LU-TH 21-22
June 2021

Using Artificial Neural Networks to optimize scattering probabilities

Erik Rustas

Department of Astronomy and Theoretical Physics, Lund University

Bachelor thesis supervised by Rikkert Frederix

LUND
UNIVERSITY

Abstract

Monte Carlo event generators are used by theoretical particle physicists to get a better understanding of the phenomena in particle physics. Given the improvements in precision and accuracy of event generators, using these tools can be very CPU intensive. A method of unweighting events using artificial neural networks is presented to improve the efficiency of event generation. An introduction to machine learning as well as an introduction to the unweighting procedure is given as a basis. Results are given by comparing the "classical" and artificial neural network unweighting. The efficiency is expressed as factors of computing time for the matrix element and the model's predicted value.

Popular Science

For centuries humans believed that matter was made of indivisible small particles called atoms (from the word atomos Greek for "indivisible"). We now know that atoms are made of even smaller constituents of matter. These are called fundamental particles such as the electron or the Higgs boson. How these fundamental particles interact with each other is best explained by the theory called the Standard Model. Further research in the field of particle physics is still an ongoing subject even today.

One way of studying how particles interact is by colliding particles together such as the Large Hadron Collider at CERN. The Large Hadron Collider accelerates particles to nearly the speed of light in two different tubes that run opposite each other in circular loops. When these tubes intersect, the particles collide into each other creating extremely high energy collisions. From these collisions one can analyze the data and find new particles. However one does not need a collider to extract useful information about particle physics. Theorists use computer simulation programs called event generators. These event generators simulate high energy particle collisions by Monte Carlo methods. The event generators are essential tools to theorists as they allow for confirmation of theoretical predictions. Given their significance, the search for higher precision composes a highly demanding computational cost. Optimizing these event generators using machine learning techniques will be this project's main goal.

Machine learning is part of the field of artificial intelligence, which learns by processing sample data. These types of artificial intelligence are used in for example data filtering such as spam mail. This project will use an artificial neural network, which is a branch of machine learning inspired by the structural construct of the brain. The brain is amazing at remembering and learning new things, so building a program inspired by how the brain works is only natural.

One of the Monte Carlo methods that event generators use is called the "unweighting" method. The generated events from the event generator have a value, or weight, mainly determined from the matrix element. Making these weighted events have a more nature-like distribution, where the events have an equal unit weight, is the process of unweighting. This process scales rapidly with the amount of particles in a given interaction. Where the complexity of the matrix element increases as well as the amount of times the matrix element has to be evaluated. Machine learning unweighting will be presented and tested in this project. This method will use mainly the artificial neural network's prediction of the matrix element instead of the actual matrix element. Machine learning and classical unweighting methods will be tested where the break even point of efficiency will be found. This is the factor that compares the time to do one evaluation of the matrix element, and the artificial neural networks prediction. The break-even point of efficiency will be how much more computational time the matrix element has to be for the machine learning unweighting to be a more favorable method.

Contents

1 Introduction **1**

2 Introduction to Artificial Neural Network **2**

2.1 Activation function 3

2.2 Loss function 4

2.3 Optimizer function 5

2.4 Data scaling 6

3 Event Generator **8**

3.1 Monte Carlo Method 8

3.2 Unweighting 9

4 Method **12**

5 Results **15**

6 Conclusion **21**

1 Introduction

The Standard Model is the most successful theory of particle physics describing all particles and the fundamental forces (except gravity) and how they interact, but the theory of particle physics is not complete. There are still things that need to be incorporated. For instance the existence of dark matter or the theory of gravity is not explained by the Standard Model. Many tests are being done to try to answer these questions. One of the experiments that are being done is by particle colliders, which is a type of particle accelerator. They work by accelerating several beams of particles to extremely high kinetic energies, for which the beams collide into each other in a head-on collision. Because of the highly kinetic energies and by the mass–energy equivalence, heavy and unknown particles may be found. The largest particle collider in the world is the Large Hadron Collider (LHC) at CERN in Geneva [1], a 27 km long ring with the current world record of a collision energy of 13 TeV.

Another approach other than the particle colliders is the use of event generators. These are computational frameworks that use Monte Carlo methods. Event generators are useful since they simulate high energy particle collisions where theoretical predictions can be tested. The Monte Carlo method that we are looking into is the method of unweighting. The event generator estimates the cross section by numerically solving an integral of a given process. In such a process the given event has a weight depending mostly on the matrix element, but also on the phase-space. In a nature-like distribution, each event has a unit weight. Going from weighted events to unweighted events is the process of unweighting.

The issue at hand is the evaluation of the high-dimensional matrix element function. This function scales rapidly with the number of final state particles in a given process such that there is a need for new unweighting methods. The topic of this project is to build an Artificial Neural Network (ANN) that can be used in what is known as ANN unweighting. We will focus on a supervised regression type model that tries to make a "fit" for the three-momenta in a given process to its corresponding matrix element. This fit or prediction of the matrix element will then be used in the ANN unweighting. Given the problem at hand, the evaluation of the matrix element, the ANN unweighting will mostly use the predicted value instead and in such a way maybe be more efficient.

This thesis will begin with a background of ANN: How it is structured and the most important functions for an ANN. In section 3 the description of

how the classical unweighting works, as well as the ANN unweighting. After the theoretical background, a method on how the ANN was built, as well as how to code the unweighting methods will be presented in section 4. In section 5, the result of how accurate our model is will be presented in the form of a plot. The efficiency of comparing classical versus ANN unweighting will be measured in terms of computing time and the physical validation will be presented in different histograms.

2 Introduction to Artificial Neural Network

Looking at the recent decade, the increasing development of computer hardware has led to an exciting topic of data handling. Machine learning (ML) is a powerful tool to use when handling larger datasets. ML is a computer algorithm that learns by itself by processing sample data. These are useful in many applications, such as reading handwritten text, removing spam mails or predicting weather forecasts. This project will look at a branch of machine learning called Artificial Neural Network. They are inspired by how the brain operates. Humans have an excellent capability of learning new things, and machine learning techniques try to imitate these. An ANN is built using two objects, nodes and synapses. These nodes are structured in layers, and each node is what is known as a weighted sum of the nodes in the previous layer. The synapses are these connections between the nodes, and they pass information along with the network.

