Quantum Binary Neural Network: Version Space Implementation

Divye Jain, Harsh Patwari, Jack Khuu

University of Washington

Paul G Allen School of Computer Science & Engineering, Seattle, WA

{*dpjain, hpat1996, jackkhuu*} *@cs.washington.edu*

Abstract

We describe an algorithm, and an implementation (in the Q# programming language) of Quantum Binary Neural Network. The overall model is based on classical Binary Neural Networks (BNN), but the weights are found by a Quantum subroutine.

Essentially we use Grover's algorithm to search over all hyper-planes of the weights and find one with sufficient accuracy. Our work primarily contributes to the implementation of a marking oracle, that is able to mark the "correct" set weights. We show that our algorithm and implementation works, by trying to predict the correct weights for the XOR network

1. Introduction

Neural networks for machine learning are extremely powerful tools of computing. One common complaint of neural networks is the time-cost of training models. Quantum, on the other hand, is often associated with the idea of improving the speed of classical algorithms. Quantum neural networks (QNN) is the intersection of these ideas. Through the use of quantum algorithms, classically slow operations in neural networks can be substituted with a faster equivalent.

QNNs have been theorized and described by many papers, but only a few describe the process of implementation in detail and even fewer have fully implemented a quantum neural network. While there are theoretical proofs of the superiority of quantum machine learning algorithms over classical equivalents, such as the case of perceptron models, its implementation and training are far from trivial to implement[1,2].

We provide a description of our implementation of a Quantum Binary Neural Network (QBNN). The basis of a QBNN is an implementation of a classical Binary Neural Network (BNN) in quantum space. We chose to implement a BNN specifically due to constraints that we discuss later in the paper.

2. Background

We start with a basic explanation of the components and algorithms used in our implementation of QBNN.

2.1. BNN

BNN is a special type of neural network where the weights and activation functions in the model are binary at runtime[3]. By reducing a model to a BNN, it can significantly reduce the memory size required and can allow for many arithmetic operations to be done through bit operations[3].

2.2. Grover's Algorithm

Grover's algorithm is a quantum search algorithm that allows for near perfect searching in $O(\sqrt{N})$ time, where N is the size of the search space[4]. This is done by creating an equal superposition over all states, marking the desired states by flipping the amplitude sign, reflecting about the mean to amplify the amplitude of the marked states, then measuring the final state to get one of the targets with a high probability[4].

3. QBNN

In a classical neural network, the model learns by passing input through a network of nodes, so that each node can learn the transformations/weights that need to be applied when evaluating new data. Specifically in a BNN, these weights are binary.

We considered two ways to improve BNN using Grover's Algorithm: we can search for data that updates the weights or we can search for a hyperplane that separates the data[1].

The first way involves marking and searching for the data points that will update the weights for the model. While this is completely valid, if the algorithm is searching to update the weights, why not just search for the weights?

That is the second tactic we considered, and ultimately decided to build. The idea is to use Grover's to search for the hyperplane separating the data. That is, do a version space search to find the sequence of weights that would separate the data.

Algorithm 1 QBNN

TrainModel (input, actual labels, hidden layers, desired accuracy, number iterations):

$\boldsymbol{w} \leftarrow H(|00...0\rangle)$

Search(input, actual labels, hidden layers, $\boldsymbol{w}$, desired accuracy, number iterations)

Measure $\boldsymbol{w}$

Algorithm 2 Grover's Search

Search (input, actual labels, hidden layers, weights, desired accuracy, number iterations):

for i in 0..number iterations:

MarkingOracle(input, actual labels, hidden layers, $\boldsymbol{w}$, desired accuracy)

Reflect $\boldsymbol{w}$

Algorithm 3 MarkingOracle

MarkingOracle (input, actual labels, hidden layers, weights, desired accuracy):

correct $\leftarrow$ [00...0]

for i in 0..Length(input):

let pred $\leftarrow$ Forward(input[i], hidden layer, $\boldsymbol{w}$)

if (pred == actual label[i]):

correct[i] = 1

if (percentage(correct) $\geq$ desired accuracy):

Mark $\boldsymbol{w}$

for i in 0..Length(input):

Adjoint Forward(input[i], hidden layer, $\boldsymbol{w}$)

3.1. Input

The *input* data can be passed as a 2D array containing the data points. An example of input could look like [[0, 0], [0, 1], [1, 0], [1, 1]]. This denotes that there are 4 data points with 2 features each. The data points are [0, 0], [0, 1], [1, 0] and [1, 1].

