UNIVERSITY
OF OSLO

Master's thesis

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

Ruben Guevara

Physics: Nuclear and Particle Physics
60 ECTS study points

Department of Physics
Faculty of Mathematics and Natural Sciences

Spring 2023

Ruben Guevara

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

Supervisors:
Professor Farid Ould-Saada
Dr. Eirik Gramstad

Acknowledgements

Thank you everybody<3<3

Abstract

Something something

Contents

This whole section is my reasoning for trying to re-weight the MC events to expected events. But is

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 Standard Model of Particle Physics

The standard model of particle physics is the combination of three gauge groups. The group explaining electromagnetism $U(1)$, the group describing the weak force $SU(2)_L$ and the group describing the strong force $SU(3)_C$. When combibing all these groups we get spontaneous symmmetry breaking resulting in the Brout-Englert-Higgs Mechanism. The whole lagrangian is of the form

$$U(1)_Y \otimes SU(2)_L \otimes SU(3)_C \Rightarrow$$

$$\mathcal{L}_{SM} = -\frac{1}{4}F_{\mu\nu}F^{\mu\nu} + i\overline{\Psi}\not{D}\Psi + \psi_i y_{ij}\psi_j\phi + h.c. + |D_\mu\phi|^2 - V(\phi) \tag{1.1}$$

where

$$V(\phi) = -\mu^2\phi^*\phi + \frac{\lambda}{2}(\phi^*\phi)^2$$

is the Higgs potential.

All of this is great at explaining what we know so far

1.2 Beyond Standard Model

1.2.1 Dark Matter

Chapter 2

Detection and Analysis

2.1 The ATLAS Detector

2.2 Classical Data Analysis

Chapter 3

Machine Learning

As mentioned there are other ways to try to isolate signal from background. The main approach of this thesis will be to use Machine Learning (ML) as it popular rise has proven to be effective at binary classification. In this thesis we will study two forms of ML to classify our new DM signal. We will use Neural Networks (NN) and Boosted Decision Trees (BDT). This project will take from granted that the reader is comfortable with linear algebra and jump straight into ML.

check if you DO mention it

cite papers

To give a a short description of the essence of ML we can start by 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 (also called loss) function $C(\boldsymbol{\beta})$ with respect to a set of data points given by a matrix $\mathbf{X}$. Since this project will only focus on Supervised Learning, meaning that we know what the output should be, we can use target values $\mathbf{t}$. Then we give the network depending on how close the predicted output is to $\boldsymbol{t}$. Then we repeat the process after tweaking the parameters $\boldsymbol{\beta}$ and see if the score gets better.

The general parameter $\boldsymbol{\beta}$ for our purposes can be seen as what are called *weights* and *biases*. The matrix $\boldsymbol{X}$ is a matrix with all the kinematic variables of every event. Lastly the target value $\boldsymbol{t}$ is for our binary classification task the label stating whether an event is signal or background.

Before getting into more details I will explain the theory behind both NNs and BDTs.

3.1 Neural Networks

As I already have written in great detail how NNs on a project I did as the curriculum for this masters, I will borrow most material from the theory section of the paper .

cite?

Before begining we can briefly explain the idea behind NNs. 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*

That takes us to what a *neuron* is.

3.1.1 Artificial neurons

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

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(u) \tag{3.1}$$

Where y, the output of the neuron, is the value of its *activation function*, which has the weighted (w_i or β_1) 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 our search for DM, 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 the high level features of the event, $\boldsymbol{X}$. Using Eq. (3.1) on every neuron on the next layer we can teach the network if there 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 on the target data $\boldsymbol{t}$ to tell the network whether it predicted correctly or wrong. We can then use a *cost function* and a specific *metric* to evaluate numerically how well the network predicted the output by giving it 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 gory details we can explore the mathematical model that illustrates what was tried to be explained above.

3.1.2 Stochastic Gradient Descent

The way we "tweak the parameters $\boldsymbol{\beta}$ to see if the network prediction gets better" is by using something called the *Stochastic Gradient Descent* (SDG). 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 this 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

$$\boldsymbol{\beta}_{i+1} = \boldsymbol{\beta}_i - \eta \nabla_\beta C(\boldsymbol{\beta}_i) \tag{3.2}$$

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, but also not too small to get stuck on a local minima rather than the global minima. Thus choosing the learning rate as a hyperparameter to be changed in a grid search is a good way to find the best one for our given task.

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 our purposes it is better to do everything as one single batch, but our data size is massive (of the order of 10^8) this simply becomes too comupationally heavy. Thus we have to instead of making a 20×10^8 matrix we could for example split it into ten smaller matrices of 20×10^7 to then perform a parameter update, making the computation possible. 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. (3.2) 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 for our purposes is that it will make the computation possible, however it is also helpful regarding computation time on other cases. Using more batches also reduces the risk of getting stuck in a local minima since it introduces a randomness of which part of the parameter space we move through, but this is at the cost of reducing statistics which might not be ideal for every problem.

3.1.3 Activation functions

As seen already, 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 I utilize a sigmoid activation function

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

which is the most basic activation function. I will 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

Cost functions have been mentioned a lot already, but what are they? 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 an 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. Since we are studing a binary classification task 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. This logistic function is the same as the sigmoid function in Eq. (3.3). Using this 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 start by looking at Eq. (3.1) 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 now introduce 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 thesis. 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.

drop this comment?

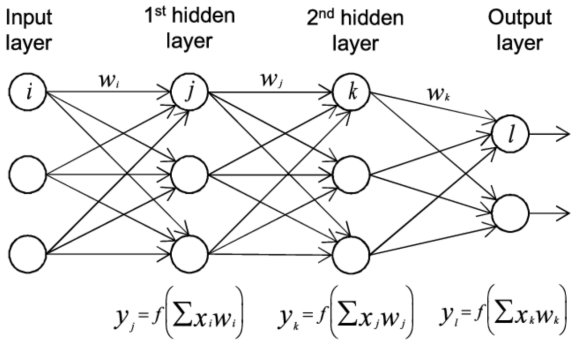


Figure 3.1: Basic illustration of a network with two hidden layers. Image taken from [1]

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 the changes in the biases and 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 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, 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

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

Furthermore, we 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 in 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. Another thing 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 as

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

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.

Something interesting 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 a layer n, after using many layers the NN 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 (which also varies 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. (3.2) 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

While the idea behind neural networks is to mimic the human brain, Boosted Decision Trees (BDTs) are different in the sense that they are not trying to mimic a brain, but rather split data sets. The idea of BDTs has been proposed as early as 2001 by Friedmann [2], where bla bla.

3.3 Sample weight

An extra term added to the derivative of the loss function, Eq. (3.14),

$$\delta^L \to \delta^L + C \sum_i \xi^i \tag{3.17}$$

where this C is a constant term we will add to each sample in the dataset. This extra term is called the *sample weight* and it is used to "even out" unbalanced data sets.

will add more details later!

3.4 The declaration of model indpedence

Part II

Implementation

Chapter 4

Data Analysis

Before making our ML networks we need to make the dataset that will be used to clasify signal from background. On this section we will discuss which kinematic cuts are used to define our signal region search. more fluff

4.1 Background Estimation

The backgrounds we will look at are W, TTbar, Drell Yan (Z + jets), Diboson, Signle top. Since we are doing a dilepton DM search, which we expect to behave similarly to a neutrino, we need to make a cut on the missing transverse energy (MET) and require the final state to have a lepton pair. As we are conductin a model independent search, we want to use minimal cuts.

The MET cut made in this thesis was chosen to be of 50 GeV, meaning we will *only* look at events where the MET is greater than this. This both because of the amount of events that are in a dilepton final state, and also because most models studied in this thesis have roughly more MET than 50 GeV.

The way we define our lepton pairs is not by just looking at Same Flavour Opposite Sign (SFOS) leptons ($e^{\pm}e^{\mp}, \mu^{\pm}\mu^{\mp}$), but also all other possible combinations as these might be important for theories such as SUSY. We will also look at the not so well defined by MC; Different Flavour Same Sign (DFSS), DFOS and SFSS lepton pairs.

Other than the standard criteria cuts, these are the only cuts that will be used in this search.

	Selection criteria
Dilepton final state	$\ell^{\pm}\ell^{\mp}$, $\ell^{\pm}\ell^{\pm}$, $\ell^{\pm}\ell'^{\mp}$ and $\ell^{\pm}\ell'^{\pm}$
Missing Transverse Energy	$E_T^{miss} > 50$ GeV

Table 4.1: Table showcasing the cuts used for this search.

4.2 Kinematic Variables

check if you write this as "variables for ML" or not

For this thesis there are many possible kinematic variables that can be used as features for our ML algorithms.

As we require the final state to only have two leptons it is therefore natural to look at the kinematics for both of these. The first thing we will look at is the transverse momentum, p_T, of each lepton. We will also look at the azimuthal angle, ϕ, to know where in the detector the leptons are located. In addition we will look at the pseudorapidity, η, to know how close to the beam the leptons are. Given that we have two leptons in the final state it is natural to look at the invariant mass, m_{ll}, of these.

There are other kinematic variables of interest that arent directly related to the lepton pair. The most important for this kind this search is the missing transverse energy, E_T^{miss}, as this how we expect DM to be recorded. Another variation of this that takes into account the uncerainty of the detector is of interest, this is called for MET-significance, E_T^{miss}/σ. We will also study the transverse mass, m_T, and tranverse energy, E_T, recorded in the events. In addition we will look at a variable called *hadronic activity*, H_T, which is the scalar sum of the all the jets (including leptonic jets) in an event. We will also look at the ratio between the missing transverse energy and hadronic activity.

write criteria for both

We willl also look at the number of b- and light jets. We will also look at a SUSY variable called the stransverse mass, m_{T2}, which as defined by Barr et.al. [3]. We will also look at the difference in azimuthal angle between: the lepton pair, $\Delta\Phi(l_1, l_2)$, the dilepton jet and MET jet, $\Delta\Phi(ll, E_T^{miss})$, the leading lepton and MET jet, $\Delta\Phi(l_l, E_T^{miss})$, and the lepton closest to the MET jet and the MET jet, $\Delta\Phi(l_c, E_T^{miss})$

The kinematic variables used are summarized in Table 4.2 and the distribution showing the agreement between MC and data is shown in Appendix A.

Kinematic variable	Feature name
p_T of both leptons	lep1pt & lep2pt
ϕ of both leptons	lep1phi & lep2phi
η of both leptons	lep1eta & lep2eta
Invariant mass of dilepton pair, m_{ll}	mll
Missing transverse energy in event, E_T^{miss}	met
Missing transverse energy significance in event, E_T^{miss}/σ	met_sig
Transverse mass in in event, m_T	mt
Stransverse mass in in event, m_{T2}	mt2
Transverse energy in in event, E_T	et
ϕ between leption pair, $\Delta\Phi(l_1, l_2)$*	dPhiLeps
ϕ between leption pair and MET jet, $\Delta\Phi(ll, E_T^{miss})$	dPhiLLMet
ϕ between leading lepton and MET jet, $\Delta\Phi(l_l, E_T^{miss})$	dPhiLeadMet
ϕ between closest lepton and MET jet, $\Delta\Phi(l_c, E_T^{miss})$*	dPhiCloseMet
Hadronic activity, H_T	ht
Ratio between E_T^{miss} and H_T	rt
Number of b-jets	nbjets
Number of light jets	nljets

Table 4.2: Table showcasing the kinematic variables that will be used as features.
* These have poor MC and data agreement.

