Model independent search for Dark Matter in dilepton + MET final states with the ATLAS detector at the LHC

Ruben Guevara

Thesis submitted for the degree of
Master in Physics: Nuclear and Particle Physics
60 credits

Department of Physics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2023

Model independent search for Dark Matter in dilepton + MET final states with the ATLAS detector at the LHC

Ruben Guevara

Acknowledgements

Thank you everybody<3<3

Abstract

Something something

Contents

List of Figures

List of Tables

Part I

The theory behind modern particle physics

Chapter 1

The Standard Model of Particle Physics and Beyond

1.1 The Basics

1.1.1 Classical Field Theory

1.1.2 Quantum Mechanics

1.1.3 Special Relativity

1.2 Quantum Field Theory

1.2.1 Fermionic and Bosonic fields

1.2.2 Quantum Electro Dynamics

1.2.3 Quantum Chromo Dynamics

1.2.4 Electroweak unification

1.2.5 The Brout-Englert-Higgs Mechanism

1.2.6 The Standard Model of Particle Physics

1.3 Beyond Standard Model

1.3.1 Dark Matter

Chapter 2

Detection and Analysis

2.1 The ATLAS Detector

2.2 Classical Data Analysis

Chapter 3

Machine Learning

3.1 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).

3.1.1 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

$$\boldsymbol{\beta}_{i+1} = \boldsymbol{\beta}_i - \eta \nabla_{\beta} C(\boldsymbol{\beta}_i) \tag{3.1}$$

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. (3.1) 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.

3.1.2 Artificial neurons

As stated by Hjorth-Jensen in [1]:

> *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

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u) \tag{3.2}$$

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. (3.2) 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.

3.1.3 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

$$f(x) = \frac{1}{1+e^{-x}} \tag{3.3}$$

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

$$f(x) = x^{+} = \max(0, x) = \begin{cases} x & \text{if } x > 0, \\ 0. & \text{otherwise.} \end{cases} \tag{3.4}$$

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

3.1.4 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. (3.3) 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

$$C(\boldsymbol{\beta}) = -\sum_{i=1}^{n} (y_i(\beta_0 + \beta_1 x_i) - \log(1 + \exp(\beta_0 + \beta_1 x_i))) \tag{3.5}$$

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

$$C(\boldsymbol{\beta}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i(\boldsymbol{\beta}) \rightarrow \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i(\boldsymbol{\beta}) + \lambda \sum_{ij} w_{ij}^2 \tag{3.6}$$

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.

3.1.5 Feed Forward network

To describe how the network "guesses" outputs in a mathematical model we can compute we can start by looking at Eq. (3.2) 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

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(z) \tag{3.7}$$

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. (3.7) 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

$$z_i^1 = \sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1 \tag{3.8}$$

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

$$y^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^{l+1} f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^l \left(\cdots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\cdots\right) + b_j^l\right) + b_i^{l+1}\right] \tag{3.9}$$

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. (3.9) 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 [1] shows the main idea of how a Feed forward network is set up, this is shown in Figure 3.1.

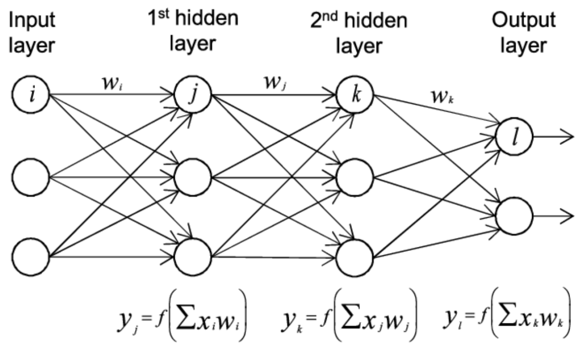


Figure 3.1: Basic illustration of a network with two hidden layers. Image taken from [1]

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

$$C(\hat{W}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - t_i)^2 \tag{3.10}$$

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. (3.8) 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. (3.7), we have

$$\frac{\partial z_j^l}{\partial w_{ij}^l} = y_i^{l-1} \tag{3.11}$$

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

$$\frac{\partial y_j^l}{\partial z_j^l} = y_j^l(1 - a_j^l) = f(z_j^l)(1 - f(z_j^l)) \tag{3.12}$$

We also need to take the derivative of Eq. (3.10) 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. (3.11) and Eq. (3.12)

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

Such that we have

$$\frac{\partial C(\hat{W}^L)}{\partial w_{jk}^L} = \left(y_j^L - t_j\right) y_j^L(1 - y_j^L)y_k^{L-1} := \delta_j^L y_k^{L-1} \tag{3.13}$$

where we have defined the error

$$\delta_j^L := \left(y_j^L - t_j\right) y_j^L(1 - y_j^L) = f'(z_j^L)\frac{\partial C}{\partial y_j^L} \tag{3.14}$$

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. (3.14), 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. (3.14) is easily computed. Thus we can also see how the weight changes the cost function using Eq. (3.13) quite easily. One thing else we can compute with Eq. (3.14) 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

$$\delta_j^L = \frac{\partial C}{\partial b_j^L}\frac{\partial b_j^L}{\partial z_j^L} = \frac{\partial C}{\partial b_j^L} \tag{3.15}$$

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. (3.13 - 3.15) we see that if a neuron output y_j^L is small, then the gradient term, Eq. (3.13), 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. (3.14) 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. (3.8) (replacing 1 with $l+1$) we get

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l) \tag{3.16}$$

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

3.1.7 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. (3.9)
- Secondly we commit all the way on Eq. (3.9) 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. (3.14) for all values j.
- Then we back propagate the error for each $l = L-1, l-2, \cdots, 2$ with Eq. (3.16).
- The last step is then to update the weights and biases using Eq. (3.1) 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.

3.2 Boosted Decision Trees

3.3 Ensemble modeling?

Part II

Implementation

Chapter 4

Data Analysis

4.1 Background Estimation

4.2 Kinematic Variables

4.3 Dark Matter samples

Chapter 5

Machine Learning

5.1 Data Preparation/LOG

We are dropping $\Delta\Phi(l_1, l_2)$ and $\Delta\Phi(l_c, E_T^{miss})$ due poor overenstemmelse with data. The first one is most likely due us not including fake leptons (and also for all non SFOS final states). The latter is a problem that PhD. Even is being haunted by. There is also the problem of missing variables. For the dataset being used I only look at, at least, two jets in the final state. This does however not always happen, to "fill the gaps" I made the p_T of the jets equal to zero if there are less than two events, which is physically reasonable. And more problematic, I set the η and ϕ to 10, which has no physical meaning. Luckily this does not seem to be an important feature when training the network using XGBoost. There is also another problem, albeit less problematic than the previous ones, with the final states that are not SFOS, as the background on these tend to be lower than the data. The number of events that are not SFOS are minimal though, and we think the reason it doesn't fit the data is because we are not including fake leptons.

5.1.1 Full Dark Matter dataset

To train the networks I will utilize two methods. The first one being this where the dataset being sent into the ML network will contain every single DM MC sample available. So far there are 154 different MC samples, these are based on three theories. A Light Vector(LV), Dark Higgs (DH) and Effective Field Theory (EFT) vertex/propagator which produces the WIMP DM particles. As well as a new theoretical particle, Z', and decays into the lepton pair observed. The three theories are divided further into MC samples with a Light Dark Sector (LDS) and High Dark Sector (HDS) which tells us the range of the Dark Matter candidate mass. And lastly it is divided further into more MC samples with different masses for Z'. This dataset includes all of these samples such that the network learns Dark Matter in a model independent way.

is this true?

5.1.2 "Ensemble" dataset

Another approach is to make multiple datasets and combine the results of every network into a "big network". This is the second approach which I call ensemble modelling. The thought behind this is that when teaching a network using the full dataset it might only focus on a few special ones that stand out more than others on the massive dataset. Also, every different DM sample has different phenomenology, specially in the future when I will be receiving SUSY samples, meaning that it also won't teach the network physics. Thus if we were to train a network one one sample at a time it would be the perfect scenario. However as will become apparent in Section 5.2.3, the datasets (even the full DM dataset) are extremely unbalanced. To put some numbers, on each DM MC sample there are roughly 40,000 MC events, and for the SM background (with a massive MET > 50GeV cut!) there are roughly 87,000,000 MC events. Factoring the weights (i.e. cross sections) gives us an extremely low statistics dataset, even in the full DM dataset.

Thus making the approach to teach the networks one MC sample at a time is impossible . So far I have tried dividing the the MC samples into 18 different categories. First into their respective theory. Then into LDS or HDS. Then into three $m_{Z'}$ regions, where I've defined the *low mass region* to be ≤ 600

for NN

GeV, the *middle mass region* to be $> 600 \cap \leq 1100$ GeV and the *high mass region* to be ≥ 1100 GeV. Using a NN with three hidden layers I get poor results, but changing this into one works! Will repeat with real weights, it didn't work. Will try DSID now...

