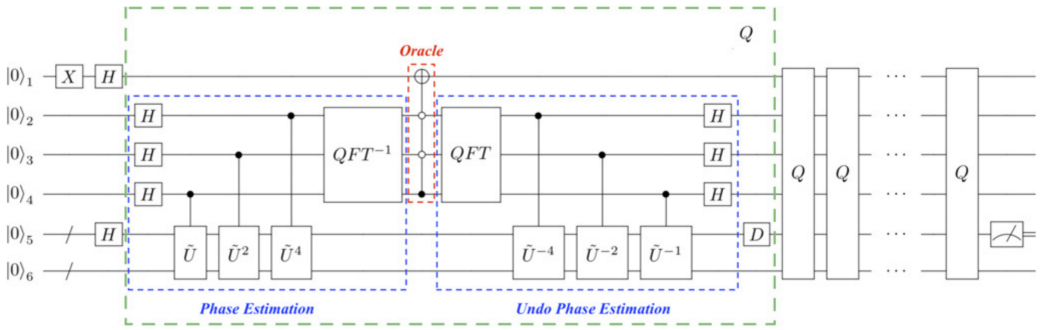


Figure 2.12: Full training cycle. The full accumulation cycle shown in Figure 2.11 is used as the control unitary in the Quantum Phase Estimation Circuit. After an inverse QFT application, the phase of each weight string will be able to be read as a binary representation. An accuracy threshold is used to determine weight strings that perform above a certain specified threshold. The phase estimation is then un-done to revert the circuit to the original states. Finally, a Grover's diffuser is used to amplify the targeted weight strings. [1]

In Figure 2.12, the green box represents a single cycle of Grover's Algorithm, labeled as Q, and the blue box represents the Phase Estimation circuit that implements a controlled unitary of the circuit found in Figure 2.11. Qubit one is set to the $|-\rangle$ state which will be used in the accuracy threshold to invert the target weight strings. Qubits two, three, and four in this example are used as the Phase Estimation registers. This means that the phase can be estimated up to $\frac{k\pi}{n}$, where m is the number of register qubits, and where $k = 2^{m-1}$. Qubit five represents all weight qubits used in the QBNN that will be placed in superposition. Lastly, qubit six represents a string of all other qubits needed to compute the network. This includes the inputs, outputs and any other ancilla bits needed for execution.

The accuracy threshold in Figure 2.12 uses a multi-controlled NOT gate that inverts the weight string if the binary representation of its phase is equal to $\frac{4\pi}{n}$. The control qubits of the circuit can be altered to implement various desired thresholds. For example, if a threshold of $> \frac{6\pi}{n}$ is desired, two multi-CNOT gates should be added with a (1,1,0), and (1,1,1) control scheme, meaning that accuracy threshold would mark weight strings that are either equal to or above $\frac{6\pi}{n}$ (max of $\frac{7\pi}{n}$ for three register

qubits). The value of the threshold, and the multi-controlled CNOTs that define it can be adjusted to fit the needs and accuracy of the network training and the desired weight string accuracy [1].

Finally, the Phase Estimation is un-computed in the second blue box which decouples the accuracy threshold inversion from the Phase Estimation registers and applies it to the weight string. A Grover's diffuser is then called, which is denoted by the D gate and inverts all states about the mean of the magnitudes of the states. This amplifies the most accurate weight strings that are equal to or above the provided accuracy threshold. This process is repeated for the amount of times needed for Grover's Algorithm and when measured, should have the highest probability of the register corresponding to the best weight strings that should be used for the network [1].

2.4.4 Register Counting Training

The next implementation of the QBNN training is the Register Counting implementation. This implementation acts almost identically to the Phase Estimation implementation, but does not use any Quantum Phase Estimation for its solution. The Phase Estimation register is replaced with a counting register that is encoded in binary, similar to the way that the phases were read by the accuracy threshold circuit. When performing the phase accumulation stage, the algorithm no longer accumulates a phase onto the weight strings. It now accumulates a counter in the register counter for each weight string. This means that there are $\lceil log_2(n+1) \rceil$ register qubits needed, where n is the number of data points in the training dataset. The binary version of the number of times a particular weight string results in a correct network output can then be read directly from the register counting qubits, and the accuracy threshold can apply an inversion based on that value. The use of the Grover's Algorithm remains the same for the rest of the implementation [1].

2.4.5 Binary Search Method for Accuracy Threshold

For the QBNN training implementations, a problem arises when the number of good weight strings are unknown for the given problem. A solution to this was given by [1] and implements a binary search on the accuracy threshold to identify the optimal weight strings. The solution requires a desired precision δ, that denotes the stopping point of the algorithm. When the number of possible correct training data matches of a given weight string is less than δ, the binary search is able to find the weight string within the precision δ of the globally optimal weight [1]. The range of possible dataset matches scales with $n2^{-i}$ where i is the i-th step of the binary search. The binary search then takes $\lceil log\frac{n}{\delta} \rceil$ steps to complete. This is not used in this thesis.

2.4.6 QBNN Speedup

The speedup of this method over the classical search for the globally optimal weight string was observed in [1] as well. Using the number of QBNN circuit calls in the quantum training with binary search, versus the number of BNN calls in classical globally optimal training, the following speedup was found:

$$\frac{N_C^{cl}}{N_C^{qm}} > \frac{2^{N/2}}{4N\ log(\frac{N}{\delta})} \tag{2.3}$$

This equation was given in [1] where N is the number of possible weight combinations. N_C^{cl} and N_C^{qm} are the number of BNN calls for the classical and quantum versions respectively.

Chapter 3

Contribution

3.1 Overview

This section details all the main contributions that this thesis brings to the field. Using the information and design described in [1], multiple Quantum Binary Neural Network approaches were designed using the Qiskit library [22]. First, a fixed version of the QPE training algorithm proposed by [1] was designed. A Register Counting circuit that was discussed, but gave no implementation, was implemented in this thesis. Finally, an Improved QPE Training circuit was implemented. This section also details all of the additional work done in this thesis including scalability equations, various set-ups, tests, and metrics that are used to examine various properties of the designs. The tests include a small three input, one output neuron to test functionality and verify accuracy of the new designs, as well as a larger, more practical test that can detect vertical edges in 2x2 pixel matrices. While this section details the design and setup of the thesis contributions, the next chapter, Chapter 4, details the results from all of the contributions.

3.2 Quantum Phase Estimation Binary Neural Network Errors and Fixes

The first Quantum Binary Neural Network (QBNN) implemented, designed by [1], utilizes Quantum Phase Estimation to accumulate the number of successes for each given weight string. This thesis details a couple of fixes of the design. This implementation was written in Python using Qiskit library [22] and IBM Quantum Lab [23] and generally follows the design that was detailed in Section 2.4. Look to Section 2.4 for more detail of this implementation. The implementation of the network itself is dependent on the network size and dataset size. In the following examples, a single neuron with two inputs and one output is used for demonstration purposes and is shown in Figure 3.1.

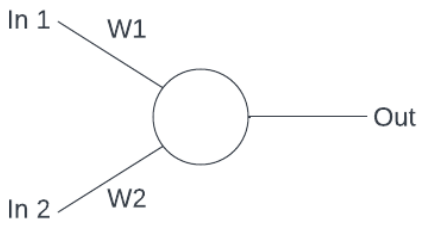


Figure 3.1: Two input example neuron.

3.2.1 Phase Accumulation Oracle (Λ) Improvement

Figure 3.1 shows the arbitrary BNN circuit that will be used for demonstration in the training circuit. To implement this as a Quantum Binary Neural Network, the neuron required two qubits for inputs, two qubits for weights, one qubit for the output, and one qubit for the dataset output. This gives a total of six qubits used for the BNN. The first step in implementing the neuron is to XOR the weights to their corresponding inputs and is done by using CNOT gates. The activation function can

be implemented by defining a threshold value and applying multi-controlled NOT gates that implement this threshold.

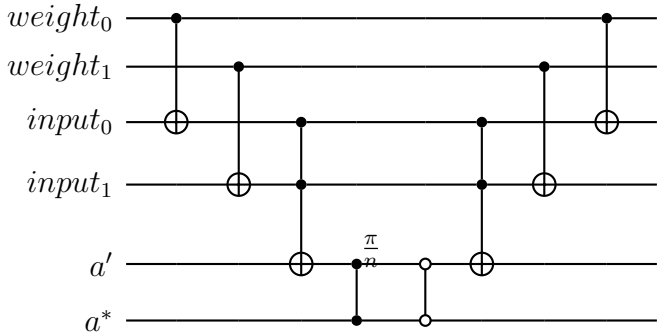


Figure 3.2: Quantum Phase Estimation binary neural network two input neuron example. This circuit implements the neuron shown in Figure 3.1 for the QPE training implementations. The first two CNOT gates implement the XOR functionality of the weights and inputs. The doubly-controlled NOT gate implements the activation function, where the threshold is 2 in this case. The 1 controlled and 0 controlled phase gates are the accumulation oracle, Λ, and is an improvement made by this thesis. The old version found in [1] can be seen in Figure 2.9.

Figure 3.2 shows the full QBNN version of the example seen in Figure 3.1. The multi-controlled NOT gate activation function utilizes a threshold $th = 2$ in this case. A key design change in the accumulation oracle, Λ, of the circuit is shown in Figure 3.1 that diverges from the design shown in [1]. Here, there are only two qubits used as the ancillary qubit proposed was removed. This is possible due to the fact that phase gates simply are doubly-controlled gates with no target qubit. The phase is simply applied if the state matches the control state. That means that the circuitry and ancillary qubit used in [1] can be removed and replaced with two multi controlled phase gates. The new design is controlled based on the values of the output and the expected output. The phase controls are both set to '1' or both set to '0' in order to ensure that the phase is only applied to a weight string if the output and expected output are both '1' or both '0'. This behavior will add the phase to the weight string if the QBNN output equals the expected output in the dataset.

3.2.2 Quantum Phase Estimation Fix

The next error that this thesis corrected was the Quantum Phase Estimation implementation that was shown in Figure 2.12. The controlled unitaries that are used in Figure 2.12 are placed in a reversed order compared to standard Quantum Phase Estimation implementations. This could cause confusion as well as incorrect results if others were to attempt the design shown in Figure 2.12. The corrected version is shown in Figure 3.3.

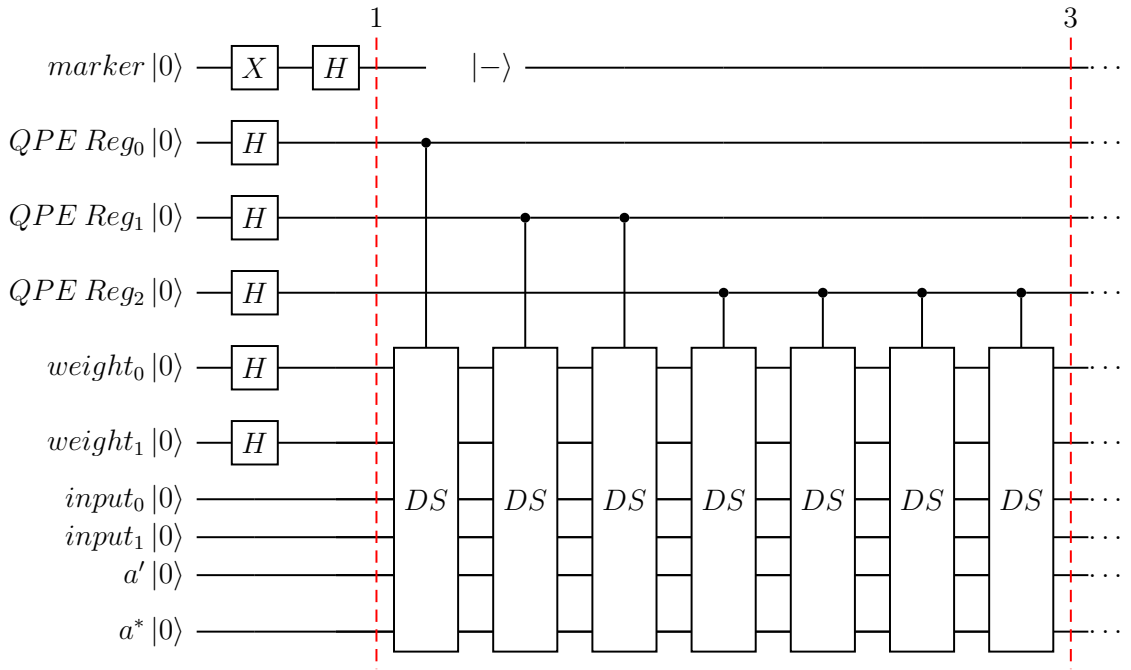


Figure 3.3: QPE QBNN training implementation. The controlled-DS gate represents the complete training dataset implementation that was shown in Figure 2.11. This controlled gate is used for the implementation of the Quantum Phase Estimation circuit and is doubled on each subsequent QPE register qubit. The orientation of the controlled-DS circuits are implemented such that the top QPE register qubit only implements one controlled-DS circuit, and the bottom QPE register qubit implements 2^n, where n is the index of the QPE register qubit. This orientation is inverted in [1] and causes confusion due to its non-standard nature.

Figure 3.3 shows the setup for the complete QPE training cycle. The 'DS' gates is the controlled version of the circuit shown in Figure 2.11 that implements the entire dataset. This controlled gate is applied by each QPE register qubit and doubles

with each successive qubit. This implements the QPE algorithm detailed in [7]. The purpose of doubling the controlled unitary for each successive qubit allows for multiples of the phase to kickback onto the phase estimation register qubits. The larger the multiple of the phase kickback corresponds to a more significant bit in the phases binary representation. This then allows the inverse QFT to convert the phase from the phase domain to a binary encoding of the phase. The notable change that differs from the method proposed in [1] is the way that the QPE algorithm is implemented. The order of how many times the controlled unitary is repeated for each qubit is inverted compared to the approach in Figure 2.12. The MSB, or $QPE\ Reg_2$ in Figure 3.3, needs to be rotated by 4θ in order for that bit to be the MSB. The approach given in [1] is backwards and will result in the wrong ordering of bits and an incorrect marking of a weight string using the accuracy threshold. After the controlled 'DS' gates, the inverse-QFT circuit will be applied to the QPE qubits to convert the phase representation in the Fourier basis to the computational basis. The marking of the weight string using the accuracy threshold will remain the same as described in Section 2.4 and is shown in Figure 3.4.

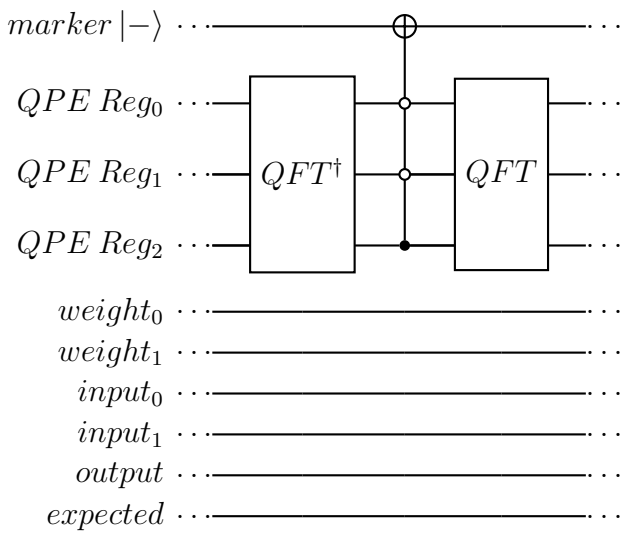


Figure 3.4: This circuit continues the circuit shown in Figure 3.3. This section is placed after the implementation of all the controlled-DS circuits. This uses the inverse Quantum Fourier Transform to convert the weight string phases to a binary encoding, with the LSB being the top most qubit. A triple-controlled NOT gate is then used to mark the weight strings that have a '100' binary encoding, or a phase of $\frac{4\pi}{n}$. The QFT is then applied to begin to uncompute the circuit.

Figure 3.4 shows the application of the inverse QFT to the phase results obtained in Figure 3.3 to give the estimated phase of $e^{2\pi i\theta}$. The accuracy threshold can then be applied to the weight strings, marking the strings that succeed above a certain threshold. In the case of Figure 3.4, the threshold chosen is a value of $\theta = 4$ that represents the number of dataset entries that the particular weight string computed correctly. After marking the weight strings with the accuracy threshold, a QFT can be applied to uncompute the inverse QFT. The full accumulation cycle seen in Figure 3.3 can also now be uncomputed to return the circuit to the original state.

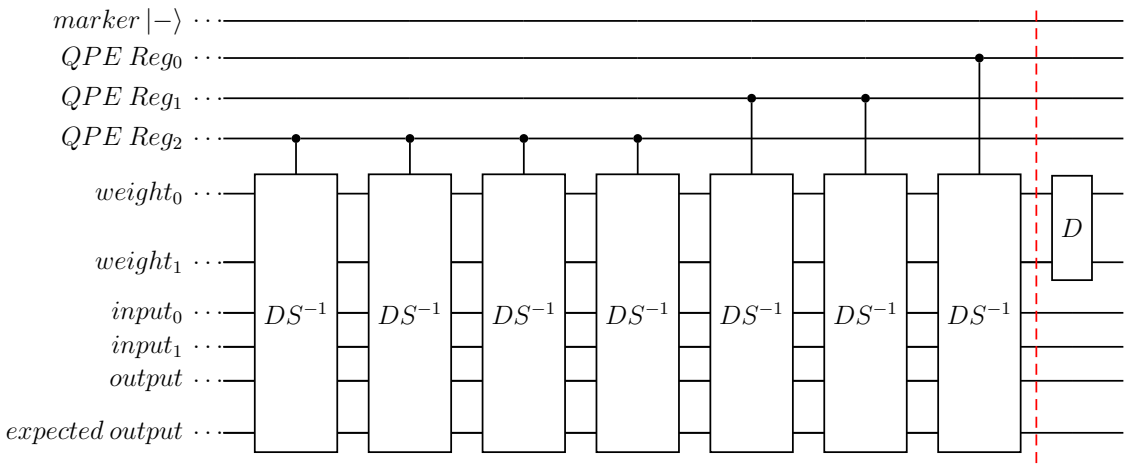


Figure 3.5: This circuit is an additional continuation of the circuit shown in Figure 3.4. This circuit uncomputes the controlled-DS circuits that were applied previously and then applies a Grover's diffuser to the weights. NOTE: A negative phase is used in the QBNN accumulation oracle for the 'DS' circuits.

