Toward A Quantum Neural Network

ERICK SERRANO, University of Nevada Las Vegas - Department of Physics and Astronomy, USA

Machine learning has become an indispensable tool for popular applications such as Netflix's recommender systems, Google's search engine, and Tesla's self-driving vehicles. Though the predictive power of deep learning is well documented, researchers seek to enhance the capabilities of the classical neural network (NN) paradigm via the introduction of algorithms that rely on the principles of quantum mechanics[3, 13]. In this report, we explore the potential of quantum algorithms and argue for the feasibility of building a quantum neural network (QNN) in the near future.

First, we provide a short introduction to the NN model and a history of its development. Its genesis arose in the 1950s via the introduction of the perceptron model, a learning protocol introduced by Frank Rosenblatt that laid the groundwork for the modern, generalized NN model. We briefly discuss its mathematical structure and introduce and discuss a real-life application of a classical NN that is trained using a set of x-ray images from a patient cohort. Using the NN learning protocol and the x-ray data, we determine whether a patient has healthy or diseased lungs.

We describe each stage of the NN learning and prediction process; image processing, forward-propagation, prediction-making, cost function calculation, back-propagation, and gradient descent.

We explore the shortcomings of the method of gradient descent, a central feature of the learning algorithm. We show how classical gradient descent requires prodigious time resources. Because gradient descent scales poorly as the size of the NN and as the volume of learning data increases, there is a need for improvement and optimization.

To address this need, we explore possible improvements or replacements to gradient descent via quantum protocols. Our long term goal is the realization of a QNN model that will surpass the capabilities of the classical NN.

We briefly explore the key differences between quantum and classical computing. We then discuss why these differences could allow for the development of a quantum algorithm suitable for optimizing or replacing classical gradient descent. The proposed QNN model is not entirely quantum; we still use classical methods to compute the prediction and cost function. However, the quantum method would ideally replace gradient descent in training the QNN. Further research is required to integrate the proposed quantum methods into a NN to create an efficient QNN.

Additional Key Words and Phrases: Quantum Information, Quantum Computing, Neural Networks, Machine Learning, Quantum Neural Networks

ACM Reference Format:
Erick Serrano. . Toward A Quantum Neural Network. In . ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

1.1 Origins of Neural Networks

The first formal description of "intelligent" machines was introduced in the mid-twentieth century, decades before researchers attempted to integrate neural networks into quantum computers. In the 1950s, Alan Turing argued that

Manuscript submitted to ACM

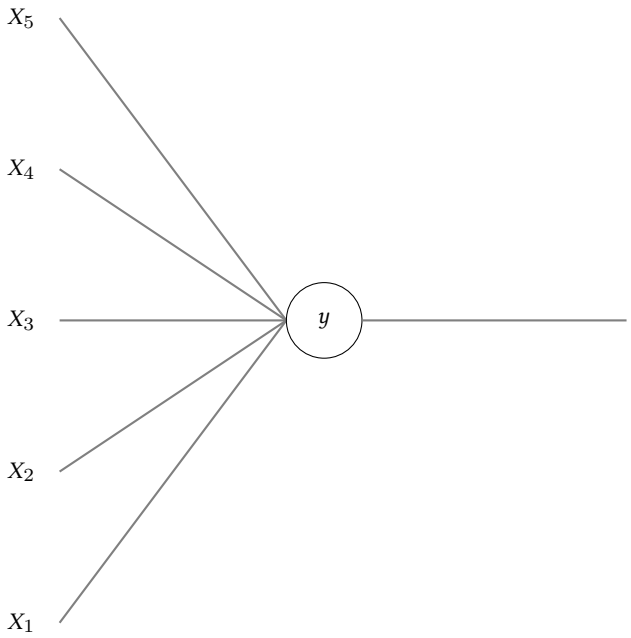

Fig. 1. Example of a perceptron with five input features, $X_1, X_2 ... X_5$. The node, y calculates a single, new number from the input features, maps it using an activation function[2], and then uses the mapping to make a final prediction. The number of input features is arbitrary for a perceptron.

modern digital computing technology holds the promise for such a capability. Turing also proposed a method, called the Turing test, by which one can gauge the efficacy of a machine to demonstrate degrees of intelligence[7].