5.2 Neural Network Preparation

For this thesis we purely utilize `TensorFlow v. 2.7.1 GPU` for NN. Before implementing everything into a real NN we need to first prepare the data in a special way when. The first and most important thing for this whole project is that the `batch_size` should be as big as possible whenever we try **anything** when using the dataset. This is because of both the size of the dataset and because of the imbalance between signal and background.

The highest possible batch size that could be used for this thesis was 2^{24} which means that there are roughly 17 million samples pr. batch. This is the best that a dedicated GPU, `NVIDIA A100-PCIE-40GB`, could handle. The batch size also decreases the more complex the NN becomes, as this takes greater computational power.

With this out of the way there are still a few things that needs to be done to get the best NN for our purposes. These have their own section as they are more difficult to tackle than the batch size.

5.2.1 Padding of data

There are two problems that need to be adressed when utilizing NNs as compared to BDTs. The first one which is the hardest one to solve, as no one has found a reliabe solution yet is the padding of the missing values. Padding is the process of filling missing values on a dataset that goes into a network. This is a problem that doesn't usually appear in other fields than physics, especially with the type of data we look at in HEP. As an example, in the dataset being used in this thesis we include two six kinematic variables that might not even be there, these are the p_T, η and ϕ of the two highest p_T jets in an event. The obvious reason for this is that there might not be two jets in every event, thus we have missing variables. As previously mentioned, the way that the dataset was made made sure that there where two jets before recording the actual values of the variables on the dataset. If there weren't at least two jets the p_T was set to zero while η and ϕ were set to 10.

remember to do this

The jet p_T being 0 is a valid form of padding the dataset, as this doesn't break any fundemental law of physics. However setting ϕ as something outside of $[-\pi, \pi]$ doesn't make much sense as this is the angle around the detector. Setting a high value of η might be physically possible (in the future) but as of today the ATLAS detector has a $|\eta| < 4.9$ as the pseudorapidity states how close to the beamline the event is recorded. However having a p_T of a jet equal to zero and still recording the η and ϕ breaks the laws of physics, so this is a problem that needs to be fixed.

this is from 2010, new source needed

As mentioned before, there is no general consensus of how the padding should be done, and there are many different methods of doing so. The classical data scientist way of solving this problem would be to just take the mean of every feature and use that as a variable for every event with missing values. That means replacing every $p_T = 0$ and $\eta, \phi = 10$ with the mean of every p_T, η and ϕ (excluding those values). However this is not popular among physicists since it breaks conservation laws when we say there are jets when there really isn't. Another approach is to use Bayesian statistics or ML to estimate the missing values, these options will not be pursued in this thesis due to time. Another approach, perhaps a naive one, is setting all the missing values to zero, as this might mean that there isn't anything there, but this also breaks conservation laws since $\eta, \phi = 0$ have physical meaning, this is also highly looked down upon by data scientists since this would affect the weighting when training the network. Another approach is to just remove all events that have a missing value completely getting rid of the problem, but this reduces the statistics of the dataset which is not desired when searching for new physis, as this might remove precious signal events. One could also just remove the features with missing values to conserve statistics, albeit make it harder for the network to see any pattern that we might miss. ***For this project it is still up to discussion what approach should be used, and so far only the latest method has been applied.***

5.2.2 Normalization of data

Moving onto the second problem, which is not as problematic as the previous, we have the Normalization of data. Since neural networks send a lot of data into multiple neurons and mulitple layers using activation functions and other mathemathical tools, then it is important to make sure that the signal doesn't die when moving around the network. A fast way for the signal to die off is to not normalize the data and send it through the network, the reason it might die is because we send in numbers which vary significantly to eachother, i.e. the p_T might be as high as thousands GeV, while E_T^{miss}/σ might be as low as 0.1. What might happen when sending such different numbers is that the network might think "obviouly the high number is more important than the low number" thus making the activation function worse for the feature, even though this feature is of high imporatance when looking at MET final states. A way to fix this problem is to normalize all features. There are many ways to do this, one could do *min max scaling* which normalizes every feature from $[0, 1]$, completely erasing the problem above. Mathematically speaking this is done by

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{5.1}$$

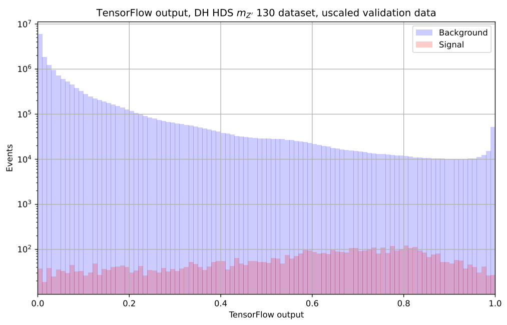
Where X is the array containing all features, while X_{min} and X_{max} is the lowest and highest value in said array. Another way to normalize the data is to make the mean of the data 0 and the standard deviation to one, this is called *Z-score normalization* is it?

$$X_{norm} = \frac{X - \bar{X}}{\sqrt{\sigma_X^2}} \tag{5.2}$$

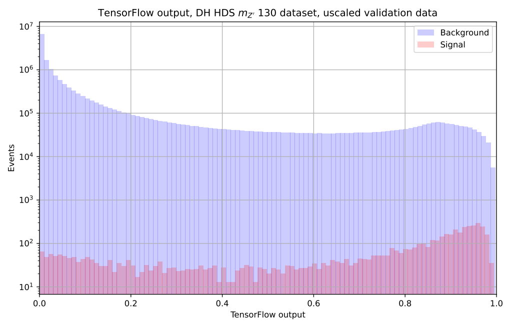
where $\bar{X}$ is the mean of said array and σ_X^2 is the variance. One could also use pre-built functions in TensorFlow that do this normalization for you, one is called `Batch_normalization` which normalizes the data that enters the network pr. batch, this is usually used in Convolutional NNs as it improves computational speed. And another one is simply `Normalize` which does the same a Eq. (5.2) for the whole training set going in, this is however very slow to use. There is also `Layer_normalization`, where you normalize the activations of the previous layer in a batch *independently*, rather than *across* a batch like `Batch_Normalization`. But of these methods are also based on Z-score normaliazation.

There is a big difference when normalizing data ourselves and using TensorFlow, and that is that TensorFlow will remember how the data was normalized when training and will normalize the testing data in the same way (using the same variables). However we by not using TensorFlow to normalize we could normalize the whole dataset before splitting, which might make for more a more model independent thoughts? normalization than doing it in bacthes that might not include all models. These different normalization methods have been tested and can be seen in the Figure 5.1. We can see that the best results come from Z-score and Batch normalization. We can see how the significance differs on both cases in Figure 5.2.

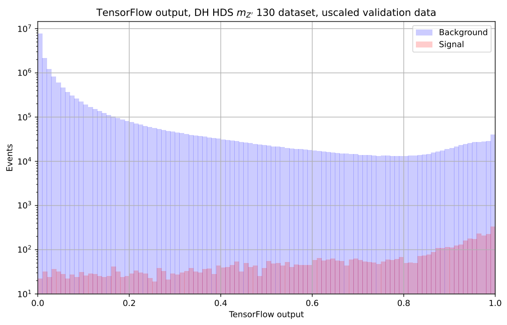
From these results it is clear that the best normalization method is Batch normalization, but is it reasonable to use this method when one is not using a CNN? ***For the following examples I have used the Z-score method, but I might change to use Batch normaliazation in the future.***



