The HiggsML and a simple Quantum-enhanced Machine Learning algorithm

Special curriculum

Ruben Guevara

September 21, 2022

Abstract

In this project we did the Higgs Boson Machine Learning challenge made by the ATLAS Collaboration in 2014. In the first part we did it by Using Neural Networks and got an accuracy of 84.1% when separating signal from background. In the second part we explore Quantum-enhanced machine learning by using quantum circuits in a variational circuit ("Quantum Neural Network") on simulated near-term quantum computers. To test our variational circuit, we did the same Higgs Boson Machine Learning challenge after reducing the dataset samples and features so it could run on 5 qubits. This project utilized Nairobi (a 7 qubit IBMQ backend) and Manila (a 5 qubit IBMQ backend). By doing this we got an accuracy of 67% on Nairobi and 66% on Manila. We came to the conclusion that it is not possible to do this challenge on near-term quantum computers, as the accuracy is low considering how much the data was manipulated. For the future, it will be interesting to see how quantum circuits with more qubits fare to do this challenge as well as others methods of doing Quantum Machine Learning.

Introduction

Machine Learning (ML) is on the rise in High Energy Physics, as it works wonderfully to separate signal from background. In this project we are going to do the Higgs Boson Machine Leaning challenge, or HiggsML for short, made by the ATLAS collaboration in 2014 using Neural Networks. This project is divided into two parts:

- *Classical ML*: Here we dive into the theory behind Neural Networks and introduce all the tools needed to tackle the HiggsML.
- *Quantum-Enhanced ML*: Here we dive into the theory behind quantum computers, and from this we will build up knowledge to do quantum machine learning. The ultimate goal is to then do the HiggsML using Quantum Neural Networks.

A preface is given in its respective part with more information. All the codes and plots can be found in my GitHub repository in the following link: https://github.com/rubenguevara/QuantumMachineLearning

Contents

Part I
The Higgs Boson Machine Learning Challenge

The Higgs Boson machine learning challenge (HiggsML for short) was a ML challenge set up to promote the collaboration between high energy physicists and data scientist in 2014. The challenge was organized by a small group of ATLAS physicists and data scientist and hosted by Kaggle. [1] For the purpose of this project I shall only explain the data and what the goal of this challenge was. The data consists of simulated events made by the official ATLAS simulator. The simulator simulates how the proton-proton collisions interact based on our knowledge of the Standard model as well as a simulation of the ATLAS detector such that the data we read is as close to real data from the ATLAS detector as possible. The data which is of interest in this project is the training set of the HiggsML, where the key feature for this project is the label. This key feature is whether the event, or sample, is either a signal or background event. To explain what the difference is we need to briefly discuss some basic particle physics.

The goal of this challenge is to use ML to learn the patters of various events, results of particle collisions, and try to determine which of these events come from a Higgs interaction. The "Higgs" is a gauge boson theorized from the 70's that was discovered in 2012 at the LHC, this is what many refer as the final puzzle of the Standard Model. There are several ways in which the Higgs boson can interact with particles, but the cross section (probability) for this to occur is smaller than other processes known in the Standard Model. The events that have a Higgs interaction are those labeled as signal, while every other event is called background. The way ML can be used here is to train a network to see the patters on the features, which there are 30 of, and see how the features of the signal differ from those of the background. The list of features is listed in Appendix A. All of these features are important to identify whether an event has had a Higgs interaction due the phenomenology that this would imply. For example the invariant mass, which is the sum of all masses of the final particles, is arguably the most important feature in this process, since it has been theorized (and proven in 2012) that the mass of the Higgs is 125GeV. Therefore if we see that an event has an invariant mass around 125GeV we can think of it as being an event with a Higgs interaction.

However although it sounds simple to check whether an event is a signal or background process manually. The problem is that there are $\approx 10^9$ collisions that happen at the LHC pr second. This is roughly equivalent to 1 Petabyte of data stored pr second [2] (although not all this data is processed). Therefore if one were to successfully train a network to identify signal from background the computation times might decrease, leading to hopefully more discoveries!

The original HiggsML wanted to use a statistical significance formula, Approximate Median Significance (AMS), as a metric to see whether the discovery significance of the ML algorithm could be higher than one used with traditional (particle physics) data analysis methods. However for this project this will not be relevant. The aim of this project is to only use the training dataset to get as high accuracy as possible and find the best parameters for the ML algorithm to then try to do the same on a simulated quantum computer.

The theory of this part is based of Morten Hjorth-Jensen lectures [3], [4] and [5].

1 Neural Networks

Since the goal is to use ML to get a high accuracy to distinguish signal from background we have to start from the ML basics. This project will take from granted that the reader is comfortable with linear algebra and jump straight into ML. The essence of machine learning is to use cost functions to tweak some parameters until it gives satisfactory predictions for the task given. In this project we will do Supervised Learning (meaning we know what the output should be) and only look at Neural Networks (NN) as our ML algorithm, since the ultimate goal is to "upgrade" this to run it on a quantum computer. The parameters mentioned above are called *weights* and *biases* for NNs. Considering a general parameter $\boldsymbol{\beta} = \{\beta_1, \beta_2, \cdots, \beta_n\}$ for a n-dimensional problem, the goal is to choose these parameters $\boldsymbol{\beta}$ such that we minimize a cost function $C(\boldsymbol{\beta})$ with respect to a set of data points given by a matrix $\mathbf{X}$, which in the case of the HiggsML are the features. And target values $\mathbf{t}$, which on the HiggsML are the labels. Before we get into that we can start by looking at Stochastic Gradient Descent (SDG).

1.1 Stochastic Gradient Descent

Before explaining the SDG we have to look at the Regular GD. Given a cost function $C(\boldsymbol{\beta})$ we can get closer to the minimum by calculating the gradient $\nabla_\beta C(\boldsymbol{\beta})$ wrt. the unknown parameters from the NN $\boldsymbol{\beta}$. If we were to calculate the gradient at a specific point $\boldsymbol{\beta}_i$ in the parameter space, the negative gradient would correspond to the direction where a small change $d\boldsymbol{\beta}$ in the parameter space would result in the biggest decrease in the cost function. In the same way we in physics would determine where the local (or global) minima at a complex multidimensional potential numerically. In GD we can chose a step size η to choose how much we want to iterate in the parameter space, this is called the *learning rate.* The mathematical function for an iteration to choose the parameter $\boldsymbol{\beta}$ such that it decreases the cost function is given as

$$\boldsymbol{\beta}_{i+1} = \boldsymbol{\beta}_i - \eta \nabla_\beta C(\boldsymbol{\beta}_i) \tag{1}$$

To converge towards a minimum we should choose a learning rate η small enough to not "step over" the minimum point of the cost-function-space. One thing to note here is that we could get trapped on a local minima rather than the global minima which is the ultimate goal. So choosing the learning rate as a hyperparameter to be changed in a grid search is a good way to find the best one.

In GD one computes the cost function and its gradient for all data points together. This quickly becomes computationally heavy when dealing with large datasets. Thus a common approach is to compute the gradient over batches of the data. For example in the HiggsML, instead of making a $30 \times 250,000$ matrix we could rather split it into smaller batches of maybe $30 \times 1,000$ to then perform a parameter update, making the computation faster. This is where SGD comes in, for each step, or epoch the data is divided randomly into N batches of size n. Then for each batch we use Eq. (1) to update the parameters, thus updating $\boldsymbol{\beta}_{i+1}$ N-times for each epoch. The idea of SGD comes from the observation that the cost function can almost always be written as a sum over n data points. As mentioned above the main advantage of SGD is the computation time, but it also reduce the risk of getting stuck in a local minima since it introduces a randomness of which part of the parameter space we move through.

1.2 Artificial neurons

As stated by Hjorth-Jensen in [3]:

> *The idea of NN is to to mimic the neural networks in the human brain, which is composed of billions of neurons that communicate with each other by sending electrical signals. Each neuron accumulates its incoming signals, which must exceed an activation threshold to yield and output. If the threshold is not overcome, the neuron remains inactive, i.e. has zero output*

To describe the behaviour of a neuron mathematically we can use the following model

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u) \tag{2}$$

Where y, the output of the neuron, is the value of its *activation function*, which has the weighted (w_i) sum of signals $x_i, \cdots, x_n$ received by n neurons.

Since the goal of NNs is to mimic the biological nervous system by letting each neuron interact with each other by sending signals, which for us is of the form of a mathematical function between each layer. Most NNs consist of an input layer, an output layer and maybe layers in-between, called hidden layers. All the layers can contain an arbitrary number of neurons, and each connection between two neurons is associated with a weight variable w_i. The goal of using NNs is to teach the network the patterns of the data to then predict something. In the context of the HiggML, by giving a NN our data as its input layer, we can then train the network to distinguish signal from background.

Explained in greater detail if we were to look at a single event of the data, we start with an input with all 30 features of the event. Using Eq. (2) on every neuron on the next layer we can teach the network if there is are any connections between the features, we can repeat this process for n layers. As an output we want a single neuron to see if it has predicted the event to be a signal or background, since this is binary output. After seeing the prediction we can use the labels to tell the network whether it predicted correctly or wrong since we are doing Supervised Learning. We can then use a *cost function* and a specific *metric* to evaluate numerically how network the predicted the output with a score. Seeing how the results fare we can then back-propagate to shift the weights and biases and repeat the process until we are satisfied with our result. Each of these iterations is called an epoch.

To generalize our artificial neuron to a whole network we can look at a Multilayer Perceptron (MLP). An MLP is a network consisting of at least three layers of neurons, the input, one or more hidden layers, and an output. The number of neurons can vary for each layer. The above explanation is a very dense and simplified one. In reality it is complicated to find out which cost function, activation function, metric, etc. is best suited the problem. But before we get into the details we can explore the mathematical model that illustrates what was tried to be explained above.

1.3 Activation functions

As seen above, an important aspect of NNs are activation functions and cost functions. As shall become apparent soon, when evaluating an activation function we get the neuron output, but what are these activation functions? Mathematically speaking, activation functions are: Non-constant, Bounded, Monotonically-increasing and continuous functions. For this project we utilize both a sigmoid activation function

$$f(x) = \frac{1}{1+e^{-x}} \tag{3}$$

which is the most basic activation function. We also utilise a Rectified Linear Unit (ReLU)

$$f(x) = x^{+} = \max(0, x) = \begin{cases} x & \text{if } x > 0, \\ 0. & \text{otherwise.} \end{cases} \tag{4}$$

which has better gradient propagation, meaning that there are fewer vanishing gradient problems compared to the sigmoidial function.

1.4 Cost functions

Another aspect are cost functions. Cost functions are what we will utilize to evaluate how well the output of the network fares against the target, i.e. if our network "guesses" right whether a event is signal or background, thus making this a very important part of our network! Before getting into this we first have to look at logistic regression. For the HiggsML we will study a binary case where the output is either $t_i = 0 \vee 1$, meaning background or signal. We can introduce a polynomial model of order n as

$$\hat{y}_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_n x_i^n$$

where we then can define the probabilities of getting $t_i = 0 \vee 1$ given our input x_i and $\boldsymbol{\beta}$ with the help of a logistic function. Using the same sigmoid function as in Eq. (3) as a logistic function, only calling it $p(t)$. We get the probability as

$$p(t_i = 1|x_i, \boldsymbol{\beta}) = \frac{1}{1+e^{-\hat{y}_i}}$$

and

$$p(t_i = 0|x_i, \boldsymbol{\beta}) = 1 - p(y_i = 1|x_i, \boldsymbol{\beta})$$

We want to then define the total likelihood for all possible outcomes from a dataset $\mathcal{D} = \{(t_i, x_i)\}$, with the binary labels $t_i \in \{0, 1\}$, to do this we use the Maximum Likelihood Estimation (MLE) principle. This gives us

$$P(\mathcal{D}|\boldsymbol{\beta}) = \prod_{i=1}^{n} \left[p(t_i = 1|x_i, \boldsymbol{\beta})\right]^{t_i} \left[1 - p(t_i = 1|x_i, \boldsymbol{\beta})\right]^{1-t_i}$$

from which we obtain the log-likelihood

$$C(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(t_i \log p(t_i = 1|x_i, \boldsymbol{\beta}) + (1 - t_i) \log[1 - p(t_i = 1|x_i, \boldsymbol{\beta})]\right)$$

By taking the parameter $\boldsymbol{\beta}$ to second order and reordering the logarithm we get

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} (y_i(\beta_0 + \beta_1 x_i) - \log(1 + \exp(\beta_0 + \beta_1 x_i))) \tag{5}$$

This equation is known as the *cross entropy* which we will use in this project. The two beta parameters used are the weight and biases as will come apparent later. The goal is to change these parameters such that it minimizes the cost function as we will see later.

Something else we will include in this project is to add an extra term to the cost function, proportional to the size of the weights. We do this to constrain the size of the weights, so they don't grow out of control, this is to reduce *overfitting*. In this project we will use the so called *L2-norm* where the cost function becomes

$$C(\boldsymbol{\beta}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i(\boldsymbol{\beta}) \rightarrow \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i(\boldsymbol{\beta}) + \lambda \sum_{ij} w_{ij}^2 \tag{6}$$

Meaning we add a term where we sum up all the weights squared. The factor λ is called the regularization parameter. The L2-norm combats overfitting by forcing the weights to be small, but not making them exactly zero. This is so that less significant features still have some influence over the final prediction, although small.