We can picture an ANN as several vertically layers with nodes, where the first layer is the input layer and the last one the output layer. These two layers have the dimensions (number of nodes) of what problems one wants to solve. If one wants a model that tries to predict a 28×28 pixel image of an integer between $[0, 9]$. Then that model would have a $28 \times 28 = 784$ dimensional input layer, that is one node for each pixel. The output layer would be 10 dimensional, one node for each possible integer. This type of model is what is known as a classification model, these are ANN models that predict discrete values. Let's take the same example as before but this time the ANN tries to predict the pixel image but for a float between $[0, 9]$. We would still have a 28×28 dimensional input layer but this time we don't have an integer with 10 possible discrete values, but a float between $[0, 9]$ which can take infinitely many values. It would be impossible to have an infinite dimensional output layer such that for this type of problem the output layer has one node which gives out a number of infinite range. This type of ANN model is called a regression model.

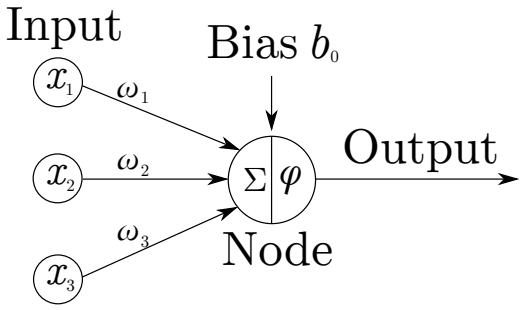

Figure 1: *Schematic representation of one node (artificial neuron) with input* $\mathbf{x} = (x_1, x_2, x_3)$*. Each input* x_k *gets weighted by a factor* ω_k*, summed up and added a bias factor* b_0 *such that* $a = \sum_k \omega_k x_k + b_0$*. Node* a *is passed through an activation function:* $\varphi(a) = \varphi(\sum_k \omega_k x_k + b_0)$*.*

There exists a third type of layer called "hidden layer" of nodes that is between the input and output layer. The hidden layer is commonly used (employed) when one has more complex problems such as a nonlinear system. There is no limit on how many hidden layers one can have and how many nodes these hidden layers have.

2.1 Activation function

As mentioned earlier the artificial synapses pass on information between nodes. If we consider just one node a we can define its input as a weighted sum

$$a = \sum_k \omega_k x_k + b_0 \tag{2.1}$$

where x_k denotes the nodes from the previous layer, ω_k are the weights that the synapses give for each connection between nodes and b_0 is a bias factor of the node a. The structure for this with 3 inputs is seen in figure 1. The output of node a is then what gets passed on to the next layer of nodes. The output of the node can be interpreted by a function called the activation function $\varphi(a)$. The activation function looks different depending on what type of problem one wants to solve (classification or regression). The activation function is often always differentiable, which is important as we will see later.

Some example of activation functions that are commonly used are

$$\begin{aligned}
&\text{Linear function,} && \varphi(a) = a \\
&\text{Rectified function,} && \varphi(a) = \max(0, a) \\
&\text{Sigmoid function,} && \varphi(a) = \frac{1}{1+e^{-a}} \\
&\text{Tangent hyperbolic function,} && \varphi(a) = \frac{e^{a}-e^{-a}}{e^{a}+e^{-a}}
\end{aligned} \tag{2.2}$$

For the output layer there is a clear distinction on what type of activation function one may choose. If it is a classification model then you want the nodes of the output layer to make discrete values, that is we want something that works like a "switch" that turns on and off a node in the layer. For this purpose the sigmoid activation function is great since it has a range of $(-\infty, \infty) \to (0, 1)$. For a regression type model, linear or rectified linear function (ReLU) is useful. That is because the output nodes have infinite range and the activation function for these models should not categorize the result. ReLU is the same as a linear function when a is positive, but sets all negative values of a to 0. The choice of activation functions for the hidden layers are more free.

2.2 Loss function

In this thesis we will use the ANN only as a regression model. A regression model takes in $\mathbf{x}$ as input and performs an output function $y(\mathbf{x})$, this is the model's prediction. The actual target value is called d_n. During the training phase, the model learns by processing sample data where the ANN is given a target value d_n for each input vector $\mathbf{x}_n$. The loss (or cost) function E is a function that compares the output function $y(\mathbf{x})$ and the given result d_n. This means that the loss function gives us a quantity that tells us how accurate the prediction is. Let $\boldsymbol{\theta}$ be all the changeable parameters (ω, b_0) that affect the output function $y(\mathbf{x})$. The loss function $E(\boldsymbol{\theta})$ compares the result d_n with the output $y_n = y(\boldsymbol{\theta}, \mathbf{x}_n)$. An example of a loss function is the mean squared error (MSE)

$$E(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n}^{N} (y_n - d_n)^2 \tag{2.3}$$

where N is the total number of outputs and the sum runs over all these. An ideal prediction would make the loss function equal to zero and we would

have an ideal ANN model. If the loss function is non-zero we need a function that interprets the result and tries to minimise it, namely, the optimizer function.

2.3 Optimizer function

So far we have an ANN model that takes in an input $\mathbf{x}_n$ and works itself through the network by each node's activation function, where finally the model makes a prediction y_n in the output layer. This is what's known as forward propagation. This prediction is then compared with the real value d_n via the loss function. The optimizer function's job is to minimize the loss function by updating the changeable parameters $\boldsymbol{\theta} = (\boldsymbol{\omega}, \mathbf{b_0})$ by an algorithm called gradient descent (GD). GD works by taking the derivative of the loss function w.r.t θ_i by using the chain rule

$$\frac{\partial E}{\partial \theta_i} = \sum_n \frac{\partial E}{\partial \varphi_n} \frac{\partial \varphi_n}{\partial a_n} \frac{\partial a_n}{\partial \theta_i}. \tag{2.4}$$

This signifies the importance that the activation functions and the loss functions are C^1-smooth functions. The GD algorithm for the θ_i'th changeable parameter is then

$$\theta_i \leftarrow \theta_i - \eta \frac{\partial E}{\partial \theta_i}, \tag{2.5}$$

where η is the learning rate and controls the size of an update. The negative sign is because we want to move closer to the minimum. This is applied to all changeable parameters

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{\nabla}_\theta E(\boldsymbol{\theta}). \tag{2.6}$$

The gradient descent starts at the output layer and works itself backwards through the network and updates each parameter layer by layer. This is what is known as backpropagation. It is important to note that $\boldsymbol{\theta}$ starts at random numbers and many updates are needed to find a minimum for the loss function.