There is also the possibility to include jet-related kinematic variables. But these will become a problem as they will create *jagged arrays*, these are arrays that might not always have a value in them. For instance, if we were to look at the η of the three jets with highest p_T in all of Run II then it becomes clear why this is a problem. The reason being that there might not always be three jets in an event! The problem with these jagged arrays is one that is better suited when discussing how to prepare our ML netoworks and will be discussed furhter there.

As of the jet kinematics, we will look at the p_T, ϕ and η of the three jets with highest p_T, we will also look at the invariant mass of the two jets with highest p_T, to not confuse this with the invariant of the dilepton pair, I will call this variable for m_{jj}. The jet kinematic variables used are summarized in Table 4.3 and the distribution showing the agreement between MC and data is shown in Appendix A.

Should I have an appendix showing the distribution of over 30 variables?

Kinematic variable	Feature name
p_T of two jets with highest p_T	jet1pt & jet2pt
ϕ of two jets with highest p_T	jet1phi & jet2phi
η of two jets with highest p_T	jet1eta & jet2eta
Invariant mass of two jets with highest p_T, m_{jj}	mjj

Table 4.3: Table showcasing the kinematic variables that need padding.

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 the poor agreement between MC and 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 this theis, as shown in Table 4.3, we will record events with up to three jets in the final state. As mentioned earlier, there isn't always three jets in the final state, to mediate this problem I chose to set the p_T to zero for the missing jets and m_{jj} to zero if there are less than two jets, this is something that is physically reasonable as it doesn't violate any conservation laws. More problematic however is the η and ϕ when there aren't jets. To mediate this I've set the values to -999, which has no physical meaning and is impossible to achieve, this I did so it becomes easier for us to identify the jagged arrays further in the network preparation. The missing variables is not a problem when making BDTs with XGBoost. There is also another problem, albeit less problematic than the previous ones, with the final states that are not SFOS, as the MC generated background on these tend to be lower than the recorded 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.

I will remove this sentence, but its just so we know

something more to add?

5.1.1 Zp 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) which produces the WIMP DM particles, and a new theoretical particle, Z', that decays into a lepton pair. The three theories are divided further into MC samples with a Light Dark Sector (LDS) and High Dark Sector (HDS) which tells us therange 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.

see next comment for the two methods

5.1.2 "Ensemble" dataset / Model indepedence

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 training a network using the full dataset it might only focus on a the more resonant models with fixed masses. Also, every different DM sample has different phenomenology, specially in the future when I will be receiving SUSY samples, meaning that it also might not train 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 to re-weight the MC events to expected events gives us an extremely low statistics dataset, which punishes the network for guessing correctly.

THIS IS OUTDATED AND WILL MOST LIKELY BE CHANGED TO THE SIGNAL

This subsection and is outdated with our current plan, and this will change when we discuss further how to implement the model-independent aspect.

REGION APPROACH, WHERE WE STATISTICALLY COMBINE WHAT THE NETWORKS LEARN IN A MODEL DEPENDENT WAY 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 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.

5.2 Neural Network Training

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. 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, as explained in section 3.1.2.

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 jagged arrays. 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 high energy particle physics. To repeat the problem, in the dataset being used in this thesis, we include seven kinematic variables that might not even be there, these are the p_T, η and ϕ of the three highest p_T jets and the invariant mass m_{jj} of the two highest p_T jets. The obvious reason for this is that there might not be a jet in every event, thus we have missing variables. As previously mentioned, the way that the dataset was made, set the missing values of p_T and m_{jj} to zero and the values of η and ϕ to -999 if the jet of the variable was not present.

The jet p_T and m_{jj} 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 particles recorded are. 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.

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 $p_T, m_{jj} = 0$ and $\eta, \phi = -999$ with the mean of every p_T, m_{jj}, η and ϕ (excluding the "missing" values set). 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, but might be of interest for future projects. Another approach, 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 and create a pattern for the network which might lead to a bias.

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, but this is also not a desirable mitigation.

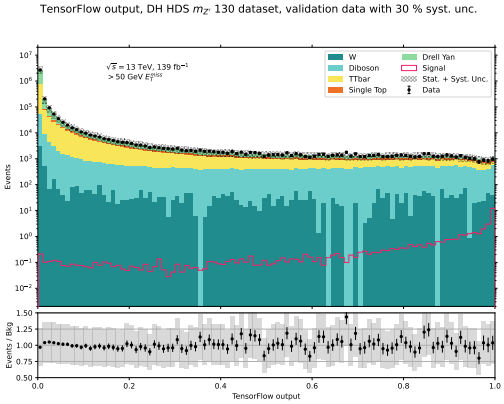
For this project we have tried to create new kinematic variables that work around the need of padding. These features are just *counting features*, meaning that they only count the number of jets that furful the criteria. In this project we decided to look at the number of b-jets with $p_T > 20$ GeV, the number of light jets with $p_T > 40$ GeV, the number of jets recorded in the central calorimeter ($|\eta| < 2.5$), and the number of jets with $p_T > 50$ GeV recorded in the forward calorimeter ($|\eta| > 2.5$). The reason why the light jets and forward calorimeter jets have a higher p_T criteria than the standard 20 GeV, is for the data and MC to agree. A summary of these variables is shown in Table 5.1.

should I write this at all

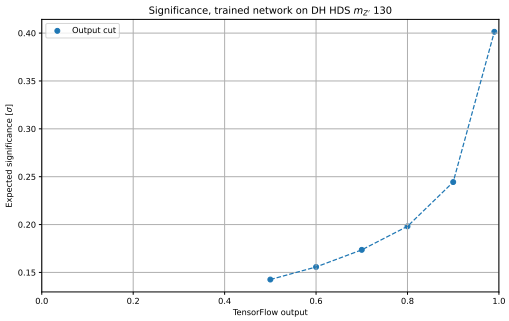
Kinematic variable	Feature name
Number of b-jets with $p_T > 20$ GeV	n_bjetPt20
Number of light-jets with $p_T > 40$ GeV	n_ljetPt40
Number of jets recorded in Central calorimeter	n_jetsetaCentral
Number of jets recorded in Forward calorimeter with $p_T > 50$ GeV	n_jetsetaForward50

Table 5.1: Table showcasing plausible kinematic variables that will not need padding.

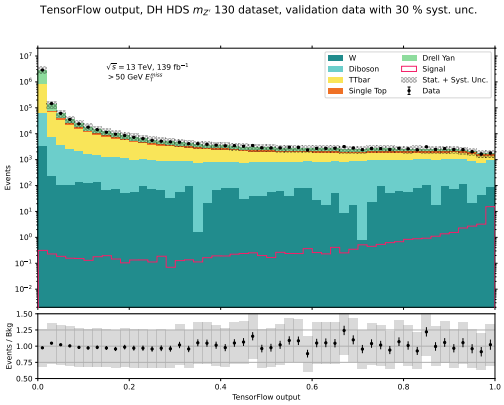
When training our NN with these new variables we would hope that it learns more physics by hopefully recognising patters between all high level features. I have however tested this and the difference in a optimized network (more info on this later) is minimal, and depending on our binning choice might make these new features less suited for our task. The resulta using 100 bins and using 50 bins can be seen in Figure 5.1. For the remainder of our NN endevaours we will opt to not use these new features, and just remove the features that have missing variables all togheter.



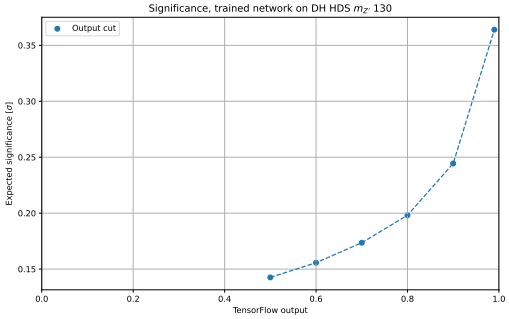
(a) When including new variables and using 100 bins



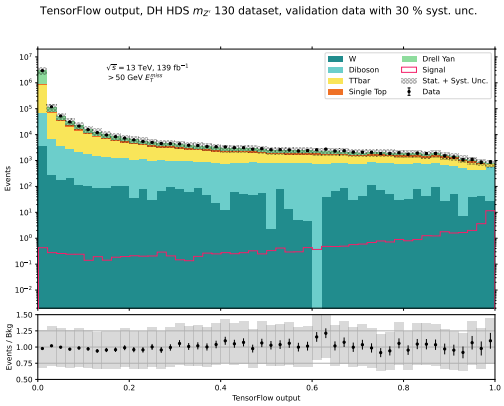
(b) Expected significance of a)



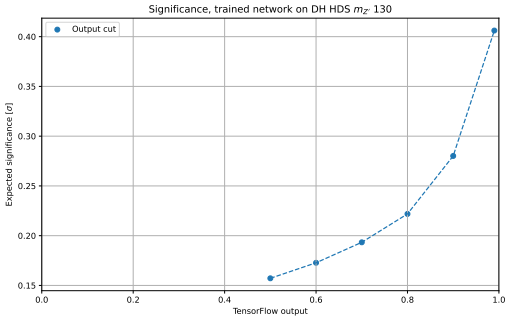
(c) When including new variables and using 50 bins



(d) Expected significance of c)



(e) When excluding new variables and using 50 bins



(f) Expected significance of e)

Figure 5.1: NN prediction when using new features to avoid padding. This is testing a dataset with a Z' DM model.

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*

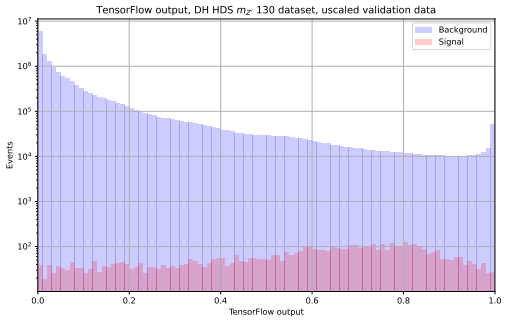
$$X_{norm} = \frac{X - \bar{X}}{\sqrt{\sigma_X^2}} \tag{5.2}$$

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 computationally heavy 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.

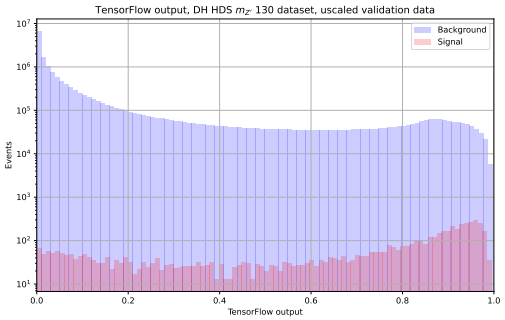
There is a big difference when normalizing data ourselves and using TensorFlow, and that is that TensorFlow will remember how the data was normalized when training and will normalize the testing data in the same way (using the same variables), making testing easier. These different normalization methods have been tested and can be seen in the Figure 5.2. We can see that the best results come from Z-score and Batch normalization. We can see how the expected significance differs on both cases in Figure 5.3.

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? The reason batch normalization might work best for our case is because when we divide the data by bacthes it might unevenly represent the SM / signal and their ratio. But by using batch normalization it takes the average of all the batches creating an closer to real distribution. For the following examples in this section I have used the Z-score method, but I will use Batch normaliazation for the signal search.