The uncomputation of the QBNNs using the full dataset can be seen in Figure 3.5. It is important to ensure that within these unitaries, the accumulation oracle, in Figure 3.2, applies a negative phase to ensure that all of the phases return to 0 before the Grover's diffuser is applied. The Grover's diffuser is then applied and shown as 'D' in Figure 3.5. To change this algorithm for other networks, there only requires a change in the BNN controlled unitary implementation and with it the number of qubits needed to implement the BNN. The number of QPE qubits can also be adjusted to add or remove precision. The rest of the algorithm stays consistent.

3.3 Improved QPE BNN Implementation

The improved version of the QPE training implementation has a subtle change that decreases the depth of the quantum circuit greatly. In the original paper proposed by [1], the author uses the generic implementation of the QPE algorithm. The general idea of the QPE algorithm is to determine the phase of a black box unitary operation. The algorithm can determine the resulting phase of the black box without knowing what the phase should be. This works by implementing the black box circuit as a

controlled unitary on each QPE register qubit, but doubles the number of controlled unitaries used for the next highest significant qubit in the register. The requirement of doubling the number of controlled unitaries begins to exponentially increase the depth of the circuit with QPE register size. This however can be avoided by removing the need to double the controlled unitary for each subsequent QPE qubit.

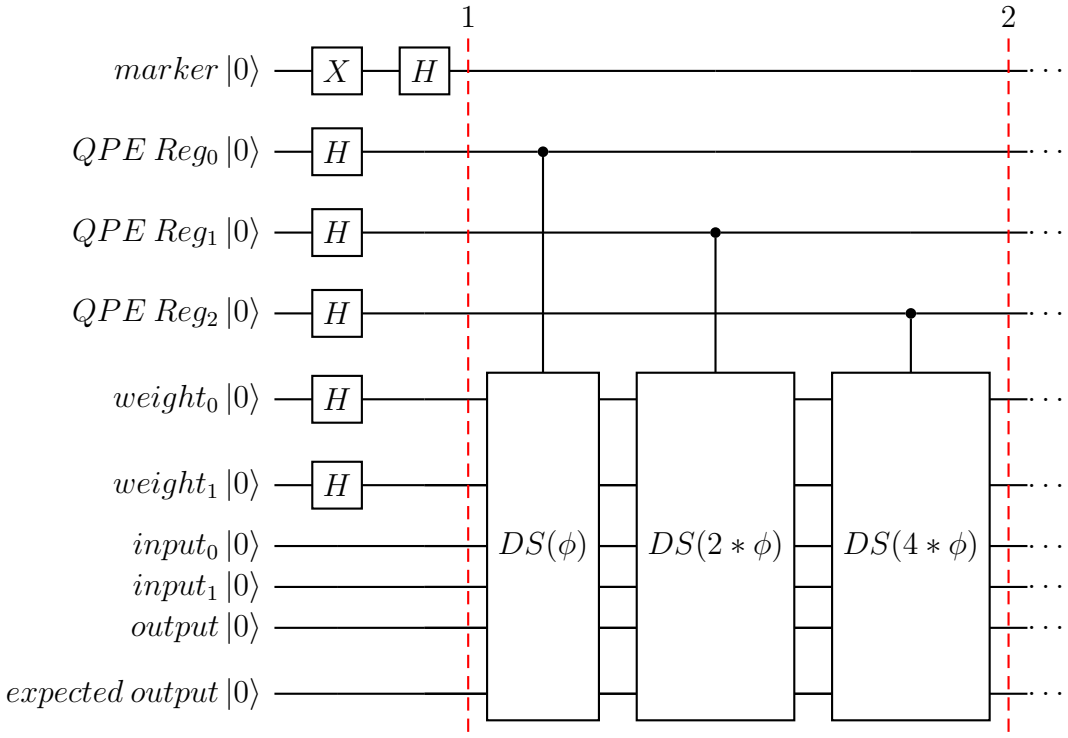


Figure 3.6: The QPE Improved training circuit follows the same structure and order of the original QPE training circuit shown in Figure 3.3. The difference, or improvement, of this circuit can be seen within the QPE part of the training. Due to the fact that the accumulation oracle is a set phase based on the number of training dataset entries, the oracle phase can simply be doubled for each higher bit QPE register as seen above. This reduces the depth of the circuitry greatly.

Figure 3.6 shows the resulting circuit, utilizing only one controlled unitary per qubit. The controlled gates in this circuit represent the 'DS' gates shown in Figure 3.3 and the ϕ represents the phase that the accumulation oracles (Λ) apply. This method is possible due to the fact that the controlled unitary in this problem is known. The controlled unitary is a circuit implemented by the user and applies a rotation that is based on the size of the dataset. That means that for each qubit in

the QPE algorithm, the phase that is added to the weight strings can be doubled for each successive qubit, essentially performing the same operation of the original QPE algorithm shown in Figure 3.3, without the added depth. This method is also applied to the uncomputation section of the training circuit where the controlled unitaries in Figure 3.6 are reversed and the phases that are applied are negated.

3.4 Register Counting Quantum Binary Neural Network Implementation

The Register Counting implementation follows a similar structure to the Quantum Phase Estimation implementation provided in [1]. The paper proposes the implementation of the register counting method, without providing the circuitry for it. The paper also notes that $\lceil \log_2 (dataset\ size + 1) \rceil$ qubits are required to have a register that can increment for each dataset point. The incrementation performed in the register counting method is similar to the QPE method. The registers for both methods are incremented after a weight string provides an output equal to the expected output given by the dataset. The difference between the two is that the incrementation in the QPE example increments the phase of the circuit, whereas the implementation of the Register Counting method increments a binary representation of the count that is stored in the register.

3.4.1 Register Counting Accumulation Oracle Implementation

The incrementing operation is performed by the accumulation oracle in the QBNN implementation and replaces Λ. This means that an oracle for the RC implementation needs to be designed such that the binary representation increases by one on each successful output. First, a circuit that increments a register was designed and tested for various register sizes.

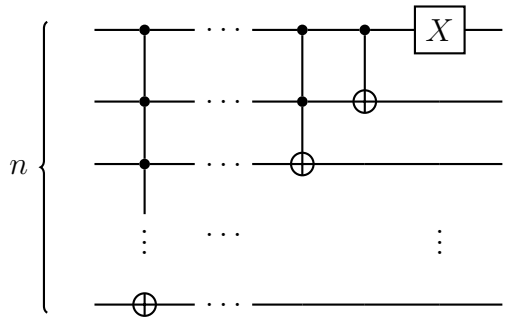


Figure 3.7: The register incrementation circuit is the replacement of the accumulation oracle (Λ) that is used for the Register Counting training circuit implementation and was developed by this thesis. The circuit simply increases the binary value in the n-bit register by a value of one. This circuit can then be converted to a controlled circuit and used as the accumulation oracle (Λ) for the Register Counting QBNN.

Figure 3.7 uses only n multi-controlled NOT gates for an n sized register. This circuit works by starting with the most significant bit and changing its value, only if all of the bits that are less significant than it are in state $|1\rangle$. It then moves to the next highest bit and performs the same controlled not. As an example, a binary value 0111 will accumulate to 1000 after the incrementation circuit. The implementation of the increment circuit shown in Figure 3.7 will eventually be used to replace the accumulation oracle (Λ) found in Figure 2.9. It will be replaced because the count of the number of times a weight string produces a correct output will now be encoded directly as a binary encoding rather than as a phase. An issue that may arise from using this circuit is that with a large register, there requires a CNOT with a large amount of controlled bits. With real quantum computers, this will pose a challenge as not all qubits can be used to control certain qubits and must adhere to the systems coupling map. An additional circuit needs to be created to allow for decrementing or uncomputing after the a accuracy threshold application, marking weight strings above a certain threshold, is complete. This can simply be done by reversing the circuit.

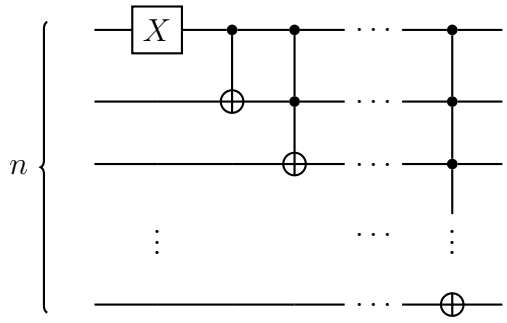


Figure 3.8: The register decrementation circuit is the inverse of the circuit seen in Figure 3.7. This performs the same functionality but decrements the register by a binary value of one as opposed to incrementing it by one.

Figure 3.8 can be seen to have simply reversed the circuit shown in Figure 3.7. This allows the values in the register to be decremented, and therefore allows the training circuit to be reverted to its original state.

The next step in creating the oracle (Λ) for the RC implementation is to allow this circuit to be doubly controlled.

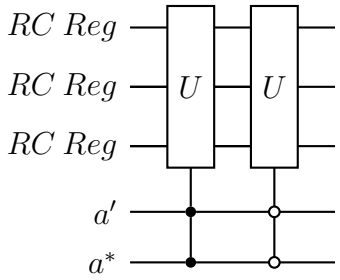


Figure 3.9: The Register Counting accumulation oracle implementation uses the circuit shown in either Figure 3.7 as a doubly controlled circuit. Using controlled by '1' and controlled by '0' versions, the oracle can detect if the circuit is either both in state $|1\rangle$ or $|0\rangle$.

Two of these doubly controlled incrementation circuits will be used: one for when the output and expected outputs are both $|1\rangle$, and one for when they are both $|0\rangle$. The gate U represents the circuits shown in Figure 3.7. The circuit shown in Figure 3.9 can now be used as the accumulation oracle Λ. Besides the phase oracle circuitry, Λ, the rest of the QBNN circuit does not change. And example is shown in Figure 3.10.

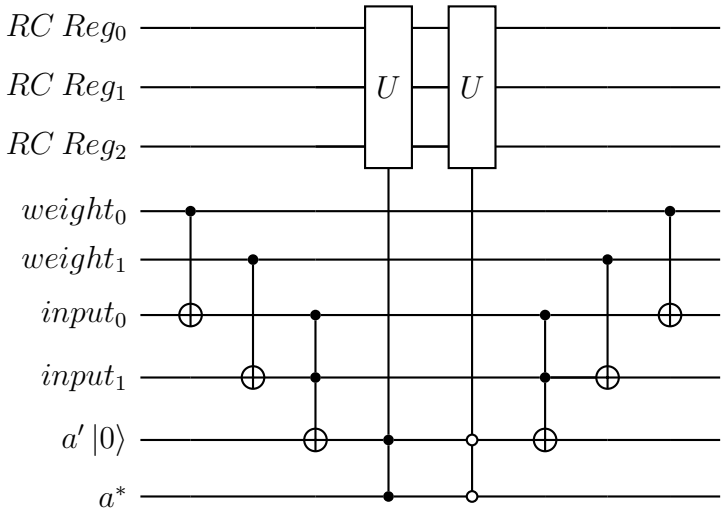


Figure 3.10: The complete register counting QBNN implementation is shown, implementing the 2-input neuron example. Here the RC accumulation oracle is implemented instead of the QPE oracle.

The QBNN circuit is repeated for each dataset entry, accumulating a count for each weight string that is stored in the register. Due to the fact that the RC circuit is not in the Fourier domain, the QPE circuitry, and therefore the controlled unitaries, are not needed. Once all dataset entries have been applied to the QBNN circuit, the accuracy threshold could then be applied and mark the weight strings that meet the threshold. The next step in the training process is to uncompute by reversing the circuitry of the QBNN, utilizing the decrementing circuits in Figure 3.8, and running the circuit for each dataset entry. Finally, the Grover's diffuser can be used to amplify the weight strings. This full training algorithm is shown in Figure 3.11 and Figure 3.12.

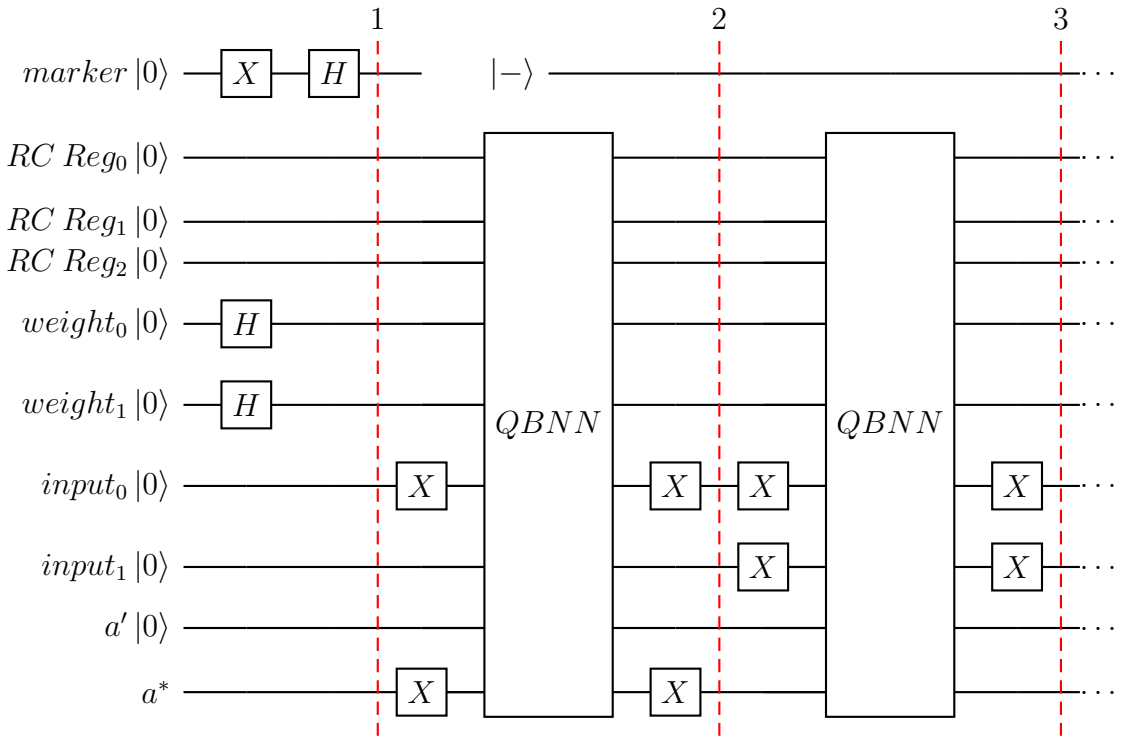


Figure 3.11: Due to the fact that the Register Counting implementation records each successful weight string with a binary encoding, no Quantum Phase Estimation Circuitry is required. This means that the training dataset only needs to be implemented once before the accuracy threshold. The caveat is that the Register Counting implementation needs to have enough qubits to represent the number of training dataset entries in a binary encoding. The example can only handle up to 7 dataset entries.

Figure 3.11 demonstrates the beginning of the RC Training Circuit. It is set up in the same way as the QPE training circuit, but uses the QBNN circuit designed in Figure 3.10. The X gates before and after the QBNN represent the values from the given dataset. In the example shown in Figure 3.11, it can be seen that the first data entry designates that $input_0$ should be '1', $input_1$ should be '0', and the *expected output* should be '1'. Whereas in the next dataset entry, the $input_0$ should be '1', $input_1$ should be '1', and the *expected output* should be '0'. This process is repeated for each data entry.

