

TECHNISCHE
UNIVERSITÄT
DRESDEN

Efficiency Improvements Using Machine Learning in Event Generators for the LHC

Master-Arbeit
zur Erlangung des Hochschulgrades
Master of Science
im Master-Studiengang Physik

vorgelegt von

Johannes Krause
geboren am 17.10.1990 in Frankfurt (Oder)

Institut für Kern- und Teilchenphysik
Fachrichtung Physik
Fakultät Mathematik und Naturwissenschaften
Technische Universität Dresden
2015



Eingereicht am 13. Oktober 2015

1. Gutachter: Dr. Frank Siegert
2. Gutachter: Prof. Dr. Michael Kobel

Summary

Abstract

In particle physics, event generators play an important role in connecting theoretical predictions with experimental results. Especially for experimental issues one often wants to generate events which follow a probability distribution based on the underlying physics. The generation of such events is currently done by a ‘hit-or-miss’ method, which is particularly for processes with many final-state particles very expensive in terms of CPU time.

In this thesis a modified variant of this algorithm is introduced, which combines the ‘hit-or-miss’-method with machine-learning techniques in order to improve the performance. Several possible applications of this new method are presented and compared with currently available methods of event generation in SHERPA for different processes.

Kurzdarstellung

Ereignis-Generatoren sind in der Teilchenphysik unerlässlich, um theoretische Vorhersagen mit experimentellen Befunden zu verknüpfen. Insbesondere für experimentelle Belange werden häufig simulierte Ereignisse benötigt, welche ihrer zugrundeliegenden physikalischen Wahrscheinlichkeitsverteilung folgen. Zur Erzeugung solcher Ereignisse wird momentan die ‘Hit-or-Miss’-Methode genutzt, diese ist jedoch insbesondere für Prozesse mit vielen Teilchen im Endzustand sehr ineffizient.

In dieser Arbeit wird eine neue Methode vorgestellt, welche die klassische ‘Hit-or-miss’-Methode mit Algorithmen aus dem Maschinellen Lernen kombiniert. Mehrere mögliche Anwendungen werden vorgestellt und für verschiedene Prozesse mit den aktuell verfügbaren Methoden der Erzeugung von Ereignissen in SHERPA verglichen.

Contents

1	Introduction	1
2	Fundamentals of Particle Physics	3
2.1	From Electrons to Higgs Bosons – a Historical Introduction	3
2.2	The Standard Model	5
2.2.1	The Electroweak Interaction	5
2.2.2	The Strong Interaction	6
2.2.3	Fundamental Particles	7
2.3	Experimental and Theoretical Methods in Particle Physics	9
2.3.1	The Large Hadron Collider	9
2.3.2	Feynman Rules, Cross Sections and Perturbation Theory	10
2.3.3	Monte Carlo Event Generators	12
3	Algorithms and Methods	15
3.1	Monte Carlo Methods	15
3.1.1	Monte Carlo Integration	16
3.1.2	Monte Carlo Sampling	17
3.1.3	Unweighting in SHERPA	18
3.2	Introduction to Machine Learning	19
3.2.1	Concepts of typical Regression Algorithms	20
3.2.2	Bagging and Boosting – Meta Techniques	25
4	Machine Learning based Unweighting	27
4.1	Basic Idea	27
4.2	Choice of Algorithms and Hyper-Parameter Optimization	29
4.2.1	Software	29
4.2.2	Generation of Training Data	30
4.2.3	Error Definition	33
4.2.4	First Attempts with Scikit-Learn	34
4.3	Optimization in example of Random Forest	36
4.3.1	Choice of Methods	36
4.3.2	Hyper-Parameter Optimization	39

4.4	Optimization of Neural Networks	45
4.4.1	Configuration by AutoWeka	45
4.4.2	Choice of Algorithms	46
4.4.3	Activation Function and Target Normalization	46
4.4.4	Feature Normalization	47
4.4.5	Learning Rate and Momentum	47
4.4.6	Optimal Number of Training Points	49
4.4.7	The Network's Structure	49
5	Evaluation	59
5.1	Direct Use of Machine-Learning based Unweighting	59
5.1.1	Error Estimation	60
5.1.2	Comparisons of Timings	61
5.2	Second Unweighting	63
5.2.1	Determination of the Second Unweighting Efficiency	64
5.3	Evaluation of the Training Process	65
5.3.1	Direct Methods	65
5.3.2	Second Unweighting	67
5.4	Further Processes	69
5.4.1	$Z + \text{jets}$	69
5.4.2	$W + \text{jets}$	70
6	Conclusion and Outlook	73
A	Run Cards	75
A.1	$Z + \text{jets}$	75
A.2	$W + \text{jets}$	76
B	Plots	77
B.1	$gg \rightarrow e^+e^- \bar{d}dgg$	77
	List of Figures	81
	List of Tables	83

1 Introduction

The question of the origin and composition of visible matter has been challenging people for a very long time. One of the first approaches was developed by the Greek philosopher Democritus around 400 BC. He proposed matter consisting of indivisible 'atoms' in empty space. This theory of course was not based on experiments as is common today but purely philosophical.

Today, answering the question above is still a big challenge. Many experiments and large colliders were constructed to achieve higher and higher energies, allowing to zoom in on smaller and smaller distances. Several theories were developed in order to describe the observations made, leading to our modern view of elementary particles as described by the 'Standard Model of Particle Physics'.

Evaluating experiments and developing new theories always involves comparing the two, in particle physics this is usually done through event generators. Event generators use Monte Carlo methods to generate simulated events and estimate the cross sections of given processes. These simulated events are very useful, as they can be used to obtain predictions for physical observables. These predictions can then be compared to the measurement.

The probability distribution of the desired events is defined by the underlying physical process. There are many applications that require events which indeed follow this distribution. In the event generator SHERPA[1] such events are generated by a 'hit-or-miss' approach, called 'unweighting'. The probability that a given event occurs is mainly determined by the processes' matrix elements. In order to generate one weightless event by using the 'hit-or-miss' method, many matrix elements have to be calculated. This procedure can be very inefficient for processes with many participating particles. The number of necessary calculations increases as well as the complexity of the matrix elements, calculating them becomes therefore more expensive, as well. Especially for processes with e.g. many jets, the current unweighting procedure is stretched to its limit. In order to be prepared for the analyses of current and upcoming experiments, new unweighting methods are needed which perform significantly better.

Generally speaking, a main problem of the approach currently used is the frequent evaluation of a complex function. This general problem exists in many other applications, too, and several methods were developed to deal with it. A very popular field of computer science that can address this issue is 'machine learning'. A subcategory deals hereby with the prediction of target values by learning from known function values, this method is often called 'supervised

regression'. The accuracy of these algorithms was very sophisticated in the last decades and, furthermore, the algorithms are very fast.

It is tempting to employ these powerful algorithms to accelerate the current unweighting procedure. The aim of this work is to investigate whether this is possible and to obtain an estimate for the potential performance gain.

This thesis contains three main parts, divided into four chapters. The first chapter covers the physical basics which are relevant for the thesis. After a brief historical overview, the Standard Model of Particle Physics is discussed. Furthermore, this section covers some experimental and theoretical aspects. Finally event generators are motivated in more detail and their principles are explained.

The second chapter still belongs to the foundations, but by contrast it focuses on algorithms and more technical aspects. Here Monte Carlo methods are introduced and the unweighting procedure is described as it is currently used. Furthermore, this chapter also gives a small overview of the wide field of machine learning and presents some popular algorithms which are used in this thesis.

The second part begins with the third chapter. After presenting a new unweighting procedure based on machine learning, several methods are tested to improve the prediction. An example process for the optimization is chosen as well as a suitable error definition, further different algorithms are compared and their hyper parameters optimized.

The last part, and fourth chapter, deals with the evaluation. The aim of this thesis is to study whether machine learning can improve the unweighting performance. For this reason, new methods are developed to compare the new approach with the current techniques of event generation without the necessity to implement them already. Finally, these methods are used to estimate the possible performance gain for different processes.

2 Fundamentals of Particle Physics

2.1 From Electrons to Higgs Bosons – a Historical Introduction

It took a long time and much effort for generations of physicists to receive our current knowledge of the fundamental particles and their interactions. During this evolution there were several situations, where a current model was established and only a few signs indicated new physics. A similar situation is given today after the discovery of the Higgs boson, the Standard Model is complete but open questions remain. To motivate the study of particle physics, this thesis begins with a small introduction of its historical evolution.

There are several sources which give an overview of this evolution which has led to today's picture of particle physics, this section follows the structure of [2].

The discovery of the electron by J.J. Thomson can be considered as the origin of what is nowadays known as 'particle physics'. He measured the deflection of cathode beams by electric and magnetic fields and was thus able to determine the charge-to-mass ratio of the electron for the first time. His findings led him to propose his famous "plum pudding" model of the atom. This model was disproved by Rutherford's scattering experiments some years later. Rutherford studied the scattering of helium ions off a thin gold foil and showed that nearly the whole mass, despite the positive charge, is located in a point-like center of the atom. Rutherford managed to identify the hydrogen nucleus which he dubbed the 'proton'. In the following decades there were many unsuccessful attempts to explain the known elements by positively charged nuclei particles and negatively charged electrons only. The dilemma why multiply charged nuclei have a much higher mass than expected could only be solved with the discovery of the neutron in 1932.

The classic trio of proton, neutron and electron gave a simple answer to the question of what constituents matter is made of. Nevertheless, there were already clear signs that there had to be more. Planck had postulated that light can only appear in quantized amounts. He assumed this behavior to be due to the emission process, but Einstein already recognized it as feature of the radiation itself. This dispute was solved through an experiment by Compton in 1923: He measured the shift in wavelength of light scattered off electrons at rest and his results were compatible with Einstein's theory of light quanta with zero rest mass. These quanta were later called photons. Moreover, it was the first sign that the classic picture of 'particles' cannot hold

in an exact way – light has characteristics of waves as well as of particles.

A further open question of the classic picture of atoms was the conundrum why multiple positively charged protons in an atomic nucleus do not repulse each other. In order to solve this Yukawa proposed a strong interaction between protons and neutrons. He predicted a massive exchange particle and called it meson. By studying cosmic radiation indeed two new particles were found: the meson expected by Yukawa (called ‘pion’ and discovered in 1947) and another particle similar to the electron (but much heavier) called ‘muon’ (discovered 1937).

Independent from these studies two other particles were found at the same time. By developing a relativistic quantum theory Dirac predicted a particle similar to the electron – but with opposite charge. This antiparticle to the electron was found 1931 by Anderson and called it ‘positron’. It was later discovered that all fermions have corresponding antiparticles. The second discovery at this time were neutrinos. Neutrinos are nearly massless and weakly interacting particles and were first only discovered indirectly: a continuous energy spectrum of emitted electrons was observed in the nuclear beta decay and missing momenta were observed on bubble chamber images.

Around 1947 many physicists thought the particle zoo to be complete, but it did not take long and many new particles similar to the pion and the proton were discovered, for example the kaon which decays into two pions and the lambda particle which decays into a proton and a pion. In 1952 the first modern particle accelerator was built and even more particles were discovered. Furthermore, it was found that these so-called hadrons can be arranged in two groups: mesons (like the pion and the kaon) and baryons (such as the proton and lambda). In an attempt to simplify and give structure to the complex world of hadrons, the physicist Gell-Mann introduced ‘The Eightfold Way’ in 1961. This theory allows for all mesons and baryons to be arranged into multiplets ordered by their electric charge and their ‘strangeness’, a quantum number already introduced earlier to help distinguish between hadrons. The method works well and is often seen as the beginning of modern particle physics. Furthermore, it set the stage for today’s quark model: Experiments performed in the 70s where highly energetic electrons were scattered off protons revealed the structure of the proton. The discovery of the J/ψ meson in 1974 is often regarded as overwhelming experimental evidence in favor of Gell-Mann’s theory. The quark model could explain the existence of the J/ψ by adding a new quark flavor (called “charmed”). It was also able to predict more mesons consisting of this new type which were later found by experiments. Today’s quark model was completed by the discovery of the bottom quark (around 1981) and the top quark in 1995. First experimental evidence of the gluon, which mediates the strong interaction between quarks, was provided by the PETRA accelerator in 1979.