Turing test aside, the first machine learning algorithms that could demonstrate "intelligence" appeared in the latter half of the 20th century. In 1958, Frank Rosenblatt introduced the perceptron, a model that is a precursor to the modern NN [7]. The input of the perceptron is a training set composed of a sequence of digits, where each set of digits represents a feature of the set. For example, we can separate a gray-scale image into individual pixels, and each pixel can be partitioned into numbers that denote its color. For a given pixel, 0 could represent a black background and 255 a white background. Given a picture composed of 1024 pixels, the sequence of numbers, ranging from 0 to 255, represent all 1024 features. Thus, the goal of the perceptron is to use every sequence of numbers, or every input feature, to infer a binary-valued decision. Known as binary classification, the perceptron predicts the answer to a simple yes or no question, or binary decision problem. Rosenblatt's model paved the way for a generalized version of the perceptron, which is now recognized and identified as a neural network. Figure (1) illustrates a perceptron with 5 input features, labeled by $X_1, X_2, ...X_5$. Every input feature serves as input to a node, represented by the circles. The node is a mathematical construct that simultaneously processes every input feature. A given node represents a set of weights and biases, $w_1, w_2, ..., w_5$ and b_1 that are used to convert every feature into a single value, a 1 or 0. The node in figure (1) converts every feature into a single value using an activation function[2], which we discuss further at a later point.

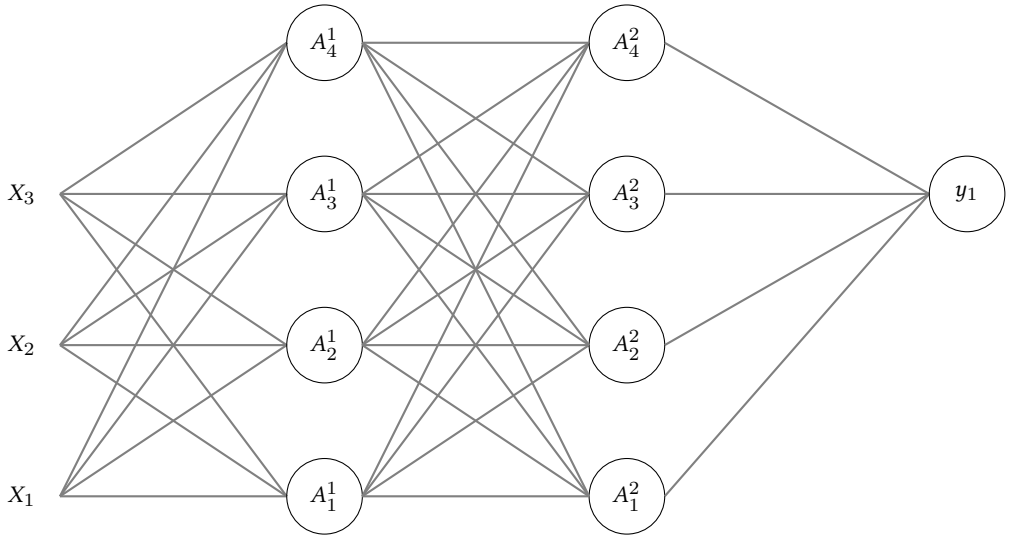

Fig. 2. The figure is a NN Structure with one input, hidden, and output layer. There are four nodes per input and hidden layer, and there is one node per output layer. The NN is due to process three input features X_1, X_2, X_3.

Note that the number of output, hidden, and input nodes can vary independently of each other. The number of hidden layers and input features also change.

1.2 Modern Neural Networks

Figure (2) illustrates the general structure of a NN. It differs from a simple perceptron in that the NN can contain multiple layers of nodes aside from the single output node, y_1. The architecture of the NN is partitioned into three sections: the input, the hidden, and the output layer. Figure (2) has one input and one hidden layer containing multiple nodes and one output layer with a single node. The input features in figure (2), X_1, X_2, X_3, are converted by the first layer into a new set of features: $A_1^1, A_2^1, A_3^1, A_4^1 = A^1$, where A^1 represents the new features of the first layer. This process is repeated until we traverse through the final hidden layer (if any), at which point we have reached the terminus of the process. The final set of features are then converted into a single value, and a prediction is made by the final node, labeled y_1 in the figure. After making a prediction, the model improves its accuracy by adjusting its weights and biases according to a benchmark known as the cost function and procedures known as back-propagation and gradient descent[8]. In the discussion below, we elaborate on the processes and features needed to create a proper NN model: image processing, forward-propagation, the cost function, back-propagation, and gradient descent.

1.3 Implementing a Neural Network

To get a better sense of how a neural network operates, we introduce a specific example of a NN application[14]; the diagnosis of pneumonia in a chest x-ray image is such an application. We used a web resource called Kaggle[10] that provided a collection of x-ray images of patients' lungs. The data also specified whether an x-ray image represented a healthy or diseased lung. We used that data to train a NN to discern the x-ray image of a healthy lung from that of an unhealthy lung. The output of the NN is either a yes, indicating that the patient presents symptoms of pneumonia, or a no, indicating a healthy lung.