1.5 Feed Forward network

To describe how the network "guesses" outputs in a mathematical model we can compute we can start by looking at Eq. (2) where we got an output y from an activation function f that receives x_i as input. We can expand the function as as following

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(z) \tag{7}$$

where w_i is still the weight and we introduced a bias b_i which is normally needed in case of zero activation weights or inputs. The difference comes now in the interpretation where in the activation $z = (\sum_{i=1}^{n} w_i x_i + b_i)$ the inputs x_i are the outputs of the neurons in the preceding layer. Furthermore an MLP is fully-connected, meaning that each neuron received a weighted sum of the output of **all** neurons in the previous layer. To expand Eq. (7) we can first look at the output of every neuron i in a weighted sum z_i^1 for each input x_j on a layer

$$z_i^1 = \sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1 \tag{8}$$

Such that if we evaluate the weighted sum in an activation function f_i for each neuron i, then the output of all neurons in layer 1 is y_i^1

$$y_i^1 = f(z_i^1) = f\left(\sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1\right)$$

Where M stands for all possible inputs in a given neuron i in the first layer, we have also assumed that we utilize the same activation function in the layer. To generalize this for l-layers, which may have different activation functions, we write it as

$$y_i^l = f^l(u_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right)$$

Where N_l is the number of neurons in layer l. Thus when the output of all the nodes in the first hidden layer is computed, the values of the subsequent layer can be calculated and so forth until the output is obtained. With this we can show that we only need the the inputs x_n to calculate the output with l hidden layers

$$y^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^{l+1} f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^l\left(\cdots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\cdots\right) + b_j^l\right) + b_i^{l+1}\right] \tag{9}$$

This shows that an MLP is nothing more than an analytic function, specifically a mapping of real-valued vectors $\hat{x} \in \mathbb{R}^n \to \hat{y} \in \mathbb{R}^m$. We can also see that the above equation is essentially a nested sum of scaled activation functions of the form

$$f(x) = c_1 f(c_2 x + c_3) + c_4$$

where the parameters c_i are the weight and biases. By adjusting these parameters we shift the activation function to better match the label we are training the data on, this is the flexibility of a NN. Something else we can note is that Eq. (9) can easily be changed into matrix notation, since this is trivial for high energy physicists I will spare myself the writing of matrix form on this project. However this realization can help make computing the values a much easier task by for example utilizing TensorFlow or other mathematical packages in Python. An illustration taken from [3] shows the main idea of how a Feed forward network is set up, this is shown in Figure 1.

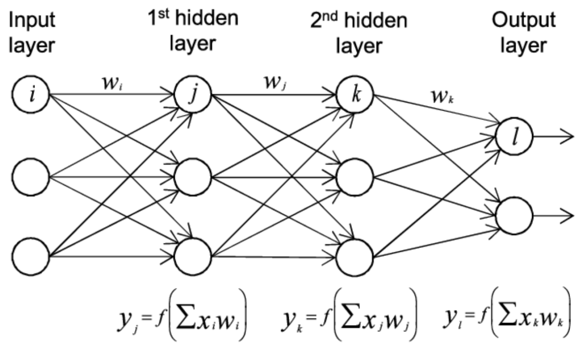

Input layer
1st hidden layer
2nd hidden layer
Output layer
w_i
w_j
w_k
i
j
k
l
$y_j = f\left(\sum x_i w_i\right)$
$y_k = f\left(\sum x_j w_j\right)$
$y_l = f\left(\sum x_k w_k\right)$

Figure 1: Basic illustration of a network with two hidden layers. Image taken from [3]

1.6 Back Propagation algorithm

So far we have only explained Feed Forward networks, which helps us to compute the output of the NN in term of basic vector multiplications. It has also been mentioned that we can adjust the weight and biases, but never explained how. Now is the time to dive into that subject, as we will explain the back propagation algorithm. What we want to know is how do changes in the biases and the weights in the network change the cost function, and how we could use the final output to modify the weights? Before we derive these equations we an start by a plain regression problem and using the Mean Squared Error (MSE) as a cost function for pedagogical reasons

$$C(\hat{W}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - t_i)^2 \tag{10}$$

where $\hat{W}$ is the matrix containing all the weights and more importantly t_i are our targets, which is in the HiggsML are the labels of events telling us whether we have a signal or background event. To generalise this we first have to go back to Eq. (8) generalize it for a layer l

$$z_i^l = \sum_{j=1}^{M} w_{ij}^l y_j^{l-1} + b_i^l \Leftrightarrow \hat{z}^l = \left(\hat{W}^l\right)^T \hat{y}^{l-1} + \hat{b}^l$$

where the right side is written on matrix notation. From the definition of z_j^l with an activation function, i.e. Eq. (7), we have

$$\frac{\partial z_j^l}{\partial w_{ij}^l} = y_i^{l-1} \tag{11}$$

and

$$\frac{\partial z_j^l}{\partial y_i^{l-1}} = w_{ij}^l$$

which again, with the definition of the activation function gives us

$$\frac{\partial y_j^l}{\partial z_j^l} = y_j^l(1-a_j^l) = f(z_j^l)(1-f(z_j^l)) \tag{12}$$

We also need to take the derivative of Eq. (10) with respect to the weights, doing so for a respective layer $l = L$ we have

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^L} = \left(y_j^L - t_j\right)\frac{\partial y_j^L}{\partial w_{jk}^L}$$

where the last partial derivative is easily computed using the chain rule with Eq. (11) and Eq. (12)

$$\frac{\partial y_j^L}{\partial w_{jk}^L} = \frac{\partial y_j^L}{\partial z_j^L}\frac{\partial z_j^l}{\partial w_{jk}^L} = y_j^L(1-y_j^L)y_k^{L-1}$$

Such that we have

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^L} = \left(y_j^L - t_j\right)y_j^L(1-y_j^L)y_k^{L-1} := \delta_j^L y_k^{L-1} \tag{13}$$

where we have defined the error

$$\delta_j^L := \left(y_j^L - t_j\right)y_j^L(1-y_j^L) = f'(z_j^L)\frac{\partial C}{\partial y_j^L} \tag{14}$$

or in matrix form

$$\delta^L = f'(\hat{z}^L)\circ\frac{\partial C}{\partial \hat{y}^L}$$

where on the right hand side we wrote this as a Hadamard product. This error δ^L is an important expression, since as we can see on the index form of this expression on Eq. (14), we can measure how fast the cost function is changing as a function of the j-th output activation. This means that if the cost function doesn't depend on a particular neuron j, then δ_j^L would be small.

We also notice that everything in Eq. (14) is easily computed. Thus we can also see how the weight changes the cost function using Eq. (13) quite easily. One thing else we can compute with Eq. (14) is

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial y_j^L}\frac{\partial y_j^L}{\partial z_j^L}$$

which can be interpreted in terms of the biases b_j^L

$$\delta_j^L = \frac{\partial C}{\partial b_j^L}\frac{\partial b_j^L}{\partial z_j^L} = \frac{\partial C}{\partial b_j^L} \tag{15}$$

where we see that the error δ_j^L is exactly equal to the rate of change of the cost function as a function of the bias.

Somerhing interesting as briefly mentioned above is that when using Eq. (13 - 15) we see that if a neuron output y_j^L is small, then the gradient term, Eq. (13), will also be small. We say then that the weight learns slowly, meaning that the contribution of said neuron is less important "to fix" than those that have a higher weight. Of course this example is a very simple one to wrap our heads around, but the magic comes when the algorithm is evaluating a random neuron in layer 20 on a deep learning algorithm, after so many layers it all becomes a **black box** for us to wrap our heads around!

It is also worth noting that when the activation function is flat at some specific values (depend on the chosen function) the derivative will tend towards zero making the gradient small meaning the network is learning slow as well. To finish up our back propagation algorithm we still need one more equation. We are now going to propagate backwards in order to determine the weight and biases. We start by representing the error in the layer before the final one $L-1$ in term of the errors of the output layer. If we try to express Eq. (14) in terms of the output layer $l+1$. Using the chain rule and summing over all k entries we get

$$\delta_j^l = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_j^l}$$

recalling Eq. (8) (replacing 1 with $l+1$) we get

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l) \tag{16}$$

Which is the final equation we needed to start back propagating.

1.7 Summary of idea

To summarize the whole process of the NN

- First take the input data $\mathbf{x}$ and the activation $\mathbf{z}_1$ of the input later, and then compute the activation function $f(z)$ to get the next neuron outputs $\mathbf{y}^1$. Mathematically this is taking the first step of the feed forward algorithm, i.e. choosing $l = 0$ on Eq. (9)
- Secondly we commit all the way on Eq. (9) and compute all $\mathbf{z}_l$, activation function and $\mathbf{y}^l$.
- After that we compute the output error $\boldsymbol{\delta}^L$ by using Eq. (14) for all values j.
- Then we back propagate the error for each $l = L-1, l-2, \cdots, 2$ with Eq. (16).
- The last step is then to update the weights and biases using Eq. (1) for each l and updating using

  $$w_{jk}^l \leftarrow w_{jk}^l - \eta \delta_j^l y_k^{l-1}$$

  and

  $$b_j^l \leftarrow b_j^l - \eta \delta_j^l$$

This whole procedure is usually called an epoch, which we can repeat as many times as we want to better reduce the cost function in hopes of getting to the global minima.

2 Implementation

In this project I have utilized TensorFlow and Keras on the "classical" machine learning part, since the better optimized the code is, the better the chance to get the best parameters η, λ and number of neurons is. TensorFlow is an open source Python machine learning library. It was developed by the Google Brain team for international use. It was released under the Apache 2.0 open source license in November 9, 2015. As stated by them [6]:

> *The system is flexible and can be used to express a wide variety of algorithms, including training and inference algorithms for deep neural network models, and it has been used for conducting research and for deploying machine learning systems into production across more than a dozen areas of computer science and other fields, including speech recognition, computer vision, robotics, information retrieval, natural language processing, geographic information extraction, and computational drug discovery.*

Keras is a high level NN that supports TensorFlow as a backend. As mentioned all the codes and plots can be found in my GitHub repository in the following link:
https://github.com/rubenguevara/QuantumMachineLearning

2.1 Missing data problem

The most important part of any ML analysis is the collection and processing of data. The collection of data is not a hard task in this project since the HiggsML is already formatted in a very understandable way, as well as having well written documentation in of the features. However there is a big problem in this dataset that is not common for other fields, and that is that there are missing, NaN, values on the dataset. (The HiggsML team has however set these missing values as -999 instead of NaN) The reasoning behind these missing values is physically very easy to understand, for example there might be events where there are no τ leptons in the final state (See Appendix A). However these values are very problematic when we are using NNs since the inputs might give the neurons in the layers wrong weights due the high value -999, or give us an error due the NaN. This wouldn't be a problem if I were to choose Decision Trees to do the HiggsML, as the "winner" who created XGBoost did. However since I chose NN this needs to be addressed.

There are many ways to tackle this problem, one could do the "standard ML" approach, use Bayesian statistics to predict what the values should be, set the missing values as 0 or remove all events that have missing values. For the purposes of high energy physics removing events is not possible, since we want to have as high statistics as possible when coming to conclusions. Setting all the missing values to zero is a "naive" approach, physically it makes sense to set the missing value to zero for certain features, like the transverse momentum of a τ lepton (since this means it isn't there). However it is also physically unreasonable to set it to zero, if we had a missing value in for example the angle, phi, since this is not the same as saying it is not there, and the angle 0 has a physical meaning. Setting the values to zero is also highly looked down upon from data scientists, as this would also affect the weights, specially when we *normalize* the data. Due time constraint we will in this project use the standard ML approach to "walk around" this problem, since the Bayesian approach and other more advanced approaches are beyond the scope of this project. The standard approach in ML is to set the missing values of each feature as the average of all the values in said feature. However this is disliked by high energy physicists as it might violate conserved principles on some events, i.e. energy. But as mentioned above, to find out what the best solution to this problem is could be a whole project by itself. So average of each feature it is!

2.2 Pre-processing of data

Now that we have discussed arguably the biggest problem of this challenge we are ready to pre-process the data for the NN to take as inputs. In this project I used Pandas and NumPy to prepare the data. With Pandas I used the DataFrame class to read the file from the HiggsML. DataFrame also has inbuilt functions where we can replace specific values with whatever we want, making the averaging of features for missing values incredibly easy to address. Since the HiggsML data has the labels for signal and background as s and b I changed these to 1 and 0 respectively. I also removed the *EventID* and *Weight* features since these are not interesting for our purposes. Lastly I made a specific variable with only the labels, as this will be our target values.

After that I used sci-kit learn's function *train_test_split* to split the features and targets into training features and labels, X and Y, and testing features and labels x and y. The training data will be used to train the NN as we discussed in the previous theory-section, while the testing data is to see how well the network predicts on data it has not learned the patters of yet, this is to reduce a plausible bias that might come from only looking at the same samples. The *train_test_split* function randomly splits the labels and features into a train-test percentage we choose. For this project I choose to use 80% training data and 20% testing.

Then on the pre-processing step I made all the datasets X, x, Y, y into NumPy arrays such that it became more compatible with TensorFlows syntax, and also so it is the same format as what we will use on out Quantum ML algorithm. Lastly I used TensorFlow NN layer class *Normalization* to adapt the network to normalize the data based on the training features X on a scale of 0 to 1. This we do since we want the values of the numeric columns in the dataset to use a common scale, such that the model is less likely to prioritize some features over others. This is important since the values of the different features vary by a lot on the HiggsML! With all of this we are now ready to make the network.

2.3 Algorithm

As already stated we will make use of TensorFlow and Keras. To first set up the NN we use the Keras *Sequential* class, which is used to very easily add layers manually. Afterwards we create our input layer using Keras *Dense*, which is just a densely connected NN layer. The first layer has as many neurons as features on the dataset, so 30 in this case. In this project we only have two hidden layers, the reason behind this is so we can more easily replicate the same network on a simulated quantum computer using less computational power. On the two hidden layers the number of neuron was chosen to be a hyperparameter. Also on the first three layers we utilize ReLu, Eq. (4), as an activation function, as well as the L2-norm, Eq. (6), with λ as a hyperparameter. Lastly we have a layer with one neuron as the output with the sigmoid activation function, Eq. (3), since we only are interested in a binary output, signal or background. For the loss function we use cross entropy, Eq. (5). We also use SGD as an optimizer, with the learning rate η as a hyperparameter. A Figure showing the network layout is shown in Figure 2.

