A deep Quantum Neural Network for solving the XOR problem

Andre Sequeira*

University of Minho, Braga, Portugal
andresequeira401@gmail.com

November 15, 2020

Abstract

The XOr problem is a classic problem in ANN research. It is the problem of using a neural network to predict the outputs of XOr logic gates given two binary inputs. This is well studied classic problem, however, in the field of quantum computing and quantum machine learning there isn't a clear implementation of a quantum neural network on an actual quantum processor that's able to solve the XOr problem, because there is a limitation on the pure quantum training algorithms. In this article we tackle this problem by implementing Quantum Dynamical Descent (QDD), a quantum version of a gradient descent training algorithm, suggested in [1] on the IBM ibmqx5 quantum processor.

I. The Perceptron Model

The perceptron is the fundamental building block of an Artificial Neural Network. It's purpose is trying to mimicing the biological neuron within our brains.
Proposed by Frank Rosenblatt in 1958, the perceptron is a *linear binary classifier*, this means that like in our neurons, it receives a set of inputs $\in \{0,1\}$ wich each input has a weight associated and apply some *threshold function* that outputs a value 0 or 1 (inactive/active). A graphical representation of the perceptron can be seen in the figure 1.

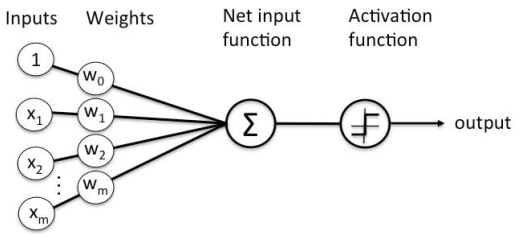


Figure 1: *The Perceptron Model*

*A thank you or further information

II. The Activation Function

The function that computes the weighted sum $\sum_{n=1}^{N} W_n^T X_n$ and decide by applying some *threshold function* if the perceptron should output 0 or 1.
There are several activation functions like the *sigmoid* function, but given that we're interested in building a quantum version of the perceptron model, we will focues on the *step function* because is simpler to implement in a quantum fashion, as we will see in section 6.

III. Perceptron Training Algortithms

Training a perceptron will be a *Supervised Learning* task such that we will have a set of inputs with a binary label associated meaning the output of the perceptron for a given input.
Now in order for us to train the perceptron we need to adjust the weights in such a way that the perceptron can recognize an input and outputs the real value.
A way of doing so, is by a method so called *gradient descent*. This method consists in mini-

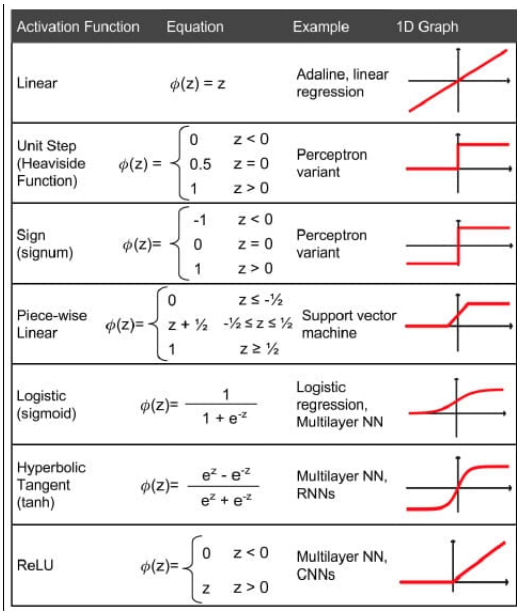


Figure 2: *Several activation functions*

mizing a *cost function* by calculating gradients of this function. It's called descent because we want the minimum of the function.
There are several *cost functions* but one particularly used in the ML field is the *Mean squared error loss function*, $J = \sum_{n=1}^{N} (\overline{y} - y)^2$ where calculates the difference between the output of the perceptron and the real output (label associated to a ginven input). This difference can be interpreted as the error of the perceptron prediction.
Now we understand wy doing the gradient of the loss function works, because we are walking towards the minimum error.

IV. The XOr Problem

Back in 1969, Minski and Papert continued the work done by Rosenblatt on perceptrons and trained these to Learn some logic gates like AND , NOT, OR and XOR. The perceptron predicted well for the first three logic gates, but not worked for the XOr logic gate. This is the so called XOr problem.
They soon discovered that all first three logic gates have one particular feature in common. They all have *linearly separable* data, and that's the reason why the perceptron doesn't work good for the XOr logic gate. That's not the end of the story of course, there is a solution for the XOr problem, and the solution is *multilayer perceptrons* constituting a *Deep Neural Network*.