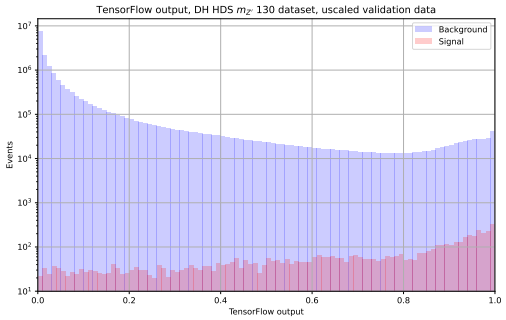
should I even mention this if I will not use NNs further?



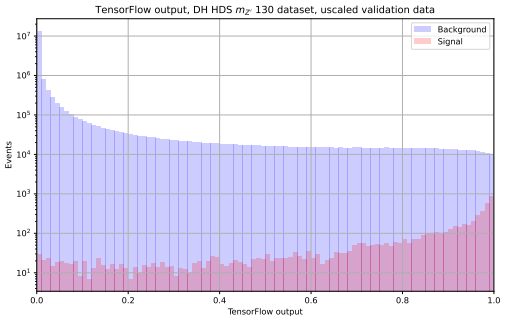
(a) No normalization



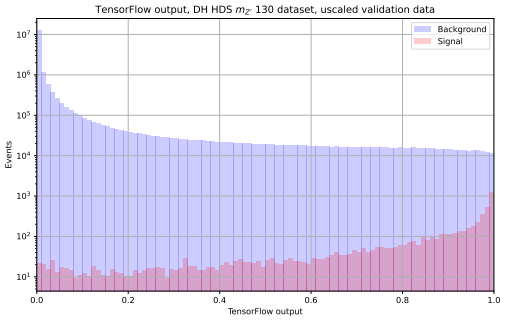
(b) Layer normalization



(c) Min max scaling

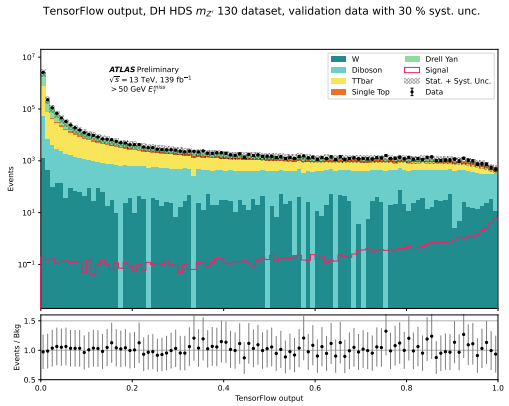


(d) Z-score

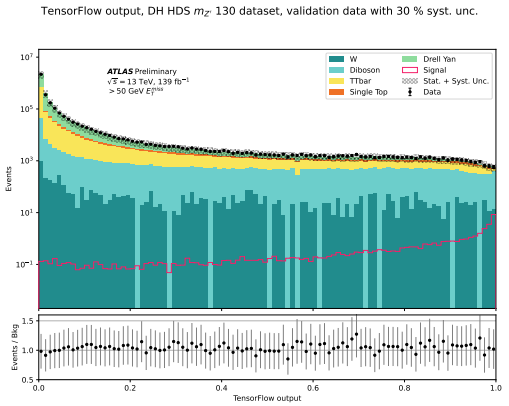


(e) Batch normalization

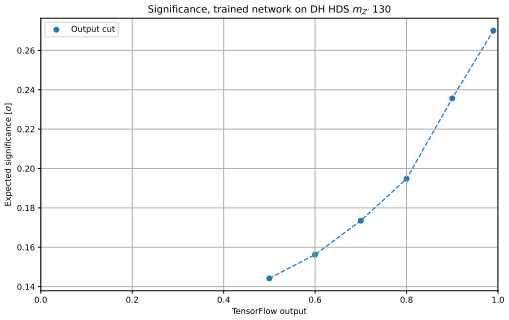
Figure 5.2: NN prediction when using different normalization methods. This is testing a dataset with a Z' DM model.



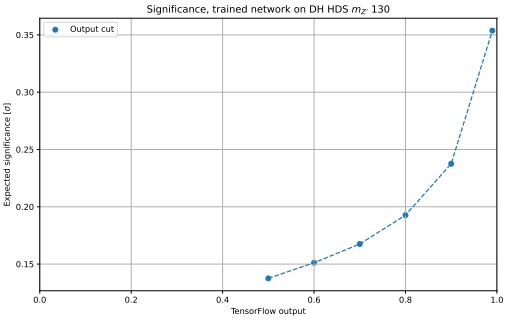
(a) Z-score



(b) Batch normalization



(c) The expected significance of a)



(d) The expected significance of b)

Figure 5.3: 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.

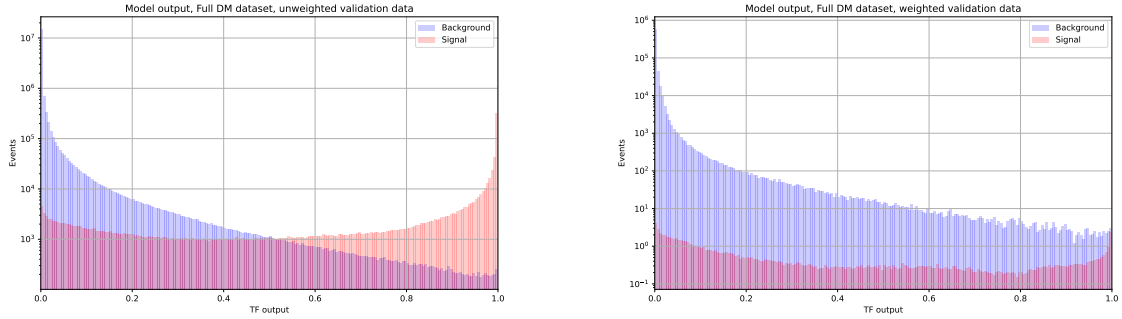
5.2.3 Weights

A big problem that needs to be adressed in this thesis is what we should use as *sample weights* (see Section 3.3). If we were to not use any form of sample weights to mitigate the unbalance in our data set it could potentially lead to undertraining where the networks could get "lazy" and guess that everything is background. If we were to split the full Z' data set into a training and test set, where 80% is used for training, and trained a network without any weights then it seems as if the network excels in identifying our signal. This is however misleading as we need to keep in mind two things.

is it called under-training?

The first one, which **always** will be applied, is that we need to re-weight the MC events the networks trains on to the expected events, this we do by applying the weights explained in Setion ... as well as factoring for the train-test split we made, that means that since our testing set is 20% of the whole data set then we need to multiply this by 5 so we are back at 100%. The difference is shown in Figure 5.4.

will add this later!



(a) Unscaled validation plot.

(b) Scaled validation plot using MC weights.

Figure 5.4: Validation plot of uwweighted network on the first version of the FULL Z' DM dataset.

The second thing, which is more important, is that we are not going to be testing our network on a "full data set", but rather on on one model with fixed parameters. For the purposes of most examples in this chapter, we will conduct our testing on one of the Z' models, more specially it will be a Dark Higgs (DH) with Heavy Dark Sector (HDS) and $m_{Z'} = 130$ GeV. The reason I mention this here, and highlight its importance, is because the data imbalance **will** become a problem when we are studying just one specific model with fixed parameters. To put some numbers, the whole SM background we are studying has roughly 87 million MC events, the whole Z' signal has roughly 3 million MC events, and the DH HDS $m_{Z'} = 130$ GeV model has 40,000 MC events! So if the network were to say that every event is a background event when testing on the small model with fixed parameters, it would be correct over 99.9% of the time. This is obviously something we do not want, as our goal is to make a ML algorithm that actually learns DM signatures.

I trained a network to find the Diboson background channel among the other SM backgrounds to find a general solution to what weights will be used to balance using the sample weight feature, as this is something we know exists. The imbalance of this data set can be seen in Table 5.2.

	Number of MC events	Number of expected events	MC events × expected events $[10^{11}]$
Signal	8,813,716	93,304.9	8.2
Background	61,201,010	2,621,498.9	1,604.4

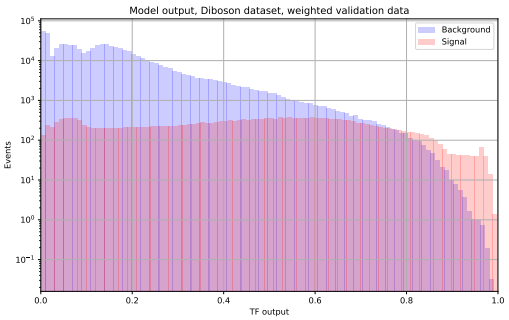
Table 5.2: Table Showcasing how uneven the training dataset is between signal and background. This is on the Diboson dataset which incorporates all the SM MC samples

should I write this or skip it entirely?

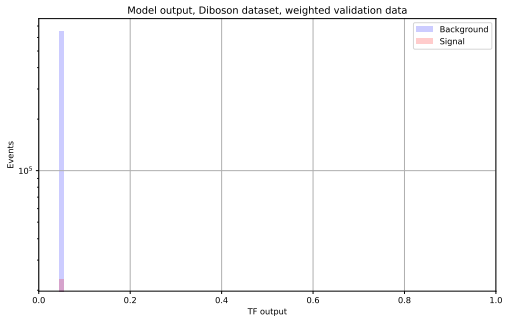
The reason for this choice and not a DM signal, is because I wrongly assumed when first testing the network, that the network *correctly* predicted that there were no DM events. Put in rougher words, I thought that the network completely dismissed the model by claiming it did not exist, and thus excluding the whole model, a very bold, pretentious and wrong claim. As my first attempt to use weights used the

same weights used to re-weight MC events to expected events, which included the cross section of every event... this will become important later.

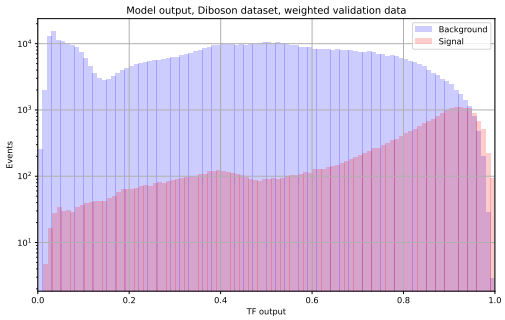
As expected, my interpretation was wrong, the network trained using the weights to re-weight MC events on the Diboson search also predicted that there were "0 Diboson events", something we can empirically and confidently say is wrong. Moreover, the validation plot of the unweighted network works, but gives poor results. As a method to balance the signal and background, I made a new weight array to be used as sample weight. This array weights down all background events with the ratio of MC signal events over MC background events, $\frac{N_{sig}}{N_{bkg}}$. The results are shown in Figure 5.5 and the ROC scores in Figure 5.6.



(a) Validation plot for the unweighted training.



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



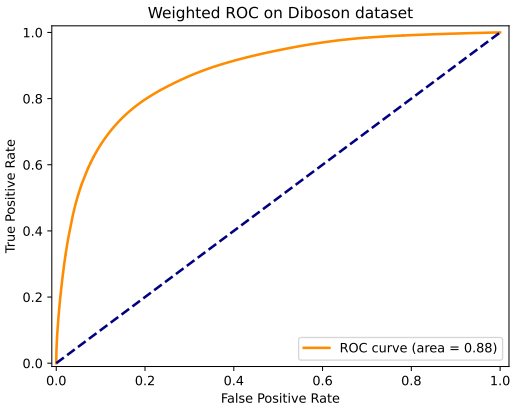
(c) Validation plot for the weighted training using $\frac{N_{sig}}{N_{bkg}}$ as weights on the background.

