MACHINE LEARNING BY ADIABTIC QUANTUM COMPUTING: AN ADIABATIC QUANTUM NEURAL NETWORK

by

Erick Serrano

Honors Thesis submitted in partial fulfillment
for the designation of Research and Creative Honors

Department of Physics & Astronomy
Dr. Bernard Zygelman, Dr. Maria Jerinic-Pravica, Dr. Brendan Morris
College of Sciences, College of Engineering

University of Nevada, Las Vegas
October 2021

ABSTRACT

MACHINE LEARNING BY ADIABTIC QUANTUM COMPUTING: AN ADIABATIC QUANTUM NEURAL NETWORK

by

Erick Serrano

⟨Bernard Zygelman⟩, Examination Committee Chair
Professor of Physics and Astronomy
University of Nevada, Las Vegas

With the surge in popularity of machine learning, many researchers have sought optimization methods to reduce the complexity of neural networks – commonly known for its many engineering applications. Over the past decade, researchers have attempted to optimize neural networks via quantum algorithms, and in this thesis, we outline steps to attempt the same.

We first describe the feed-forward neural network (FFNN) "training process" and its respective time complexity. We highlight the inefficiencies of the FFNN training process when using gradient descent and introduce a call to optimize an FFNN. Afterward, we discuss the strides made in quantum computing to improve the time complexity of machine learning; we study Edward Fahri's Adiabatic Quantum Computing (AQC) algorithm, which is a precursor to the well-known Quantum Approximate Optimization Algorithm (QAOA). We then simulate AQC to find the "ground

state" of an arbitrary hamiltonian; we find we can connect the hamiltonian ground state to a cost function, but this is difficult for complex systems.

We finally use the AQC simulation to replace the back-propagation in an FFNN. We first describe the labeled 5-bit dataset we used. We analytically derive the gradient formula for a one-layer FFNN – also known as a perceptron; we then translate it to a hamiltonian. We denote the derived hamiltonian as the "trivial" cost hamiltonian for minimizing a given cost function; this hamiltonian does not take advantage of the benefits of AQC, and it, therefore, does not reduce the complexity of the FFNN. However, the derivation of the cost hamiltonian is simple – hence "trivial"– and the simulated AQC quantum neural network (AQC-QNN) proves the possibility of merging AQC to replace the learning of a typical FFNN.

ACKNOWLEDGEMENTS

I wish to acknowledge the support of the UNLV honors college for the encouragement to continue my work through the Honors thesis program. I would personally like to thank my thesis committee members, Dr. Bernard Zygelman from the department of Physics & Astronomy, Dr. Maria Jerinic from the Honors College department, and Dr. Brendan Morris from the Department of Electrical and Computer Engineering.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The neural network model is a machine learning algorithm that classifies data and allows accurate and precise predictions. Based on that data, a Feed Forward Neural Network (FFNN) has many applications in the real world; for instance, an FFNN can optimize the usage of air conditioning, heat pumps, and refrigeration[21] and general electronic motors [5]. However, we will reveal how an FFNN can take extremely long periods of time to classify large sets of data.

We then discuss how a quantum algorithm may be the key to reducing the run-time of the FFNN. We introduce quantum mechanics in the scope of quantum computing; namely we introduce quantum parallelism and interference, qubits and operators, and finally Hamiltonians and quantum algorithms.

Due to the inefficiency of an FFNN, we were driven to focus on quantum *optimization* algorithms. We pay close attention to both the adiabatic quantum computing algorithm (AQC) or the quantum approximate optimization algorithm (QAOA). The AQC[13] and QAOA[12] algorithms are used to find the "ground-state" of a function; a properly encoded function should reveal its global minima/maxima after using the AQC and QAOA algorithms. Both algorithms may reveal a minimized cost function

in linear time[1].

We introduce both AQC and QAOA toward the end of the introduction. Then we move toward the experimental AQC-based QNN.

Feed Forward Neural Network

The science of the standard FFNN model is defined by classical computation; at the highest level, an FFNN "trains" by the following steps (in sequential order):

1. Forward Propagation
2. Cost Measurement
3. Backpropagation
4. Parameter Updating (Gradient Descent)
5. Repeating (1-4) (Gradient Descent)

We execute these five steps under the following assumptions: 1) We exclude data processing, where we convert a training example (say, an image) to raw data (RGB values), 2) The data we receive is – mostly – labeled, 3) The FFNN uses a single output node for binary classification, and 4) The FFNN uses gradient descent, which, at the cost of run-time, dramatically improves the accuracy of an FFNN. Figure 1.1 is an example of the architecture of a FFNN.

[1]We should also consider the time it takes to *encode* the cost function, but we are able to neglect this due to the small training data we used for our experiment.

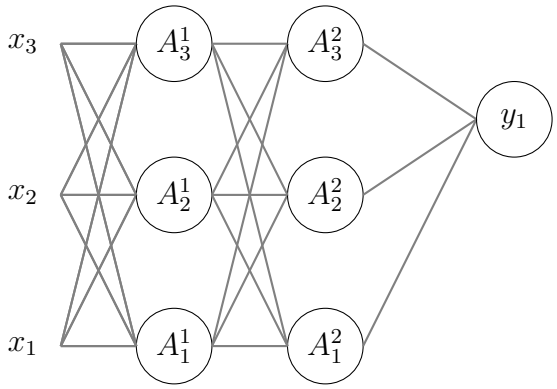

Figure 1.1: The figure is an example of a standard FFNN with one input, hidden, and output layer. The number of layers and the nodes per layer are arbitrary; for example, the output layer y_1 can have multiple nodes $y_1, y_2, ...y_k$. The FFNN processes three data points – or three training features – x_1, x_2, x_3.

One common application of FFNNs is image classification[11], where an FFNN identifies patterns and objects in the image; we previously trained six FFNNs to identify viral or bacterial infections in an image[25]. While the accuracy of each FFNN was satisfactory, we discovered that image classification of an FFNN required prodigious CPU resources. Image classification with an FFNN is known to have an exponential run-time that is dependent on the resolution and dimensions of the image[11]. We roughly generalize the run-time of an FFNN using steps (1-5), which we discuss in the following sections.

Sub-par Performance of a FFNN

An FFNN executes four steps iteratively to become an accurate classifier. Before we incorporate all four steps into a single run, we must understand the time complexity of each step. Let's begin with forward propagation.

Run-time of Forward Propagation

Forward propagation is the process where raw data $x_1, ...x_n$ is converted to a set of predictions, (or in our case, a single prediction y_1). One suspects the run-time is affected by the raw data size and the architecture of the FFNN. Specifically, the complexity depends on:

1. Input features
2. Training Examples
3. A single node
4. The number nodes in a layer
5. The number of layers per neural network

We begin with the input features. From figure 1.1, we see that our input features, x_1, x_2, x_3, are of size $n = 3$. In general, input data can become much larger, so we generalize to n input features.

If our FFNN were a single node, then we would make two calculations: the linear function calculation and the activation function calculation. The linear function is characterized by

$$f_k(a, w, b) = \sum_{i}^{n} w_i a_i + b_i \quad \text{(Forward Propagation)} \tag{1.1}$$

for the k-th node. w_i, b_i are parameters, known as *weights and biases*, that belong to a node, and a_i is an "input feature" of the preceding "layer" with n features total;

in other words, we need n weights and n biases to make n calculations, so we define the time complexity in asymptotic notation as

$$\mathcal{O}(n)$$

In sequence with the linear function, we also use the activation function, $z(f)$, which we characterize using the sigmoid function:

$$z_k(f_k(a, w, b)) = \frac{1}{1 + e^{-f_k(a,w,b)}} \quad \text{(Activation Function)} \tag{1.2}$$

So for every input feature, we need to 2 calculations – one for the linear function, f_k and one for the activation function, z. That means the time it takes to complete our calculations increase by $n + 1$, and our time complexity is then

$$\mathcal{O}(n + 1) \approx \mathcal{O}(n) \quad (\textit{Time Complexity of 1 node})$$

for one node. The notation $\mathcal{O}$, denotes the asymptotic complexity of an algorithm, or the number of elementary steps needed to complete an algorithm, generalized to infinity[30]. This notation allows us to observe the limits of the algorithm only, excluding the hardware and software limitations.

For *multiple nodes* we must describe the repetition of calculations by introducing

a factor of j. This added factor summarizes the complexity of a "layer" of nodes:

$$\mathcal{O}(nj) \quad (\textit{Time Complexity of multiple nodes, 1 layer})$$

Now we consider multiple layers of an FFNN. When we calculate the activation function for a single node, $z_k(f)$, this value is passed into to the next layer as the value a_i. We traverse "deeper" through an FFNN by passing the activation functions $z_1^l(f)..., z_k^l(f), ...z_l^l(f)$, where l is the layer of an FFNN, through multiple layers. The procedure is repeated until we reach the final layer L. With a more transparent notation, we replace our equation with

$$\sum_{l=0}^{L-1} j^l j^{l+1}$$

where j indicates the number of nodes per layer l. We note that $j^0 = n$ for the number of input features $x_1, ...x_n$ and $j^L = 1$ for our output layer y_1; we will only be considering *binary classification* for our FFNN. The total complexity evolves with the previous equation as

$$\mathcal{O}(\sum_{l=0}^{L} j^l j^{l+1}) \quad (\textit{Time complexity of forward prop.})$$

An FFNN provides an abundant amount of freedom to a programmer. We can create unique sizes for every layer of a neural network, and we can also create several layers – or zero layers. The freedom of and FFNN implies a unique and unrelated time

complexity for each layer of nodes.