The *labels* can be passed as a 1D array where each index i contains the label for $input[i]$. An example of labels could look like [1, 0, 0, 1]. This denotes that the data point [0, 0] has label 1, [0, 1] has label 0 and so on.

The *hidden layers* can be passed as 1D array where each index i has the number of hidden nodes at the hidden layer i. An example of hidden layers could look like [2]. This denotes that there is only one hidden layer between the input and output layers. The hidden layer has 2 nodes.

The *desired accuracy* can be passed in as a double. A *desired accuracy* of 1.0 would mean that we want only the weights that predict all the labels for the input data correctly or a 100% accuracy.

The *number iterations* can be passed as an integer. If *number iterations* = -1, then the algorithm treats it as a hyperparameter and runs Grover's Search for various number of iterations and finally chooses the weight state that satisfies the desired accuracy condition. However, if the number of correct weight states is known, then we can run Grover's Search for a specific computed number of iterations.

3.2. Output

The output is one of the weight states that correctly predicts the input data with a desired accuracy. This weight state is returned as an integer array, where each element in the array represents the weight associated between two distinct nodes in the neural network.

3.2.1. Algorithm

We initialize the weights with the $|00...0\rangle$ state. The size of the weights can be computed given the number of features in each data point or the input dimension, the dimensions of the hidden layer and the output dimension. We then create a superposition over all the possible states for the weights by applying the Hadamard gate on each qubit of the weight vector. We pass the input data to the neural network and by applying the forward algorithm, we compute the predicted label for each data point. Given the actual labels and having computed the predicted labels, we can calculate the accuracy for each distinct combination of the weights and the predictions. We then move on to perform the Grover's Search algorithm. The first step of the search algorithm uses the marking oracle to flips the state of all the weights for which the computed accuracy is greater than or equal to the desired accuracy. In the second step, we reflect the states about the mean to amplify the probability of the marked states and consequently, decrease the probability of all the other states[1]. We repeat the Grover's algorithm for a certain number of iterations, which is one of the hyperparameters, to get the best probability of the marked states. Finally, we measure the weights to get one of the marked states with high probability and we return it.

3.3. Previous Work

The first paper that we encountered was "Quantum Perceptron Models"[1], which explains two ways of applying Grover's to a quantum implementation of perceptron. It goes into detail about the math behind the idea, so we used it for influence when deciding to do a hyperplane search over data point search.

Another paper was "Training a quantum neural network"[2], which explains the idea behind how to do the training process for a quantum neural network, but did not describe in depth how to implement such an algorithm. Our training process is similar to a combination of these last two papers.

A third paper was "Binarynet: Training Deep Neural Networks With Weights and Activations Constrained to +1 or-1", which is a classical version of our implementation. They describe a similar idea in a classical setting, and apply it to higher dimensions and with inputs that were generated from converting non-binary data into a binary state.

4. Grover Search Components

4.1. Preparation: Initital state

Since we are doing grover's search on all states of the weight. So our input state to Grover (Ψ_{in}) is just a uniform distribution over all states. This can be done by applying the Hadamard to each bit in the weights.

$$\Psi_{in} = \frac{1}{\sqrt{2}^n}(|w_1\rangle + ... + |w_{2^n}\rangle) \qquad n : \text{Number of bits required for the weights}$$

4.2. Marking Oracle

There are 3 major components to the marking oracle

1. Entagle Ψ with a Qubit vector $|c\rangle$

$$|c\rangle = \begin{bmatrix} c_1 \\ c_2 \\ ... \\ c_N \end{bmatrix}_{N\times 1} \qquad \text{N: The number of data points}$$

$$c_j = \begin{cases} 1 & \text{model “}w\text{” with input “}x_j\text{” predicted “}y_j\text{” correctly} \\ 0 & \text{model “}w\text{” with input “}x_j\text{” predicted “}y_j\text{” incorrectly} \end{cases}$$

