Toward a Quantum Neural Network: Using the QAOA algorithm to replace a Feed-Forward Neural Network

Erick Serrano*
Department of Physics & Astronomy
University of Nevada Las Vegas
(Dated: June 17, 2021)

With a surge in the popularity of machine learning as a whole, many researchers have sought optimization methods to reduce the complexity of neural networks. Over the past decade, researchers have attempted to optimize neural networks via quantum algorithms, and in this proposal, we outline steps to do 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, particularly when implemented with gradient descent, and introduce a call to action for optimization of an FFNN. Afterward, we discuss the strides made in quantum computing to improve the time complexity of machine learning; we study recent attempts to improve the time complexity of a neural network, including through a complete quantum analog and a hybrid analog. We propose to use the QAOA protocol to create a hybrid analog to an FFNN; supported by previous implementations of QAOA, we believe there is potential for expanding the QAOA algorithm toward a neural network. Finally, we describe the process for which we hope to implement the QAOA algorithm and how we will compare it to the runtime of an FFNN.

I. INTRODUCTION

A. FFNN Description

The neural network model is a common machine learning algorithm that classifies complicated data into accurate and precise predictions. An FFNN has many applications in the real world; for instance, an FFNN can optimize the usage of air conditioning, heat pumps, and refrigeration[1] and general electronic motors [2]; we have also implemented an FFNN toward identifying pneumonia within patients' lungs[3], and we will briefly discuss our results to describe the inefficiencies of an FFNN.

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. Back Propagation
4. Parameter Updating (Gradient Descent)
5. Repeating (1-4) (Gradient Descent)

These five steps are executed under the following assumptions: 1) We exclude data processing, where we convert a training example to raw data, 2) The data we receive is – mostly – labeled, 3) The FFNN will use a single output node for binary classification, and 4) The FFNN is trained using gradient descent, which, at the cost of run-time, dramatically improves the accuracy of an FFNN. Figure 1 is an example of the architecture of a FFNN.

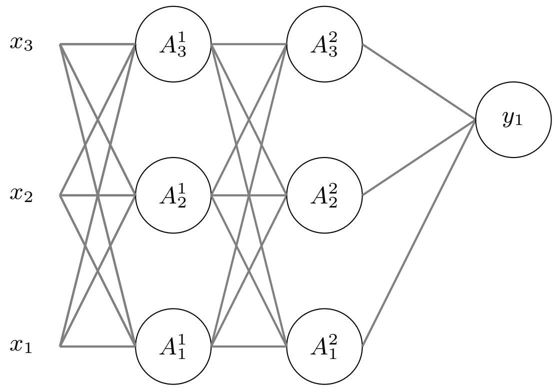


FIG. 1. The figure is an example of a standard FFNN with one input, hidden, and output layer. The number of layers and the nodes per layer are arbitrary; for example, the output layer y_1 can have multiple nodes $y_1, y_2, ...y_k$. The FFNN is due to process three data points – or three training features – x_1, x_2, x_3.

One common application of FFNNs is image classification[4], where an FFNN identifies patterns and objects in the image; we previously trained six FFNNs to identify viral or bacterial infections in an image[3]. 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[4]. We can generalize the runtime of an FFNN using steps (1-5), which we discuss in figure I B.

* serrae4@unlv.nevada.edu

B. Sub-par performance of a FFNN

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

1. *Runtime of Forward Propagation*

Forward propagation is the process of converting raw data $x_1, ...x_n$ to a set of predictions, (or in our case, a single prediction y_1). We can intuitively see that the runtime is affected by the size of the raw data and the architecture of the neural network. Specifically, we are affected by the complexity of

1. Our input features
2. A single node
3. The number nodes in a layer
4. The number of layers per neural network

Let's begin with our input features. Considering figure 1, we know that our input features, x_1, x_2, x_3, is of size $n = 3$. In general, our 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

$$f(a_i) = w_i a_i + b_i$$

where w_i, b_i are parameters, known as *weights and biases*, that belong to a node, and a_i is a single input feature; in other words, we need n weights and n biases to make n calculations, so we have a time complexity of

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

so far.