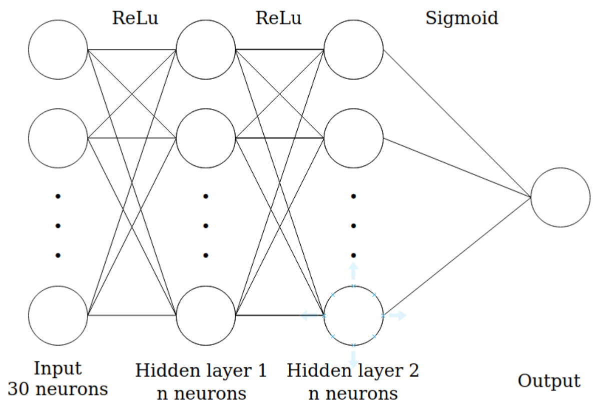


Figure 2: Illustration of the network used in this project.

Another thing, which is the main interest of this project, is to look at the accuracy of the network prediction when compared with the testing data. In this project we will use a metric from Keras called *BinaryAccuracy* which gives us a percentage answer of how many of the outputs match with the target values. The model as seen in Python can be seen in Algorithm 1.

Algorithm 1: Neural network definition using TensorFlow

```python
import tensorflow as tf
from tensorflow.keras import layers

normalize = layers.Normalization()
normalize.adapt(data)
def NN(inputsize, n_layers, n_neuron, eta, lamda):

    model=tf.keras.Sequential([normalize])

    for i in range(n_layers):
        if (i==0):
            model.add(layers.Dense(n_neuron, activation='relu', kernel_regularizer=
             tf.keras.regularizers.l2(lamda), input_dim=inputsize))
        else:
            model.add(layers.Dense(n_neuron, activation='relu', kernel_regularizer=
                    tf.keras.regularizers.l2(lamda)))

    model.add(layers.Dense(1,activation='sigmoid'))

    sgd=tf.optimizers.SGD(learning_rate=eta)

    model.compile(loss=tf.losses.BinaryCrossentropy(),
                optimizer=sgd,
                metrics = [tf.keras.metrics.BinaryAccuracy()])
    return model
```

2.4 Parameter tuning and network testing

As mentioned before, we have three hyperparameters we are interested in tuning to improve the performance of the network. In this project we only looked at 4 different values for each parameter and performed a grid search to find which one gave the best accuracy. The hyperparameters are

$$\eta = [0.001, 0.01, 0.1, 1],\ \ \lambda = [10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}] \text{ and } n = [1, 10, 100, 1000] \tag{17}$$

To more rigorously judge how well the model fares to separate the signal from background we can utilize two tools. Reconstruction error histograms and Receiver Operating Characteristic (ROC) curves. Reconstruction error histograms are plots where we can visualise how well the NN sorts signal from background using a histogram. ROC curves illustrate how well the model fares by plotting the True Positive Rate (TPR) against the False Positive Rate (FPR). The TPR we get by dividing the True Positive (TP), which in the HiggsML would be whenever the model correctly guessed whether an event was signal, by the Positive (P), which is the number of signal events in the data.

$$\text{TPR} = \frac{\text{TP}}{\text{P}}$$

FPR is when the False Positive (FP), wrongly guesses signals, divided by the Negatives (N), number of background events in the data.

$$\text{FPR} = \frac{\text{FP}}{\text{N}}$$

The goal is to have as many TPR as possible. To get a numerical value of how well the NN does we calculate the area under the ROC curve. Since TPR and FPR are both rates they go from 0 to 1, meaning that if we had a perfect NN we would only get TPR and thus an area of 1. If we were to get an area of 0.5 this would mean that the NN is randomly guessing whether an event is a signal or background, basically making a coin toss for every event. So we want out network to score as closely to 1 as possible. It is also of interest to see how much the network overfits by seeing the model accuracy and loss as a function of epochs, to then choose a number of epochs.

3 Results

The result of the grid search is seen in Figure 3. Here we see that the best parameters are

$$\lambda = 10^{-5},\ \eta = 1 \text{ and } n = 1000$$

for the training data giving an accuracy of 85.2% and

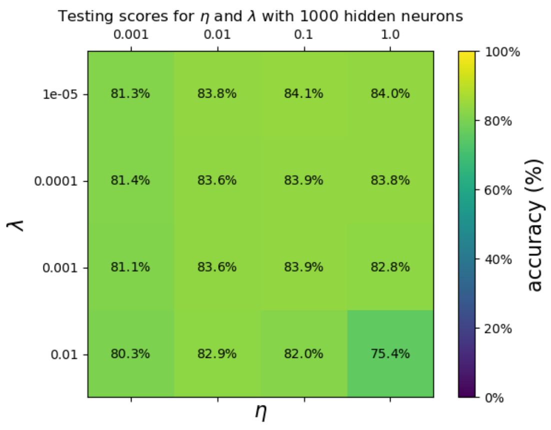
$$\lambda = 10^{-5},\ \eta = 0.1 \text{ and } n = 1000$$

for the testing data giving an accuracy of 84.1%. We can already see that there is an ovetraining on the training data even though we used the L2-norm. We will will use the test data result as a "real" result of our network. In Figure 4 we see that we get a ROC score of 0.91 with our model, and in Figure 5 we see that the NN does really well in sorting signal from background, with only minimal overlap. So far so good! In Figure 6 and 7 we see however that after 10 epochs the model severely overtrains and give us much better results with the training dataset than the test dataset. Thus we will choose 10 epochs of training for the next part.

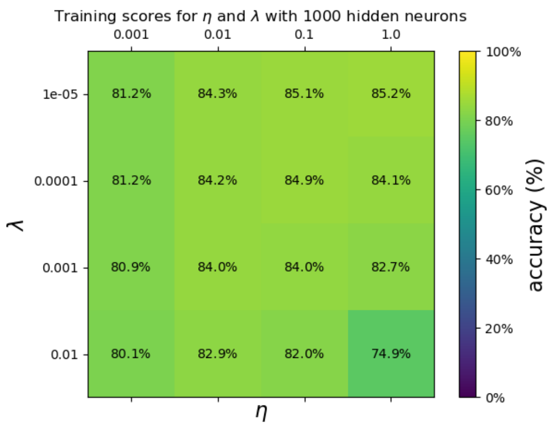
To summarize we have the parameters

$$\boxed{\lambda = 10^{-5},\ \eta = 0.1 \text{ and } n = 1000} \tag{18}$$

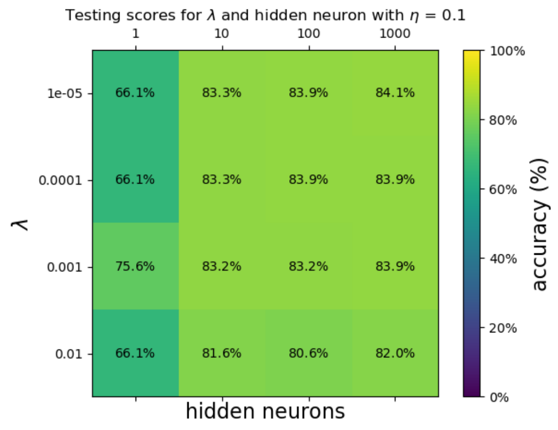
with an accuracy of 84.1% and a ROC score of 0.91 with 10 epochs. This are the parameters we will use for the next part of this project.



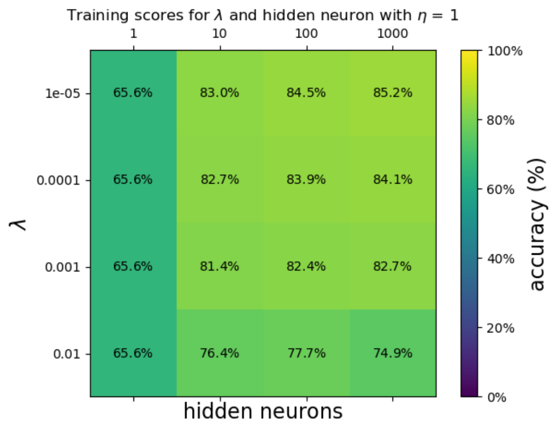
(a)



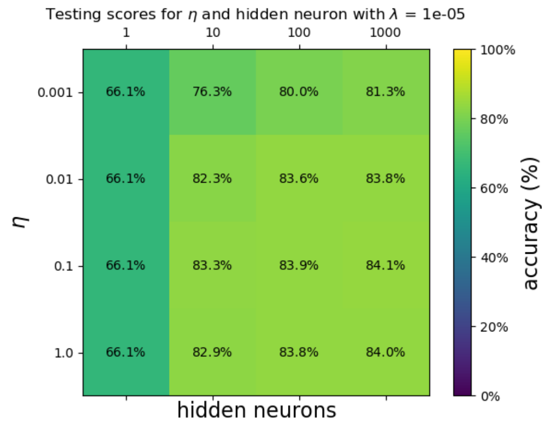
(b)



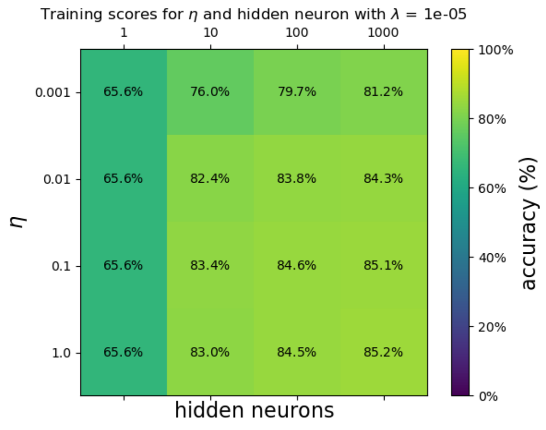
(c)



(d)



(e)



(f)

Figure 3: Result of the grid search on the train and test data.

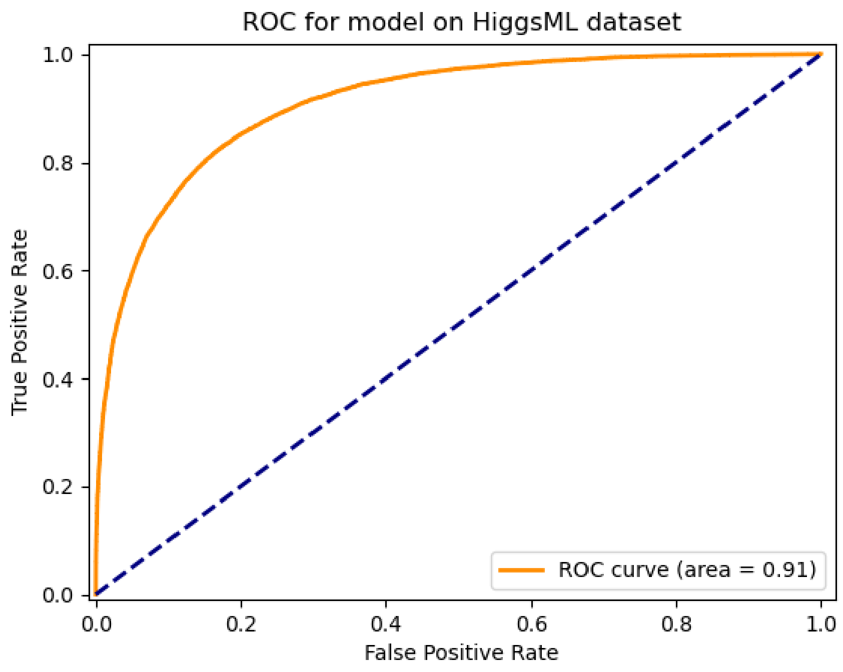


Figure 4: ROC curve of our NN. The dashed line is for random guessing.

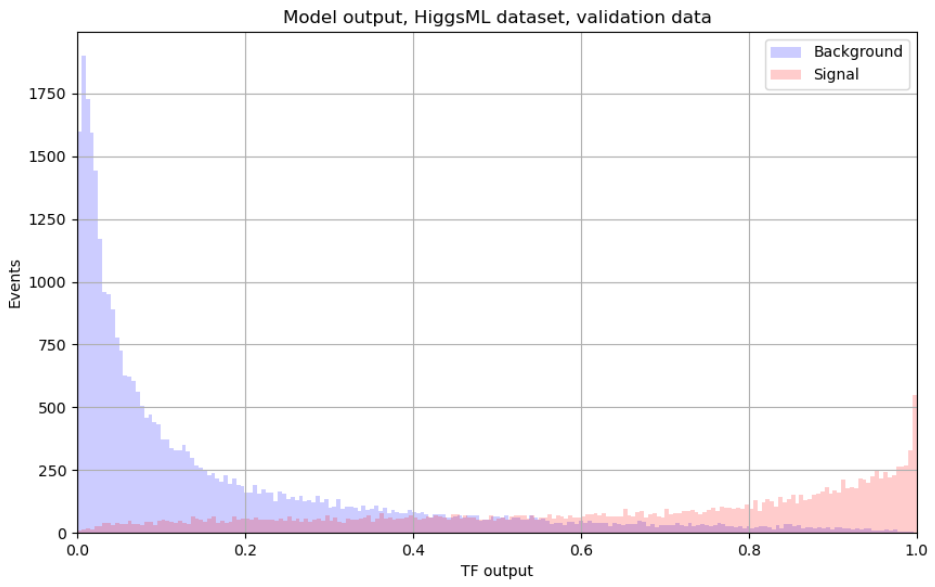


Figure 5: Reconstruction error histogram of our NN.

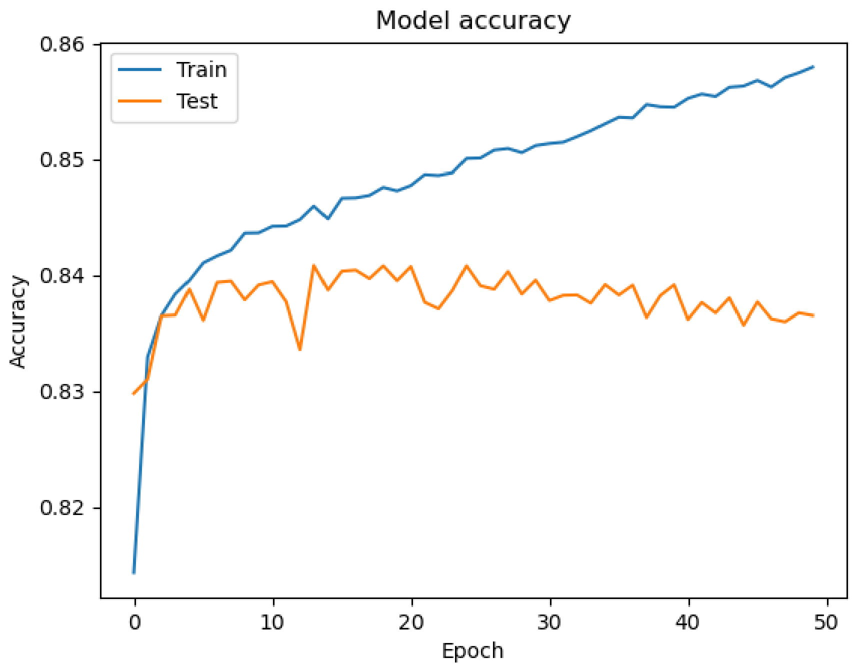


Figure 6: Accuracy of training and test data pr epoch.