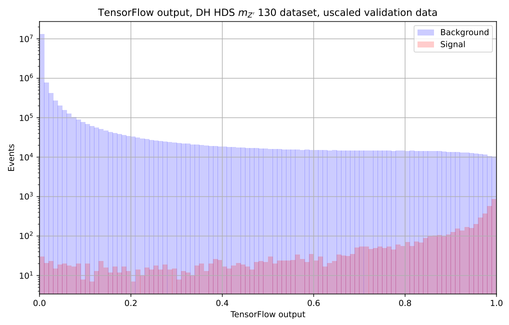
(a) No normalization



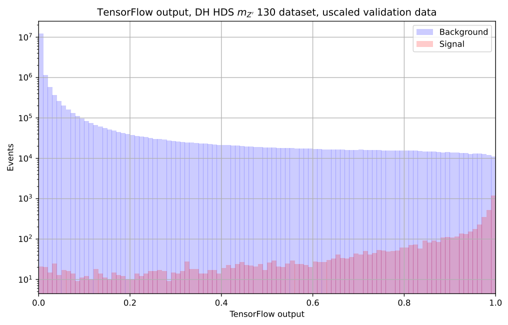
(b) Layer normalization



(c) Min max scaling

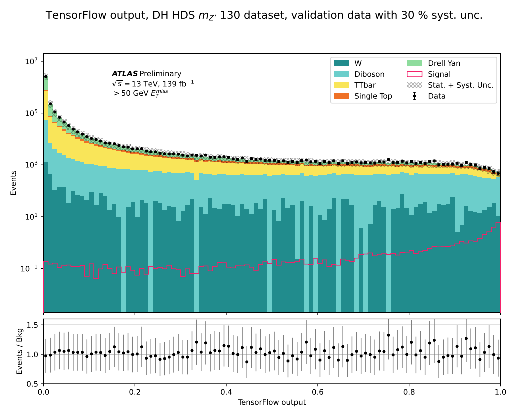


(d) Z-score

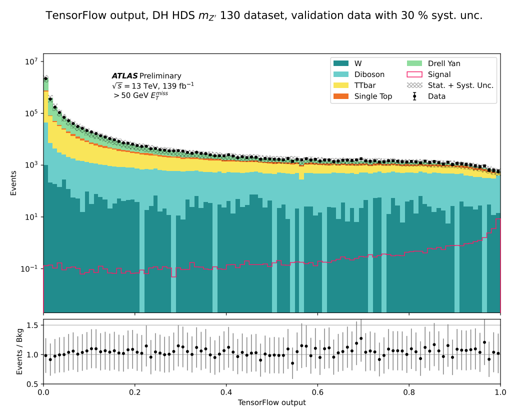


(e) Batch normalization

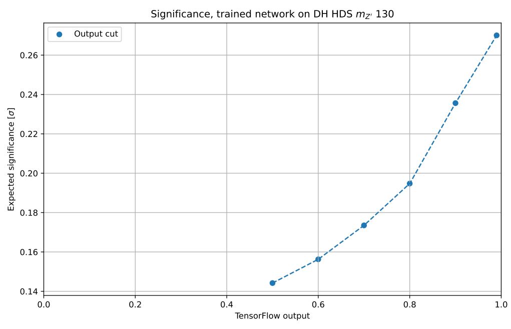
Figure 5.1: NN prediction when using different normalization methods. This is testing a dataset with a Z' DM model.



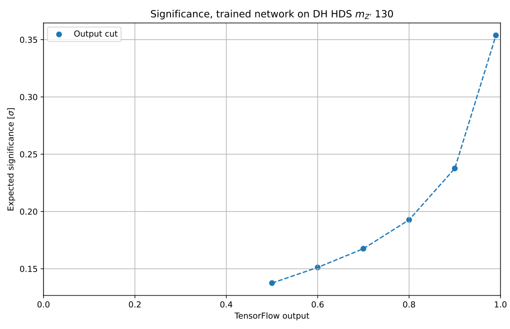
(a) Z-score



(b) Batch normalization



(c) The expected significance of a)

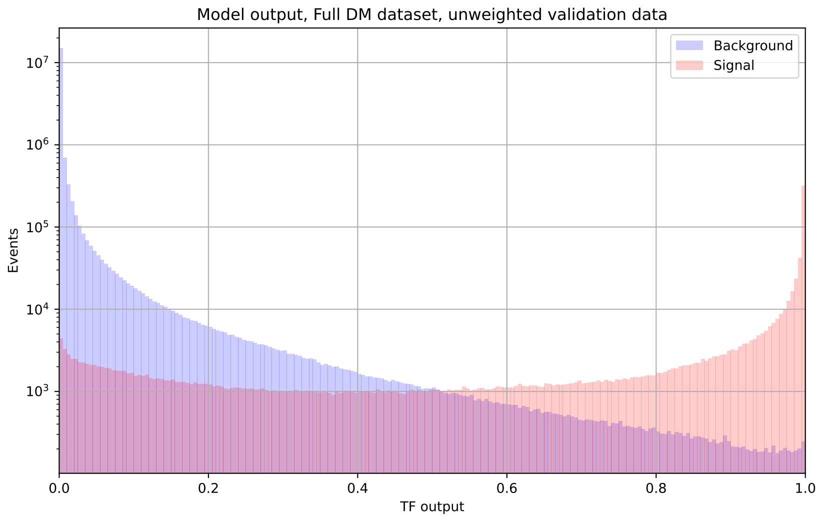


(d) The expected significance of b)

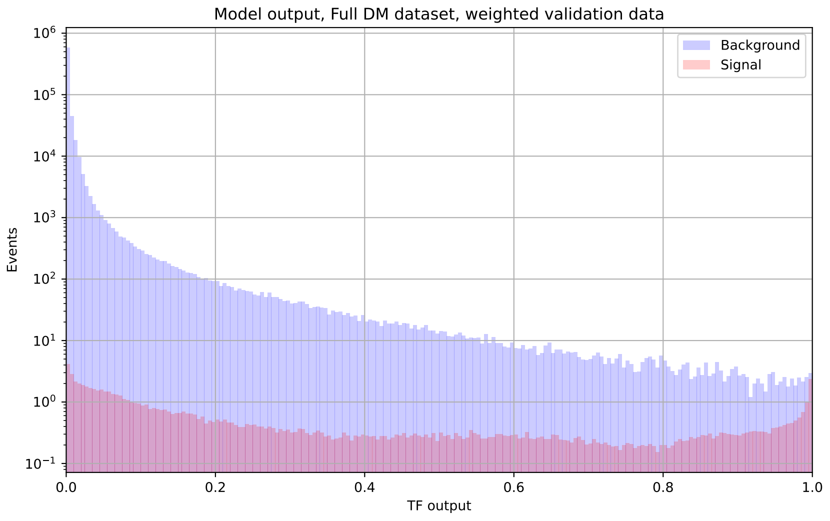
Figure 5.2: Comparison of the best normalization methods. Figure a) and b) show the validation data of both cases, c) and d) show the expected significance of the validation plots when making a cut on the output.

5.2.3 Weights

The results so far (my *OLD* interpretation with only three different DM theories) show that the network is completely capable of distinguishing signal from background in the unweighted training, however if we were to scale the predictions (the networks output) with the weights (sample weights * luminosity / sum of weights) we would see that the signal is hidden underneath the background. Both validation plots are shown in Figure 5.3. If we train the network using the weights of the MC samples we see that the network still gets the same accuracy, and an even lower loss function score. But the AUC becomes exactly 0.5. When looking at the ROC curve and validation plots (plots showing how well the network sorts signal from background) we see that everything "is messed up" the ROC is 0.5, and there is only **one** background bin. When printing how many signal events the weighted network predicted it is easy to understand why, it is because it says there is not a single dark matter event. Which I first interpreted as being in agreement with Figure 5.3b.



(a) Unscaled validation plot.



(b) Scaled validation plot using MC weights.

Figure 5.3: Validation plot of uwweighted network on the first version of the FULL DM dataset.

The italics were my thoughts at the time

From my understanding it is preferable to train the network with weights, because then we can use it to predict data events, which have physical significance compared to MC events. This will be specially useful *in my head when we predict how many dark matter events there are using real data, as this cannot be "scaled up" at a later point.*

As to which weights one shall use I am unsure, I like the physical weights for the reason above. But I know that for uneven datasets one could "weight down" the background events such that it "looks to be a 50-50 ratio" between background in signal. This sounds like making a bias in my head, and I would not know how to interpret the network predictions (if it predicts samples or events), but this appears to be a "standard" technique used in data science.

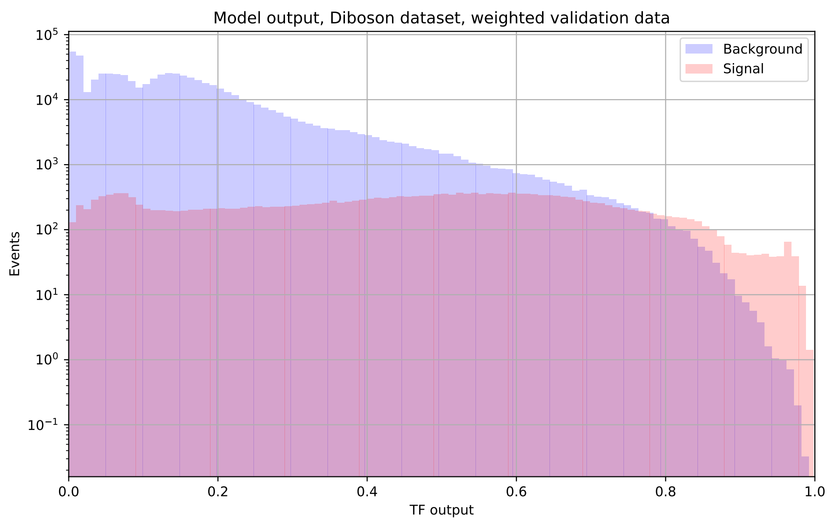
T*o check whether the network is working as intended I will now try to sort out "Diboson" events from other SM backgrounds, since we know this is real. If it works as intended I am unsure as to where to go next.* One thing worth noting is that the weights for the dark matter models being used may not necessarily be correct, as we don't have any empirical proof for the variables being used when calculating the weights (i.e. the cross section).