In sequence with the linear function, we must also use the activation function, which is characterized by

$$z(f(a_i)) = \frac{1}{1 + e^{-f(a_i)}}$$

So for every input feature, we have 2 calculations to make – one for the linear function and one for the activation function. That means that our calculations increase by $2n$, and our actual complexity is

$$\mathcal{O}(2n) \quad (\textit{Time Complexity of 1 node})$$

for a single node. What if we consider more than one node? Then we must make $2n$ calculations of n input features for j nodes. In other words, there are nj calculations and our complexity evolves to

$$\mathcal{O}(2nj) \quad (\textit{Time Complexity of multiple nodes, 1 layer})$$

This is what the complexity of a single layer of nodes is. Now we must consider multiple layers of an FFNN. Initially, we may believe to be able to get away with the following equation to calculate the number of computations

$$\sum_{i=0}^{L} 2n_i j_i$$

For a total of L layers. However, when we move deeper towards a neural network, j_i replaces the input feature, and a new set of nodes, j_{i+1} are used to make $j_i j_{i+1}$ calculations, and this repeats until we reach the final layer. For an easier notation, we replace our equation with

$$\sum_{i=0}^{L} 2j_i j_{i+1}$$

where $j_0 = n$ for our 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. Our total complexity now must evolve with the summation equation we have

$$\mathcal{O}(2\sum_{i=0}^{L} j_i j_{i+1}) \quad (\textit{Time complexity of forward prop.})$$

Given the freedom that an FFNN provides a programmer, we can create unique sizes for every layer of a neural network. This means there is a set of nodes j_i for the i-th layer. Now we move toward the complexity of cost measurement.

2. *Runtime of cost measurement*

To measure the complexity of the cost function for a neural network, we consider the following equation

$$J = \frac{1}{m}(\sum_{i=1}^{m} y^i log(z^i) + (1 - y^i) log(1 - z^i)$$

where z^i is the prediction made after iterating through forward propagation, y^i is the actual label, and we repeat this for every *training example* 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

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

3. *Runtime of back propagation*

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, in agreement with the name, starts at the output node of the neural network and computes the derivative, or the *gradient*, associated with our parameters, w_i^l, b_i^l for weight and bias in node i and layer l. However, we already did much of the complexity analysis through forward propagation, which we will elaborate on.

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

$$dA^l = (W^{l+1,T} * A^{l,T})/m$$

the *weight gradient*,

$$dW^l = (dA^{l+1} * A^{l,T})/m,$$

and the *bias gradient*,

$$db^l = \sum dA^l/m,$$

For a list of gradients dA, dW, db in layer l. In addition, T indicates a matrix transpose.

The output layer gradient for the activation function can be calculated instead using

$$dA^L = -(\frac{Y}{A^L} - \frac{1-Y}{1-A_L})$$

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

Considering these three equations (with the replacement of the final equation at the output layer) and the fact that there is a 1-to-1 with the number of gradients and the number of parameters, our complexity then expands to

$$\mathcal{O}(3\sum_{i=0}^{L} j_i j_{i+1}) \quad (\textit{Time complexity of Back. Prop.})$$

for j nodes in layer i. We expand our complexity to a constant of 3 because we now calculate 3 equations – the activation, weight, and bias gradient. This concludes our analysis of backward propagation.

4. *Runtime of Parameter Updating*

Now that we have our gradients and parameters, all we need to do is update every weight and bias with the corresponding gradients. The time complexity of this portion is trivial and is measured by

$$\mathcal{O}(2\sum_{i=0}^{L} j_i j_{i+1}) \quad (\textit{Time complexity of Param. Updating})$$

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

5. *Runtime of Gradient Descent*

As we previously stated, gradient descent implements all four previous steps – forward propagation, cost measurement, backward propagation, and parameter updating – iteratively. In other words, we sum the number of steps taken for each process and multiply it by some number of iterations g such 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 a neural network. We can simplify this and create an asymptotic complexity such that

$$\mathcal{O}(g(7\sum_{i=0}^{L} j_i j_{i+1} + m))\text{FFNN with Grad. Desc.}$$

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 I.