However, new discoveries not only in the strong sector but also in weak interactions were made. The original theory of Fermi (1933) considering the beta decay was valid only for low energies. In the 60s Glashow, Weinberg and Salam developed a new model which predicted

heavier vector bosons. These particles (two charged W bosons and one neutral Z boson) were expected for a long time and were finally discovered at the SPS in 1983. In order to explain why these particles are massive (unlike the photon and the gluon), Brout, Englert and Higgs developed what is known as the ‘Brout-Englert-Higgs mechanism’ in the 60s. This theory also predicts an additional particle, the Higgs boson which was finally discovered at the LHC in 2012.

2.2 The Standard Model

All the steps mentioned above and many more lead to the current model of particle physics – the Standard Model. The following section is meant to give a short overview of this model, a deeper introduction is for example given in [3]. The Standard Model describes the electroweak and strong interactions as well as interactions with the Higgs field. Only gravity is not included, but it is too weak to have a noticeable impact at the physics scales studied at modern particle accelerators.

The Standard Model is a renormalizable, relativistic quantum field theory that describes all known fundamental particles and their interactions. Virtually all predictions of the Standard Model are in excellent agreement with the measurements. Although there is already some evidence of physics beyond the Standard Model (e.g. neutrino oscillations, dark matter) and new extensions of the Standard Model, such as supersymmetry, are being studied, the Standard Model still serves as the reference model.

Like many other quantum field theories the Standard Model is also a gauge theory. The corresponding symmetries are based on the symmetry groups $SU(2)_I \otimes SU(1)_Y \otimes SU(3)_C$. Here $SU(2)_I \otimes SU(1)_Y$ is the symmetry group of the electroweak unification and $SU(3)_C$ corresponds to quantum chromodynamics (QCD). According to Noether’s theorem a conserved quantity can be assigned to each particle participating in one of these interactions due to the underlying symmetry.

2.2.1 The Electroweak Interaction

In the Standard Model the weak and electromagnetic interactions are unified to the electroweak interaction as described by the model of Glashow, Weinberg and Salam. In this unified theory there are four massless gauge bosons which mediate the interaction: one B^0 boson (weak isospin singlet) and three W^i bosons ($i = 1, 2, 3$; weak isospin triplet). Here the weak isospin I is the conserved quantity of the $SU(2)_I$ group mentioned above, while the weak hypercharge is the one corresponding to the $SU(1)_Y$.

Through electroweak symmetry breaking the B^0 and the $W^{1,2,3}$ states are rotated by the weak mixing angle (also known as the Weinberg angle) into the physical Z , $W^\pm$ and γ bosons. A $SU(1)_Q$ symmetry remains with the electric charge Q as the corresponding conserved quantity.

This electric charge Q is related with the weak hypercharge Y and the third component of the weak isospin I_3 by $Q = I_3 + Y/2$.

All gauge bosons can only couple to particles with a nonzero corresponding charge. This means that the photon can only interact with electrically charged particles. For the Z and W bosons the situation is more complicated, since the weak interaction violates parity. In order to describe this correctly one has to introduce chirality. Chirality divides all fermions into “left-handed” and a “right-handed” particles¹. The left-handed particles can be written as doublets of the weak isospin ($I = \frac{1}{2}$, the right-handed particles are singlets ($I = 0$)). The W boson can only couple to left-handed fermions and right-handed antifermions, whereas the Z couples to all of them.

In the Standard Model the electroweak symmetry is broken spontaneously by the Brout-Englert-Higgs mechanism. This has several consequences: on the one hand the coupling to the Higgs field is the reason why the W and Z bosons are massive, while the photon remains massless. Furthermore, the mechanism predicts a new particle which can be interpreted as an excitation of the Higgs field: the Higgs boson. It couples to all massive particles, the coupling strength being proportional to the mass of the particle it interacts with.

2.2.2 The Strong Interaction

The strong interaction, described by quantum chromodynamics (QCD), is the interaction of quarks and gluons². As mentioned above each symmetry is related to a conserved charge, in QCD this is called color charge. There are three different colors, called red, blue and green.

The QCD gauge bosons are gluons and they are color-charged, too. Each gluon carries both a color and an anticolor charge. In total there are eight gluons which differ only in their color configuration. Gluons have the ability to couple to themselves, there are vertices (interaction points) with three and four gluons.

However, QCD has some more special properties.

The first one is called “confinement”, meaning that color-charged particles cannot be observed as free particles and all observable hadrons are color-neutral: baryons consist of three differently color-charged quarks and mesons of two quarks with color and corresponding anticolor charge.

The second feature is dubbed “asymptotic freedom” and refers to the property of QCD to become asymptotically weak with increasing energy which limits the strong interaction to very short distances.

¹With one exception: there is still no evidence of the existence of a right-handed neutrino.

²Quarks and gluons are often referred to partons.

2.2.3 Fundamental Particles

In the Standard Model there are – ignoring all the antiparticles and different charges – 17 fundamental particles. Four of them are gauge bosons (the photon, the gluon and the W and Z bosons) and have spin 1, the Higgs boson as mentioned in Section 2.2.1 has a special role as it is the only particle in the Standard Model with spin 0. The properties of the gauge bosons are summarized in Table 2.1.

All remaining particles in the Standard Model are fermions which have spin $\frac{1}{2}$. One distinguishes between leptons and quarks: Quarks are particles which can interact both strongly and electroweakly, whereas leptons can only interact electroweakly. Both fermion types can be divided into three generations, where the first generation is the one that has the smallest masses associated with it, while the others can be considered as heavier copies of the first.

The properties of the leptons are shown in Table 2.2 and the properties of the quarks are shown in Table 2.3. All masses are approximate, the corresponding errors / limits can be found in [4].

name	interactions	spin	mass [GeV]	width [MeV]
gluon	strong	1	0	stable
photon	electromagnetic	1	0	stable
$W^{+/-}$	electroweak	1	80.385 ± 0.015	2.085 ± 0.042
Z	weak	1	91.1876 ± 0.0021	2.4951 ± 0.0023
H	weak	0	125 ± 0.21	4.21 (predicted)

Table 2.1: Properties of the Standard Model bosons. Data taken from [4] and [5].

generation	name	charge [$ e $]	mass [MeV]	lifetime
1	electron (e)	-1	0.511	stable
1	electron neutrino (ν_e)	0	$< 2 \cdot 10^{-6}$	stable ?
2	muon (μ)	-1	105.66	$2.197 \cdot 10^{-6}$ s
2	muon neutrino (ν_μ)	0	$< 2 \cdot 10^{-6}$	stable ?
3	tau (τ)	-1	1776.82 ± 0.16	$3.4 \cdot 10^{-13}$ s
3	tau neutrino (ν_τ)	0	$< 2 \cdot 10^{-6}$	stable ?

Table 2.2: Properties of the Standard Model leptons. Data taken from [4]. In the Standard Model neutrinos are assumed to be massless but this has been disproved experimentally.

generation	name	electric charge [$ e $]	bare mass
1	up (u)	$+ 2/3$	2.3 MeV
1	down (d)	$- 1/3$	4.8 MeV
2	strange (s)	$+ 2/3$	95 MeV
2	charmed (c)	$- 1/3$	1.275 GeV
3	bottom (b)	$+ 2/3$	4.18 GeV
3	top (t)	$- 1/3$	173.5 GeV

Table 2.3: Properties of the Standard Model quarks. Data taken from [4].

2.3 Experimental and Theoretical Methods in Particle Physics

2.3.1 The Large Hadron Collider

In order to test the predictions of the Standard Model it is necessary to measure the particle properties and interactions experimentally. In the early years of particle physics cosmic radiation and bubble chambers were used to discover new particles. However, neither the intensity nor the energy were high enough to look for heavy particles such as the Higgs boson. For this reason one began to develop accelerators in the middle of the 20th century which were able to achieve much higher energies [6].

The accelerator with the highest available energy to date is the Large Hadron Collider (LHC) at CERN, the European Organization for Nuclear Research, near Geneva. It currently accelerates protons and antiprotons to a center-of-mass energy of 13 TeV, first achieved in May 2015 [7]. The main synchrotron is built in a 27 km long underground tunnel and uses over 1000 superconducting magnets to keep the protons in their lanes. The tunnel was also used by the previous electron-positron accelerator called LEP, where the properties of the W and Z bosons have been measured very precisely.

Altogether there are four big detector systems used to study collisions at the LHC. Whereas the ATLAS and CMS experiments are big all-purpose detectors, LHCb is specialized in measurements of hadron decays and ALICE in measurements of properties of the quark-gluon plasma.

Running such big experiments requires big effort: over 3000 people work on the ATLAS experiment alone, coming from more than 177 universities in 38 countries [8]. The detector is about 45 m long and more than 25 m high. Its weight is about 7000 tons. A schematic view is shown in Figure 2.1b. Furthermore, the computational challenge is enormous. In order to process all the incoming data, a complex trigger system is used which selects in fractions of a second what data is potentially interesting and should be stored. Nevertheless, the remaining data by ATLAS alone still fills 27 compact discs per minute.

All these numbers illustrate the big challenge that comes with the search for new particles and potentially new physics, but nevertheless, these big machines are only half the work: in order to make sense of the data, one always has to compare to a theory prediction. The next section explains in a simplified way how this can be done.

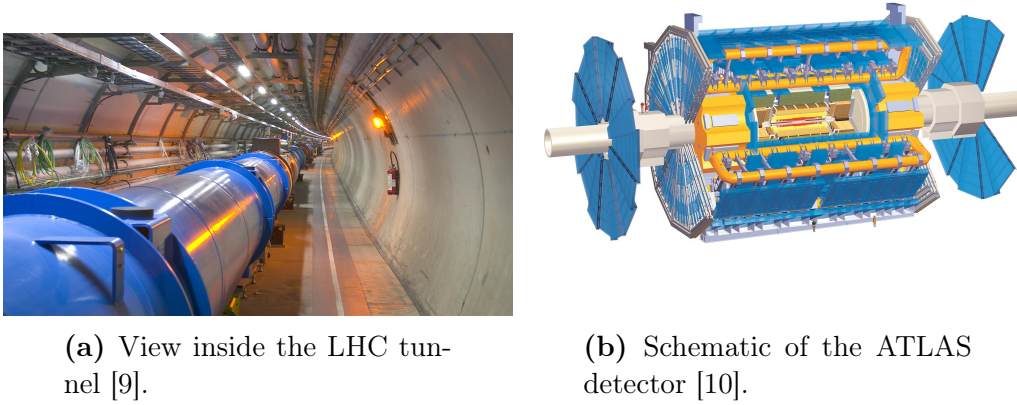


Figure 2.1

2.3.2 Feynman Rules, Cross Sections and Perturbation Theory

The following is meant to give a small introduction to the basic concepts of theoretical calculations in particle physics. It only covers what is necessary for the understanding of this thesis. Further information can be found in nearly all introductory books about particle physics (e.g. [2]) or quantum field theory (e.g. [11]).

One often uses Fermi's golden rule to begin theoretical calculations concerning interactions between particles. It describes how the transition rate depends on the squared absolute matrix element and the available phase space:

$$\text{transition rate} \propto |M|^2 \times (\text{phase space factor}). \quad (2.1)$$

Using the transition rate it is possible to calculate cross sections and decay rates. The total cross section σ specifies the probability that a given process takes place and has the unit barn, where 1 barn is equal to 10^{-28} m^2 . It is directly proportional to the transition rate:

$$\text{transition rate} = \sigma \cdot L. \quad (2.2)$$

The proportional factor L is the luminosity which is a measure for the number of incoming particles per unit time and unit area. Using Equation (2.1) and (2.2) one can express the differential cross section for a $1 + 2 \rightarrow 3 + 4 + \dots + n$ process as a function of the squared absolute matrix element $|M|^2$ and the Lorentz invariant phase space elements $d\Phi_i$:

$$d\sigma \propto |M|^2 \prod_{i=3}^n d\Phi_i. \quad (2.3)$$

At hadron colliders the initial particles are partons (quarks or gluons) and they only carry fractions of the hadrons' original momenta. As a consequence the integral over these fractions also contributes when evaluating Equation (2.3).

Generally speaking, differential cross sections can be interpreted as projections of the total cross section into different phase space regions. The resulting distributions have a big importance in particle physics and are often used to compare theoretical predictions with measurements. However, when evaluating Eq. (2.3) one first has to calculate the matrix elements. This can be done using Feynman rules. Feynman rules are a recipe for calculating matrix elements M from Feynman diagrams which are graphical representations of processes. An example of a Feynman diagram for the production of a Z boson from two leptons which subsequently decays into two quarks is given in Figure 2.2a. One distinguishes between inner lines, external lines and vertices. Feynman rules relate mathematical expressions to each of these components and also how to combine them to give an amplitude.