Run-time of Cost Measurement

After forward propagation for a single training example, we should make use of two values: z^i and y^i. z^i is the activation function at the final layer, which we call the *prediction* of the neural network, and y^i is the pre-defined *label* at the output layer [2]

To gauge the complexity of the cost function for a neural network, we consider the following equation for the cost function J:

$$J = \frac{1}{m}(\sum_{i=1}^{m} y^i log(z^i) + (1 - y^i) log(1 - z^i) = \frac{1}{m}(\sum_{i=1}^{m} J^i) \quad \text{(Cost Function)} \qquad (1.3)$$

We repeat this for every *training example* $1...i...m$ with n input features. We previously only considered a single training example, but the cost function takes the predictions of m training examples to make a final measurement of accuracy. Since the cost function is dependent on the number of training examples, m, then solving this equation will take m steps and have a total complexity

$$\mathcal{O}(m) \quad (\textit{Time complexity of Cost Measurement})$$

[2] Do not confuse this with the previous section's superscript l and L; these values were used to define the number of layers in a neural network. The superscript i is used to indicate a *training example*, i.

Run-time of Backpropagation

We now consider the complexity of backpropagation. Given our cost function J, we now need to modify all parameters w_i^l, b_i^l for node i in layer l. Backpropagation, as the name suggests, starts at the output node of the neural network and computes the derivative, or the *gradients* $\frac{\partial J}{\partial z}$, $\frac{\partial J}{\partial w}$ and $\frac{\partial J}{\partial b}$, associated with the weights and bias, w_i^l, b_i^l (also known as the parameters), and our input feature z_i^l for node i and layer l. However, we already did much of the complexity analysis using forward propagation, which we will reference later.

For multiple layers and multiple nodes, we must generalize our three equations to the *activation gradient* dz^l,

$$dz^l = (W^{l+1,T} * z^{l,T})/m \quad \text{(Activation Gradient)} \tag{1.4}$$

the *weight gradient* dw^l,

$$dw^l = (dz^{l+1} * z^{l,T})/m, \quad \text{(Weight Gradient)} \tag{1.5}$$

and the *bias gradient* db^l,

$$db^l = \sum dz^l/m, \quad \text{(Bias Gradient)} \tag{1.6}$$

For a list of gradients dz, dW, db in layer l[3]. In addition, T indicates a matrix trans-

[3]For simplicity we replaced $\frac{\partial J}{\partial}$ for the simple derivative symbol, but keep in mind

pose.

At the output layer, dz^L, we can also analytically find the activation gradient to yield the following equation:

$$dz^L = -(\frac{Y}{z^L} - \frac{1-Y}{1-z^L})$$

for some label Y and our prediction z^L. We no longer rely on the weight of the next layer, as there are no weights after the output layer.

These three equations (and the final equation at the output layer), in combination with the fact that there is a 1-to-1 correspondence with the number of gradients and parameters, our complexity then evaluates to

$$\mathcal{O}(\sum_{i=0}^{L} j_i j_{i+1}) \quad (\textit{Time complexity of Back. Prop.})$$

Which is equivalent to same as forward-propagation, with j nodes for layer i.

Run-time of Parameter Updating

Armed with the gradients and parameters, we update every weight and bias with the corresponding gradients. The time complexity of this portion is trivial and is

---

that these two interpretations are equivalent

measured by

$$\mathcal{O}(\sum_{i=0}^{L} j_i j_{i+1}) \quad (\textit{Time complexity of Param. Updating})$$

where we update two parameters per node, w_i, b_i, for $j_i j_{i+1}$ nodes up to layer L.

Run-time of Gradient Descent

As previously stated, gradient descent implements all four previous steps – forward propagation, cost measurement, backpropagation, and parameter updating – iteratively. In other words, we sum the complexities in each process and multiply it by some number of iterations g so that

$$g(2\sum_{i=0}^{L} j_i j_{i+1} + m + 3\sum_{i=0}^{L} j_i j_{i+1} + 2\sum_{i=0}^{L} j_i j_{i+1})$$

is the total number of steps to train an FFNN. We can simplify this and create an asymptotic complexity such that

$$\mathcal{O}(g(\sum_{i=0}^{L} j_i j_{i+1} + m)) \quad \text{FFNN with Grad. Desc.}$$

This is our entire complexity for training an FFNN. We could simplify our equation further by evaluating the summation; however, this is not important for the subject. Our complexity analysis of each FFNN phase is tabulated in table 1.1.

The complexity is not desirable, since a large input size ($j_0 >> 1$) will dramatically

Table 1.1: This table details the time complexity, in Big $\mathcal{O}$ notation, of an FFNN during the training process. j_i is the number of nodes in layer i, up to the final layer L, g is the number of iterations, and m is the number of training examples we consider. We are cautious in removing "small" values, as an FFNN can vary in size.

Phase	**Complexity**
Forward Propagation	$\mathcal{O}(\sum_{i=0}^{L} j_i j_{i+1})$
Measuring the cost function	$\mathcal{O}(m)$
Backpropagation	$\mathcal{O}(\sum_{i=0}^{L} j_i j_{i+1})$
Parameter Updating	$\mathcal{O}(\sum_{i=0}^{L} j_i j_{i+1})$
FFNN with Gradient Descent	$\mathcal{O}(g(\sum_{i=0}^{L} j_i j_{i+1} + m))$

increase the time it takes to train a neural network. In addition, a large number of training examples $m >> 1$ can also dramatically lengthen the training process. We conclude an FFNN that is implemented via gradient descent has a non-polynomial time complexity, which highlights the inefficiency of the neural network as it scales with training examples m, the initial input features $j_0 = n$, and the number of iterations. This inefficiency presents an opportunity to begin optimization. A linear algorithmic complexity ($\mathcal{O}(n)$) or less has long been the goal of researchers seeking improved Neural Network models. The goal of many researchers and this thesis is to answer the following question: can a neural network classify complex data in polynomial time? With quantum computing, we attempt to provide an answer to this question.

In the following sections, we provide an introduction for quantum computing, AQC, and QAOA. We then detail an implementation of an AQC-based FFNN – defined as an AQC quantum neural network (AQC-QNN). Finally, we look towards

improving the AQC-QNN by instead applying the QAOA algorithm and/or by modifying the "problem Hamiltonian."

Quantum Computing Basics

In this section, we discuss the key elements of quantum computing. We attempt to encapsulate the significance of quantum computing with a few key concepts:

1. The qubit and superposition
2. Multiple qubits
3. Quantum gates, inner products, and outer products
4. Matrix representation of qubits and operators

The qubit provides us the capability of applying quantum mechanical effects toward information, and we will describe its notation and how it differs from a classical bit. Then, we move toward defining the notation for multiple qubits in a system, followed by a description of the operators and quantum gates. We will provide a brief explanation to the quantum operator known as the Hamiltonian – we applied this operator in our experiment. Once we finish our brief overview, then we will translate our notation into matrices; it is simpler to translate into classical simulation.

The qubit and superposition

Consider the original bit, which is the simplest form of containing information in modern computing. The notation for a single register will be defined as

$$[s]$$

Where s is the current state the register [] is in. For a bit, s belongs only to two values, $s \in \{0, 1\}$. Note that s can only occupy one value at a time. Either s is defined by

$$[s] = [1] \quad s \neq [0]$$

or s is defined by

$$[s] = [0] \quad s \neq [1]$$

It seems trivial to be adding the second non-equivalence; clearly, s can not be equal to zero if s is equal one (and vice versa). With quantum information however, that is not necessarily the case – which is why we make the distinction. Consider instead a quantum register – or a qubit – of information, which is defined as

$$|s\rangle$$

$|s\rangle$ is the state of the qubit in bra-ket notation[32]. In a similar fashion as the classical bit, the state s belongs to a binary set of values $s \in \{0, 1\}$, so then

$$|s\rangle = |1\rangle$$

and

$$|s\rangle = |0\rangle$$

Which we can observe upon measurement. However, we do not imply the second inequality as we did with a classical qubit. In fact, we find that s could be represented as

$$|s\rangle = |0\rangle + |1\rangle$$

Where s is *sometimes* equal to 1 and sometimes equal to 0 upon measurement. This is unlike anything we have seen in classical computation; a qubit is now capable of displaying either of the binary states upon measurement without any intentional change in information[32]. We now introduce the superposition principle, in which we assume that the state of the qubit can be *equal to one and zero* at a single point in time. This is a key-element of the quantum information; with superposition for example, we can apply a function toward one and zero at the same time instead of sequentially (as a classical computer would do).