It seems that my interpretation was wrong, the weighted network for the Diboson search also predicted "0 Diboson events" (meaning my interpretation of the network predicting number of events is wrong), something we can empirically say is wrong (i.e. literally every figure of the kinematic variables). The scaled validation plot of the unweighted signal still "shows" that the network struggles a bit to see the difference between signal and background, overlapping almost all the way to 0.8. However it is still capable of sorting it out. Now I will try the "data analysts" way to weight the samples. That means weighting all background samples by $\frac{N_{sig}}{N_{bkg}}$, as is a common practice of weighting unbalanced datasets in data analysis.

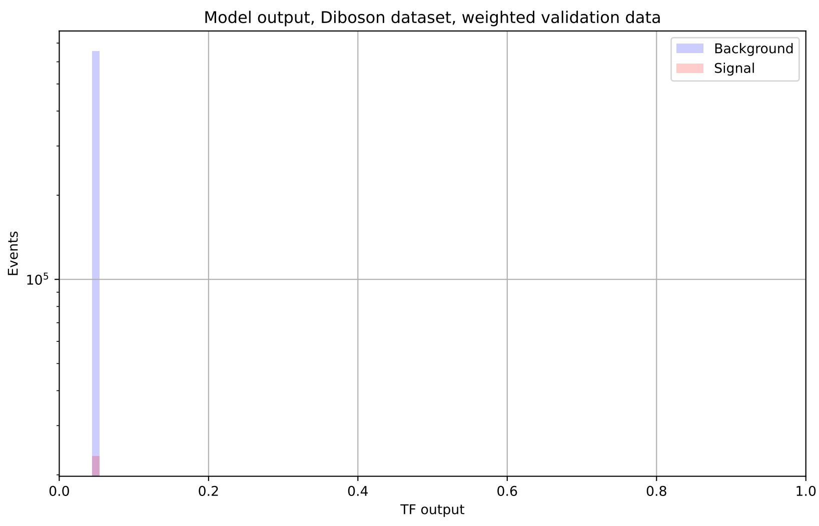
The new weighting works! This is the method that will be used in the further studies using NNs. The *Validation plots* showing the results of the network when trying to sort out the "Diboson" channel from the other are shown in Figure 5.4 and the ROC scores in Figure 5.5. A table showing how unbalanced the data is is showcased in Table 5.1.

	Number of events	Sum of weights	Events $\times$ SOW $[10^{11}]$
Signal	8,813,716	93,304.9	8.2
Background	61,201,010	2,621,498.9	1,604.4

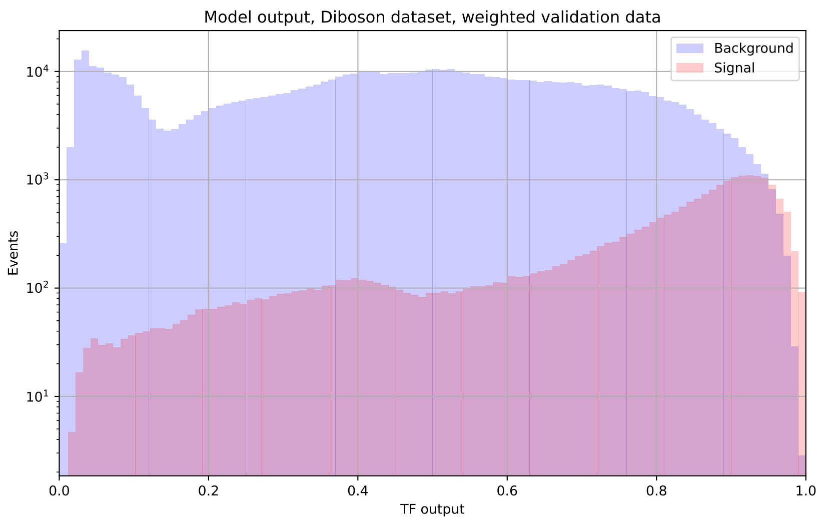
Table 5.1: Table Showcasing how uneven the training dataset is between signal and background. This is on the Diboson dataset which incorporates all the SM MC samples



(a) Validation plot for the unweighted training.

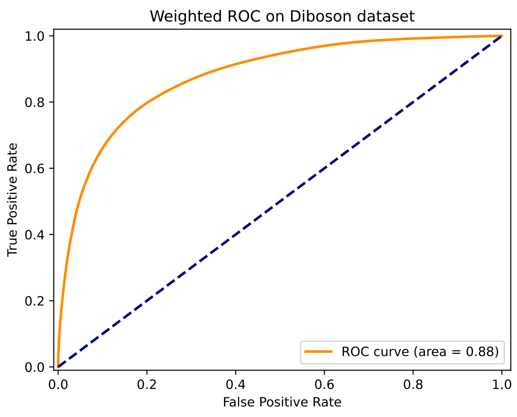


(b) Validation plot for the weighted training using MC weights.

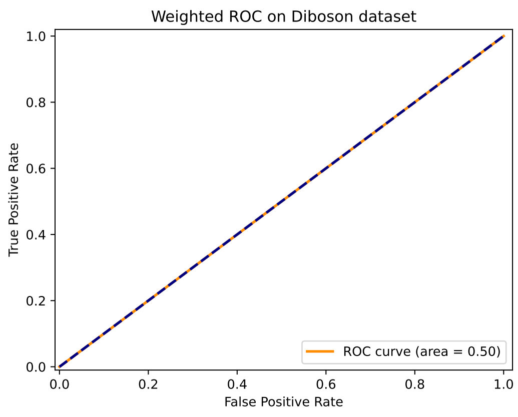


(c) Validation plot for the weighted training using $\frac{N_{sig}}{N_{bkg}}$ as weights on the background.

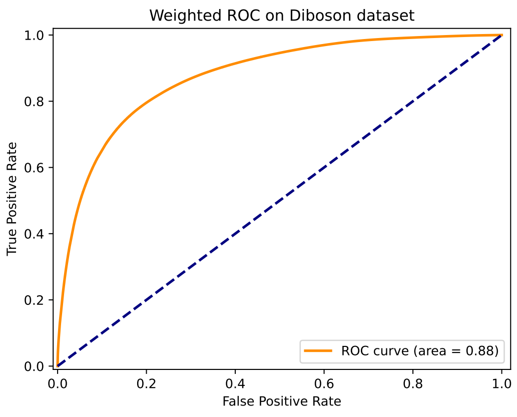
Figure 5.4: Result of the different network training weighting.



(a) ROC score for the unweighted training.



(b) ROC score for the weighted training using MC weights.

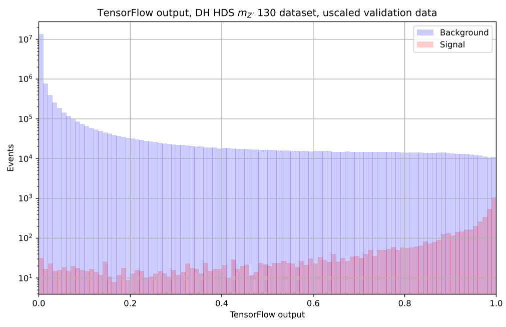


(c) ROC score for the weighted training using $\frac{N_{sig}}{N_{bkg}}$ as weights on the background.

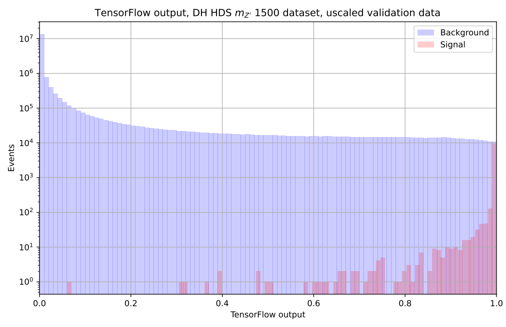
Figure 5.5: Result of the different network training weighting.

5.2.4 Balanced weights

Even if the weighting method previously described helps the network to give reasonable results. It most likely won't distinguish the importance between signal events that have high resonance and those that do not. As an example, if we look at the Z' DH model in the HDS and compare how the network classifies a model with a $m_{Z'}$ of 130 GeV to one with a $m_{Z'}$ of 1500 GeV we might get an idea of how the network classifies things. We can see how the network predicts wether an event is signal or background using validation plots, this is seen in 5.6.