However, to calculate this process exactly it is not sufficient to only consider Figure 2.2a. In Equation (2.1) and also in (2.3) one always has to calculate all possible Feynman diagrams, all amplitudes are summed and the squared absolute value is taken afterwards. In theory, however, there are infinitely many different diagrams which describe the same main process, so that it is not possible to calculate the cross sections analytically.

Perturbation theory is a widely used approximation. In perturbation theory one orders all diagrams by the number of vertices. When the coupling constant is small ($g < 1$), diagrams with more vertices are suppressed. This method works fine for many applications, although there are some cases when either the coupling constant becomes too high (e.g. when considering hadronization, which means the formations of hadrons from quarks) or large logarithmic terms cancel the decreasing coupling terms. Diagrams with the lowest possible number of vertices are referred to as ‘leading order’ (LO) or ‘tree level’, the next order is ‘next-to-leading order’ (NLO) and so forth. Figure 2.2a shows only the LO contributions, two examples of the NLO QCD contributions are pictured in Figure 2.2b and Figure 2.2c. Only the squared matrix elements determine the order and hence also virtual diagrams with two additional vertices are part of the NLO correction as they can be multiplied with the LO diagram.

Up to now some processes have already been calculated at NNLO in QCD (e.g. $pp \rightarrow WW$, $pp \rightarrow ZZ$), but only one process ($gg \rightarrow H$) at NNNLO in QCD. This process is quite important as it is the main production channel of the Higgs bosons at the LHC. The NNNLO correction still has an effect of 2.2 % [12]. A more detailed overview of the progress in higher-order calculations is given in [13].

All these calculations are very important to be able to compare experimental results with theoretical models. Nevertheless, these calculations alone are insufficient due to the fact that the interacting particles are partons, whereas only hadrons can be measured by detectors. Solving this dilemma requires a combination of different approaches, namely event generators which will be presented in the next section.

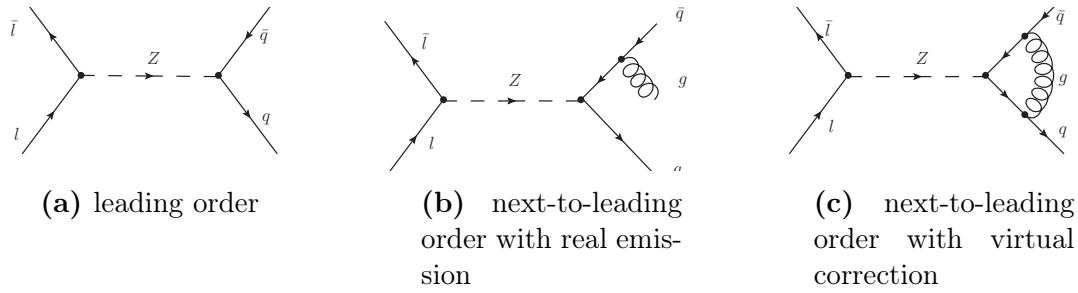


Figure 2.2: Feynman diagrams of Z production where the Z boson decays into two quarks.

2.3.3 Monte Carlo Event Generators

As introduced in Section 2.3.2 perturbation theory is a powerful tool to calculate expectation values of observables in quantum field theories. However, processes as measured at the LHC have many more aspects which have to be considered. Many processes involve final states with a large number of particles. Calculating these analytically is often not feasible, but there are even bigger problems. The colliding particles as well as most of the particles measured by detectors are hadrons, whereas only partons are involved in the hard process. Collision events need to be fully hadronized in order to properly evaluate the measured data or to simulate detector effects.

Calculating such events analytically is difficult for reasons mentioned already in Section 2.2.2 and 2.3.2: It is not possible to measure free, color-charged particles, but it is also infeasible to calculate the transition from highly energetic partons to low-energy hadrons analytically. By contrast an event generator is a piece of software that has been explicitly developed to approximate these calculations numerically. They are widely used and have great importance for data analyses (e.g. for background estimation and the calculation of significances) as well as for theorists who make predictions for collider experiments.

Nowadays many different generators are available for many several applications. This thesis is done with SHERPA (**S**imulation of **H**igh-**E**nergy **R**eactions of **P**Articles), an all-purpose event generator which combines all the essential steps into one program. The software is available for free from [14]. Further information about its structure and methods are published in [1].

The basic idea of all event generators is similar and sounds simple. The whole event is divided into several subprocesses characterized by different energy scales. An example is shown in Figure 2.3 where one Higgs boson and two top quarks are produced. Partons coming from the incoming protons can radiate further partons and interact in a hard scattering process, drawn in red. This hard scattering process has the highest energy scale and is usually calculated using perturbation theory at some fixed order. The outgoing particles of this hard scattering process tend now to emit lower energetic particles like quarks and gluons, shown in blue. These emissions are often collinear (i.e. close in angle) or soft (i.e. low in energy), the resulting cascade of particles is called ‘jet’.

This emission process starts at relatively high energy scales and evolves down to the hadronization regime. It is implemented through an algorithm called parton shower, which also describes the initial radiation before the hard process. Parton showers are able to resum the leading logarithmic terms in the perturbative series to all orders and describe radiations exactly if they are collinear. Many splittings are indeed nearly collinear which makes this procedure a good approximation. Finally, if all partons are hadronized (bright green), they can still decay into even lower energetic particles (dark green). Furthermore, some photons can be radiated through bremsstrahlung, shown in yellow, but also the remainder of the incoming particles can interact (multiple parton interactions, drawn violet) [15].

The concept of “divide and conquer” works well, but it has its difficulties when it comes to connecting all the separate steps.

One aspect is the subject of this thesis and will be presented in more detail. Both the hard scattering process (determined by the matrix element) as well as the parton shower determine the distribution of final-state particles. The number of jets is an important criterion when looking for new physics and depends strongly on the parton configuration. In order to describe this as accurately as possible one can include tree-level diagrams with additional final-state particles participating in the hard scattering. This makes it necessary to keep track of parton configurations in order to avoid double counting.

A solution to this problem is given by the CKKW[16] algorithm which divides the phase space into a part for jet production and a part for jet evolution. The method is presented in some more detail in [17]. This approach motivates further attempts at making calculations of processes with many final-state particles more efficient (or even possible) and has also motivated the studies presented in this thesis.

Of course it would be preferable to calculate the matrix element of the hard scattering process directly to higher orders and merge it with the parton shower. In 2012 it has been possible for the first time to do that up to next-to-leading order in an automated way [18]. Higher orders are a huge computational challenge and remain work in progress.

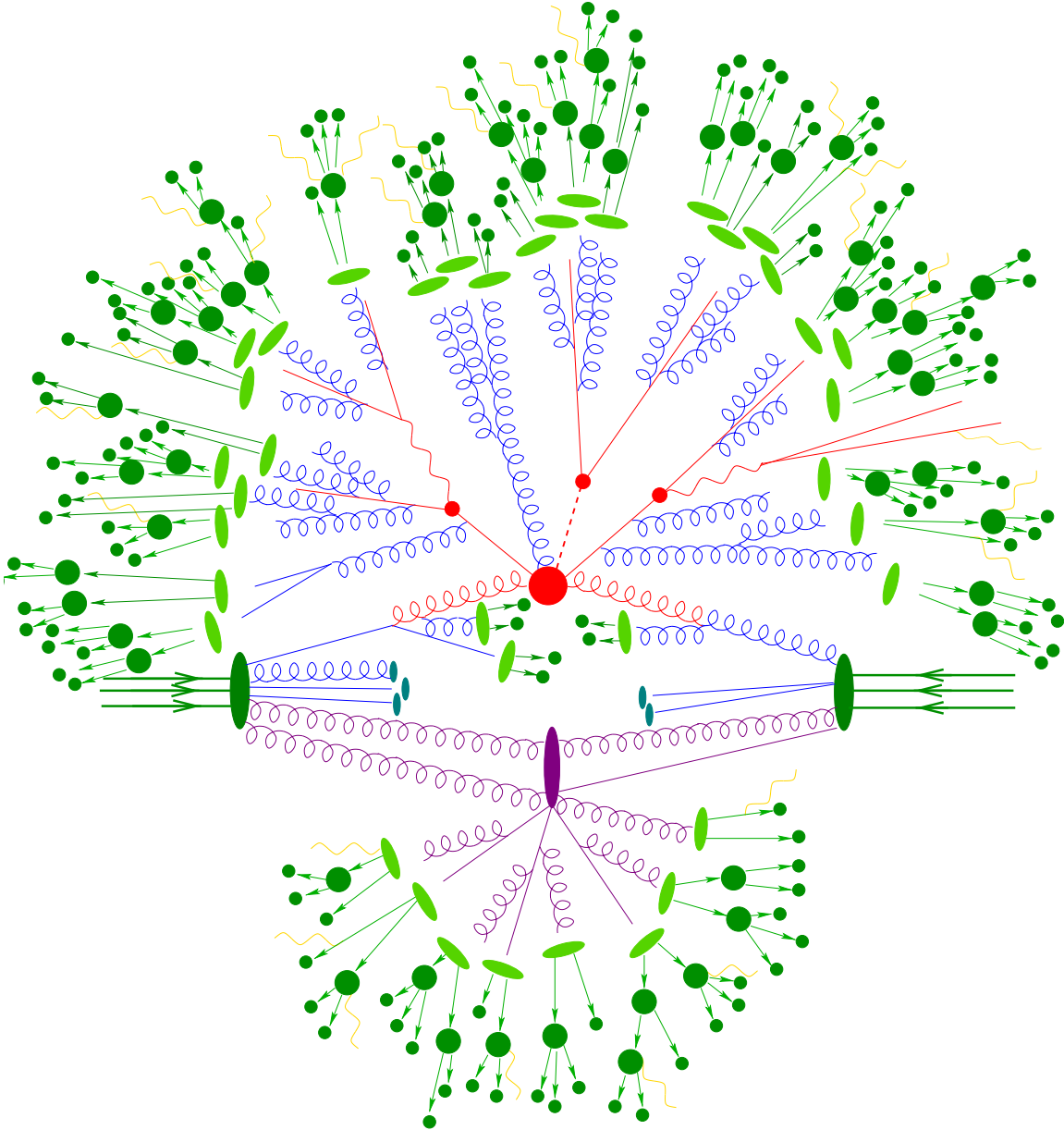


Figure 2.3: Steps in the simulation of the production of a Higgs boson and two top quarks [19].

3 Algorithms and Methods

3.1 Monte Carlo Methods

In the previous section Monte Carlo event generators were introduced without saying anything about Monte Carlo methods themselves. However, they are significant, especially in the context of this thesis which deals with the performance of event generators. This section gives a small overview of the basics, focusing on methods relevant for this thesis.

Generally speaking, Monte Carlo methods are an umbrella term for numerical methods using (pseudo-)random numbers. First experiments in this direction have already been conducted by Fermi in the 1930s, when he studied neutron diffusion. However, the first modern approaches, done with random numbers from tables, arose in the late 40s at Los Alamos, USA. Since this time – and with the increasing availability of computers – Monte Carlo methods have made great progress and are being used today in many different fields. Details of the historical evolution are given in [20]. A simple example on how the number π can be calculated using random numbers is shown in Figure 3.1.

Especially methods for integration and sampling are crucial for the usage in event generators. Both will be discussed next.

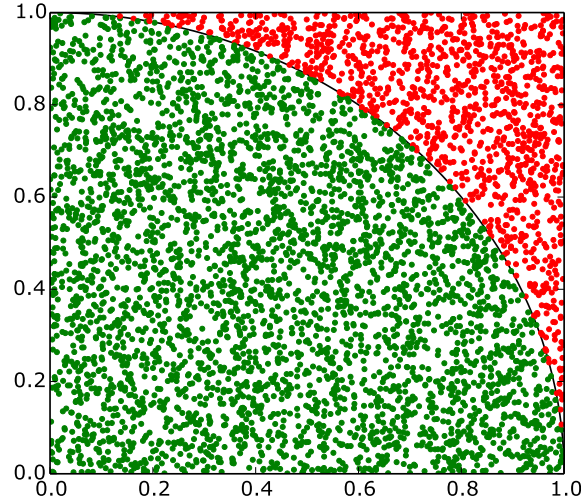


Figure 3.1: Monte Carlo approximation of π with $n = 5000$ points. Pairs of random numbers were generated and filled in a unit square. After counting all k points in the quarter unit circle, π can be estimated as $4 \cdot \frac{k}{n}$. This example gives an estimate of $\pi \approx 3.0896$.

