Machine learning techniques applied to Monte-Carlo integration

Léo Roche, Supervised by Ben Ruijl and Valentin Hirschi for HEP Proseminar (ETH Zürich)

May 2019

Contents

Abstract

Machine Learning is computational method invented in 1959 by Arthur Samuel, a pioneer in the field of game computing and artificial intelligence [1]. It is nowadays a very famous techniques because we have now an easier access to a big computational power compared to the one existing in the 60's for example. Machine Learning is used in a lot of field for example data science, deep learning, etc... In this study, we will begin to define precisely what a neural network is. Then we use machine learning to train our neural network to compute as precisely as possible matrix element of particle physics process for a given point in phase space. Finally we will try to use these neural network approximation to implement a variant of the Monte Carlo algorithm using presampling.

1 Introduction

1.1 Introduction to Machine Learning

A neural network is a computational system that takes as input a vector x of dimension n_{input} and that calculates an output vector $\mathrm{NN}(x)$ (in our case a scalar) where NN is the action of the neural network on the input variable. The goal of machine learning is that we train the neural network so that the neural network replicates a specific function or classifies data in categories. In our case we want to replicate a function, i.e. we want $\mathrm{NN}(x) = f(x)$ for a specific function f. A neural network is composed of different layers of neurons (see fig. 1). A neurons is a cell which has one scalar value, which is also taken into account to compute the neurons within next layer. The first layer is the *input layer*, which is composed of n_{input} neurons is actually encoding the value of x, the input vector to evaluate. During a calculation, each neuron has a scalar value. Then after we have the *hidden layers*. Finally the *output layer* of dimension n_{output} gives us the answer. The machinery behind the calculation of the value of each neurons in the output layer is that : each neuron value is determined by the value of the neurons from the previous layer pondered by the specific weights (intrinsic values for each neuron) plus a constant (the bias), and everything is then evaluated by a specific activation function. Let's note $x^{i,j}$ the value of the ith neuron on the jth layer. If $\{x^{k,j-1}\}_k$ is then the set of the values of the previous layer, then the value of $x^{i,j}$ is:

$$x^{i,j} = \sigma\left(\sum_k \omega_k^{i,j} x^{k,j-1} + b^{i,j}\right) \tag{1}$$

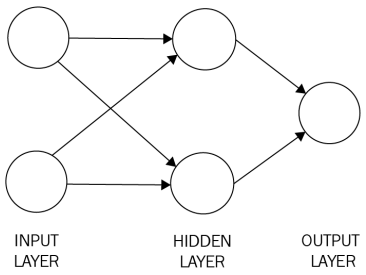

Figure 1: Simplest neural network graphical representation [4]. Neurons are represented by circles and layers of neurons are represented vertically. Arrows indicate the use of the neurons for next layer's calculation.

Where the set $\{\omega_k^{i,j}\}_k$ is the set of the weights used during the calculation of the neuron at the $(i,\ j)$ position. $b^{i,j}$ is the bias, a constant associated to this neuron. Let's call the ensemble of parameters (weights and biases) of a neural network W. The function σ is the *activation function*. There exists a lot of different activation functions. In this study, we will use the *ELU* function for no particular reason (see fig. 2) : $\mathrm{ELU}(x) = \max(0, x) + \min(0, \alpha(\exp x - 1))$. The purpose of the activation function is to makes it possible to reproduce non-linear function, because activation functions are usually not linear.

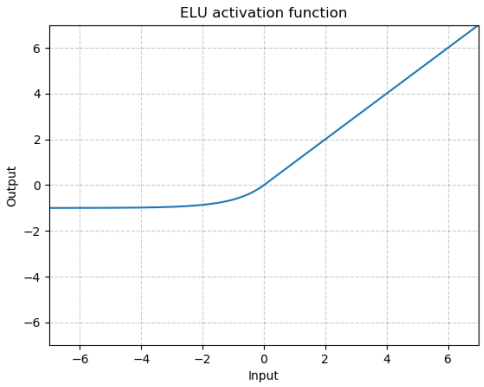

Figure 2: Graphic representation of the ELU activation function.

The purpose of the training of a NN is that every weight and every bias are tuned so that the behaviour of the NN gets as close as possible to the specific function $f(x)$. In order too train the NN, we feed it a

consequent amount of data called batches : inputs with the correct associated outputs. More formally, if x_a is the ath input, let's note $y_a = f(x_a)$ the desired output, then we will feed the neural network a set of $X = \{x_1, x_2, ..., x_{n-1}, x_n\}$ and $Y = \{y_1, y_2, ..., y_{n-1}, y_n\}$ for a n-points training batch.

The notion of distance between the current NN and the wanted behaviour of the NN is described by the *loss function*. There are different ways to quantify this distance, the error. A common way is to compute the mean value of the absolute difference for each points of the testing batch:

$$L(W) = \frac{1}{a} \sum_{k=1}^{a} |\mathrm{NN}_W(x_k) - y_k| \tag{2}$$

The aim of the training protocol is then to find the right parameters W_{min} such that $L(W_{min})$ is a global minimum of $L(W)$. There exist different algorithms which can do it. Some of them are more efficient in some cases. One of the most famous algorithm is the gradient descent. The idea is that the gradient of the loss function $\nabla_W L(W)$ is calculated. The values of W are then changed such that the point "goes down", i.e. $W_{new} = W_{old} - \beta \nabla_W L(W)|_{W=W_{old}}$. Where β is the learning rate of the training process. The bigger the learning rate is, the bigger the step in the W space are made during each training iteration.

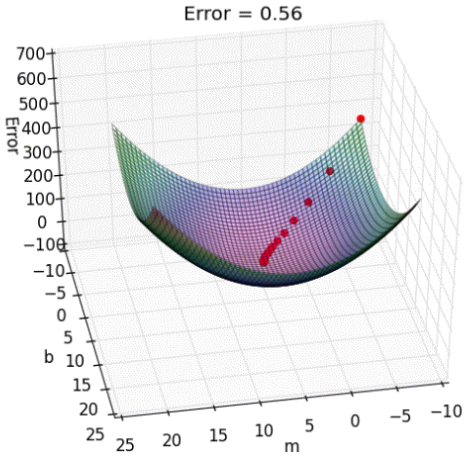

Figure 3: Concept of the gradient descent algorithm for a loss function with 2 parameters (linear regression). [5].

After each training iteration, it is important to test the neural network. In order to test it, we just have to calculate the loss function again for another set of points.

One important thing is to separate the data used to train the NN and the data used to test the NN. We have generated two different batch files of not necessarily the same size. From one batch we will use every point to test the NN after every training iterations, the same file is always used. The other file is the training batch (typically bigger than the testing batch) which is then divided into $N \in \mathbb{N}$ mini batches. The idea is to use a different mini batch at each training iteration so that we avoid to train the NN with the same point during two consecutive training iterations.

The purpose of our neural network is to give a good approximation of the matrix element of the process for one specific point in the phase space. This induces that the output layer will be composed of only one neuron. As input we will try several variable input configurations in order to see how it influences the learning process of the NN. The most basic input variable we can think of are the 4-momentum of every external particles of the process. One can also try to use as input variable the Mandelstam variables (squared sum of 4-momenta). In between the input and the output layer, the NN is composed of hidden layers. We will also train the NN with different amount of hidden layers.

1.2 Implementation

The implementation of the neural network has been done using the framework *TensorFlow* [6] used with Python. The whole architecture of the neural network is created with TensorFlow through these Python lines :

```
layer = X
    for i in range(0, n_hidden_layers+1):
        layer = tf.layers.dense(layer, n_neurons_hidden_layer, activation=tf.nn.elu)
    output_layer = tf.layers.dense(layer, n_output, activation=tf.nn.elu)
```

`X` is the input layer. The function `tf.layers.dense` , from TensorFlow creates a new fully connected layer with the last neuron layer created. A dense neural network is a network where each neurons in the hidden layers are connected to every neurons of the previous layer. The first argument of the function is the previous layer in question, the second argument is the number of neurons of the new layer we want to create and the last one is the activation used during the calculation of the values of the neurons in the newly created layer (we used the ELU function as explained before). Each time a new layer is created this way, weights and biases are also created and managed within TensorFlow.

1.3 Relation to existing works

In this subsection we will look at existing works which are exploring the different possibilities of using neural network to calculate the cross section of more or less complicated elementary particle physics processes.

1.3.1 Neural Network-Based Approach to Phase Space Integration [2]

In the work of Matthew D. Klimek and Maxim Perelstein, *Neural Network-Based Approach to Phase Space Integration* [2], they explore the way of using a neural network based on Monte Carlo integration method directly on particle physics problems. In particular they look at the integration of probability density function where there are some resonances which theoretically are harder to take into account using a classic Monte Carlo method. The philosophy of their NN-event generation is that the NN learns to map uniformly sampled input space of dimension d (unit-hypercube) $p_x(\mathbf{x}) = 1$ where $\mathbf{x} \in \mathbb{R}^d$ to a non-trivial probability density function $p_y(\mathbf{y_w}(\mathbf{x})) = |\nabla_x \mathbf{y_w}(\mathbf{x})|^{-1}$ where $\mathbf{y_w}(\mathbf{x})$ is the NN-map. The ambition of this procedure is that it catches the resonances of the probability density function in a more productive way than some classical Monte Carlo integration method directly applied on the probability density function (e.g. VEGAS). They quantify the distance between the probability density function induced by the NN and the true probability density function $f(\mathbf{y})$ - which is precisely the fully differentiable cross section of the considered process - by the following loss function:

$$L(\mathbf{w}) = D_{KL}(p_y(\mathbf{y_w}(\mathbf{x})); f(\mathbf{y_w}(\mathbf{x}))) \tag{3}$$

Which is precisely the Kullbeck-Leibler (KL) divergence function:

$$D_{KL}(p_y(\mathbf{y}), f(\mathbf{y})) = \int p_y(\mathbf{y}) \log \frac{p_y(\mathbf{y})}{f(\mathbf{y})} d\mathbf{y} \tag{4}$$

Their NN architecture is a dense NN with the same number of neurons in the input and the output layer both equal to the phase space dimension. They use the ELU function as activation function and the *soft-clipping function* as output function :

$$SC_p(x) = \frac{1}{p} \log \left(\frac{1 + e^{px}}{1 + e^{p(x+1)}} \right) \tag{5}$$

They have to transform the phase space coordinate so that they lie in the unit-hypercube, i.e. that each transformed coordinate $\in [0, 1]$. For N particle in the final state of the process, they form $N - 2$ subsets: $\{1, ..., N-1\}, ..., \{1, 2\}$. For each subset corresponds the invariant mass, for instance if we consider the subset $\{1, ..., N-k\}$ the invariant mass then lies in the range:

$$m_{1,...,N-k} \in \left(\sum_{i=1}^{N-k} m_i, \sqrt{s} - \sum_{i=N-k+1}^{n} m_i \right) \tag{6}$$

Then they just have to normalise this interval so that it lies into the $[0, 1]$. Added to that, they use the angles θ_k between the total momenta of the particles in the center of mass frame. These angles are also normalized in order to fit into the hypercube.