It is important to define an epoch, since they are essential to this topic. Let us define a training set with N samples then we have N inputs x_n and results d_n. One epoch would be using the entire training set one time, or in other words, the ANN sees the entire training set one time. After each epoch the GD will utilize the whole training set and averaging the desired change making one carefully calculated step towards a local minimum. This

is not ideal for larger training sets with many weights and biases since it is computational slow to do one huge update instead of smaller ones. It is often smart to divide this epoch into smaller subsets or batches. One minibatch is a smaller subset of the training set and every time we have used one minibatch to update the parameters, we have performed one iteration. Each epoch is then the number of iterations times the size of the minibatch: epoch = iterations $\times$ minibatch. When the minibatch has the same size as the epoch, meaning one iteration, then we have GD making one iteration of update per epoch. Minibatch stochastic gradient descent (minibatch SGD) instead uses random (stochastic) minibatches of the training set. Let the whole training set be divided into i minibatches. The minibatch SGD are defined as

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \boldsymbol{\nabla}_{\theta} E_i(\boldsymbol{\theta}). \tag{2.7}$$

Which means that we sum up and average all the training points in one minibatch and makes a small step towards the minimum by updating $\boldsymbol{\theta}$. This means that minibatch SGD makes more updates to $\boldsymbol{\theta}$ then GD and makes an approximate solution but this is good to avoid overfitting. The number of updates minibatch SGD does is the number of iterations defined above.

In this project we will use the Adam optimization algorithm [2]. Adam stands for adaptive moment estimation and is a more complex version of minibatch SGD. In its most simple explanation Adam changes the learning rate η, which would otherwise be a constant in minibatch SGD. Adam adapts/updates the learning rate, during the training, based on previous iterations.

2.4 Data scaling

The weights and biases are initially randomly generated and the optimization algorithm updates these via the loss function. This means that the scale between the real and the prediction value matters. Scaling the data matters in the optimization and implementing scaling transformation for the input and output can help stabilize the training. Examples of scaling your data are normalization, non-linear transformation or standardization. A normalization is where the dataset is normalized to an interval, often $[0, 1]$ such as the min-max normalization

$$\overline{\mathbf{x}}_n = \frac{\mathbf{x} - \mathbf{x}_{\min}}{\mathbf{x}_{\max} - \mathbf{x}_{\min}}, \tag{2.8}$$

where $\mathbf{x}$ is the dataset, $\overline{\mathbf{x}}_n$ is the normalized dataset and $\mathbf{x}_{\min/\max}$ is the minimum/maximum value of that dataset. To see that this normalization works

one can input the boundary points $x = x_{\min}/x_{\max}$ in the equation for which the output will be $\overline{x}_n = 0/1$. Any value of x that is between $[x_{\min}, x_{\max}]$ will then get a normalized value in the interval $[0, 1]$. This type of normalization is good since it keeps the ratio between data points in the dataset after normalization but it does not work as well if you have many potential outliers (datapoints that differ immensely compared to the rest) since they can affect the size of $x_{\min}/x_{\max}$ and in so shift the whole distribution towards that outlier. Standardization on the other hand scales the dataset to a distribution around 0. It is defined as

$$\overline{\mathbf{x}}_s = \frac{\mathbf{x} - \mu}{\sigma} \tag{2.9}$$

where $\overline{\mathbf{x}}_s$ is the standardized dataset, μ is the mean value and σ the standard deviation. If x is the mean value then $\overline{\mathbf{x}}_s = 0$, if x is larger than the mean value then $\overline{\mathbf{x}}_s > 0$, and if x is lower than the mean value then $\overline{\mathbf{x}}_s < 0$. This means that we have scaled the dataset into a distribution centered around zero. The standard deviation looks like

$$\sigma = \sqrt{\frac{\sum(x_i - \mu)^2}{N}} \tag{2.10}$$

where again μ is the mean value and N is the amount of data points in the dataset. The sum measures the difference of each datapoint to the mean and squares it. Standardization also struggles with possible outliers but deals with it better than normalization. The small probability of outliers does not affect the mean values as much as the boundary points of min-max normalization such that the overall standardized distribution is not as affected. Standardization on the other hand does not have a constant interval. This means that the output interval changes with the dataset.

Non-linear transformation such as the log transformation operates the logarithmic function on the dataset. This is useful when you have "skewed" dataset, or a dataset with high variance. Log transformation is useful when one wants to reduce the variance, but still account for the magnitude of change. One could keep the non-linear transformation as the scaled dataset, or use it in a normalization or standardization.

3 Event Generator

Event generators, or Monte Carlo event generators, are essential tools to test theoretical predictions. They create simulated events of high energetic particle collision, making it possible to compare the result to experimental particle colliders. The event generator used to produce all the sample data in this project is the MadGraph5_aMC@NLO [3]. The event generator produces phase-space points. This is a point in the phase-space, for which all possible states of momenta exist. The MadGraph5_aMC@NLO generator allows us to slice this phase-space such that we can control the amount of sample data we want to have for the ANN. MadGraph5_aMC@NLO also gives the corresponding matrix element for each phase-space point, such that we can use it for training the ANN.