3.1.1 Monte Carlo Integration

As shown in Section 2.3.2, calculating cross sections requires the integration over all momenta of the final-state particles, but also over the energy fraction of the incoming partons. In a $2 \rightarrow 6$ process this results in 20 dimensions that are to be integrated over. Numerical quadrature rules are uneligious for this kind of integration. The error bound of e.g. the trapezoidal rule scales with $\mathcal{O}(N^{-2/d})$, where N is the number of sampling points and d the number of dimensions. This behavior makes it impossible to achieve the desired accuracy in a feasible CPU time.

A solution is offered by Monte Carlo integration. Here the integral over a volume V is approximated with the function averages over N random points [21]:

$$I = \int dx_1 \dots \int dx_N f(N) \approx V \cdot \frac{1}{N} \sum_{i=1}^N f(x_i) \pm \sqrt{\frac{\langle f(x)^2 \rangle - \langle f(x) \rangle^2}{N}}. \quad (3.1)$$

The variance of this approach always scales with $\mathcal{O}(N^{-1})$ which is independently of the number of dimensions, making this method the best choice for high-dimensional integration problems. Nevertheless, a N^{-1} -convergence is still quite slow. For this reason many methods were developed to reduce the variance's prefactor, some of which are importance sampling and multi-channel integration.

Importance Sampling / Multi-Channel Integration

The approach in Equation (3.1) is not well suited for functions with peaks. If the function has peaks, they will probably be underrepresented in the sum and thus lead to a high variance. In event generators, the function – which contains the absolute squared matrix elements –

has many such elements. There are often several Feynman diagrams with different possible resonances, each propagator term represents a peak.

Functions with a peak or significant structure can be approximated well with importance sampling. Importance sampling requires a change of integration variables. If $f(x)$ is the function to be integrated, one looks for a well known function $g(x)$ with a similar peak structure. The Monte Carlo step is evaluated with $f' = f/g$ as new function and random numbers distributed according $G(x) = \int dx^N g(x)$ instead of uniformly distributed numbers as in Equation (3.1).

This method holds only if all peaks can be mapped into one single auxiliary function, which still must be simple enough to generate random numbers. In case of processes with many contributing Feynman diagrams this method can not be applied.

A solution for functions with a more complex peak structure is provided by multi-channel integration. Here the auxiliary function is composed as weighted sum of m different auxiliary functions, each of these functions representing one peak or structure:

$$g(x) = \sum_{i=1}^m \alpha_i \cdot g_i(x) \quad , \quad \sum_{i=1}^m \alpha_i = 1. \quad (3.2)$$

The integral can then be written as follows:

$$\int dx f(x) = \sum_{i=1}^m \alpha_i \int \frac{f(x)}{g(x)} dG_i(x). \quad (3.3)$$

Using Eq. (3.1), the Monte Carlo estimate E follows as:

$$E = \frac{V}{N} \sum_{j=1}^m \sum_{i=1}^{N_i} \frac{f(x_i)}{g(x_i)} \quad , \quad N_i = \alpha_i \cdot N. \quad (3.4)$$

As well as for importance sampling the second sum has to be evaluated using random numbers distributed according to $G_i(x)$. This method often yields a huge reduction of variance, but the success is of course strongly dependent on the choice of channel weights, α . α_i represents the probability of selecting channel i and has to match the relevance of the corresponding structure in the function f . An automated approach to adjust these weights is presented in [22], a widely used approach implemented also in SHERPA. Of course, there are many other methods for variance reduction in Monte Carlo integration. An overview of these as well as more details on importance sampling and multi-channel integration are given in [21].

3.1.2 Monte Carlo Sampling

The previous section dealt with the problem of high-dimensional integration as needed for calculations of cross sections. A second important part where event generators use Monte Carlo methods is the event generation. A naive approach would be to generate uniformly

distributed random momenta and simulate their evolution, but these events would not follow the physical probability distribution and therefore have to be corrected by using weights. These weights are often undesirable. When creating histograms for e.g. differential cross sections, each event contributes according to its weight. Hence weights which have large differences can cause large errors.

Solving this problem is the aim of Monte Carlo sampling methods [21]. Technically one has a known probability distribution p and wants to generate random numbers distributed according to p . If the function p is simple enough, one can use e.g. the inverse transformation method. It uses the inverse of the distribution function ($P(x)^{-1}$) to create random numbers distributed according to p . However, this method requires the function $P(x)^{-1}$ to be known which is often – and also here – not the case [21].

Acceptance-Rejection Method

A more general and very powerful method for generating random numbers according to a arbitrary complex probability distribution p is offered by the acceptance-rejection method, developed by von Neumann [23]. The method requires a well known distribution $h(x)$ with the condition $p(x) < C \cdot h(x)$, where C denotes a constant. Random numbers can then be generated using the following algorithm:

1. choose x according to $h(x)$,
2. choose a uniform distributed random number R : $0 < R < 1$,
3. if $(R \cdot C \cdot h(x) > p(x)) \rightarrow$ reject x , restart with 1,
4. accept x .

This method ensures all values x follow $p(x)$ [21]. It is a very powerful algorithm, but can result in poor performance if $h(x)$ differs much from $p(x)$.

3.1.3 Unweighting in SHERPA

In order to generate events with SHERPA many different algorithms have been combined. In a first step one uses the multi-channel separation (as already used for the integration as explained in Section 3.1.1), but also other methods as e.g. the RAMBO algorithm [24] to generate weighted events. The weight w of each event is given by the product of the absolute squared matrix elements ($\text{ME} = |M|^2$) and a phase-space weight PSW:

$$w = \text{ME} \cdot \text{PSW}. \quad (3.5)$$

These weights define a probability distribution p which is the desired distribution of events. In the second step – called ‘unweighting’ – this distribution is used to generate events with

uniform weights (‘unweighted events’). This is done by the acceptance-rejection method mentioned above. However, p is through its high-dimensional dependencies too complex to find an appropriate envelope $C \cdot h(x)$. For this reason one assumes $C \cdot h(x)$ as constant ($= w_{\max}$) and proceeds as described in Section 3.1.2.

This method has a big disadvantage. In order to receive one unweighted event a large number of weighted events have to be calculated until the acceptance condition is fulfilled. Furthermore, the calculation of w (i.e. of ME) is very expensive in terms of CPU time. The fraction of accepted events can be expressed by the unweighting efficiency $\epsilon = \frac{\bar{w}}{w_{\max}}$ which is on the order 10^{-4} or smaller for processes with many final-state particles.

At least a little improvement can be achieved by using ‘partially unweighted events’. This method allows for a small fraction of the data to be weighted. The constant $w_{\max}$ will be decreased and all events with $w > w_{\max}$ receive as new weight $\frac{w}{w_{\max}}$.

Nevertheless, the poor unweighting performance is still a huge problem. Improving this by means of machine learning is the aim of this thesis.

3.2 Introduction to Machine Learning

In recent decades the development of computer hardware has made great progress, due to which the amount of data which can be handled also increased rapidly. A powerful method to deal with large data sets is a class of algorithms able to improve their own performance during run time. These algorithms are often summarized under the name ‘machine learning’.

Many applications of these methods are already in use. One can find software that is able to read handwritten texts or recognize human voice, robots can drive autonomously and online shops try to predict consumer behavior.

A common classification for machine learning algorithms distinguishes between supervised learning, unsupervised learning and reinforcement learning [25] which is illustrated in Figure 3.2.

In supervised learning the algorithm receives a set of labeled training data, consisting of pairs of input values and the corresponding desired output values. Using this ‘experience’ the algorithm attempts to predict the output values for unseen input values. Examples are e.g. the price forecast of used cars or the decision whether or not customers are credit-worthy. Most supervised learning tasks can therefore be divided into ‘classification’ or ‘regression’ tasks. Classification tries to predict discrete values (e.g. ‘yes’ or ‘no’). These values are often called “classes”. Each data point belongs to one and only one class. On the other hand regression creates a continuous output (e.g. a price), it can be seen as the generalization to an infinite number of classes.

In the case of unsupervised learning no labeled data is available for training and so the aim is to find some feature or pattern in the data, e.g. clusters. Examples of cases where unsupervised

learning is used are customer profiles and image compression, but also the analysis of DNA sequences [26].

Reinforcement learning deals with problems where not a single decision has to be made, but rather a sequence of actions is supposed to deliver an optimal result. The aim is to generate a strategy by evaluating the success of foregone decisions. Typical examples are game simulations as well as robotics.

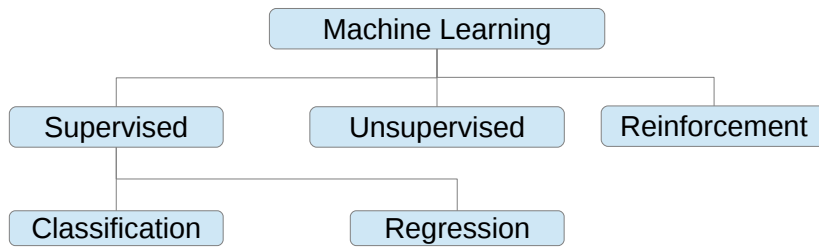


Figure 3.2: classification of machine learning methods

3.2.1 Concepts of typical Regression Algorithms

The following selection is meant to give a short overview of the ideas behind some of the most popular algorithms used for supervised regression. Of course there are much more of them available and also lots of variants of the presented algorithms exist. The selection of the chosen methods is motivated by a study from 2006, in which several supervised learning algorithms have been compared empirically [27].

***k*NN-Algorithm**

The *k*-Nearest Neighbors-algorithm was devised in 1951 [28]. Although it does not belong to the winners of the aforementioned study, it was one of the first and also simplest approaches for a non-parametric classification or regression. Nevertheless, it is still used today for many applications in bioinformatic or face recognition.

The idea of this method is straightforward: similar inputs shall give similar outputs. In order to predict a new point, the algorithm looks for the *k* nearest neighbors in the training data set. The output is then calculated as the average of the output values related to the determined neighbors [25].

The disadvantage of this method is that all distances between all points have to be calculated and stored, along with all the training data points. This results in a time complexity for the prediction of this algorithm of $\mathcal{O}(knd)$, where *n* is the number of training data points, each of dimension *d*, and *k* is the number of nearest (i.e. contributing) neighbors.

There are several ways to optimize this procedure, e.g. one can define a weighting function which increases the relative contribution of points with smaller distances. An overview of some

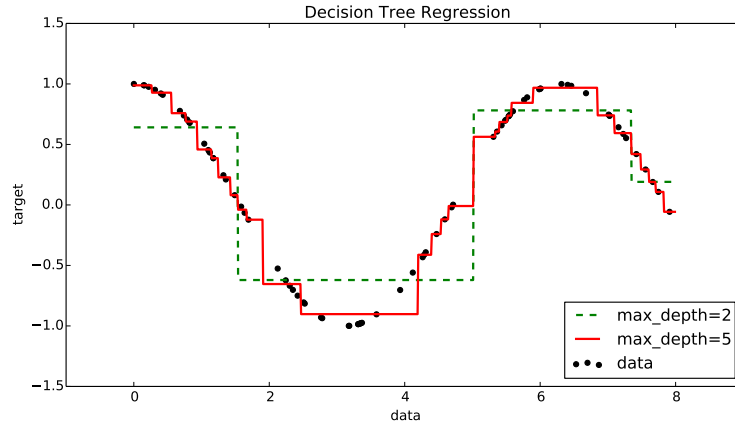
of the methods used nowadays is given in [29].