Multiple qubits

Armed with the information of a single qubit and superposition, we now introduce the concept of multiple qubits in a system. Consider two qubits in a single state $|0\rangle$ and $|0\rangle$. Then the proper representation is defined as

$$|0\rangle \otimes |0\rangle$$

Where $\otimes$ is the *direct product* of qubits. We can simplify this notation to instead forgo the direct product (knowing that its presence still remains).

$$|00\rangle$$

A similar procedure applies for the state $|01\rangle$, $|10\rangle$, and $|11\rangle$. Note that with two qubits, we suddenly have $2^2 = 4$ possible states. As some may guess, superposition allows us to also create the state

$$|\psi\rangle = (|0\rangle + |1\rangle) \otimes (|0\rangle + |1\rangle) = |00\rangle + |01\rangle + |10\rangle + |11\rangle$$

Where we replace s with ψ. So now, upon measurement, the function will *sometimes* yield any one of four possible values. This applies for even more qubits: for three qubits, we have a superposition of $2^3 = 8$ values; for four qubits we have a superposition of $2^4 = 16$ values; for any number of qubits n, we conclude that we could have a superposition of 2^n values. In terms of advantage, it is clear that there is an exponential spacial advantage that scales with the number of qubits; this concept is known as **quantum parallelism**. A classical computer is capable of parallelism with tasking multiple computers to a single task[9], but quantum parallelism fundamentally differs in that the qubit is inserting all combinations of input into the problem. Quantum computing is commonly hailed as the future of computing due to quantum

parallelism, and it is this defining concept that has been employed by fast quantum algorithms which we will discuss later.

Quantum Gates, Inner Products, and Outer Products

Now that we have introduced the qubit and its superposition states, we can consider the quantum operators, or gates. The analog to these operators in classical computing could be addition, subtraction, etc... but this does not entirely justify the capabilities and limitations of a gate. Gates are used to modify and measure the qubit at any point in time. Consider one of the most common gates, known as the Hadamard gate[32]:

$$H = \frac{1}{\sqrt{2}}(|0\rangle\langle 0| + |1\rangle\langle 0| + |0\rangle\langle 1| - |1\rangle\langle 1|) \tag{1.7}$$

The Hadamard gate is capable of converting the qubit into a superposition of states, consider the state $|\psi\rangle = |0\rangle$, then a Hadamard operation would yield

$$H|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle\langle 0| + |1\rangle\langle 0| + |0\rangle\langle 1| - |1\rangle\langle 1|)|0\rangle$$

$$= \frac{1}{\sqrt{2}}(|0\rangle\langle 0|0\rangle + |1\rangle\langle 0|0\rangle + |0\rangle\langle 1|0\rangle - |1\rangle\langle 1|0\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

In the introduction to the Hadamard gate and the example operation, we have not explained two key concepts which allowed us to create a superposition of state for $|\psi\rangle$. We applied the *outer product* to define our Hadamard gate, and we applied

the *inner product* to simplify the operation $H|\psi\rangle$. The outer product provides us the opportunity to create gates which then operate on a set of qubits. The inner product has more quantitative significance; given a qubit $\langle s_1|$ and $|s_2\rangle$, the inner product is

$$\langle s_1| \, |s_2\rangle = \langle s_1|s_2\rangle = \delta_{s_1,s_2}$$

where δ_{s_1,s_2} is the kronecker delta of s_1 and s_2, which evaluates to 1 if $\langle s_1|$ and $|s_2\rangle$ are orthogonal, or $s_1 = s_2$. For general states, we typically express the kronecker delta as $\delta_{i,j}$ for states $\langle i|$ and $|j\rangle$.

Referring to our Hadamard operation example, we were able to apply the inner product to simplify our final state $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. In general, any gate that wants to *modify* the a set of qubits defined by $|\psi\rangle$ must be **unitary**. In other words, the Hadamard gate (and any gate) must have the relation:

$$HH^\dagger = \mathbb{1}$$

Where $H^\dagger$ is the *complex conjugate transpose* of the Hadamard gate and $\mathbb{1}$ is the identity gate. If instead we wanted to *measure* the set of qubits defined by $|\psi\rangle$, then the gate must be **hermitian**, or

$$H = H^\dagger$$

An example of a unitary gate are the pauli matrices, which we will define later in our experiment. The resulting measurements of a unitary gate are equal to the eigenvalues

of said gate. The value, $\frac{1}{\sqrt{2}}$ allows us to normalize $|\psi\rangle$ [32]; in other words, the inner product of $|\psi\rangle$ must yield

$$\langle\psi|\psi\rangle = 1$$

Which is the case for our final state of $|\psi\rangle$. In general, n qubits defined by $|\psi\rangle$ will have 2^n possible coefficients, $c_1, ..c_n$. In our experiment, we modify these coefficients by applying a **Hamiltonian** operator.

The Hamiltonian is a hermitian operator that can change $|\psi\rangle$ over time by modifying a unitary gate (we apply a different method in our experiment since we simulate our quantum algorithms). In particular if we have a Hamiltonian $\mathcal{H}$, then a unitary gate $U(t, t_0)$ (from time t_0 to time t) is defined by

$$i\hbar\frac{dU(t,t_0)}{dt} = \mathcal{H}(t)U(t,t_0) \tag{1.8}$$

Where $U(t, t_0)$ is changing in accordance to the Hamiltonian's change over time. We then modify the state $|\psi\rangle$ by the same unitary to yield

$$|\psi(t)\rangle = U(t,t_0)\,|\psi(t_0)\rangle \tag{1.9}$$

In our experiment, we will use the Hamiltonian to propagate the coefficients of $|\psi\rangle$ over time (and we explain our procedure in chapter two).

Matrix representation of qubits and operators

For our classical simulation of our quantum algorithms, states, and gates, we use a matrix representation. Consider again the case of the single qubit; our translations to matrix notation are:

$$|0\rangle \rightarrow \begin{pmatrix} 1 \\ \\ 0 \end{pmatrix}$$

and

$$|1\rangle \rightarrow \begin{pmatrix} 0 \\ \\ 1 \end{pmatrix}$$

Where we use column matrices. The notation $\langle 0|, \langle 1|$ is defined by

$$\langle 0| \rightarrow \begin{pmatrix} 1 & 0 \end{pmatrix}$$

and

$$\langle 1| \rightarrow \begin{pmatrix} 0 & 1 \end{pmatrix}$$

These are simple representations of our qubits; now consider multiple qubits. For

a two qubit gate equal to zero (i.e. $|00\rangle$), our translation is:

$$|0\rangle \rightarrow \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Where our matrix expands by 2^n for n qubits. A similar procedure applies to the previous three states I described. For quantum gates, consider the Hadamard gate once again; it's translation is defined as

$$H = \frac{1}{\sqrt{2}}(|0\rangle\langle 0| + |1\rangle\langle 0| + |0\rangle\langle 1| - |1\rangle\langle 1|)$$

$$\rightarrow \frac{1}{\sqrt{2}}(\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & -1 \end{pmatrix}) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Since we will only use single-qubit operations, this example should suffice to understand translation of any single-qubit operator. In our experiment, we will be using matrix representations of our quantum qubits, gates, and algorithms. The matrix notation makes it simpler to translate for classical computation, but as we saw for larger amounts of qubits, the classical representation scales poorly with an increasing number of qubits; it becomes harder to compute operations of a 2^n column matrix with a $2^n \times 2^n$ gate.

Quantum Computing Potential

Previously, we identified inefficiencies with the classical version of the neural network. We can primarily attribute this to the sequential manner in which we find the gradients of our weights and biases, $\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}....., \frac{\partial J}{\partial w_n}$ and $\frac{\partial J}{b_1}, \frac{\partial J}{b_2}..., \frac{\partial J}{b_n}$, for n features total. A quantum computer *could* calculate the gradients of the cost function with respect to each weight[27] and possibly at faster times – though, according to our literature review – that has yet to be proven. This is due to the concept known as **Quantum Parallelism**, which uniquely takes in all 2^n combinations of inputs, for some number of qubits n, at the same time for – ideally – an optimized solution[32].

The power of Quantum Parallelism was affirmed in the 1990s primarily due to the famous Shor[26] and Grover [15] algorithms. The Shor algorithm factors integers with a time complexity of

$$\mathcal{O}(log(n)) \quad \text{(Shor Algorithm)}$$

No classical algorithm is known to beat or even approach the efficiency of this algorithm. Indeed, [26] is a case where the quantum algorithm is an exponential speed-up compared to the fastest classical algorithm. Yan[31] further elaborates on the inefficiency of classical algorithms (years after the Shor algorithm was published):

"At present, we still do not have an efficient algorithm for factorizing large integers, the famous RSA cryptosystem [has] in fact based its security on the intractability of the integer factorization problem."

Meanwhile, the Grover algorithm is a search algorithm that can find an item from an unsorted list. The time complexity of this algorithm is

$$\mathcal{O}(\sqrt{n}) \quad \text{(Grover Algorithm)}$$

Though not as powerful as the Shor algorithm, the Grover algorithm is quadratically faster than any known classical analog. The Shor and Grover algorithms have since attracted many researchers to find optimized algorithms in the field of quantum computing.

One primary goal of researchers is to find exponentially reduced solutions to NP-complete problems[13][10]; this includes optimization problems such as the minimization of the cost function in a neural network[4]. This will be the focus of this paper: to find an optimized solution for minimizing a neural network cost function.

Quantum Optimization

Part of our focus for this project is in the field of Quantum Optimization. Researchers have attempted to apply quantum optimization algorithms either through derived "problem Hamiltonians"[28][3][2] or through unitary-gates[28][16][17]. However, we note that a problem Hamiltonian H_P can be converted into unitary gates by exponentiation [32], which may be useful for implementation in the circuit model of quantum computers. Thus, it may be favorable to first derive a problem Hamiltonian.