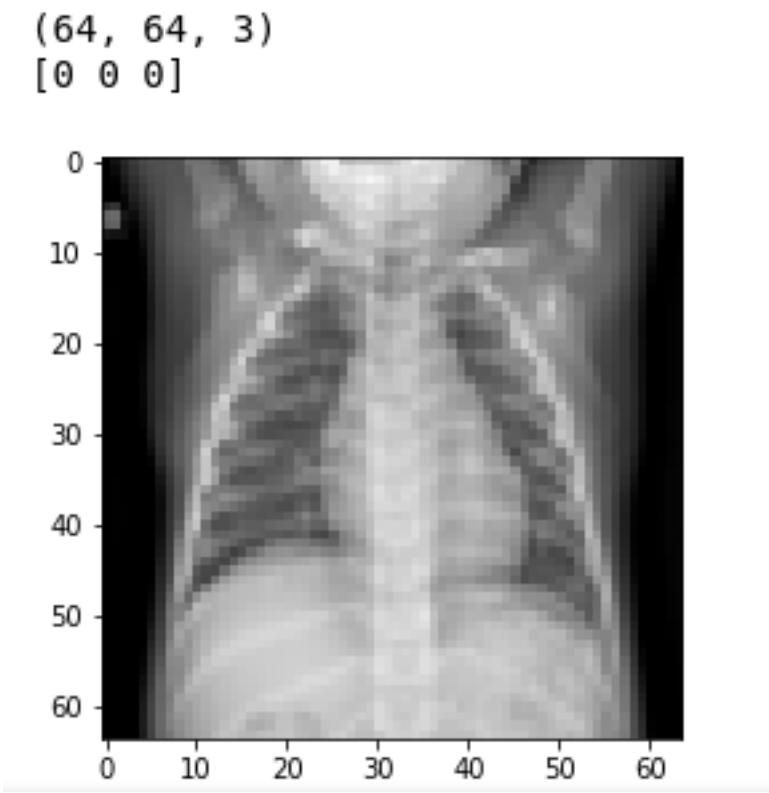

Fig. 3. The figure is a training example of a chest x-ray. The first set of numbers above the image denotes the dimensions and color values (in this case, the dimensions are represented by a 64×64 resolution and 3 RGB values), and the second set of numbers above the image denotes the RGB values for the first pixel. In this case, the first pixel in the x-ray image is black, since each value is zero.

1.3.1 Image Processing. Using the Kaggle input data, we gauge the efficacy of a NN to perform binary classification. Before the x-ray images are given to the NN, the image data is formatted to fit the required input of the NN. In our example, the NN processes each pixel's RGB values. In figure (3), we have two lists of values and a sample image of a patient's x-ray. The first list of values identifies that the x-ray is a 64×64 pixel image with 3 values, RGB values, at each location. Thus, this single image contains $12,288$ $(64 \times 64 \times 3)$ input features that are used by the NN.

In figure (4), we provide a snippet of code that displays the number of features in a single image, $12,288$ features, and the number of x-ray images, 10. This leads to $12,288 \times 10$ features used by the NN for computation. Figure (4), then, expresses every feature in a matrix, known as the input matrix or x-matrix. Along with the x-matrix is a corresponding output matrix, the y-matrix. The y-matrix is a $(1 \times m)$ table of numbers whose entries label a corresponding image in the x-matrix. For example, if the first label, $y_0 = 1$, then the first image, x_0, is an infected lung. When the NN's prediction does not align with the y-matrix, then we can infer that the NN has made an error. Because there are 10 x-ray images, there are also 10 labels – displayed in figure (4) as y_{final}. Each image and label is interpreted as a column of the x and y-matrix, and each row of the x-matrix is a feature of an image.

In [4]:
```
"""
Created a seperate, much smaller training_set to test pre-processing functions
"""

#TEST FILES (5 images per file)

pathOne = "chest_xray/trainTEMP/normal"
pathTwo = "chest_xray/trainTEMP/pneumonia"
x_temp, y_temp = load_xy_set(pathOne, pathTwo)

#Verify contents and shapes match each other
#x_train.shape ==> (total_training_examples,width,height,RGB)
print("x_final: ",x_temp.shape)
#print("x_final[0]: ", x_temp[:,0])

#y_train.shape ==> (1,total_training_examples)
print("y_final: ",y_temp.shape)
#print("y_final[0]:", y_temp[:,0])
```

```
x_final:  (12288, 10)
y_final:  (1, 10)
```

Fig. 4. The dimensions of the x and y-matrices are displayed here. There are 10 images that each have a 64×64 resolution, with 3 RGB values at each pixel. Thus, the x-matrix has $64 \times 64 \times 3 = 12,288$ rows, representing $12,288$ features of an image, and 10 columns, representing 10 images to process. Meanwhile, there is a corresponding y-matrix with simple 1's and 0's. The y-matrix is a "label" for the image. In other words, if the first column, $y_0 = 1$, then x_0 presents a lung with pneumonia.