Figure 5.5: Result of the different network training weighting.

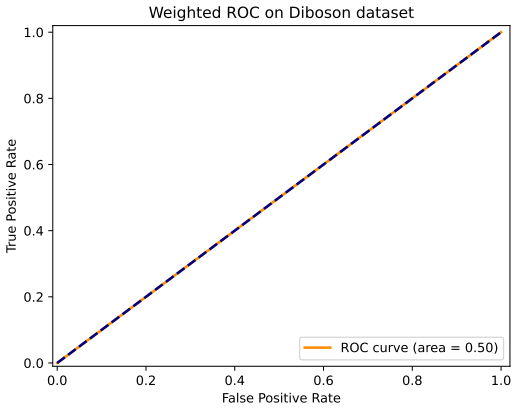
From the results we see that the new weighting method has a poorer result than the unweighted one, and the weighting method that re-weight MC events to expected events just crashes the whole network. The reason for why the latter does not work might be since the network gets punished by guessing "too easily" that an event is signal, and instead of trying harder to learn the patterns and predict with "more confidence" signal events just guesses that everything is a background event and still get a high *accuracy*[1]. Something else to mention, as to why the network does such a poor job at classifying the diboson background, is that the network here was not optimized for the search, and rather just used the defaul NN settings on TensorFlow. And to make matters worse, when testing this I had a bug where I used zero hidden layers by accident, meaning that there was in fact no NN.

mention this or remove completely

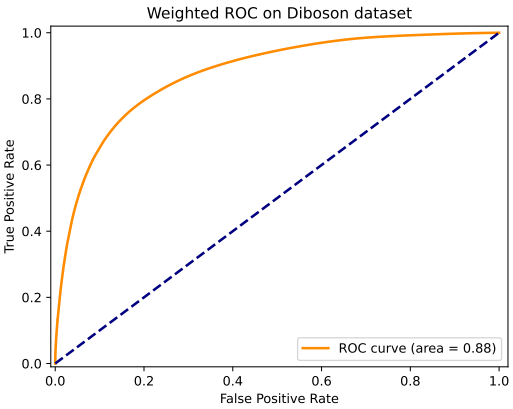
[1] Not to be confused with AUC!



(a) ROC score for the unweighted training.



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



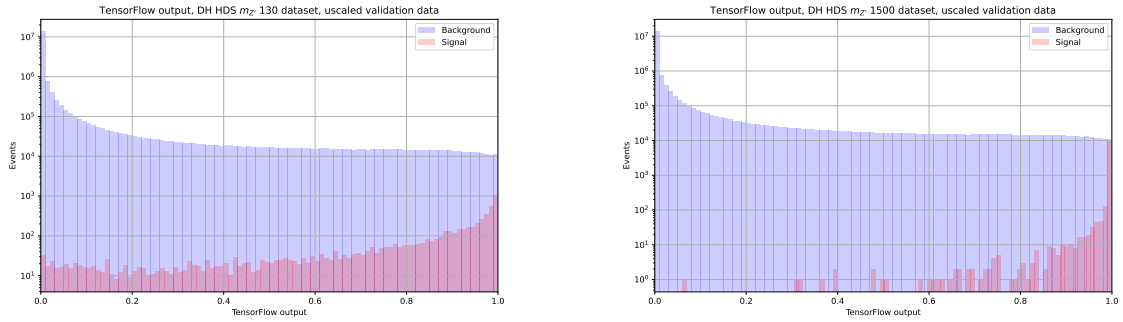
(c) ROC score for the weighted training using $\frac{N_{sig}}{N_{bkg}}$ as weights on the background.

Figure 5.6: Result of the different network training weighting.

5.2.4 Balanced weights

This whole section is my reasoning for trying to re-weight the MC events to expected events. But is just plain wrong.

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 5.7.

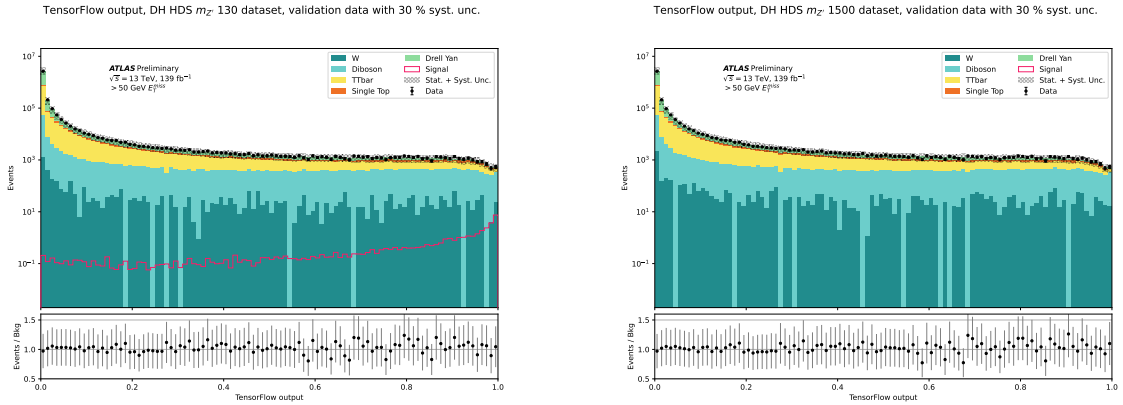


(a) NN prediction of DH HDS $m_{Z'} = 130$ GeV (b) NN prediction of DH HDS $m_{Z'} = 1500$ GeV

Figure 5.7: 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 5.8: 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`.

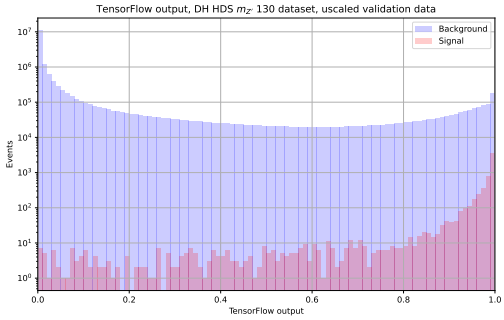
`class_weight` works as a dictionary that weights events that are keys on the dictionary. For our purposes we can make a dictionary where we weight signal and background events differently, this is the same type of scaling that was done in the previous section. `sample_weight` takes in individual weights 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.

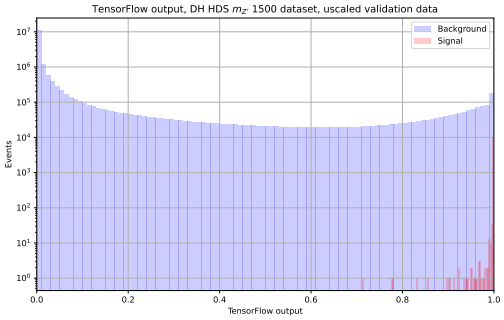
We can see in Table 5.3 how uneven the data (FULL Z') is. To balance the data we have four options. We can either *weigh up the signal* events by the ratio of background over signal. Or *weigh down the bacgkround* with the ratio of signal over background. However this ratio might differ a lot when using 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 5.3: 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.

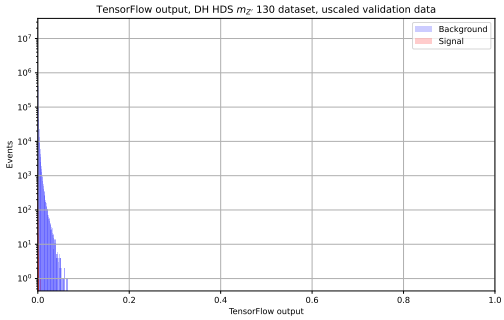


(a) NN prediction of DH HDS $m_{Z'} = 130$ GeV

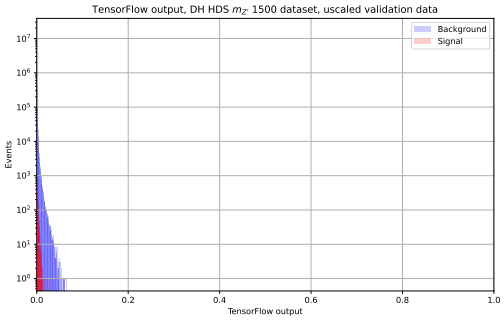


(b) NN prediction of DH HDS $m_{Z'} = 1500$ GeV

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

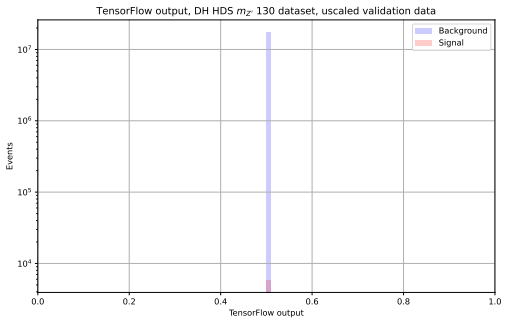


(a) NN prediction of DH HDS $m_{Z'} = 130$ GeV

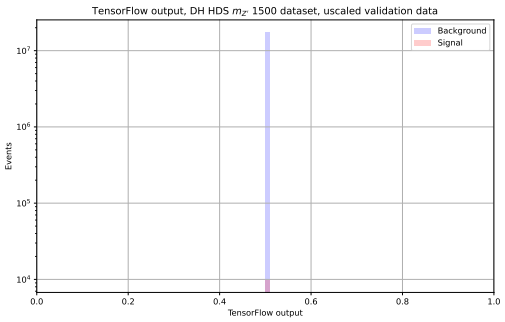


(b) NN prediction of DH HDS $m_{Z'} = 1500$ GeV

Figure 5.10: NN prediction when weighting the signal with the raw event ratio.

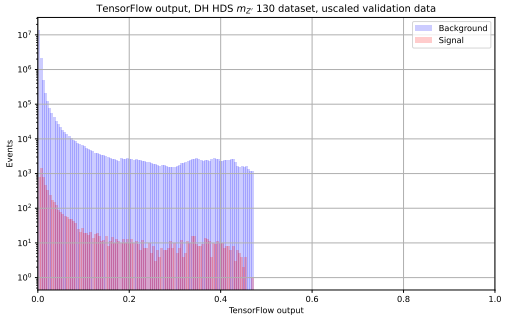


(a) NN prediction of DH HDS $m_{Z'} = 130$ GeV

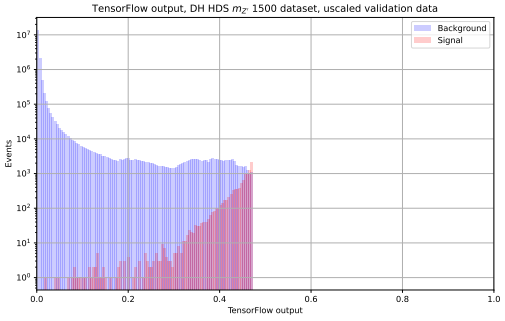


(b) NN prediction of DH HDS $m_{Z'} = 1500$ GeV

Figure 5.11: NN prediction when weighting the background with the SOW ratio.