The complexity is not desirable, since a large input size ($j_0 >> 1$) will dramatically 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. An FFNN that is implemented via gradient descent is incredibly inefficient, which presents an opportunity to begin optimization. Before we confirm our complexity analysis via experiment, we will first tour a brief history of quantum machine learning, which we propose to use to optimize an FFNN.

II. LITERATURE REVIEW

A. History of Quantum Machine learning

Due to the second revolution of quantum mechanics – namely, due to the Shor and Grover algorithms [5] – researchers have developed quantum computers[6] to compete against classical computers, and scientists have been discovering algorithms that apply polynomial solutions toward complex problems[7]. Over the past decade,

TABLE I. This figure 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, and m is the number of training examples we consider. We are cautious to remove any constants or ”small” values, as an FFNN can vary in size.

Forward Propagation	$\mathcal{O}(2\sum_{i=0}^{L} j_i j_{i+1})$
Measuring the cost function	$\mathcal{O}(m)$
Backward Propagation	$\mathcal{O}(3\sum_{i=0}^{L} j_i j_{i+1})$
Parameter Updating	$\mathcal{O}(2\sum_{i=0}^{L} j_i j_{i+1})$
FFNN with Gradient Descent	$\mathcal{O}(g(7\sum_{i=0}^{L} j_i j_{i+1} + m))$

researchers in the quantum field have attempted to create analogs of machine learning algorithms[8]. Using unique neural network architectures as a baseline (such as Helmholtz machines, perceptrons, or FFNNs), there are now quantum [9][10][11] and hybrid [12][13] analogs of a neural network model, each of which aims to improve the time complexity of an FFNN.

B. Neural Networks: Quantum Analog Attempts

The most recent attempts to implement a full quantum neural network have yielded improved run-time results[9][10]; these runtime improvements were made on insignificant data sets, and each model has been restricted by the hardware of a quantum neural network. Abbas et al. have demonstrated the feasibility of a quantum neural network on the largest available quantum computer (27 qubits) and Beer et al. has done a similar implementation on a quantum simulator. Additionally, the conclusion of Beer et al. mentions issues with enlarging the quantum neural network architecture, which scales with the qubits of a quantum computer.

On the other hand, hybrid quantum neural networks[12] have yielded promising results as well without the restriction of a small quantum computer. Benedetti, Realpe-Gomez, and Perdomo-Ortiz, primarily focused on using a Helmholtz machine and the quantum computing framework to create an accurate machine learning model. The benefit of their research is that the quantum computer used for experimentation has $2,000$ qubits.

Additionally, there is use in exploring attempts to solve optimization problems, as our project proposal is to improve upon gradient descent, an optimization algorithm. Harragin et al. apply the more commonly known QAOA method toward minimization problems[14]. While QAOA *does not* serve as a replacement for gradient descent, this method may be appealing toward minimizing the cost function in a quantum neural network. The only known issue is that QAOA may not scale well with an increased depth of the circuit, since the error of the quantum hardware increases.

Of the literature reviewed and cited, there has been no application of a quantum FFNN hybrid with QAOA. The QAOA algorithm shows promise in the short term, and we believe our proposal may reduce the time complexity of an FFNN.

TABLE II. This figure details the architectures of each FFNN implemented in our experimental results. In the case of Architecture 1, $j_0 = \{x_1, ...x_{12288}\} = 12288$, $j_1 = 25$, $j_2 = y_1 = 1$.

Architecture	[Layer Dimensions]	Learning Rates, α
1	[12288, 25, 1]	.099
2	[12288, 35, 1]	.016
3	[12288, 35, 1]	.099
4	[12288, 25, 1]	.016
5	[12288, 30, 1]	.099
6	[12288, 30, 1]	.016

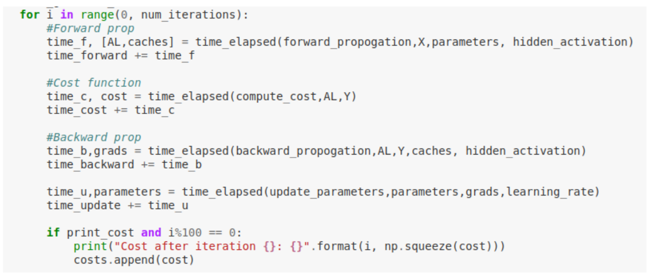
III. EXPERIMENTAL RESULTS OF AN FFNN

We have indicated that researchers are on the verge of finding a standardized, quantum neural network, and we claimed that an FFNN could yield accurate predictions over a sub-optimal runtime. To confirm our suspicions, we confirm our runtime analysis of an FFNN with experimental results[3].