1.3.2 forward-propagation. Each image we processed remains in a single column of the x-matrix. In the neural network, the column data is inserted into every input node to begin what is called "forward-propagation." In forward-propagation, a node performs on old input features with a linear transformation, followed by the application of an activation function[2], to generate new input features for subsequent layers. Equation (1) quantifies this linear transformation, which uses weights, biases, and input data. The input data is simply the x-matrix. However, the NN is required to initialize a list of values for the biases and weights.

The biases and weights are initialized at small, random values to encourage an accurate prediction. The biases are initialized as a list of column vectors of size $(n \times 1)$, where n is the number of nodes in the respective layer. Using figure (2) as reference, there would be three column vectors b_1, b_1, b_3. The first column vector, b_1, would be added to the first linear combination and produce the layer features, Z^1; we apply the other bias vectors would similarly. Meanwhile, the weights are initialized as a list of matrices of size $(n \times m)$. Here, m is the number of input features to be computed. The size of m varies depending on the layer of the NN; using figure (2) again, the hidden layer would have a weight matrix, w_2, of size (4×4); there are four nodes in the hidden layer, so $n = 4$, and there are four features from the preceding layer. At the output layer, the weight matrix and bias column vector are of size $1 \times m$ and 1×1, respectively

$$Z^1 = w_1 X + b_1 \tag{1}$$

From our practical example, let us also understand the intuition behind the choice of the matrix dimensions, $(12,288 \times 10)$ for X, $(n \times 12,288)$ for w_1, and $(n \times 1)$ for b_1, and let us understand why n is of arbitrary value. Given the nature of equation (1), the dimensions of the matrix, w_1, must partially match the dimensions of the matrix, X, as explained by the rules of the dot product[1]. The dimensions we provided for X and w_1 follow the requirements of the dot product; when we multiply the two, we are left with $w_1 X$, a matrix with $(n \times 10)$ dimensions. When we add the

bias, b_1, to the equation, we are left with a matrix previously defined by equation (1): $w_1X + b_1$. The dimensions of $w_1X + b_1$ are the same as w_1X, but the values within are modified by matrix addition. The dimensions of b_1 and w_1X do not meet the rules of matrix addition, but this is corrected in our implementation using Python, a programming language. Python applies the bias to each column of w_1X; this is called broadcasting[12]. Thus, the previously stated sizes of X, w_1, and b_1 are appropriate for our NN model. We now explain and justify the value chosen for n.

The value, n, is dependent on the number of nodes within a layer; in figure (2), n has the value 4 for the first layer, as there are four input nodes; thus, A_1 would have the dimensions (4×10). The hidden layer in figure (2) also has four nodes, implying that $n = 4$; thus, the next matrix, A_2, would have the dimensions (4×10), similar to A_1. The perceptron in figure (1) implies that $n = 5$ at the input layer. We can use different values of n to create unique architectures without losing the efficacy of a NN model. Thus, we recognize that n, at any layer, is arbitrary.

After we use equation (1) at any point in the forward-propagation process, we are also required to apply an activation function[2]. The most popular activation function for neural networks is currently the ReLu function, first introduced in 2012 by AlexNet[7]; others have been commonly used too, such as the classic sigmoid function[8] or the hyperbolic tangent function[4]. While we will need the sigmoid function for the output node, we can freely apply an activation function to other nodes. In our practical example, we will be using the sigmoid function for the output node and the ReLu function for every other node. The following equations represent the sigmoid, ReLu, and hyperbolic tangent functions, respectively. For some i-th layer, A^i:

$$A^i = \sigma(Z) = \frac{1}{1 + e^{-Z}},$$

$$A^i = ReLu(Z) = Z,$$

where Z > 0,

$$A^i = ReLu(Z) = 0,$$

where Z <= 0, and

$$A^i = Tanh(Z),$$

The final product of our linear combination and activation function is A^i with $(n \times m)$ dimensions. The new matrix means we have passed through the input layer, and we have acquired new data points to be processed by the next layer.