For quantum optimization, we consider two algorithms expressed through Hamil-

tonians or unitary-gates, respectively:

1. Adiabatic Quantum Computing (AQC)

2. Quantum Approximate Optimization algorithm (QAOA)

AQC

AQC was proposed by Fahri et al. as an analog to the traditional circuit model of quantum computing. It follows the evolution of a time-dependent Hamiltonian, $H(t)$, described by equation 1.10:

$$H(t) = \left(1 - \frac{t}{T}\right) H_b + \left(\frac{t}{T}\right) H_P \tag{1.10}$$

This equation is a quantitative description of the adiabatic theorem. the **adiabatic theorem**[6] states that

"A physical system remains in its instantaneous eigenstate if a given perturbation is acting on it slowly enough and if there is a gap between the eigenvalue and the rest of the Hamiltonian's spectrum"

In our explanation, the eigenstate in-question is the "ground-state," or the state in which we can yield the ground-state energy (also known as the smallest eigenvalue, $\lambda_g \in \mathbb{R}$). The time-dependent Hamiltonian we use comes from equation 1.10. A more specific and relevant definition of the adiabatic theorem would then be

"A physical system remains in its ground-state if a given perturbation is acting on it slowly enough and if there is a non-degenerate ground-state energy, λ_g, in the Hamiltonian's spectrum"

Where we simply note that we will remain in the ground-state regardless of the final state of $H(t)$. This time-dependent Hamiltonian is used to solve the ground-state ,$|\psi(T)\rangle$[4], for some later time T, of the time-dependent Hamiltonian, $H(T) = H_P$; to solve this, we start with the ground-state of H_b, which we define as $|\psi(0)\rangle$ for $t = 0$. With these equations and states in-hand, we implement AQC by propagating $|\psi(0)\rangle$ through time using the Schrodinger equation, which is known to scale poorly with complex amounts of data [8].

We propagate the wave function $|\psi(t)\rangle$ using the following equation:

$$i\frac{\partial |\psi(t)\rangle}{\partial t} = H(t)\,|\psi(t)\rangle \tag{1.11}$$

Where $|\psi(t)\rangle = \sum_i^k c_i\,|i\rangle$ for $k = 2^n$ and n is the number of qubits in the system. c_i is the associated coefficient with $|i\rangle_k$. $|i\rangle_k$ is simply one of k possible states in $|\psi\rangle$. Note that this equation differs from equations 1.9 and 1.8 since we are instead applying a Hamiltonian to modify the state $|\psi\rangle$. Equation 1.11 is also known as the Schrodinger equation, where we express the time derivative of $|\psi\rangle$ as an operator, $H(t)$. Using equation 1.11, we can *simulate AQC* on classical computers. We substitute this into

[4]This is known as the time-dependent wave function, which we write in Dirac notation.

the original equation to yield

$$i\frac{d}{dt}\sum_{i}^{k} c_i \left|i\right\rangle = H(t)\left|\psi(t)\right\rangle$$

which simplifies the equation to

$$i\sum_{i}^{k} \frac{dc_i}{dt}\left\langle j|i\right\rangle = \sum_{j}^{k}\left\langle j\right| H(t)\left|\psi(t)\right\rangle$$

where $\langle j|$ is the associated bra state with c_j in the n-qubit system. Using orthogonality between $\langle j|i\rangle$ – known as the kronecker delta δ_{ij} – and transferring the imaginary constant, i, to the other side, we find

$$\frac{dc_j}{dt} = -i < j|H(t)|\psi(t) > \tag{1.12}$$

where we replace c_i with c_j due to δ_{ij}. Equation 1.12 describes the propagation of the c_j for the associated state $|j\rangle$ in $|\psi(t)\rangle$. As each state of $|\psi(t)\rangle$ changes over time, we slowly reach our final ground-state $|\psi(T)\rangle$.

The complexity of AQC in a quantum computer is not entirely known, and we do not attempt to define the complexity in our classical simulation. The objective of this thesis is to define a possible algorithmic advantage between classical computers and quantum computers, so it does not make sense to make an algorithmic comparison of a simulated quantum algorithm with a classical algorithm. Therefore, we are unable to define the time complexity in a real implementation.

Via a quantum computer, the time complexity of AQC is independent of of the size of state $|\psi(t)\rangle$ and instead relies on the time that required to approximate the ground-state. This algorithm possibly runs at a linear time (we can not yet prove this), which is the desired complexity we explained in chapter 1.

Once we find $|\psi(T)\rangle$, we can derive the solution/ground-state energy of the Problem Hamiltonian via the *expectation value*[13]

$$\lambda_g \approx \langle\psi(T)|\, H(T)\, |\psi(T)\rangle = \langle\psi(T)|\, H_P\, |\psi(T)\rangle \quad \text{(Ground-State Energy)} \tag{1.13}$$

Researchers such as Farhi et al. have attempted to apply AQC toward NP problems[14]. Other researchers have laid out strategies to numerically solve problems via the provided problem Hamiltonian[28][3].

One of the biggest challenges to AQC is defining the problem Hamiltonian H_p. There is no known standard or generalization of embedding optimization problems into the ground state of a Hamiltonian. Some researchers had issues embedding the correct cost function[2], which negates the advantage of AQC altogether. Thus, we require that a problem Hamiltonian accurately encodes a cost function for the associated problem.

QAOA

The second algorithm – proposed by [12] in 2014 – applies a newer, gate-based method to find the ground-state of a given problem Hamiltonian H_C. In this case, we imple-

ment the algorithm by:

- Encoding a Problem Hamiltonian (sometimes called a cost Hamiltonian) H_C with a variable weights $\gamma_1, ...\gamma_p \in \mathbb{R}$ for a circuit depth p.
- Choosing a Mixing Hamiltonian H_M (arbitrary) with a variable weight $\beta_1, ...\beta_p \in \mathbb{R}$ for a circuit depth p.
- Setting the initial state $|\psi\rangle$ via a series of Hadamard gates defined in equation 1.7.
- Creating parameterized quantum gates; we exponentiate H_C, H_M so that our final gates are $U(H_c, \gamma_1)....U(H_c, \gamma_p)$ and $U(H_M, \beta_1), ...U(H_M, \beta_2)$.
- Employing gradient descent to optimize the parameterized quantum state to the desired output.

As expected, encoding the problem Hamiltonian creates the most hassle for implementing this algorithm. However, once we overcome this challenge, it is simple to implement the rest of the algorithm. The difference between QAOA and AQC is that we "exponentiate" the problem and mixing hamiltonians to create *a series of unitary gates*, which we apply toward the state $|\psi\rangle$.

In addition, the application of the gates *alternate* between the base and problem Hamiltonian gates, which both contain variable weights. Similarly, we find the ground

state energy [5] of the Hamiltonian matrix H_C by calculating the value

$$\lambda_g \approx \langle \gamma, \beta | H_C | \gamma, \beta \rangle \quad \text{(Ground state energy)} \tag{1.14}$$

Where we are tasked with deriving $|\gamma, \beta\rangle$ via QAOA. We define a high-level representation of the transition as:

$$|\gamma, \beta\rangle = U(H_M, \beta_p)U(H_c, \gamma_p)...U(H_M, \beta_1)U(H_c, \gamma_1) |s\rangle$$

This series of gates $U(H_M, \beta_p)U(H_c, \gamma_p)...U(H_M, \beta_1)U(H_c, \gamma_1)$ creates a depth in the circuit, p. The depth is simply a description of the number of gates a state must pass before being measured.

In QAOA, increasing depth p implies an approach toward the minimization/maximization of the Hamiltonian H_C. Unlike AQC, the initial state $|s\rangle$ is not the ground-state of the mixing Hamiltonian; in fact $|s\rangle = |+\rangle \otimes |+\rangle \otimes .... = |+\rangle_n$. Thus, we start the circuit with a series of Hadamard gates acting on $|s_0\rangle = |0_k\rangle$ such that

$$(H_1 \otimes H_i... \otimes H_k)(|0_1\rangle \otimes ..... |0_k\rangle) = (H_1 \otimes H_i... \otimes H_k) |s_0\rangle = |s\rangle$$

where $H_i = \frac{1}{\sqrt{2}}(|0\rangle \langle 0| + |0\rangle \langle 1| + |1\rangle \langle 0| + |1\rangle \langle 1|)$ acts on the $i - th$ qubit.

We derive the unitary gates by exponentiating each Hamiltonian we provided

[5] The ground state energy is cost Hamiltonian is ideally the minimum of a desired cost function

along with an associated parameter γ_i, β_i. The result is

$$U(H_c, \gamma_i) = e^{-i\gamma_i H_c}$$

for the problem Hamiltonian and

$$U(H_M, \beta_i) = e^{-i\beta_i H_M}$$

for the mixing Hamiltonian.

Given that companies such as IBM provide a quantum computing service based on the circuit model, it may be advantageous to use QAOA and use a quantum computer for the algorithm. Unlike AQC (without modifications), QAOA can be implemented on currently available quantum computers based on the gate model. This makes QAOA a promising algorithm to experiment with.

QAOA has been used to find the solutions of graph-based problems[12][17]. The Max-cut problem is well known and has a solution through QAOA. The Max-cut-QAOA algorithm also has a defined problem Hamiltonian H_C. Some researchers have also demonstrated an improvement on QAOA[16]; this lies outside the scope of this paper.