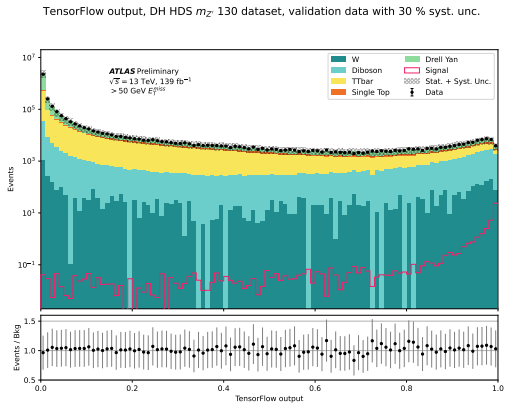
(a) NN prediction of DH HDS $m_{Z'} = 130$ GeV



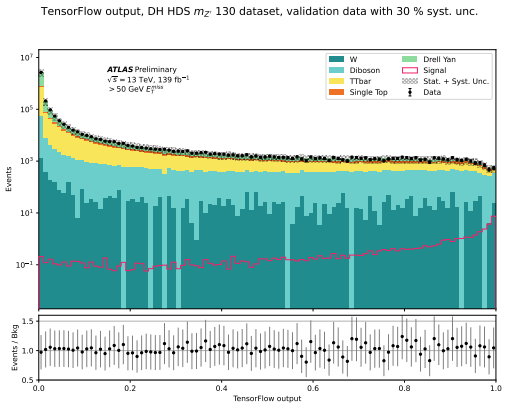
(b) NN prediction of DH HDS $m_{Z'} = 1500$ GeV

Figure 5.12: NN prediction when weighting the bacgkround with the raw event ratio.

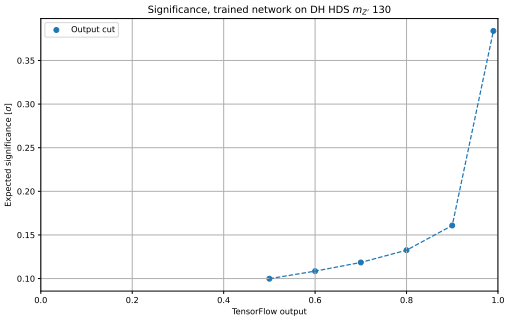
As we can see almost every way of doing this balanced weighting gives poor results. The only one that seems to work is when we we scale up the MC weights with the ratio between the SOW of the bacgkround over signal, however it does not seem to learn the model with lower mass better compared to how it 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.



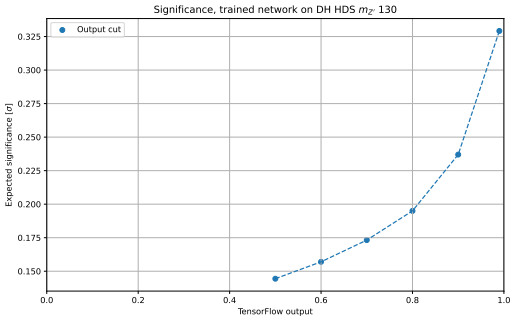
(a) MC weights + scaling signal up with SOW ratio



(b) Only using raw MC events weigh down bacgkround



(c) The expected significance of a)



(d) The expected significance of b)

Figure 5.13: 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.

Should I add here that this will not be pursued further? Or rather include it on the "results" part of the thesis

5.2.5 Architecture

The architecture of the NN utilized in this project is of the form shown in Figure 5.14.

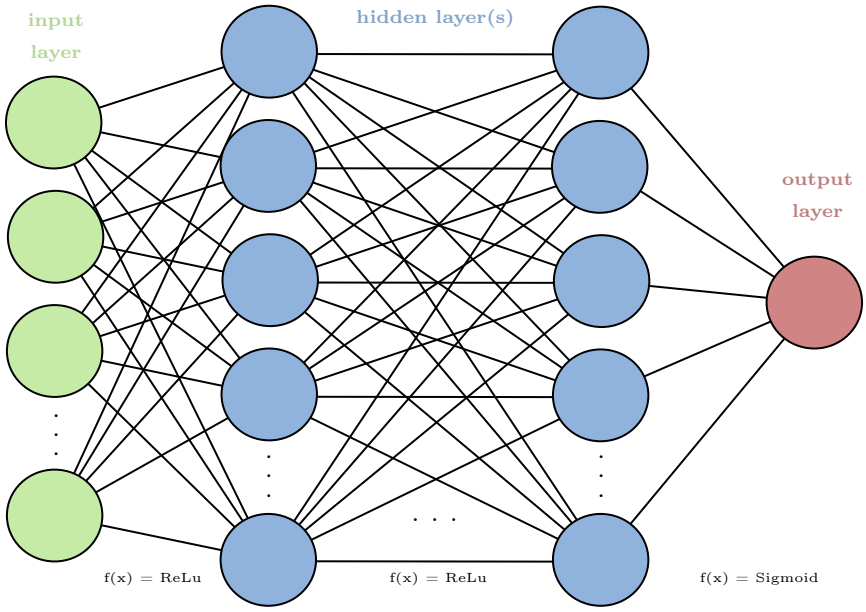


Figure 5.14: Architecture of the NN used on this thesis. The neurons on the hidden layer(s) is a hyperparameter, as well as the number of hidden layers.

Making this type of NN using TensorFlow is easy. An algorithm showing one of the possibilites can be seen in Algorithm 5.1.

Algorithm 5.1: Neural network definition using TensorFlow

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

def Neural_Network(inputsize, n_layers, n_neuron, eta, lamda):

    model=tf.keras.Sequential()

    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.Adam(learning_rate=eta)