When we look at figure (5), we confirm that the dimensions we evaluated for w_1 and b_1 are correct. In this example, $n = 5$, which implies that there are five nodes in the input layer. Using equation (1) as our guide and the dimensions of X in figure (4), we conclude that A^1 must be a matrix with dimensions (5×10). In the next layer, the process is repeated to calculate A^2, where A^1 is now a replacement for our input, X: $A^2 = ReLu(w_2A^1 + b_2)$. This process repeats until it has reached the output layer, at which point the NN makes a prediction.

1.3.3 Making a prediction. We now discuss the output layer, which is a vital part of the prediction process. The output layer maps the linear combinations previously calculated into a single value. If we continue to use our x-ray example, we have some final set of features, A^k, of size $(1x10)$; the reason that there is only a single row is that the output layer consists of a single node, similar to figures (1) and (2). Instead of using yet another linear combination, the output layer applies an activation function to help us make a prediction. In our example, we will be using the sigmoid function. The sigmoid function maps our values between 0 and 1, which is exactly what we need for binary classification. If an x-ray image is mapped to a value of at least 0.5, then we identify that the patient has a case of pneumonia. Otherwise, the

```
In [8]: """
        Test the initialize_parameters function:
        -Confirm that it produces the proper shapes for a given set of layer_dims

        Pseudo-code:

        for x in range(1,len(layer_dims)):
         weight.shape = (layer_dims[x], [x-1])
         bias.shape = (layer_dims[x], [x-1])

        """
        layer_dims = [x_temp.shape[0],5, 4, 3, 1]
        counter = 1
        print("X_input: ", x_temp.shape[0])

        parameters = initialize_parameters(layer_dims)

        #Test for proper bias & weight shapes
        for x in parameters:
            print(x, ": ", parameters[x].shape)

X_input:  12288
W1 :  (5, 12288)
b1 :  (5, 1)
W2 :  (4, 5)
b2 :  (4, 1)
W3 :  (3, 4)
b3 :  (3, 1)
W4 :  (1, 3)
b4 :  (1, 1)
```

Fig. 5. The figure shows the dimensions of the weights and biases in every layer of a typical NN model. Note that the number of rows contained in a weight matrix denotes the number of nodes for the corresponding layer. For example, w_1 shows that there are five nodes in the first layer.

patient does not present pneumonia. Using the sigmoid function, we can answer a simple question: "does the patient have pneumonia?"

1.3.4 The Cost Function. However, the NN process does not conclude here. After making our prediction, we must ensure that this prediction is accurate. We now introduce the cost function, which gauges the accuracy of the NN prediction.

The cost function measures the accuracy of a NN. The higher the value of the cost function, the more error a NN contains in its prediction and forward-propagation process. A low value, or a small "cost," in the cost function implies that the NN is sufficiently accurate. The following is an equation which we use to evaluate the cost of the NN[14]:

$$J = (-1/m) * (\sum_{i}^{m} y^i log(PredictionX^i) + (1 - y^i) log(1 - PredictionX^i) \tag{2}$$

In equation (2) m is the number of training examples. Also, $PredictionX^i$ is the prediction of the NN for the i-th example, and y^i is the actual label for the image. In our implementation, this means we have to compute 12, 288 features for each image. For example, if there is an image with the label $y^i = 1$, then the cost function can improve, or rather, decrease in value if $PredictionX^i = 1$. However, if $PredictionX^i = 0$, then the cost function increases in value.