A. Results & Analysis

Fig. 2 presents the high-level code snippet of our FFNN application[3]. In this example, we trained $4,000$ training examples of size $64 \times 64 \times 3$ (including RGB values per pixel) resolution for 1000 iterations on six different architectures. The architecture dimensions are tabulated in table II.

The table implies that we can calculate the complexity of each FFNN architecture; we know the number of training examples $4,0000$, the dimensions of our FFNN architecture, and the iterations $g = 1000$. We tabulate the results of each function on table III. If we recall from



```
for i in range(0, num_iterations):
    #Forward prop
    time_f, [AL,caches] = time_elapsed(forward_propogation,X,parameters, hidden_activation)
    time_forward += time_f

    #Cost function
    time_c, cost = time_elapsed(compute_cost,AL,Y)
    time_cost += time_c

    #Backward prop
    time_b,grads = time_elapsed(backward_propogation,AL,Y,caches, hidden_activation)
    time_backward += time_b

    time_u,parameters = time_elapsed(update_parameters,parameters,grads,learning_rate)
    time_update += time_u

    if print_cost and i%100 == 0:
        print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))
        costs.append(cost)
```

FIG. 2. The figure is a code snippet of the full FFNN model implemented to identify viral/bacterial pneumonia in a patients’ lungs. Steps (1-4) are repeated ”*num_iteration*” $\equiv l$ times. The code can be found in reference [3].

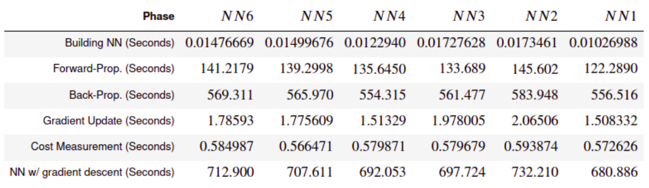
TABLE III. This figure details the architectures of each FFNN implemented in our experimental results. In the case of Architecture 1, $j_0 = \{x_1, ...x_{12288}\} = 12288$, $j_1 = 25$, $j_2 = y_1 = 1$.

Architecture	Runtime: $\mathcal{O}(g(7\sum_{i=0}^{L} j_i j_{i+1} + m))$
Architecture 1	$1000(7(12288(25) + 25) + 4000) \approx 3 \times 10^8$
Architecture 2	$1000(7(12288(35) + 35) + 4000) \approx 3 \times 10^9$
Architecture 3	3×10^9
Architecture 4	3×10^8
Architecture 5	$1000(7(12288(30) + 30) + 4000) \approx 2.5 \times 10^9$
Architecture 6	2.5×10^9

the analytical portion, we need only concern ourselves with the dimensions, training examples, and iterations to estimate the length of the training process. Table III means that we have calculations on the order of $10^8, 10^9$. We can acknowledge that the experimental implementation of the FFNN may have room for optimization, but the number of steps is unavoidable. Consider a common $3.5GHz$ processor, which can execute 3.5×10^9 cycles per second. Suppose that every cycle executes one step of the neural network. Then a 3.5 GHz processor *should* be able to train our FFNN between $\frac{6}{7} \times 10^{-1}$ and $\frac{6}{7}$ seconds. However, the actual runtime is much larger; this makes sense as most processors can not process a single-step cycle, and our implementation may not be efficient. What we should note, however, is that multiplying a constant C towards the analytical portion should scale in general towards the experiment results, which we now discuss.

Figure 3 shows the results of each function. Gradient descent is a summation of every parameter (except "building NN"). In addition, we can compare the analytical portion, which is, on average, approximately 2×10^6 larger than the number of seconds. However, the ratio between our experimental result in Figure 3 and table III tend to deviate a significant amount.