(a) NN prediction of DH HDS $m_{Z'} = 130$ GeV

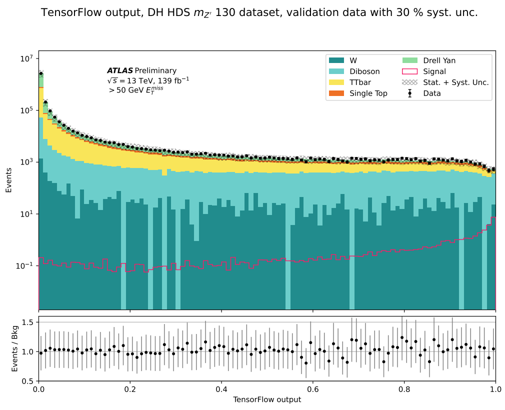


(b) NN prediction of DH HDS $m_{Z'} = 1500$ GeV

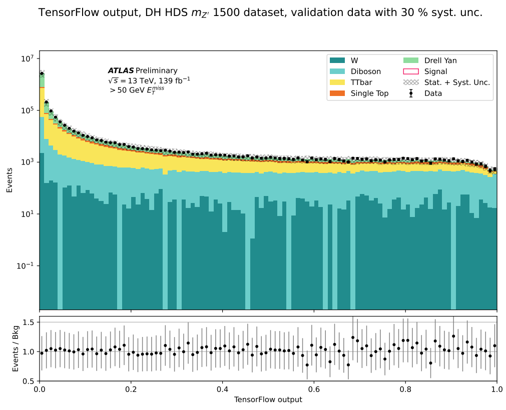
Figure 5.6: NN comparison of trained models in FULL Z' dataset

The result purely demonstrate the raw network output, meaning that the validation dataset has not been scaled up using MC weights (i.e. model cross section) or luminosity.

We can see the network is better at classifying the events from the model with $mZ' = 1500$ GeV than the other, even if this model has a much lower cross section, as seen in the correctly scaled plots below.



(a) NN prediction of DH HDS $m_{Z'} = 130$ GeV



(b) NN prediction of DH HDS $m_{Z'} = 1500$ GeV

Figure 5.7: Correctly scaled plots.

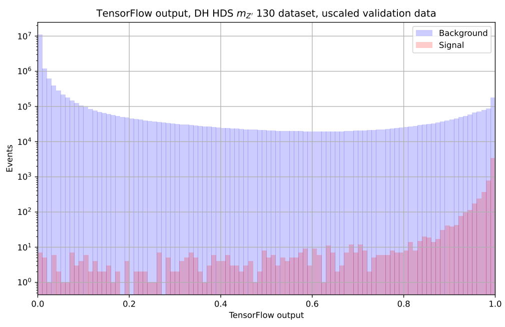
As we can clearly see, the better predicted model by the network does not even have one event when scaled up correctly. Therefore it is desirable to both take into account the data imbalance between the signal and background as well as the MC weights when training a network. To do this using TensorFlow we could make use of two parameters when training the network: `class_weight` and `sample_weight`.

`class_weight` works as a dictionary that weights events that are keys on the dictionary. For our purposes we can make a dictionary where we weight signal and background events differently, this is the same type of scaling that was done in the previous section. `sample_weight` takes in individual weights for every single event that goes into the netwrok, meaning that it is crucial that we know that the desired weight matches the desired event. Ideally we would use both weighting methods, `class_weight` to balance the signal to background ratio and `sample_weight` with the MC weights. However there is a bug in TensorFlow (up to version `GPU 2.7.1`) that makes it so the program doesn't run when using both parameters. This is not a big problem though, as when looking at the source code one can see that what TensorFlow does with both weights is multiply them togheter. Thus we could try to make use of this "balaned weighting" method to see if the network learns the different models better.

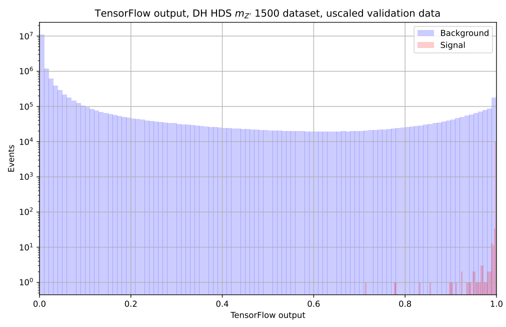
We can see in Table 5.2 how uneven the data (FULL Z') is. To balance the data we have four options. We can either *weigh up the signal* events by the ratio of background over signal. Or *weigh down the bacgkround* with the ratio of signal over background. However this ratio might differ a lot when using the MC raw event ratio or the SOW events ratio. I have tested all four possibilities and these can be see below.

	Actual events	MC events
Background	2,715,280.4	69,664,290
Signal	388.4	2,991,598

Table 5.2: Table showcasing the data imbalance between background and signal in both raw MC events and actual events. By actual events it is meant weighted events. This is for the training dataset on the FULL Z' dataset.

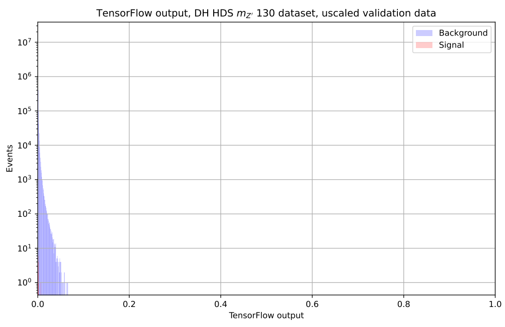


(a) NN prediction of DH HDS $m_{Z'} = 130$ GeV

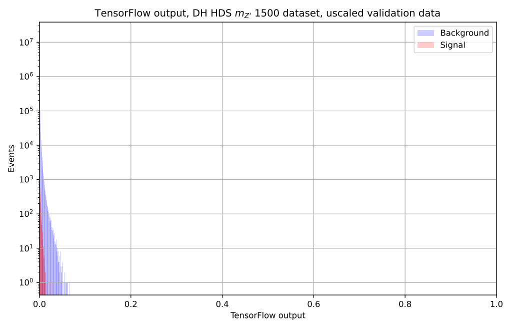


(b) NN prediction of DH HDS $m_{Z'} = 1500$ GeV

Figure 5.8: NN prediction when weighting the signal with the SOW ratio.

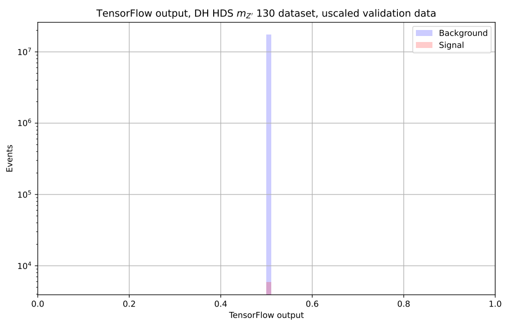


(a) NN prediction of DH HDS $m_{Z'} = 130$ GeV

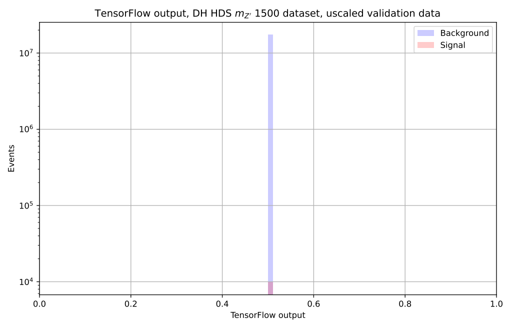


(b) NN prediction of DH HDS $m_{Z'} = 1500$ GeV

Figure 5.9: NN prediction when weighting the signal with the raw event ratio.

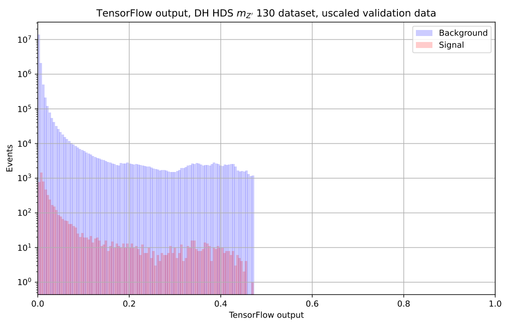


(a) NN prediction of DH HDS $m_{Z'} = 130$ GeV

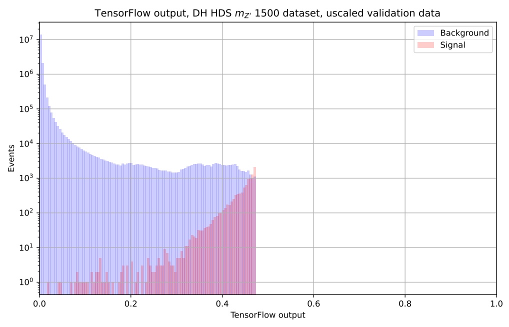


(b) NN prediction of DH HDS $m_{Z'} = 1500$ GeV

Figure 5.10: NN prediction when weighting the background with the SOW ratio.