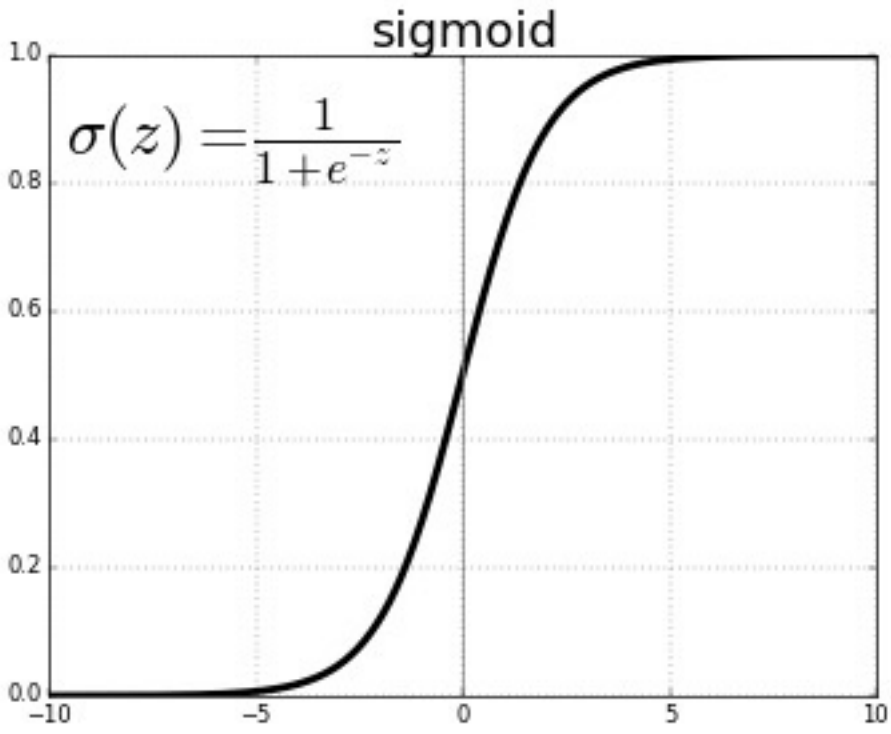

Fig. 6. Graph of the sigmoid function. For NNs using binary classification, if a point is at least 0.5 or higher, the model identifies this example positively (prediction = 1). Otherwise, the program predicts the example negatively (prediction = 0) (the figure is taken from Quora).

.

Suppose that the neural network made an incorrect claim; this is especially dangerous for patients who have pneumonia but are not correctly diagnosed. Thus, we have an incentive to minimize the cost of the NN. The job of the cost function is to quantify how large an error the NN model makes. The higher the value of the cost function, the more likely the NN is to make a mistake. Because of this, minimizing the cost function is of utmost importance, and we do this by using the back-propagation algorithm and gradient descent.

1.3.5 Back-propagation And Gradient Descent. The modern NN uses back-propagation[7] to minimize the cost function. Unlike forward-propagation, back-propagation begins at the output node and traverses the NN until it has reached the input nodes. Back-propagation produces derivatives of the weights and biases for the minimization of the error rate, or the cost. Meanwhile, the purpose of gradient descent is to apply every derivative iteratively and also minimize the cost[11]. The back-propagation and gradient descent processes are clearly dependent on one another, but these are two distinct processes that aim to minimize the cost. Figure (7) depicts a global minimum of the cost function which can be found by back-propagation and gradient descent. After applying back-propagation and gradient descent, the NN's

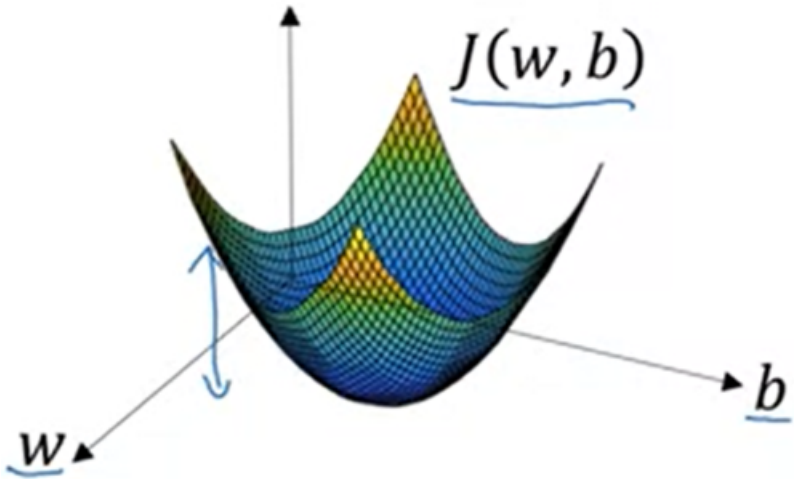

Fig. 7. three dimensional graph of the cost function in terms of the weight and bias terms (taken from Andrew Ng's Neural Network course on Coursera. The three dimensional graph clearly illustrates a point at which we have reached the minimal cost function, but this is not necessarily a global optima[6].

next prediction will have a decreased error rate. We now elaborate on the back-propagation process followed by the gradient descent technique.

Before we explain back-propagation, we introduce a new set of equations. The following formulas represent the entire process of back-propagation.:

$$dA_k = W^T * dA_{k+1} \tag{3}$$

$$dW = (dA_{k+1} * A_k^T)/m \tag{4}$$

$$db = \sum dA_{k+1}/m \tag{5}$$

Equations (3), (4), and (5) show that the weight and bias derivatives depend on the derivatives of the layer ahead of it, dA_{k+1}. This dependence means that we start from the end of the NN forward propagation and work our way backward – hence the term back-propagation. The formula for the first derivative calculation is $dA_l = -(\frac{Y}{A_l} - \frac{1-Y}{1-A_l})$, where l is the index of the last layer, dA_l. Because equation (4) also uses the features produced in every layer, $A_1, A_2, ...A_l$, we must preserve these features. Thus, we cache the matrices beforehand, during the forward propagation process, for the back-propagation process.