Our results deviate significantly because our input size overshadows the architecture size; this is the same for all architectures. The same applies to training examples and iterations; each one overshadows the architecture size. In other words, we can have an accurate measurement of analytical data by simplifying the time complexity for $m, g, j_o >> 1$:



Phase	$NN6$	$NN5$	$NN4$	$NN3$	$NN2$	$NN1$
Building NN (Seconds)	0.01476669	0.01499676	0.0122940	0.01727628	0.0173461	0.01026988
Forward-Prop. (Seconds)	141.2179	139.2998	135.6450	133.689	145.602	122.2890
Back-Prop. (Seconds)	569.311	565.970	554.315	561.477	583.948	556.516
Gradient Update (Seconds)	1.78593	1.775609	1.51329	1.978005	2.06506	1.508332
Cost Measurement (Seconds)	0.584987	0.566471	0.579871	0.579679	0.593874	0.572626
NN w/ gradient descent (Seconds)	712.900	707.611	692.053	697.724	732.210	680.886

FIG. 3. The figure is a list of tabulated runtimes for each FFNN architecture in the implementation. In addition, we also included the runtime for "building" the FFNN, or initializing the parameters. The code can be found in reference [3].

$$\mathcal{O}(g(j_0 + m_0)) \quad (\textit{Simplified FFNN Time Complexity})$$

and, since we will have insignificant variance between the architectures, the function will tend to evaluate at the same time. Our analytical hypothesis holds with our experimental results, as we can find an approximate constant, C, such that our results and runtimes line up.

Our example was a simple implementation of an FFNN, as it has a smaller number of training examples, smaller architecture, and smaller training features –compared to real-world applications. Changing the image count to $100,000$, making the training examples a $2000 \times 2000 \times 3$ resolution image (including RGB), and training the FFNN for 1500 iterations yields approximately $(1500)(1.2 \times 10^7 + 1 \times 10^5) = 1.5 \times 10^{10}$ steps to be calculated, which has experienced even non-polynomial growth. These parameters for i, j, n are not uncommon, and algorithmic complexity of linear time ($\mathcal{O}(n)$) or beyond often draws the attention of researchers seeking optimized solutions for common problems.

IV. HYPOTHESIS & METHODOLOGY

A. Hypothesis

We hypothesize that the FFNN can be optimized using the QAOA algorithm. We claim that, with the addition of the QAOA algorithm, we can produce a significant speedup for our hybrid FFNN.

B. QAOA Algorithm

Based on [15], we know that we can create a QAOA neural network by

- Encoding a Phase Hamiltonian
- Choosing a Mixing Hamiltonian
- Setting the initial state
- Creating a parameterized quantum state
- Employing gradient descent to optimize the parameterized quantum state

The mixing Hamiltonian is given by a pre-defined list of operators[15]. Additionally, we define the initial state, $|S\rangle$, as all predictions of m training examples; it is a superposition state

$$|S\rangle = \frac{1}{\sqrt{2^m}}\,|0\rangle_m + \frac{1}{\sqrt{2^m}}\,|1\rangle_m \dots \frac{1}{\sqrt{m}}\,|2^m\rangle_m$$

Finally, creating a parameterized quantum state $|\beta, \gamma\rangle$, is equivalent to initializing our FFNN for all w, b in the architecture. It will be relatively simple to employing gradient descent and optimize β, γ.

The difficulty in implementing a neural network with the QAOA algorithm lies in the encoding of a phase Hamiltonian. Previous implementations [14][15] show the cost Hamiltonian for very simple graph problems. Though there are some step-by-step instructions to deriving a cost Hamiltonian, there is still much to review for larger-scale implementation. We know that there must be a link between the phase Hamiltonian and 1) The features of the training example and 2) The label of the training example.

C. Methodology

Implementing the QAOA method to construct an FFNN would fall in line with the requirements made by [11]. With QAOA, there would be an encoding of a binary string (a string of predictions $|S\rangle$), mechanisms that reflect a typical FFNN, and evolution based on superposition and interference. Our focus for the project is:

1. Deriving the phase Hamiltonian,
2. Implementing the quantum neural network in a real quantum computer, and
3. Comparing the runtime analytically and numerically to an FFNN.