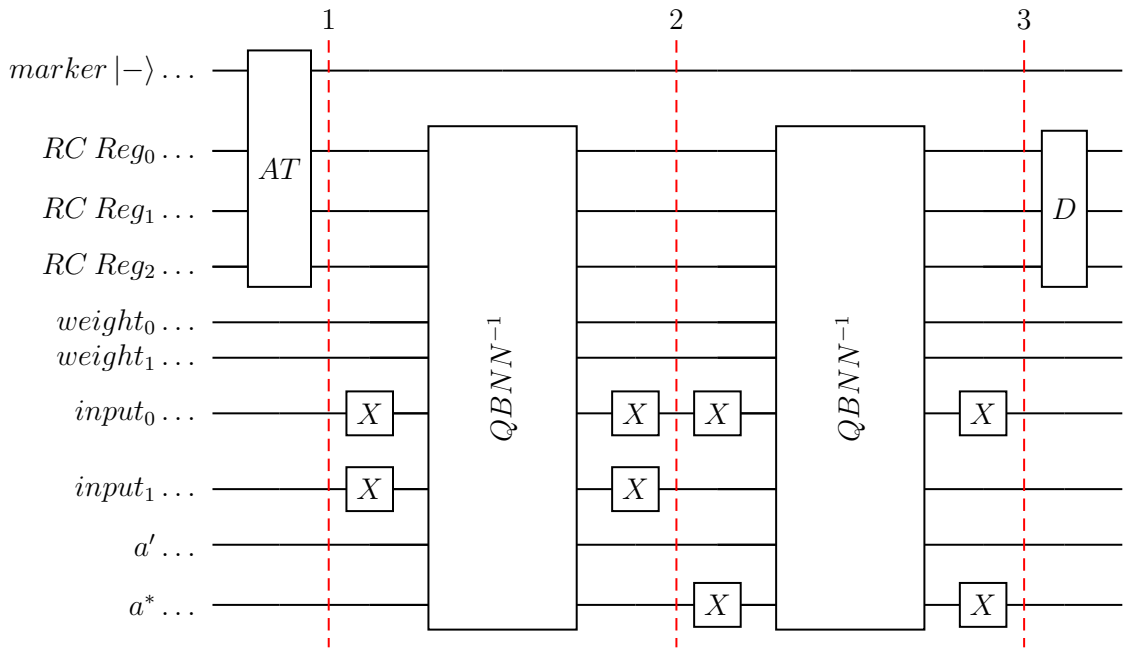


Figure 3.12: This circuit continues the circuit shown in Figure 3.11. The accuracy threshold is applied here and is a set of controlled CNOT gates that are dependent on the user's chosen threshold. The dataset is then uncomputed to return the states of all qubits to their original state, some of which are marked by the accuracy threshold. The Grover's diffuser is then applied.

Figure 3.12 shows the Accuracy Threshold(AT) and the uncomputation of the QBNN's before the Grover's Diffuser (D). The accuracy threshold applies a negative amplitude to the weight strings that have given a correct output greater than or equal to the threshold. The uncomputation utilizes the QBNN decrement oracle in Figure 3.8 and reverts the circuit back to its original state, while keeping the marked weight strings marked. The Grover's diffuser can then invert the amplitudes about the mean and amplify the marked weight strings, completing the Grover's algorithm.

3.5 Calculating Scalability of the Circuits

The scalability of these circuit implementations are very important, especially since they are implemented only in the quantum space. Current quantum computing technology is extremely sensitive to number of gates (depth) and number of qubits (width)

used. This is because it is very hard to avoid noise during quantum computation that can come from the environment. When a circuit has a larger depth, the circuit must run for a longer period of time, and the more qubits that are part of the circuit, the more difficult it is to ensure that all qubits remain without noise. The noise can alter the state of the ions or particles in the system and cause the computation to result in error. While technology is still improving for quantum systems, many other methods are being researched and employed to reduce the error. Some of these methods include error correction circuits and qubit mapping and routing. The less error that is observed from a system increases the feasibility of implementing the circuit on real quantum hardware. The expected error of a circuit can also be reduced by simply reducing the number of qubits or depth required to complete the desired functionality. It is therefore important to assess the change in quantum circuit width or depth as the problem size increases or decreases. This allows the feasibility of the implementation to be evaluated for potential real world usage in the future. The following paragraph details the method of calculating the circuit width and depth of each of the three implementations explained in Section 3.2, Section 3.3, and Section 3.4, as the size of the neural network and dataset increases.

3.6 Calculating Width of the Circuits

The best way to calculate the width and depth of each implementation is starting from the QBNN circuit. For all three implementations, the QBNN has the same number of qubits. This is defined by the number of weights, inputs, ancillas, and outputs that the QBNN requires and is defined as

$$Q_{QBNN} = Q_{input} + Q_{ancilla} + Q_{weights} + Q_{outputs} \tag{3.1}$$

Equation 3.1 shows that the number of qubits needed for the BNN implementation

is the combination of all the parameters of the QBNN plus extra ancilla bits for input copying and hidden neuron output. Upon inspection of various NN sizes the following is found

$$Q_{input} + Q_{ancilla} = Q_{weights} \tag{3.2}$$

and

$$Q_{outputs} = 2 * BNN_{outputs} \tag{3.3}$$

where the output qubits are for the final output of the NN and the expected output from the dataset. Using the observations shown in Equations 3.2 and 3.3, the width equation of the QBNN can be simplified to

$$Q_{QBNN} = (2 * Q_{weights}) + Q_{outputs} \tag{3.4}$$

It is important to note that [1] requires an extra ancilla qubit to implement the accumulation oracle of the QBNN. Whereas the implementations in this thesis only require the output qubits without an ancilla bit. For the equations shown here, that qubit will be removed.

3.6.1 Width of the Quantum Phase Estimation Training Circuits

The added width of the training portion is dependent on the implementation. The QPE and improved QPE implementations both utilize a number of QPE qubits and a marking ancilla qubit. The number of qubits required for the QPE is dependent on the desired precision, which is defined as

$$\theta = \frac{\pi}{2^{Q_{QPEQubits}}} \tag{3.5}$$

where $Q_{QPEQubits}$ is the number of QPE qubits used. Figure 3.5 is the smallest θ that can be observed from the given number of qubits. If more precision is desired, the number of qubits for QPE must be increased. The added width for the QPE implementation and improved QPE implementation is then

$$Q_{QPEImplementations} = Q_{QBNN} + Q_{QPEQubits} + 1_{marking} \tag{3.6}$$

3.6.2 Width of the Register Counting Training Circuits

The Register Counting Implementation is slightly different as the number of RC qubits is dependent on the number of data entries in the problem dataset. This is because the RC register must have enough bits to encode the maximum possible number of correct QBNN outputs, with the maximum being all data entries are outputted correctly. So the equation

$$Q_{RCRegister} = \lceil \log_2 n + 1 \rceil \tag{3.7}$$

Equation 3.7 utilizes a logarithmic function of base 2 to determine the number of bits needed to store the decimal representation, $n + 1$, into a binary representation. The n represents the number of dataset entries. The 1 is added to n as when using log to determine bits, 0 is included as a representation. So without the 1, the bits could only represent from 0 up to N-1. The result is then placed into a ceiling function that will round up to the nearest integer. This allows numbers that aren't perfect powers of two to result in an integer number of qubits. RC implementation also uses a marking qubit for the Grover's search. The total number of qubits for the RC method is

$$Q_{RCImplementation} = Q_{RCRegister} + Q_{QBNN} + 1 \tag{3.8}$$

3.7 Calculating Depth of the Circuits

The depth of the implementations is the next metric that will be calculated. The depth is an important metric because the more depth a circuit has then the more error will be found in the result. The depth for each of the three versions are different from each other and will be explained separately.

3.7.1 Depth of the Quantum Binary Neural Network Circuits

The first implementation that will be looked at is the original QPE implementation. Beginning with the QBNN depth the following generic equation can be used.

$$QBNN_{Depth} = 2*(Q_{Weights}+InputCopy_{Depth}+Activation_{Depth})+AccumulationOracle_{Depth} \tag{3.9}$$

Equation 3.9 shows that the depth of the BNN circuit is dependent on multiple parts of the circuit. These parts include the number of weights that the NN contains, the number of times the inputs need to be copied, the depth of the activation functions, and the depth of the accumulation oracle.

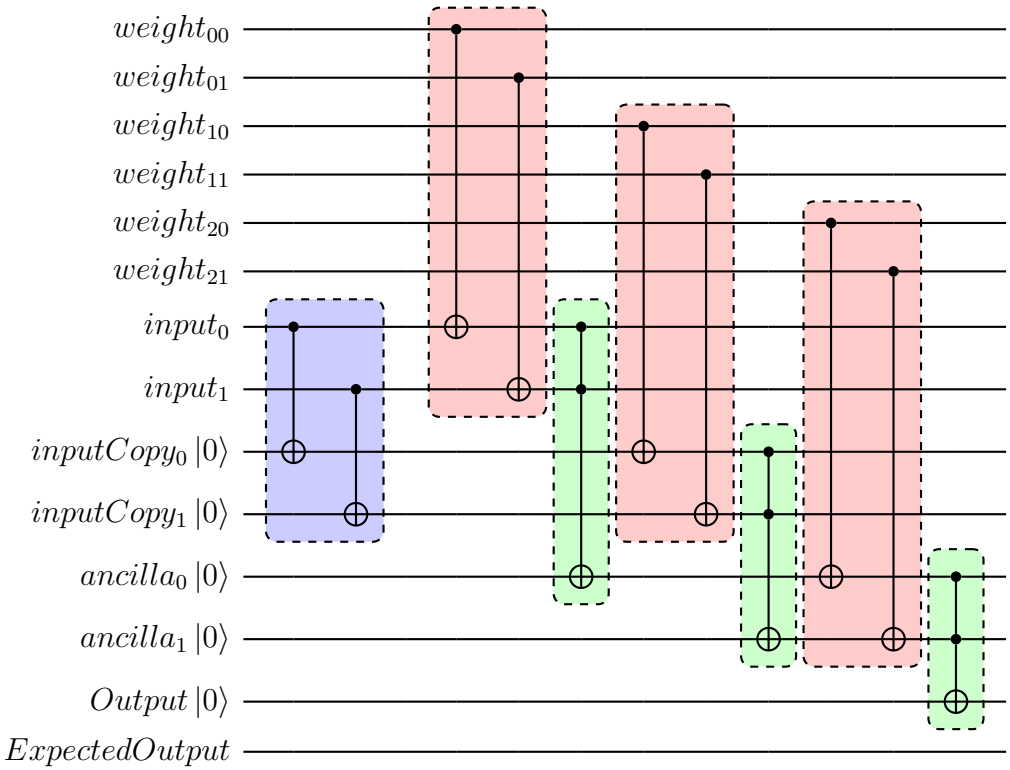


Figure 3.13: This figure shows the different color-coded parts of the QBNN circuit. While all three of these parts can be used for the QBNN, they will look different for each neural network implementation. The blue part is the input copy circuitry, the red is the application of the weights, and the green is the activation threshold circuitry. The example implements a 2-2-1 circuit, with two inputs, two hidden layer nodes, and one output.

Figure 3.13 shows the parts of the QBNN circuit as described in this section. The blue box represents the input copy circuitry composed of CNOT gates. The red colored boxes are the CNOTs that apply the weights to the corresponding inputs or ancilla qubits throughout the QBNN. Finally, the green boxes represent the activation functions of the QBNN for each neuron. It can be observed that the number of CNOTs required for the weights is equal to the number of weights as one CNOT gate is needed per gate. The number of input copies that are required is explained as

$$InputCopy_{Depth} = Inputs * (Nodes^{l} - 1) \tag{3.10}$$

Equation 3.10 shows that the depth of the input copy CNOTs are dependent on

the number of inputs of a single neuron, and the number of nodes in the layer l. This is because for any given layer l, the output of the nodes in the layer before it must be copied $(Nodes^l - 1)$ times for the inputs of layer l, where the number of nodes in layer l is $(Nodes^l$. This means all nodes, except for one node in a given layer, receive a copy of the previous layers outputs to use as its intputs. This is done in order to prevent the weights that act on the original outputs to disturb their values before the other nodes apply their weights to it.

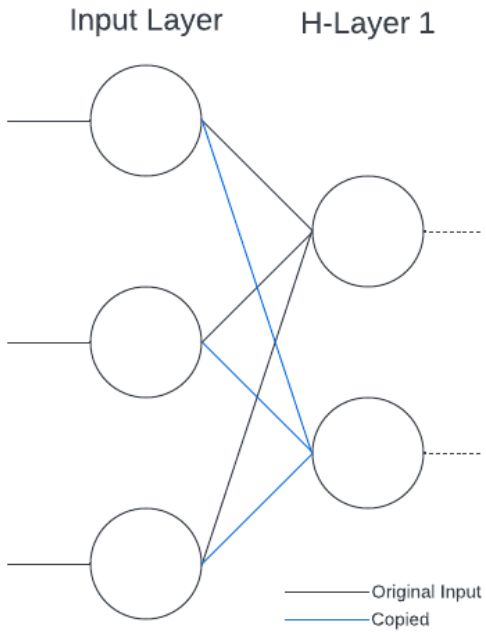


Figure 3.14: Example of copying inputs. The 3 input, 2 node hidden layer example above shows the need to copy inputs if there is more than one weight that will be applied to it. The black lines represent applications of weights onto inputs that are the original inputs. The blue lines are applications of weights that need to use a copied version of the inputs.

Figure 3.14 shows when inputs will be copied and how that translates to the depth of the QBNN circuit. The black lines indicate weights in the NN that will be applied to the original inputs, the blue lines indicate the weights that are applied to a copy of the inputs.

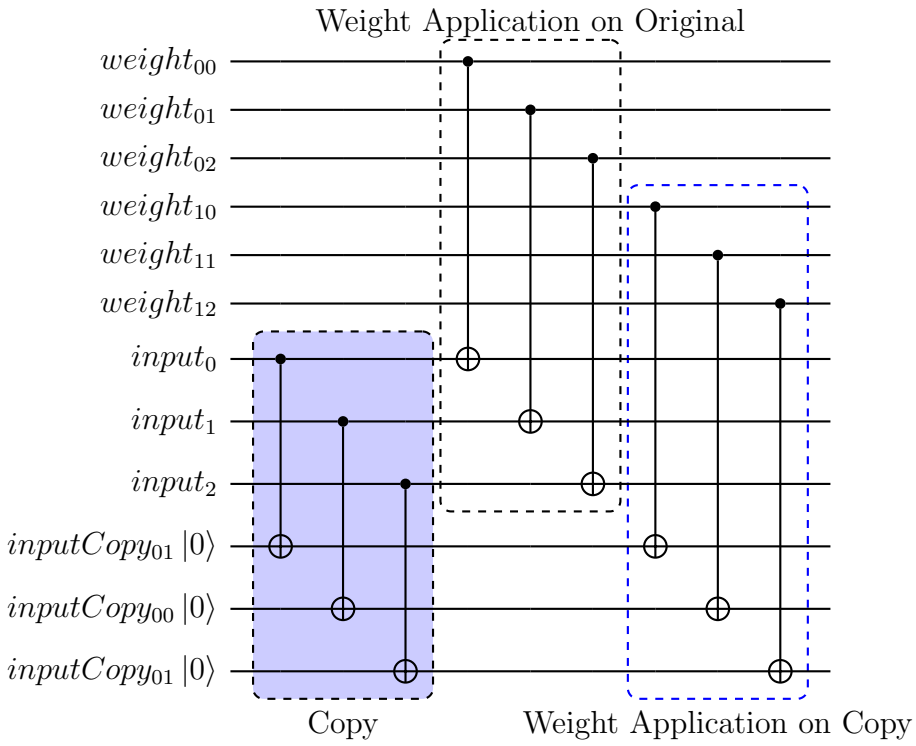


Figure 3.15: This quantum circuit implements the network shown in Figure 3.14. The blue highlighted section is the circuitry to copy the inputs. The blue dashed box is the weight applications onto the copied inputs.

Figure 3.15 shows the example of Figure 3.14 if it were implemented as a QBNN. It can be seen that Equation 3.10 works in this example as $InputCopy_{Depth} = 3*(2-1) = 3$.

The depth of the activation functions, or the green boxes in Figure 3.13 is dependent on a desired activation threshold, and the number of qubits that the activation function is using, which is the same as the number of inputs to any given node.

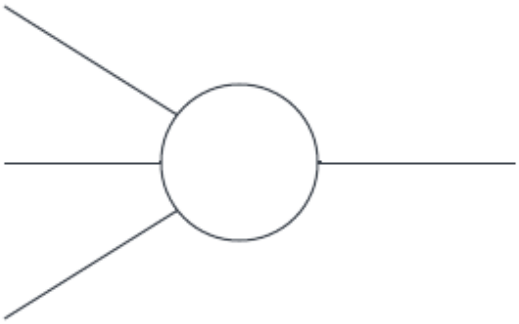


Figure 3.16: This three input, one output neuron will be used as an example to show the activation function circuitry.

Figure 3.16 will be used to demonstrate the calculation of the activation function depth. It can be seen that the node has three inputs, meaning the activation function will access those inputs and compare them to a threshold. The threshold can be in a range from 0 to the number of inputs. The inputs are summed together and if they are greater than or equal to the threshold, the output of the node will be $|1\rangle$. To implement this in a quantum circuit requires a multi-controlled CNOT gate for each possibility of bit combinations that are greater than or equal to the threshold.

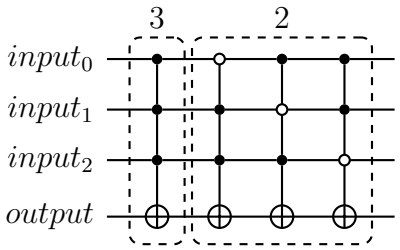


Figure 3.17: The circuit above implements a possible activation threshold for the neuron seen in Figure 3.16. The threshold in this case is a threshold of th=2 meaning that after the application of weights, the values must sum to greater than or equal to 2. This is implemented as a multi-controlled CNOT gate for each possible combination of qubits equal to, or greater than th=2.

Figure 3.17 shows a possible activation circuit for Figure 3.16. The chosen threshold for this circuit is two, meaning that the sum of the inputs must be greater than

or equal to two. The circuit implements a multi-controlled NOT gate for all of the input combinations that satisfy the threshold. The first region labeled '3' shows the MCNOT gate for if the inputs are all $|1\rangle$. The region labeled '2' shows the MCNOT gates for any combination of the inputs containing exactly two $|1\rangle$ states. The depth of the activation function can be generalized as

$$ActivationFunction_{Depth} = \sum_{k=threshold}^{Q} \binom{Q}{k} \tag{3.11}$$

In Equation 3.11, Q represents the number of input qubits to the activation function. The value k is initialized to the chosen activation function in which $0 \geq k \geq Q$.