The first process they study is the 3-body decomposition of a scalar ($X \to 1 + 2 + 3$) where the matrix element is constant. Then they studied a process which implies a resonance in the 3-body decay of either the form ($X \to 1 + (Y \to 2 + 3)$) or either ($X \to 3 + (Y \to 1 + 2)$). Depending on the phase space coordinate they use for this process, the output coordinate can be either aligned with the resonance or not. In both cases, the NN adapts and it works. The last process studied in their work is the 3-body decay with 2 intermediate resonances. This time it is impossible to align both of the resonances to the coordinates. They chose to treat the two resonances at the same time rather than doing it separately with two channels. The results are apparently quite good. During the study of these 3 processes, they also studied the impact of the architecture of the NN on the quality of the results. They see according to their results that increasing the number of nodes doesn't decrease the final loss value but that it accelerates the learning process.

We will also study the architecture of the NN used in this very study. We will also test our NN implementation on several processes in order to try to determine it limits.

1.3.2 Efficient Monte Carlo Integration Using Boosted Decision Trees and Generative Deep Neural Networks [3]

In his work, Joshua Bendavid explores the way of using Boosted Decision Tree and Deep Neural Networks in order to improve Monte Carlo integration. From these two approaches, the Generative Deep Neural Network Integration is the closest thing of our following work. The principle is the following: some data is sampled from a simple probability distribution $p(\bar{z})$. Then a generative model G (based on a deep neural network, i.e. a neural network with a lot of hidden layers) takes samples of $\bar{z}$ to turn it into the output $\bar{x}$. In the case of the Monte Carlo integration, it is important that they both have the same dimension. Those two probability distributions are linked by the following relation:

$$p(\bar{z}) = p_g(\bar{x}) || \frac{\partial \bar{G}(\bar{z})}{\partial \bar{z}} || \tag{7}$$

The aim of the deep neural network is that $p_g(\bar{x})$ is as close as possible to $p_f(\bar{x})$, where $p_f(\bar{x})$ is the target probability density function already normalized. The error between $p_f(\bar{x})$ and $g(\bar{\bar{x}})$ is also calculated using the KL-divergence: $D_{KL} = \int g(\bar{x}) \log \frac{p_g(\bar{x})}{p_f(\bar{x})} d\bar{x}$. Once the training of the deep neural neural network is done, they numerically integrate the target phase space points by accessing the prior probability density function $p(\bar{x})$ and transforming it with $G(\bar{z})$. As an example/test of their method and implementation, they train the NN to replicate the 1-dimensional Cauchy distribution where $f(\bar{x}) = \frac{1}{\pi} \frac{\Gamma}{\left(x - \frac{1}{2}\right)^2 + \Gamma^2}$ for $x \in [0, 1]$. For this precise case, the analytic expression of what G should be is known. They found very good results; the generative model fits perfectly with the expected analytic solution.

Then they tested their model on a multidimensional camel function where we don't know the analytic solution of what G should be in order to replicate exactly the desired function. By doing the integration of this function in several number of dimensions they find out that their method (Deep Neural Network approach) completely overcomes classical integration using VEGAS or even Foam in terms of low variance value of the integral calculation.

We will see in this study that we will face more or less the same challenges as those from these two works. More precisely the aim is to explore a way of using a neural network to overcome classic integration performances like for instance VEGAS or Foam, in terms of precision of the final integral result and even of calculation time if possible. We will also have to face the degrees of freedom of choosing what could be the optimal NN architecture for specific problems. One particular aspect of this study is that we will try to determine the consequences of adding some redundant data as input of the NN.

2 Process 1 : $[e^+ + e^- \to t + \bar{t}]$

The beam energy in the center mass frame is 500 GeV for following processes (it will considered as constant for all processes). The mass of the Z particle used is 91.188 GeV and the mass of the top is 173 GeV.

In the first part, we will study how a neural network can calculate the matrix element of a simple process:

$$e^+ + e^- \to t + \bar{t} \tag{8}$$

The Feynman diagram of the process is :

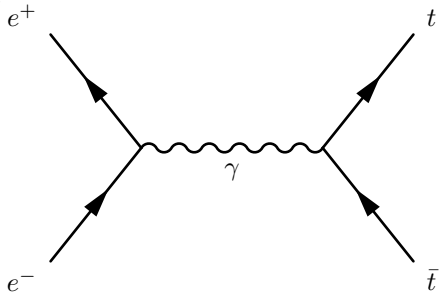

Figure 4: Feymann diagram of process I : $e^+ + e^- \to t + \bar{t}$

2.1 Phase-space parametrisation

It is important to parametrise the phase-space of the considered process in order to reduce the number of variables we use to integrale on the whole phase-space. The integral over the phase-space of a process with n final particles is:

$$\int d\Pi_n = \left(\prod_f \int \frac{d^3 p_f}{(2\pi)^3} \frac{1}{2E_f} \right) (2\pi)^4 \delta^{(4)} \left[P - \sum p_f \right] \tag{9}$$

Where P is the total initial 4-momentum and p_f is the f^{th} 4-momentum of the external particles in the final state. In our case, we have 2 final external particles. Moreover, these particles are massive, thus we should multiply the previous expression by $\delta[p_i^2 - m_i^2]$ for $i = 1, 2$ where p_i is the 4-momenta of the ith external particle (the two tops). Therefore the integral over phase-space become:

$$\int d\Pi_2 = \int \frac{d^3 p_1 d^3 p_2}{16\pi^2 E_1 E_2} \delta^{(4)} \left[P - p_1 - p_2 \right] \delta \left[p_1^2 - m_1^2 \right] \delta \left[p_2^2 - m_2^2 \right] \tag{10}$$

We put ourselves in the center of mass frame: ($E = E_{cm}$ and $P = 0$) and we split the 4-dimensional delta distribution: $\delta^{(4)} \left[P - p_1 - p_2 \right] = \delta \left[E_{cm} - E_1 - E_1 \right] \delta^{(3)} \left[0 - \mathbf{p_1} - \mathbf{p_2} \right]$. By integrating over $\mathbf{p_2}$, we have $\mathbf{p_2} = -\mathbf{p_1}$. Moreover by taking into account the two deltas of the mass invariance condition of eq. 10, we have $E_1 = \sqrt{\mathbf{p_1}^2 + m_1^2}$ and $E_2 = \sqrt{\mathbf{p_2}^2 + m_2^2} = \sqrt{\mathbf{p_1}^2 + m_2^2}$. We now write the expression of the integral over $d^3 p_1$ in spherical coordinates :

$$\int d\Pi_2 = \int \frac{dp_1 d\Omega \; p_1^2}{16\pi^2 E_1 E_2} \delta \left[f(p_1) \right] \tag{11}$$

Where $f(p_1) = E_{cm} - E_1(p_1) - E_2(p_1) = E_{cm} - \sqrt{p_1^1 + m_1^2} - \sqrt{p_1^1 + m_2^2}$. Let's label p_1' the point which satisfies $f(p_1') = 0$. Then we use delta distribution property:

$$\delta \left[f(p_1) \right] = \frac{\delta[p_1 - p_1']}{\left| \frac{df(p_1)}{dp_1} \right|_{(p_1 = p_1')}} \tag{12}$$

We find that the denominator of eq. 12 is equal to $\frac{p_1'}{E_1(p_1')} + \frac{p_1'}{E_2(p_1')}$. The phase-space integral is now:

$$\int d\Pi_2 = \int \frac{dp_1 d\Omega \, p_1^2}{16\pi^2 E_1 E_2} \delta \left[p_1 - p_1' \right] \left(\frac{p_1'}{E_1(p_1')} + \frac{p_1'}{E_2(p_1')} \right)^{-1}. \tag{13}$$

We now integrate p_1 and relabel p_1' with p_1 for notation convenience:

$$\int d\Pi_2 = \int \frac{d\Omega}{16\pi^2 E_1 E_2} \frac{p_1^2}{\frac{p_1}{E_1} + \frac{p_1}{E_2}} = \int \frac{d\Omega}{16\pi^2} \frac{p_1^2}{p_1 (E_2 + E_1)} = \int \frac{d\Omega}{16\pi^2} \frac{p_1}{E_{cm}} \tag{14}$$

Note that $d\Omega$ is the angular element of the spherical coordinate, i.e. $d\Omega = \sin(\theta)\, d\theta d\phi$.

$$\int d\Pi_2 = \int \frac{d\theta d\phi \, \sin(\theta)}{16\pi^2} \frac{p_1}{E_{cm}} \tag{15}$$

For this process, we have a symmetry around the collision axis i.e. $\int_0^{2\pi} d\phi = 2\pi$:

$$\int d\Pi_2 = \int \frac{d\theta \, \sin(\theta)}{8\pi} \frac{p_1}{E_{cm}} = \int \frac{d\cos(\theta)}{8\pi} \frac{p_1}{E_{cm}} \tag{16}$$

Finally we find that for this process the phase-space has been reduced to a 2-dimensional phase-space ($\cos\theta$[1], E_{cm}).

Note that we can also find the number of free variable of the process by counting the number of constraints in the process: By default, each external particle has 4 variables, hence we have start with 16 variables. Taking into account the conservation of the total input and output momenta reduces this number to 12. Then we have the fact that each particle's mass is invariant which mean that we have 4 additional equations. This reduces the number of free variables to 8. Now if we consider that the two incoming particles are in the center mass frame, the 8 variables of the two 4-momenta reduces to 1 variable: we lost again 7 variables. That means that if the energy is not fixed; we have a total of 2 free variables. This variables correspond to the flat inputs. There are different ways to general these two variable, more precisely isomorphisms are here to link these set of variables.

The two variables we use are said to be *flat* because the Jacobian of the isomorphism creating these variables in the MadGraph phase-space generator is simply 1 and because they are generated randomly according to a uniform probability distribution.