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

```

5.2.6 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. (5.4) just in case there is too few events after the network prediction. The expected significance will also be calculcated 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.

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 5.15, while the AUC is in Figure 5.16.

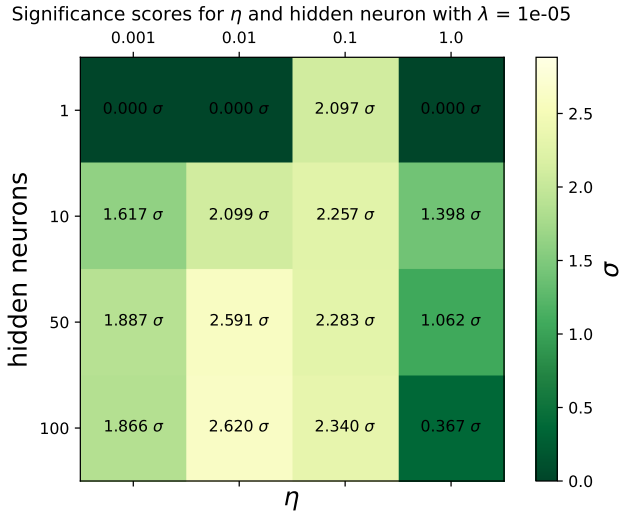
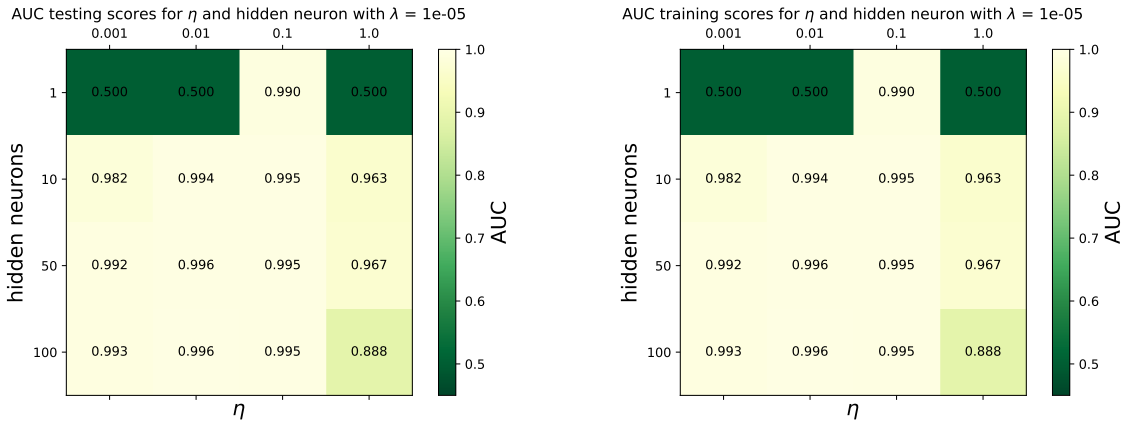


Figure 5.15: Grid search significance with $\lambda = 10^{-5}$ and n_layers $= 2$



(a) Testing AUC

(b) Training AUC

Figure 5.16: 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 5.17, while the AUC is in Figure 5.18.

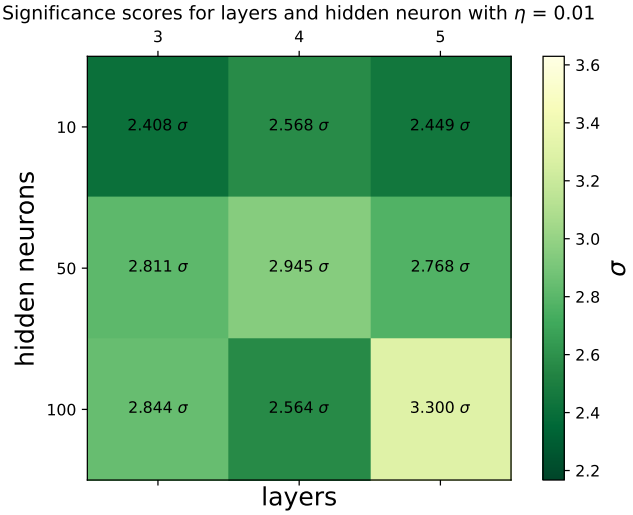
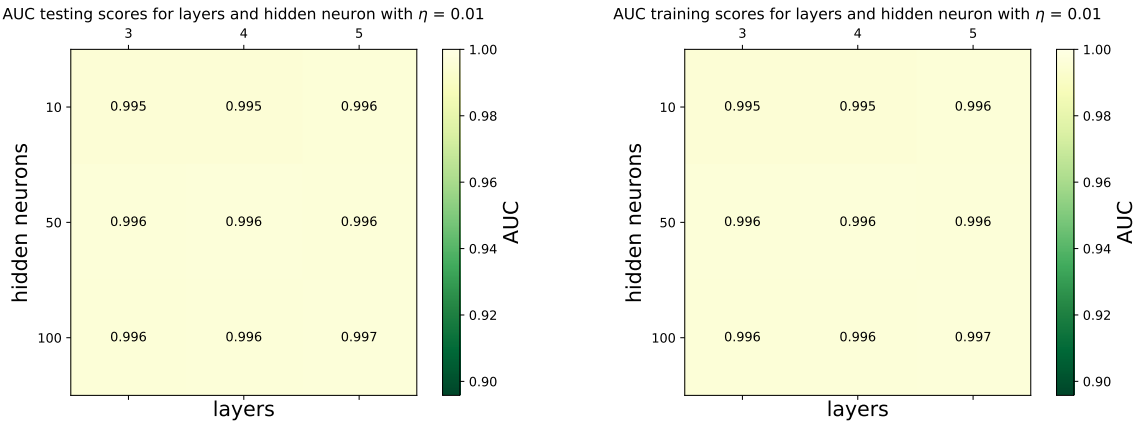


Figure 5.17: Grid search significance with $\lambda = 10^{-5}$ and and $\eta = 0.01$

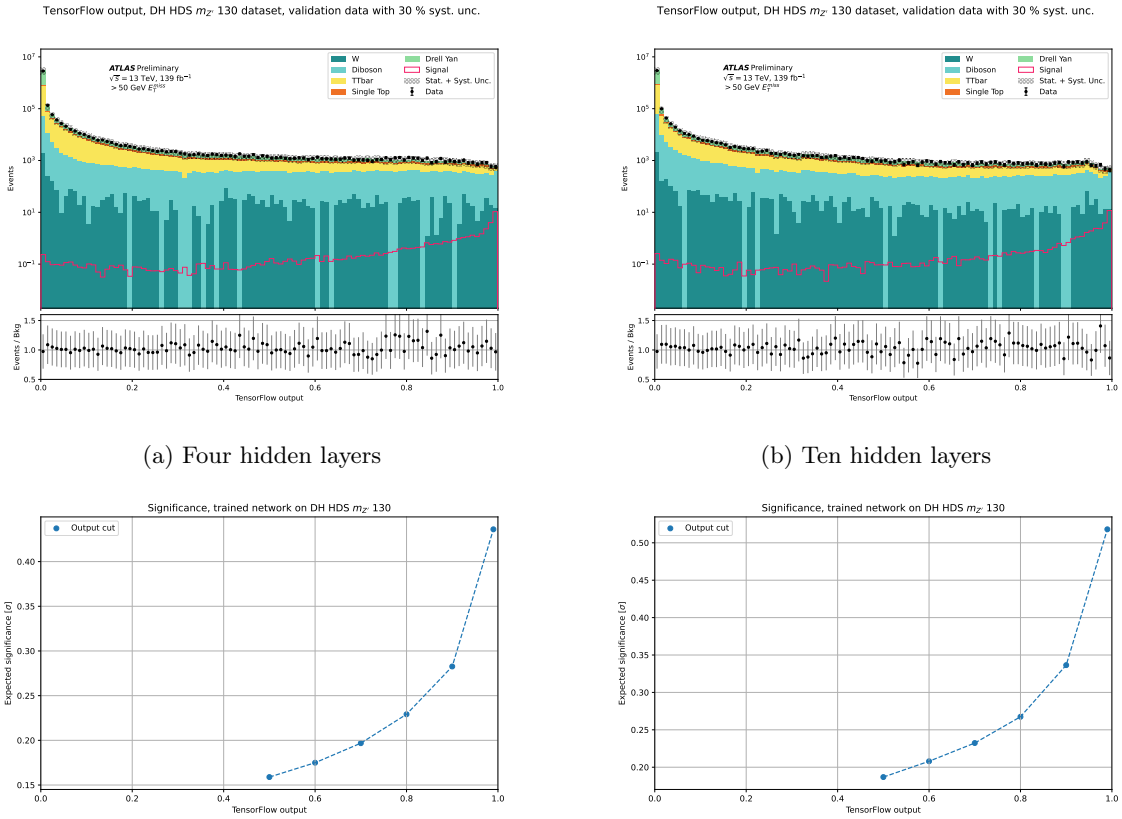


(a) Testing AUC

(b) Training AUC

Figure 5.18: 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 5.17, 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 5.19: 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, one could try the Parametrized NN approach, which could potentially avoid the imbalance problem, but this is proposed as a plausible new research project due to time constrain on this thesis.

5.3 Boosted Desicion Tree Training

When working with BDTs we do not run into as many problems as we do with NNs. For example the padding and normalization of data can be completely avoided, making the whole procedure a lot easier when one uses "weird data" as we do in HEPP. There is however one other problem that need to be dealt with when working with this type of ML, this is the weights, which will be discussed in the next section.

remember to mention it

For this project we will as mentioned utilize the Extreme Gradient Boosting, or XGBoost for short, package made for the HiggsML whenever we mention BDTs. This project utilized version `1.5.0` without GPU adaptability. XGBoost also helps to avoid padding as it is integrated with a `missing_variable` variable where we can simply write the number of the variable that is missing.

add source/remove this comment?

5.3.1 Weights

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. Meaning that the whole problem of section 5.2.4 completely dissapears, meaning we could use the *real* weights calculated in the MC generators. Sadly it is not that easy, XGBoost does not have to possibility to include negative weights, which this dataset has a few of. Some MC generators generate events with negative weights, such as Sherpa , that take into acount higher order diagrams and needs to add negative weights to "counter" the overcounting of diagrams [4], which are important to correctly scale the simulated events to real data. In the future this might no longer be a problem as future MC generators might only have positive weights.

cite sherpa?

I have no other sources to this other than spoken words at conferenes...

A method to mitigate this problem is to use the absolute value of the weights when training to solve, or rather avoid, this problem. This is however not generally accepted as a solution, and some even say it should be avoided. There are other options however, one is to not include events with negative weights on the training set.

Another one that has been used on a published ATLAS (internal) article (chapter 9.3) [5] is to normalize the weights when using the absolute value with respect to the sum of weights over the sum of absolute weights. The reason behind this is because the sum of weights is obviously not the same when we take the absolute value. Mathemathically speaking, if we have an array of weights W, we can update this like

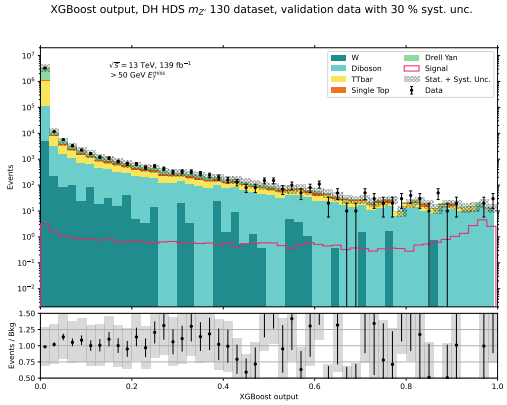
$$W \to |W| \frac{\sum_i W_i}{\sum_i |W_i|} \tag{5.3}$$

such that the weights are at least in the same scale. I have tried all of these options and the results can be seen in Figure 5.20.

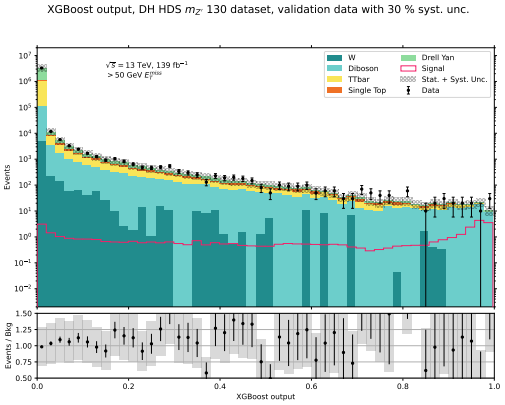
As we can see it makes a significant difference whether we use the weights to re-weight MC events to expected events. But there is no mathematical reason as to why we should include this re-weighting weight as sample weights, as the reason to use sample weights is to *only* balance signal and bacgkround. In theory it makes no sense whatsoevert to include these weights either as the network doesn't really care for cross sections or luminosity, and what we are doing in principle is making it harder for the network to learn anything. But as seen on the results, the BDT learns the background extremely well when using the weights, while it also does a poorer job in learning the signal we are testing.