This equation is then used for each neuron that utilizes an activation function and is summed as follows

$$ActivationFunctions_{Depth} = \sum_{l=0}^{L} \sum_{n=0}^{N} \sum_{k=threshold}^{Q} \binom{Q}{k} \tag{3.12}$$

Equation 3.12 shows the summation of the activation function, for all nodes N, in each layer L.

The depth of the accumulation oracle is defined by the number of outputs to the NN. In the general case, the number of outputs for the NN is one, meaning that the depth of the oracle does not change with NN size. However, when NNs have more than one output, for example using a one-hot encoding, the depth of the oracle grows as

$$QPE\,Oracle_{Depth} = 2^{Outputs} \tag{3.13}$$

In the case where the number of outputs is 1, the depth of the accumulation oracle stays at a value of two. The Register Counting accumulation oracle uses a circuit that was designed to accumulate the RC registers by one on each successful output. The depth of this circuit grows with the number of register qubits as well as the number

of outputs.

$$RC\ Oracle_{Depth} = RegisterQubits * (2^{Outputs}) \tag{3.14}$$

It is important to note that in Equation 3.9, the Copied inputs, weight size, and activation depths are doubled as the QBNN needs to be uncomputed after the oracle. After calculating the depth of the QBNN for the specific implementation, the depth of the complete circuit with training can be calculated.

3.7.2 Depth of the Original Quantum Phase Estimation Training Circuit

For the original QPE training implementation, the depth of the circuit depends on the number of data entries in the dataset as well as the number of QPE register qubits used. The QPE Implementation also requires the use of the Quantum Fourier Transform circuit. The QBNN circuit must be repeated for each dataset entry, for each QPE register qubit.

$$QPE\ Training_{Depth} = (\sum_{q=0}^{QPE\ Reg-1} n * 2^q * QBNN_{Depth}) * 2 \tag{3.15}$$
$$+(QFT_{Depth} * 2) + AccuracyThreshold_{Depth} + Grovers\ Diffuser_{Depth} + 2$$

Equation 3.15 defines the complete depth of the QPE Training circuit. The $QBNN_{Depth}$ is the depth that was calculated in Equation 3.9. This depth is multiplied by the number of data entries n, and by 2^q. 2^q comes from the requirements of the QPE algorithm, where the circuit controlled unitary is repeated 2^q times, where q is the index of one of the QPE register qubits. The product that results is then summed for each qubit in the QPE register, where the index of the qubits go from 0 to $QPE\ Reg - 1$. The resulting summation is doubled as it needs to be uncomputed. The QFT_{Depth} is the depth of the QFT circuit, which is doubled since an inverse QFT

is needed as well. The $Grovers\ Diffuser_{Depth}$ is the depth of the Grover's diffuser circuit, which is dependent on the number of $QPE\ Reg$ qubits.

The depth of the Accuracy Threshold (AT), $AccuracyThreshold_{Depth}$, is consistent for all three implementations. The depth can generally be seen to have the following equation.

$$AccuracyThreshold_{Depth} = 2^{Q_{Reg}} - AT_{th} \tag{3.16}$$

Equation 3.16 shows that the depth of the accuracy threshold is dependent on the number of register qubits (Q_{Reg}) that the given training implementation uses (Either the QPE version or RC version). It also depends on the chosen threshold of the Accuracy Threshold (AT_{th}). The depth is simply the max decimal value of the binary registers minus the decimal values threshold.

3.7.3 Depth of the Improved Quantum Phase Estimation Training Circuit

The QPE Simplified circuit, follows the same methodology as the original QPE circuit, with a minor difference when calculating the number of times the QBNN circuit repeats.

$$\begin{aligned} QPE\ Simplified\ Training_{Depth} &= (n * QPE\ Reg * QBNN_{Depth}) * 2 \\ +(QFT_{Depth} * 2) &+ AccuracyThreshold_{Depth} + Grovers\ Diffuser_{Depth} + 2 \end{aligned} \tag{3.17}$$

Equation 3.17 shows the updated equation for the QPE simplified circuit. Notice the summation is removed and the 2^q is replaced by simply the number of QPE qubits ($QPE\ Reg$) in the circuit. This change will serve to show how the depth of the QPE simplified circuit is greatly reduced compared to the original QPE circuit.

3.7.4 Depth of the Register Counting Training Circuit

For the RC Implementation, the depth of the circuit, like the QPE versions, is dependent on the number of data entries in the dataset. The circuit consists of set-up circuitry, the repeated QBNN circuits for each of the data entries, the accuracy threshold, the QBNN uncompute, and the Grover's diffuser.

$$RCTraining_{Depth} = (n * 2 * QBNN_{Depth}) + AccuracyThreshold_{Depth} + GroversDiffuser_{Depth} + 2 \tag{3.18}$$

The difference in the depth for the Equation 3.18 is that the QBNN circuit only needs to be repeated for twice the amount of the dataset size. Twice because the RC implementation does not use the controlled unitary circuits that are required for the QPE algorithm and only need to run through the complete dataset once to compute, and once to uncompute. The +2 that is at the end of the three depth equations: Equation 3.15, Equation 3.17, and Equation 3.18, is added to represent the setup gates including Hadamard's and X gates. The QFT and inverse QFT Depth[8] is dependent on the number of QPE qubits (QPE_{Reg}).

$$QFT_{Depth} = \sum_{i=1}^{Q_{Reg}} i + \lceil \frac{Q_{Reg}}{2} \rceil \tag{3.19}$$

Equation 3.19 is summing 1, up to Q_{Reg} register qubits which accounts for the Hadamard and controlled phase gates of the QFT. The ceiling function calculates the number of swap gates needed for the QFT.

Finally, the Grover's diffuser Depth [11] is always a depth of 5. This is because the circuit consists of a two sets of Hadamard gates and X gates on each qubit and a multi-controlled Z gate.

3.8 Initial Testing Implementations

A very small test was run to prove the functionality of each design. The test used a single neuron with three inputs and one output. A dataset was taken from [1] in order to compare results. The dataset is shown in Table 3.1.

in_0	in_1	in_2	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 3.1: 3-1 Dataset

The design of the circuit only required 12 qubits: three weight qubits, three input qubits, one output qubit, one dataset output qubit, three register qubits, and one marker qubit. The same QBNN circuit was used for all three implementations. It also utilized a activation function threshold value, $th = 2$.

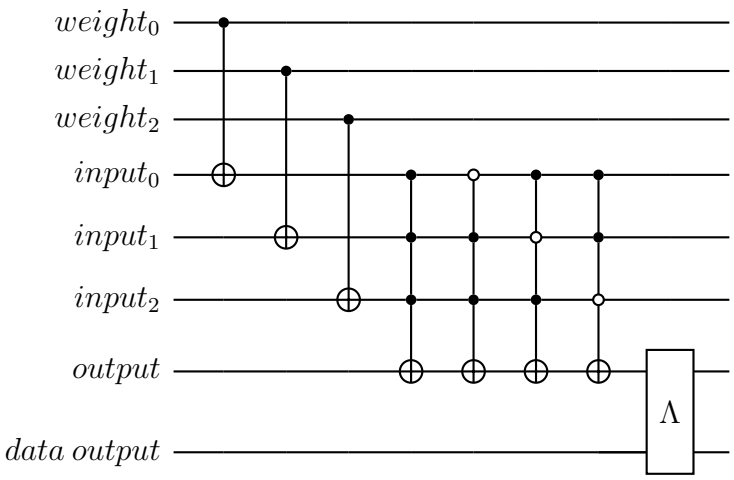


Figure 3.18: The first half of the 3-1 QBNN circuit is shown above, up to the accumulation oracle. The full implementation will mirror the CNOTs on the other side of the oracle.

As seen in Figure 3.18, the QBNN circuit only required a handful of CNOT gates and the accumulation oracle Λ. The activation function with a threshold $th = 2$ is implemented as the 4 multi-controlled CNOT gates. The training of the QBNN was implemented using the original QPE method, the QPE Simplified method, and the Register Counting method.

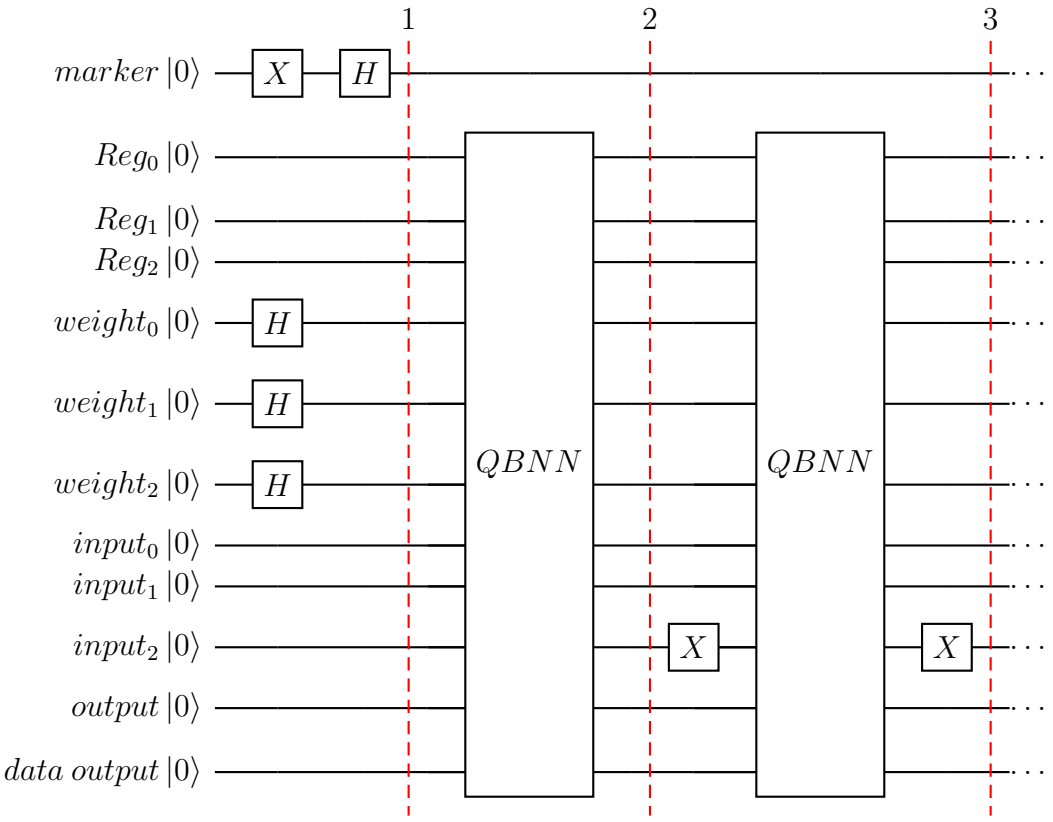


Figure 3.19: This circuit implements the training circuit of the 3-1 network. The QBNN box represents the circuit shown in Figure 3.18. The figure is set-up for the Register Counting training implementation. The QPE methods will have Hadamard gates placed on the Register qubits, and the QBNNs will be replaced with controlled versions that implement the whole training dataset.

Figure 3.19 shows the beginning of the training algorithm for the Register Counting algorithm. For the implementation of the QPE methods, Hadamards will be placed on the *Reg* qubits, and the dataset implementation will be replaced with controlled QBNN circuits. The results of this test are shown in the next chapter. This neuron simply finds the best set of weights that supports its activation threshold of two.

3.9 Practical Machine Learning Implementations

The original QPE training circuit shows various small scale implementations that demonstrate the functionality of the QPE implementation. The datasets given seemed

arbitrary and created just to prove that the circuit works. Their use of small scale networks are justified as attempting to run larger issues can result in many issues including but not limited to: the processing requirements of simulators, the representation of large weight string outputs, and circuit runtime. A goal of this thesis is to attempt to provide a real world example or problem that can be solved using the original QPE circuit proposed by [1], as well as the new circuits developed for this thesis. The work done in this thesis allow for the testing of larger circuits due to the reduced size of the designed implementations. These issues were encountered while trying to achieve this goal. However, a small scale real world problem was found and implemented to demonstrate the potential that the quantum circuits hold when quantum systems are improved in the future. The problem that was tested is convolutional edge detection image filters or in this use case, simply edge detection networks.

A 3x3 and 2x2 convolution edge detection circuits were tested using the three versions of the QBNN training circuits. For these tests, a neural network was created to identify vertical edges from given pixel values. The output of the network is a one when the pixels in the filter make an edge, and a zero when the pixels do not represent a vertical edge.

The 2x2 filter was used to detect vertical edges in images. There are 4 inputs needed for this network totaling a maximum dataset size of $2^4 = 16$ possible inputs. The dataset used for this filter is shown in Table 3.2.

0	0	0	0	0	0	0	0
0	0	0	1	1	0	1	1
0	1	0	1	0	1	0	1
0	0	0	1	1	0	1	1
1	0	1	0	1	0	1	0
0	0	0	1	1	0	1	1
1	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1

Table 3.2: 2x2 Dataset

The Table 3.2 shows all possible input combinations, where the highlighted inputs are considered to be a vertical edge while the others are not. Due to the much smaller size of this dataset, the full dataset was able to be trained on the neural network. The neural network used for the 2x2 edge detection was a 4-2-1 network, meaning 4 inputs, 2 hidden layer nodes, and one output.

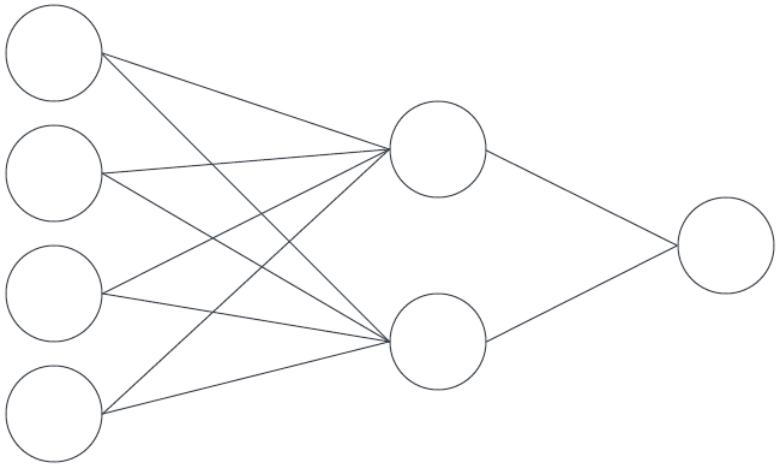


Figure 3.20: 2x2 Convolution edge detection filter neural network. 4-2-1

Training the network shown in Figure 3.20 with the three quantum algorithms still resulted in some complications that will be discussed later. However, the circuits were still able to run and produce a correct output. The quantum circuit for the 4-2-1 network is shown in Figure 3.21 for demonstration purposes.

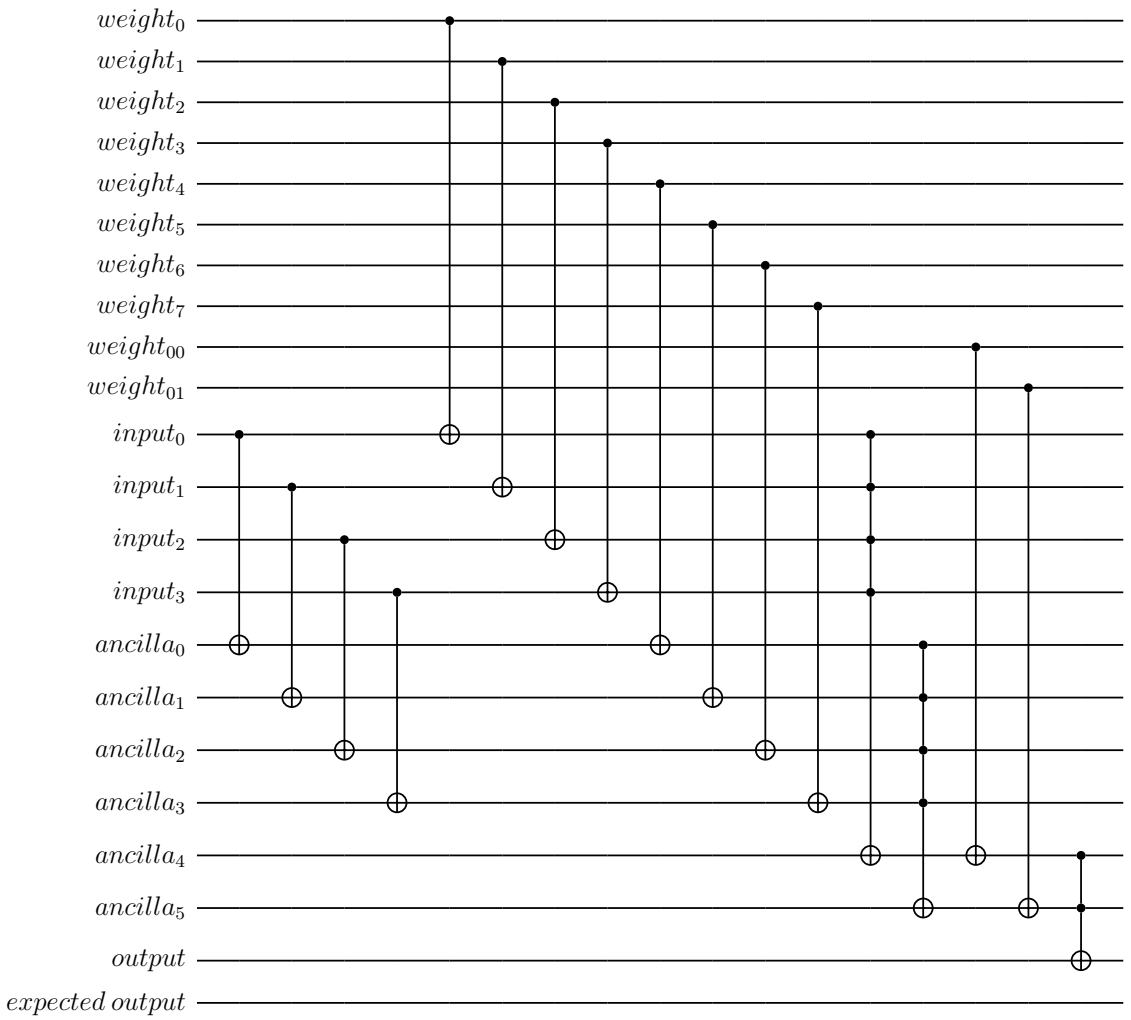


Figure 3.21: 4-2-1 Quantum binary neural network circuit.