2.2 Theoretical matrix element

In order to calculate the analytic expression of the matrix element of this process, we put ourselves in the center of mass frame (fig 5). p is the 4-momentum of the electron, p' is the 4-momentum of the position, k is the 4-momentum of the top and k' is the 4-momentum of the anti-top.

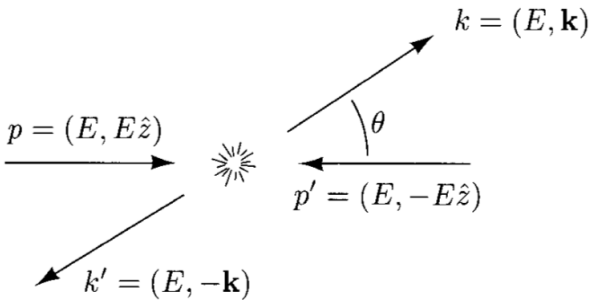

Figure 5: Kinematic description of process I in the center of mass frame. [7]

By using the Feynman rules of QED, the trace theorem and by saying that $|\mathbf{k}| = \sqrt{E^2 - m_t^2}$ and $\mathbf{k} \cdot \hat{z} = |\mathbf{k}| \cdot \cos\theta$. In this process, according to the QED laws, there is no divergence. In fact the expression of the matrix element of this process is:

$$|M|^2 = \frac{1}{4} \sum_{spins} |M|^2 = \frac{4}{9} e^4 \left(\left(1 + \frac{m_t^2}{E^2} \right) + \left(1 - \frac{m_t^2}{E^2} \right) \cos^2\theta \right) \tag{17}$$

Where m_t is the mass of the top. We plot the matrix element in function of θ on fig. 6.

[1]Recall that θ is the angle between the collision axis in the center mass frame and the direction of the first particle of the final state.

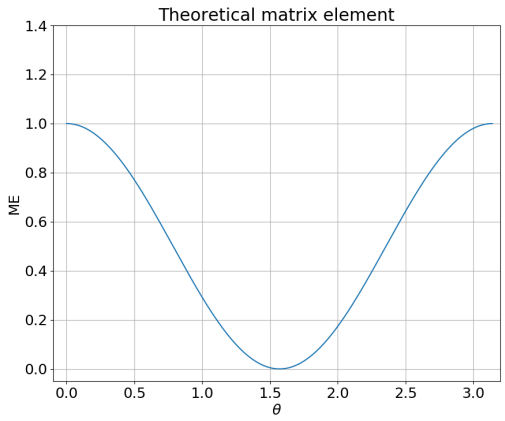

Figure 6: Theoretical matrix element in terms of the angle θ

Indeed, the matrix element in this process only depends on the angle between the incoming and outgoing particles (and the energy obviously).

2.3 Results

The training batch is composed of 10^5 points and the mini batches of 10^3 points. One training iteration is when the NN learn on one mini batch. We have the following results of the evolution of the loss function where we trained the NN with 3 different types of input variables (see. fig 7).

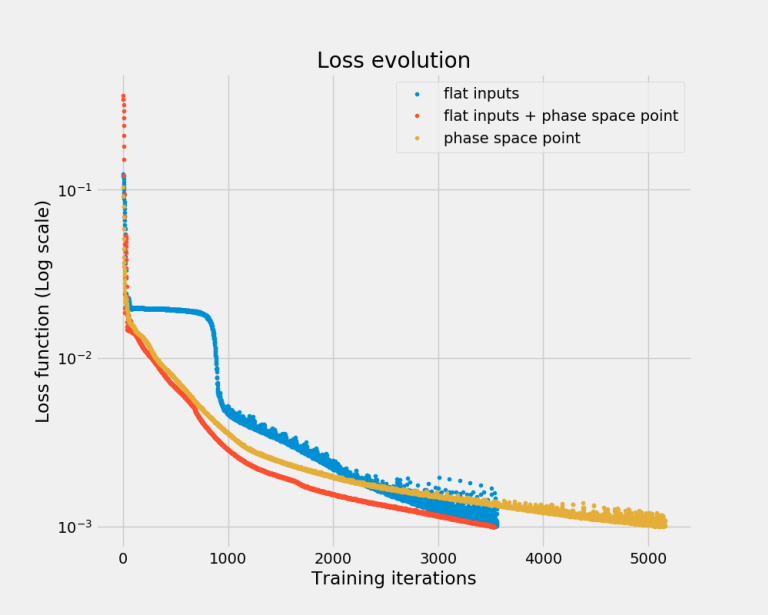

Figure 7: Evolution of the loss function in term of the number of iteration using mini batches, with as input: the 4-momenta, the flat inputs and both.

We see that for this specific case, using recurrent variables (all 4-momentum) make the training process slower in terms of number of training batches necessary. Figure 8 shows the evolution of the loss function for different numbers of hidden layers.

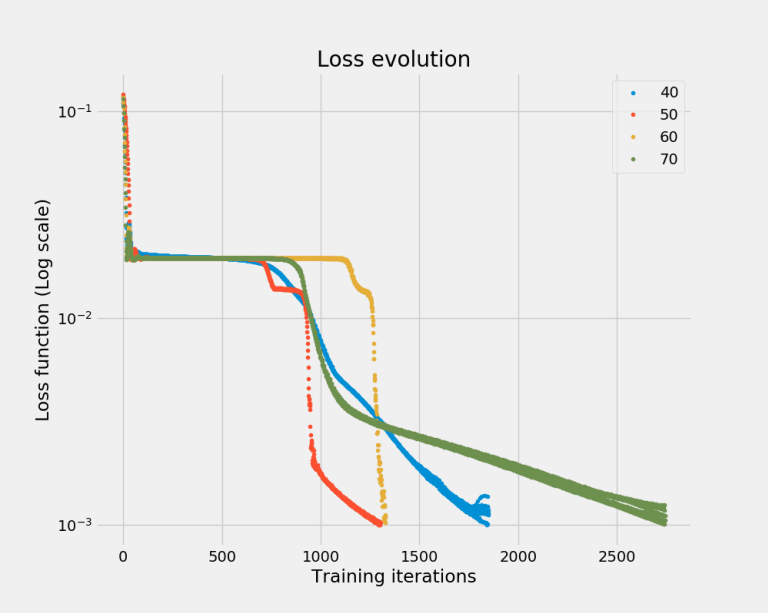

Figure 8: Study of the evolution of the number of hidden layer for training the NN on process I with flat inputs.

This result suggest that something is blocking the learning process for some iterations -hence the pateau- until approximately the 1000th training iteration. What we could have expected is that the training process is faster for a NN with more hidden layers, but according to this result, it is not necessarily what happens.

3 Process 2 : $[e^+ + e^- \to t + \bar{t} + z]$

We will now study how a neural network can approximate the matrix element of more complex process:

$$e^+ + e^- \to t + \bar{t} + z \tag{18}$$

This process has 9 different Feynman diagrams of order 3 (see fig. 9). We won't calculate the explicit expression of the matrix element.

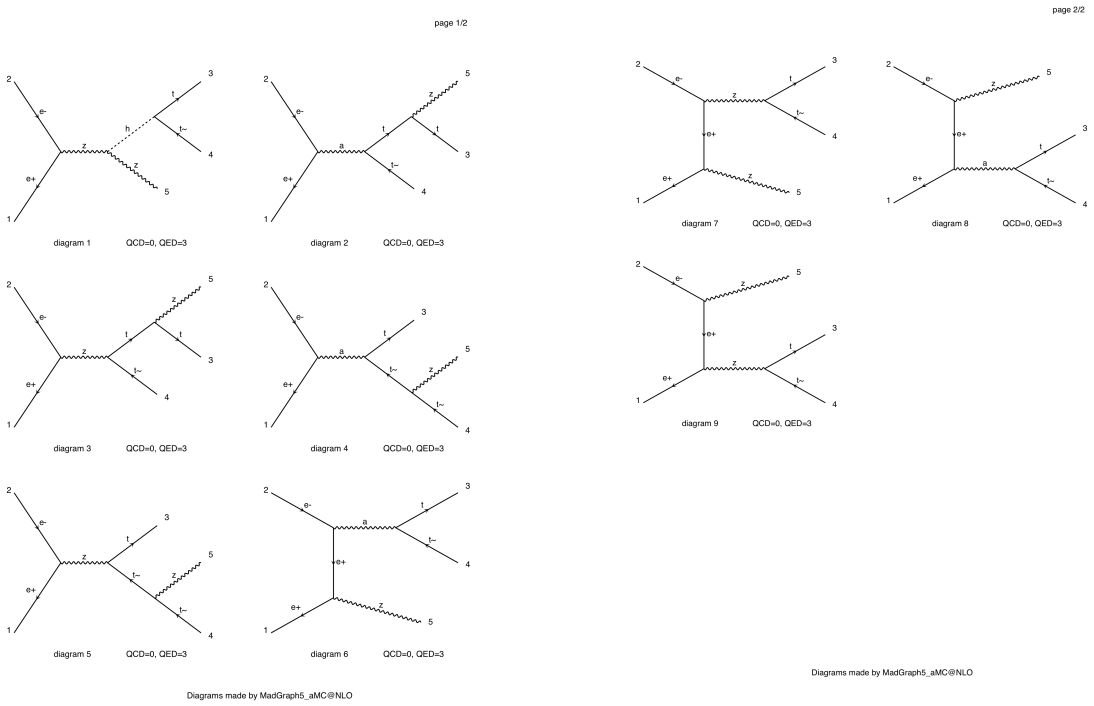

Figure 9: Feynman diagrams of the process 2 generated by MadGraph [9]

3.1 Phase-space parametrisation

We parametrise the phase-space in the same way as for process 1 but for a three-body process. Starting from eq. 9 applied for 3 final massive particles and by considering the center of mass frame, we find:

$$\int d\Pi_3 = \int \frac{d^3p_1 d^3p_2 d^3p_3}{8(2\pi)^2 E_1 E_2 E_3} \delta^{(4)}\left[E_{cm} - E_1 - E_2 - E_3\right] \delta\left[\mathbf{p_1} + \mathbf{p_2} + \mathbf{p_3}\right] \delta[p_1^2 - m_1^2] \delta[p_2^2 - m_2^2] \delta[p_3^2 - m_3^2] \quad (19)$$