3.1 Monte Carlo Method

The cross section or the probability that a process would occur is proportional to

$$\sigma \propto \int |\mathcal{M}|^2 \, \mathrm{d}\Phi(n) \quad \dim\left[\Phi(n)\right] \sim 3n \tag{3.11}$$

where $\mathcal{M}$ is the matrix element. This function is dependent on n, which is the amount of final state particles in the process. This integral may be extremely hard to solve and an exact analytic solution may not exist such that there is a need for a numerical solution.

The basic idea of the Monte Carlo method is to approximate the integral I by using a probabilistic way. The integral become

$$I = \int_{x_1}^{x_2} f(x) \, \mathrm{d}x \approx \frac{V}{N} \sum_{i=1}^{N} f(x_i), \tag{3.12}$$

where V is the integration volume, N the number of sample points and the sum runs over random samplings of the function. The method of approximating the integral in such a way is called a Monte Carlo integration.

3.2 Unweighting

In this project we want to investigate how the ANN can help the efficiency of an event generator. Given the complexity of the matrix element for many final state particles, the computational cost of finding the cross section for that interaction is expensive. An approach to reduce this computational cost is by a method called unweighting. The event generator generates each event with a given weight. This weight is given by the matrix element and the phase-space weight. Having weighted events is undesirable from a view of accuracy and efficiency. In a region where the cross section is small, there would be a large sample of weighted events in that region such that the desired statistical accuracy would be satisfied. This means that there would be a need for a larger sample size to get the total physical distribution. It is also undesirable since nature has unit weight distribution. In small cross sectional regions the nature-like property means that there would be fewer equal weighted events. Extending the previously mentioned Monte Carlo integration where one goes from weighted events to unit equal weighted events is the unweighting method.

The best way to illustrate the classical unweighting method is by looking at the one-dimensional case. Let us try to approximate the integral $I = \int_0^1 f(x)\,\mathrm{d}x$ numerically. The unweighting method works by extending eq. (3.12) such that it will save events with unit weight but still approximate the integral. The classical unweighting procedure can be found in table 1 and in figure 2. See Ref. [4] or [5] for more information about this. The classical unweighting method works by first taking a random point on the x-axis and compute $f(x)$ (black dots in figure 2). Next we take a random point y that is between $0 < y < f_{\max}$. If $f(x) > y$ is true, the point is inside the area and is accepted, whereas if the statement is false we disregard the point (representing the green and red dot, respectively in figure 2). The integral becomes $I =$ accepted/total events times the total surface of the figure, in this case $(x_{\max} - x_{\min}) \cdot f_{\max} = (1 - 0) \cdot 1 = 1$.

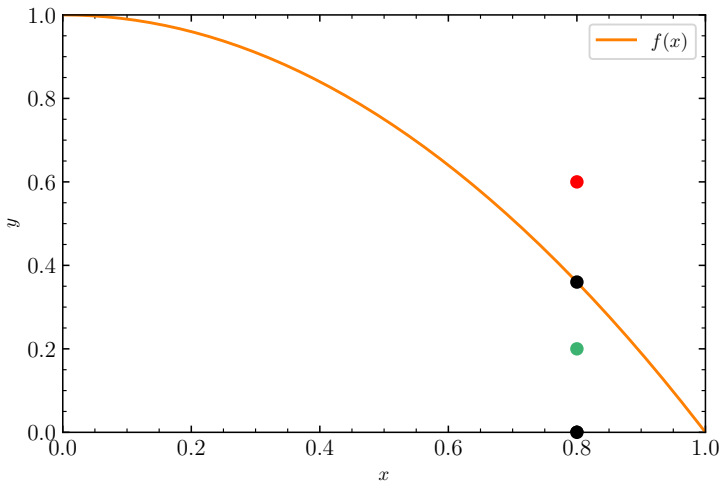

Figure 2: *Example of an one-dimensional unweighting process.*

Table 1: *Procedure for classical unweighting.*

Classic Unweighting
1. Pick x randomly.
2. Compute $f(x)$.
3. Pick random $y = [0, f_{\max}]$.
4. If $f(x) > y$:
accept event with unit weight,
else reject.

The actual time consuming part of this process is the evaluation of the matrix element $f(x)$. In our example in table 1 we can notice that we still need to compute $f(x)$ for those points that are not ending up as our unweighted events. These computations of $f(x)$ is the factor we want to reduce by introducing machine learning into the process. The ANN prediction, that we call $g(x)$, is a much faster computation but an approximation of the matrix element. Using the $g(x)$ in the unweighting method would in theory be a faster method but one cannot assume that the ANN model has a 100% accuracy such that $g(x) = f(x)$. This error or difference between the real function and prediction has to be accounted for. Doing the unweighting method with the prediction $g(x)$, the unweighted events would still be partially weighted and the physics would differ. Thus, the events that survive the unweighting

method would need a weight factor that takes into account this difference between $f(x)$ and $g(x)$. The weight factor has to be $w(x) = f(x)/g(x)$ and to prove this consider the probability of an event surviving times the outcome weight. For the classical unweighting using the matrix element $f(x)$, the probability for each event to survive the comparison of step 4 in table 1 are $P^{\text{cl}} = f(x)/f_{\max}$. The outcome weight of that process is of course unity. Consider instead using the prediction $g(x)$. Comparing the sum of the probability P times the outcome factor r we see that

$$\sum_i P_i^{\text{cl}} \cdot r^{\text{cl}} = \sum_i \frac{f(x_i)}{f_{\max}} \cdot 1 = \sum_i \frac{g(x_i)}{f_{\max}} \cdot \underbrace{\frac{f(x_i)}{g(x_i)}}_{w(x_i)} = \sum_i P_i^{\text{ANN}} \cdot r_i^{\text{ANN}} \tag{3.13}$$

where r_i is the outcome weight factor, and P_i the probability for the classical and ANN unweighting. We can also see from this comparison of sums that the probability of each event to be accepted has to be $g(x_i)/f_{\max}$. It is possible to use $g_{\max}$ instead since

$$\sum_i \frac{g(x_i)}{f_{\max}} \cdot w(x) \cdot \frac{g_{\max}}{g_{\max}} = \sum_i \frac{g(x_i)}{g_{\max}} \cdot w(x) \cdot \frac{g_{\max}}{f_{\max}}. \tag{3.14}$$

Using $g_{\max}$ means that each event has to be multiplied by $g_{\max}/f_{\max}$ to still be correct. Whichever situation one may choose $f_{\max}$ stills needs to be defined. The whole purpose of using the ANN prediction is to reduce the number of times the evaluation of $f(x)$ is computed. If all matrix elements had to be generated such that an $f_{\max}$ can be found the whole process would be redundant. This means that $f_{\max}$ has to be defined in some way outside the actual machine learning unweighting.