As seen in Figure 3.21, the first four CNOTs implement the input copy circuitry, and the next eight CNOTs are the initial weight applications to the input values in the first layer. The first two multi-CNOT gates are the activation functions of each of the neurons in the hidden layer. Here the threshold of the neurons are set to $th = 4$. The last two weights in the next layer are then applied to the outputs of the hidden layer neurons, an finally the activation function of the output layer is applied. The results of the circuits are explained in the next chapter.

The 3x3 edge detection filter was then implemented using the quantum neural network. Due to the fact that there are 9 input values associated with the 3x3 filter,

there is a potential of $2^9 = 512$ possible input combinations. However, due to the large size of the dataset, it was reduced to 16 entries.

0	1	1	0	0	0	1	1	0	1	1	1
0	1	1	1	1	1	1	1	0	1	1	1
0	1	1	1	1	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0	1	1	1	1
1	0	0	0	0	0	0	0	1	0	0	0
1	0	0	1	1	1	0	0	1	0	0	0
0	0	1	1	0	0	1	1	1	1	1	1
0	1	1	1	1	0	1	1	0	0	1	1
1	1	1	1	1	1	1	0	0	0	0	1
0	0	0	1	0	0	1	1	1	0	1	1
0	1	1	1	1	0	1	1	0	0	1	1
1	1	1	1	1	0	0	0	0	0	0	1

Table 3.3: 3x3 Dataset

The data entries shown in Table 3.3 have two entries that were considered vertical edges while the others were not considered edges. The highlighted inputs of Table 3.3 were considered edges whereas the non-highlighted inputs were not. The two highlighted entries shown were chosen to be edges over the entries that are directly beneath them because the center pixel has a value of 1. For convolution, the main use of this practical application, the output of the image filter would be applied to the center pixel value. Reducing the number of entries that are considered vertical edges to only the once with a pixel value of 1 in the center will remove extra wide vertical

edges in the resulting filtered image. A neural network was designed to handle 9 input nodes and one output node. Originally, a network with only the input and output layers was used.

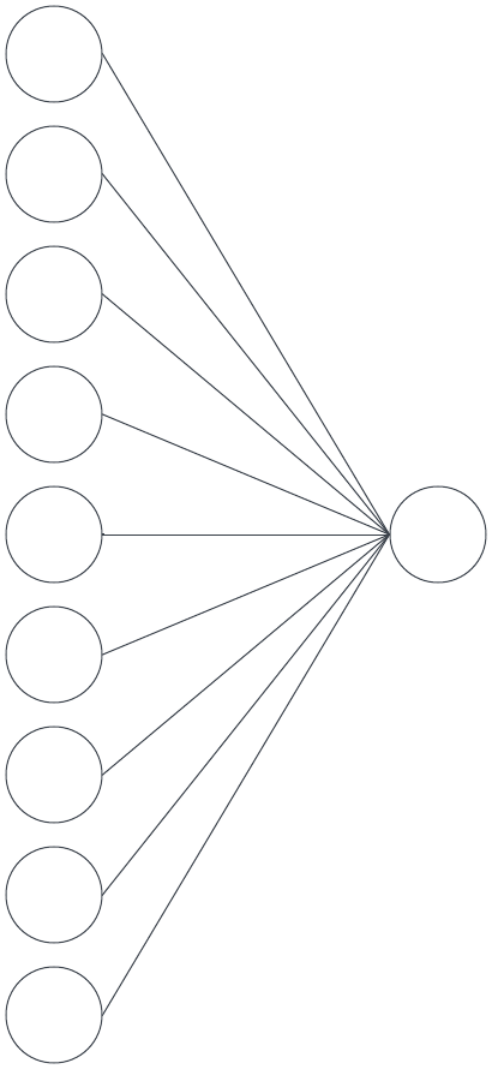


Figure 3.22: 3x3 Convolution edge detection filter 9-1.

However the accuracy of the model shown in Figure 3.22 could only correctly determine around 50% of the dataset using the best weight string. A second neural network was then designed to try and increase the accuracy of the model further. This involved including a hidden layer with 3 hidden nodes, creating a 9-3-1 neural

network.

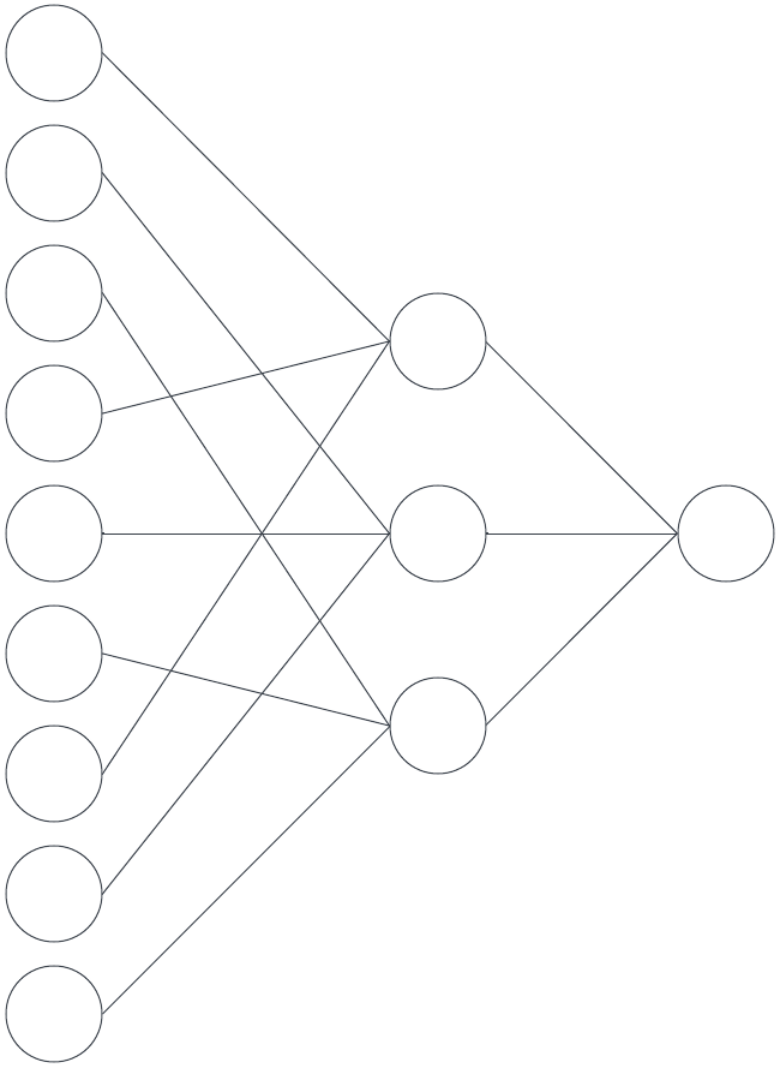


Figure 3.23: 3x3 Convolution edge detection filter. This is not a fully connected network (9-3-1).

However, as seen in Figure 3.23, the neural network is not fully connected. There are only 3 input nodes connected to each hidden layer node instead of the full nine. This allows for a larger network with an addition of only 3 weights compared to Figure 3.22. This network was tested with the dataset shown in Table 3.3.

Chapter 4

Results

4.1 QBNN Circuit Scalability

Using the equations and methods found in Section 3.5, the various circuit implementations could be analyzed. The figures shown in this section will detail how the circuits grow while adjusting different variables of the network. This section will observe how circuits grow with NN layers, NN layer nodes, dataset size as well as adjusting various parameters within the NNs. For each neural network in this section, it will be assumed that the NN nodes will be a fully connected network. First the width and depth of the neural network will be observed with varying hidden layer nodes. The NN will start as a 4-2-1 network, meaning there is 4 input nodes, 2 hidden layer nodes, and 1 output node. The network will also assume the following parameters:

QPE, RC Qubits	5
Dataset Size	16
Grover's Iterations	1
Accuracy Threshold	3
Neuron Thresholds	2

Table 4.1: Hidden layer nodes variation parameters

The number of hidden layer nodes will grow to show how the quantum neural

	QPE	QPE Simp	RC
2	77436	12540	2782
3	123068	19900	4254
4	176636	28540	5982
5	246076	39740	8222
6	347260	56060	11486
7	470268	75900	15454
8	636924	102780	20830
9	861116	138940	28062
10	1158716	186940	37662
11	1547580	249660	50206
12	2047548	330300	66334
13	2680444	432380	86750
14	3470076	559740	112222
15	4442236	716540	143582
16	5624700	907260	181726
17	7047228	1136700	227614
18	8741564	1409980	282270
19	10741436	1732540	346782
20	13082556	2110140	422302

Table 4.2: The depth for each of the three implementations given based on the number of hidden layer nodes.

network scales. This is shown for all three implementations: QPE, QPE Simplified, and Register Counting.

Table 4.2 shows the results from the initial hidden layer node depth test. This table and future experiment tables will be converted to a graph to better show the trends of the experiments.

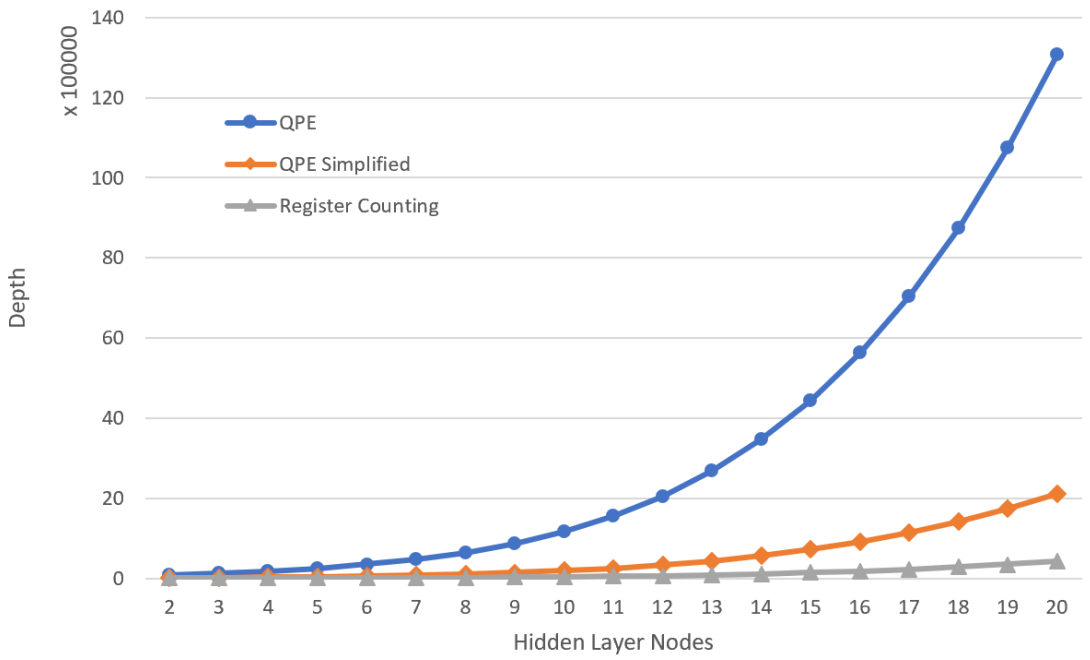


Figure 4.1: Quantum training circuit depth as the number of hidden layer nodes increase.

Figure 4.1 Shows the depth of the QBNN with training for each implementation. Each version increases exponentially as the number of hidden layer nodes increase. However, the rate that the original QPE implementation increases is much higher than that of the QPE Simplified version, as well as the Register Counting Implementation. This behavior is expected as shown by previous equations, and occurs due to the exponential nature of the original QPE algorithm. The QPE Simplified version is still larger than the Register counting implementation in this case. This should very often be the case as the QPE versions contain multiples of the dataset implementation whereas the Register counting version only implements the dataset once. The only case where the RC implementation depth would be more than the QPE depth would be if the QPE implementations only contained one QPE qubit and the dataset size is larger than two. However it would not be common to only use one QPE qubit as the precision of the results would not be high.

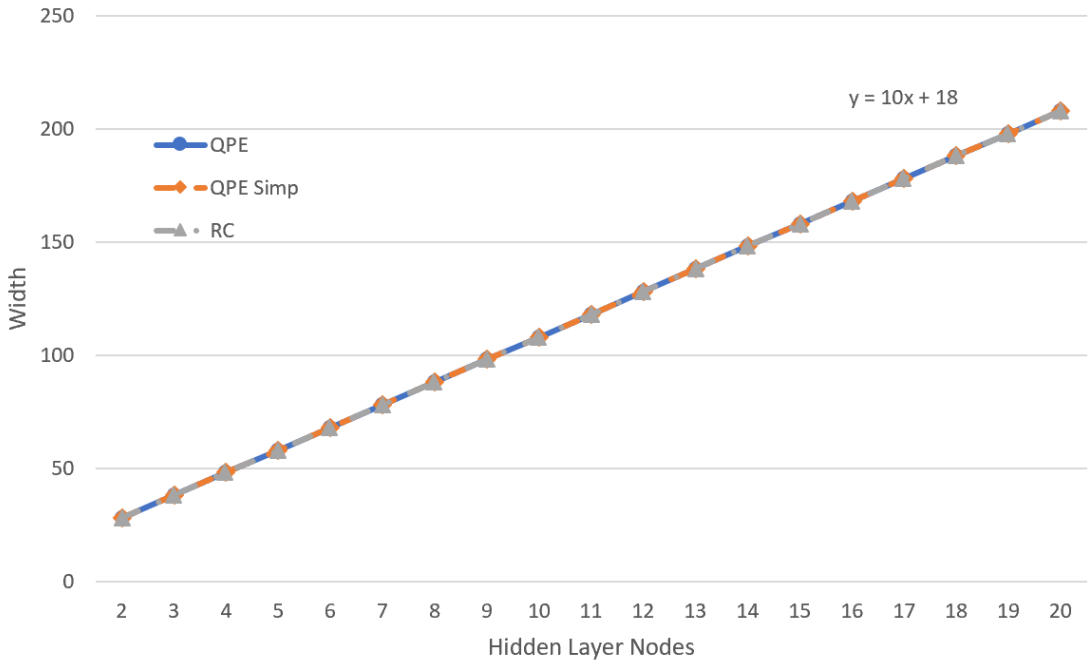


Figure 4.2: Quantum training circuit width as the number of hidden layer nodes increase. All three implementations follow the same width when adjusting the number of hidden nodes.

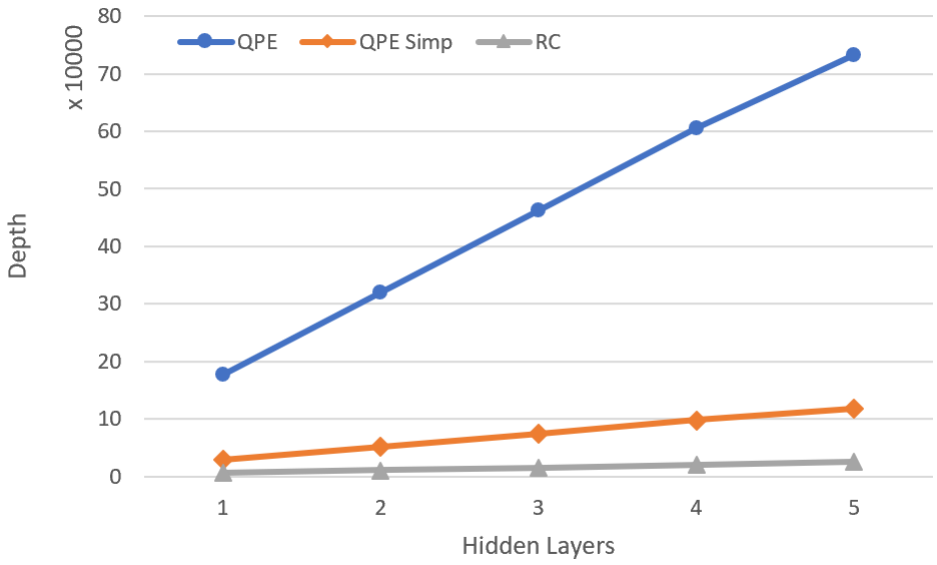
As seen in Figure 4.2, all three implementations grow linearly at the same values. This is because the QPE, RC Qubits are kept constant and the same. The QBNN for each implementation is implemented in the same way and therefore also contain the same number of qubits.