We use the second Dirac distribution and integrate over d^3p_3, which implies that $\mathbf{p_3} = -\mathbf{p_1} - \mathbf{p_2}$. Then we take into account the three last Dirac distributions which restrict the energies to be: $E_1 = \sqrt{p_1^2 + m_1^2}$, $E_2 = \sqrt{p_2^2 + m_2^2}$ and $E_3 = \sqrt{p_3^2 + m_3^2} = \sqrt{(p_1 + p_2)^2 + m_3^2} = \sqrt{p_1^2 + p_2^2 + 2p_1p_2 + m_3}$. The phase-space integral is now:

$$\int d^3p_1 \int \frac{dp_2\, p_2^2 d\Omega}{8(2\pi)^5 E_1 E_2 E_3} \delta[E_{cm} - E_1 - E_2 - E_3] \quad (20)$$

We use the spherical coordinates for p_2. We now consider that the argument of the remaining Dirac distribution is a function of $p_2 = |\mathbf{p_2}|$, i.e.:

$$f(p_2) = E_{cm} - E_1 - E_2 - E_3 = E_{cm} - E_1 - \sqrt{p_2^2 + m_2^2} - \sqrt{p_1^2 + p_2^2 + 2p_1p_2 + m_3} \quad (21)$$

In order to use the identity of the Dirac distribution (eq.12), we calculate the derivative of the previous function:

$$\left|\frac{df(p_2)}{dp_2}\right| = \frac{p_2}{E_2} + \frac{p_2 + p_1}{E_3} \quad (22)$$

The integral is then:

$$\int d^3p_1 \int \frac{dp_2\, p_2^2 d\Omega}{8(2\pi)^5 E_1 E_2 E_3} \delta[p_2 - p_2'] \left(\frac{p_2'}{E_2} + \frac{p_2' + p_1}{E_3}\right)^{-1} \quad (23)$$

We integrate on dp_2 and relabel the p_2' by p_2:

$$\int \frac{d^3p_1\, d\Omega}{8(2\pi)^5} \frac{p_2^2}{E_1 E_3 p_2 + E_1 E_2 (p_1 + p_2)} = \int \frac{d^3p_1\, \theta_2 \phi_2 \sin\theta_2}{8(2\pi)^5} \frac{p_2^2}{E_1 E_3 p_2 + E_1 E_2 (p_1 + p_2)} \quad (24)$$

The process is still invariant by rotation around the collision axis, thus we integrate on $d\phi_2$:

$$\int d\Pi_3 = \int \frac{d^3p_1 \, d\cos\theta_2}{8(2\pi)^4} \frac{p_2^2}{E_1 E_3 p_2 + E_1 E_2 (p_1 + p_2)} \tag{25}$$

Finally we find that according to this derivation, the 5 free variables of the phase-space parametrisation are (p_1^x, p_1^y, p_2^z, $\cos\theta_2$, E_{cm}). The only additional equation that we get is the mass invariance. Thus we now have 5 free variables. As told before there are different ways to parametrise the 5 dimensional phase space for this kind of process. For instance *RAMBO on diet* [8] describes a phase space generation algorithm which is flat for massless particles and approximately flat for massive particles.

3.2 Results

For this process, the creation of the training/testing data is a bit different than before. We here use the software called MadGraph [9] that can, for a given process, generate points in phase space with their associated matrix element. Each point of the input data generated by MadGraph has the values (E, p_x, p_y, p_z, m) for each particles, i.e. the 4-momentum plus the mass of the particle. For this very process, the dimensionality of each input point is then 25. MadGraph also generates directly the flat inputs. Each point of the output is still the matrix element as before.

Here, the batches and mini batches have the same size as before. Figure 10 presents the result of the input study.

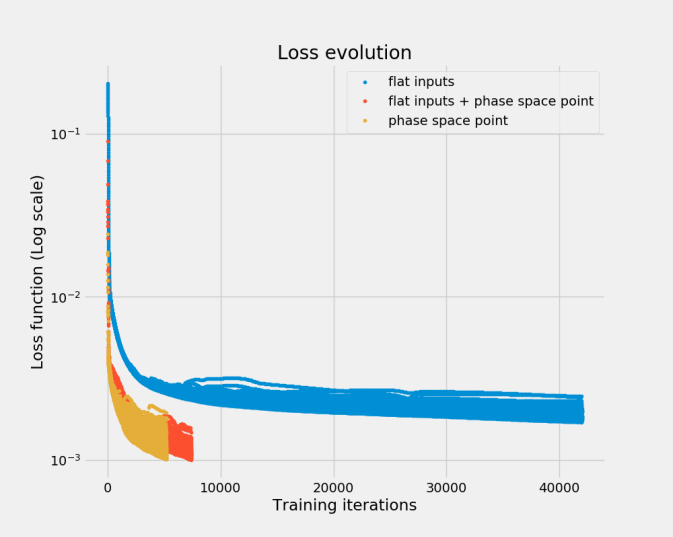

Figure 10: Evolution of the loss function for process 2 by training with 4-momenta, flat inputs and both.

One can see that in this case, the flat inputs are not the most optimal training variables to use because the loss function is stuck at some point whereas by using the phase space point one can have more accurate results. For this process: the recurrent variables (full phase space point) given as input for the learning process are relevant and even necessary in order to get the desired NN performances.

One would like, in a general way, to look at the result of the NN directly without the loss function, to see if the results are relevant. One way to visualize the performance of the neural network compared to the true value of the matrix element, we will visualize a section of the input space and plot on the figure the output of the neural network and the value of the matrix element for this section. For instance, considering the NN that calculate the matrix element of the process 2 taking as input the flat input, fig. 11 shows this result for the section in the flat inputs space, i.e. in the subspace : $\{x, 0.82, 0.680, 0.419, 0.070), x \in]0, 1[\}$.

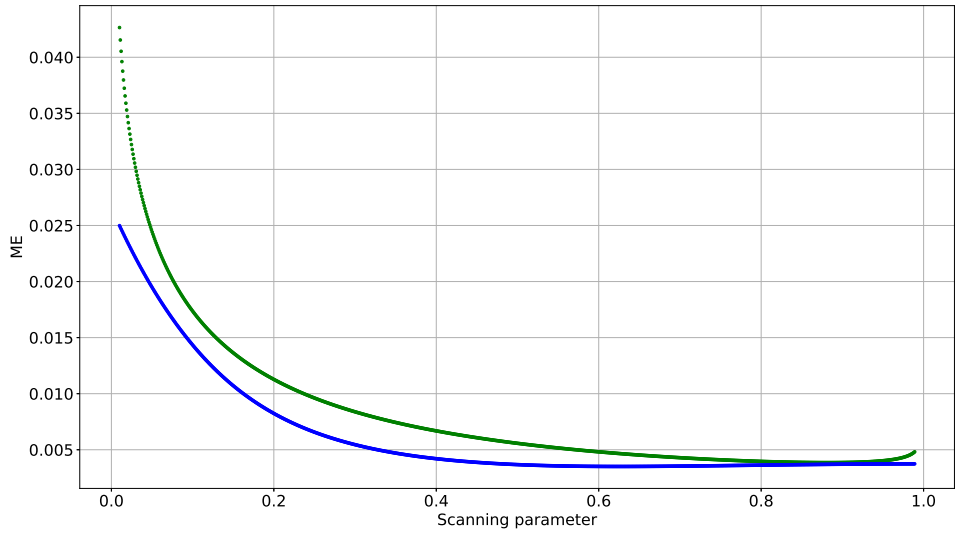

Figure 11: Comparison of the true matrix element and the neural network output for a section of the flat_input space (In green: true matrix element, in blue: NN approximation).

We see that the neural network gives us a good approximation of the matrix element. The approximation is good enough to be used in the presampled Monte Carlo integration in the next section.

4 Process 3 : $[e^+ + e^- \to t + \bar{t} + z + z]$

This process is the previous process with a z particle added. There are 55 Feynmann diagrams associated with this process.

4.1 Phase-space parametrisation

By the same arguments as for the previous processes, we have here $5 + 3$ free variables.

4.2 Results

Here we tried several architectures of NN in order to get a reasonable approximation of the matrix element, but it is obviously a failure. For instance, a NN trained on process 3 with flat inputs, a training batch of 10^5 points, mini batches of 500 points, 200 hidden layers and 16 neurons per hidden layers gives the loss function evolution plotted on fig. 12.

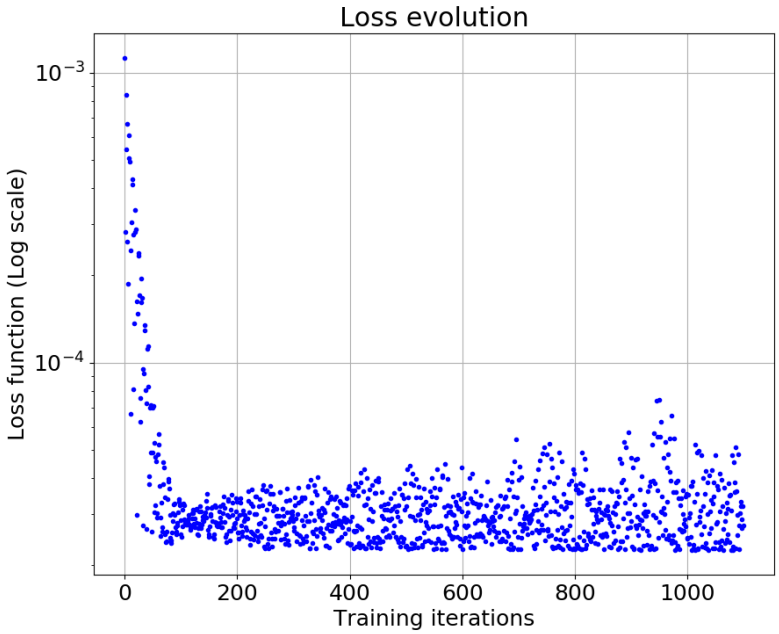

Figure 12: Evolution of the loss function of NN trained on process 3.

One can see that the loss function cannot reach a value below $2 \cdot 10^{-5}$ (under these conditions).

Figure 13 shows the results of the NN at the end of the training in function of the first component of the flat inputs, for all points of the testing batch (green: true value, blue: NN output).