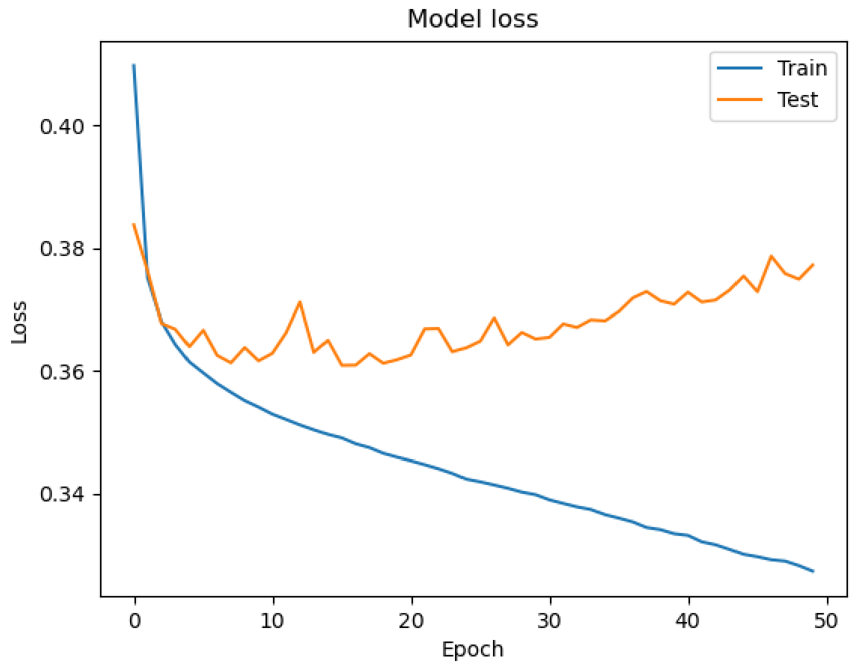


Figure 7: Loss of training and test data pr epoch.

Part II
Let's go Quantum

As we all know, quantum mechanics changed the way physics thought making them look at all aspects of their discipline from a new perspective, so naturally the question of what quantum mechanics can contribute to information processing arose. This was already a thought of physicists as earl as Einstein's "spooky action at a distance". However with the advancement of new technologies, and promises of quantum supremacy in the future, the idea of a *quantum computer* has become quite popular in media. Since there is a promise of algorithmic speedups and better computational power, it is also natural to try to implement this in fields that need heavy computational power. As mentioned in part I, LHC collects very large amounts of data every second. If we were able to analyze the data in a faster and more efficient way it could possibly lead to more discoveries in the field. We are at an age in physics where ML is on the rise and helping us analyse data in more efficient and faster way. Thus the question arose, what about Quantum ML (QML)? Is this the future new wave of computational tools, or is it just media fanaticism? That is the motivation behind the second part of this project, where the aim is to make an algorithm to run the HiggsML on a (simulated) quantum computer.

One thing to clarify is what we mean by QML in this project. There are four ways in which we can do QML. These are divided into whether the data was generated by a classical (C) or quantum (Q) system, and if the information is being processed by a quantum (Q) or classical (C) device. So *CC* is classical data being processed on a classical computer, i.e conventional approach to ML. *QC* is quantum data on classical machines, this is used to see how ML can help with quantum computing. Then we have *CQ* which is classical data using quantum computers, this is often referred to as quantum-enhanced ML and is what we will use in this project. The last one is *QQ*, which is closely related to *CQ*, but uses quantum data instead. Making the task of loading the data and plausible computation times faster than *CQ*. However as this project aims to make a tool to be used by high energy physicists, where the data is produced classically *today*, we shall only look at *CQ*. In Figure 8 we can see all the different types.

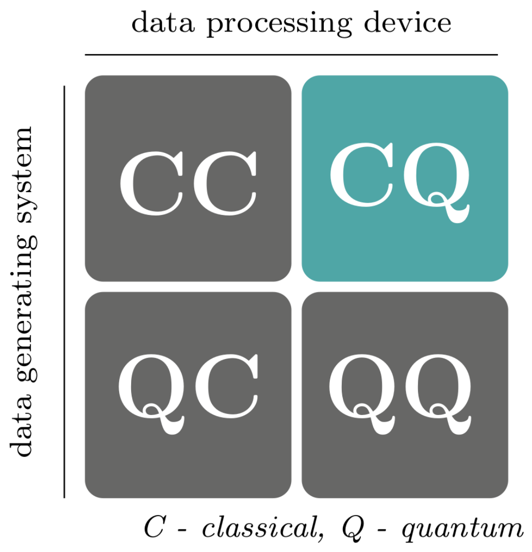


Figure 8: Four ways to combine quantum computing and machine learning. Highlighted is classically processed data being used on quantum machines, the emphasis of this project. From [7]

Before getting into any new algorithms we have to quickly review what quantum computing is. This project has used Schuld and Petruccione's textbook *Machine Learning With Quantum Computers* [7] as a basis for the theory as well as Wold's Masters thesis *Parameterized Quantum Circuits for Machine Learning* [8]. We will also take for granted that the reader is comfortable with quantum mechanics and only remind the reader of some important aspects where it is needed.

4 Quantum Computing

So what is a quantum computer? A short description would be as a physical implementation of n *qubits* with precise control on the evolution of the state. Following this definition of quantum computers a *quantum algorithm* would be a controlled manipulation of the quantum system with a subsequent measurement to retrieve the information from said system. Basically meaning that a quantum computer can be seen as a special kind of sampling device, but since this is a quantum state it heavily depends on experimental configurations. There is a theorem in quantum information that states that any quantum evolution can be approximated by a sequence of elementary manipulations, these are called *quantum gates* [9]. Based on this, quantum algorithms are widely formulated as *quantum circuits* of these quantum gates.

4.1 Qubits

The idea of a qubit came from "upgrading" classical bits, which are 0 or 1, to a quantum state.

$$0 \longrightarrow |0\rangle = \begin{pmatrix}1\\0\end{pmatrix}, \quad 1 \longrightarrow |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$$

But what are qubits? A qubit is often called the simplest possible quantum system, as it is a two-level system defined on $\mathbb{C}^2$. We can formulate this state as

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle \tag{19}$$

with α_0, $\alpha_1 \in \mathbb{C}$ and $|\alpha_0|^2 + |\alpha_1|^2 = 1$, where $|0\rangle$ and $|1\rangle$ are orthonormal states referred to as *computational basis states* which are defined by the hardware. The qubit is important since it is in a *superposition*, meaning its neither $|0\rangle$ nor $|1\rangle$, but both at the same time, meaning that when compared to classical bits we have a mixture of both. We can generalize this to include n untangled qubits using tensor products as

$$|\psi\rangle \equiv |q_1\rangle \otimes |q_2\rangle \otimes \cdots \otimes |q_n\rangle \tag{20}$$

where $|q_i\rangle$ are qubits. However if the qubits were entangled, the state $|\psi\rangle$ would no longer be separable, and every qubit wold either be $|0\rangle$ or $|1\rangle$, thus we would get

$$|\psi\rangle = \alpha_0 |0 \cdots 00\rangle + \alpha_1 |0 \cdots 01\rangle + \cdots + \alpha_{2^n-1} |1 \cdots 11\rangle$$

with $\alpha_i \in \mathbb{C}$, and $\sum_{i=0}^{2^n-1} |\alpha_i|^2 = 1$. Where we use the shorthand notation $|a\rangle \otimes |b\rangle := |ab\rangle$. To make the notation more elegant, we notice that the basis states could be translated from binary numbers to integers, meaning that we could write i.e. $|000\rangle \leftrightarrow |0\rangle, \cdots, |111\rangle \leftrightarrow |7\rangle, \cdots$ giving us the simple equation

$$|\psi\rangle = \sum_{i=0}^{2^n-1} \alpha_i |i\rangle \tag{21}$$

Thus we have $\{|0\rangle, \cdots, |i\rangle\}$ as a computational basis for n qubits. As pointed out in Wold's masters thesis [8]. We see that since there are 2^n unique strings, we realize that one needs 2^n amplitudes α_i to describe the state of n qubits. That means that the information stored in a quantum state with n qubits is exponential in n, where in the classical systems it is linear in n. Making quantum information "larger" than classical information in a sense. Which so far suggests quantum improvements!

4.2 Quantum Circuits

To make a quantum algorithm, or quantum circuit as mentioned above, we need to start by looking at quantum gates. Quantum gates, or rather quantum logic gates, are realize by unitary transformations. To quickly remind ourselves what this means we can look at a simple transformation

$$|\phi\rangle = U |\psi\rangle$$

where U is a *unitary* operator that acts on the same vector space, $|\phi\rangle$ and $|\psi\rangle$ live in. By unitary we mean that the operator is linear and has the property that the hermitian conjugate is the inverse, $U^\dagger = U^{-1}$. This is important since we can use this to i.e. show

$$\langle\phi|\phi\rangle = \langle\psi| U^\dagger U |\psi\rangle = \langle\psi|\psi\rangle = 1$$

where if $|\psi\rangle$ is normalized so is $|\phi\rangle$ by construction.

4.2.1 Single-Qubit Operator

Back to the topic of quantum gates, if we were to measure a qubit, Eq. (19), the state would either be in the state $|0\rangle$ with probability $|\alpha_0|^2$ or in the state $|1\rangle$ with probability $|\alpha_1|^2$. Starting with single-qubit gates, these are formally described by 2×2 unitary transformations. To start we can consider the X gate, which is the quantum equivalent of the classical NOT gate, as it acts as

$$|0\rangle \mapsto |1\rangle$$

and vice versa. We can pretty easily recognize this matrix as one of the Pauli matrices, and these are by definition unitary. Thus we know we can have at least X, Y and Z gates which have the Pauli matrices as their unitary operator

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \ \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \ \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{22}$$

making it easy to see that $|0\rangle \mapsto |1\rangle \Leftrightarrow \sigma_x |0\rangle = |1\rangle$. Another interesting single qubit quantum gate is the Hadamard gate

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{23}$$

The Hadamard gate can be used to produce superpositions

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

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

Another useful set of gates are the *Pauli rotations*. Which are formulated as exponential Pauli gates

$$R_j(\theta) = e^{-i\frac{\theta}{2}\sigma_j} \tag{24}$$

where $j \in \{x, y, z\}$. Since we can write any quantum state in terms of the azimuthal (θ) and polar (ϕ) angle and global phase $(e^{i\gamma})$

$$|\psi\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle = e^{i\gamma}(\cos\frac{\theta}{2} |0\rangle + e^{i\phi} \sin\frac{\theta}{2} |1\rangle)$$

what the Pauli rotations do is to rotate the state around the j-axis for an amount of θ radians.

4.2.2 Multi-Qubit Operators

Since we will want to operate on multiple qubits at the same time we introduce the controlled U gate. Below we demonstrate how it looks with two qubits, but this can easily be generalized to n qubits

$$|00\rangle \mapsto |00\rangle, |01\rangle \mapsto |01\rangle, |10\rangle \mapsto |1\rangle \otimes U |0\rangle, |11\rangle \mapsto |1\rangle \otimes U |1\rangle \tag{25}$$

where U is an arbitrary single-qubit unitary gate. For example, if we let $U = \sigma_x$ we get the CNOT gate, the NOT (X) operation is performed when the first qubit is in state $|1\rangle$; otherwise the first qubit is unchanged

$$|00\rangle \mapsto |00\rangle, |01\rangle \mapsto |01\rangle, |10\rangle \mapsto |11\rangle, |11\rangle \mapsto |10\rangle \tag{26}$$

The CNOT can be represented with $\begin{pmatrix} \mathbb{1} & \mathbb{0} \\ \mathbb{0} & \sigma_x \end{pmatrix}$. An interesting implementation of the CNOT gate is to use it on a state

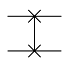
$$|\psi\rangle = H |0\rangle \otimes |0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$$

this gives us

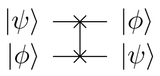
$$CNOT |\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle \otimes \mathbb{1} |0\rangle + |1\rangle \otimes X |0\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{27}$$

This is a famous state called the *Bell state*. It has the property that the qubits are correlated, meaning that if the first qubit is measured as 0 or 1, the second qubit will be found in the exact same state, and vice versa. Another two qubit gate is the SWAP gate, as one would guess the SWAP gate swaps the information of qubits. It is defined as

$$SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \tag{28}$$

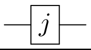
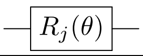
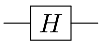
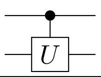
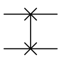


Where on the RHS. I wrote this using circuit representation. If we have two quantum states $|\psi\rangle$ and $|\phi\rangle$ what the SWAP gate does is change their places



In Table 1 is a summary of all the gates in circuit and matrix representation.