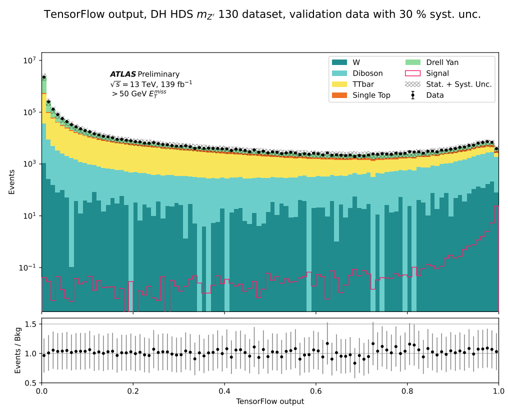
(a) NN prediction of DH HDS $m_{Z'} = 130$ GeV



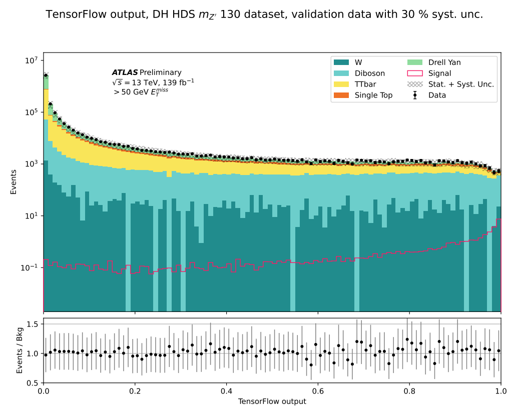
(b) NN prediction of DH HDS $m_{Z'} = 1500$ GeV

Figure 5.11: NN prediction when weighting the bacgkround with the raw event ratio.

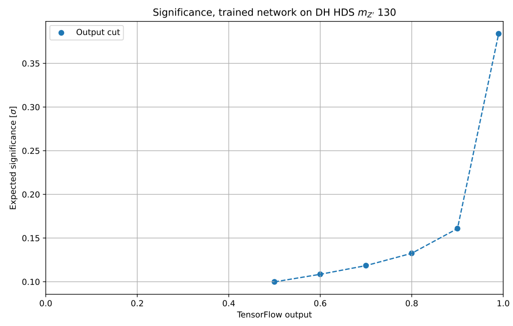
As we can see almost every way of doing this balanced weighting gives poor results. The only one that seems to work is when we we scale up the MC weights with the ratio between the SOW of the bacgkround over signal, however it does not seem to learn the model with lower mass better compared to how it learned the model with higher mass. It seems to have generally learned both models better, which might be a hint that it learns other models worse, i.e. EFT models with much lower cross section. If we now compare the correctly scaled validation plots of DH HDS $m_{Z'} = 130$ GeV.



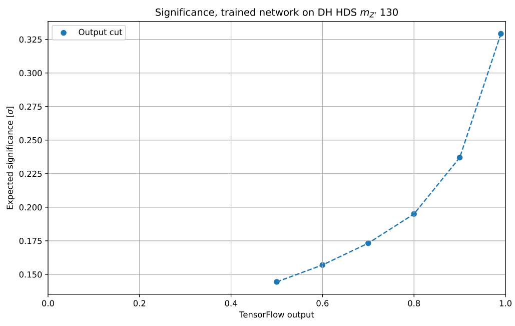
(a) MC weights + scaling signal up with SOW ratio



(b) Only using raw MC events weigh down bacgkround



(c) The expected significance of a)



(d) The expected significance of b)

Figure 5.12: Comparison of the best balanced weighting method to the weighting method of the previous section. Figure a) and b) show the validation data of both cases, c) and d) show the expected significance of the validation plots when making a cut on the output.

From this we can see that the the network that trained when balancing the data is better at classifying the DH HDS $m_{Z'} = 130$ GeV model correctly as signal than the other network. We also observe that it wrongly classifies more background as signal than the other network, but even if this is the case the expected significance is greater in the new network. The reason that the new network wrongly classifies more background as signal might be becuse I utilized the same hyperparameters when training both newtworks, when it might be better to use different hyperparameters on each. Another thing that might be of significance is that I used the ratio of the training set, it might make a difference (better or worse) to use the ratio of the full dataset when training as this is more general. Since the ratio most likely differs for the training and testing set. ***Both of this options are up to discussion to further pursue. As well as if this appreoach is worth pursuing if it only learns the models with higher cross section.***

5.2.5 Grid Search

To get the best performance on our NN, we need to find which hyperparemters helps the network reach highest significance. To do this, we need to do a gridsearch. For our neural network we will mainly focus on four hyperparameters explained on section 3.1:

- The learning rate η
- The L2-regressor variable λ
- The number of neurons on each hidden layer `n_neuron`
- Possibly the number of layers `n_layers`, exluding the output. (NB! Meaning that `n_layers` $= 1$ means no hidden layer!)

The metrics that will be used to estimate the best hyperparameters are **AUC**, **binary accuracy** and most importantly **expected significance**. The expected significance for this section has been calculated using the low statistics formula Eq. (5.3) just in case there is too few events after the network prediction. The expected significance will also be calculculated when making a cut on 0.85 on the netowrk prediction, meaning only looking at events which the network rates as signal with 85% confidence and above.

not 100, so we have many back-ground events!

The dataset on which I've trained so far is the FULL Z' DM dataset including DH, LV and EFT. The raw number of events on this dataset is roughly 3 million signal events to 70 million background events, I have however split this into a 80% training set and 20% testing set. ***The reason for doing this, is so we get the best hyperparameters for doing a model independant search.***

The full results of the gridsearch when setting `n_layers` $= 2$ (one hidden layer) and $\eta \in [0.001, 0.01, 0.1, 1]$, $\lambda \in [10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}]$ and `n_neuron`$\in [1, 10, 50, 100]$ can be found in my GitHub under `Plots/NeuralNetwork/FULL/GRID_lamda_eta_neurons`, but for the sake of this thesis not being too long I will only show the significance plot as well as the AUC for the testing and training set when setting $\lambda = 10^{-5}$. The significance is seen in Figure 5.13, while the AUC is in Figure 5.14.

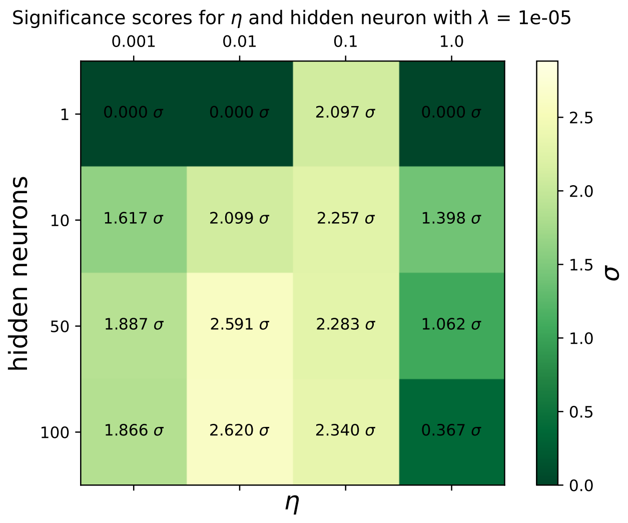
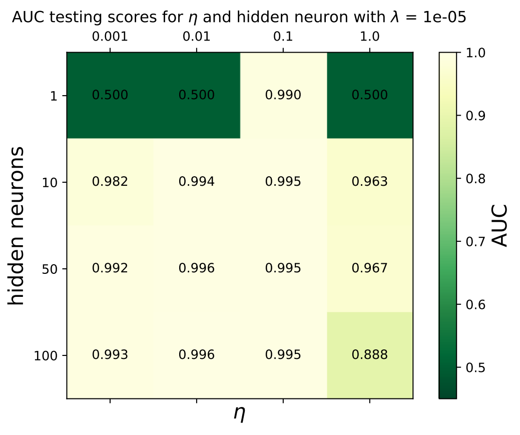
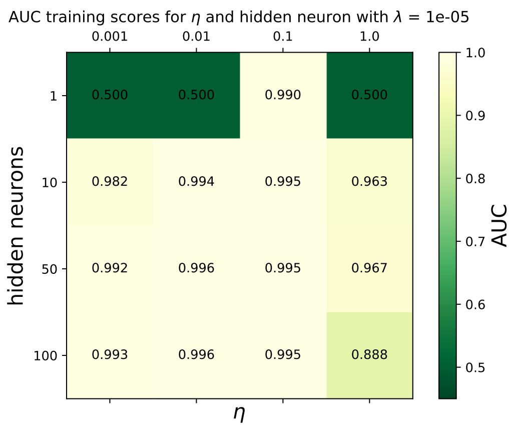


Figure 5.13: Grid search significance with $\lambda = 10^{-5}$ and n_layers $= 2$



(a) Testing AUC



(b) Training AUC

Figure 5.14: Grid search AUC with $\lambda = 10^{-5}$ and n_layers = 2

Doing the same but with more hidden layers and setting $\lambda = 10^{-5}$ we get the results shown in GitHub under `Plots/NeuralNetwork/FULL/GRID_layers_eta_neurons`, for the sake of this thesis not being too long I will again only show the significance plot as well as the AUC for the testing and training set but this time when setting $\eta = 0.01$. The significance is seen in Figure 5.15, while the AUC is in Figure 5.16.