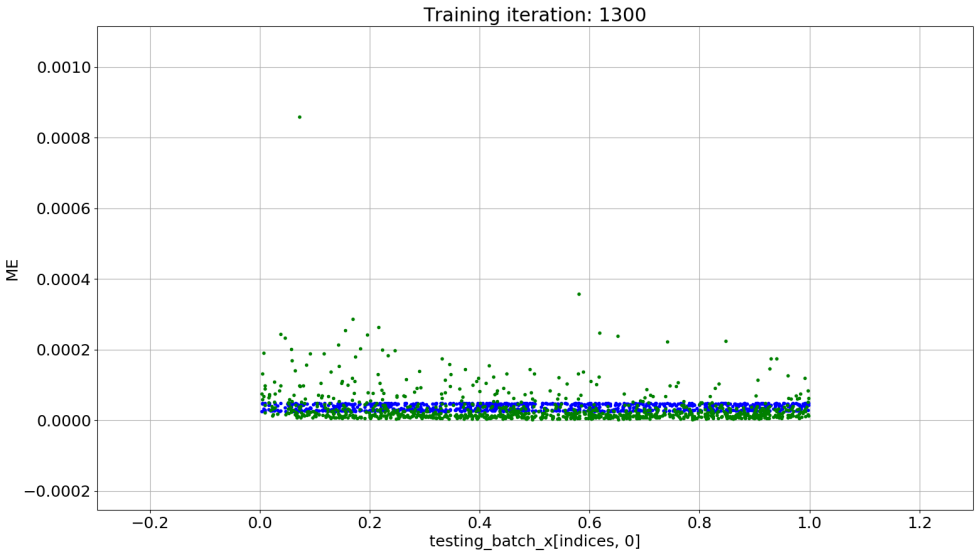

Figure 13: Comparison of the output of the NN with the theoretical matrix element in function of the first component of the flat inputs vector, at the end of the training process.

Like before it is quite convenient to plot the section of the input space. Here we also did a scan on the first component of the flat_inputs. The section is $\{(x, 0.433, 0.696, 0.926, 0.622, 0.203, 0.819, 0.456), x \in]0, 1[\}$.

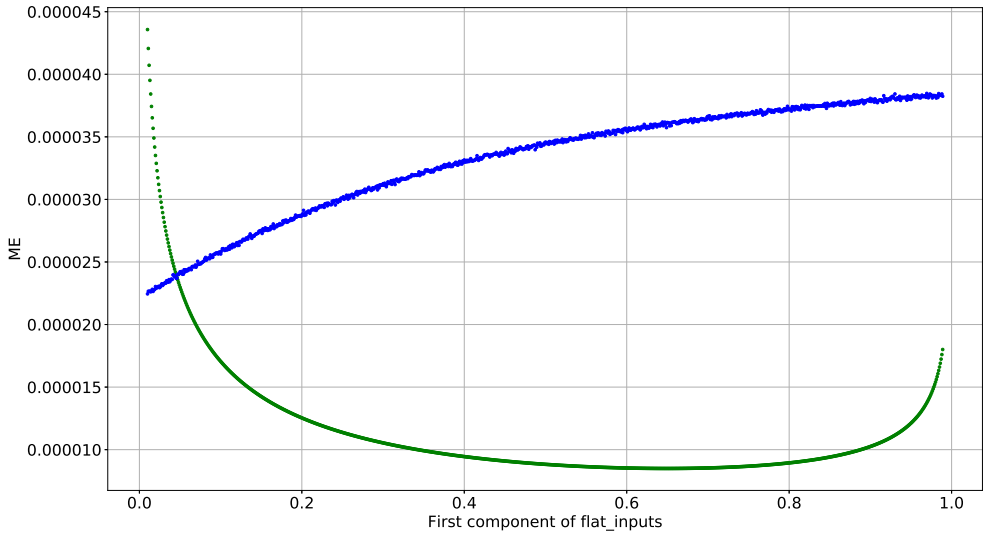

Figure 14: Comparison of the result of the NN and of the true matrix element for a section in the space of the flat inputs.

One can say that the result is pretty bad and one could guess that this is due to the fact that the training batch file is too small for the NN to learn such a complex process. Several tries were done with a training batch file of 10^6 points, but a personal computer like the one used during this study cannot handle such big files. For future studies, one could think about trying to train the NN with different phase space parametrisations at the same time.

5 Monte Carlo integration with Neural Network approximation

Knowing that our implementation of the NN can give use the approximation of the matrix element of a process for a specific point in phase space, one would like to use the NN to integrate on these weights in the phase space in order to calculate the cross section of the process. We will here compare two different integration techniques: the classical Monte Carlo integration and the Monte Carlo integration using presampling with a neural network approximation.

5.1 Integration technique

5.1.1 Monte Carlo

One basic approach in this case would be to use the Monte Carlo (MC) algorithm. This algorithm consists of randomly generating a point in the domain of integration. In our case it would be in the phase space of the 4-momenta of every external particles of the considered process respecting the physical constraints implied by the Quantum Field Theory. Let's call this point x_i. Then the integral estimation using MC is :

$$I_{MC} = \frac{1}{N} \sum_{i=1}^{N} f(\omega_i) \tag{26}$$

Where $\omega_i = f(x_i)$ is the value of the integrand (given by MadGraph) for the specific phase point x_i and N is the number of phase space points generated. The Monte Carlo error is :

$$\Delta I_{MC} = \sqrt{\frac{\left(\frac{1}{N}\sum_{i=1}^{N} \omega_i^2\right) - I_{MC}^2}{N}} \tag{27}$$

The variance is calculated by :

$$\sigma_{MC} = \sqrt{N} \cdot \frac{\Delta I_{MC}}{I_{MC}} = \sqrt{\left(\frac{1}{N I_{MC}^2} \sum_{i=1}^{N} w_i^2\right) - 1} \tag{28}$$

5.1.2 Monte Carlo with presampling using neural network approximation

We will also use another method consisting of doing a change of variable of the integrand so that it is flat using the approximation of the integrand of the NN. Let's call x_i the ith point randomly picked from the integration space, NN(x_i) the approximation of the integrand by the NN for this point, $f(x_i)$ the true value of the integrand evaluated at the x_i point, NN$_{max}$ our estimation of the maximum value of the NN's output and $y_i \in U(0,\text{NN}_{max})$ a uniformly random variable. For every phase space point x_i randomly generated, we do the following: For the ith iteration:

1. Computing NN(x_i) and picking a random $y_i \in U(0,\text{NN}_{max})$
2. If $y_i <$ NN(x_i) the point is *hit* (accepted), otherwise we skip this point and we pick another one.
3. If the point is hit, we compute the weight : $w_i = \frac{f(x_i)}{\min(\text{NN}(x_i),\text{NN}_{max})} \cdot \text{NN}_{max}$

The loop goes on until we have the sufficient amount of hit points. Let's call N_{trials} the number of iteration of the loop we had to do so we have the right number of hit points N_{hit}. This presampling machinery transform a simple MC into a subtle integration by performing a numerical change of variable, as we will see explicitly later.

This idea of presampling is particularly efficient in the situation where the computational time of evaluating the approximation NN(x_i) is much smaller than evaluating the real integrand $f(x_i)$ that we do with MadGraph. Indeed, it is more important for us to compute a weight that is really worth it, i.e. we suppose that the approximation will us give a hint of its importance and then we decide to compute $f(x_i)$ or not (hit or not).

The estimation of the integral is trivially computed like before but with the new weights (PMC stands for Presampled Monte Carlo):

$$I_{PMC} = \frac{1}{N_{trials}} \sum_{i=1}^{N_{trials}} w_i \tag{29}$$

The expression of the Monte Carlo error is also the same as before (but with the new weights):

$$\Delta I_{PMC} = \sqrt{\frac{\left(\frac{1}{N_{trials}} \sum_{i=1}^{N_{trials}} \omega_i^2\right) - I^2}{N_{trials}}} \tag{30}$$

One also need to introduce the numerical efficiency γ:

$$\gamma = \frac{N_{hit}}{N_{trials}} \tag{31}$$

The numerical efficiency quantifies the ratio of hit points for a finite set of points $\{x_i\}_i$. The analytic efficiency is defined by:

$$\Gamma = \frac{1}{\text{NN}_{max}} \int_{D_x} dx\, \text{NN}(x) \tag{32}$$

Where D_x is the general integration domain of $f(x)$. According to our algorithm, we see that for each point x_i, the probability that it is *hit* is $P_{hit}(x_i) = \frac{\text{NN}(x)}{\text{NN}_{max}}$. It means that for an infinite number of trials, $\gamma = \int dx\, P_{hit}(x) = \frac{1}{\text{NN}_{max}} \int dx\, \text{NN}(x)$. Thus,

$$\Gamma = \lim_{N_{trials} \to \infty} \gamma \tag{33}$$

We can also define the intrinsic version (noted with a prime) of the previous quantities by taking into account the number of *hit* points rather than the total number of point, i.e. N_{trials}. These quantities correspond to the previous quantities but where the change of variable has been made analytically. Respectively the intrinsic integral estimate I'_{PMC}, the intrinsic Monte Carlo Error $\Delta I'_{PMC}$ are then defined by:

$$I'_{PMC} = \frac{1}{N_{hit}} \sum_{i=1}^{N_{hit}} w_i \ , \ \Delta I'_{PMC} = \sqrt{\frac{\left(\frac{1}{N_{hits}} \sum_{i=1}^{N_{hits}} \omega_i^2\right) - I^2}{N_{hits}}} \tag{34}$$

A efficient way to estimate the integral I by only using the intrinsic integral estimation I'_{PMC} and the analytic efficient Γ is:

$$I = I'_{PMC} \cdot \Gamma \tag{35}$$

This result is only useful only if it is possible for us to calculate the exact integral of the approximation. Let's note $T(x)$ an approximation of $f(x)$ in the general case. In order to prove the previous result, we have to consider the continuous version of the intrinsic integral estimation (eq. 34), which is relevant in a situation in which we perform an infinite Monte Carlo algorithm. We first express the analytic number of hit points with the Heaviside step function[2] which is embedding sampling process:

$$N_{hit} = \int_{D_x} dx \underbrace{\int_0^{T_{max}} dy\, \theta(T(x) - y)}_{\min(T(x);T_{max})} = \int_{D_x} dx\, \min(T(x); T_{max}) = \int_{D_x} dx\, T(x) \tag{36}$$

Where we used that the Heaviside step function is equal to 1 only if $y \leq T(x)$, 0 otherwise and the fact that $T(x) \leq T_{max}$ all over the integration domain D_x by definition of T_{max}. Using the same formalism, we can also express the discrete sum of weights of the intrinsic interval estimate (eq. 34) in a continuous way:

$$\sum_{i=1}^{N_{hit}} w_i \to \int_{D_x} dx \int_0^{T_{max}} dy\, \theta(T(x) - y) \frac{f(x)}{\min(T(x); T_{max})} T_{max} \tag{37}$$

$$= T_{max} \int_{D_x} dx\, \frac{f(x)}{\min(T(x); T_{max})} \int_0^{T_{max}} dy\, \theta(T(x) - y) = T_{max} \int_{D_x} dx\, f(x) \tag{38}$$

Where, like previously, we turned the integral on y into the minimum of $T(x)$ and T_{max}. Eq. 38 also shows that the selection of points (hit or missed) coupled with these weights make that it performs a change of variable. Then we have:

$$I'_{PMC} \cdot \Gamma = \underbrace{\left(\frac{1}{\int_{D_x} dx'\, T(x')}\right) \cdot \left(T_{max} \int_{D_x} dx\, f(x)\right)}_{I'_{PMC}} \cdot \underbrace{\left(\frac{1}{T_{max}} \int_{D_x} dx'' T(x'')\right)}_{\Gamma} = \int_{D_x} dx\, f(x) \quad \square \tag{39}$$

The error associated to this integral estimation is $\Delta I'_{PMC} \cdot \Gamma$ where $\Delta I'_{PMC}$ has been defined in 34.