Table 1: Various quantum gates, their circuit and their matrix representation. The U is a general unitary operator that is used in a controlled gate. We use the notation $j \in \{x, y, z\}$ and that σ_j is the respective Pauli matrix Eq. (22)

Gate	Circuit representation	Matrix representation
j		σ_j
$R_j(\theta)$		$e^{i\frac{\theta}{2}\sigma_j}$
H		$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
U		$\begin{pmatrix} \mathbb{1} & \mathbb{0} \\ \mathbb{0} & U \end{pmatrix}.$
SWAP		$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

4.3 Measurements and uncertainty

We have talked about the quantum circuits that make a quantum evolution, the last step in the theory for quantum computers is to look at the measurements. We know from quantum mechanics that the probability of measuring a state is given by projectors of the eigenspaces. For the qubit, Eq. (19), we have two projector operators

$$P_0 = |0\rangle\langle 0| \text{ and } P_1 = |1\rangle\langle 1|$$

making it easy to see that the probability to measure $i = \{0, 1\}$ is

$$p(i) = \mathrm{Tr}(P_i |\psi\rangle\langle\psi|) = \langle\psi|P_i|\psi\rangle = |\alpha_i|^2 \tag{29}$$

After the measurement the state of the qubit changes to

$$|\psi\rangle \to \frac{P_i |\psi\rangle}{\sqrt{\langle\psi|P_i|\psi\rangle}} = |i\rangle$$

To more generally estimate the expectation value of qubits we can write our observables as a spectral decomposition of the computational basis

$$\hat{O} = \sum_{i=1} \lambda_i |i\rangle\langle i|$$

where we see P_i is included. To computationally check whether the state is in state $|0\rangle$ or $|1\rangle$ we can easily see that if we use a Z gate as our observable we get an eigenvalue of $+1$ for $|0\rangle$ and -1 for $|1\rangle$. Expanding Eq. (29) we get

$$\langle\psi|\hat{O}|\psi\rangle = \sum_i |\alpha_i|^2 \lambda_i \tag{30}$$

Since we know the eigenvalues λ_i, we only need to estimate the amplitudes of the state. Since we cannot directly measure the amplitudes of states we can use statistics. If we introduce a Bernoulli variable y_{ij} which is random such that we have $P(y_{ij} = 0) = 1 - |\alpha_i|^2$ and $P(y_{ij} = 1) = |\alpha_i|^2$. If we repeatedly prepare the state $|\psi\rangle$ and measure it in the computational basis, which is referred to as performing many *shots*, we can gather S samples $\{y_{i1}, \cdots, y_{iS}\}$. We also know that $|\alpha_i|^2$ can be estimated by a *frequentist estimator* $\hat{p}_i$ by

$$|\alpha_i|^2 \approx \hat{p}_i = \frac{1}{S}\sum_{j=1}^{S} y_{ij}$$

where the standard deviation of $\hat{p}_i$ is

$$\sigma(\hat{p}_i) = \sqrt{\frac{\hat{p}_i(1-\hat{p}_i)}{S}}$$

where we can say that the error goes as $\mathcal{O}(S^{-1/2})$. With this we can now approximate Eq. (30) to

$$\langle\psi|\hat{O}|\psi\rangle \approx \sum_i \hat{p}_i\lambda_i \pm \sqrt{\frac{\hat{p}_i(1-\hat{p}_i)}{S}} \tag{31}$$

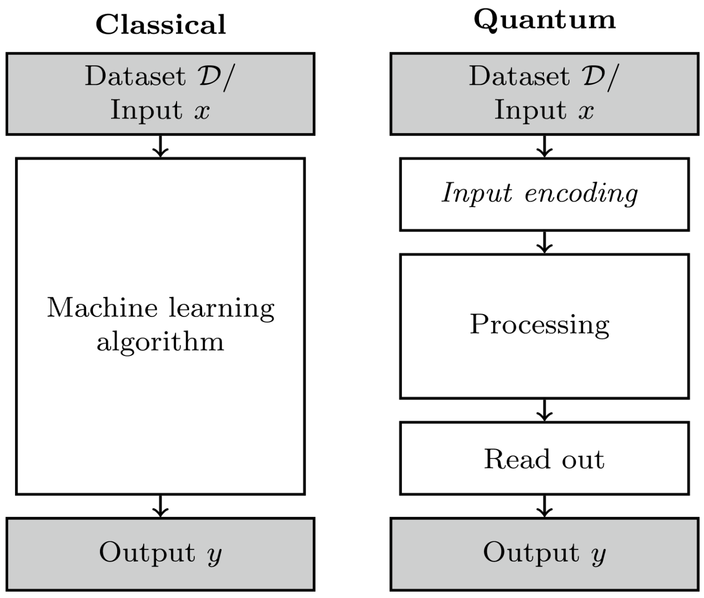
where we see that the error of the expectation value is also in the $\mathcal{O}(S^{-1/2})$. From this alone we see that it is computationally expensive to reduce the errors of measurements on quantum computers, i.e. to reduce the error by a factor of 10 we need at least 100 shots, meaning we need to measure the quantum states 100 times at least. This is a major drawback when studying *CQ* QML.

So far we have talked about an ideal quantum computer, and have shown that this ideal system still has a major drawback when it comes to measuring the state of the qubit. In reality the output of quantum circuits tents to be noisy, adding to even more errors. Specially when looking at Quantum computing on near-term quantum hardware, called *noisy intermediate-scale quantum computing* (NISQ) [10]. We can simulate these NISQ by using *backends* that simulate noise. With this we have a basic understanding of how the quantum circuits to be simulated work and we also have an idea of the potential barriers that might come.

5 Quantum Neural Network

The time has come. After going to the theory of NN's and quantum computing, we are ready to combine both fields to start exploring a QML algorithm. As in part I, we will only look at Quantum Neural Networks (QNN), also called variational circuits. As we want to simulate a QML algorithm on a near-term quantum computer, we should use as few qubits and quantum gates as possible, since larger routines would be frowned in noise when attempted to simulate on classical machines. However variational circuits changed the aim of QML, instead of aiming at speedups for known models, they create completely new models whole usefulness is still largely unknown. In this project we will focus on *probabilistic quantum models*, as we did on Eq. (31). So a *supervised probabilistic quantum model* in short, as explained in Chapter 5 on [7].

As in part I, the most important step is getting the data ready. Getting the data ready on quantum computers however is specially difficult when we are studying the *CQ* QML case as shown in Figure 8. Processing the classical data to be read in a quantum computer is by far the most critical part in this project. Since the goal is to do the HiggsML we have to first process the data in the same way, then encode it into a quantum state. As shown in Figure 9, the process doesn't "prepare" the data in the same way we do in classical ML. This process is often the most time consuming one.



Classical
Quantum
Dataset $\mathcal{D}$/ Input x
Machine learning algorithm
Output y
Dataset $\mathcal{D}$/ Input x
Input encoding
Processing
Read out
Output y

Figure 9: Showcase of how supervised learning tasks differ from *CC* to *CQ*. Taken from [7].

The first part of a QNN is to encode a feature vector $\boldsymbol{x}$ into the qubits of the quantum computer. This is done using what is often called a quantum feature map

$$|\psi_{\boldsymbol{x}}\rangle = U_{\phi(\boldsymbol{x})}|0\rangle \tag{32}$$

The second step is to use a circuit $U_{\boldsymbol{\theta}}$, parameterized by $\boldsymbol{\theta} = (\theta_1, \cdots, \theta_{n_\theta})$, on the state $|\psi_{\boldsymbol{x}}\rangle$

$$|\psi_{\boldsymbol{x},\boldsymbol{\theta}}\rangle = U_{\boldsymbol{\theta}}|\psi_{\boldsymbol{x}}\rangle \tag{33}$$

Here $U_{\boldsymbol{\theta}}$ is called an *ansatz* and works as a parameter-dependant way to transform the encoded data state $|\psi_{\boldsymbol{x}}\rangle$. Since the information of quantum states is not directly accessible, we can measure the model output as the expectation value of the QNN in the style of Eq. (31)

$$\hat{y} = f_{QNN}(\boldsymbol{x};\boldsymbol{\theta}) = \langle\psi_{\boldsymbol{x},\boldsymbol{\theta}}|\hat{O}|\psi_{\boldsymbol{x},\boldsymbol{\theta}}\rangle \tag{34}$$

But this is only equivalent to a feed forward network. After getting our output $\hat{y}$ we want a way to backwards propagate to optimize the parameters $\boldsymbol{\theta}$ by minimizing a cost function.

5.1 Input encoding

To fill the details of Eq. (32) we will use the simplest way of input encoding, *qubit encoding*. This encoding method requires p qubits, where p is the number of features in the dataset, and can be applied at a constant circuit depth. The way this goes is by first pre-processing the data by a function $\phi(x_i)$, for example scaling it. Then it is encoded by performing a Pauli rotation, Eq. (24), on every qubit with a rotational angle $\phi(x_i)$. This is seen in Figure 10.

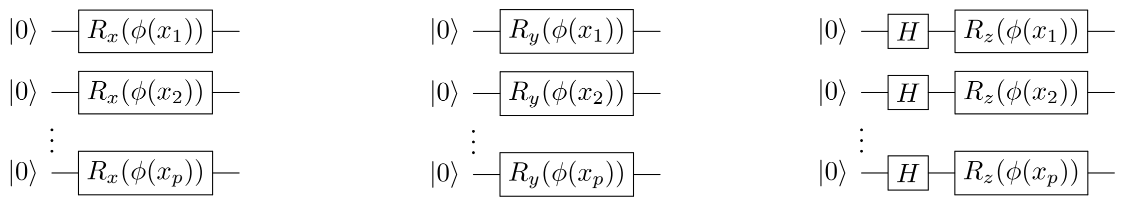


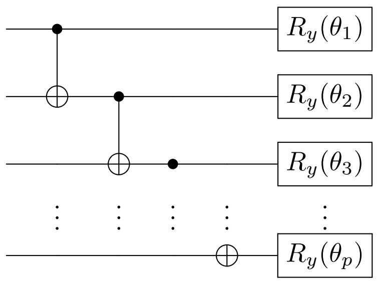
Figure 10: Qubit encoding using Pauli rotations, Eq. (24), on p qubits for each feature on the data. $\phi(x_i)$ illustrates pre-procesising of data inputs x_i. Taken from [8]

There we see that we need to apply a Hadamard gate, Eq. (23), when using R_z rotations, as otherwise these rotations would leave $|0\rangle$ unchanged. Qubit encoding gives a state where the amplitudes encode interaction between the features.

5.2 Processing

There are many unitary transformations we could use as an ansatz to process the data. We know that the vast majority of unitary transformations are inaccessible on ideal quantum computers, thus they would never work on NISQ. In this project we will use the same ansatz that Wold used, since it respects the limitations of NISQs. The ansatz is as follow

$U_{SA}(\boldsymbol{\theta}) =$



(35)

He called this the *simple ansatz* [8]. What it does is applies CNOT gates, Eq. (26), on neighboring qubits in sequence until the final qubit, creating entangled states between every qubit thus giving a larger space of transformations. The second step of the simple ansatz is to apply a R_y rotation to each qubit, each parameterized with its own parameter θ_i.

5.3 Readout

The last part of our quantum FFN is the readout, or model output $\hat{y}$. We know that we need to estimate the value of a state prepared by the encoder and ansatz from Eq. (34). But what operator should we use to estimate the expectation value? Since we want to ultimately do the HiggsML on this project, and this has a binary output, the easiest operator we can use to estimate the probabilities is the Z gate with the eigenvalues ± 1. However as we cannot directly estimate this value this proves difficult. Thus by following the example of Abbas et al. [11] and again Wold [8], we will choose the *parity* operator. The parity of an n qubit state can be formulated as

$$P = \frac{1}{2}\left(\mathbb{1}^{\otimes n} + \bigotimes_{i=1}^{n} \sigma_z\right) \tag{36}$$

which we see that when applied to a computational basis state as Eq. (20) gives

$$P\,|q_1 \cdots q_n\rangle = \frac{1}{2}\left(|q_1 \cdots 1_n\rangle - (-1)\sum_{i=1}^{n} q_i\,|q_1 \cdots q_n\rangle\right) = \bigoplus_{i=1}^{n} q_i\,|q_1 \cdots q_n\rangle$$

where $\bigoplus_{i=1}^{n} q_i$ is the mod 2 sum of the terms q_i, or parity of the bitstring $q_1 \cdots q_n$, i.e. the numbers in the ket. To summarize what this operator does, is set the parity to zero if the numbers of qubit in state $|1\rangle$ is even, and 1 if odd. Thus following Eq. (31) we have

$$\langle \psi_{\boldsymbol{x};\boldsymbol{\theta}} | P | \psi_{\boldsymbol{x};\boldsymbol{\theta}} \rangle \approx \frac{1}{S} \sum_{j=1}^{S} p_j$$

5.4 Network Optimization

Since we have now made the equivalent of a FNN, we now need to make the algorithm to optimize the parameters $\boldsymbol{\theta}$ such that we can closely get to the target values. Following Wold's master thesis [8]. If we have a circuit parameterized by $\boldsymbol{\theta}$ that prepares a state $|\psi_{\boldsymbol{\theta}}\rangle = U_{\boldsymbol{\theta}} |0\rangle$, Eq (32). Then we know the expectation value of some observable $\hat{O}$ can be written as

$$y = \langle \psi_{\boldsymbol{\theta}} | \hat{O} | \psi_{\boldsymbol{\theta}} \rangle = \langle 0 | U_{\boldsymbol{\theta}}^{\dagger} \hat{O} U_{\boldsymbol{\theta}} | 0 \rangle$$

Since we have constructed our circuit such that θ_i only works in a single gate. Then we can decompose the circuit as $U_{\boldsymbol{\theta}} |\mathbf{0}\rangle = A\mathcal{G}(\theta_i)B$, where $\mathcal{G}$ is the gate dependent on θ_i, and A and B is the rest of the circuit. With this we can write

$$y = \langle \psi' | \mathcal{G}(\theta_i)^{\dagger} \hat{O}' \mathcal{G}(\theta_i) | \psi' \rangle$$

where $|\psi'\rangle = B|0\rangle$ and $\hat{O}' = A^{\dagger}\hat{O}A$. Such that we can begin to compute

$$\frac{\partial y}{\partial \theta_i} = \langle \psi' | \mathcal{G}(\theta_i)^{\dagger} \hat{O}' (\partial_{\theta_i} \mathcal{G}(\theta_i)) | \psi' \rangle + h.c.$$