If we continue to compute and cache the derivatives until we reach the input layer, our cache will contain a list of "gradients." The gradients of the NN are derivatives corresponding to the weights and biases, or "parameters," of the model. Then, we can adjust our parameters to minimize the cost. This concludes the back-propagation process; We now place focus on gradient descent.

Gradient descent refers to parameter adjustments and iteration until we achieve convergence. Let j and h be the number of parameters and gradients, respectively, that need to be updated. Additionally, let k be the number of updates made to the parameters and gradients; in other words, we update j and h parameters and gradients for k iterations. Given a large size of calculations and iterations, it is evident that gradient descent requires prodigious CPU resources. Our practical example, which predicted pneumonia in an patient's x-ray, lasted approximately 1 hour to process 3800

images before it achieved the desired degree of accuracy. Thus, a shortcoming of gradient descent is its large time consumption. When training the NN, it takes hours to process low-resolution images in the thousands, but it can take days, weeks, or months to process millions of high-resolution images. These large computations result in a "time complexity" for each process described in Table 1. Using asymptotic analysis, explained in Big O' notation[15], we determine the time complexity, or the time it takes to complete an algorithm without accounting for hardware and software restrictions; we elaborate on the time complexity of gradient descent in equation (6).

Algorithm	Time Complexity
Iteration	$O(k)$
updating gradients	$O(j)$
updating parameters	$O(h)$

Table 1. This figure details the time complexity, in Big O notation for our asymptotic analysis, of back-propagation and gradient descent. Gradient descent depends on the number of iterations k, the number of gradients, j, and the number of parameters, h. The parameters are the list of weights and biases in the NN model, and the gradients are the list of corresponding derivatives.

Each parameter of the NN model has a corresponding gradient (dw_1 <=> w_1,db_1 <=> b_1, etc...); therefore, the time complexity in Table 1 implies that $j = h$ and $O(j) = O(h)$. Because we update the parameters and gradients in a single iteration, we can conclude that the two time complexities are sequential and can be merged using addition. Gradient descent, however, updates the gradients and parameters k times. The result is that we multiply the time complexity $O(j + h) = O(2j)$ by k. Equation (6) simplifies the time complexity of the back-propagation process:

$$O(k)(O(j) + O(h)) = O(k)(O(j + h)) = O(k)O(2j) = O(k2j) \tag{6}$$

Thus, we conclude that the time it takes back-propagation to improve is reliant on the size of the neural network and the number of iterations in which we initiate back-propagation. The size of the neural network affects the number of gradients and parameters we must modify, and the number of iterations requested to compute the gradients and parameters affects the time complexity of equation (6) as well. We now explore and compare the differences between classical and quantum computation.

2 TOWARD A QUANTUM NEURAL NETWORK

2.1 Quantum Vs. Classical Computation

Recall that the NN "learns," by minimizing the cost function implemented by the method of gradient descent; unfortunately, gradient descent has poor scaling properties, making quantum alternatives for error minimization problems highly desirable. As the spatial complexity of the learning data and NN increases, the time resources required for gradient descent increases as well. The application of quantum methods to replace classical gradient descent could improve the efficiency of the learning cycle.

The state of a classical system is described by the assignment of a sequence of bit values. Figure (8) illustrates the range of a classical bit, a single state, 1 or 0, at any instance in time.

The quantum analog of a classical bit, the quantum bit, or "qubit"[16], can exist, as illustrated in figure (9), in more than a single state at a given moment in time. Though it is still possible for a qubit to be in a definite state, either $|1\rangle$ or $|0\rangle$, it can also be found in a superposition, $\alpha|0\rangle + \beta|1\rangle$[16]. The superposition property in quantum mechanics,

$$[0] <=> [1] \tag{7}$$

Fig. 8. The figure displays the values for a classical bit. Only one number at a time, 1 or 0, can be shown inside the bit at any point in time.

$$|0\rangle <=> |1\rangle <=> \alpha|0\rangle + \beta|1\rangle \tag{8}$$

Fig. 9. The figure displays the values for a quantum bit or a "qubit." The numbers 1 or 0 can be defined inside the bit one at a time or at the same time. In other words, we can have a 0 value, a 1 value, or a superposition [16] of either value at any point in time. α and β define a generalization of a single qubit, where if α is zero, then we are left with $|1\rangle$ and vice versa if β is zero

along with quantum entanglement, has been exploited in protocols that may enhance the efficiency of a QNN over its classical counterpart[3, 5, 9, 13].