In a sense this is not a negative thing for our purposes, but stricly speaking we are invoking a semi-unsupervised learning method by punishing the network if it learns the signal too quickly. To get to the point as why this is good for our purposes, we are indirectly making our network more model indepentent! Because of this, the method that will be pursued further in this thesis will be to take the positive weights and balance the data.

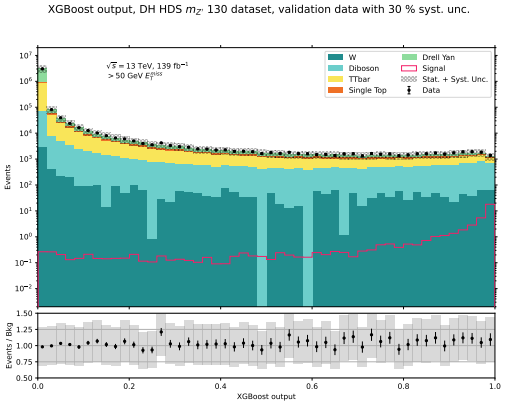
Add that ATLAS used it, so because of that so can I?



(a) Using the scaled absolute value of the weights



(b) Using only positive weights



(c) Using no weights

Figure 5.20: Difference when using different weighting methods. All networks were trained using the balancing method explained in Section 5.2.3

5.3.2 Grid Search

To get the best performance on our BDT we have to do a grid search here as well. The trainable hyperparameters here are different than for NNs though. With XGBoost we have the following hyperparameters

- Tree depth: how many times we split the data
- Number of estimators: how many estimators we use to do out gradient boosting
- The learning rate η
- L2-regressor λ, to stop overtraning

The data set used for this grid search is the same one used in the NN grid search. That is the FULL Z' DM data set containing DH, LV and EFT models. The metrics used to evaluate the score for this case was again the **AUC** and **expected significance**. One thing to note, for this section I did not include the weighting method showcased in the previous section, I only used the data balancing method, as this is its intended purpose.

At first I set the number of estimators to be 120 and fixed the L2 λ to 10^{-5}. Then conducted a grid search on the tree depth and learning rate only. I used the same range on the learning rate as I did with the NNs, $\eta \in [0.001, 0.01, 0.1, 1]$, and first did a treee depth from 3-6. This showed however a trend hinting to the expected significance only increasing with more depth, because of this I ended up continuing the grid search up to a depth of 30. The results can be shown in Figure 5.21. This is however highly radical as the convention is to normally not have a depth greater than 7, the reason being that the network is highly likely to overtrain and give wrong predictions. However this was not the case for me as seen for example in Figure 5.22.

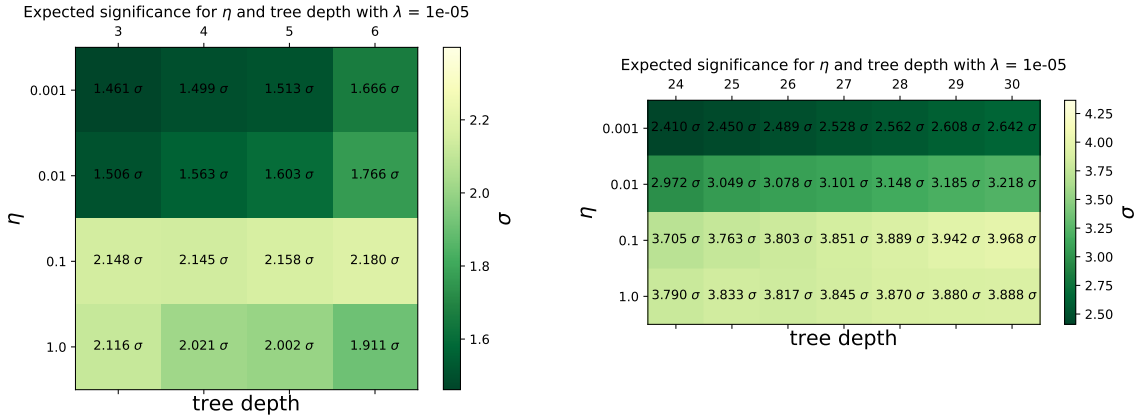
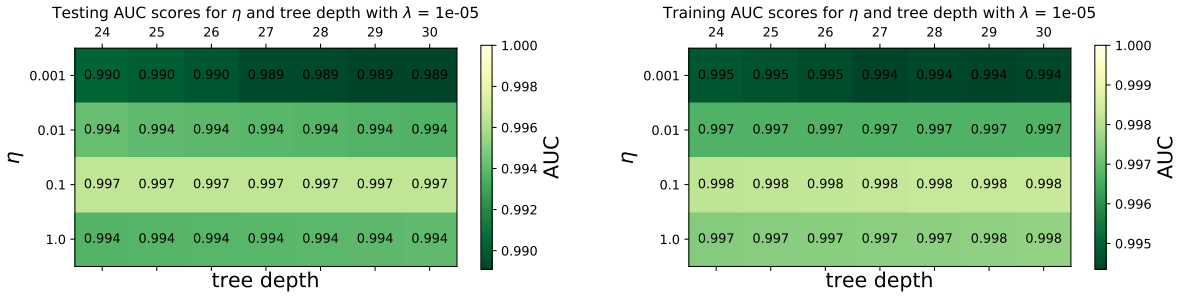


Figure 5.21: Grid search expected significance going to a depth of up to 30

I understand however that this is controversial since we are splitting a data set, that is at best of size 2^{27}, 30 times. That means that after a depth of 27 there is exaclty one event pr branch. So how does a depth of 30 make sense? To help with this we could use a feature in XGBoost to see which features are most important when evaluating a signal. When testing the network trained on the FULL Z' DM data set on a DH HDS $m_{Z'} = 130$ GeV model we get the features shown in Figure 5.23 as most important.



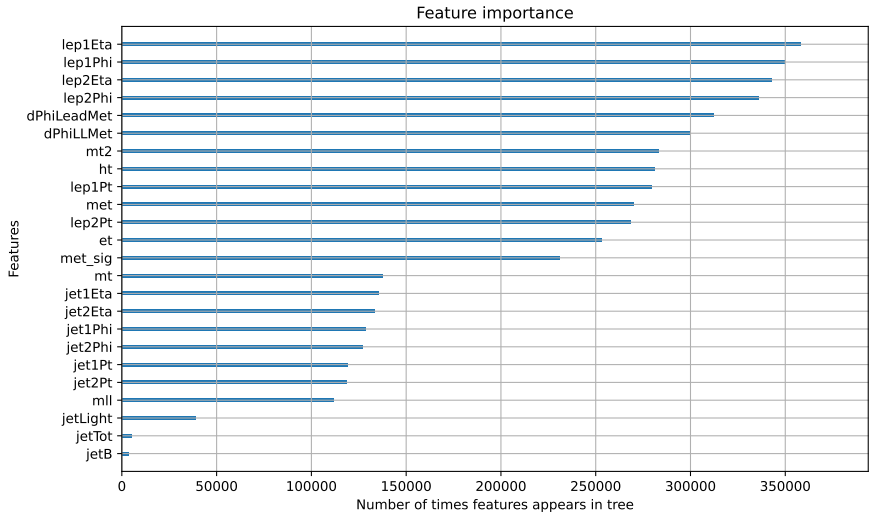
(a) Testing AUC

(b) Training AUC

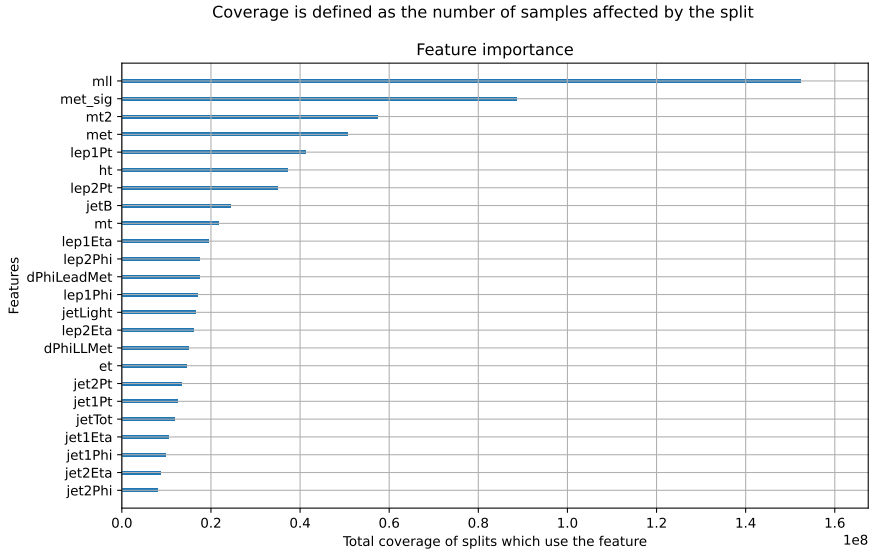
Figure 5.22: Grid search AUC going to a depth of up to 30

As we can see these features vary a lot depending on which metric we use to evaluate the importance. When using the "coverage" metric, which as stated is defined as the number of samples affected by the split, we get the features we physically expect to be important when trying to single out a DM model. And this metric is arguably the one we need to use to define what features are important. Since the more samples a feature split, the more powerful it is to separate signal from background.

We can see however that when we use "weight" as a metric, which is the XGBoost standard metric, we get completely unexpected features that we physically don't expect to be important when trying to single out a DM model. But as described by the metric, the "weight" is the number of times a feature appears in a tree. Which might explain that the reason the pseudorapidity and ϕ range so high on this list, is simply because the tree is struggling to find a pattern here and is trying extra hard to single out DM from SM.



(a) Using "weight" metric



(b) Using "coverage" metric

Figure 5.23: Feature importance of depth 30 network trained on FULL Z' DM data set when testing it on DH HDS $m_{Z'} = 130$ GeV model.

should I conduct a new grid search with different λ and loss functions?

As the previous results are to be taken with a heavy grain of salt, I conducted another grid search. On the second grid search I set the values of $\eta = 0.1$ as the trend showed this giving the best results with less overtraining, and $\lambda = 10^{-5}$. This grid search had `n_estimators` $\in [10, 100, 500, 1000]$ and depth $\in [3, 4, 5, 6]$. The expected significance is shown in Figure 5.24. The testing and training AUC can be seen in Figure 5.25.