   Essentially we should get, $\Psi = \alpha_1 |w_1\rangle + ... + \alpha_k |w_k\rangle \rightarrow \alpha_1 |w_1\rangle |c_1\rangle + ... + \alpha_k |w_k\rangle |c_k\rangle$
2. Next we can mark the “target” based on this $|c\rangle$ and the desired accuracy. For example, if we want at-least 80% accuracy, then we mark the target when $|c\rangle$ is filled with at-least “$\lceil 0.8 \times N \rceil$ 1's
3. Finally we need to un-entagle $|c\rangle$ and $|w\rangle$

4.2.1. Entangling $|w\rangle$ and $|c\rangle$

To understand this section, the reader should be familiar with how a forward pass works in a neural net. We are going to explain how to do it for one neuron then generalize it for the whole network.
Given that a neuron is taking input from a layer with 2 units, we can generalize the neuron output by:

$$\text{Neuron output} = \sigma(W_h^T X)$$

σ : any activation function,
W_h : weights stored in neuron “h”,
X : input to the neuron “h”, (the 1 is appended for the bias)

$$W_h = \begin{bmatrix} w_{h1} \\ w_{h2} \\ b_h \end{bmatrix}; X = \begin{bmatrix} x_{h1} \\ x_{h2} \\ 1 \end{bmatrix}$$

Now in the case of a BNN the w's and the x's are all 1's and -1's and the activation function is just the “sign” function. So essentially we can write the output of the neuron as being a “majority function” on the xnor output between the w's and the x's.

Now we know how do get the output of one neuron, so we can just do the same for all the neuron in a layer and then recursively pass the output of the layer as an input to the next until we get the output of the last layer, which is the “prediction” of the model. Finally we can compare the output prediction with the true prediction to set the value of $|c\rangle_i$ for each data point i.

4.2.2. Marking target from $|c\rangle$

This is can be done easily by applying “Controlled” X on the target depending on the 1's in the $|c\rangle$

4.2.3. Un-Entangling $|w\rangle$ and $|c\rangle$

This can be done by running the “forward” algorithm in reverse at the end of the marking orcale.

5. Impact

We think this project has a lot of impact in the Quantum community, because essentially we are introducing an algorithm that given binary input and a single binary output can predict a BNN that works, we can even think of algorithm as a declarative paradigm. Where we just define what we want in terms of the input and the output and it gives us a Quantum circuit (with some probability of-course, since we are using Grover's and setting the desired accuracy) that works on the specifies input/output.

6. Results

We initally tested the implementation to predict the outputs for the AND and OR gates for an input size of 2. These gates did not require a hidden layer. We got a 100% accuracy.

We tried to test our implementation on a more challenging problem, the XNOR gate. XNOR cannot be solved by a classical single layer perceptron algorithm and this is what makes this gate interesting. We successfully ran the algorithm to predict the output of the XNOR gate for an input size 2 and one hidden layer containing 2 nodes, with a 100% accuracy. The number of qubits required to run the algorithm was 25.

7. Conclusion

We have described our implementation of a QBNN through the use of Grover's Algorithm to determine the weights of the model. The validity of the model is reinforced by the ability to predict the correct weights for an XOR arrangement of data (a task impossible with a basic single layered perceptron).

7.1. Next Steps

One possible extension would be to increase the dimensions of the network to be more effective. As it currently stands, there is only one true hidden layer and 2 attributes due to qubit constraints. In order to add dimensions, it is worth trying to reduce the number qubits used to represent the model, to allow for the introduction of more layers and attributes (The physical algorithmic change is trivial).

Branching off of this would be testing the model using a more interesting input, either through the reduction of classical machine learning data sets into a binary form or to upgrade this model to a non-binary model (albeit making the model much more qubit expensive).

8. References

[1] Wiebe, N., Kapoor, A. & Svore, K. M. Quantum perceptron models. Adv. Neural Inform. Process. Syst. 29, 39994007 (2016)

[2] Ricks B, Ventura D (2004) Training a quantum neural network. In: Thrun S, Saul LK, Scholkopf B (eds) Advances in neural information processing systems, vol 16. MIT Press, Cambridge, MA, pp 10191034

[3] Courbariaux, Matthieu and Bengio, Yoshua. Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or-1. arXiv preprint arXiv:1602.02830, 2016.

[4] Figgatt, C. et al. Complete 3-qubit grover search on a programmable quantum computer. Preprint at https://arxiv.org/abs/1703.10535 (2017)