We cannot calculate this on a quantum computer since they aren't expectation values, we can however write them as a linear combination of two expectation values

$$\begin{aligned} \frac{\partial y}{\partial \theta_i} = \frac{1}{4}(&\langle \psi' | [\mathcal{G}(\theta_i) + 2\partial_{\theta_i}\mathcal{G}(\theta_i)]^{\dagger} \hat{O}' [\mathcal{G}(\theta_i) + 2\partial_{\theta_i}\mathcal{G}(\theta_i)] | \psi' \rangle - \\ &\langle \psi' | [\mathcal{G}(\theta_i) - 2\partial_{\theta_i}\mathcal{G}(\theta_i)]^{\dagger} \hat{O}' [\mathcal{G}(\theta_i) - 2\partial_{\theta_i}\mathcal{G}(\theta_i)] | \psi' \rangle) \end{aligned} \tag{37}$$

But what we still don't know is whether $[\mathcal{G}(\theta_i) + 2\partial_{\theta_i}\mathcal{G}(\theta_i)]$ and $[\mathcal{G}(\theta_i) - 2\partial_{\theta_i}\mathcal{G}(\theta_i)]$ are unitary operators? If they weren't we couldn't use them in our circuit. But since we are only using Pauli rotations, they are easy to implement! If we write $\mathcal{G}(\theta_i) = R_j(\theta_i)$ we get

$$\mathcal{G}(\theta_i) \pm 2\partial_{\theta_i}\mathcal{G}(\theta_i) = (I \mp i\sigma_j)\mathcal{G}(\theta_i) = \sqrt{2}\mathcal{G}\left(\theta_i \pm \frac{\pi}{2}\right)$$

since $I \mp i\sigma_j = \sqrt{2}\mathcal{G}(\pm\frac{\pi}{2})$ and $R_j(a)R_j(b) = R_j(a+b)$. Inserting this in Eq. 37 we get

$$\frac{\partial y}{\partial \theta_i} = \frac{1}{2}\left(\langle \psi' | \mathcal{G}\left(\theta_i + \frac{\pi}{2}\right)^{\dagger} \hat{O}' \mathcal{G}\left(\theta_i + \frac{\pi}{2}\right) | \psi' \rangle - \langle \psi' | \mathcal{G}\left(\theta_i - \frac{\pi}{2}\right) \hat{O}' \mathcal{G}\left(\theta_i - \frac{\pi}{2}\right) | \psi' \rangle \right) \tag{38}$$

The tool used here is called the *parameter shift rule*. It says that to calculate the derivative of the expectation value of a circuit, we have to estimate the expectation value at least twice. Here we shifted by $\pm\frac{\pi}{2}$. Thus we find the derivative by combining the two results in a linear combination.

With this we can expand our loss minimization using the MSE, Eq. (10), as a cost function and find the gradient as in part I

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^L} = \left(y_j^L - t_j\right) \frac{\partial y_j^L}{\partial w_{jk}^L}$$

where $w \to \theta$ in this case. We also know from Eq. (13) that this equals

$$\frac{\partial C(\hat{W}^L)}{\partial \theta_{jk}^L} = \delta_j^L y_k^{L-1}$$

or rather

$$\delta_j^L y_k^{L-1} = \left(y_j^L - t_j\right) \frac{\partial y_j^L}{\partial \theta_{jk}^L} \tag{39}$$

such that we can update the parameter the same way as in part I

$$\theta_{jk}^l \leftarrow \theta_{jk}^l - \eta \delta_j^l y_k^{l-1}$$

5.5 Summary

To summarize how an algorithm works in QML we again make a list

- Scale the data and do qubit encoding so the data is made into a quantum state
- Process the data so we entangle the states and introduce a parameter $\boldsymbol{\theta}$ that works as our weight to be trained
- Estimate the output S times since the output is probabilistic.
- To update the weights we minimize a loss function, to find the gradient we use the parameter shift rule
- Repeat as many times as desired

6 Implementation

The last step is now to implement this into code. To do so we first have to introduce the most important tool, how to simulate our quantum circuit. What has been used in this project is a Python library called Qiskit. Qiskit is an open-source quantum development tool at the level of pulses, circuits and application modules. [12]. On this project we also utilized two backends from IBM Quantum, a 7 qubit backend: IBM Q team, "IBM Q 7 Nairobi backend specification V.1.0.25" and 5 qubit backend: IBM Q team, "IBM 5Q Manila backend specification V.1.0.36" (2022) both retrieved from https://quantum-computing.ibm.com. There are also backends with 50+ qubits on IBMQ, however these are not free. I will refer to the quantum circuits with the respective backends as just the name of the city. As we have seen on the theoretical section of this part, we didn't need to find the number of neurons in a grid search, to also make the computation easier we will not use the L2-norm either. With this we can get right into how to simulate our quantum circuit. As mentioned all the codes and plots can be found in my GitHub repository in the following link: https://github.com/rubenguevara/QuantumMachineLearning

6.1 Algorithm

The first step is to use Qiskit to actually make the quantum circuit, this is easily done by using the inbuilt class *QuantumCircuit* which takes a our initial n qubit state this can be made using the class *QuantumRegister*. Then we can make our own function to do the qubit encoding. In this project we will use an R_X rotation, which can easily be implemented when calling the quantum circuit using the function $rx()$ which takes the quantum state as well as our data inputs $\boldsymbol{x}$ as inputs, the same way as Eq. (32). After that we include our simple ansatz by making a function that first entangling a qubit with the next one using a CNOT gate when calling the circuit with the function $cx()$, which takes as arguments the qubits to entangle. Then we complete our simple ansatz by doing a R_y rotation with the trainable parameter $\boldsymbol{\theta}$ by calling the circuit using the function $ry()$ which takes the parameter θ and the qubit as input.

Then to ensure that the circuit applies the gates in the given order we call the circuit and use the function *measure_all* to set it. Afterwards we can use the *execute* class to create a "job" where we can count the output of the circuit using a backend and specifying the number of shots. This I will plot so we can see the distribution of quantum states for the different backends.

The last step is to implement the parity operator. Using the counts as argument we can create a function that does the same as Eq. (36). The algorithm for the the QNN can be seen in Algorithm 2

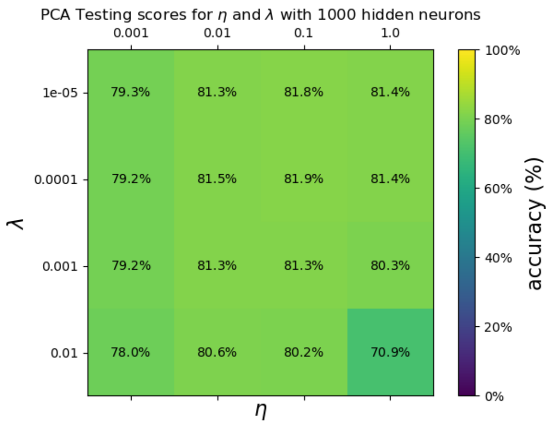
Algorithm 2: Quantum Neural network algorithm using Qiskit.

```python
def qnn(x, theta, backend, shots):
    n_qubits = len(x[1])
    data_reg = qk.QuantumRegister(n_qubits)
    clas_reg = qk.ClassicalRegister(n_qubits)
    circuit = qk.QuantumCircuit(data_reg, clas_reg)

    circuit = qubit_encoder(circuit, data_reg, x)          # 5.1
    circuit = ansatz(circuit, data_reg, theta, n_qubits)    # 5.2
    circuit.measure_all()

    job = qk.execute(circuit,
                     backend,
                     shots = shots)
    counts = job.result().get_counts(circuit)

    y_pred = parity(counts, shots, n_qubits)                # 5.3
    return y_pred, counts
```

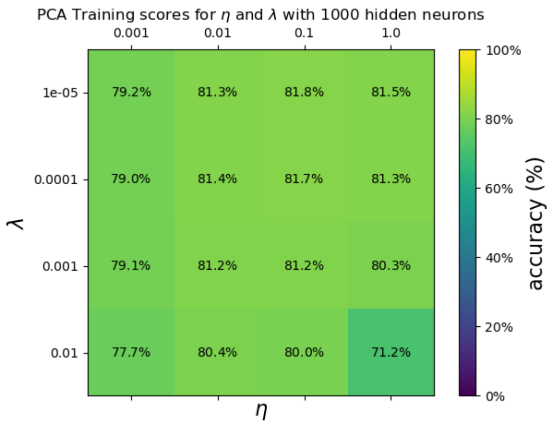
6.2 Principal Component Analysis

We're almost there, before doing the HiggsML with Qiskit we need to reduce our number of features to reduce the required number of required qubits, since we are using qubit encoding, Figure 10. How can we reduce the number of features you ask? Well we can use a mathematical tool called Principal Component Analysis (PCA). The way PCA works, is by rotating the feature-space-matrix and seeing how the features are correlated. In other words, it rotates the whole matrix to reduce the number of eigenvalues to one of our choosing.

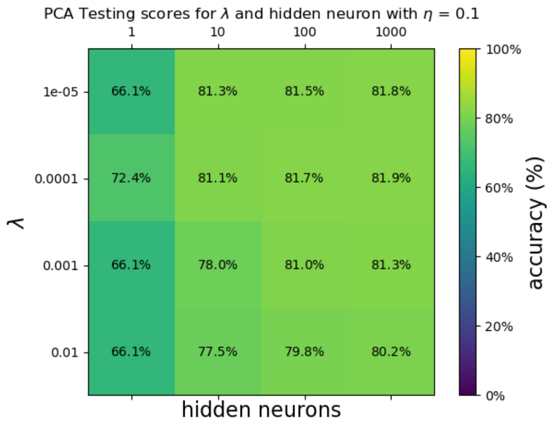
Before beginning to do our analysis we need to make a new grid search to see if the parameters to use on our ML model change from those in part I. Choosing as a starting point to reduce our number of features to 7, which still holds 97.8% of all the information of the original dataset (see Table 2). Doing the same grid search as in part I of this project we get the parameters shown in Fig. 11.



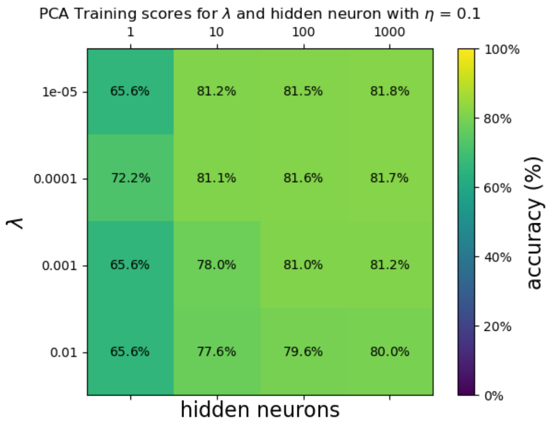
(a)



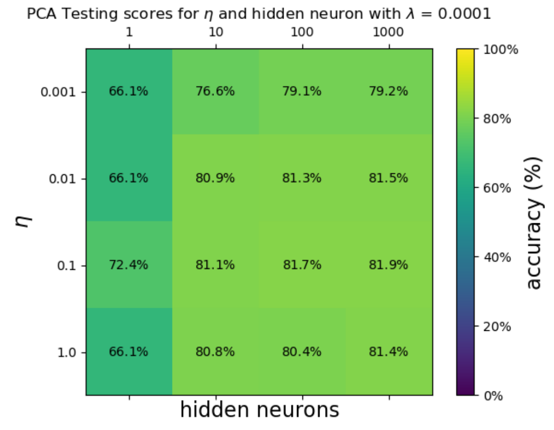
(b)



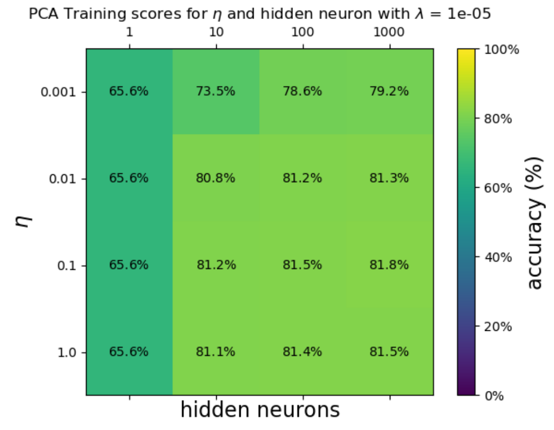
(c)



(d)



(e)



(f)

Figure 11: Grid search for best hyperparameters when reducing the number of features to 7 using PCA.

If we assume that the parameters are the best for all PCA reduced datasets, which seems to be the case since we have about the same parameters as the full feature cases, then we can make a table showcasing all the information for different number of features. This is shown in Table 2.

Table 2: How the information of the dataset looks after reducing the number of features with PCA. The first column is the number of features, the second is how much information is kept and the last two are ML scores. The last row with 30 features holds all the features (no PCA).

Number of features	Information after PCA [%]	Training accuracy [%]	Testing accuracy [%]
2	87.7	69.2	69.6
3	92.1	73.1	73.5
4	94.1	74.1	74.4
5	95.6	78.8	79.0
6	96.9	81.4	81.6
7	97.8	81.5	81.4
8	98.6	82.3	82.3
9	99.0	82.7	82.7
10	99.4	83.0	82.9
15	99.99	83.6	83.5
20	100	83.9	83.5
30	100	85.1	84.1

To illustrate the pattern that emerges from this, we can look at Figure 12 that summarized Table 2

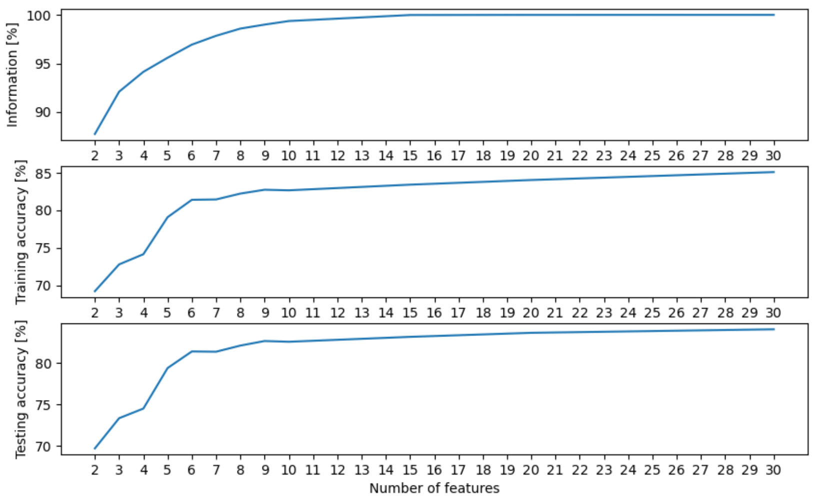


Figure 12: How different number of features affect the model.