This method of using the intrinsic integral value estimation and the analytic efficient is way more accurate, as we will see in the example in the next subsection. Nevertheless it is necessary that we have access to the exact value of the approximation's integral (Γ).

Because the second method of estimation is equivalent to integrate after doing a change of variable that make the integrand flat, we suppose that the variance of the final integral estimation with be less than with the Monte Carlo integration algorithm. We will estimate these integrals for process 1 and 2 because the NN trained before give us a good approximation of the integrand.

5.1.3 Illustrative example

MC As an example to illustrate these two methods, we will consider the integration of the function $f(x) = 2\cos\frac{\pi}{2}x$ integrated over $[0, 1]$. We trivially find that the analytic result the integral of $f(x)$ is $\int_0^1 dx\, f(x) = \frac{4}{\pi}$. Moreover knowing that the integration domain is $[0, 1]$, the mean value of the function ($\bar{f}$) is also equal to $\frac{4}{\pi}$. The variance (σ) of the function over its integration domain without presampling is:

$$\sigma_{[0,1]}(f) = \int_0^1 dx \left[f(x) - \bar{f}\right]^2 = 2 - \frac{16}{\pi^2} \approx 0.378 \tag{40}$$

MC with presampling The Taylor development of $f(x)$ of order 1 is $T_1(x) = 2 - \frac{\pi^2}{16}x^2$. The integral of the approximation is: $\int_0^1 dx\, T_1(x) = 2 - \frac{\pi^2}{48}$.

[2]Heaviside step function is defined by $\theta(x) = 1$ if $x > 0$ and $\theta(x) = 0$ if $x < 0$

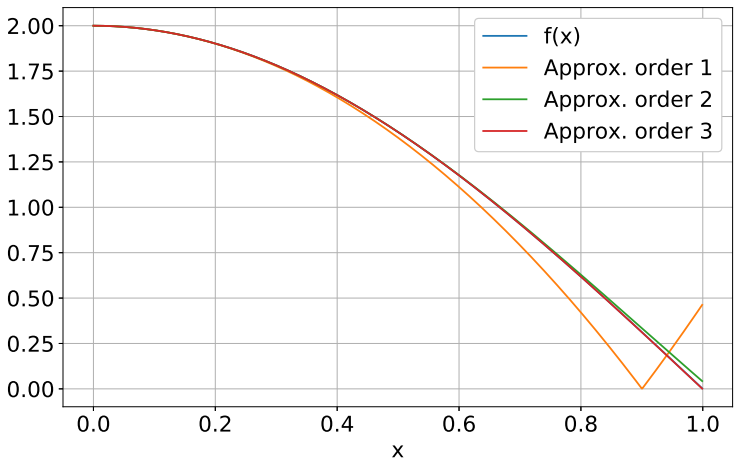

Figure 15: Plot of the function $f(x)$ and of its Taylor expansions of order 1, 2 3.

Its intrinsic variance is then:

$$\sigma(f') = \int_0^1 dx \left[\frac{2\cos\frac{\pi}{2}x}{2-\frac{\pi^2}{16}x^2} - \bar{f}'(x)\right]^2 \approx 0.085488 \tag{41}$$

The analytic change of variable for this integral would be:

$$I = \int_0^1 dx\, 2\cos\frac{\pi}{2}x = \int_0^1 dx T_1(x)\frac{2\cos\frac{\pi}{2}x}{T_1(x)} = \int_{\xi[0]}^{\xi[1]} d\xi \frac{dx}{d\xi} T_1(x[\xi])\frac{2\cos\frac{\pi}{2}x[\xi]}{T_1(x[\xi])} \tag{42}$$

In order for the change of variable to be correct, we should have:

$$\frac{dx}{d\xi}T_1(x) = 1 \Rightarrow \frac{dx(\xi)}{d\xi} = \frac{1}{2-\frac{\pi^2}{16}x[\xi]^2} \Rightarrow \int dx(\xi) = \int \frac{d\xi}{2-\frac{\pi^2}{16}x[\xi]^2} \tag{43}$$

Therefore (by using that $\frac{d\,\mathrm{arctanh}\,x}{dx} = \frac{1}{1-x^2}$),

$$x[\xi] = \frac{2\sqrt{2}}{\pi}\,\mathrm{arctanh}\left(\frac{\pi}{4\sqrt{2}}\xi\right) \quad \text{and} \quad \xi[x] = \frac{4\sqrt{2}}{\pi}\tanh\left(\frac{\pi}{2\sqrt{2}x}\right) \tag{44}$$

The now boundaries are then $\xi_1 = 0$ and $\xi_2 = 2\frac{\pi^2}{48}$. The whole integral become:

$$I = \int_0^{2-\frac{\pi^2}{48}} d\xi\, \frac{2\cos\left(\sqrt{2}\,\mathrm{arctanh}\left(\frac{\pi}{4\sqrt{2}}\xi\right)\right)}{2-\frac{1}{2}\,\mathrm{arctanh}\left(\frac{\pi}{4\sqrt{2}}\xi\right)^2} \tag{45}$$

We will now prove the equation 35 for the function $f(x) = 2\cos\frac{\pi}{2}x$. The analytic efficiency for this case is:

$$\Gamma = \frac{1}{\max T_1(x)}\int_0^1 dx\, T_1(x) = \frac{1}{2}\int_0^1 dx\, \left(2-\frac{\pi^2}{16}x^2\right) = 1-\frac{\pi^2}{96} \tag{46}$$

Let's calculate the intrinsic intregral value by using eq. 36 and 38:

$$I'_{PMC} = \frac{1}{\int_{D_x} dx'\, T(x')} T_{max}\int_{D_x} dx\, f(x) = \frac{2}{2-\frac{\pi^2}{48}}\frac{4}{\pi} \tag{47}$$

Therefore:

$$I'_{PMC}\cdot\Gamma = \frac{2}{2-\frac{\pi^2}{48}}\frac{4}{\pi}\cdot\left(1-\frac{\pi^2}{96}\right) = \frac{4}{\pi} = \int_0^1 dx\, f(x) \tag{48}$$

We performed these two algorithms with the constraint that it compute 10^5 integrands (it means that $N_{hits} = 10^5$ for the presampled Monte Carlo algorithm). Fig. 16 is a plot of the error between the true

integral value ($\frac{4}{\pi}$) and the estimations provided by the Monte Carlo with and without presampling. The point plotted on the 0 order means no use of any approximation (classic Monte Carlo algorithm). In red we have the error regarding to the integral value calculated with eq. 26 for the order 0 and eq. 29 for the presampled Monte Carlo for order 1, 2 and 3. The points in green are only possible to calculate with the presampled Monte Carlo algorithm and thus are plotted only for order 1, 2 and 3. These points are the errors between the true value of the integral and the formula that uses the intrinsic integral estimate and the full integral of the approximation, i.e. eq. 35.

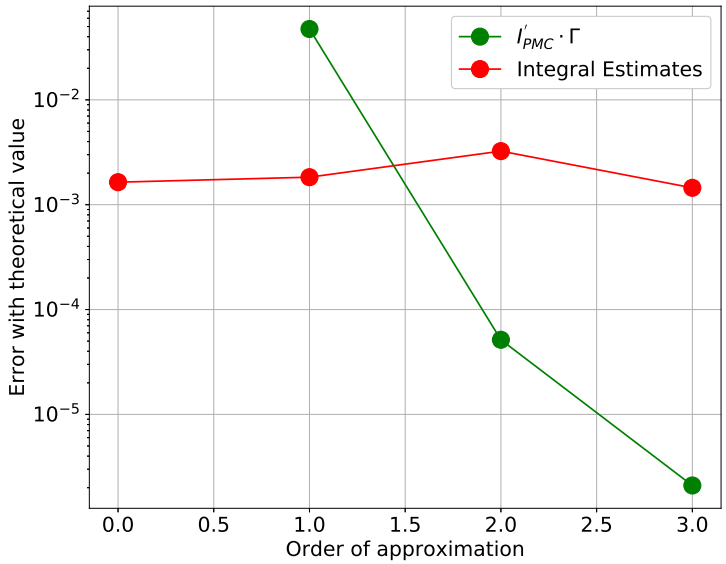

Figure 16: Comparison of the error between the true integral value and two integral estimations using the Monte Carlo algorithm with and without presampling.

The important thing to get from this result is that regardless which Monte Carlo algorithm we used (classical or presampled), the direct evaluation of the integral by summing the weights (in red, eq. 26) doesn't get better, even if the approximation gets closer to the true function. Nevertheless, the error with estimation using eq. 35 (in green on the plot) decreases exponentially in terms of the order of the Taylor expansion used. Note these results were produced with the efficiency Γ being the analytic efficient i.e. the exact theoretical efficiency. We can get this improvement of the integral estimation **only and only if** we have access to the exact value of the integral of the approximation (which is the case in fig. 16).

We now continue the comparison of these two integral estimations by looking to the numerical errors associated. The formulas used are 30 (in red) and $\Delta I'_{PMC} \cdot \Gamma$ (in green) associated to the estimation using the simple sum of weights and the one using the efficiency. Like before we look at the different values of these quantities for different order or the Taylor approximations. The result is plotted on fig. 17.