Per the literature we reviewed[10][27], there has been no quantitative attempt to create a heuristic FFNN algorithm with QAOA nor with AQC; although machine learning attempts on AQC have been done before[23]. Implementing a QAOA neural network may set a new precedent given its success with other, albeit smaller problems

in the field of graph theory.

Quantum Neural Networks

We have discussed the two algorithms used for our thesis: AQC and QAOA. Other researchers have attempted neural networks through other algorithms [1], but these algorithms have not drawn as much attention as AQC and QAOA to the quantum optimization field. AQC and QAOA both guarantee the minimization of a cost Hamiltonian H_c (or the ground-state of H_c) given the limit $T \to \infty$ and $p \to \infty$, respectively[13][12], and both algorithms yield excellent approximations to the ground state.

The search for a quantum neural network(QNN) reaches as far back as 1995 [22]. More recently, hybrid neural networks – where the algorithm splits between a classical machine learning and quantum minimization technique – have become increasingly popular[27]. Based on this trend, the AQC and QAOA methods may be the next step in supervised learning for hybrid models.

For any neural network, [28] states the following requirements need to be met for the neural network to be a "meaningful" QNN model:

1. The initial state of the quantum system encodes any binary string of length n and the QNN produces a stable output configuration encoding the one state of 2^n possibilities.

2. The QNN reflects one or more basic neural computing mechanisms

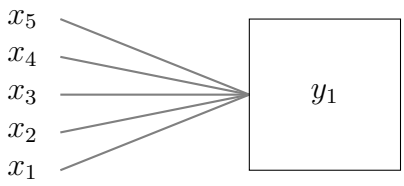

Figure 1.2: The figure is a simplified version of the FFNN described in 1.1 – also known as a perceptron. Note that the node is instead shaped as a square; this is to indicate the quantum process that remains inside the output node. We will use the perceptron model to describe the quantum neural network we propose.

3. The evolution is based on quantum effects, such as superposition, entanglement and interference, and it is fully consistent with quantum theory.

Where the first and third requirements are equally satisfied by the rules of AQC and QAOA. The second requirement will be completed by the simplified FFNN/perceptron modeled in figure 1.2, which we elaborate upon in the next section.

Consider the context of a neural network: we must minimize the cost function by adjusting the associated weights w_i by ∂w_j for all $j = 1, ...n$ for n features. There are no biases. There are no hidden layers. Recall equation 1.3; for a single training example, the gradient of w_j can be described as

$$\frac{\partial J^i}{\partial w_j} = \frac{\partial J^i}{\partial z^i}\frac{\partial z^i}{\partial f^i}\frac{\partial f^i}{\partial w_j}$$

where we apply the chain rule to find the proper derivative for the i-th training example. We find that each partial derivative is evaluated as

$$\frac{\partial J^i}{\partial A^i} = \frac{Y^i}{A^i} - \frac{1 - Y^i}{1 - A^i} = \frac{Y^i - A^i}{A^i(1 - A^i)},$$

$$\frac{\partial A^i}{\partial f^i} = A^i(1 - A^i),$$

and

$$\frac{\partial f^i}{\partial w_j} = x_j$$

Thus, our final gradient with respect to w_j is

$$\frac{\partial J^i}{\partial w_j} = (y^i - A^i)x_j \quad \text{(Final Gradient for } w_i\text{)} \tag{1.15}$$

Equation 1.15 is the function we hope to describe for our **Cost Hamiltonian**.

To apply AQC or QAOA on an FFNN, we recognize that finding the minimum cost function J_min is unlikely to yield a run-time improvement; the calculation of equation 1.15 virtually instantaneous (it makes two operations, so $\mathcal{O}(1)$), so there's no possible improvement that could be made to this implementation. Unlike the max-cut problems demonstrated by other researchers[12], we find that the neural network minimizes the cost function by *parameter adjustment* or adjusting the weights, $w_1, ..., w_n$ so that the cost is minimized. In the max-cut problem, the weights and inputs were pre-defined and constant; here, the weights are variable.

It is therefore in our best interest to minimize the cost function by minimizing the cost per weight. We define the "cost-minimization-per-weight" method as the "trivial solution" for minimizing the cost function of a QNN.

We first encode the gradient as the Hamiltonian ground-state energy; we can then easily calculate the individual gradient with respect to w_j. Using AQC, we will find

the ground-state of the Hamiltonian, or in this case, find the gradient of the weight.

CHAPTER 2

THE AQC-BASED QNN PROPOSAL AND IMPLEMENTATION

The AQC-based QNN Proposal

Given the perceptron model we have discussed in the previous section, we will note the following restrictions and terms:

1. $x_j \in \{-1, 1\}$[1] for all training features. We use $\{-1, 1\}$ instead of $\{0, 1\}$ to prevent the gradient in equation 1.15 from being zero.

2. y^i is a pre-determined value depending on the training example.

3. $y^i - A^i$ should calculate the overall difference between the prediction and the label, while x_j determines the polarity.

With this in mind, we create the following transitions to create our problem Hamiltonian. An input feature becomes

$$x_i \to \sigma_Z$$

where σ_Z is the pauli-z matrix. For simplicity, we will substitute Z for σ_Z. We choose the translation x_i to σ_Z since $\lambda_{\sigma_z} = \pm 1$; this matches with our input feature, which

[1] We require the input features $x_1 ... x_n$ to be binary due to restrictions of AQC [7]

belongs to $x_i = \pm 1$.

The label of training example i becomes

$$Y^i \to \mathbb{1}, \mathbf{0}$$

where $\mathbb{1}$ and $\mathbf{0}$ are the identity and zero matrices, respectively. We choose these matrices for their eigenvalues, which are equivalent to $y \in \{0, 1\}$. The associated prediction Hamiltonian, H_A^i is defined as

$$A \to H_A = \begin{pmatrix} A & 0 \\ 0 & -A \end{pmatrix}$$

So that $\lambda_A = \pm A$. Finally, the problem Hamiltonian is

$$H_{\partial w_j} = (Y^i - H_A^i)Z = \begin{cases} [\mathbb{1} - H_A^i]Z & Y^i = 1 \\ -H_A^i Z & Y^i = 0 \end{cases}$$

to mirror the character of its classical analog. With this information at hand, we can minimize the cost function via AQC and QAOA. In the next chapter, we implement an AQC-QNN and follow up with the next steps for a QAOA-QNN.

The AQC-based QNN Implementation

By simulating AQC on python[24], we created an AQC-based FFNN model. There have been attempts to create a general AQC machine learning model[23], an AQC

hopfield network[18], and an AQC boltzmann machine[20] but there is no literature regarding an AQC-based FFNN or perceptron model.

We used the Jupyter notebook platform to implement the code of the AQC-based QNN in segments. First, we simulated AQC by applying the complex ODE solver known as *scipy.integrate.complex_ode*. We found that *complex_ode.set_f_params*() and the initialization of a *complex_ode* object does not allow for multiple arguments (i.e. we could not insert our time-dependent Hamiltonian to our argument by conventional function calls), so we worked around it by creating a separate class with our needed arguments[24]. Afterward, we were easily able to solve the complex ODE that we will define in our implementation of AQC following sections. Following the AQC implementation, we merged our classical FFNN implementation with this algorithm; in our case, we replaced the calculation of the gradient $\frac{\partial J}{\partial w_j}$ with the AQC algorithm. We will define this variable soon after our of the simulated AQC algorithm.

Finally, we should note that we limited our training data to five-bit data. We set our input to five-bits since we can interpret this instead as classically simulated qubits. With this interpretation in mind, we then see that multiple qubits would be increasingly difficult to simulate on our classical computer.

Implementation of AQC

First, we verified our numerical implementation of simulated adiabatic quantum computing. We solved for the ground-state of an arbitrary 3-qubit Hamiltonian, whose ground-state is clearly described in figure 2.1 as $\lambda_g \approx -4$. The time-dependent

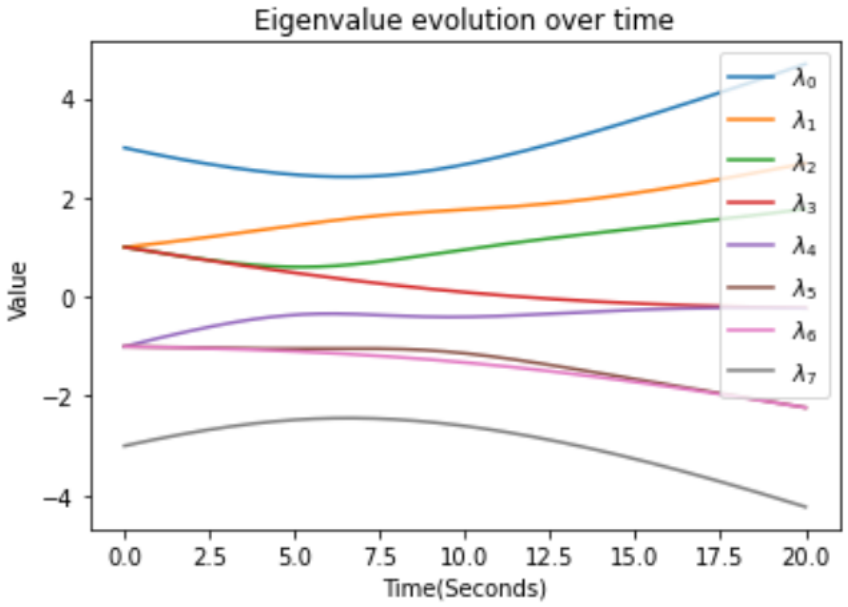

Figure 2.1: The following graph describes the expected evolution of a time-dependent Hamiltonian's eigenvalues over a range of 20 (simulated) seconds. Since the Hamiltonian is made for three qubits ($2^3 \times 2^3$ matrix), we must have $2^3 = 8$ possible eigenvalues. The seventh eigenvalue, λ_7, is clearly the smallest eigenvalue value – or the ground-state eigenvalue – at the final time evolution $H(T)$. Thus, we expect the ground-state energy to be close to -4, or $\lambda_g \approx -4$. The concrete value will be found with the AQC algorithm.