Using the prediction $g(x)$ leads to partially weighted events, but one can do a second unweighting such that real unweighted events could be obtained. Both the first and second ANN unweighting method can be seen in table 2 below. The first ANN unweighting works similar to classical unweighting, that is a random x is generated. The prediction $g(x)$ is computed for that point. A random y is generated between $0 < y < f_{\max}$. Then a comparison is being made where we check if $g(x) > y$. If that is the case then that point x is being saved with the weight factor $w(x) = f(x)/g(x)$, else the point is rejected. These weight factors are still partially weighted such that they can be sent through the second ANN unweighting. This method is the exact same as the classical unweighting. This means for every weight factor $w(x)$ there is a randomly generated y between $0 < y < w_{\max}$. Since all weight factors are already obtained from the first unweighting, finding $w_{\max}$ can be found

by iterating through all these weight factors. Having generated a random y, the next step is to make the comparison $w(x) > y$. If this is true, that event is saved with a unit weight and are the final unit unweighting events using machine learning unweighting methods.

Table 2: *Machine Learning Unweighting*

First Unweighting
1. Pick x randomly.
2. Compute $g(x)$.
3. Pick random $y = [0, f_{\max}]$.
4. If $g(x) > y$:
accept with weight $w(x) = \frac{f(x)}{g(x)}$,
else reject it.

Second Unweighting
1. Pick random $y = [0, w_{\max}]$.
2. If $w(x) > y$:
accept event with unit weight,
else reject it.

4 Method

The underlying idea is to build a machine learning algorithm to fit matrix elements from the 3-momenta. We will use this arbitrary interaction: $e^+e^- \to u\bar{u}ggg$ with an centre-of-mass energy of 1 TeV but the setup is general and can be applied to other processes as well. The phase-space points where generated with the same cuts as used in Ref. [6]:

$$\min(s_{ij}/\hat{s}) > 0.01 \tag{4.1}$$

where $s_{ij} = (p_i + p_j)^2$ and $\hat{s}$ is the centre-of-mass energy. That is the minimum relative invariant mass squared of any two final state partons. We decided to remove the energy component of the 4-momentum because of its relation with 3-momenta[1] to reduce redundant factors for the model. The ANN will be built in Python [7] using Keras [8] with TensorFlow [9] backend. As mentioned earlier we will use the MadGraph5_aMC@NLO [3] event generator to generate the sample data for our process. The ANN model will be trained on 100000 phase-space points where 80% are used for training and 20% are for testing. The training data has a validation split of 20% which means that each epoch is training on 64000 points and validating on 16000 points. The input layer will be a 21-dimensional array (7 particles in the interaction times 3 for the momentum in each direction) and output a scalar

[1] $E^2 = p_x^2 + p_y^2 + p_z^2$, where $m \approx 0$

(matrix element). There will be three hidden layers that contain 20-40-20 nodes inspired by Ref. [6]. The activation functions for the hidden layers are either hyperbolic tangent or ReLU, and linear for the output layer. The loss function will be a mean squared error as eq. (2.3) with Adam optimizer. The learning phase will have 200 epochs with an early stop if there is no improvement after 30 epochs. We will scale our dataset by applying the non-linear log transformation on the matrix element (output dataset). The problem with our dataset is that it covers over 5 orders of magnitude, its quite small ($\sim 10^{-15}$) and has a huge variance. Applying a logarithmic function worked the best in our case. We did not scale our input data since the momenta is already between the interval $[-500\,\text{GeV}, 500\,\text{GeV}]$.

To check the accuracy of our ANN code we took inspiration by Ref. [6]. They plot the accuracy by doing a histogram of the logarithmic difference between the predicted and real value. Their histogram is also made by generating several trained models all with the same setup and hyperparameters and then taking the average accuracy of them. This is to ensure a more accurate representation.

The second step of this project is to see if the unweighting method using the trained ANN model is possible and what kind of efficiency we can get. To start off we need a large sample dataset with many weighted events, such that in the end we have at least some unweighted event that we can use to check the accuracy and physical validation. For the classical unweighting we also need the corresponding matrix element $f(x)$ for each event. Once again we use MadGraph5_aMC@NLO [3] to generate a list of 500000 weighted phase-space events for which it also supplies a corresponding matrix element. From the list of matrix elements we can use Python's max() function to get f_{max}. To get the list of y-values we generate a 500000 long list of random numbers between $[0, 1]$ using numpy.random.uniform. The y list then multiplies with the factor f_{max} such that the right interval is obtained. The classical unweighting is then done by iterating through the matrix element and comparing if $f(x) > y$ and save these events that survive.

As was mentioned earlier in section 3.2 using the machine learning unweighting there is a problem of acquiring f_{max}. Since we are comparing classical and machine learning unweighting we already have defined f_{max} from the classical method. In practice this is not how it should be done. f_{max} has to be predefined in some other way. That could for example be by using several smaller sample sizes and taking the average f_{max} of these. One could also use the already generated phase-space points from the training set and find f_{max}, but then it could not be guarantee that there would exist an event such that

$f(x_i) > f_{\max}$ in the actual unweighting method, but that should be quite rare. The process of machine learning unweighting is similar to the classical approach. A list of 500000 prediction $g(x)$ is processed for each generated x. A list of y is defined as the previous method. Once again we iterate through the prediction data and compare $g(x) > y$ with random y values, these events that survives gets saved with the weight factor $w(x) = f(x)/g(x)$.