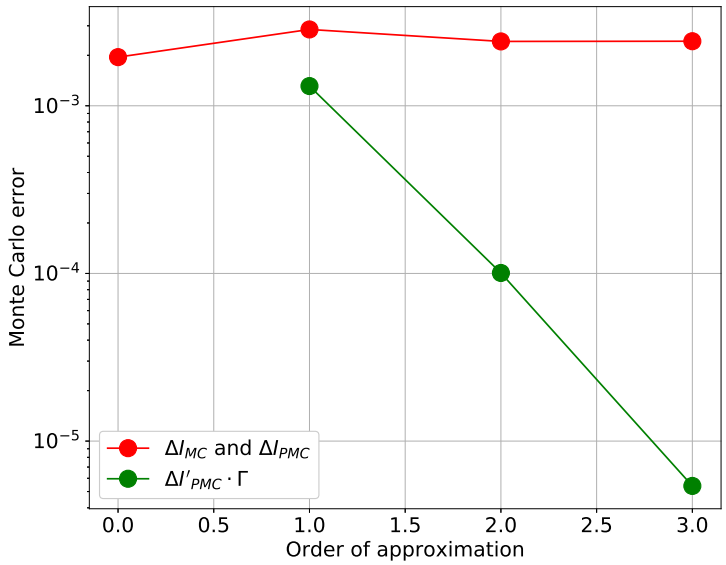

Figure 17: Comparison of the Monte Carlo error associated to the two integral estimations.

We can see that the MC error in red corresponding to the estimation without using the analytic efficiency doesn't decrease when the approximation gets closer to the true function. Nevertheless the MC error of the integral estimation calculated with the efficiency decreases exponentially as we increase the order of the Taylor expansion used as approximation.

In conclusion, if we have access to the analytic efficiency of an approximation $\Gamma = \frac{1}{T_{max}} \int_0^1 dx T(x)$, then using the estimate of the integral using eq. 35 is way more accurate by making use of the approximation. Unfortunately we won't have access to the analytic efficiency if the approximation we use is the NN. Nevertheless using the presampling algorithm will help us reducing the computational time of the integration by selecting the relevant integrand to calculate. Moreover, knowing that the use of the presampling method flattens the integrated function, we will see that the variance decreases compared to a classic MC integration.

5.2 Process I : $[e^+ + e^- \to t + \bar{t}]$

The neural network used for the approximation during the presampling is the following: It has 18 inputs neurons (it takes the flat inputs and all external 4-momenta), 30 hidden layers, 18 neurons per hidden layers. It was trained with a total batch file composed of 10^5 points and with mini-batch files composed of 10^3 points. The learning rated is constant during the training and is $\beta = 10^{-4}$. The loss function used is the mean-squared error function.

We look at the precision of the final result with both algorithms by looking at the evolution of the error estimation and of the intrinsic variance for different number of integrands calculated (see fig. 21). A relevant way to see the distribution of the weights regarding to the two algorithms is to do histograms (see fig 24) and to compare directly the two integration technique regarding to the variance of their weights (see fig. 25). These result are from an integration with 10^5 integrands calculated.

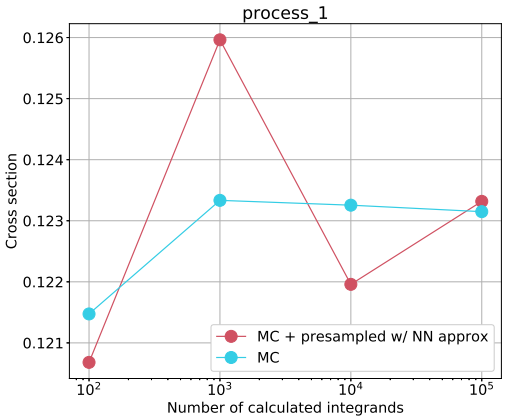

Figure 18: Comparison of the integral estimate for the Monte Carlo algorithm and the Monte Carlo + presampling algorithm for different amounts of calculated integrands.

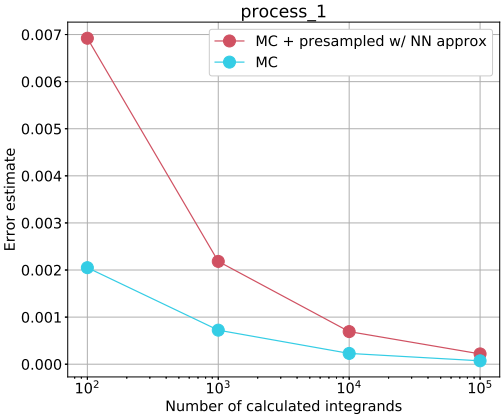

Figure 19: Error estimate

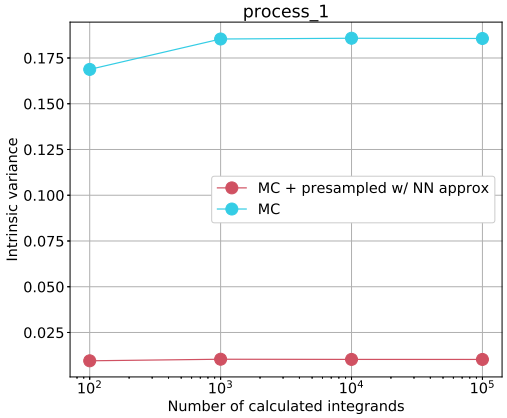

Figure 20: Intrinsic Variance

Figure 21: Evolution of the result precision in function of the number of integrands calculated

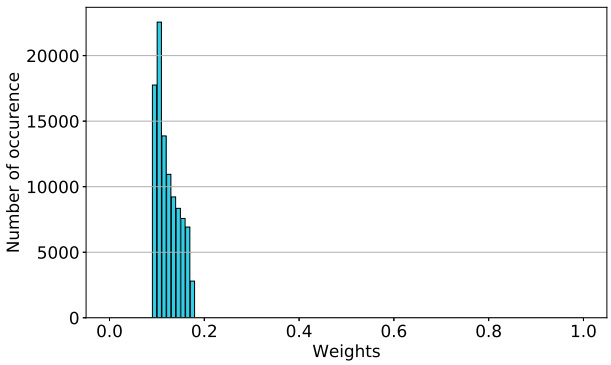

Figure 22: Monte Carlo

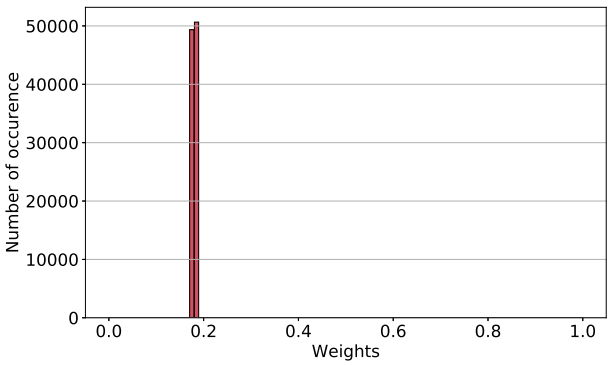

Figure 23: Monte Carlo with presampling technique

Figure 24: Weight distribution for the two integration algorithms described above (10^5 integrands calculated).

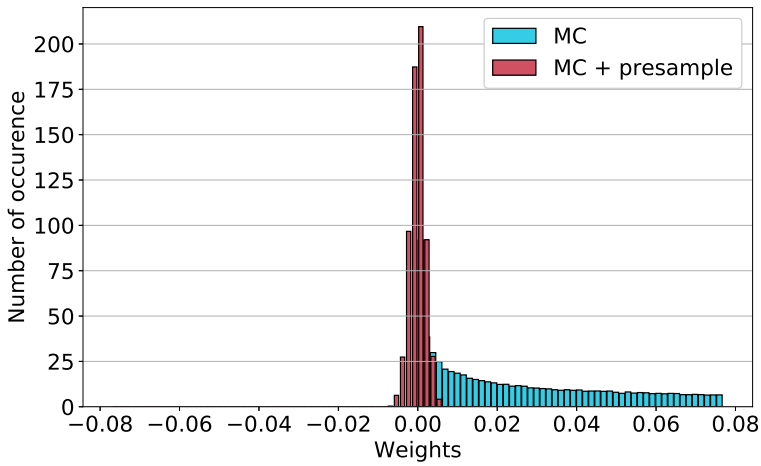

Figure 25: Comparison of the two integration techniques by comparing their centered weight distribution (10^5 integrands calculated).

One can clearly see that the variance of the weights from the second algorithm is way smaller than the variance form the weights calculated with the Monte Carlo algorithm.

One can then have a look at the time it takes for both algorithm to compute one integrand. The result is shown on fig. 26. As we could have expected, using the NN (with the second algorithm) is a little bit slower than directly computing the integrand. If we take respectively the average value of the time duration for both algorithm, we see that on average, the algorithm using the NN approximation is 1.588 slower than the other one.

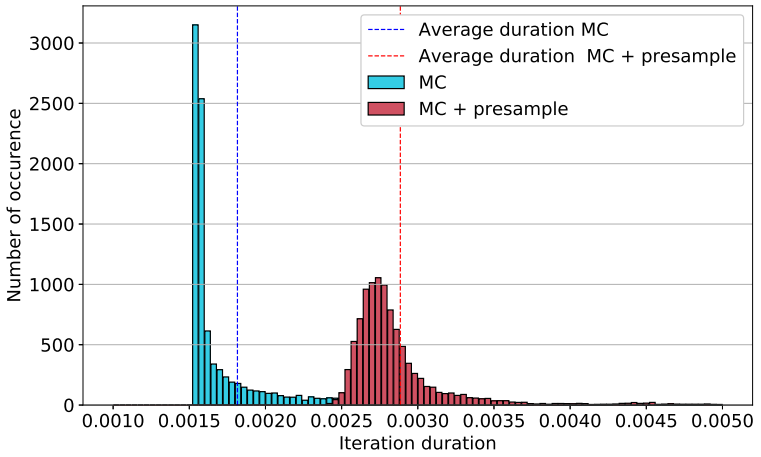

Figure 26: Computational time study for the 2 algorithms (in seconds).

5.3 Process 2 : $[e^+ + e^- \to t + \bar{t} + z]$

The neural network used for the approximation during the presample is the following: It has 25 inputs neurons (it takes the flat inputs and all external 4-momenta), 40 hidden layers, 25 neurons per hidden layers. It was trained with a total batch file composed of 10^5 points and the mini-batch files have 10^3 points. The learning rated is constant during the training and is $\beta = 10^{-4}$. The loss function used is the mean-squared error function.