Hamiltonian in question is

$$H(t) = \left(1 - \frac{t}{T}\right) H_0 + \left(\frac{t}{T}\right) H_{new}$$

Before we continue, we will make use of the following defined gates:

The Pauli-Z gate

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},$$

The identity gate

$$\mathbb{1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},$$

and the Pauli-x gate

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$

We will make use of this along with two coefficients $h\omega = 1$ and $h\Omega = 1.23$ for our experiment[2].

We define H_0 as

$$H_0 = h\omega(Z \otimes \mathbb{1} \otimes \mathbb{1} + \mathbb{1} \otimes Z \otimes \mathbb{1} + \mathbb{1} \otimes \mathbb{1} \otimes Z)$$

[2]These values are arbitrary and can be modified for different results

```
[[ 3  0  0  0  0  0  0  0]
 [ 0  1  0  0  0  0  0  0]
 [ 0  0  1  0  0  0  0  0]
 [ 0  0  0 -1  0  0  0  0]
 [ 0  0  0  0  1  0  0  0]
 [ 0  0  0  0  0 -1  0  0]
 [ 0  0  0  0  0  0 -1  0]
 [ 0  0  0  0  0  0  0 -3]]
```

```
[[ 0.    1.   -1.    1.23  1.    1.23  1.23  0.  ]
 [ 1.    0.    1.23 -1.    1.23  1.    0.    1.23]
 [-1.    1.23  0.    1.    1.23  0.    1.    1.23]
 [ 1.23 -1.    1.    0.    0.    1.23  1.23  1.  ]
 [ 1.    1.23  1.23  0.    0.    1.   -1.    1.23]
 [ 1.23  1.    0.    1.23  1.    0.    1.23 -1.  ]
 [ 1.23  0.    1.    1.23 -1.    1.23  0.    1.  ]
 [ 0.    1.23  1.23  1.    1.23 -1.    1.    0.  ]]
```

Figure 2.2: The following figure details the expanded version of the arbitrary H_0 and the problem Hamiltonian H_{new}. Clearly, the initial ground state is $|\psi(0)\rangle = |7\rangle_3$ with an associated eigenvalue $\lambda_g = -3$.

and we define the much more complex Hamiltonian, H_{new} as

$$H_{new} = [X \otimes \mathbb{1} \otimes \mathbb{1} + \mathbb{1} \otimes X \otimes \mathbb{1} + \mathbb{1} \otimes \mathbb{1} \otimes X] + h\Omega \times [X \otimes X \otimes \mathbb{1} + X \otimes \mathbb{1} \otimes X + \mathbb{1} \otimes X \otimes X]$$

The expanded versions of each of these matrices are in figure 2.2. At $t = 0$,

$$H(t) = H(0) = H_0$$

We can visually affirm from figure 2.2 that the ground-state energy, or the smallest eigenvalue, is $\lambda_g = -3$. However, this ground-state energy is only revealed when we apply H_0 toward the state $|7\rangle_3$; therefore, the ground-state is $|7\rangle_3$. The ground-state of H_{new}, however, is not so easy to find in figure 2.2; hence, the practicality of AQC arises.

With both $H(t)$ and $|\psi(0)\rangle$ at-hand, we can solve for equation 1.12 for multiple iterations up to twenty seconds – or $T = 20$. Doing so will change $|\psi(t)\rangle$ over time, whose coefficients are modified and resemble an oscillatory change in figure 2.3.

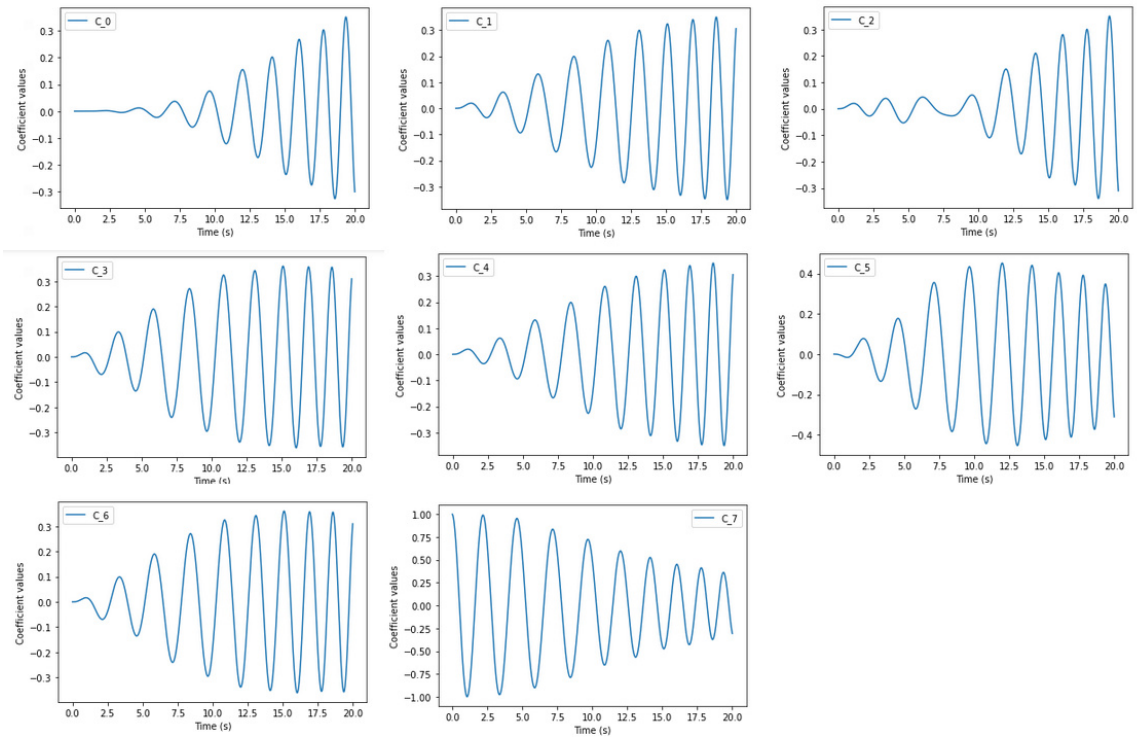

Figure 2.3: The following series of graphs depicts the evolution of the three qubit state $|\psi(t)\rangle$. Since we have a three qubit state, there must be $2^3 = 8$ coefficients to consider. This evolution was developed from the given $H(t)$ and $|\psi(0)\rangle$ in figure 2.2.

```
"""
Find the ground state using the final |psi(t)> value:
"""
#H_new.shape
eig, eig_state = np.linalg.eig(H_new[:,:,0])
ground_eig = min(eig)

print("Expected ground state: ",ground_eig)

#Experimental value:
e_g = np.dot(np.matrix.getH(psi_f),
             np.dot(H_t[:,:,t.shape[0]-1],
                    psi_f))
print("Experimental ground state: ",e_g)

```

```
Expected ground state:  -4.2299999999999995
Experimental ground state:  (-4.229523418717363+0j)
```

Figure 2.4: The code snippet compares the experimental ground-state of the time-dependent Hamiltonian to the expected value derived via a numerical eigensolver, 'numpy.linalg.eig(H_{new})'. The difference between the two is already small, but can be improved upon by increasing the time span t; in general the ground state is calculated as exact when $t \to \infty$. This is why increasing t will yield an near-exact ground-state calculation

The final state, $|\psi(T)\rangle$ is the supposed ground-state of H_{new}. We can verify this by finally applying equation 1.13 to yield

$$\langle\psi(T)|\, H(T)\, |\psi(T)\rangle = \langle\psi(T)|\, H_{new}\, |\psi(T)\rangle = -4.2295234...$$

The expectation value appropriately aligns with the ground-state of figure 2.1. In addition, we want to compare this to the expected value using the NumPy eigensolver, "numpy.linalg.eig." The results of both simulations are shown in figure 2.4. Given that the error rate is less than .1%, we affirm that the implementation of simulated AQC, as per [13], is successful.

x_1	x_2	x_3	x_4	x_5	y
1	X	X	X	X	1
1	1	1	1	1	0

Table 2.1: The following table is the training data that is used for our implementation of an AQC-based QNN. As long as $x_1 = 1$, $y = 1$; the only exception is when every input feature $x_1 = x_2...x_5 = 1$, in which case $y = 0$. Note that we could easily modify the training data to fit any other requirements; the AQC-based QNN would still find the minimum of the cost function.