The Grover and Shor algorithms[16] are transformative in their use of the quantum properties of massive parallelization and interference to solve problems, out of reach by classical computers, in cryptography and unsorted search. Encouraged by the success of both algorithms, we believe the aforementioned quantum properties could also be exploited to enhance and realize the promise of machine intelligence.

We propose to further explore NN optimization by employing quantum methods, which would replace the inefficient gradient descent protocol.

2.2 Expected Results and Conclusion

We will continue our investigation into quantum algorithms for application to efficient neural networks. Such algorithms, if realized, could ideally replace gradient descent, an inefficient and time-consuming classical algorithm. For the machine learning pneumonia diagnostic discussed in the previous sections, a quantum algorithm could conceivably produce accurate diagnosis in a much shorter time frame. Further research needs to be conducted to produce conclusive results.

3 ACKNOWLEDGMENTS

I would like to thank Dr. Bernard Zygelman, my mentor for the duration of this project. Dr.Zygelman introduced me to the research environment and helped me understand the expectations and level of commitment required for research opportunities.

I would also like to thank the C.A.E.O. Summer Research Institute for sponsoring my project. With their assistance, I was afforded the opportunity of pursuing research in machine learning and quantum computing.

REFERENCES

[1] Khan Academy. 2015. Matrix multiplication dimensions (article). https://www.khanacademy.org/math/precalculus/x9e81a4f98389efdf:matrices/x9e81a4f98389efdf:properties-of-matrix-multiplication/a/matrix-multiplication-dimensions

[2] Afshine Amidi and Shervine Amidi. 2020. Deep Learning cheat sheet. (2020). Retrieved October 10th 2020 from https://stanford.edu/~shervine/teaching/cs-229/cheatsheet-deep-learning

[3] Kerstin Beer, Dmytro Bondarenko, Terry Farelly, Tobias J. Osborne, Robert Salzmann, Daniel Scheiermann, and Ramona Wolf. 2020. Training deep quantum neural networks. *Nature Communications* 11 (Feb 2020), 808. https://doi.org/10.1038/s41467-020-14454-2

[4] Science Direct. 2020. Hyperbolic Tangent Function. Retrieved October 13th 2020 from https://www.sciencedirect.com/topics/mathematics/hyperbolic-tangent-function

[5] Vedran Dunjko and Hans J Briegel. 2018. Machine learning and artificial intelligence in the quantum domain: a review of recent progress. *Reports on Progress in Physics* 81, 7 (Jun 2018), 074001. https://doi.org/10.1088/1361-6633/aab406

[6] Mathworks. 2020. *Local vs. Global Optima.* The Mathworks, Inc. https://www.mathworks.com/help/optim/ug/local-vs-global-optima.html

[7] Hugo Mayo, Hashan Punchihewa, Julie Emile, and Jack Morrison. 2018. *History of Machine Learning.* Imperial College London. Retrieved September 30th 2020 from https://www.doc.ic.ac.uk/~jce317/history-machine-learning.html

[8] Hugo Mayo, Hashan Punchihewa, Julie Emile, and Jack Morrison. 2018. *Introduction to Neural Networks.* Imperial College London. Retrieved October 1st 2020 from https://www.doc.ic.ac.uk/~jce317/introduction-neural-nets.html

[9] Tammy Menneer and Ajit Narayanan. 1995. *Quantum-inspired Neural Networks.* Technical Report. Pennsylvania State University.

[10] Paul Mooney. 2017. Chest X-ray Images (Pneumonia). (2017). https://www.kaggle.com/paultimothymooney/chest-xray-pneumonia

[11] Andrew Ng. 2017. *Gradient Descent - Neural Networks and Deep Learning.* DeepLearning.AI. https://www.coursera.org/lecture/neural-networks-deep-learning/gradient-descent-A0tBd

[12] Python. 2020. Broadcasting - NumPy. (2020). https://numpy.org/doc/stable/user/basics.broadcasting.html

[13] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. 2014. The quest for a Quantum Neural Network. *Quantum Information Processing* 13, 11 (Aug 2014), 2567–2586. https://doi.org/10.1007/s11128-014-0809-8

[14] Erick Serrano. 2020. pneumonia identification neural network. (2020). https://github.com/erickserr125/pneumonia_identification_neural_network

[15] Wikipedia. 2020. Big O notation. https://en.wikipedia.org/wiki/Big_O_notation

[16] Bernard Zygelman. 2018. *A First Introduction to Quantum Computing and Information.* Springer International Publishing, Gewerbestrasse 11, 6330 Cham, Switzerland. 233 pages. https://doi.org/10.1007/978-3-319-91629-3