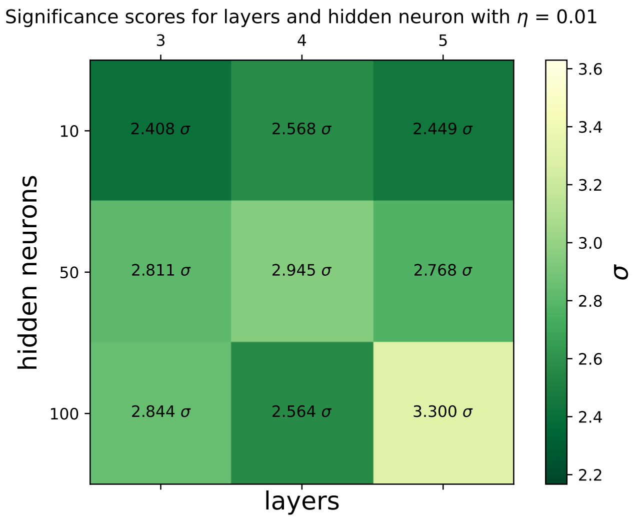
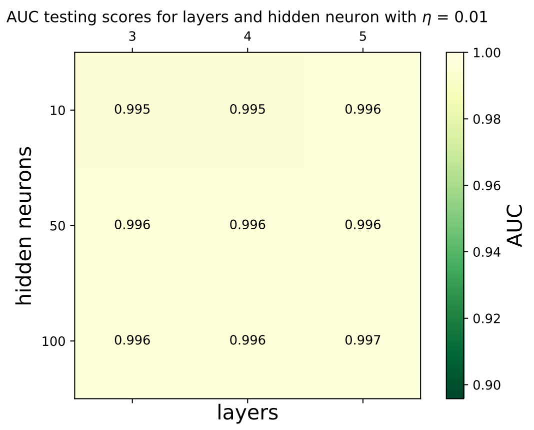
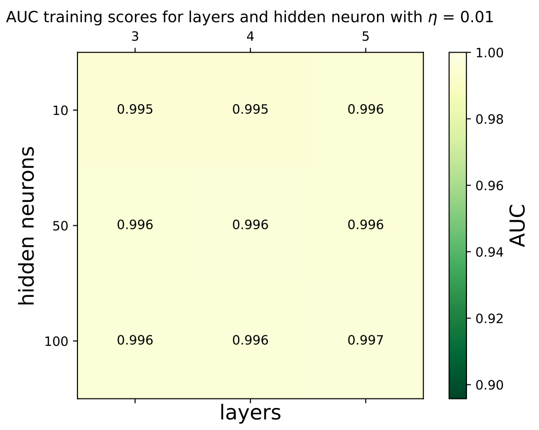


Figure 5.15: Grid search significance with $\lambda = 10^{-5}$ and and $\eta = 0.01$



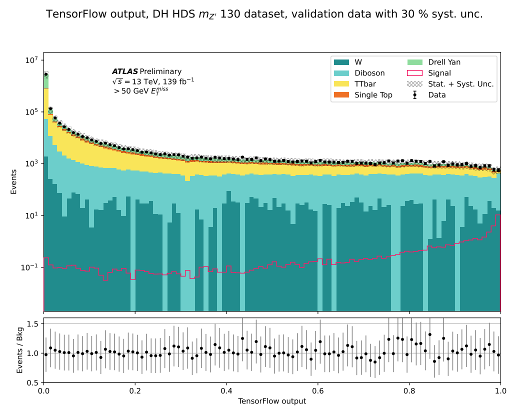
(a) Testing AUC



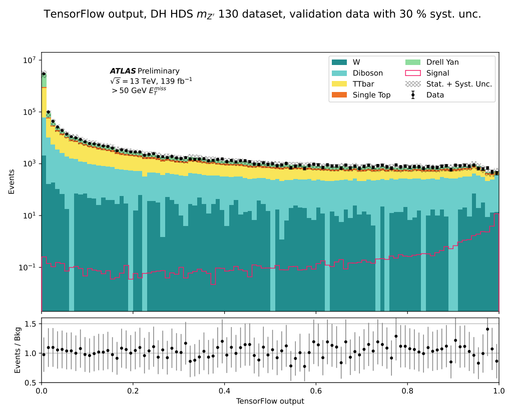
(b) Training AUC

Figure 5.16: Grid search significance with $\lambda = 10^{-5}$ and $\eta = 0.01$

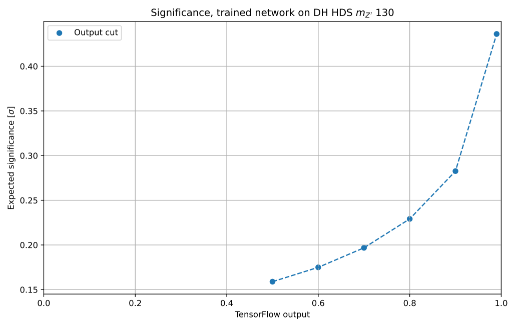
I also made a test network with the same hyperparameters as the best one in Figure 5.15, the only difference being that it has 10 hidden layers. I plotted the validation data to see how different the networks were at predicting the DH HDS $m_{Z'} = 130$ GeV model. These were trained and tested when using the Z-score normalization, Eq. (5.2), and the weighting method explained in section 5.2.3. The results are shown in the figure below.



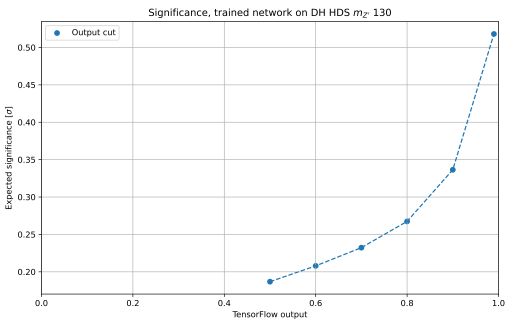
(a) Four hidden layers



(b) Ten hidden layers



(c) The expected significance of a)



(d) The expected significance of b)

Figure 5.17: Comparison of the network performance when having four and ten hidden layers. Figure a) and b) show the validation data of both cases, c) and d) show the expected significance of the validation plots when making a cut on the output.

This hints that we could be able to make a DNN with more hidden layers and get better results. However there are a few things that need to be noted when doing this, aside from the padding which is an even greater problem. The first and smallest one is that we could have used batch normalization instead of Z-score, but this is again something to be further discussed.

The second being that I have not used the "balanced weighting" method when training the network, which might be for better or worse if the network really does ignore all EFT models...

The third one which is more technical is that since more complex networks require more computational power, then this leads to us decreasing the batch size. Which lowers the statistics of signal, and might even lead to the network training on batches without any signal sample at all. So the trade off is also something to be discussed.

The last thing to be noted is that having a DNN completely removes the possibility of combining the results of multiple networks trained on a single model, as the imbalance becomes too much for the network to see anything. A solution to this however, is that instead of combining the results of multiple networks trained on a singular model, we could try the Parametrized NN approach, which could potentially avoid the imbalance problem. This is obviously something we have to discuss further.

5.3 XGBoost Training

For XGBoost there is a different problem when it comes to weights. XGBoost has a variable called `scale_pos_weight` where we can help the network deal with unbalanced data, such as the one we have. Thus we can use the *real* weights that are calculated in the MC generators, except not really, XGBoost does not have to possibility to include negative weights. In this project I have therefore used the absolute value of the weights when training. Other than that there are few complications.

5.4 Pure log

Some problems that have happened is that I had previously made a Deep Neural Network, using three hidden layers. This is however not optimal as the DM statistics is non existent compared to the SM background. Another big problem I had was that I used small bacth sizes when training the Neural Networks. A batch being a portion of the data which is used when training. The reason we batch is to reduce computational power and divide the task of learning. I had a batch size of around 30,000 MC events pr batch. To remind ourselves of our data size, we have around 90,000,000 MC events, from which roughly 40,000 corresponds to a single DM MC DSID. So when batching we trained the network to recognize DM, when there most likely wasn't a single DM sample in the batch itself... this explain the abrupt end, and peak of backgrounds on the signal region seen in Fig. 5.19b. Thankfully, since I have the supercomputer `hepp03` available I could increase the batch size to a massive amount such as 2^24 which gives roughly 16 million MC events pr bacth. This is the highest the GPU of the supercomputer can handle!

With this fixed there was still a big problem, the expected significance of both NN's and XGB is at best half of what a very rough cut and count gives. As mentioned in section 5.1.2, we can split the data in different forms. While splitting it pr DSID works with one hidden layer now it doesn't make much physical sense to do it the way I'm currently doing it when looking at the Z' DM models, since the DM MC samples are splitted into ee and $\mu\mu$ final states. However if we train the network on either final state we are removing the model independent part of the plan (although this increases the significance!). The plan fowards so far is to combine the final states of the Z' DM models as long as they only differ in final lepton state. And compare the significance of these to a statistically combined significance of cut and count.

One last thing to add as to why the significance might be so much lower for our networks is that I have not yet done a grid search for the best hyperparameters and loss functions. Doing this with XGBoost is easy, but there is a memory leak in the codebase of TensorFlow that makes the process tedious... This is where I am at in the present time, as well as waiting for more DM data.