Plotting all of these into a ROC curve we can see how the different models compare, this is shown in Fig. 13 As we can see we get a decent ROC area with the model that has reduced the number of features to 6. This is great since the aim of this project is to be used in NISQ. Therefore we have now reduced the features from 30 to 6, meaning we should be able to run the code with 6 qubits. To further showcase that reducing the number of features does not ruin our accuracy we can see how well the model separates signal from background. This is shown in Fig. 14. There we see that the results are very good, specially when considering that we reduced a 30 dimensional feature space to just 6! However this is not good enough. If we tried to run just one "Feed Forward" to get a single output from this quantum circuit, we would have to go through 250,000 samples which is just too computationally heavy to be used.

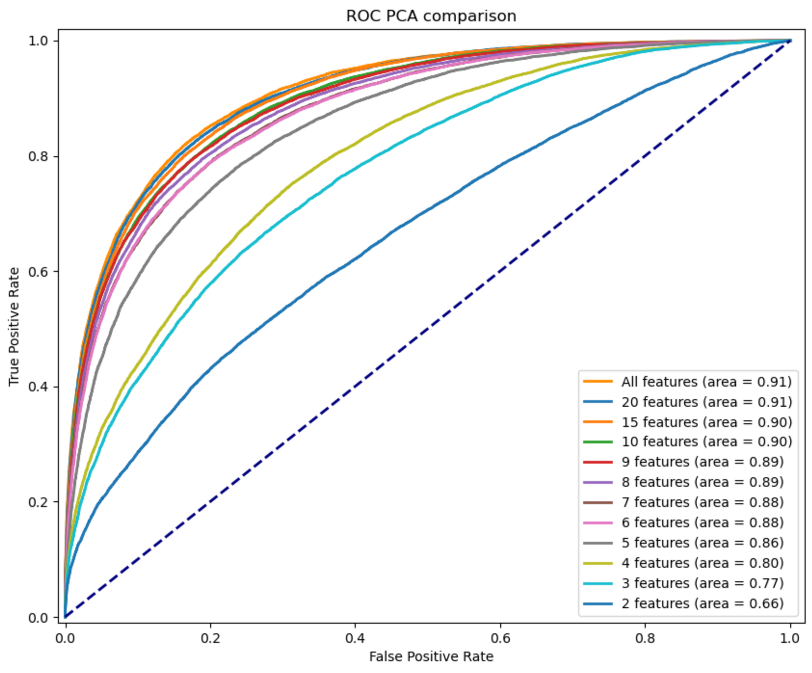


Figure 13: ROC curve for model with different number of features.

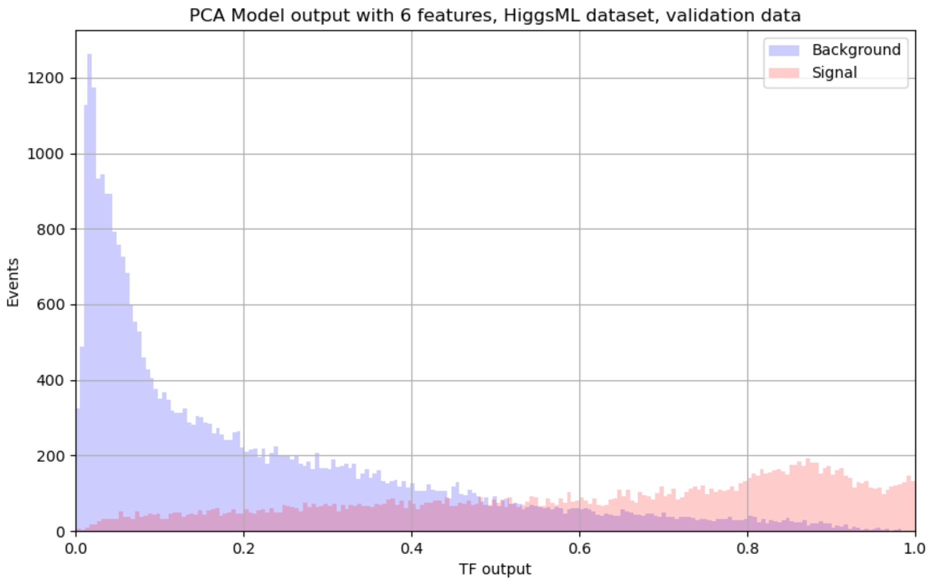


Figure 14: Reconstruction error histogram after reducing the number of features to 6 with PCA.

6.3 Low Statistics Regime

The next problem is reducing the 250,000 samples we have. Since we want to reduce the run time as much as possible. Even though we have reduced the number of features to 6, meaning we only need 6 qubits to start the quantum circuit, we are on the edge of what the quantum circuit can do. To reduce the number of samples we first need to find the signal to background ratio from the original HiggsML training set. Then with this we can use Pandas DataFrame to randomly reduce the number of samples while keeping the same signal to background ratio. We do this randomly to reduce all possible bias from the equation. I choose to reduce the number of samples from 250,000 to 1,000 to still have a small amount of statistics. Doing the same good old analysis we get the results shown in Figure 15-18.

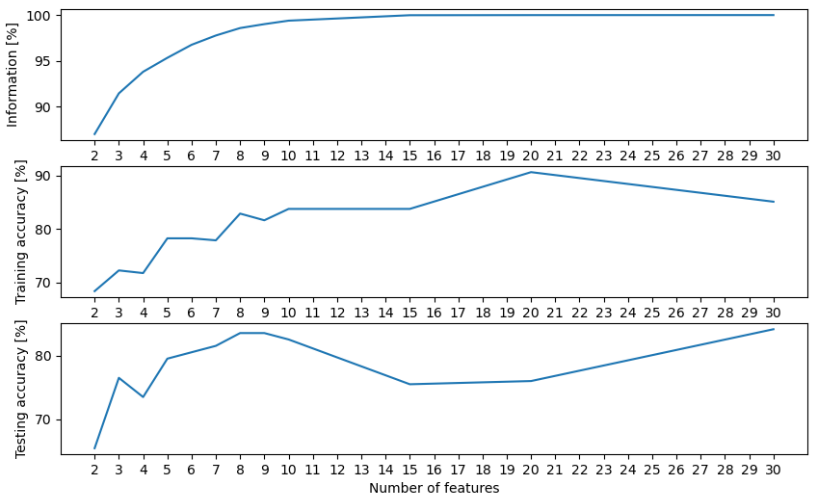


Figure 15: How the number of features affect our model trained on 1,000 samples.

From this analysis we notice a few things: the network get generally lower accuracy scores, different features over train to accuracies to up to 90% and most importantly we can reduce the number of features to 5.

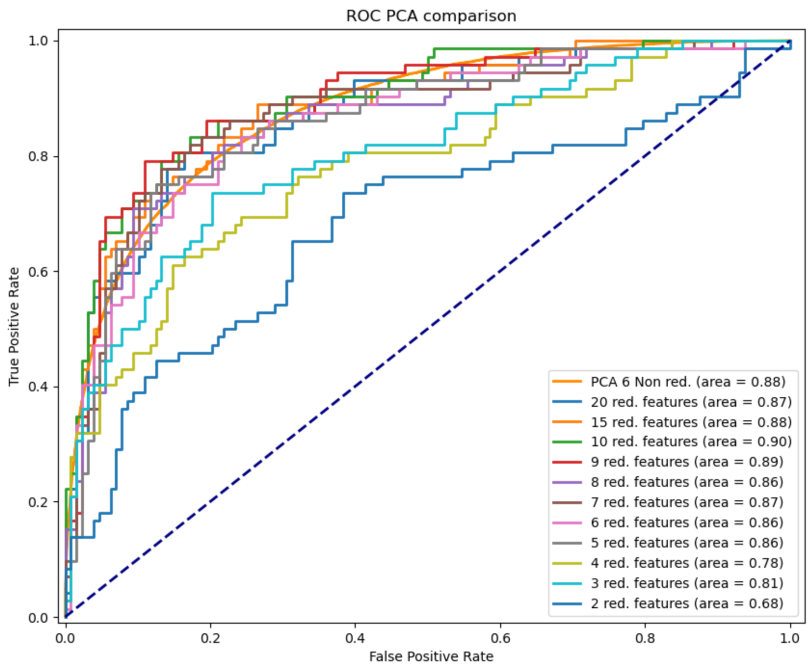


Figure 16: ROC curve for model trained with 1,000 samples with different number of features.

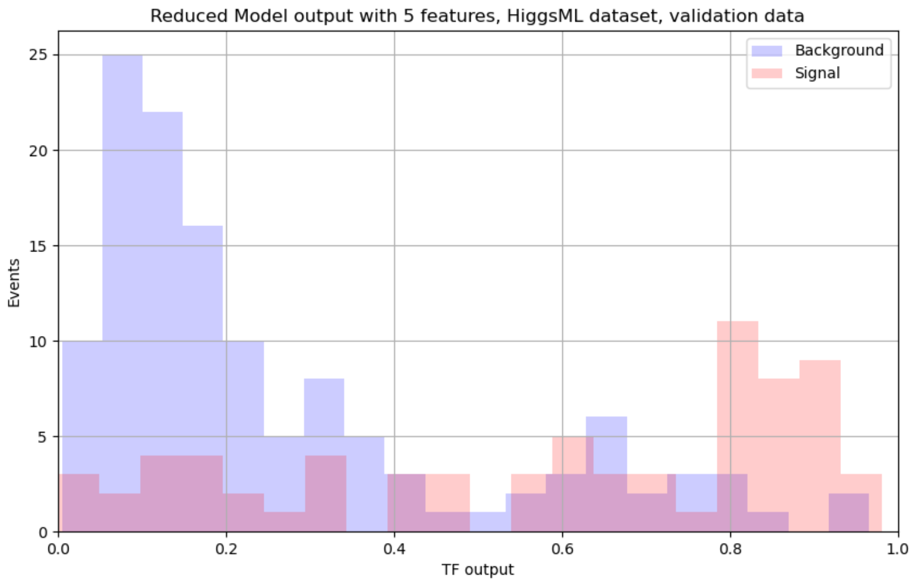
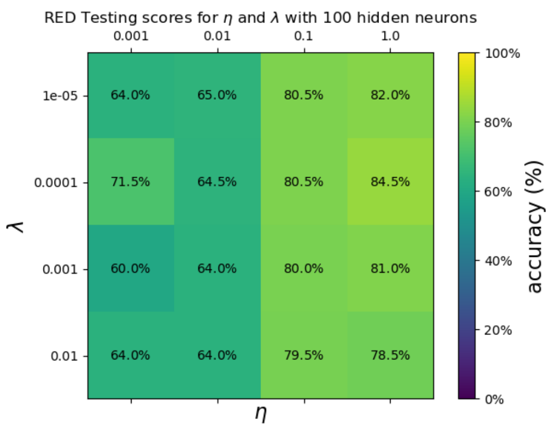
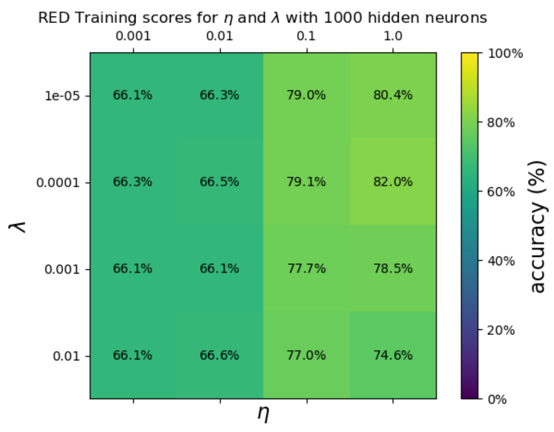


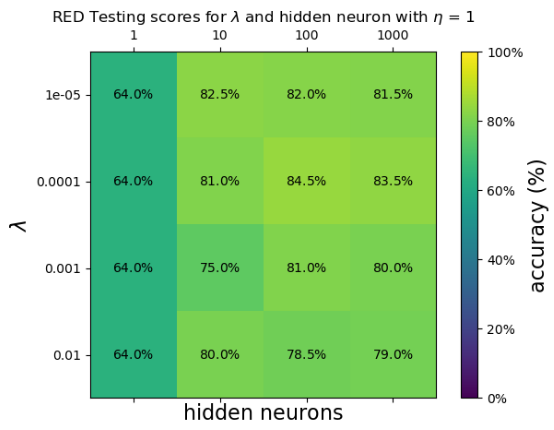
Figure 17: Reconstruction error histogram after reducing the number of samples to 1,000 an the number features to 5.