It can be seen that the circuit implementations increase in depth and width as the number of hidden layer nodes increases. The circuits were then tested to see how they scale as the number of hidden layers increased. The following parameters were kept constant for this test:

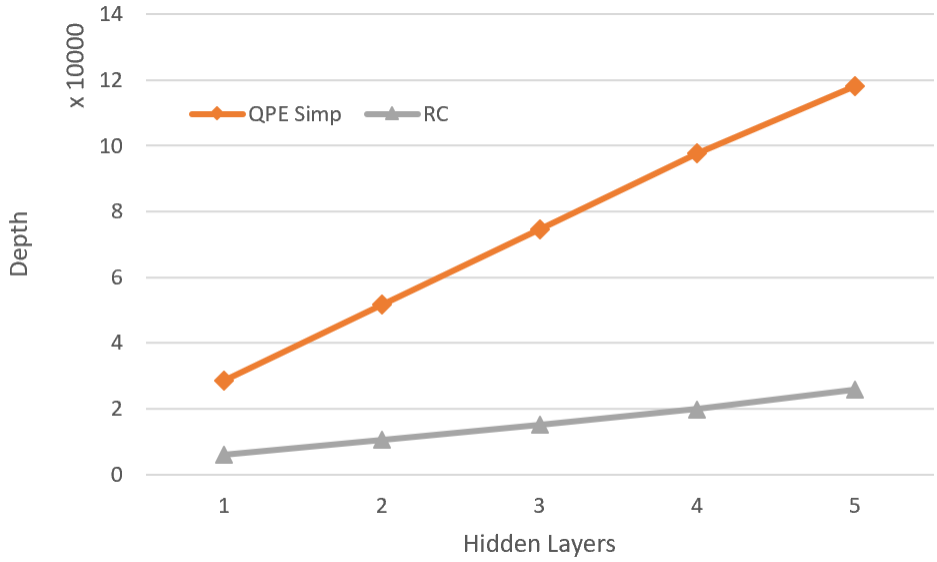
QPE, RC Qubits	5
Dataset Size	16
Grover's Iterations	1
Accuracy Threshold	3
Neuron Thresholds	2

Table 4.3: Hidden layer variation parameters

The neural network that was used for this test started as a 4-4-1 network: 4 inputs, 4 node hidden layer, 1 output. The network then added a 4 node hidden layer repeatedly and measured the width and depth of the circuit.



(a) All implementations



(b) Only QPE and RC implementations shown for clarity

Figure 4.3: Quantum training circuit depth as the number of hidden layers increases.

Figure 4.3 shows that the depth of the implementations increase linearly as the number of hidden layers increase in the circuit. The rate that the QPE implementation increases is large at first compared to the depth shown in Figure 4.1. However, the rate of Figure 4.1 eventually surpasses the rate of increase in Figure 4.3 for a large number of hidden layer nodes. The rate of increase for the Simplified QPE and

Register Counting implementations are much less than the QPE version in this test as well.

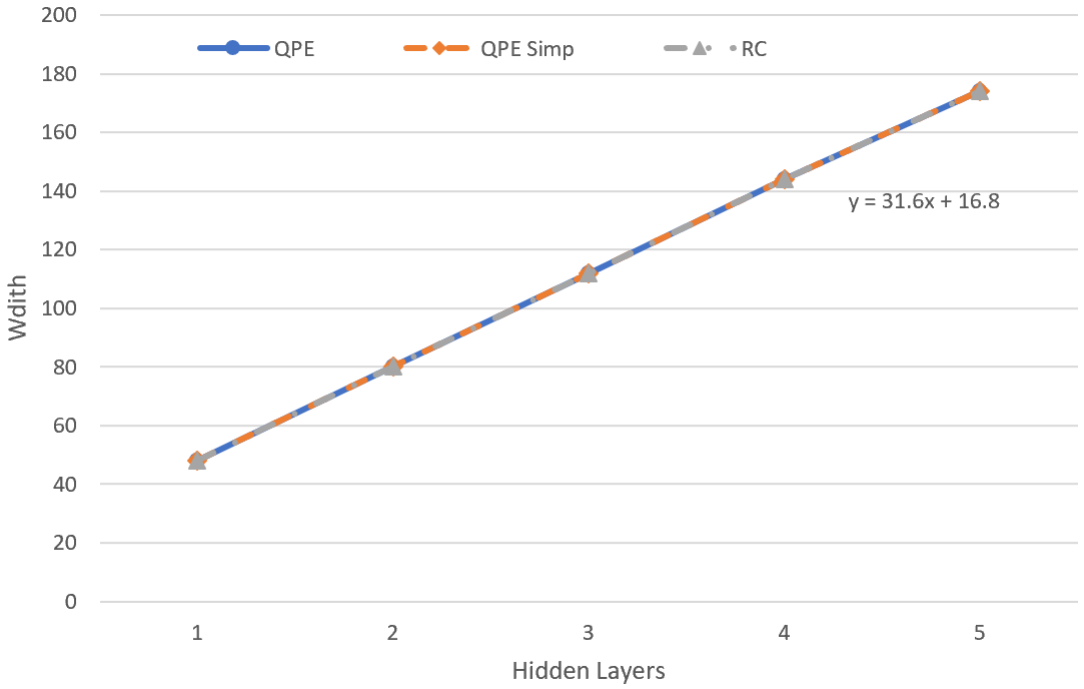


Figure 4.4: Quantum training circuit width as the number of hidden layers increase. The number of qubits for each implementation are all the same in this case as well.

The width of all three circuits, similar to what is seen in Figure 4.2, are all linear and the same values. The rate of increase in Figure 4.4 is 3.16 times larger compared to Figure 4.2. The reason this test is seen to be linear is due to the fact that with each additional hidden layer, a constant number of weights are added, and with it a constant number of qubits for each layer. For this case specifically, there are 16 weights added on each layer, meaning the circuit should increase by 32 qubits for each iteration.

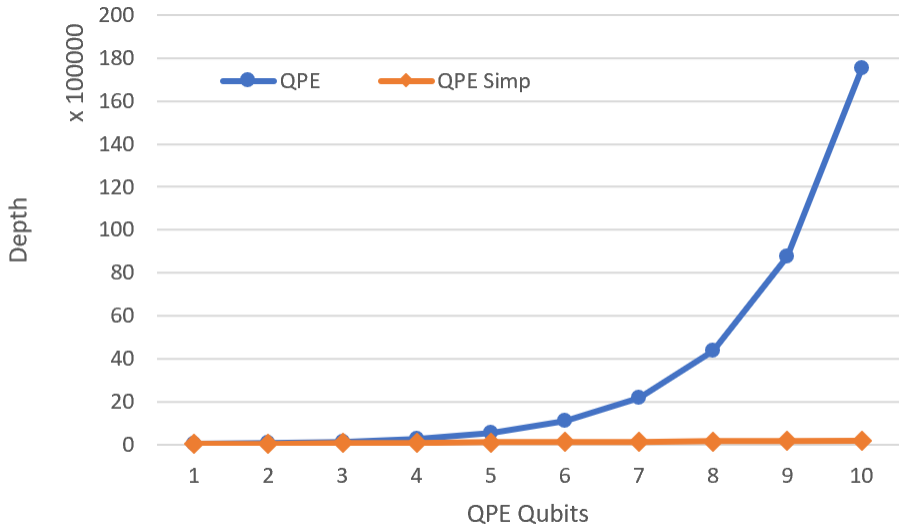
Next, the scalability of the circuits will be tested by increasing the number of QPE qubits used. The number of QPE qubits directly relates to the precision of the QPE algorithm, less qubits means less precision and therefore less control for what Grover's algorithm can return as a successful weight string. However, using less

qubits is always preferred do to scalability issues with quantum systems. The QPE and QPE Simplified implementations will have a correlation with changing the QPE qubits, whereas the RC implementation should remain constant for this test as the number of RC bit needed is dependent on the dataset size. The following parameters will be used for this test:

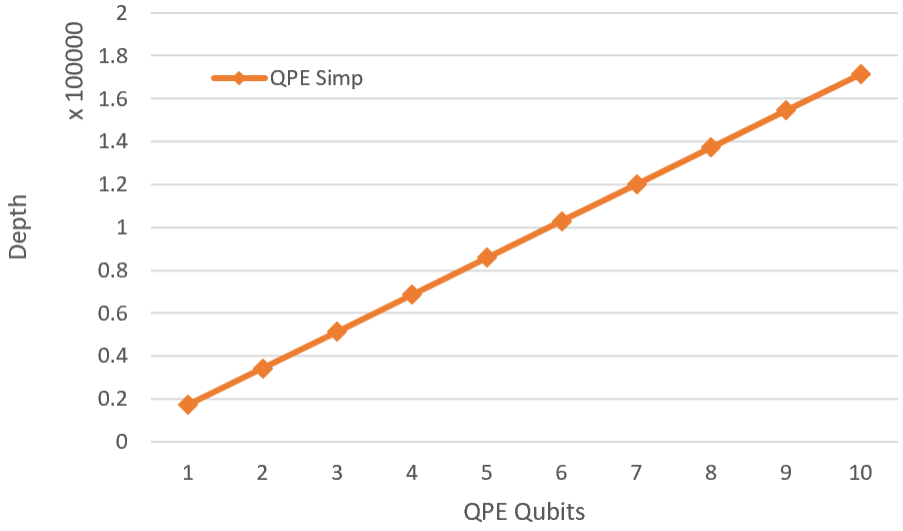
Nerual Network	4-4-3-1
Dataset Size	32
Grover's Iterations	1
Neuron Thresholds	2

Table 4.4: QPE qubit variation parameters

As seen in Table 4.4, the Neural Network used in this experiment is a 4-4-3-1 network: four inputs, two hidden layers with four and three nodes respectively, and one output node. The dataset is fixed to a size of 32 bits, meaning that the number of qubits needed for the Register counting method should remain at six qubits ($\lceil \log 232 + 1 \rceil$). The depth of the Register Counting method should also remain constant as the depth relies on the number of dataset elements.



(a) Original QPE Implementation



(b) Only Improved QPE implementation shown.

Figure 4.5: Quantum training circuit depth as the number of QPE qubits is increased. Note that the Register Counting implementation is not shown because it is dependent on the number of dataset elements.

Figure 4.5 shows that the depth of the original QPE circuit increases exponentially. The QPE Simplified circuit increases linearly and at a much slower rate than the original QPE circuit. This is because the QPE Simplified circuit increases by twice the depth of the QBNN depth for each QPE qubit added. Whereas the original QPE implementation increases exponentially by powers of 2.

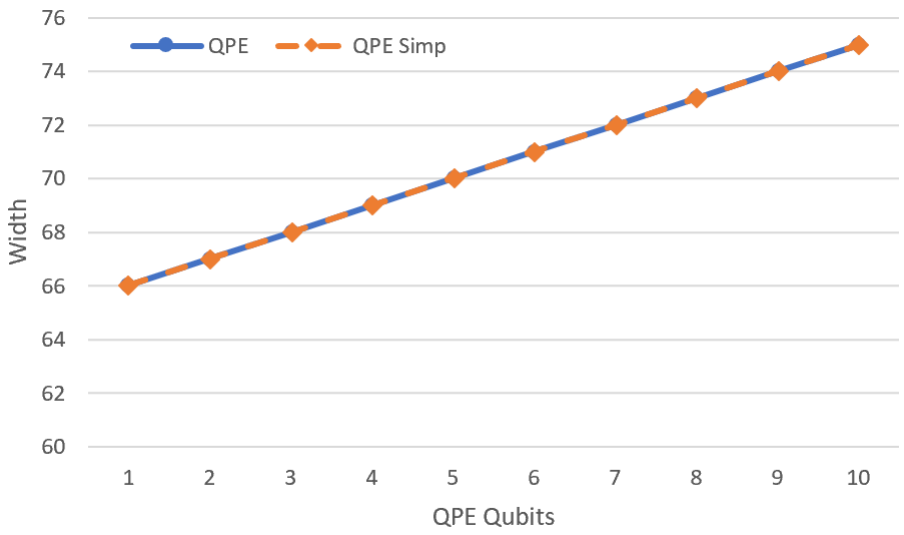


Figure 4.6: Quantum training circuit width as the number of QPE qubits is increased. Note that the Register Counting is not shown because it is constant and dependent on the number of dataset elements.

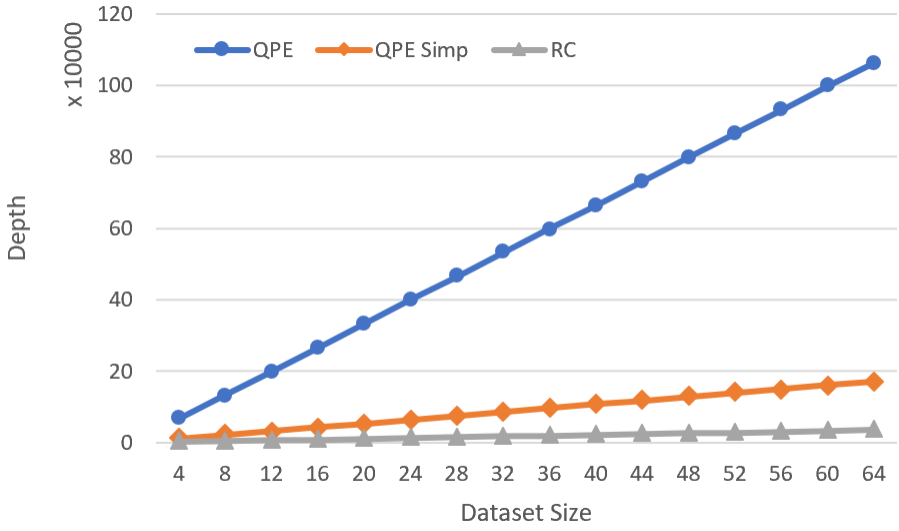
Figure 4.6 shows the width of the quantum circuit with training, as the number of QPE qubits are increased, the QPE and QPE Simplified implementations increase linearly and as the same size. When the dataset size is large, a smaller number of QPE qubits may be used to approximate the accumulation portion of the circuit, and therefore reduce the qubits needed.

As seen in Figure 4.6, changing the number of QPE qubits alters the width and depth of the QPE implementation, but does not effect the RC implementation. The size of the dataset will effect the RC implementation and is tested in the next experiment. The following parameters are kept constant for this test:

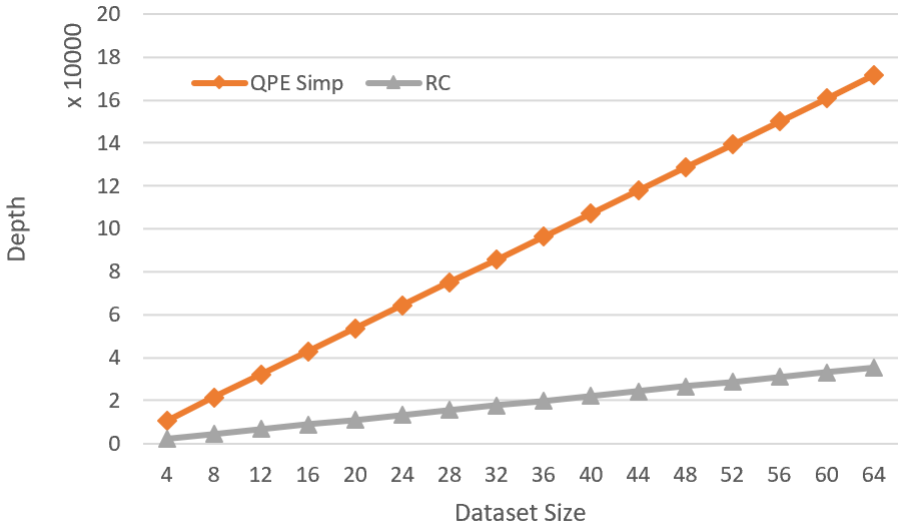
Nerual Network	4-4-3-1
QPE Qubits	5
Grover's Iterations	1
Neuron Thresholds	2

Table 4.5: Dataset size variation parameters

As seen in Table 4.5, the neural network used is kept constant at 4-4-3-1. The QPE qubits are kept at a constant 5 qubits for this test, and the other variables, Grover's Iterations and Neuron Thresholds, are kept the same as they have been in the previous tests. The size of the dataset starts at a value of 4 data entries, and increases by 4 for each iteration, up to 64.



(a) Quantum training circuit depth as the size of the training dataset increases.



(b) Quantum training circuit depth as the size of the training dataset increases. Only QPE and RC implementations shown for clarity

Figure 4.7: Dataset size effect on circuit depth.

Figure 4.7 shows the depth of the circuits as the size of the dataset increases. It can be seen that the increase in dataset size causes all three implementations to increase in depth linearly. Like previous tests, the original QPE method increases at the highest rate, followed by the simplified QPE version, and finally the Register Counting method. The width of the circuits are then tested.

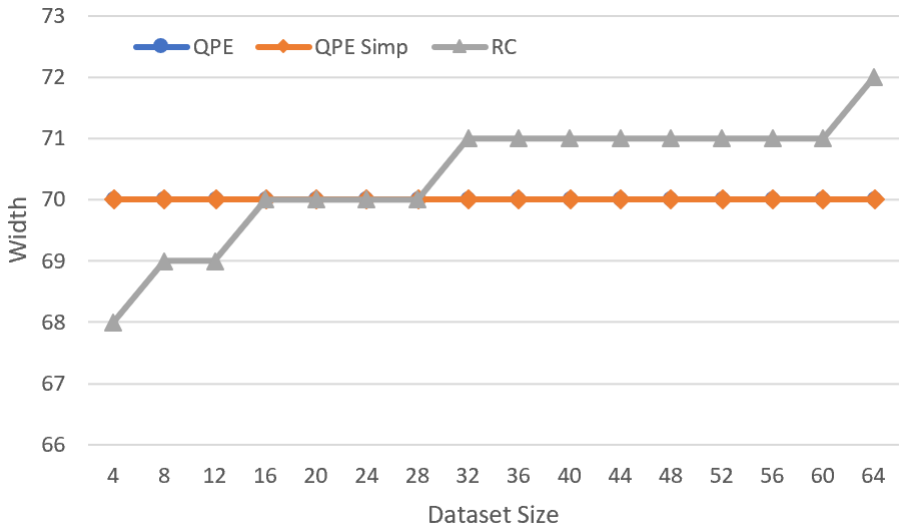


Figure 4.8: Quantum training circuit width as the size of the training dataset increases.