5.5 Comparison to cut and count

Testing three models using the classical data analysis way we apply cuts to kinematic variables and try to isolate the signal from the background to then calculated the expected significance. The three models I chose to test are all High Dark Sector models with $m_{Z'} = 130$GeV. They are a Light Vector (LV), Dark Higgs (DH) and Effective Field Theory (EFT) models. The cuts I made on these are shown in Table 5.3.

	Cut
E_T^{miss}/σ	> 10
m_T	> 160 GeV
m_{ll}	> 120 GeV
Number of B-jets	0
m_{T2}	> 110 GeV

Table 5.3: Table showcasing the cuts used when doing the cut and count method.

Since the cross section to find Dark Matter is really small we have to use the low-statistics expected significance formula to find the closest to correct significance. The formula is

$$Z = \sqrt{2\left[(s+b)\ln\left(1+\frac{s}{b}\right)-s\right]} \tag{5.3}$$

Where s is the number of signal events and b is the number of background events. Using this we get the results shown in Table 5.4 for the electron channel and Table 5.5 for the muon channel. Also included on the tables are the number of events. One thing worth mentioning is that when adding another cut on the maximum invariant mass increases the significance. The significance for LV on the electron channel was at 1.2σ when adding a cut stating that $m_{ll} < 150$ GeV. This makes sense since the models in question all have a $m_{Z'} = 130$GeV. This cut was not added since we do not want to put a cap on the mass of the propagator, as we don't know what the real mass is.

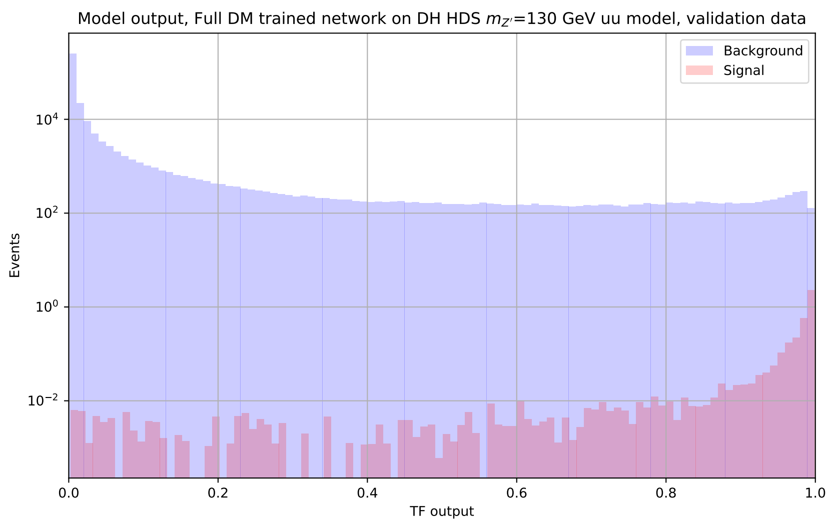
	LV	DH	EFT	Background
Events before cuts	15	20	0	1,256,624
Events after cuts	4	6	0	117
Expected significance [σ]	0.4	0.6	0	

Table 5.4: Table showcasing the result of the cut and count method for the electron channel.

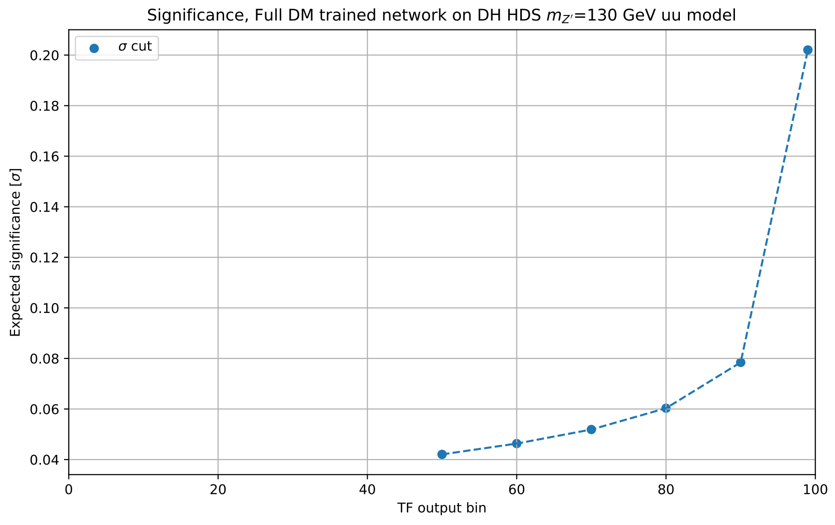
	LV	DH	EFT	Background
Events before cuts	14	19	0	1,626,098
Events after cuts	3	5	0	108
Expected significance [σ]	0.36	0.51	0	

Table 5.5: Table showcasing the result of the cut and count method for the muon channel.

If we were to compare these results with what our NN and BDT that trained on the full dataset we see that we can calculate the expected significance in different locations for the validation plots. Testing on the networks that trained using the data scientist method on the full DM dataset we get the results shown in Figure 5.18 for XGBoost and Figure 5.19 for the Neural Network.

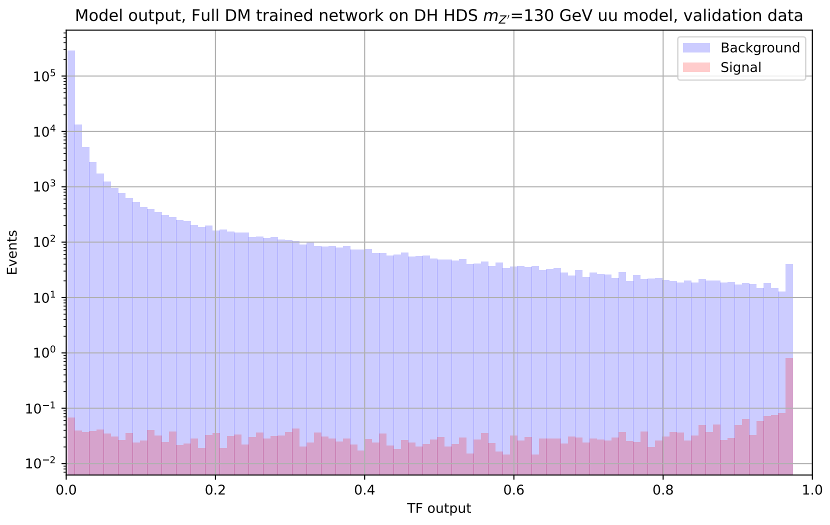


(a) Validation plot.

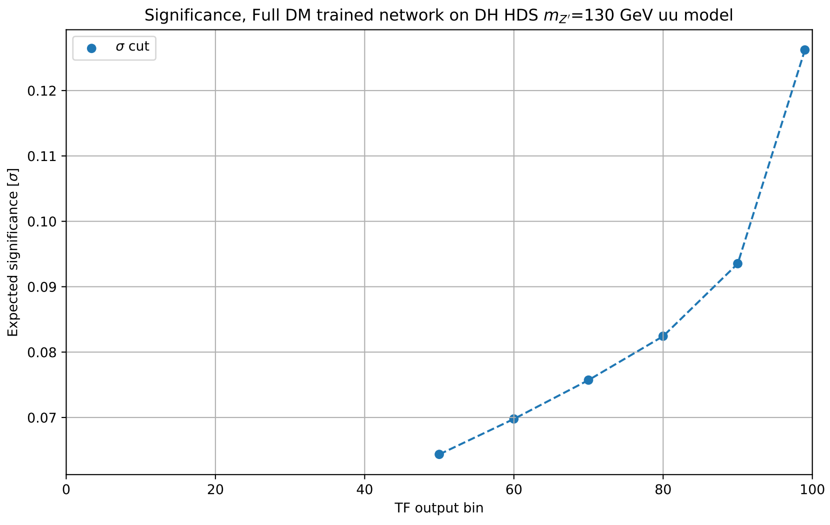


(b) Expected significance when looking at bins and forth.

Figure 5.18: Expected significance of XGBoost when trained on the Full DM dataset for the DH HDS $m_{Z'} = 130$ GeV muon model.



(a) Validation plot.



(b) Expected significance when looking at bins and forth.

Figure 5.19: Expected significance of the Neural Network when trained on the Full DM dataset for the DH HDS $m_{Z'} = 130$ GeV muon model.

As we can see the expected significance is lower using ML than a rough cut and count. My theory for why this is the case is because we are testing just ***one*** sample out of 154 different ones that are included for the three different theories I have acquired so far. And the ML networks shown above have both trained on a dataset including all 154 DM samples. The models that I tested might also not have been one of the "important" models the network learned from. Thus if I were to train the network individually based on the theory it might give better results.

Part III

Results

	Number of events	Sum of weights	Events $\times$ SOW $[10^{13}]$
Signal	2,991,543	36,327,943.99	1.08
Background	69,664,345	36,327,944.03	25.3

Table 5.6: Table Showcasing how uneven the training dataset is between signal and background. This is on the dataset which incorporates all the different DM MC samples

in spacetime.