To implement step 1., a further literature review is merited. We will understand how to derive a general phase Hamiltonian and hone in on a neural network derivation.

For step 2., we will use IBM's quantum experience, which allows access to quantum computers up to 20 qubits, to implement our proposed algorithm.

Finally, for step 3., we will attempt to derive the runtime of a QAOA neural network with analytical observation and experimental results.

The application of a quantum neural network has already shown significant results with a Helmholtz machine[12], but it was restricted by generating images – which requires qubit measurements. Since an FFNN is only required to output predictions of data[3], which is significantly less in size than the output of a Helmholtz machine, there is a chance we can improve upon current research in quantum neural networks. While we need to iterate by gradient descent to improve the accuracy of the hybrid FFNN, the solution would ideally optimize the original time complexity of an FFNN model (primarily steps one through three).

Unlike Benedetti, Realpe-Gomez, and Perdomo-Ortiz, we will have access to a 15-qubit quantum computer through IBM's quantum experience. Additionally, we may not have the restrictions explained by Beer et al. due to the hybrid nature of our model.

The implications of reducing the complexity of an FFNN are extremely large; an FFNN with an exponentially reduced run-time could convert millions of high-dimensional data within milliseconds. Armed with the knowledge presented thus far, we hope to apply our proposed algorithm toward the same image data presented in our implementation[3].

ACKNOWLEDGMENTS

I wish to acknowledge the support of the UNLV honors college for the encouragement of this proposal 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 department, and Dr. Brendan Morris from the Department of Electrical and Computer Engineering.

[1] M. Mohanraj, S. Jayaraj, and C. Muraleedharan, Renewable and Sustainable Energy Reviews **16**, 1340 (2012).

[2] B. K. Bose, IEEE Transactions on Industrial Electronics **54**, 14 (2007).

[3] E. Serrano, pneumonia identification neural network (2020).

[4] M. Egmont-Petersen, D. de Ridder, and H. Handels, Pattern Recognition **35**, 2279 (2002).

[5] B. Zygelman, *A First Introduction to Quantum Computing and Information* (Springer International Publishing, Gewerbestrasse 11, 6330 Cham, Switzerland, 2018) p. 233.

[6] I. H. Deutsch, PRX Quantum **1**, 020101 (2020).

[7] A. M. Childs, R. Cleve, E. Deotto, E. Farhi, S. Gutmann, and D. A. Spielman, Proceedings of the thirty-fifth ACM symposium on Theory of computing - STOC '03 10.1145/780542.780552 (2003).

[8] V. Dunjko and H. J. Briegel, Reports on Progress in Physics **81**, 074001 (2018).

[9] A. Abbas, D. Sutter, C. Zoufal, A. Lucchi, A. Figalli, and S. Woerner, The power of quantum neural networks (2020), arXiv:2011.00027 [quant-ph].

[10] K. Beer, D. Bondarenko, T. Farelly, T. J. Osborne, R. Salzmann, D. Scheiermann, and R. Wolf, Nature Communications **11**, 808 (2020).

[11] M. Schuld, I. Sinayskiy, and F. Petruccione, Quantum Information Processing **13**, 2567–2586 (2014).

[12] M. Benedetti, J. Realpe-Gómez, and A. Perdomo-Ortiz, Quantum Science and Technology **3**, 034007 (2018).

[13] T. Menneer and A. Narayanan, *Quantum-inspired Neural Networks*, Tech. Rep. (Pennsylvania State University, 1995).

[14] M. P. Harrigan, K. J. Sung, M. Neeley, K. J. Satzinger, F. Arute, K. Arya, J. Atalaya, J. C. Bardin, R. Barends, S. Boixo, and et al., Nature Physics **17**, 332–336 (2021).

[15] S. Hadfield, Z. Wang, B. O'Gorman, E. G. Rieffel, D. Venturelli, and R. Biswas, Algorithms **12**, 34 (2017).