Figure 4.8 shows the width of the circuit implementations as the dataset size increases. Notice that the QPE versions do not change as their depth does not depend on the size of the dataset. The width of the Register Counting method increases logarithmically with the size of the dataset. With small datasets, the width of RC can be seen to be less than the QPE versions, however the number of qubits quickly exceeds that of the QPE versions. This shows the potential downfall of the RC implementation, as with large datasets, the width of the quantum circuit will increase greatly. The trade-off of the RC to the QPE methods will be discussed further in later sections.

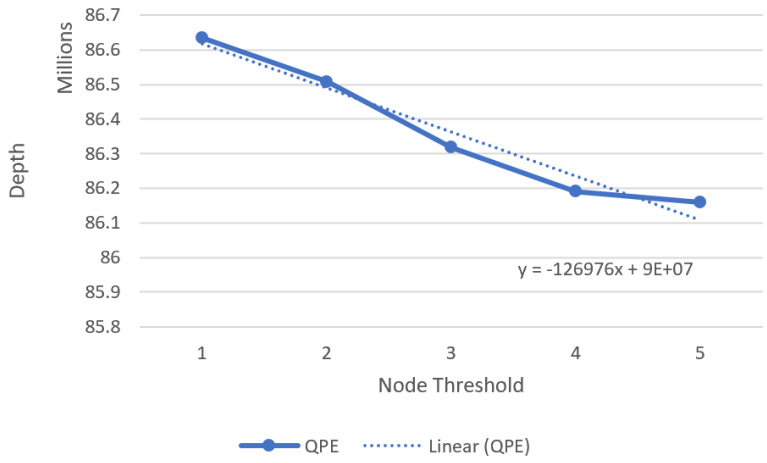
Finally, the quantum circuit dimensions are tested when the individual neurons(nodes) activation thresholds are increased. The following parameters are set

for this experiment:

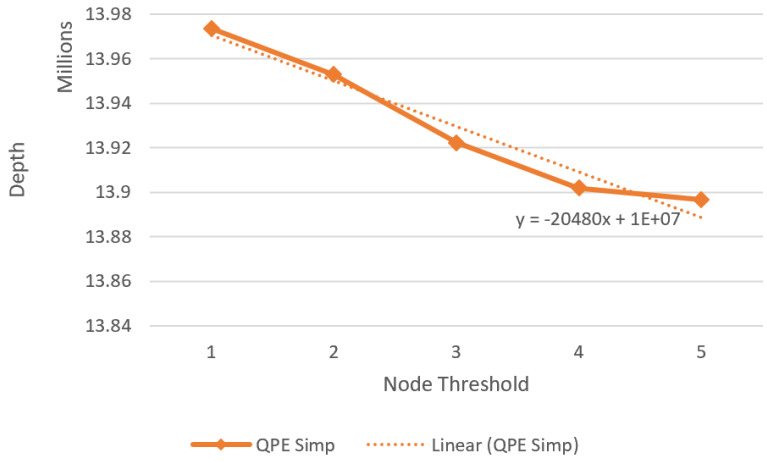
Neural Network	4-16-16-1
QPE, RC Qubits	5
Dataset Size	16
Grover's Iterations	1
Accuracy Threshold	3

Table 4.6: Node activation threshold variation parameters

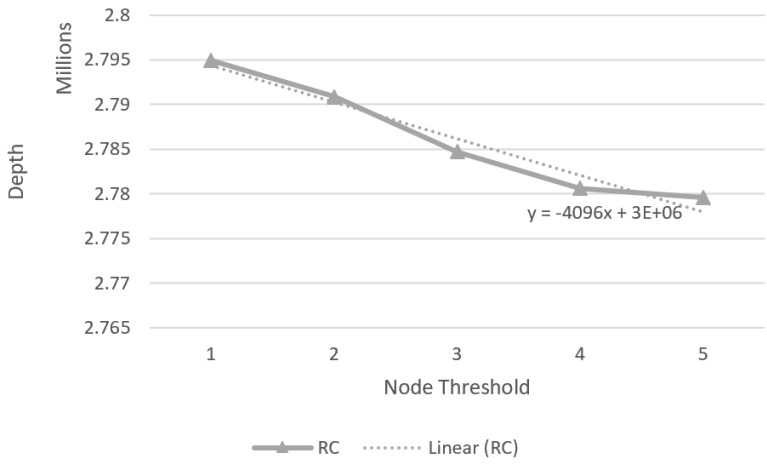
As seen in Table 4.6, the neural network used is slightly larger to show the effect of increasing the activation threshold with more detail. All other variables in the circuits are kept constant besides the nodes activation thresholds, that begin at 1 and are increased by 1 for each iteration.



(a) QPE



(b) QPE Simplified



(c) Register Counting

Figure 4.9: Effect of node threshold on circuit depth.

Only the depth was tested for this experiment as the width of the circuits are not effected by the activation functions of the neurons, as seen in previous sections. The depth of the circuits can be seen to decrease as the activation functions are increased. This is because the circuits require more multi-controlled CNOT gates as the activation functions are decreased. The multi-controlled CNOT gates must account for the binary representation of the threshold value, plus any binary encoded value above the threshold.

4.2 Variations in QPE bits

As seen and explained throughout this thesis, there does not need to be a fixed number of QPE register qubits. In Section 4.1, it can also be seen to reduce both the width and depth of the QPE implementations when a smaller number of QPE qubits are used. The trade-off however, comes with the reduced precision of what the accuracy threshold can detect. This behavior comes directly from the description and implementation of the Quantum Phase Estimation algorithm. As seen in [7], the number of QPE registers will determine what kind of result can be read from the QPE binary encoding. For example, if a qubit was rotated eight times by $\frac{\pi}{8}$, or in the case of QBNNs, achieved eight correct outputs, you would expect to find the final rotation to be $\frac{8\pi}{8}$ or π. You could easily see and detect this rotation by using four QPE qubits. This is achieved by rotating the index 0 qubit by $\frac{8\pi}{8}$, the index 1 qubit by $\frac{8\pi}{4}$, the index 2 qubit by $\frac{8\pi}{2}$, and the index 3 qubit by $\frac{8\pi}{1}$. Performing the inverse QFT will result in an output in state '1000' or binary for 8. However, less qubits can be used in this case. Since the goal is to observe 8 rotations of $\frac{\pi}{8}$, three qubits can be used for the QPE algorithm instead. In this setup, the index 3 qubit used previously will be removed. Which leaves only the index 0, 1, and 2 qubits with rotations $\frac{8\pi}{8}$, $\frac{8\pi}{4}$, and $\frac{8\pi}{2}$ respectively. This will result in an output of '100' or binary for four. This means that when fewer than the the required number of bits needed

to represent the rotations in binary are used. The QPE output is scaled by a power of 2 directly related to the number of missing qubits. So in this example, the output of the QPE algorithm will output $\frac{R}{2}$ where R is the number of rotations.

There is an issue however when using less QPE bits than required for the full binary encoding. In cases where $\frac{R}{2}$ results in a non-integer, the result is encoded in a distribution of all states, with the mean centered around the non-integer value. However this output could still provide correct results when training the QBNN. This is due to the fact that the accuracy threshold could be placed on one of the midpoint values, which would have a less probable, yet still valid marking that will allow for a successful Grover's algorithm. The obvious benefits to reducing the QPE qubits is reducing the overall width of the circuit. Small amounts of QPE qubits could be used with large datasets and still detect large rotations. Where the Register Counting method would need to be able to represent all dataset entries in a binary encoding which can grow large with large datasets.

4.3 Accuracy Threshold Variation

A parameter of the training circuits that can be adjusted to give varying results is the accuracy threshold value. This threshold determines which weight strings will be marked by the circuit for the Grover's diffuser to invert about the mean. The variation of this threshold value changes the depth of the circuit in a similar way to how the activation thresholds of neurons change the depth. However, the accuracy threshold is only run once per Grover's iteration. So this change in depth is only slight compared to the full depth of the training circuits.

Adjusting this value can allow the circuit to return weight values that are less accurate. For example, if a circuit was to find a weight string that correctly identifies 16/16 dataset entries, and a weight string that identifies 15/16 entries, setting the accuracy threshold to 15 would result in an accuracy threshold circuit depth of 2,

and return both the 15/16 and 16/16 weight string results. If the threshold was 16, the depth of the threshold circuit would be 1, and only return the weight string that results in 16/16 correct entries.

The accuracy threshold could be adjusted by the QBNN training user at their own discretion and to achieve any desired weight string accuracy. This only effects the depth very slightly and will not make a huge computation time difference.

4.4 Initial Testing Results

As discussed earlier in this thesis, the functionality of the Simplified QPE training circuit, and the Register Counting circuit were tested. For these tests, the 3-input, 1-output neuron was used and trained on a dataset provided by [1]. This was done to compare functionality against the method described in [1].

in_0	in_1	in_2	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 4.7: 3-1 Dataset

By observing the dataset, the 3-1 Neuron should be expected to perform best when the three weight qubits are set to $|000\rangle$. This would allow the inputs to enter the activation threshold unchanged. Due to the fact that the threshold value is set to

$th = 2$, the neuron should output a $|1\rangle$ state if the inputs sum greater than or equal to $th = 2$. This functionality can be observed in Figure 4.7. The QPE implementation proposed by [1], is implemented first as a control.

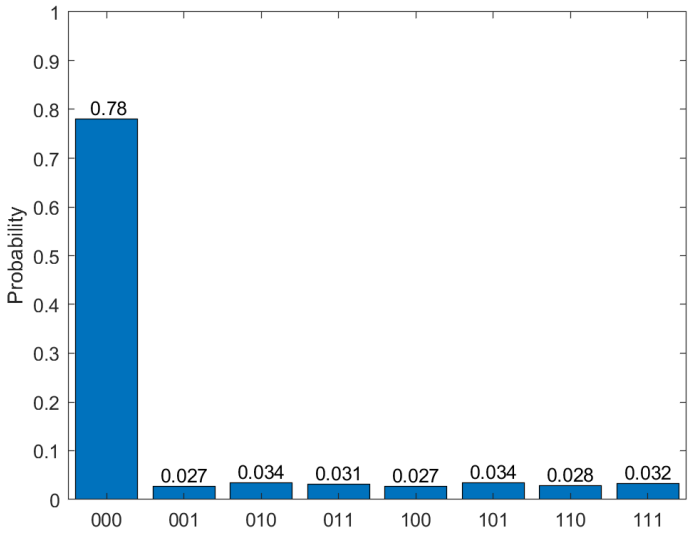
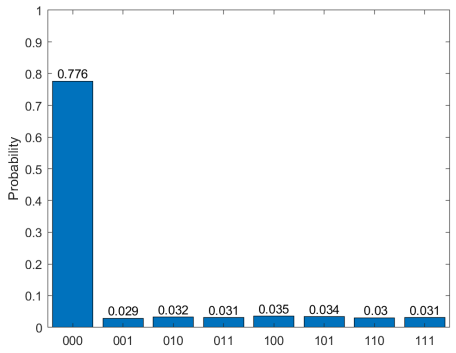
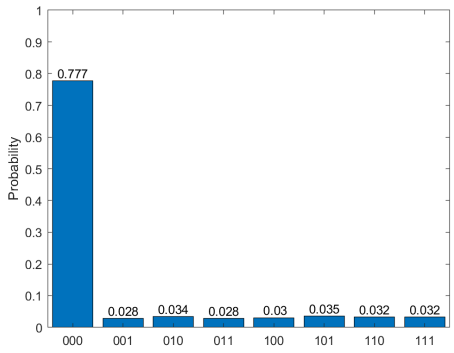


Figure 4.10: QPE histogram results.

As seen in Figure 4.10, the basis encoded result of the quantum circuit is the expected '000' weight string. This histogram represents a collection of 'shots' computed on the quantum circuit by the quantum simulator. Each shot is an execution of the circuit with artificial noise added. On each measurement of the quantum state, the shot is collapsed into one of these eight states. The state that it collapses into is dependent on both the noise added for that shot and the probability of ending up in that state denoted by the state-vector of the circuit. For this test, 4096 shots were used with this circuit, but converted to a fraction. The same test will reproduced using the circuit designs developed in this thesis and compared to Figure 4.10 for accuracy.



(a) QPE simplified histogram results.



(b) Register Counting histogram results.

Figure 4.11: 3-1 Histogram results

As seen in Figure 4.11, the results are very similar to what was observed in Figure 4.10. It can be seen that the amplitude of the results in the three cases are slightly different. This is the result of the Qiskit Aer simulator that adds a small amount of artificial noise during each shot. It was found that all three implementations result in the exact same output. The amplitude is centered around 0.78125 for the correct circuit output.

While all three circuits are able to produce the same output, it is important to examine the size differences.

QBNN Training Implementation	**Width**	**Depth**
QPE	12	2032
QPE Simplified	12	880
Register Counting Method	13	356

Table 4.8: Initial testing circuit width and depth results.

For each circuit tested in the 3-1 Neuron example, a total of 12 qubits were used, except for the RC implementation that required 13, as seen in Table 4.8. The depth of the quantum training circuit is reduced greatly when using the QPE Simplified and Register Counting Circuits. This proves that the work of this thesis can produce

the same NN training results on a much smaller quantum circuit.

4.5 2x2 Convolution Filter Results

After proving the functionality of the circuits in Section 4.4, the more practical example of 2x2 convolutional edge detection filter results were tested. For these experiments, more hardware resources were required than was was available on local machines. Rochester Institute of Technology's Research Computing Cluster [4], was used to access more memory and computation time needed for these circuits.

During initial testing of these circuits, the original setup consisted of generating an output histogram plot similar to what was shown in the previous section, Section 4.4. Upon running the circuit and obtaining the results, the following output was seen.

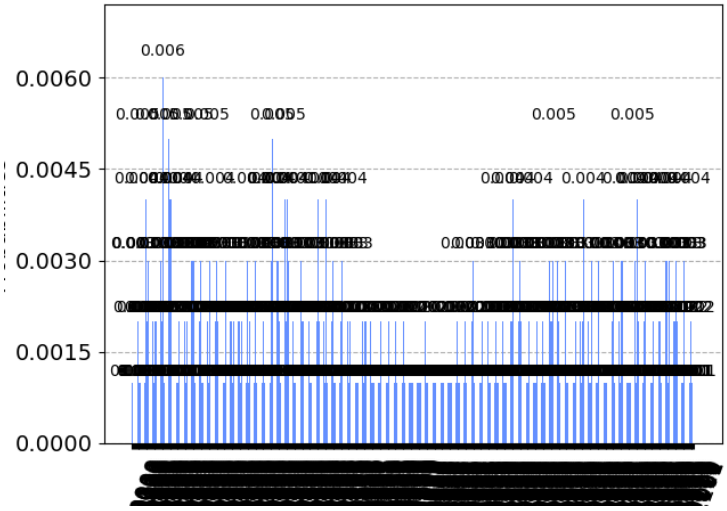


Figure 4.12: Illegible 2x2 convolution edge detection histogram results.

Due to the fact that there are 10 weights involved in this circuit, the number of possible measured states are 2^{10} or 1024. Not only does this number of outputs produce an illegible output plot, but the number of shots required to show a decent histogram would result in a very long circuit runtime. Due to the fact that the

histogram is a collection of the number of shots that collapse into each state, a large amount of shots would need to be run in order to produce a histogram with good results. Specifically, a decent multiple more that 2^10 shots must be simulated to ensure that the shots are collapsing enough times to see any states that are more probable than others.

A different method was then used to obtain the results from larger circuits. Instead of running multiple shots and measuring the states, the statevector of the circuit was calculated using the Qiskit statevector simulator[22]. The statevector of the quantum circuit will be a vector calculated by applying matrix representations of quantum gates to a zero vector. The process of applying the matrix representation of quantum gates is the mathematically intensive part of the quantum simulation. This is because for each layer in the circuit, a matrix of size $qubits * qubits$ must be multiplied against a statevector of size $1 * qubits$. The number of layers in a circuit is approximately the depth of the circuit. The result after computing each layer is a vector of complex values, representing the state amplitudes. If the circuit contains any number of measurement gates, the circuit statevector will then be collapsed on those specified qubits.

It was found during testing that the use of measurement gates using Qiskit's statevector simulator resulted in extremely long runtimes that did not finish. However, the removal of the measurement gates results in a statevector of the complete circuit and runtime of a couple hours. This vector can then be artificially collapsed on certain substates, which is a state made of part of the complete circuit state, by summing the magnitude squared of each equivalent substate. An example of the artificial collapsing is shown in Table 4.9.

State	Substate	Complex Value
0011100	111	0.25+0j
1011101	111	0.25+0j
0010100	101	0.125+0j
0110111	101	0.375+0j

Table 4.9: Substate collapsing process example. The states on the left hand side represent the full state of the complete quantum circuit. However, if only the middle three qubits are needed, the magnitude of each complex state that has the middle three qubits in the same states need to be summed together.