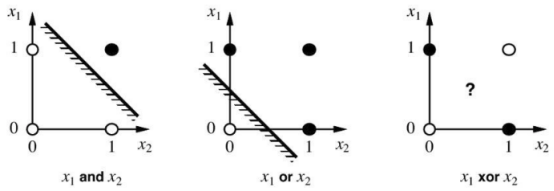


Figure 3: *Linear Separability*

V. Deep Neural Networks

Consists in a group of perceptrons (the fundamental building block) interconnected. It has an *input layer* where all the input training data will be fed, an *hidden layer*, and an *output layer*. An output of a perceptron will serve as an input of another layer perceptron.

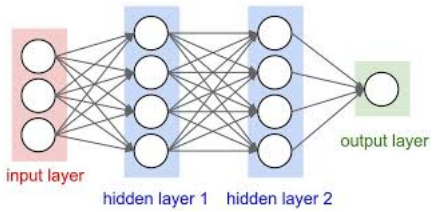
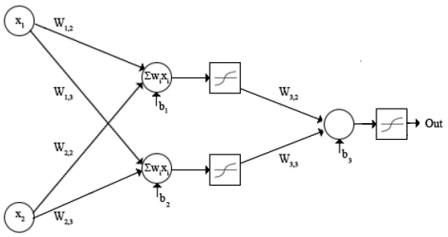


Figure 4: *Deep Neural Network*

This way we are able to construct a *multi linear regression* scheme and separate the non-linear data in XOr logic gate.
For training Deep Neural Networks, given a set of input training data (*Batch*), we need to know the predicted value in the output layer for each input pairs in the set. This process is called *feedforward*. Then, we need to calculate the error using the *Mean Squared Error Loss Function* defined previously. After calculating the error we need to communicate the error back in to the parameters in order for the next step in training (*epoch*) we use the updated

weights and climb towards the minimum, thus, the correct prediction. This process is called *Backpropagation*.

Figure 5: *Xor Neural Network*



VI. The Quantum Perceptron

The quantum perceptron circuit [2] is based on the idea of writing the normalised net input $h(\vec{w}, \vec{x}) = \varphi \in [0, 1)$ into the phase of a quantum state $|x_1, ..., x_n >$ and applying the phase estimation algorithm with a precision of τ. This procedure will return a quantum state $|J_1, ..., J_\tau >$ which is the binary fraction representation of θ or, equivalently, the binary integer representation of j in $\theta = \frac{j}{2^\tau}$, which is in turn a good approximation for φ.
Given that there isn't a clear implementation of a quantum version of the *sigmoid function* we implement the *step activation function*. For that we just need to measure the most significant qubit and that gives us the output of the Quantum Perceptron.

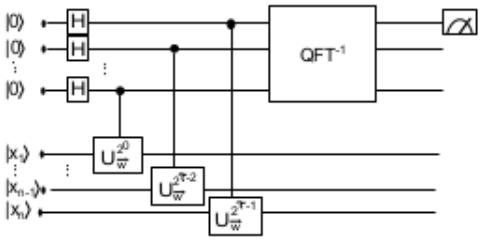


Figure 6: *Quantum Perceptron Model*

Initially we have the input state quantum registers and ancilla register with the same dimensions of the input register initialized at 0 -> $|0\rangle^{\otimes N}$. The rotations will be controlled by the auxiliary register. The rotations are the weights of the perceptron.
For Phase estimation it will be necessary to design an oracle that apply powers of the controlled rotations, let's call $U(w)$. So if the last qubit is in 1, apply $U(w)$ once, if the second last qubit is also in 1, apply $U(w)$ twice more, and if the kth last qubit is in 1 apply $2^{k1}U(w)$ times more.

VII. A Training Algorithm for Quantum Perceptrons

The model proposed in [1] tries to mimic the the behaviour of the classical training algortithm *gradient descent*, by using the *Phase kickback* principle.

i. Quantum Phase Kickback

Consider the case of a controlled-displacement on the position basis of two registers, $e^{-i\phi_c \prod_t} : |\phi_c\rangle |\phi_t\rangle \rightarrow |\phi_c\rangle |\phi_t + \phi_c\rangle$. It would seem that the operation had no effect on the control register. However, we can see this is not so by examining the action on the momentum basis, $e^{-i\phi_c \prod_t} : |\prod_c\rangle |\prod_t\rangle \rightarrow |\prod_c - \prod_t\rangle |\prod_t\rangle$ Hence, the control register is only left unchanged if in a position eigenstate, and the target register is only left unchanged if in a momentum eigenstate. We see that the operation not only affects the target qubit, but also the control qubit. More precisely, the operation changes the computational value of the target qubit and the phase of the control qubit, hence the name phase kickback.