When testing the best network with a depth of 6 and 1000 estimators on the same DH HDS $m_{Z'} = 130$ GeV model we get the feature importance plots shown in Figure 5.26. Here we see that the "weight" metric gives us the expected features as most important. But the "cover" metric seems to be less of what we expect since the jet kinematic variables score higher, this might just be a curiosity rather than something to be suspect of.

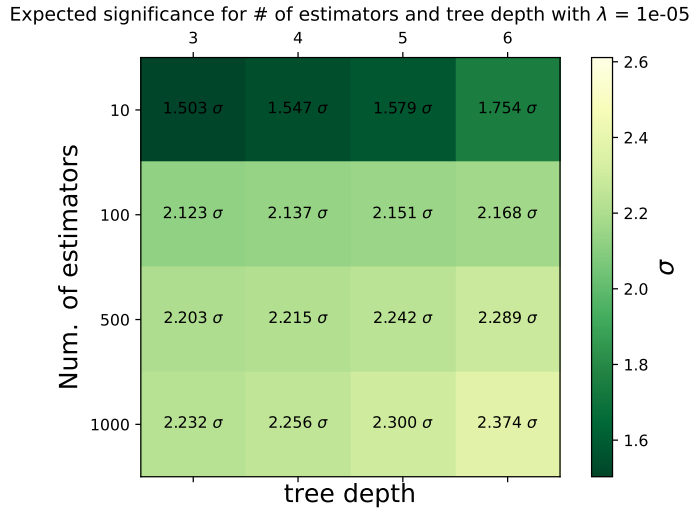
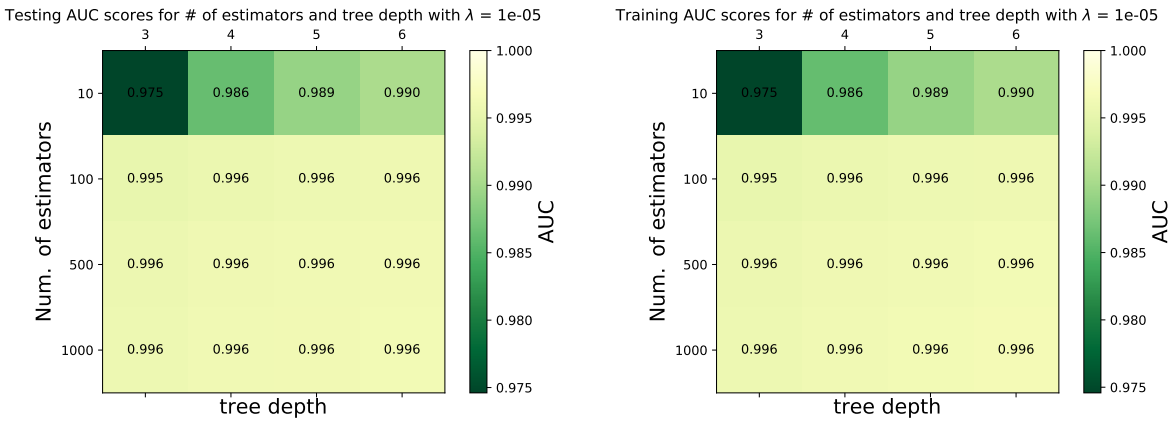


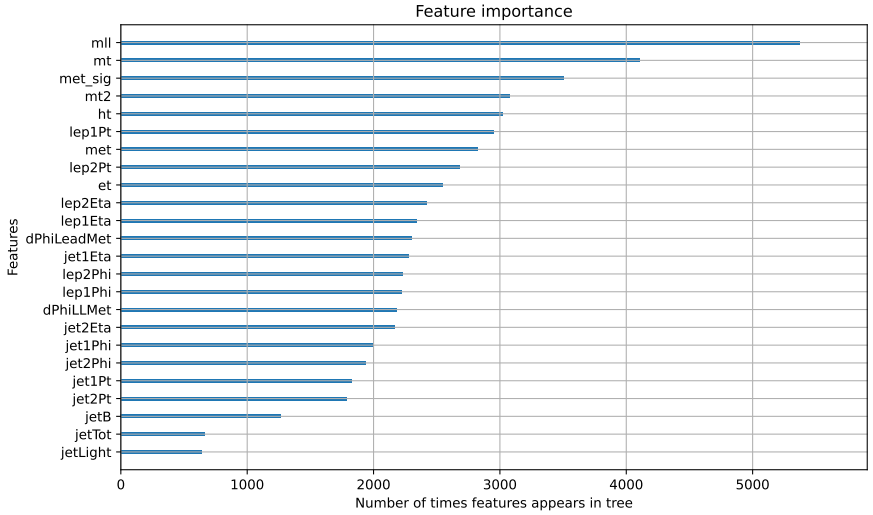
Figure 5.24: Grid search expected significance when setting $\lambda = 10^{-5}$ and $\eta = 0.1$



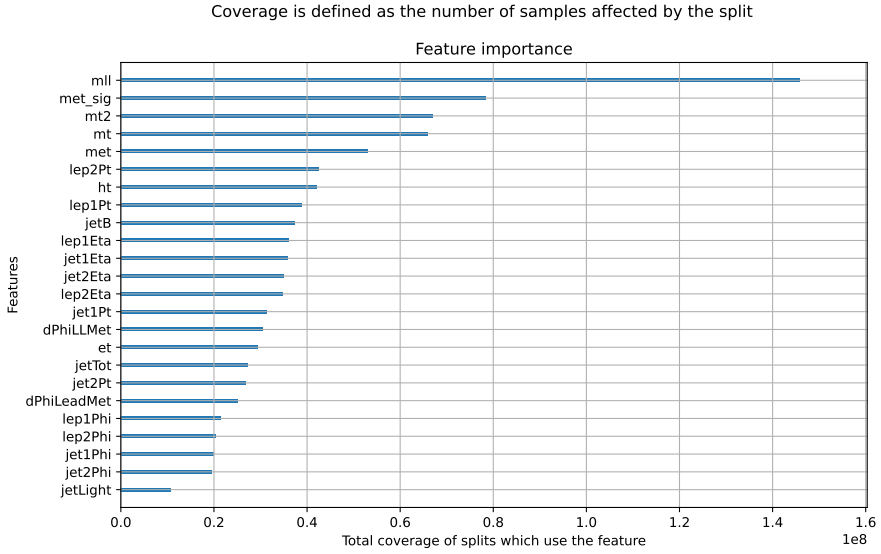
(a) Testing AUC

(b) Training AUC

Figure 5.25: Grid search AUC when setting $\lambda = 10^{-5}$ and $\eta = 0.1$



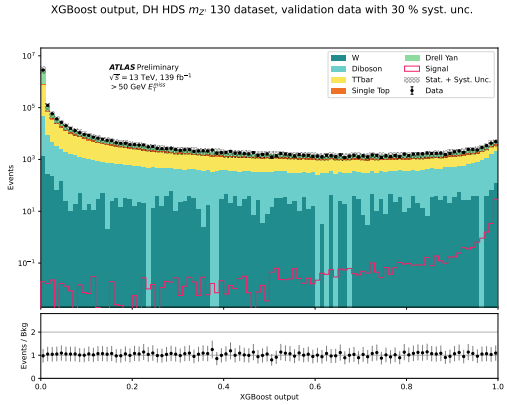
(a) Using "weight" metric



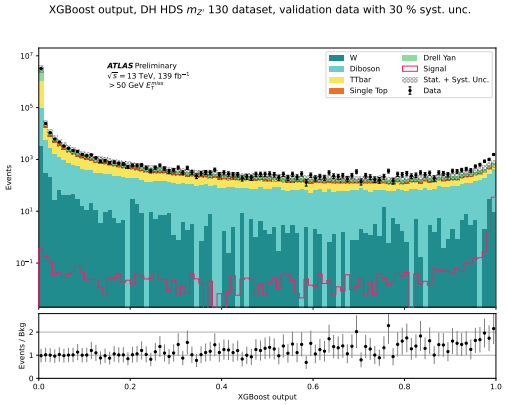
(b) Using "coverage" metric

Figure 5.26: Feature importance of depth 30 network trained on FULL Z' DM data set when testing it on DH HDS $m_{Z'} = 130$ GeV model.

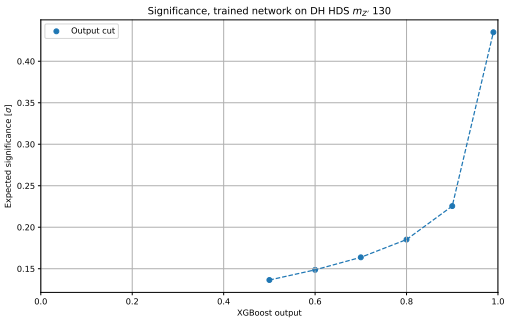
To showcase the difference in signal recognition between the monstrous 30 depth BDT to the more sensible 6 depth BDT, I again tested the networks on the good old DH HDS $m_{Z'} = 130$ GeV model. The results as well as their expected significance can be seen below.



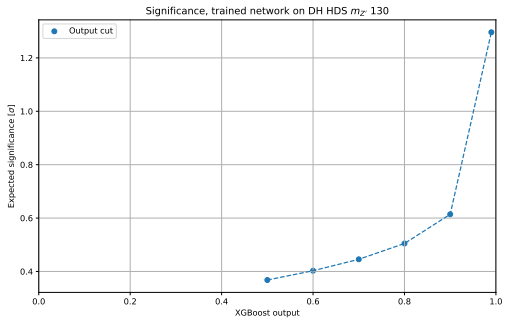
(a) Depth of 6



(b) Depth of 30



(c) The expected significance of a)



(d) The expected significance of b)

Figure 5.27: Comparison of the network performance when having a depth of 6 and 30. 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.

The difference is extreme, when looking at the monster of depth 30 we can get an expected significance of 1.2 σ (without uncertainties) on our model of max 15 events, only having made a cut of 50 GeV on the missing transverse energy. We can however see that the data and background do not agree to the same degree of the network with depth 6. Using purely statistical uncertainty and assuming a systematic uncertainty of 30%, we see that a few data ponts do not agree with the MC background. These data points are points the network classified as signal, so if we completely trusted the network this would be a hint of new physics! However this is the last thing we should assume, and rather take this as a hint that the network is doing something fishy.

when should I use them?

If I had access to XGBoost with built GPU support, I would increase the number of estimators even more to check if this increases the significance while still having a depth of maximum 6. However as of now this is not possible. As the weighting method explained in the previous section was not included here, we will drop going to a tree depth of 30, and have a maximum of 6.

Part III

Results

5.4 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 5.4.

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

Table 5.4: 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

$$Z = \sqrt{2\left[(s+b)\ln\left(1+\frac{s}{b}\right) - s\right]} \tag{5.4}$$

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 5.7 for the electron channel and Table 5.6 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.

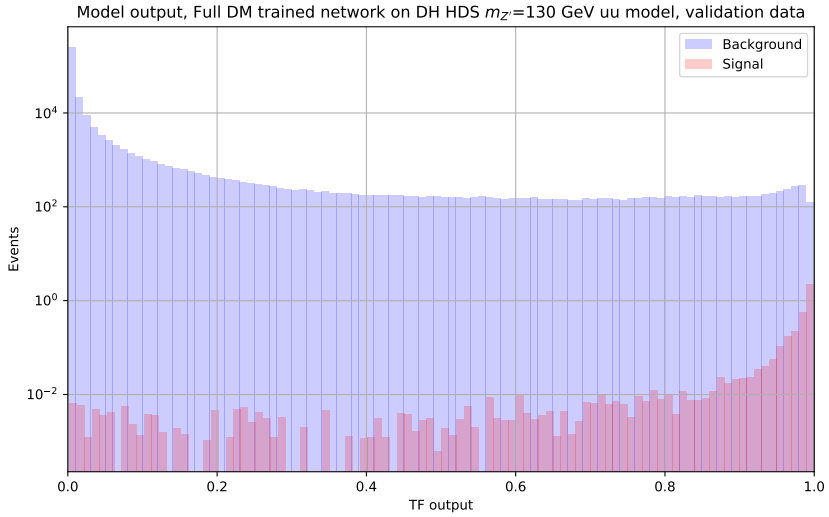
	LV	DH	EFT	Background
Events before cuts	15	20	0	1,256,624
Events after cuts	4	6	0	117
Expected significance $[\sigma]$	0.4	0.6	0	

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

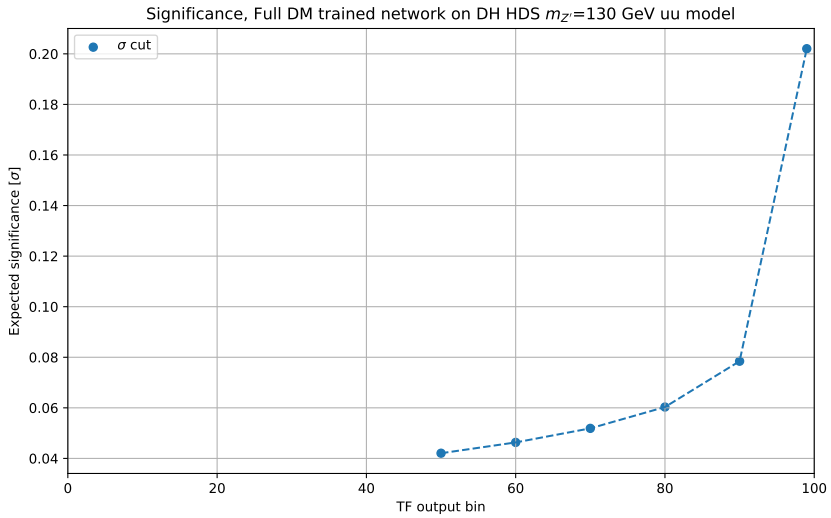
	LV	DH	EFT	Background
Events before cuts	14	19	0	1,626,098
Events after cuts	3	5	0	108
Expected significance $[\sigma]$	0.36	0.51	0	

Table 5.6: 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 5.28 for XGBoost and Figure 5.29 for the Neural Network.

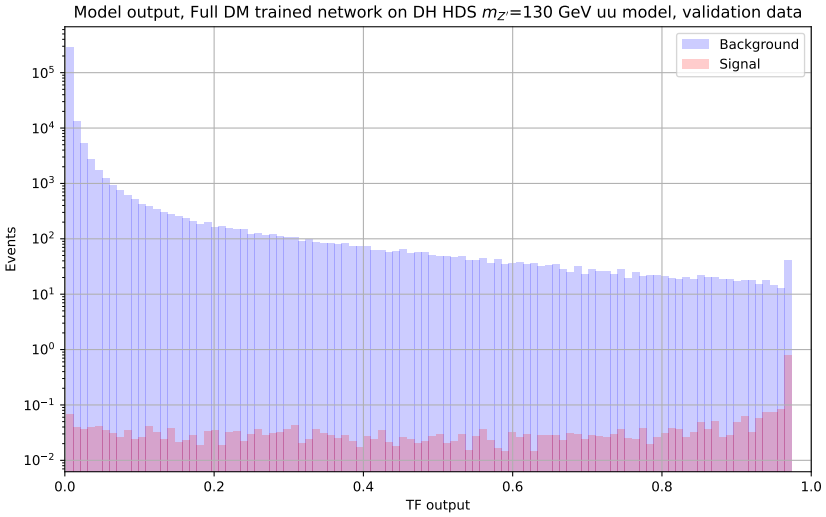


(a) Validation plot.

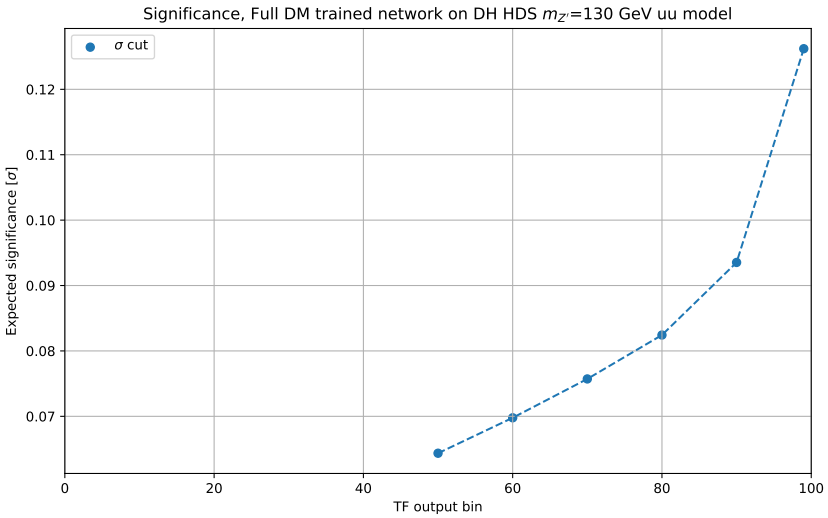


(b) Expected significance when looking at bins and forth.

Figure 5.28: 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 5.29: 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.

	Signal	Background
MC events	2,990,986	69,664,902
Sum of "Weight"	388.75	2,714,091.3
Sum of generator weights	236.3	55,446,228,776,354.8
Sum of (generator weights*lumi / SOW)	9,167.1	61.1
Sum of (generator weights/SOW)	199.21	1.52

Table 5.7: Table showcasing the result of the cut and count method for the electron channel.

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

Table 5.8: 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.