For the second machine learning unweighting we need to define $w_{\max}$ which once again uses Python's max() function. Furthermore, the same operator as earlier is done to find the y values for the second ANN unweighting but this time we only need to generate the same amount of points as the list of weight factors. The process once again is to check the comparison $w(x) > y$ and these x values that survives is saved and is our final unweighted events using ANN.

Doing the machine learning unweighting would be useless if the actual physics would change compared to the classical approach such that a physical validation is needed. This will be done by three histograms with the unweighted events for both the classical and ANN unweighting. They have to be normalized as the amount of unweighted events will differ between the two methods. The normalization will be done by matplotlib.pyplot.hist own normalization function density=True. Our focus in this project was the interaction $e^+e^- \to u\bar{u}ggg$. These histograms will focus on the up-quark since that particle will always be due to an interaction. The three physical validation histograms are the transverse momentum of the up-quark w.r.t z-axis, invariant mass of the up and anti-up quark, and the pseudorapidity of the up-quark. Eq. (4.2)-(4.4) shows these equations. The histograms are made in Python using matplotlib.pyplot.hist [10].

$$p_T^z = \sqrt{p_x^2 + p_y^2} \tag{4.2}$$

$$m_{u\bar{u}} = \sqrt{(E_u + E_{\overline{u}})^2 - (p_u^x + p_{\overline{u}}^x)^2 - (p_u^y + p_{\overline{u}}^y)^2 - (p_u^z + p_{\overline{u}}^z)^2} \tag{4.3}$$

$$\eta = \operatorname{arctanh}\left(\frac{p_z}{\|\mathbf{p}\|}\right) \tag{4.4}$$

5 Results

The factors and functions of our ANN that we used for the unweighting can be seen in table 3. As mentioned in section 4 the output was normalized by taking the logarithmic function. The training was done with a generated sample dataset of 100000 phase-space points.

Table 3: *The implemented Hyperparameters for the ANN.*

Hyperparameter	Selected values
Hidden Layers	3
Nodes per Layer	$\{20, 40, 20\}$
Activation Function	Tangent Hyperbolic
Optimiser Function	Adam
Loss Function	Mean Squared Error
Epochs	200

In figure 3 the accuracy of this model is shown. Here $\Delta = g(x)/f(x)$ that is the predicted divided by the real matrix element. This distribution is an average of 20 models. That is, we have run the training phase using the same parameters above 20 times and calculate Δ for each model, then taking the average result. One can notice that this distribution is slightly shifted to the right which means that the prediction $g(x)$ is slightly larger than $f(x)$.

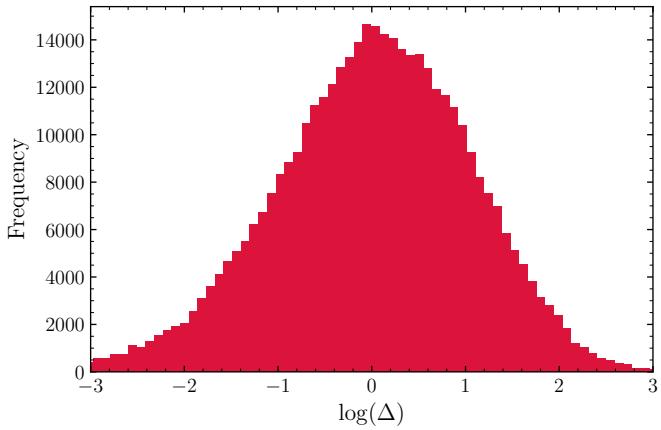

Figure 3: *Representation of our models accuracy, where an average of 20 trained models are used. The output is taken as the logarithmic difference between the real and predicted value.*

For the classical unweighting procedure, the process to evaluate N_{cl} unweighted events are

$$N_{\text{cl}} = N_w \epsilon_{\text{cl}} \quad \Longrightarrow \quad N_w = \frac{N_{\text{cl}}}{\epsilon_{\text{cl}}} \tag{5.1}$$

where N_{cl} is the amount of events that survives the process, N_w is all the weighted events and ϵ_{cl} is the classical unweighting coefficient i.e. the fraction that survives. For the ANN unweighting efficiency, we define two coefficients ϵ_1 and ϵ_2. The ANN unweighting was split up into two processes. For the first ANN unweighting we used the prediction $g(x)$, and these events that survived this process are

$$N_1 = N_w \epsilon_1 \tag{5.2}$$

where ϵ_1 is the fraction that survives the first ANN unweighting. For the second ANN unweighting these events that survived the first ANN unweighting are used and the events that survives both unweighting methods we call N_{ANN}

$$N_{\text{ANN}} = N_1 \epsilon_2 = N_w \epsilon_1 \epsilon_2 \quad \Longrightarrow \quad N_w = \frac{N_{\text{ANN}}}{\epsilon_1 \epsilon_2}. \tag{5.3}$$

Setting these equation to equal we can find the fraction on survived events comparing classical to ANN method

$$\frac{N_{\text{cl}}}{\epsilon_{\text{cl}}} = \frac{N_{\text{ANN}}}{\epsilon_1 \epsilon_2} \quad \Longrightarrow \quad N_{\text{cl}} = \frac{\epsilon_{\text{cl}}}{\epsilon_1 \epsilon_2} N_{\text{ANN}}. \tag{5.4}$$

The coefficients can be seen in table 4 for a run of 500000 phase-space points ($N_w = 500000$).

Table 4: *Coefficient for* 500000 *weighted phase-space points.*

Coefficient	Value	Survived events
ϵ_{cl}	0.0062	3100
ϵ_1	0.003432	1716
ϵ_2	0.08508	146

These coefficients were found by doing the classical and ANN unweighting, counting all the survived events and then using the above equations. The process of doing the unweighting with $N_w = 500000$ took around 2 days on a normal laptop such that there was only one run at the process. Possible improvement on how to get better results will be explained later on but one can see the difference between ϵ_{cl} and ϵ_2. They differ by approximately $0.08508/0.0062 \approx 13.7$. This means that comparing the unweighting with the matrix element $f(x)$ and weighted event $w(x)$, it's 13.7 times more efficient to use $w(x)$. This gives us a good indicator that the first ANN unweighting gives us weighted events that have lower variance, and using these weighted events in a second ANN unweighting is actually making an improvement.