ii. Quantum FeedForward and Backpropagation

The Quantum Feedforward and Baqprop (QFB) algorithm, evaluates the gradient of the loss function for a single training example and stores it in the momenta of the parameter registers via an effective phase kick.
Given an unitary that can perform the *mean squared error loss function*, L, the change in momentum for a single data point is proportional to the kicking rate η times the negative gradient of the effective loss function L. After batching multiple data points, the momentum kicks accumulate to change the momentum according to the negative gradient of the total effective cost function:

$$\prod = \prod -\eta \nabla L(\theta)$$

We can write the unitary corresponding to the quantum dynamical descent (QDD) as:

$$U_{QDD} = \prod_k e^{-i\gamma_k \Pi^2} e^{-i\eta_k L(\theta)}$$

where we call each parameter η_k the phase kicking rate and γ_k the kinetic rate for epoch k. We will argue that a heuristic one may wish to use for choosing the phase kicking rate and kinetic rate is to begin with $\gamma_k >> \eta k$ for small k (early time), and for large k (late time) shift towards $\gamma_k << \eta_k$. Beginning with a large kinetic rate and transitioning to a (relatively) larger phase kicking rate aids in converging to a local minimum. In figure 7 we see two iterations of Quantum Dynamical Descent Algorithm.

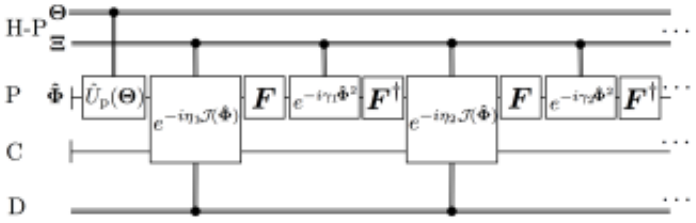


Figure 7: *Two iterations of Quantum Dynamical Descent*

VIII. Results

The implementations follow in the jupyter notebook alongside with the article.
The results show the testing of the network for the 4 possible inputs of a Xor Logic gate. The figures show the counts obtained by the qasm simulator with 4 bitstrings with 4 bits each representing the test for the 4 possible inputs.
From left to right each bitstring of 4 bits means: (prediction,real output,input2,input1). The results show the testing for different epochs.

For larger epochs , the the results obtained is equal to epochs=4.
So we see that the results are not by far the expected, given that the network is only able to predict well 50% of the test inputs.

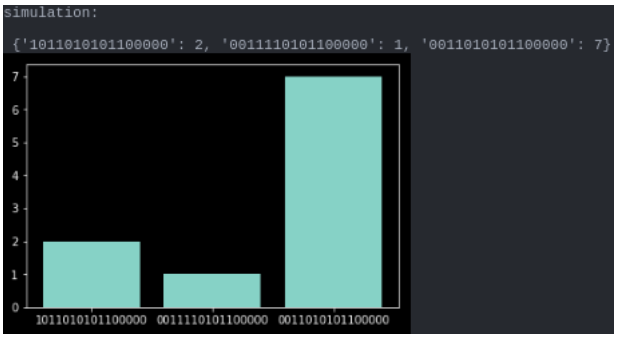


Figure 8: *Testing For epochs=1*

Figure 9: *Testing For epochs=2*

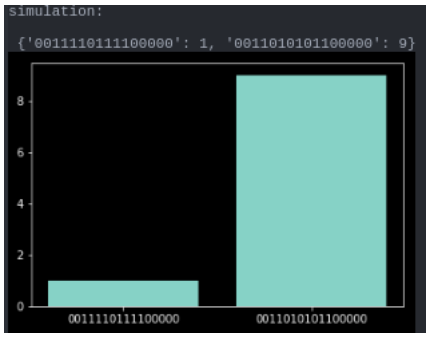


Figure 10: *Testing For epochs=3*

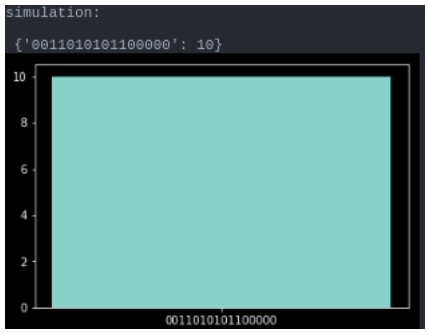


Figure 11: *Testing For epochs=4*

There are several reaons for these results to happen, such as:

- Could be that there were flaws in the implementation of the algorithm.
- The learning rate and kinetic rate were chosen a bit at random wich could help to affect the training of the network.
- Here we just use register with one qubit for the weights of the network, for limitations in the simulator. These could give small accuracy to the weights.

Although the results were not what was expected, it was a rewarding experience, given that I learned a lot about machine learning in general and started to have a taste of quantum machine learning. It was a pity I did not have more time to improvise the neural network and improve the results. In the future it would be interesting to continue the work done so far and to verify what is best possible to make the quantum neural network using this training algorithm, Quantum Gradient Desncent.

References

[1] Verdon, G. Pye, J. Broughton, M.. A Universal Training Algorithm for Quantum Deep Learning (2018).

[2] Schuld, M. Petruccione, F. Simulating a Quantum Perceptron on a quantum computer (2014).