Implementation of an AQC-based QNN

With the FFNN implementation[25], our proposed AQC-based QNN, and the AQC simulation implementation[24], we can now implement our AQC-based QNN, which we also find in [24].

Consider the data set described in table 2.1. The training data is labelled as $y = 1$ so long as $x_1 = 1$; the only exception is when $x_1 = x_2.. = x_5 = 1$, in which case $y = 0$. Any other combination is assigned a zero label, $y = 0$.

Figure 2.5 illustrates the technical implementation of the AQC-based QNN; however, this can be described at a high-level by the following psuedocode; after initializing the weights $w_1, ...w_n$ for n input features:

1. For each iteration k
    (a) For each training example i
        i. For each training feature j
            A. Calculate forward propagation f^i and the activation function z^i for all training features and examples
            B. Compose the time-dependent Hamiltonian $H(t)$, where $H_C = (Y -$

```
for k in range(iterations):
    c = 0
    for training_ex in range(y.shape[1]):
        #print("Training example #",training_ex)
        for training_feat in range(x.shape[0]):
            #Prevent log(0) errors
            #Forward Propagation
            f = np.dot(w,x)
            a = sigmoid(f)
            #print("Changing Cost:", c)
            z_x = np.array([[1,0],[0,-1]])
            H_a = np.array([[a[0,training_ex],0],[0,a[0,training_ex]]])
            if(y[:,training_ex] == 1):
                H_y = np.identity(2)
            else:
                H_y=np.zeros((2,2))
            H_p = np.dot((H_y-H_a),z_x)
            H_p = H_p.reshape((H_p.shape[0],H_p.shape[1],1))

            #SOLVE FOR MINIMUM GRADIENT VIA AQC
            H_t = (1-t/T)*H_0 + t/T*H_p
            psi_t, psi_f = solve_comp(ode_ham, H_t, psi_0, t)
            expect = np.dot(
            np.matrix.getH(psi_f),np.dot(
            H_t[:,:,H_t.shape[2]-1],psi_f))
            if(x[training_feat,training_ex] == y[:,training_ex]-1 or
               x[training_feat,training_ex]==y[:,training_ex]):
                #Smallest gradient
                weight_evolution = np.append(weight_evolution, w[:,training_feat])
                w[:,training_feat] += np.real(expect)
            else:
                #Reverse Polarity
                weight_evolution = np.append(weight_evolution, w[:,training_feat])
                w[:,training_feat] -= np.real(expect)
            weight_evolution = np.append(weight_evolution,w[:,training_feat])

    #Take the cost of the training example if you would like to see how it changes per training example
    c = -1/x.shape[1]*(y*np.log(a)+(1-y)*np.log(1-a))
    #Sometimes, our prediction can be SO accurate that we end up computing np.log(0)
    #We check for these answers and modify it to zero instead
    #print("Before nan/inf checks: ",c)
    c[np.isinf(c)] = 0
    c[np.isnan(c)] = 0
    #print("After nan/inf checks: ",c)

    cost_total = np.append(cost_total,np.sum(c))
    if k %1 == 0:
        print("Weight values: ", w)
        print("Current Cost", cost_total[k])
print("Cost Chart[indexed by 'iteration']: ",cost_total)
```

Figure 2.5: The code snippet encapsulates the AQC-based QNN. The highlighted portion is the construction of the time-dependent Hamiltonian and the propagation of the wave function $|\psi(t)\rangle$ to T via AQC.

$H_A)Z$ and find the embedded gradient

C. Subtract the weight w_j by the gradient (reverse the polarity if $y \equiv x$) and repeat the loop

(b) Calculate the cost of the entire training set, $J = -1/m \sum_i^m J^i$.

There are a couple of differences between the AQC-based QNN and the classical FFNN. First, forward propagation and the activation function are calculated for every change in the weight, $w_{new} = w_{old} - \frac{\partial J^i}{\partial w_j}$. This mostly due to the dependence of $H(t)$ on z^i, which changes as we modify our weights.

Assessing the performance of the AQC-QNN, we measure the cost function over many iterations. Our results are described in figure 2.6. It is clear from the graph

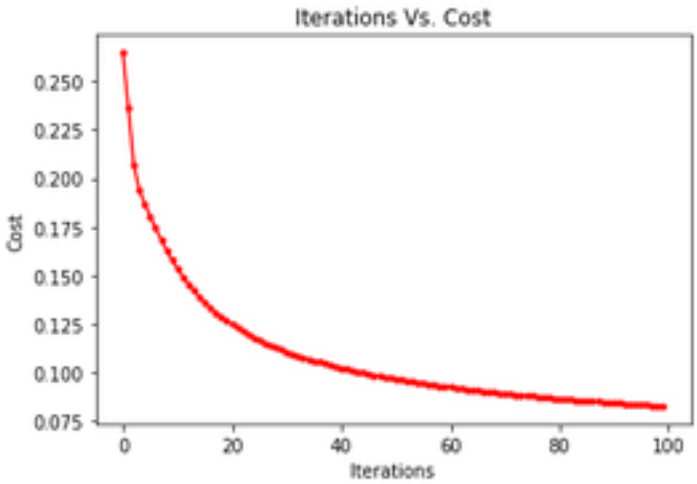

Figure 2.6: The figure depicts the cost function over multiple iterations and weight adjustments. The AQC-based QNN successfully reduces the cost function over multiple iterations; this is an excellent indicator that the algorithm is working as expected.

that the AQC-QNN experiences a clear downward trend that plateaus after many iterations. This is an indicator that the AQC simulation is properly solving for the QNN.

Finally, we want to compare the AQC-QNN to a test set; given that there are only 32 combinations with 5 bits, we decide to test the QNN against all 32 combinations. The result is shown on figure 2.7; the AQC-QNN yielded 96.875% accuracy, which exceeds our expectation of the hybrid model.

```
print(w)
#We can test with a single set of values, but since there are only 2^5 possiblities, we can just test using
#Our training set
f = np.dot(w,x)
a = sigmoid(f)
c = -1/x.shape[1]*np.sum(y*np.log(a) + (1-y)*np.log(1-a))
print("Test Cost: ",c)
predictions = sigmoid(f)
predictions[predictions>=0.5] = 1
predictions[predictions<0.5] = 0
print("predictions: ", predictions)
print("Labels: ", y)

accuracy = (1- np.sum(np.abs(predictions-y))/y.shape[1])*100
print("Accuracy: ",accuracy)
```

```
[[13.58233438 -1.20876995 -1.03624025  4.58926448 -7.98409394]]
Test Cost:  0.09618369287929512
predictions:  [[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1.
  1. 1. 1. 1. 1. 1. 1. 0.]]
Labels:  [[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1.
  1. 1. 1. 1. 1. 1. 1. 0.]]
Accuracy:  96.875
```

Figure 2.7: The following code snippet details the final weight values, the predictions made by the AQC-QNN, and the actual labels of the function for y. We then calculate the Accuracy of our neural network to 96.875%. This is high for an AQC-QNN and exceeds expectations for the hybrid model.

CHAPTER 3

ANALYSIS, NEXT STEPS, AND CONCLUSION

Analysis

Indeed, AQC is a viable substitute for calculating the gradients via backpropagation. However, its implementation comes with a few issues

1. The model sometimes experienced the "vanishing gradient" issue (large values of w_i)
2. The model does not have any additional layers
3. The model does not include a bias
4. The model does not include regularization[19]
5. The model sometimes yielded errors for the cost function, where the prediction is equal to the label, or $A^i = Y^i$.
6. The model's problem hamiltonian, H_C, is too simple to be advantageous on a neural network.

The solution for the first issue can be solved via *hyperparameter tuning*[29], so this is not a problem. Issues $2 - 4$ are left to be explored since this algorithm lays

the groundwork for a deep and thorough neural network algorithm. The fifth issue can be corrected by simply labelling $J^i = 0$, since the prediction is exact.

Issue 6 is important because AQC shines when the problem hamiltonian is exceedingly complex; a complex matrix makes it difficult for humans or numerical "eigensolvers" such as numpy's "numpy.linalg.eig(H)," function. Hence, AQC is desirable in these situations. For our hamiltonian, the matrix either evaluated to

$$H^i_{\partial w_j} = \begin{pmatrix} -A^i & 0 \\ 0 & A^i \end{pmatrix}$$

for $y = 0$ or

$$H^i_{\partial w_j} = \begin{pmatrix} 1 - A^i & 0 \\ 0 & A^i - 1 \end{pmatrix}$$

for $y = 1$. Any human or eigensolver can easily determine the ground-state of the hamiltonian. The AQC-QNN model does not take advantage of the key feature in AQC, but it successfully classifies data nonetheless.

Next Steps

We have seen that the current hamiltonian does not take advantage of the features of adiabatic quantum computing. In addition, it is computationally expensive to propagate the wave function by solving the schroedinger equation on a classical machine[8]. For these two reasons, it may be worth re-creating a hamiltonian matrix through the QAOA algorithm, which will be the next step of the quantum neural

network.

The QAOA algorithm applies a series of unitary gates to find the ground-state of a hamiltonian. Unlike AQC, which scales with period T, QAOA scales with depth p; there is a functional relation with p and T whose details we are unaware of. Because of this, QAOA may remove a portion of run-time from the algorithm. In addition, finding a complex Hamiltonian that can calculate gradients in parallel would emphasize the benefits of AQC and QAOA.