Decision Trees

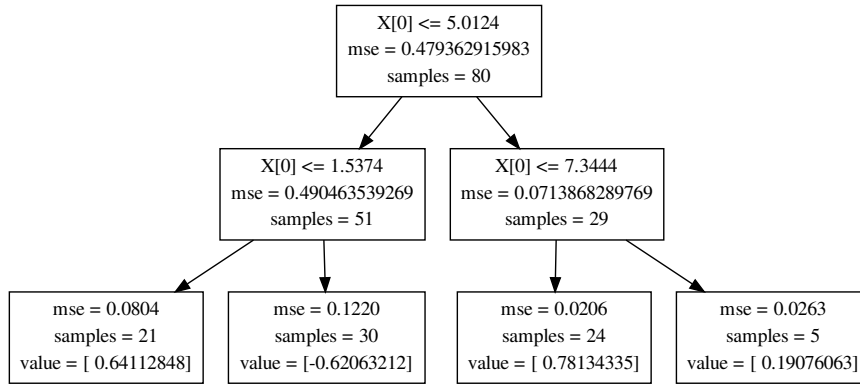
Decision trees are another class of algorithms used for both classification and regression which are easy to understand. The aim is to split a complex decision into a sequence of simple decisions, all possible decisions can then be represented by a tree structure. Decision trees are a descriptive example for supervised learning. The tree is optimized using some training data and all new data can then be processed by it. The big advantage of such a procedure is the possibility to understand and examine every assignment. The first algorithm that created such a tree automatically was the “CART”-Algorithm presented by L. Breiman in 1984 [30]. An example for a simple decision tree constructed by a variant of this algorithm is given in Figure 3.3.

The principle of the “CART”-algorithm is to make a decision at every classification node which maximizes the purity of the target class. In case of regression this impurity can be measured e.g. by calculating the mean square error between the known target values and the predicted ones in the following layer of nodes. The trick is to find an effective way of choosing the best of all possible decisions. The computational time complexity for constructing a decision tree is $\mathcal{O}(d \cdot n \cdot \log(n))$, where d is the number of dimensions and n the number of training points [31].

One problem with these algorithms is over-fitting, which means that the algorithm begins to model the statistical fluctuations and hence loses its ability to generalize. Decision trees can be very sensitive to outliers. There are several possibilities to handle this problem, e.g. one can require that every leaf still contains a certain percentage of data points, one can limit the depth of the tree or there are also certain methods of pruning the tree after the training, where pruning refers to the removal of certain subbranches which have only a vanishing relevance [25].



(a) Comparison for different values of the maximal depth



(b) Tree structure for maximum depth = 2

Figure 3.3: example: decision tree, $f = \cos(x)$, 80 training points

Support Vector Machines

Support vector machines are a powerful method for classification and regression, where the separation of the classes or the regression function, respectively, is only described by a subset of the available training data, the so-called ‘support vectors’ [25].

The aim of support vector regression is to find a flat function, such that all the given training data do not deviate further from the function than some user-defined distance ϵ . In case of a linear correlation this problem can be solved analytically. Using Lagrange multipliers, one obtains a linear equation describing the support vectors mentioned above. In case of nonlinearity one can use kernel functions. The features X are transformed nonlinearly into a higher-dimensional feature space where the linear regression is performed. The trick lies in not performing this mapping explicitly: It is sufficient to know the kernel function $k(x, x') = \langle \Phi(x), \Phi(x') \rangle$, where Φ is the mapping function. There are several conditions a kernel function has to fulfill. A deeper introduction to the theoretical concepts is given in [32]. Nevertheless, the choice of kernel function has a big practical importance as it determines the accuracy of

the regression as well as the time complexity. Popular kernel functions are e.g.

- linear function: $\langle x, x' \rangle$
- polynomial function: $(\langle x, x' \rangle + 1)^k$
- sigmoid function: $\tanh(\gamma \langle x, x' \rangle + r)$
- radial basis function: $\exp(-\gamma |x - x'|^2)$

It is difficult to specify the overall time complexity for support vector machines. The critical component is the number of support vectors R . Merely calculating them requires inverting a matrix, the calculation of which grows like $\mathcal{O}(R^3)$, verifying them is additional of $\mathcal{O}(R \cdot n)$, where n is the number of training samples. In practice, this results in a typical behavior between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ [33].

Neural Networks

Due to the unique human ability of learning, understanding the human brain has been an objective in medicine as well as in computer science since the beginning of research. Nowadays one thinks that the brain consists of a huge amount (10^{11}) of neurons working in parallel. They seem to be much simpler than e.g. a processor, but they are connected extremely well: every neuron has a fast connection to around 10^4 other neurons. This structure is the theoretical concept for artificial neural networks [25].

The simplest unit in neural networks is a perceptron, which was first introduced by R. Rosenblatt in 1958 [34] and is the computational equivalent of the biological neuron. It defines a hyperplane in the input space by assigning a weight to each input parameter and creating an output value, as shown in Figure 3.4. The output value will be calculated by an activation function $g(x)$, the simplest case is a linear function, but also using other functions like the sigmoid function has its advantages. This activation function is motivated by biology, too, as a certain input threshold is needed to activate the neuron [31].

In order to treat nonlinear problems, additional perceptrons can be added, arranged in intermediate ('hidden') layers, which is illustrated in Figure 3.5. Every node in the hidden layer has the same structure as in the single-layer perceptron. Complex functions can be modeled using multilayer perceptrons.

A big challenge when considering neural networks is the adaption of the weights. With an increasing number of neurons the number of weights increases rapidly and the final error depends on all of them. Minimizing this high-dimensional error function is the goal of the training phase. In order to reach this, several algorithms were developed.

One of them, which is very popular, is the backpropagation algorithm. Here, the network is first evaluated using some default, random values. After each iteration ('epoch') one compares

the output with the target result. Starting from the output node all weights will be adjusted according to their influence on this error afterwards. The error is therefore propagated back to the input values, how this works is shown in more detail in [35]. An overview of some modifications and newer developments which led to modified algorithms is given in [36]. Artificial neural networks generally have the highest accuracy (if trained properly) and have the advantage of making only few assumptions about underlying relationships. The disadvantages are a high computational effort and the huge number of hyper-parameters. First of all, the choice of the best algorithm hangs on the problem and data. Each algorithm has its own parameters which determine their actual performace in finding the minima. Furthermore, the scaling of the data as well as the choice of activation functions and the structure of the network play an important role.

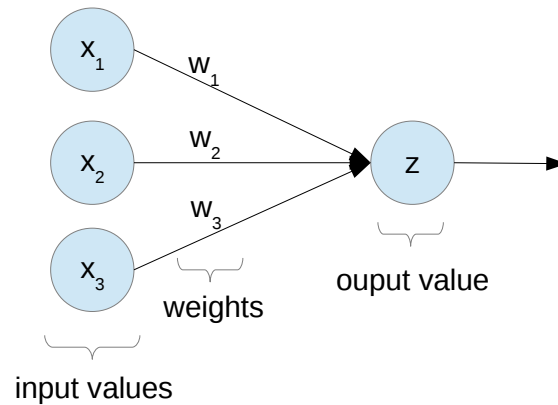


Figure 3.4: Example for a single-layer perceptron. The output value z will be calculated by $z = g(\mathbf{w}^T \cdot \mathbf{x})$, where $\mathbf{w}$ is the vector of all weights, $\mathbf{x}$ the input vector and g is the activation function.

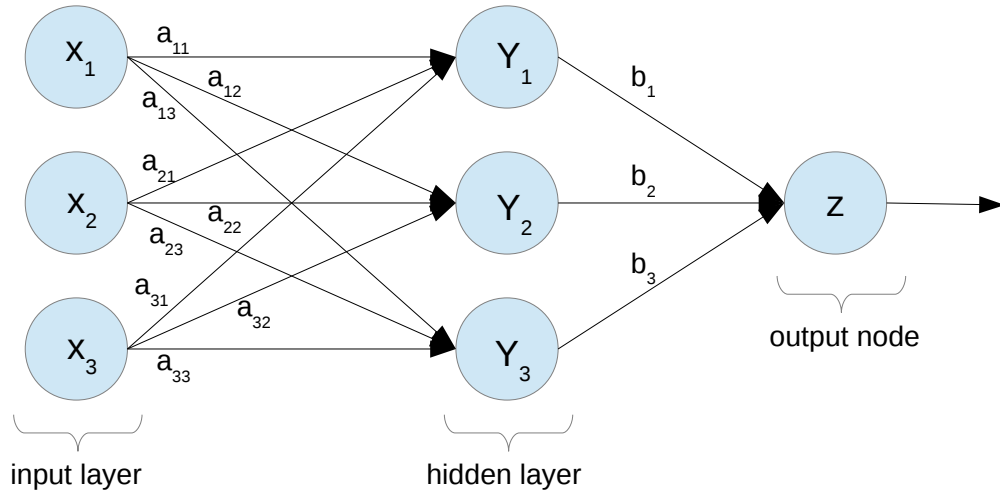


Figure 3.5: A multilayer perceptron with one hidden layer. The hidden values are given by $y_j = g(\sum_i a_{ij} \cdot x_i)$, the output value is then given by $z = g(\mathbf{b}^T \cdot \mathbf{y})$.

3.2.2 Bagging and Boosting – Meta Techniques

As described e.g. in [37] there are several techniques for combining methods like decision trees to improve the overall performance. The first approach, called ‘bagging’, was introduced by L. Breiman in 1996 [38]. Applying this technique is the first step to obtain a random forest as described below.

Another famous and often used method is ‘boosting’, which can improve machine-learning algorithms via refining the training procedure by adjusting the node weights, so as to concentrate more on optimizing nodes that do not perform as well as others [37].

Random Forests

As mentioned in Section 3.2.1 decision trees sometimes suffer from overfitting. Random forests can deal with that problem and also improve the overall accuracy [25]. The idea of a random forest was first published by L. Breiman in the year 2001 [39].

A random forest consists of t decision trees, each constructed from a random subset of the training data. This method is called bagging. Furthermore, each tree can be partly randomized by only using a subset of features at each node. This means that even with the same random subset of training data different trees will be trained. In order to predict new values all trees are used. The forest votes which result is the most likely [39].

Random forests show good performance on data with high dimensionality and many data points. They can also deal well with noisy or missing data. Due to this they have become one of the most popular type of algorithms in machine learning [37].

Considering the performance of a single decision tree, the time complexity for the training of a random forest consisting of t trees can be approximated with $\mathcal{O}(t \cdot d \cdot n \cdot \log(n))$, where n

denotes the number of training points of dimension d .

Boosted Decision Trees

Just like a random forest a boosted decision tree uses an ensemble of trees to improve the performance. An introduction to one of the most widely used boosting algorithms called AdaBoost is given by Freund and Schapire in [40].

A boosted decision tree trains a decision tree t times on the entire training data set. At the beginning all data points are uniformly distributed. After each iteration all misclassified samples receive a higher weight, in case of regression the weight is increased if e.g. the error exceeds a certain threshold. In the next iteration a new tree is initialized using the adjusted weights. Moreover, every tree is assigned a weight according to its performance. The final prediction is then calculated from the weighted sum of all of the individual trees [25].

Boosted decision trees have many advantages. They are simple and have only a few parameters to adjust. In principle the performance of every machine-learning algorithm can be improved by boosting, providing their performance is better than pure chance [40]. Of course, the actual performance also depends on the data.

4 Machine Learning based Unweighting

4.1 Basic Idea

As mentioned in Section 3.1.3, the current unweighting procedure suffers from huge performance losses for many processes. The main problem hereby is caused by the expensive calculation of a large number of weights, which finally cannot be used by failing the acceptance condition.

However, as shown in Section 3.2 there are many algorithms which are able to learn complex functions to make fast and precise predictions. The power of these algorithms is meant to be used and make the unweighting procedure perform better by using approximated, fast weights (g) for the acceptance condition.

Of course, the algorithm's approximation will not always be correct. Ignoring this would cause errors, an imprecise prediction changes the probability to survive the acceptance condition. The only possible way to correct this lies in the assignment of a new weight. In order to compensate this possible error, the new weight has to be $\frac{w}{g}$, where g is the prediction and w the exact weight. This can be shown by comparing the sum over the acceptance probability (p_i) times the final weight (t_i) for the classic unweighting procedure ('cl') and the machine learning based approach ('ml')

$$s^{cl} = \sum_i p_i^{cl} \cdot t_i^{cl} = \sum_i \frac{w_i}{w_{\max}} \cdot 1 = \sum_i \frac{w_i}{w_{\max}} = \sum_i \frac{g_i}{w_{\max}} \cdot \frac{w_i}{g_i} = \sum_i p_i^{ml} \cdot t_i^{ml} = s^{ml}. \quad (4.1)$$

The machine learning based attempt would also allow the usage of a maximum which differs from $w_{\max}$ such as $g_{\max}$. In order to still be correct, all final weights have then to be multiplied with $\frac{g_{\max}}{w_{\max}}$. The algorithms for the generation of one unweighted event are for both the classic as well as for the machine learning based attempt compared in Figure 4.1.