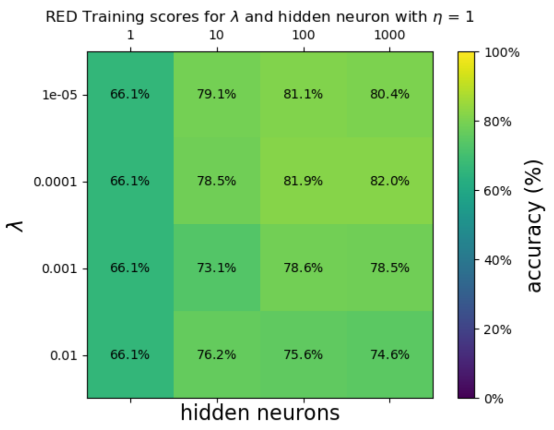
(a)



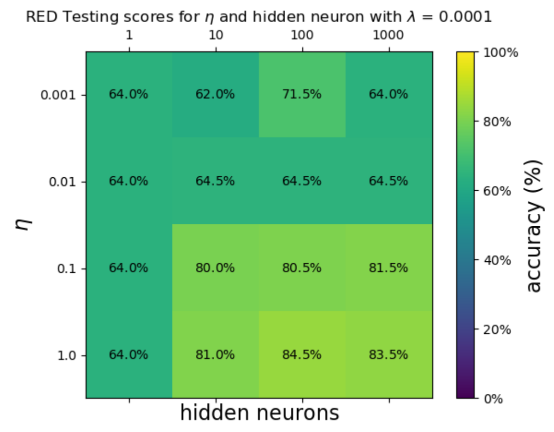
(b)



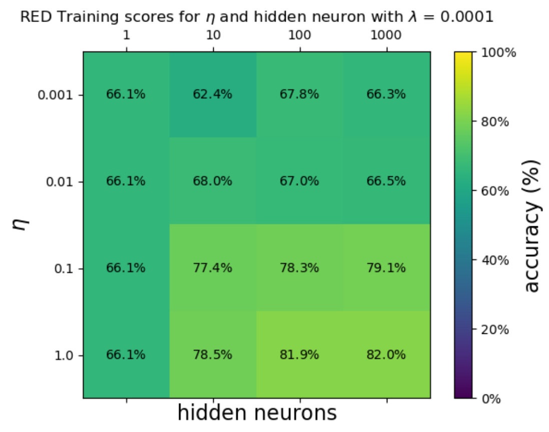
(c)



(d)



(e)



(f)

Figure 18: Grid search when reducing data from 250,000 samples with 30 features to 1,000 samples with 6 features.

7 Results

The last step is to scale the data $\boldsymbol{x}$ for the qubit encoding, I will for the purposes of this project scale the data from $[0, 1]$. With the number of features reduced down to five and the samples reduced to 1000, we can finally begin talking about QML. When running our quantum circuit on just one "Feed Forward" to see how the output looks we get the counts shown in Figure 19 for Nairobi and Manila using 1000 shots, showcasing the probabilistic nature of qubits. To also make sure that Qiskit makes the quantum circuit we want, we can print the circuit. This is shown in Figure 20. After training the network for 10 epochs on the training set we can see how the loss function evolves pr. epoch in Figure 21. We can also see the ROC scores of this test in Figure 22 and 23 for Nairobi and Manila, respectively.

From these results we see that the QML algorithm is not that good at separating the signal from the background. Giving us an accuracy of 71% and 64% on the training sets and 67% and 66% on the testing sets for Nairobi and Manila respectively. We see that the model over-trains and does generally better on Nairobi, this I assume is due that Nairobi is a "better" backend than Manila since it has more qubits. We also see that the model scores generally lower for Manila but also under-trains, this I do not know the reason for. Something noteworthy is that Nairobi took less time to train than Manila, which supports my idea of it being "better", however, the training process took over an hour to finish for each backend.

With that in mind we come to the disappointing conclusion that doing the HiggsML on NISQ is not possible, **yet!** However one thing that was done, was destroying all statistics that we had by reducing the number of samples to 1,000. Something which is highly controversial, specially for high energy physics which is notorious for its statistics. For future projects to do high energy physics related QML, we should first of all wait until we have more qubits available (or pay IBMQ to use their premium backends) in a system such that the need to reduce the number of features is not so crucial. On the simulated quantum circuit we see that when using a backend with more qubits makes the algorithm run faster, such that we hopefully could do the HiggsML with 10 features, still holding 99.4% of the information, and all samples in the future. As for a "quantum speedup", from the results of this project nothing seems to indicate that there might be an advantage to do this this way. However since we are at an early stage of quantum computing this can quickly change in the future. Also if we were to try the HiggsML on the same computing stage of quantum computers on classical systems it might run even slower than this algorithm, this is something that can further be studied. For the future an exiting and ambitious idea (far from NISQ) would be to explore the possibility of QQ QM. Where the data could directly be generated into quantum states by detectors at particle collider, this is already a subject that has been brought up at CERN [13]. If we were to use quantum generated data we could develop an algorithm that doesn't require us to measure a state as many times as we do now, drastically reducing computational power. We're at an exiting time right now, since there still at an early stage of quantum technology and this project shows promise that it will be possible to do QML on high energy physics.

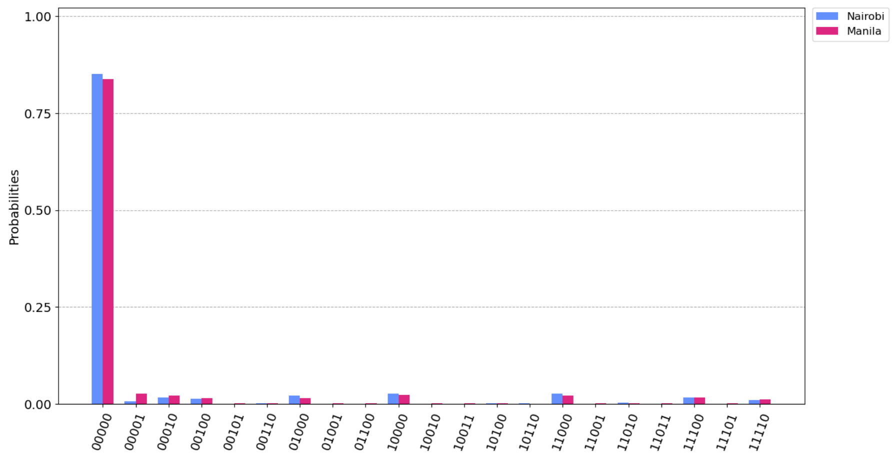


Figure 19: Probabilities to measure a quantum state using the Nairobi and Manila backend with 1000 shots.

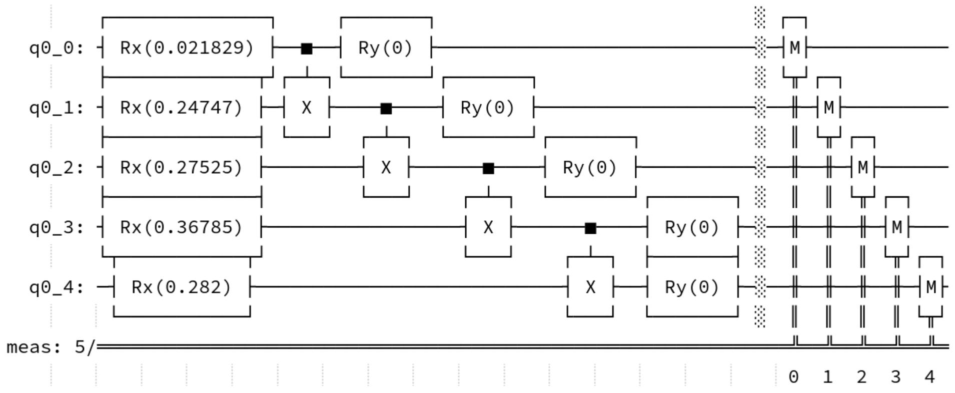


Figure 20: Output from Qiskit showing how our quantum circuit looks. This is for the first sample of the data points.

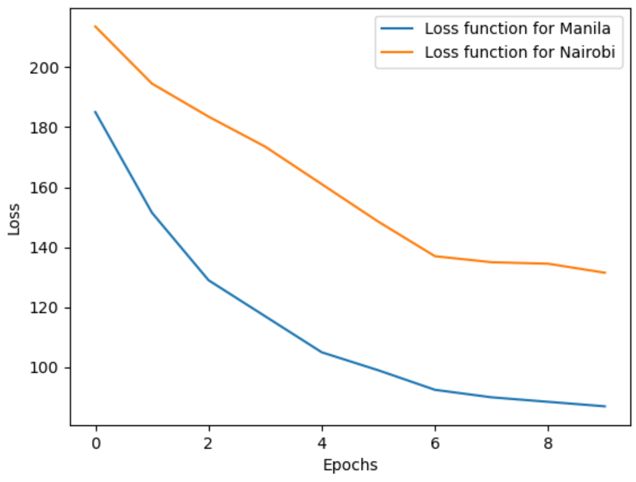


Figure 21: How the loss function changes pr epoch on our QML algorithms.

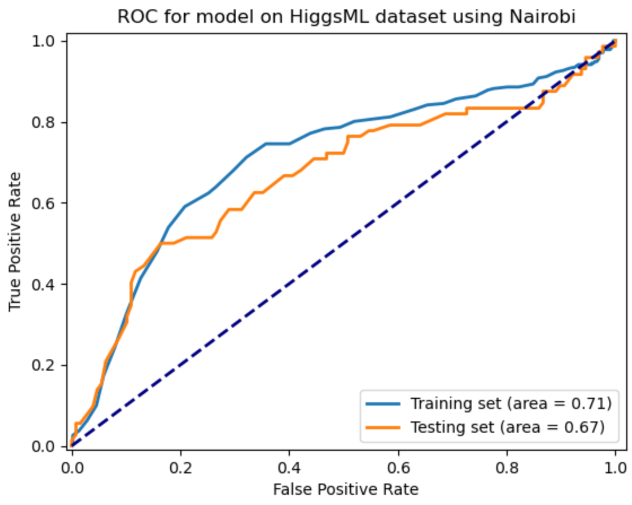


Figure 22: ROC curve for our QML algorithm using Nairobi.

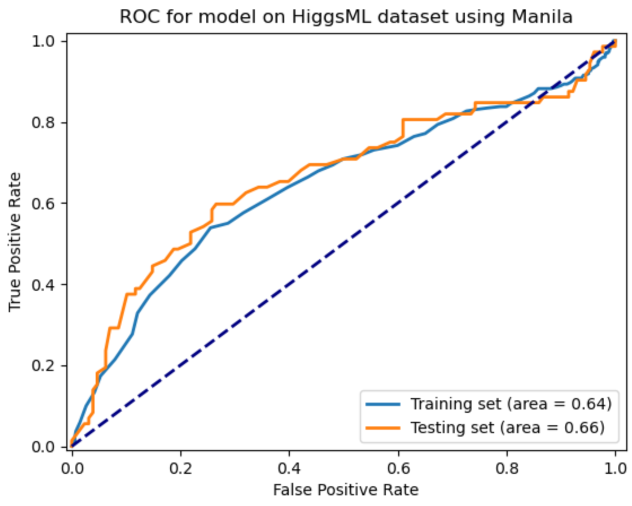


Figure 23: ROC curve for our QML algorithm using Manila.

A HiggsML Features

Of the features three are prefix-less variables: *EventId* (which is the ID of the event), *Weight* (used to calulate the AMS) and *Label* (stating whether the event is a signal or background process). All the other features are shown below.

- DER_mass_MMC
- DER_mass_transverse_met_lep
- DER_mass_vis
- DER_pt_h
- DER_deltaeta_jet_jet
- DER_mass_jet_jet
- DER_prodeta_jet_jet
- DER_deltar_tau_lep
- DER_pt_tot
- DER_sum_pt
- DER_pt_ratio_lep_tau
- DER_met_phi_centrality
- DER_lep_eta_centrality
- PRI_tau_pt
- PRI_tau_eta
- PRI_tau_phi
- PRI_lep_pt
- PRI_lep_eta
- PRI_lep_phi
- PRI_met
- PRI_met_phi
- PRI_met_sumet
- PRI_jet_num
- PRI_jet_leading_pt
- PRI_jet_leading_eta
- PRI_jet_leading_phi
- PRI_jet_subleading_pt
- PRI_jet_subleading_eta
- PRI_jet_subleading_phi
- PRI_jet_all_pt

Where their physical significance is explained in the original HiggsML article. [1]

References

1. Adam-Bourdarios C, Cowan G, Germain C, Guyon I, Kégl B, and Rousseau D. The Higgs boson machine learning challenge. *Proceedings of the NIPS 2014 Workshop on High-energy Physics and Machine Learning*. Ed. by Cowan G, Germain C, Guyon I, Kégl B, and Rousseau D. Vol. 42. Proceedings of Machine Learning Research. Montreal, Canada: PMLR, 2015 Dec :19–55. Available from: https://proceedings.mlr.press/v42/cowa14.html
2. Gaillard M. CERN Data Centre passes the 200-petabyte milestone. CERN 2017 Jul 6. Available from: https://home.cern/news/news/computing/cern-data-centre-passes-200-petabyte-milestone
3. Hjorth-Jensen M. Week 40: From Stochastic Gradient Descent to Neural networks. 2021 Nov. Available from: https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40.html
4. Hjorth-Jensen M. Week 41 Constructing a Neural Network code, Tensor flow and start Convolutional Neural Networks. 2022 Aug. Available from: https://compphysics.github.io/MachineLearning/doc/pub/week41/html/week41.html
5. Hjorth-Jensen M. 6. Logistic Regression. 2021. Available from: https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter4.html
6. Google Brain team. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Available from: https://www.tensorflow.org/
7. Schuld M and Petruccione F. Machine Learning with Quantum Computers. Springer Cham, 2021. DOI: https://doi.org/10.1007/978-3-030-83098-4
8. Wold K. Parameterized Quantum Circuits for Machine Learning. MA thesis. Universitet i Oslo, 2021. Available from: https://www.duo.uio.no/handle/10852/89534
9. Barenco A, Bennett CH, Cleve R, DiVincenzo DP, Margolus N, Shor P, Sleator T, Smolin JA, and Weinfurter H. Elementary gates for quantum computation. Physical Review A 1995 Nov; 52:3457–67. DOI: 10.1103/physreva.52.3457
10. Preskill J. Quantum Computing in the NISQ era and beyond. Quantum 2018 Aug; 2:79. DOI: 10.22331/q-2018-08-06-79
11. Abbas A, Sutter D, Zoufal C, Lucchi A, Figalli A, and Woerner S. The power of quantum neural networks. Nature Computational Science 2021 Jun; 1:403–9. DOI: 10.1038/s43588-021-00084-1
12. IBM Research and Qiskit community. Qiskit: An Open-source Framework for Quantum Computing. 2021. DOI: 10.5281/zenodo.2573505
13. Di Meglio A, Doser M, Frisch B, Grabowska D, Pierini M, and Vallecorsa S. CERN Quantum Technology Initiative Strategy and Roadmap. Version 1.0_Rev2. 2021 Oct. DOI: 10.5281/zenodo.5571809