By comparing the number of unweighted events we can notice that they differ by $3100/146 = \epsilon_{\text{cl}}/\epsilon_1\epsilon_2 \approx 21.23$. This means that we need to create approximately 21.23 more weighted events for the ANN unweighting procedure to have as many unweighted events as with the classical procedure.

So right now we have come up with the idea that the classical unweighting produces 21.23 times more unweighting events than the ANN unweighting. This is to be expected since ANN unweighting is doing two unweighting methods. But we have not mentioned time yet. The whole idea of this operation was to reduce the amount of evaluation of the matrix element $f(x)$ since this is the time consuming part. Let t_f be the time it takes to do one evaluation of $f(x)$ and t_g the time to do one ANN prediction. For the classical unweighting we compute $f(x)$ for each weighted event, such that the time to do the unweighting method for N_w events are

$$t_{\text{cl}} = N_w t_f. \tag{5.5}$$

This is the time to produce $N_{\text{cl}} = 3100$ unweighted events. We are interested in finding how long it takes to do the ANN unweighting but still produce the

same amount of 3100 unweighted events. One process of ANN unweighting using $N_w = 500000$ gave us 146 unweighted events such that we need to have a dataset of weighted events that is 21.23 times larger. That is we need to have $21.23N_w$ weighted events such that we can get 3100 unweighted events using ANN unweighting. We also compute $g(x)$ for each weighted event and an additional $f(x)$ for each event that survives the first ANN unweighting when doing the computation of the weight factor $w(x) = f(x)/g(x)$. Accounting for all of this the time it takes to produce the same amount of unweighted events (3100) as the classical method is

$$t_{\text{ANN}} = 21.23(N_w t_g + N_w \epsilon_1 t_f) = 21.23 N_w (t_g + \epsilon_1 t_f). \tag{5.6}$$

This time does not consider any contribution from the second ANN unweighting. That is because in that process no prediction or matrix element are computed, only a comparison is made for precalculated weights and a random generated list y. Inserting the value from table 4 we get

$$\begin{aligned} t_{\text{cl}} &= N_w t_f = 5 \times 10^5 t_f \\ t_{\text{ANN}} &= 21.23 N_w (t_g + \epsilon_1 t_f) \approx 1.06 \times 10^7 t_g + 3.64 \times 10^4 t_f. \end{aligned} \tag{5.7}$$

We can see that the number of evaluations of $f(x)$ is significantly higher in the classical unweighting, this is comforting since this means that a time improvement could be possible. Setting the eq. (5.7) equal to each other we get the break-even point of efficiency

$$\begin{aligned} 5 \times 10^5 t_f &= 1.06 \times 10^7 t_g + 3.64 \times 10^4 t_f \\ t_f &= 22.9 t_g. \end{aligned} \tag{5.8}$$

This means that the ANN model is more efficient if the time to compute one $f(x)$ is 22.9 times longer than one prediction $g(x)$. The ANN training time is disregarded in these calculations since its time was significantly slower compared to the time of the unweighting process. The break-even factor critically depends on the scattering process under consideration and the accuracy of the ANN prediction. Using only 5 final state particles it is not efficient to use ANN unweighting, but the factor 22.9 might not be unreasonable. The models prediction does not scale in times as much as the evaluation of the matrix element, such there would be an interaction where ANN is useful.

The physical validation comparing both processes are presented in three different histograms. The amount of surviving points are 3100 and 146 for the classical and ANN unweighting respectively from 500000 weighted events, this means that the plots are normalized using matplotlib.pyplot.hist density function. For the histogram of transverse momentum and the pseudorapidity the up quark in the interaction $e^+e^- \to u\bar{u}ggg$ is used. The invariant mass is however plotted using the sum of the up quark momentum and the anti-up quark momentum.

In figure 4 the transverse momentum defined as eq. (4.2) are summed in each bin with a total of 25 bins, they are then normalized such that we have an arbitrary y-axis of frequency. This is the same for all three figures. The given peaks of the ANN unweighting may be caused by possible outliers and are probably statistical fluctuations. Within the large statistical fluctuations, the classical and ANN unweighting agree; however, more unweighted events are needed to confirm these findings at higher precision. This was not done due to computation time of the unweighting process.

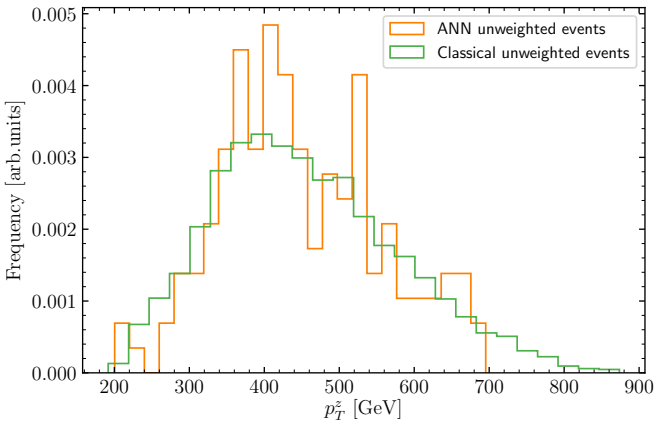

Figure 4: *Histogram of transverse momentum of the up quark comparing the Classical and ANN unweighting procedure.*

In figure 5 the invariant masses $m_{u\overline{u}}$ are shown from up and anti-up quark momentum, defined from eq. (4.3).