If the approximation is not perfect – which cannot be expected – the final events of this machine learning approach are, in contrast to the real unweighted ones, still weighted. However, they will hopefully have a much smaller variance than the previous weighted events while not being much slower in calculation. This opens up many possibilities:

First, they could replace the weighted events in all applications where they are used directly, e.g. in predictions by theorists. Theorists often use event generators to obtain simulated differential cross sections for a given theory, these differential cross sections can be obtained by entering all generated events into histograms by weighting each event with its actual weight. With the new method such events may be obtained with a lower weights' variance, leading to a faster generation for a predefined error.

Second, if the prediction is good, the variance of the final weights might be low enough to use them also for experimental issues such as detector simulations. Only the detector answer can be measured when studying processes at colliders. In order to compare simulated events with measured ones, the detector behavior of the simulated events has to be simulated, too. These detector simulations cause usually much computational effort, making it more efficient to use less weightless events instead of many weighted ones. This situation can change if the differences in the weights are only small.

Finally, it is also possible to obtain real unweighted events by the machine learning based attempt. A second unweighting can be performed with the distribution of new weights $\frac{w}{g}$ as basis. If the distribution is narrow enough, this could be more efficient than the current implementation.

These possible applications define the requirements of the desired machine learning solution. The prediction has to be exact but also fast.

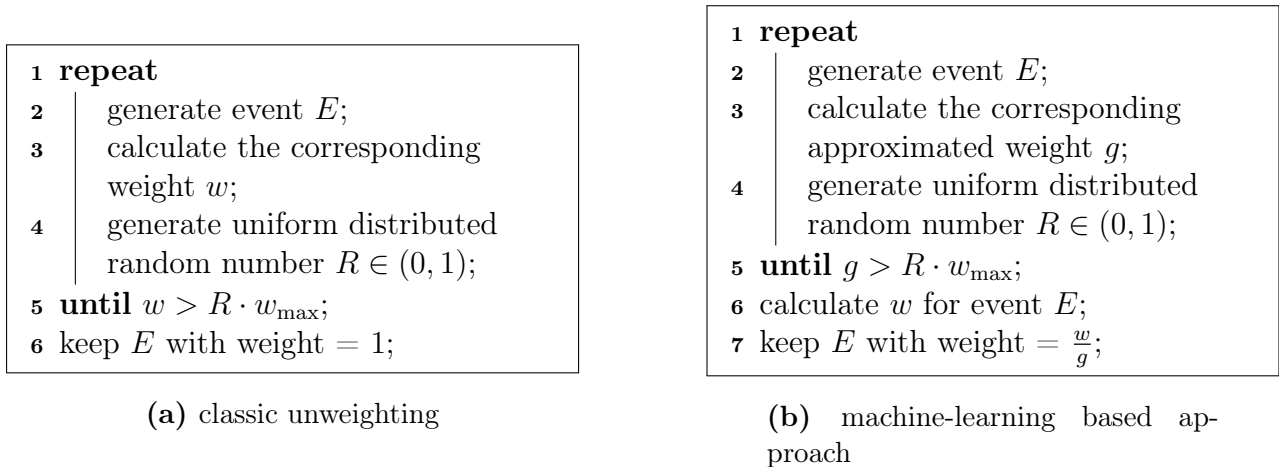


Figure 4.1: Comparison of the classic and machine-learning based unweighting attempts.

4.2 Choice of Algorithms and Hyper-Parameter Optimization

The search for a suited machine learning solution can be divided into two main steps. The first optimization phase (this chapter) is about finding a machine learning solution, which is able to predict accurately and fast. By contrast, in the subsequent evaluation phase (Chapter 5) will be discussed, whether and how the optimal solution, found in the first step, can improve the unweighting performance of SHERPA.

The search for an optimal machine learning solution requires many different steps. First, one has to select software which provides algorithms. The software used for this thesis is described in Section 4.2.1. Furthermore, training data have to be generated (described in Section 4.2.2) and a measure for the goodness of the prediction (f) has to be defined (described in Section 4.2.3). These steps are the foundation of the optimization phase, afterwards f can be minimized by extracting features from the training data and comparing algorithms and parameters.

4.2.1 Software

In order to experiment with machine learning, there are many different open source packages available. Three of them were used in this thesis. The first one is called ‘scikit-learn’ [41] and is a well documented and maintained Python package. It is available through the standard Python repositories and offers many useful, but also easily usable tools for data preprocessing and a huge choice of available algorithms. Unfortunately it does not support artificial neural networks for supervised learning¹.

The second software used is ‘Auto-WEKA’ [42]. Auto-WEKA is a powerful software, able to find an optimal algorithm and adjust its hyper parameter settings by itself, using sequence model based optimization algorithms. How these algorithms work is explained in more detail in [43]. Auto-WEKA is based on the ‘WEKA’ package, both are written in Java and have a graphical interface. They offer a large choice of algorithms (also artificial neural networks), but Auto-WEKA does not support the Random Forrest algorithm for regression².

The third software package which was used is the ‘FANN’ library [44], where ‘FANN’ means ‘Fast Artificial Neural Networks’. It is written in *C*, but bindings to Python are available. FANN is – as indicated by its name – specialized in the training of neural networks and offers a great flexibility to configure them.

¹state of July 2015, scikit-learn – version 0.16.1

²state of July 2015, Auto-WEKA – version 0.5

4.2.2 Generation of Training Data

Supervised regression algorithms need to be trained with labeled training data. This section describes how this data was generated and which process was used during the optimization phase.

The task of the regression lies in the prediction of the weight $w = \text{ME} \cdot \text{PSW}$ by usage of some input parameters, called features. There are different ways to construct the features. A possible basis consists of the random numbers r_i which were used by SHERPA for the event generation. In a $2 \rightarrow n$ process $3n$ random numbers were generated for one event³. Starting from three degrees of freedom for each of n particles, two more are needed for the energy fraction of the initial-state particles⁴. Furthermore, two additional random numbers are needed for the channel selections (two multi-channel methods are used, one for the initial-state composition and another one for the final-state composition⁵), but four degrees of freedom can be removed by the conservation of energy and momentum. This results finally in $3n$ random numbers.

However, as well as these random numbers, also the four-momenta themselves, calculated by the random numbers, could serve as features. A priori, it is not obvious which of these possibilities will result in the best prediction, particularly one can also think of predicting ME and PSW independent and combine them afterwards. For this reason the training file should contain all random numbers, but also the momenta, the phase space weights PSW and the matrix elements ME.

In order to generate training data one still has to specify a process. As mentioned in Section 3.1.3, processes with many final-state particles have often a poor unweighting performance. Such processes have further an important role when looking for new physics. A possible extension of the Standard Model of Particle Physics are e.g. supersymmetric models which predict supersymmetric partners to all particles of the Standard Model. Searching such new particles requires good knowledge of processes which are expected to give similar results, called ‘background’. Especially for supersymmetric models this background are Standard Model processes with a high multiplicity, particularly processes with many jets. As described in Section 2.3.3, the accuracy of predictions with many jets can be increased by calculating the tree-level processes of such events.

For this reason such a process was chosen for the training, i.e. the process $gg \rightarrow d\bar{d}e^+e^-gg$. Figure 4.2 shows two of over 500 possible leading-order Feynman diagrams contributing to this process and Figure 4.3 the frequency distribution of weights, generated with 1 million sample points.

³Each on-shell particle has three degrees of freedom.

⁴According to their probability distribution in the colliding hadrons, defined by the so called ‘parton density function’. Details are given e.g. in [45]

⁵These channels have their origin in the multi channel integration as explained in Section 3.1.1, to put it simply each propagator term is mapped into a separate channel.

Of course, there are nearly infinitely many processes which are important and could profit from machine learning, but it is not feasible to optimize each process individually. Hence the machine learning based unweighting attempt can only be applicable if all processes can be treated by the same algorithm and a similar configuration. The optimization procedure is therefore done with only one representative process if the found configuration can be generalized has to be validated afterwards.

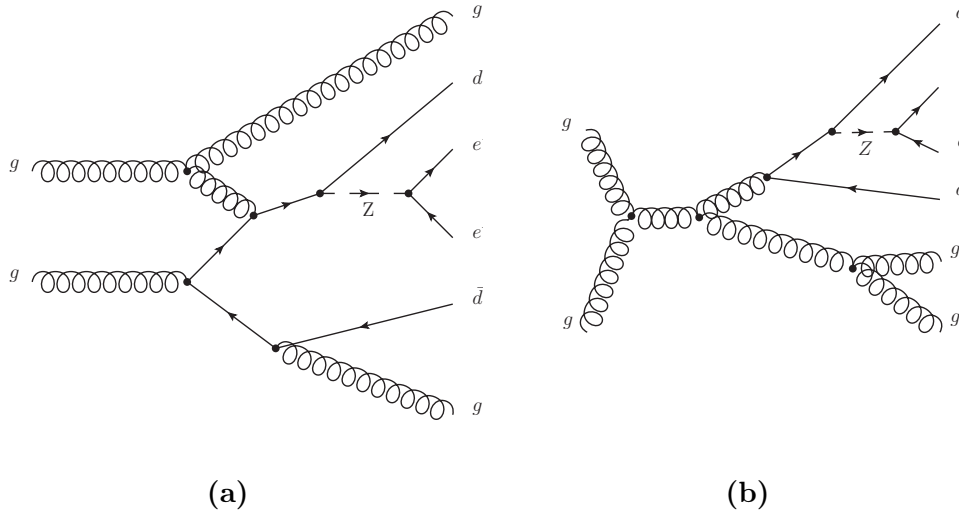


Figure 4.2: two Feynman diagrams, representing the $gg \rightarrow d\bar{d}e^+e^-gg$ process

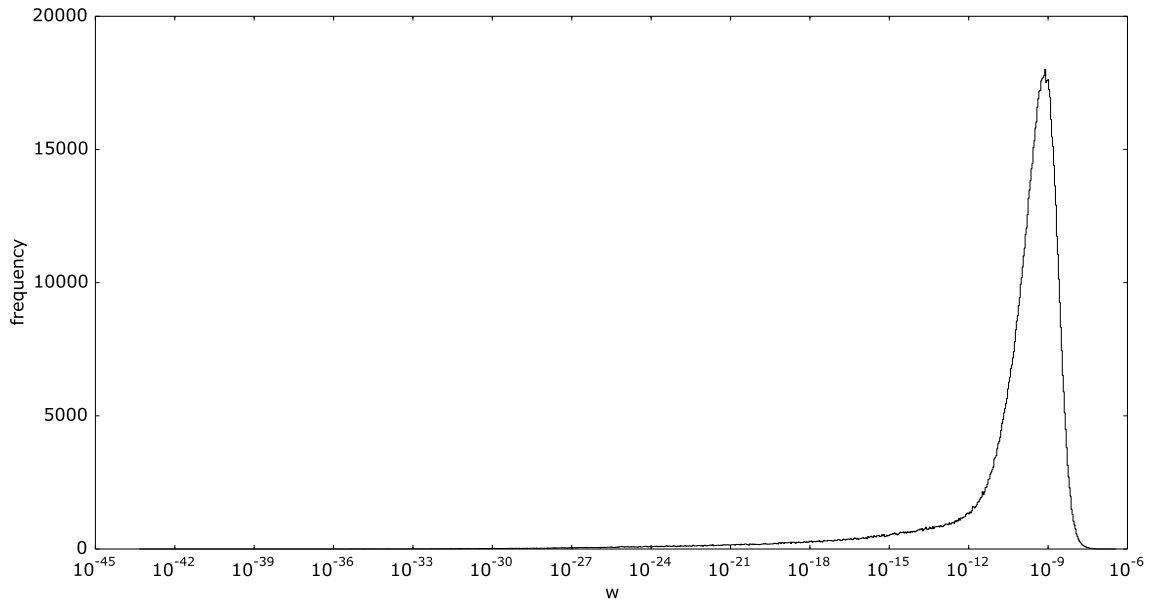


Figure 4.3: Histogram of the weights w , generated by 1 million sample points. Process: $gg \rightarrow d\bar{d}e^+e^-gg$, the expected unweighting efficiency is predicted by SHERPA to 0.067 %.