We do the same thing for the process 2. This time since the process is more complicated, the NN takes much more time to evaluate a point, and make the integration with 10^5 is too long. The following results (fig 30, 33 and 34) are then the same as before but for process 2.

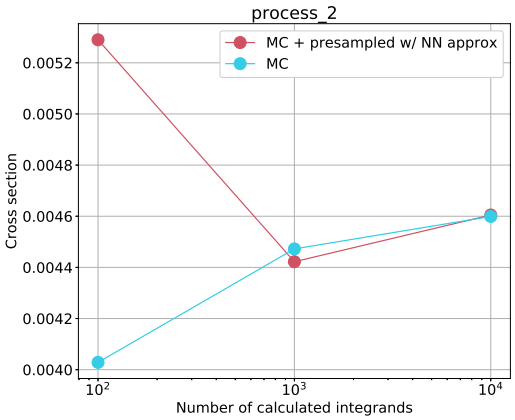

Figure 27: Comparison of the integral estimation for the Monte Carlo algorithm and the Monte Carlo + presampling algorithm for different amounts of calculated integrands.

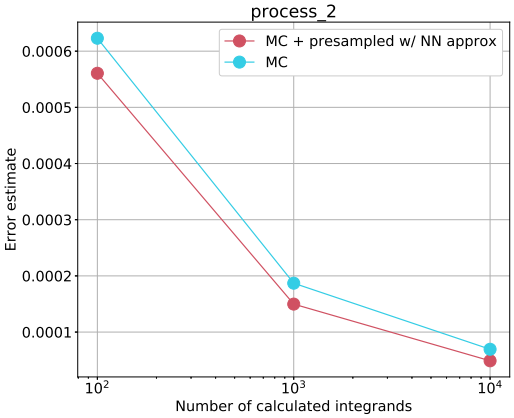

Figure 28: Error estimate

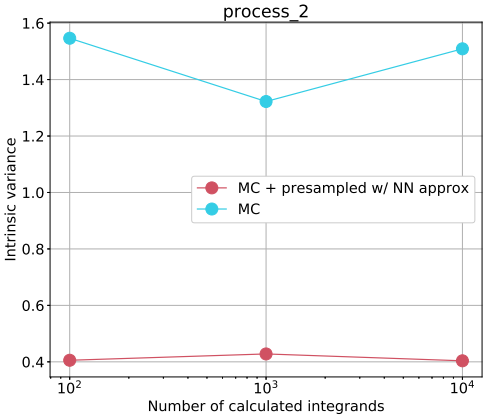

Figure 29: Intrinsic Variance

Figure 30: Evolution of the result precision in function of the number of integrands calculated

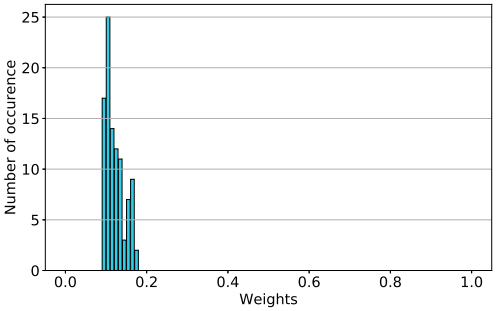

Figure 31: Monte Carlo

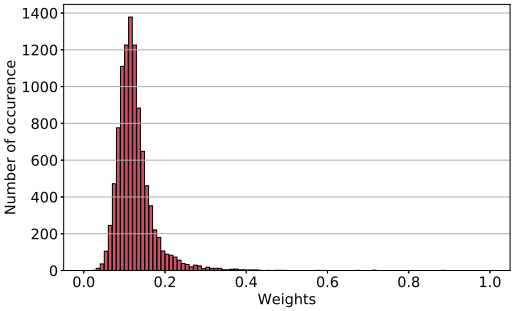

Figure 32: Monte Carlo with presampling technique

Figure 33: Weight distribution for the two integration algorithms described above.

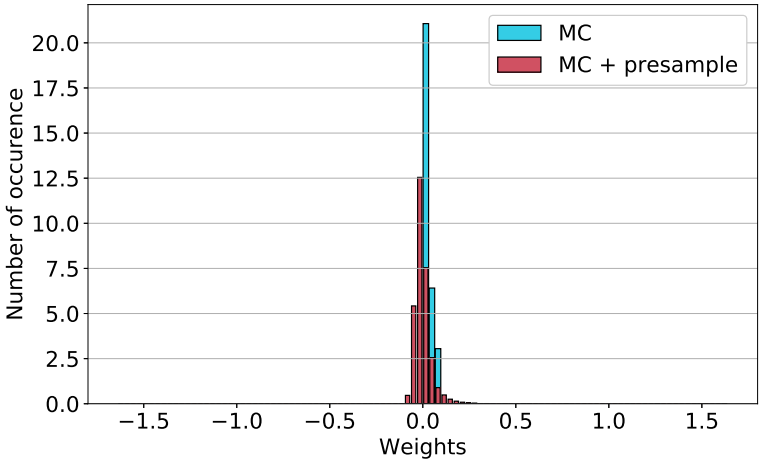

Figure 34: Comparison of the two integration techniques by comparing their centered weight distribution.

Unfortunately we see that for this process, the variance has not been reduced by using the NN with the presampling method. This is due to the fact that the NN approximation is too bad. Theoretically if a more powerful computational system was used instead of a personal laptop, the NN could be better trained and hence the variance of the weights decreased.

We look at the computational time for both algorithms, like before. The result can be seen on fig. 35. This time the multiplication factor between the average duration between the algorithm 1 and 2 is approximately 1.47. One can have the same conclusion as before about the use of the NN in the integrand computational process.

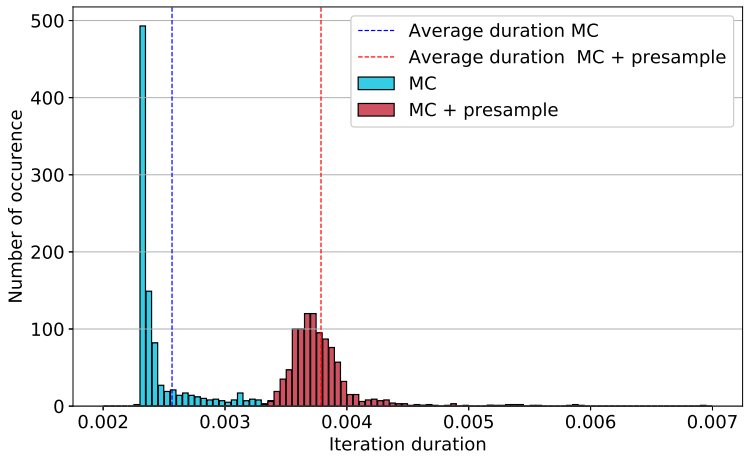

Figure 35: Computational time study for the 2 algorithms.

6 Results summary

The tabular 1 gives us a good summary of the main results produced during this study for the three different processes.

Process		1	2	3
Best loss value achieved	Value	$1 \cdot 10^{-3}$	$1 \cdot 10^{-3}$	$2.9 \cdot 10^{-5}$
	Quality of approximation	Good	Good	Bad
Monte Carlo	Integral value	$1.23149 \cdot 10^{-1}$	$4.5991 \cdot 10^{-3}$	$3.8315 \cdot 10^{-5}$
	Error estimation	$0.000723 \cdot 10^{-1}$	$0.0694 \cdot 10^{-3}$	$0.226 \cdot 10^{-5}$
	Intrinsic Variance	$1.856 \cdot 10^{-1}$	1.508	1.63
Monte Carlo + presample	Integral value	$1.23317 \cdot 10^{-1}$	$4.6048 \cdot 10^{-3}$	None
	Error estimation	$0.02188 \cdot 10^{-1}$	$0.0488 \cdot 10^{-3}$	None
	Intrinsic Variance	$1.027 \cdot 10^{-2}$	$4.036 \cdot 10^{-1}$	None

Table 1: Results summary

7 Conclusion

We have seen that NNs can approximate integrand elements in a good way as long as the computational power of the machine running the training process is powerful and as long as we have a sufficient amount of data. In general we see that the more we have hidden layer the fastest the training process is. Nevertheless adding a lot of hidden layers in a NN slows down the evaluation of one point; this has to be taken into account in the context of integration where the NN has to be the approximation useful because of its quick evaluation capacity (compared to MadGraph). In a future study it would interesting to explore how much slower the integration of a process would be for a specific amount of added hidden layers. We also only used dense neural networks whereas it could be interesting to explore the performances of others type of neural network, for instance convolutional neural networks.

We found that during the training of process 2, feeding the NN with recurrent variables is necessary to have a good approximation of the matrix element. Moreover during this study, for each process we used only one parametrisation of phase-space. One interesting development of the NN approximations would be to train the NNs with different flat inputs parametrisations on one hand, and, with several parametrisations at the same time.

Moreover these NN approximations can be used to reduce the variance of the final result of the Monte Carlo approximation when a presampling strategy is performed. This method can give better results in terms of MC error and of time performance.

Another relevant exploring way would be to try to teach the same NN different processes. For instant learn the NN how to approximate process 1, then process 2, then process 3. The main difficulty in this case would be to choose the relevant variable to use as input so that one training process doesn't bring down the previous one.

References

[1] Machine Learning Wikipedia Page

[2] Neural Network-Based Approach to Phase Space Integration by Matthew D. Klimek and Maxim Perelsteina (arXiv:1810.11509v1 [hep-ph] 26 Oct 2018)

[3] Efficient Monte Carlo Integration Using Boosted Decision Trees and Generative Deep Neural Networks by Joshua Bendavid (arXiv:1707.00028v1 [hep-ph] 30 Jun 2017)

[4] Oreilly website: https://www.oreilly.com/library/view/practical-convolutional-neural/9781788392303/5201f645-3f1f-4a24-8f9e-65023396e340.xhtml

[5] Alykhan Tejani's GitHub repository: https://alykhantejani.github.io/images/gradient_descent_line_graph.gif

[6] Tensorflow : https://www.tensorflow.org

[7] *An Introduction to Quantum Field Theory* by Daniel V. Schroeder and Michael Peskin

[8] textitRAMBO on diet by Simon Plätzer (arXiv:1308.2922v1 [hep-ph] 13 Aug 2013)

[9] *MadGraph* software for Standard Model phenomenology http://inspirehep.net/record/1293923