Conclusion

We found that an FFNN took prodigious resources due to the complexity of gradient descent. In general, machine learning algorithms take extreme amounts of time to classify increasingly complex amounts of data; many researchers have sought to find analogs to quantum computers[2][1][16][10]. However, there has been no standardized method for replacing a neural network as of yet[28].

Because of the lack of standardization, we explore forms of optimization to minimize the cost function of a machine learning algorithm; we find that QAOA [12] and AQC[13] are the most popular optimization algorithms to date. In an effort to merge the two ideas, we attempt to merge both the neural network and quantum optimization algorithms to create a hybrid neural network.

We implemented an AQC-QNN, in which the AQC algorithm replaces backpropagation to find the encoded gradient of the respective weights ∂w_j. However, finding the solution was fairly trivial without AQC (using equation 1.15, we yield an $\mathcal{O}(1)$

operation). Though we are unable to define the time complexity of AQC in a quantum computer, we affirm that there is no way to improve upon the original classical calculation of equation 1.15.

With the issues described earlier in mind, the AQC-based QNN was still a success and was able to accurately classify the labelled data we gave it earlier.

We find that the next step is two fold

1. Find a complex Hamiltonian that could give us the value of multiple gradients ∂w_i.

2. Implement a cost Hamiltonian using QAOA on a quantum computer.

The first step will rid us of problem 6 described in the analysis section. The second step will rid us of the added complexity of the coefficients explained by Deutsch[8]. With these proposed changes, it is still possible to clearly reduce the run-time of a neural network.

BIBLIOGRAPHY

[1] Beer, Kerstin, Dmytro Bondarenko, Terry Farrelly, Tobias J Osborne, Robert Salzmann, Daniel Scheiermann, and Ramona Wolf. "Training Deep Quantum Neural Networks." *Nature Communications* 11, no. 1 (2020): 808–808. https://doi.org/10.1038/s41467-020-14454-2.

[2] Benedetti, Marcello, John Realpe-Gómez, and Alejandro Perdomo-Ortiz. "Quantum-Assisted Helmholtz Machines: A Quantum-Classical Deep Learning Framework for Industrial Datasets in Near-Term Devices." *Quantum Science and Technology* 3, no. 3 (2018): 34007. https://doi.org/10.1088/2058-9565/aabd98.

[3] Blanzieri, Enrico, and Davide Pastorello. "Quantum Annealing Learning Search for Solving QUBO Problems," 2018. https://doi.org/10.1007/s11128-019-2418-z.

[4] Blum, A. L, and R. L Rivest. "Training a 3-Dose Neural Networks Is NP-Complete." Neural Networks 5, no. 1 (1992): 117–27.

[5] Bose, B.K. "Neural Network Applications in Power Electronics and Motor Drives-An Introduction and Perspective." *IEEE Transactions on Industrial Electronics* (1982) 54, no. 1 (2007): 14–33. https://doi.org/10.1109/TIE.2006.888683.

[6] Cohen, E, and B Tamir. "Quantum Annealing – Foundations and Frontiers." *The European Physical Journal. ST, Special Topics* 224, no. 1 (2015): 89–110. https://doi.org/10.1140/epjst/e2015-02345-1.

[7] Date, Prasanna, Davis Arthur, and Lauren Pusey-Nazzaro. "QUBO Formulations for Training Machine Learning Models." *Scientific Reports* 11, no. 1 (2021): 10029–10029. https://doi.org/10.1038/s41598-021-89461-4.

[8] Deutsch, Ivan H. "Harnessing the Power of the Second Quantum Revolution," 2020.

[9] D'Hollander, D'Hollander, E, and ParCo95. *Parallel Computing: State-of-the-Art and Perspectives*. Amsterdam; New York: Elsevier, 1996.

[10] Dunjko, Vedran, and Hans J Briegel. "Machine Learning & Artificial Intelligence in the Quantum Domain: a Review of Recent Progress." *Reports on Progress in Physics* 81, no. 7 (2018): 074001–074001. https://doi.org/10.1088/1361-6633/aab406.

[11] Egmont-Petersen, M, D de Ridder, and H Handels. "Image Processing with Neural Networks—a Review." *Pattern Recognition* 35, no. 10 (2002): 2279–2301. https://doi.org/10.1016/S0031-3203(01)00178-9.

[12] Farhi, Edward, Jeffrey Goldstone, and Sam Gutmann. "A Quantum Approximate Optimization Algorithm," 2014.

[13] Farhi, Edward, Jeffrey Goldstone, Sam Gutmann, and Michael Sipser. "Quantum Computation by Adiabatic Evolution," 2000.

[14] Farhi, Edward, Jeffrey Goldstone, Sam Gutmann, Joshua Lapan, Andrew Lundgren, and Daniel Preda. "A Quantum Adiabatic Evolution Algorithm Applied to Random Instances of an NP-Complete Problem." *Science (American Association for the Advancement of Science)* 292, no. 5516 (2001): 472–76. https://doi.org/10.1126/science.1057726.

[15] Grover, Lov. "A Fast Quantum Mechanical Algorithm for Database Search." In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, 212–19. ACM, 1996. https://doi.org/10.1145/237814.237866.

[16] Hadfield, Stuart, Zhihui Wang, Bryan O'Gorman, Eleanor Rieffel, Davide Venturelli, and Rupak Biswas. "From the Quantum Approximate Optimization Algorithm to a Quantum Alternating Operator Ansatz." *Algorithms* 12, no. 2 (2019): 34. https://doi.org/10.3390/a12020034.

[17] Harrigan, Matthew P, Kevin J Sung, Matthew Neeley, Kevin J Satzinger, Frank Arute, Kunal Arya, Juan Atalaya, et al. "Quantum Approximate Optimization of Non-Planar Graph Problems on a Planar Superconducting Processor." *Nature Physics* 17, no. 3 (2021): 332–36. https://doi.org/10.1038/s41567-020-01105-y.

[18] Kinjo, Mitsunaga, Shigeo Sato, Yuuki Nakamiya, and Koji Nakajima. "Neuromorphic Quantum Computation with Energy Dissipation." *Physical Review. A, Atomic, Molecular, and Optical Physics* 72, no. 5 (2005). https://doi.org/10.1103/PhysRevA.72.052328.L

[19] Lever, Jake, Martin Krzywinski, and Naomi Altman. "Regularization." *Nature Methods* 13, no. 10 (2016): 803–4. https://doi.org/10.1038/nmeth.4014.

[20] Liu, Jeremy, Federico Spedalieri, Ke-Thia Yao, Thomas Potok, Catherine Schuman, Steven Young, Robert Patton, Garrett Rose, and Gangotree Chamka. Adiabatic Quantum Computation Applied to Deep Learning Networks." *Entropy (Basel, Switzerland)* 20, no. 5 (2018): 380. https://doi.org/10.3390/e20050380.

[21] Mohanraj, M, S Jayaraj, and C Muraleedharan. "Applications of Artificial Neural Networks for Refrigeration, Air-Conditioning and Heat Pump systems—A Review." *Renewable & Sustainable Energy Reviews* 16, no. 2 (2012): 1340–58. https://doi.org/10.1016/j.rser.2011.10.015.

[22] Narayanan, Ajit, and Tammy Menneer. "Quantum Artificial Neural Network Architectures and Components." *Information Sciences* 128, no. 3 (2000): 231–55. https://doi.org/10.1016/S0020-0255(00)00055-4.

[23] Pudenz, Kristen L, and Daniel A Lidar. "Quantum Adiabatic Machine Learning," 2011. https://doi.org/10.1007/s11128-012-0506-4.

[24] Serrano, Erick. "AQC-QNN." Last modified October, 2021. https://github.com/erickserr125/AQC-QNN.

[25] Serrano, Erick. "Pneumonia Identification Neural Network." Last modified June, 2021. https://github.com/erickserr125/pneumonia_identification_neural_network.

[26] Shor, P. W. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." *SIAM Journal on Computing* 26, no. (1997): 1484–1509. https://doi.org/10.1137/S0097539795293172.

[27] Schuld, Maria, and Francesco Petruccione. *Supervised Learning with Quantum Computers*. Cham: Springer International Publishing AG, 2018.

[28] Schuld, M, I Sinayskiy, and F Petruccione. "The Quest for a Quantum Neural Network," 2014. https://doi.org/10.1007/s11128-014-0809-8.

[29] Wong, Jenna, Travis Manderson, Michal Abrahamowicz, David L Buckeridge, and Robyn Tamblyn. "Can Hyperparameter Tuning Improve the Performance of a Super Learner?: A Case Study." *Epidemiology (Cambridge, Mass.)* 30, no. 4 (2019): 521–31. https://doi.org/10.1097/EDE.0000000000001

[30] Wong, Richard T. "Combinatorial Optimization: Algorithms and Complexity." *SIAM Review* 25, no. 3 (1983): 424–25. https://doi.org/10.1137/1025101.

[31] Yan, Song Y. *Primality Testing and Integer Factorization in Public-Key Cryptography*. Vol. 11. Springer, 2013.

[32] Zygelman, Bernard. *A First Introduction to Quantum Computing and Information.* Cham: Springer International Publishing AG, 2018. https://doi.org/10.1007/978-3-319-91629-3.