In the example shown in Table 4.9, the full circuit possible states are assumed to only be the binary strings shown in the left column. If it is desired to collapse those states, or essentially measure those states on the center three qubits, the two possible substates are then '111' and '101'. To find the probability of the circuit falling into those states, the magnitude of each complex number is found, squared and added to each of the same substate. The result for the example shown in Table 4.9 is a probability of 0.5 for '111' and 0.5 for '101'. A Python script was developed to perform this operation on any full statevector output. This could then be used to extract the probabilities of the 2x2 Convolutional filter results without using the histogram or the measurement gates.

The test involved the 4-2-1 neural network that utilized the same number of qubits in order to properly compare all of the circuit implementations. The size of the circuits were calculated for depth comparison.

QBNN Training Implementation	Width	Depth
QPE	28	41714
QPE Simplified	28	6770
Register Counting Method	28	1620

Table 4.10: 4-2-1 Network width and depth

As seen in Table 4.10 and in previous observations, the use of the circuits designed by this thesis result in circuits that are much smaller than the one originally proposed by [1]. The results of all three implementations correctly found the best weight strings for the 4-2-1 NN and the dataset shown in Figure 3.2.

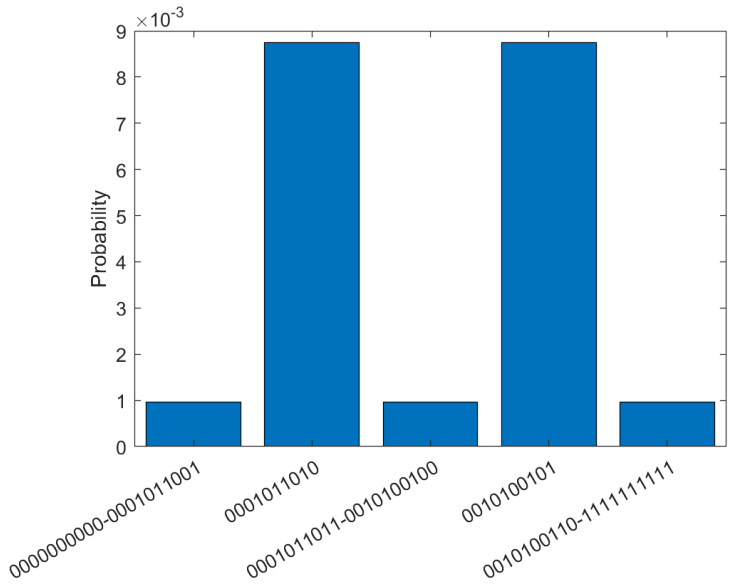


Figure 4.13: 2x2 Convolution histogram results. Two states have a much higher probability than all other states, meaning that those are the two resulting weight strings. All states that are not the two amplified states are represented by the lower bars. Each state has a probability equal to those bars.

The probability outputs found by the collapsing python script are shown in Figure 4.13. All of the states or weight strings that are less probable are summarized into the three separate smaller histogram bars. All three implementations gave these results and showed that the most optimal weight strings are '0001011010' and '0010100101'. These two weight strings result in correctly identifying 16 out of 16 dataset entries. Utilizing another Python script that tests all of the possible weight strings on the neural network classically, these results were found to be accurate and the only two weight strings that result in perfect vertical edge detection. While in classical machine learning, there is a separation between the training dataset, and testing dataset, here, there is only a combined training and testing dataset due to the small dataset. This

results in overfitting and the potential of being unable to identify new data.

QBNN Training Implementation	Time to Run (hrs)
QPE	13:23:50
QPE Simplified	2:46:39
Register Counting	1:22:05

Table 4.11: 2x2 Filter training timing results

For each test case, the runtime of the quantum circuit was recorded to show how the depth of the various circuits effect the training time. The results of the three versions are shown in Table 4.11 where the last two rows are the circuits developed in this thesis. It can be seen that the much smaller circuit sizes result in a much faster computation time. While this does improve the performance of quantum simulation on classical computers, the results translate into the quantum realm as well. First, quantum computers will need to perform less gate applications on qubits when the circuit depth is less, which reduces the runtime of the quantum computer. Second, the smaller depth of quantum circuit produces less noise and error than longer circuits.

4.6 3x3 Convolution Filter Results

Circuits utilizing all three training implementations were developed to test the 9-3-1 Neural Network. The QBNN circuits were setup to be trained using the dataset seen in Figure 3.3. There was a total of 32 qubits used for all three implementations in this case. The QPE and QPE simplified circuits could have been reduced however they were kept at 32 to be able to compare against the Register Counting Implementation. There is a total of 12 weight values in this circuit, meaning that there are $2^{12} = 4096$ weight strings that this circuit will evaluate. Similar to the 2x2 edge detection example, the histogram output of this circuit is illegible and is difficult to determine the correct weight strings. This is again due to both the large number of output

possibilities and the inability to run an adequate amount of shots for the result. This issue was once again fixed by utilizing the python script developed to measure the weight strings. The results of the training are shown in Figure 4.14.

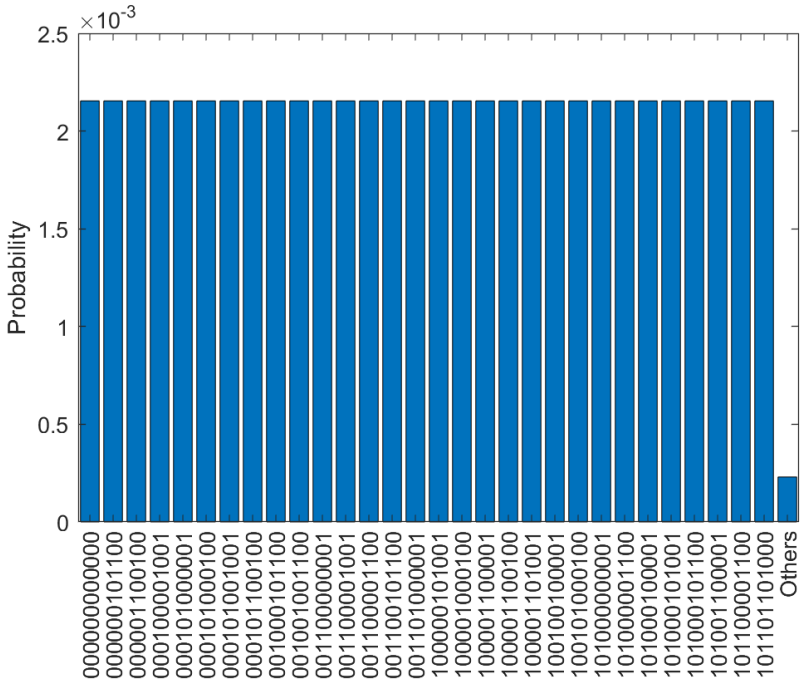


Figure 4.14: 9-3-1 Neural network weight strings. These 30 weight strings are able to correctly identify all of the edges in the dataset. The 'others' bar represents the probability of each weight string that was not amplified.

As seen in Figure 4.14, the training algorithm for both the Improved QPE circuit and Register Counting circuit resulted in 30 weight strings that produce a 100% accuracy of the training dataset. A bar named 'Other' is added to show the probabilistic differences between the amplified weight strings and the non-amplified weight strings. The amplified weight strings are about 9.409% more probable than all the non-optimal weight strings. When observing the resulting weight strings classically, the weight strings $[0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1]$ and $[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$, which are two of the weight strings produced in Figure 4.14, gives a 96.09% accuracy when testing against the complete dataset of 512 entries. This test therefore shows the potential of these training methods to work well for larger and more practical problems.

The scalability and runtime of the various circuit implementations are shown 4.12a.

QBNN Training Implementation	Width	Depth
QPE	32	41702
QPE Simplified	32	6758
Register Counting Method	32	1608

(a) Scalability

QBNN Training Implementation	Time to Run (hrs)
QPE	∼10 Days
QPE Simplified	∼48 hrs
Register Counting	22:02:49

(b) Runtime

Table 4.12: The scalabilty and the runtime of the 9-3-1 neural network training methods. The QPE method described in [1] was not able to complete within the allowed runtime of the research cluster [4]. This shows the benefits of using the designs developed in this thesis.

Table 4.12 shows scalability and runtime results of the 9-3-1 neural network. The scalability of the circuit can be seen to be around the same size as the circuit used for the 2x2 edge detection. This is because the 9-3-1 network is not a fully connected network and does not use as many weights. It is important to notice that the original QPE circuit designed in [1] was not able to complete its test as the time limit of the testing cluster[4] was reached. This improves the potential of the designs proposed in this thesis as they are able to train the QBNN in much less time than the original circuit proposed in [1].

Sensitivity	1.0
Specificity	0.960784
Precision	0.090909
Accuracy	0.960938
F1 Score	0.166667

Table 4.13: Statistical classifications of the 3x3 vertical edge detection test

As seen in Table 4.13, the F1 score of this test is very low. The F1 score is extremely low in this case due to the dataset imbalance. The dataset of all possible pixel values is 2^9 or 512 with only 2 cases that are considered a vertical edge and 510 cases that are not considered a vertical edge. The imbalance causes the precision of the model to be low and therefore the F1 score will be low. To fix this issue, a better dataset or larger NN could be used to improve the confusion matrix of these results. The confusion matrix is shown in Figure 4.15 below for reference.

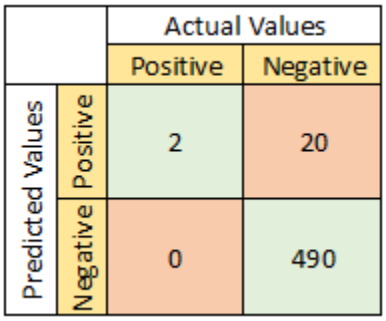


Figure 4.15: 3x3 Vertical Edge Detection Confusion Matrix

The low F1 score and precision seen in Table 4.13 is a result of the dataset and neural network structure. The low scoring results are found in both the quantum and classical implementation of this binary neural network and do not define the performance of the quantum algorithm.

Chapter 5

Conclusion

5.1 Conclusion

This thesis was able to design and improve upon a Grover's Quantum Binary Neural Network that was originally proposed by [1]. A design was made based on a idea continuation from [1] that utilized a Register Counting method as apposed to the Quantum Phase Estimation method. Improvements to the original design found in [1] was made, and an improved version that simplifies the Quantum Phase Estimation algorithm was also implemented. This thesis was written to help explore the field of quantum machine learning and provide some more practical demonstrations that show the potential of quantum neural networks and their use. This was done through the contribution of two new quantum binary neural networks, and their application on pixel edge detection models.

The first contribution involved the improvement of the design found in [1]. This included the removal of an unnecessary qubit that was utilized for applying a phase to correctly classified dataset entries. This was possible due to the functionality of the quantum phase gate that treats the target qubit as an additional control bit. A fix was also implemented in regards to the quantum phase estimation algorithm. The setup suggested in [1] goes against standard practices when implementing the Quantum Fourier Transform and would result in incorrect results.

The second contribution is the design of a training circuit that was proposed, but not designed in [1]. This circuit utilized a Register Counting method that replaces the Quantum Phase Estimation qubits with a set of register qubits that will store a binary encoding of the number of times a weight string produces a correct result. This removes the need to encode the number of correct outputs in the phase of the weight string, and therefore removes the need to perform the Quantum Phase Estimation algorithm. With this circuit the count of correct outputs can be read after training each dataset entry. This is then able to reduce the depth of the training algorithm for the Quantum Binary Neural Network greatly. However, the width of the qubit register grows with the size of the training dataset and can results in a higher number of qubits when compared to the QPE algorithms. Depending on the problem, or on the capabilities of the quantum hardware, the increase in width may be more desirable than the depth of the circuit. Due to the extreme depth that can be seen from these circuits, the circuit run time and error rates will be much higher with a higher depth.

An improvement was made based on the original circuit design from [1]. This design is the Improved Quantum Phase Estimation training algorithm and utilized a shortcut in the Phase Estimation algorithm that was possible due to the nature of the QBNN training process. Due to the fact that the phase applied to the weight string is known, the repeated unitary circuits that are part of the QPE algorithm can be replaced with controlled unitary QBNNs that apply half of the phase for each successive QPE register qubit. This implementation was able to cut the depth of the original circuit seen in [1] by at least 50% in most cases. This will allow for the benefits of a reduced number of qubits, and remove the drawback of a large circuit depth.

The product of this thesis resulted in two implementations that perform better than the implementation in [1]. The implementations were tested on small neural networks as well larger practical circuits. The 2x2 edge detection networks showed

a practical implementation that utilized a larger circuit than what was found in [1]. While the training of this 4-2-1 neural network provides two weight strings with 100% accuracy, the small dataset does not allow new, unseen data, to be tested on the model. This means that the accuracy of the network on new data that is not part of the dataset was not able to be tested. The use of the 3x3 edge detection neural network utilized an even larger circuit than the 2x2 networks and allowed for a much larger possible dataset. The 3x3, or 9-3-1 network, was trained using only 16 data points that resulted in 30 possible weight strings that gave 100% accuracy to the dataset. When testing these weight strings against the complete possible dataset, it was found that 2 weight strings result in an accuracy of 96.09375%. While this is still a limited dataset and problem, it shows that the training of the neural network was successful and the potential for this quantum machine learning method is very promising. To improve the results of this test, a larger NN could be used as well as increasing the number of edges in the dataset.

The utilization of these approaches on real quantum hardware would allow for an accelerated training of globally optimal binary neural network weights compared to classical training methods. This is because the quantum computer is able to assess every possible weight string all at once due to the properties of superposition. As real quantum computers improve in size and reduce noise, the new implementations designed in this research can be used to train larger problems faster than the training time of classical training algorithms.

Future work could be done using the designed implemented in this thesis. One very important direction to research is the implementation of other classical machine learning methods to be used on this quantum training technique. Some classical topics that could be adapted to this model is the use of various activation functions. There may be a way to implement non-threshold activation functions or even new activation functions that could provide benefit to quantum neural network implementations.

Another classical machine learning topic that is important for this specific area of ML are methods to reduce over-fitting in Binary Neural Networks. Various methods could be adapted to quantum machine learning using the training techniques discussed in this thesis. Finally, an quantum training method that uses a non-binary neural network implementation could be developed and tested. This could potentially allow for other practical models to be implemented in quantum hardware.

Bibliography

[1] Y. Liao et al, "Quantum speed-up in global optimization of binary neural nets," *New J. Phys.*, vol. 23, no. 063013, 2021.

[2] "Quantum phase estimation," https://learn.qiskit.org/course/ch-algorithms/quantum-phase-estimation, accessed: 2023-11-21.

[3] "Grover's algorithm," https://learn.qiskit.org/course/ch-algorithms/grovers-algorithm, accessed: 2023-11-30.

[4] Rochester Institute of Technology, "Research computing services," 2022. [Online]. Available: https://www.rit.edu/researchcomputing/

[5] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016. [Online]. Available: http://arxiv.org/abs/1602.02830

[6] Qiskit contributors. (2020) Learn quantum computation using qiskit. [Online]. Available: https://qiskit.org/textbook/

[7] A. Y. Kitaev, "Quantum measurements and the abelian stabilizer problem," 1995.

[8] D. Coppersmith, "An approximate fourier transform useful in quantum factoring," 2002.

[9] A. Pavlidis and E. Floratos, "Arithmetic circuits for multilevel qudits based on quantum fourier transform," *arXiv preprint arXiv:1707.08834*, 2017.

[10] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134.

[11] L. K. Grover, "A fast quantum mechanical algorithm for database search," 1996.

[12] A. R. Haverly, "A comparison of quantum algorithms for the maximum clique problem," Master's thesis, Rochester Institute of Technology, 2021.

[13] A. Haverly and S. L. Alarcon, "A comparison of quantum algorithms for the maximum clique problem," Nov 2021. [Online]. Available: https://medium.com/xanaduai/a-comparison-of-quantum-algorithms-for-the-maximum-clique-problem-4cd8984cea59

[14] D. Fernandes and I. Dutra, "Using grover's search quantum algorithm to solve boolean satisfiability problems: Part i," *XRDS*, vol. 26, no. 1, p. 64–66, sep 2019. [Online]. Available: https://doi.org/10.1145/3358251

[15] D. Fernandes, C. Silva, and I. Dutra, "Using grover's search quantum algorithm to solve boolean satisfiability problems, part 2," *XRDS*, vol. 26, no. 2, p. 68–71, nov 2019. [Online]. Available: https://doi.org/10.1145/3368085

[16] S. A. Cook, "The complexity of theorem-proving procedures, stoc'71: Proceedings of the third annual acm symposium on theory of computing," 1971.

[17] M. Fürer, "Solving np-complete problems with quantum search," in *LATIN 2008: Theoretical Informatics*, E. S. Laber, C. Bornstein, L. T. Nogueira, and L. Faria, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 784–792.

[18] M. Schuld, I. Sinayskiy, and F. Petruccione, "An introduction to quantum machine learning," *Contemporary Physics*, vol. 56, no. 2, pp. 172–185, oct 2014. [Online]. Available: https://doi.org/10.1080%2F00107514.2014.964942

[19] S. Lloyd, M. Mohseni, and P. Rebentrost, "Quantum algorithms for supervised and unsupervised machine learning," 2013.

[20] M. Hoffnagle, S. L. Alarcon, C. Merkel, S. Ly, and A. Pozas-Kerstjens, "Accelerating the training of single layer binary neural networks using the hhl quantum algorithm," 2022.

[21] S. Gupta and R. K. P. Zia, "Quantum neural networks," 2002.

[22] Qiskit contributors, "Qiskit: An open-source framework for quantum computing," 2023.

[23] "Ibm quantum (2023). ibm quantum lab," https://quantum-computing.ibm.com/lab, accessed: 2023-2-19.