4.2.3 Error Definition

Most optimization problems involve a minimization of a target function. This holds also in machine learning, where the best algorithm / the best parameter configuration is found by minimizing a predefined error function f . A common used error function is e.g. the ‘mean squared error’, defined by the sum over the squared difference of all pairs of prediction (g) and true values(w):

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^N (w_i - g_i)^2. \quad (4.2)$$

However, there are sometimes good reasons to choose a different error function. The error function f used in this thesis correlates with the problem of data normalization. As shown in Figure 4.3 the distribution of weights covers over 30 orders of magnitude, which caused some problems with the decision tree implementation of scikit-learn. Even after scaling the data to a unit mean, there were still some outliers in the prediction which differed more then 20 orders of magnitude – probably caused by some numerical problems. This problem could be solved by using the logarithm, it transforms the weights in a range, where the algorithm does not behave unexpectedly.

However, using the logarithm has further effects. Many algorithms like e.g. decision trees or neural networks use the mean squared error as defined in (4.2) to improve their performance and learn through the training data. This is important: it makes a difference if minimizing $\text{MSE}(\log(w), \log(g))$ or $\text{MSE}(w, g)$. The first one is more suitable, because the new weight x_i yields to $\frac{w_i}{g_i}$. Truth prediction pairs of e.g. $(10^{-5}, 10^{-6})$ and $(10^{-15}, 10^{-16})$ will result both in the same x_i . However, the first pair would cause a much higher error in the non-logarithmic error definition, but the same in the logarithmic one.

Motivated by that, also the error function f , used for comparison of different algorithms or configurations, was defined by means of the logarithm

$$f = \exp \left(\hat{s} \left(\log \frac{w}{g} \right) \right). \quad (4.3)$$

Here, w is the true weight and g the prediction, $\hat{s}$ is the estimator of the standard deviation. This definition is guaranteed to be minimal, when $\text{MSE}(\log(w), \log(g))$ is minimal, too. Graphically, it minimizes the variance of the new weight x_i , an example is shown in Figure 4.4.

In order to reduce the variance of f , the whole training file is divided k times randomly in a training part, used by the algorithm for learning and a validation part, used to measure their performance by calculating f_k . f is then given by the average of all f_k . Such procedures are commonly used and often summarized as ‘cross validation’ [25].

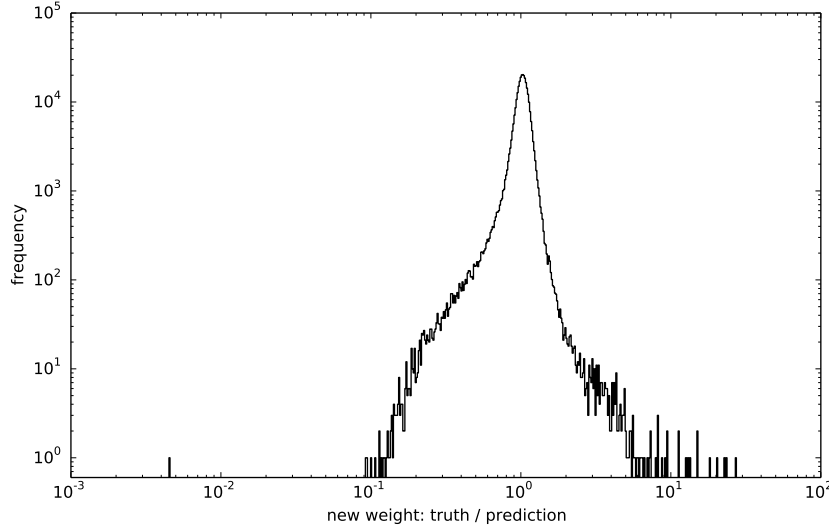


Figure 4.4: Example of a x -distribution for the process $dd \rightarrow eeg$, created with 700k training and 300k validation points. The prediction was done by an unoptimized random forest consisting of 50 trees. Here, f is equal to 1.18.

4.2.4 First Attempts with Scikit-Learn

In order to get a feeling for what kind of algorithms are suitable for regression problems like this, in a first step the performance of the most popular algorithms – implemented in scikit-learn – is compared for the process $gg \rightarrow d\bar{d}e^+e^-gg$. All algorithms were used with their default parameters⁶. One million points were generated with SHERPA to serve as training data, they were 20 times randomly divided in a training set of 700k points and a validation set of 300k points⁷. The random numbers, as described in Section 4.2.2, were used as features. The following algorithms were used:

- decision tree (‘DT’),
- random forest with 40 trees (‘RF(40)’),
- boosted decision tree with 40 trees (‘BDT(40)’),
- k -nearest neighbors (‘kNN’),
- support vector regression (‘SVR’).

The timings and results are shown in Figure 4.5, respectively in Table 4.1. The support vector regression is not included, it was aborted after two days run-time and exceeding the available memory.

⁶All hyper-parameters were adjusted by the developers to be as universal as possible. Details can be found in the online manual[41].

⁷The number of events used for training is inspired by the number of events which are used for the integration phase and is of the order of some 100k events, too. The size of the validation set is often chosen to be roughly one third of the available data and was therefore set to 300k sample points.

Considering all these tested algorithms, random forest and boosted decision trees performed best. Their error is nearly the same, but random forests have the great advantage to work in parallel during training. In this test they were executed with 20 tasks on 20 cores, making them more than 10 times faster than boosted decision trees.

Of course only a few algorithms were considered in this first comparison. In order to check whether other algorithms can perform better, AutoWeka was used. As described in Section 4.2.1, AutoWeka is able to optimize the search of suited algorithms and parameters. However, one still has to limit the overall time and the maximal training time per algorithm. The former was set to 70 hours and the other to one hour. After running AutoWeka with these parameters, using the same data as scikit-learn, it presented as best algorithm also an ensemble of trees⁸, having an error of 3.77. This is slightly better than the errors obtained by the unoptimized scikit-learn ensembles, the error obtained by a random forest is for comparison 3.80. However, ensembles of trees seem to be well suited for the prediction of weights. Furthermore, random forests have the advantage of working in parallel and will therefore be discussed in more detail in the next sections.

Possible Time Improvement

Before studying random forests in more detail, the timings of the algorithms above shall be used for a short comparison of the time necessary for an exact calculation of w , in contrast to the prediction time of the random forest.

On the same machine, SHERPA needed for the generation of 1M weighted events about 14 hours (one core), whereas the random forest was able to predict the weight of 300,000 events in one second by using 20 cores:

$$t_{\text{SHERPA}} = \frac{14\text{h}}{10^6 \text{ events}} = 0.05 \frac{\text{s}}{\text{weight}}, \quad t_{\text{RF}} = \frac{1\text{s}}{4 \cdot 10^5 \text{ events}} = 6.7 \cdot 10^{-5} \frac{\text{s}}{\text{weight}}. \quad (4.4)$$

This makes the prediction of one weight approximately 750 times faster than the exact calculation. If the unweighting efficiency is $\epsilon = 0.001$, 1000 events have to be calculated on average, before one weightless event remains. Following this, the new approach would still be more than 400 times faster than the classic one:

⁸meta-algorithm: ‘AdditiveRegression’, decision tree: ‘REPTree’

algorithm	error f	training time [s], 700k events	prediction time [s], 300k events
DT	6.9	72.4	0.23
BDT(40)	3.81	2248	9.8
RF(40)	3.8	175.6	1.4
kNN	631.6	2.1	5024

Table 4.1: comparison of some of the most popular algorithms implemented in scikit-learn

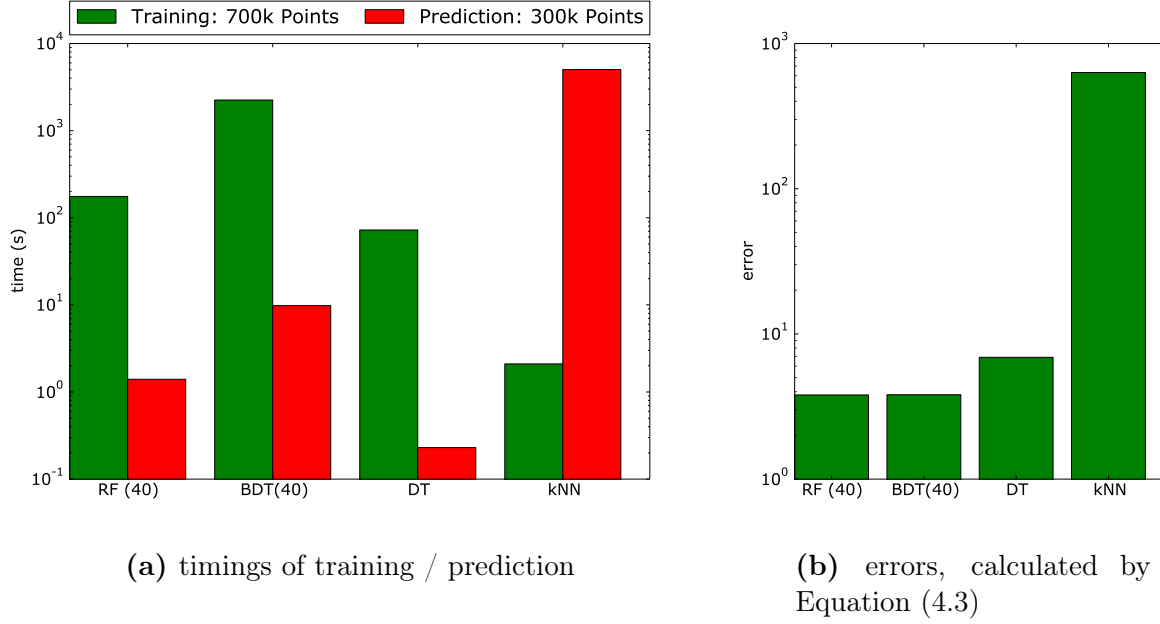


Figure 4.5: comparison of some of the most popular algorithms implemented in scikit-learn

$$\frac{t_{\text{classic unweighting}}}{t_{\text{machine learning}}} = \frac{\frac{1}{\epsilon} \cdot t_{\text{SHERPA}}}{\left(\frac{1}{\epsilon} - 1\right) \cdot t_{\text{RF}} + t_{\text{SHERPA}}} = \frac{\frac{1}{0.001} \cdot 0.05\text{s}}{\left(\frac{1}{0.001} - 1\right) \cdot 6.7 \cdot 10^{-5}\text{s} + 0.05\text{s}} \approx 427. \quad (4.5)$$

4.3 Optimization in example of Random Forest

As discussed in Section 4.2.4, the random forest algorithm is fast and has the best performance of all tested methods. Therefore some aspects of weight regression, e.g. the selection of features, will be discussed by example of a random forest. This section is divided into two subsections. In the first part some general aspects and methods of optimizing the prediction will be discussed, the second deals with the hyper-parameter optimization of a random forest.

4.3.1 Choice of Methods

Apart of the already mentioned advantages of random forests, the scikit-learn implementation offers one more useful feature. The algorithm is able to print the relative importance of all used features, normalized to unity. These importances are visualized for the example of Section 4.2.4 in Figure 4.6. Here, x_1 and x_2 correspond to the momenta fractions of the incoming partons and x_5 to x_{18} describe the final-state particles. The two additional features – x_3 and x_4 – have a special role. The first of these random numbers determines which initial-state channel is selected, the second determines the final-state channel. Figure 4.6 shows that the initial features have a big importance, whereas all other features do almost not contribute. However, there is no physical reason why the initial state should have a higher importance

than the final state. A possible explanation is the big number of final channels. This process uses 21 initial, but more than 700 final channels. These channels determine the meaning of each of the other random numbers by defining their transformations. If the algorithm is not able to handle a large number of channels, it will also be unable to interpret the random numbers.

In order to improve this situation several possibilities were tested. These methods are described in the following subsections and compared afterwards.

Choice of Features

The first study in Section 4.2.4 was done by using SHERPA's random numbers as features. These numbers are optimized to make the function of weights w as flat as possible and were therefore expected to be good features⁹.

However, the training file also contains all particles momenta. They can be used as features as well and are therefore the second tested method.

The third method is a new coordinate system, which is optimized for hadron colliders and often used in particle physics. The initial state is here described by the center-of-mass energy $\sqrt{s}$ and the rapidity η_{in} . In the final state, each particle can be described by its energy, transversal momentum (p_T) and the two angles η and ϕ . Furthermore, the sum of all leptons transversal momenta and the sum of all final-state particles momenta were used. These variables can be obtained by the particles momenta, the transformations are explained in Table 4.2. Finally, the whole system was rotated to give the first lepton ϕ equal to zero.