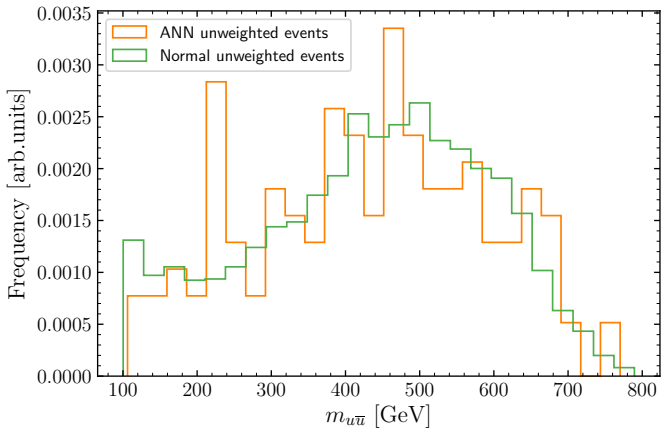

Figure 5: *Histogram of invariant mass of the up and anti-up quark, comparing the Classical and ANN unweighting procedure.*

In figure 6 the pseudorapidity is shown defined from eq. (4.4).

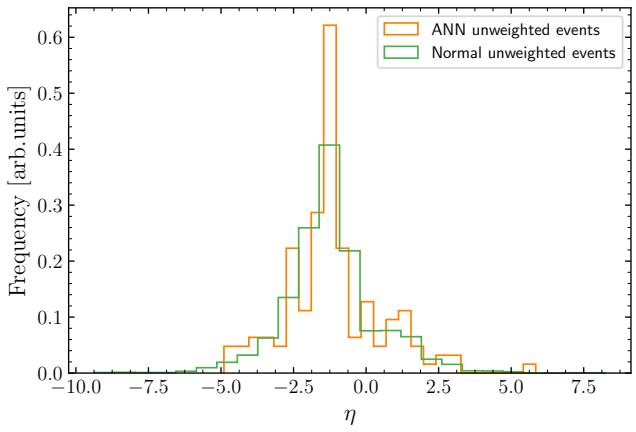

Figure 6: *Histogram of pseudorapidity of the up quark comparing the Classical and ANN unweighting procedure.*

One possible improvement in the amount of unweighted events that one could produce with the ANN unweighting is by changing the $w_{\max}$. When doing the ANN unweighting procedure, the first method is to use the predicted values $g(x)$ and those event that survives get accepted with a weight $w(x) = f(x)/g(x)$. $w_{\max}$ is then computed from this list of accepted events, but $w_{\max}$ will most likely be an outlier. A much higher $w_{\max}$ than the rest of the $w(x)$ list would lead to fewer events surviving the second ANN unweighting and thus producing less unweighted events. Another approach would be to generate several datasets with random sequences and take the mean of $w_{\max}$. This approach does not guarantee that $w_{\max}$ is the absolute maximum value of w, which means it would be partially weighted. To still achieve the same statistical accuracy as unit weighted events, there may be a need to generate more events. The time to do the classical and ANN unweighting for 500k phase-space points took about two days to run on a standard laptop, so due to lack of time this was not implemented for this project.

6 Conclusion

In theoretical particle physics event generators are essential tools to test theories and predictions. These so-called event generators strive to be more accurate and efficient. This leads to increasing CPU expense.

This thesis investigated the use of artificial neural networks to optimize an event generator. A regression type artificial neural network was constructed to evaluate the cross section for the interaction $e^-e^+ \to u\bar{u}ggg$. The regression model used the 3-momenta of all particles in the interaction and made a fit to the matrix element. This model could then be used to make predictions on a new set of momenta.

The idea with the regression model was to use the predicted matrix element $g(x)$ to reduce the amount of times the evaluation of the real matrix element $f(x)$ was done, since computing the real matrix element is a time consuming task. The implementation of the model was used in the process of going from weighted to unweighted events called the "unweighting" method. A classical and ANN unweighting method was investigated and an efficiency in factors of $f(x)$ and $g(x)$ was measured.

The break-even point of efficiency was measured to be a factor of 22.9, that is for our interaction the computing time for the matrix element has to be 22.9 times longer than the model's prediction for it to be more efficient to use the ANN unweighting. Given the complexity of computing the matrix element for more final state particles, we do not think this is a bad result but there are ways of improvement, e.g. developing a more accurate model, using the average of $w_{\max}$ or taking more sample data for the model's training phase.

Implementing this into the MadGraph5_aMC@NLO [3] event generator is an interesting future project. Other methods like importance sampling where the ANN could be useful may also be an interesting addition. It would also be interesting to see if using for example the pseudorapidity or invariant mass for the models training would be more efficient instead of the 3-momenta. One could also try out different y_{cut} when generating phase-space points.

Acknowledgments

First of all thanks to my supervisor Rikkert Frederix for his patience and inputs, I will always cherish our zoom coding sessions. I would also like to thank my friends, especially my study group (zångo') for always making studying fun. A special thanks to my friend Oskar for all his mental support. Thanks to my brother Simon for proof reading my popular science text. Finally I would like to thank my parents for always being there.

References

[1] "LHC Machine." In: *JINST* 3 (2008). Ed. by Lyndon Evans and Philip Bryant, S08001. DOI: 10.1088/1748-0221/3/08/S08001.

[2] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[3] J. Alwall et al. "The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations." In: *JHEP* 07 (2014), p. 079. DOI: 10.1007/JHEP07(2014)079. arXiv: 1405.0301 [hep-ph].

[4] Katharina Danziger. "Efficiency Improvements in Monte Carlo Algorithms for High-Multiplicity Processes." Presented 31 Mar 2020. 2020. URL: http://cds.cern.ch/record/2715727.

[5] Johannes Krause. "Efficiency Improvements Using Machine Learning in Event Generators for the LHC." 2015. URL: https://iktp.tu-dresden.de/IKTP/pub/15/masterthesis-2015-10-johannes-krause.pdf.

[6] Simon Badger and Joseph Bullock. "Using neural networks for efficient evaluation of high multiplicity scattering amplitudes." In: *Journal of High Energy Physics* 2020.6 (June 2020). ISSN: 1029-8479. DOI: 10.1007/jhep06(2020)114. URL: http://dx.doi.org/10.1007/JHEP06(2020)114.

[7] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.

[8] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras.

[9] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: 1603.04467 [cs.DC].

[10] J. D. Hunter. "Matplotlib: A 2D graphics environment." In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.