Independent from these features, there are also some general methods of feature selection / extraction. A popular approach is the principal component analysis (PCA). This method transforms all features linearly into a new space, thereby minimizing the correlation between

⁹To visualize this, the distribution of weights, projected on each feature, is plotted in Appendix B.1, for both, the momenta (Figure B.2) and random numbers (Figure B.1).

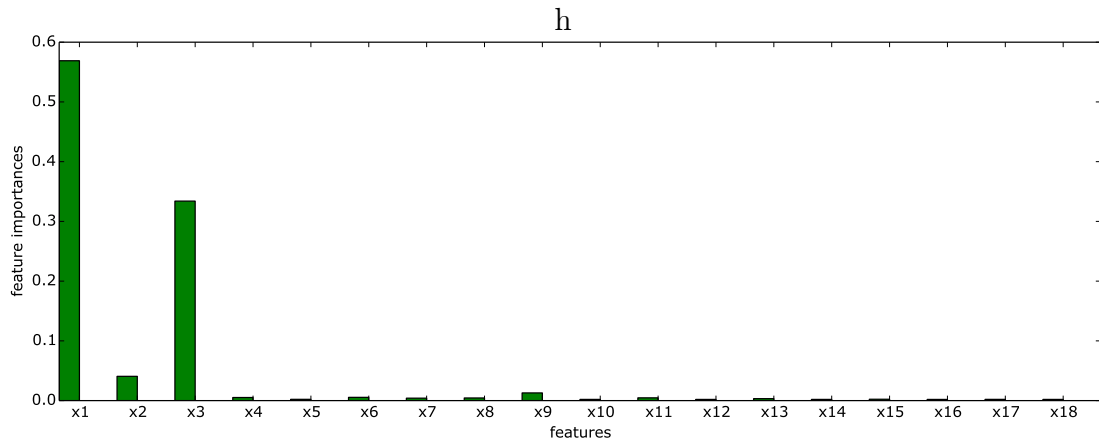


Figure 4.6: feature importances of an unoptimized random forest, trained with SHERPA's random numbers

all features. More details are given in e.g. [46]. PCA is often applied for reducing the degrees of freedom, but can also be used to transform the features in a more suited form.

Separation of Channels

As shown in Figure 4.6, the large number of channels causes some problems when using the random numbers for prediction. By using momenta or transformed momenta this problem can be avoided, because their meaning is independent from the respective channel¹⁰. Nevertheless, it can be interesting to compare the prediction of one predefined initial / final channel with the general prediction. If e.g. the random numbers allows a very precise prediction when considering only one channel, one can think of finding a way to help the algorithm distinguishing between them.

In this process events are generated most often in the initial channel with number 18 and the final channel with number 21¹¹. Therefore, a new training file, containing only events with these channels, was created.

Direct vs Indirect Prediction

However, not only the features, but also the target may be optimized. As mentioned in Section 3.1.3, the weights can be written as product of a phase space weight (PSW) and the pure matrix element (ME). The calculation of ME is much harder than calculating PSW. Using this, one can try to predict only ME and multiply it with the exact phase space weight. Of course it is also possible to predict both of them independently and multiply afterwards.

Summary

All these methods above were studied in combination, the results are shown in Table 4.3. The best choice of features are the transformed momenta. When using them for the prediction of

¹⁰The channels are used in Sherpa to optimize the integration as explained in Section 3.1.1, the channel number defines which transformation has to be used in order to calculate momenta from random numbers. When using the momenta directly, the channel numbers are not expected to have big importance.

¹¹The frequency of these channels is shown in Appendix B.1, Figure B.3 and Figure B.4.

initial	$\sqrt{s} = \sqrt{(E_1 + E_2)^2 - (\vec{p}_1 \cdot \vec{p}_2)^2}$ $\eta_{in} = 0.5 \cdot \log \frac{x_1}{x_2}$
final, for each particle	$p_T = \sqrt{p_x^2 + p_y^2}$ $\phi = \arctan \frac{p_y}{p_x}$ $\eta = -\log(\tan(\frac{\theta}{2})), \quad \theta = \arctan \frac{p_T}{p_z}$
final, additional	$\frac{\sum_{\text{leptons}} p_T}{\sum_{\text{final-state particles}} p_T}$

Table 4.2: transformations to the new coordinate system

the whole weight, the lowest error was achieved. Predicting only ME or both, ME and PSW, does not offer advantages, these methods perform worse than the direct way.

However, even when using the (transformed) momenta, predicting only one channel performs slightly better than using all. In the direct prediction of w , the difference is around 3%. This makes it not worthwhile to train the channels separately, particularly since maybe the selected channel is easy to predict by chance. Of course training the channels separately would increase the computational effort enormously, too.

The principal component analysis can also not conduce to improvement. It is not shown in the table, but was tried in the direct way by usage of the momenta and gave an error of 3.44, which is worse than using poor or transformed momenta as features. This does not surprise, the momenta are already supposed to be uncorrelated.

In summary, using the transformed momenta and predicting the weights independently from the channels in a direct way seems to be the best method to minimize f .

4.3.2 Hyper-Parameter Optimization

In the previous section several methods of improving the prediction were tried and discussed. However, all of them were done with the default hyper-parameters and 700k events for training. In order to check how much the result can be improved further, the optimal number of training events as well as the influence of the hyper-parameters were studied. There are many different methods available to optimize the hyper-parameters in an automated way. Popular methods are grid search and random search. Grid search uses a grid of all possible combinations of different hyper-parameters and searches it for the minimal error. This is very inefficient when considering many different parameters. By contrast, random search evaluates the algorithm a predefined count with random parameters and keeps the best result. In case of having many different parameters, this method can perform better than grid search [47]. Further methods are sequence model based optimization algorithms as used by AutoWeka.

All these algorithms have in common that they are pure black box optimizations, using them

prediction of:	all channels			one channel (in: CH18, out: CH21)		
	random numbers	momenta	transf. momenta	random numbers	momenta	transf. momenta
w	3.78	3.32	3.17	3.45	3.35	3.06
ME · PSW	4.71	4.29	3.5	4.08	4.47	3.36
only ME	6.3	7.9	5.9	4.67	8.5	5.85

Table 4.3: Comparison of the errors for different methods. All results were achieved by a random forest with default parameters and 50 trees. In the first row the weight was predicted directly, in the second both ME and PSW were predicted independently and multiplied afterwards. Finally in the last row only ME was predicted and multiplied with the exact phase space weight. All errors' error are smaller than 0.01 and therefore not printed explicitly.

makes it difficult to learn something about the algorithm or the problem. For this reason the role of each parameter was studied manually. The influence of the following parameters was examined:

- number of events for training (n),
- number of trees (t),
- maximal number of used features for one tree ($f_{\max}$),
- maximal number of leaf nodes ($l_{\max}$),
- maximal depth of each tree ($d_{\max}$)¹²,
- minimal number of samples remaining in each leaf (s_{leaf}),
- minimal number of samples required for splitting a node (s_{node}).

When considering one parameter, all remaining parameters were left at their default value¹³. The error was always computed using 300k events, all values are averaged over three iterations. The dependencies of timings¹⁴ and errors are plotted for both validation and training in Figure 4.7 to 4.13.

There are two possible strategies for optimization. On the one hand, one can minimize the error, ignoring the computational effort. On the other hand, one can try to minimize the error but also maximize the performance. The blue vertical line in all error plots marks the chosen values for the optimal performance configuration. In Table 4.4, all determined parameters and final results are summarized for all methods and compared to the default values. The errors on the validation data do not much differ, the default parameters are already well suited. Comparing them with the error on the training data, the latter is for all methods distinctly smaller than the validation error. This can be a sign of ‘overfitting’, which means the algorithm begins to model statistical fluctuations in the training data and loses its ability to generalize. However, in the ‘performance optimized’ configuration they are already closer. Furthermore, it predicts more than twice as fast as the default algorithm and more than three times faster than the error minimizing configuration.

Of course, it is still possible to use more training data which does not affect the time required for prediction, but doing this causes at least with the scikit-learn implementation some trouble with memory, while a random forest with 700k training points uses around 3GB memory, 3 million events already require more than 20GB.

¹²When using $l_{\max}$, the parameter $d_{\max}$ will be ignored by the algorithm.

¹³ This method can be dangerous, because it ignores all possible correlations. However, the final errors are very similar and let not expect to miss something.

¹⁴There were sometimes some other processes running on the machines, therefore the timings are to be seen with caution.

In order to check whether there are other algorithms which can deal better with these transformed momenta, a new AutoWeka run was set up. Algorithms which are often said to be able extracting relevant features from the available input data are neural networks. These networks are often slower in training than e.g. random forests. For this reason the time limit for AutoWeka was increased to 120 min training time by 60 hours optimization time. Furthermore, all features were scaled into the interval $[0,1]$ ¹⁵ and the target was scaled into the interval $[0,1]$ ¹⁶.

Indeed, AutoWeka suggests the neural-network implementation of Weka, called ‘Multilayer-Perceptron’. The resulting error is 3.81, which is higher than expected and also worse than the random forest. It is not clear, why this is actually the case. A possible explanation would be that the time limits were still too low and the training of the final configuration could not be completed, the presented error was calculated after the optimization phase by the final model which was saved. The suggested network has furthermore only 27 neurons in only one layer which is quite small and maybe insufficient to model the data.

Nevertheless, the results motivate a closer look at neural networks. Doing this with (Auto)Weka is difficult, because Weka cannot transform the data logarithmic, for the AutoWeka experiments they were prepared and evaluated with Python. To avoid these circumstances, the ‘FANN’ library was used for all further studies concerning neural networks.

¹⁵ Feature scaling is not necessary when using algorithm based on decision trees, but neural networks as well as the k NN -algorithm require all data to have the same range.

¹⁶ Neural networks can, depending on the chosen activation function, require the target to be scaled. This is explained in more detail in Section 4.4.3.

		default	performance optimized	error minimized
parameter	t	40	40	100
	$f_{\max}$	all	25	25
	$l_{\max}$	unlimited	20000	unlimited
	$d_{\max}$	unlimited	unlimited	unlimited
	s_{leaf}	1	4	4
	s_{node}	2	13	13
results	t_{train} [s]	2929	2306	6015
	f_{train}	1.56	2.36	2.01
	t_{valid} [s]	10.00	4.44	14.11
	f_{valid}	3.17	3.19	3.14

Table 4.4: comparison of different configurations of the random forest

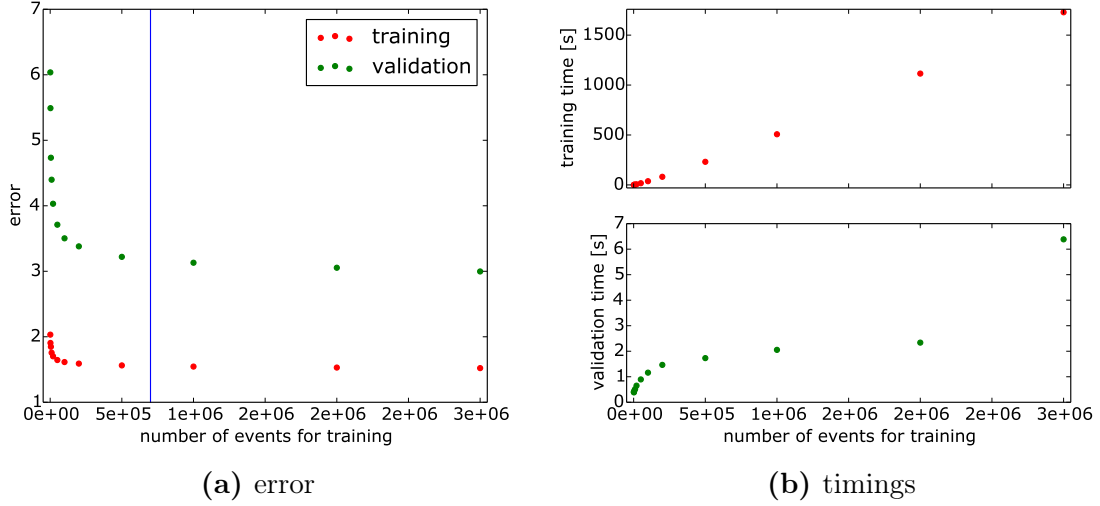


Figure 4.7: random forest: error and timings as function of the number of samples for training

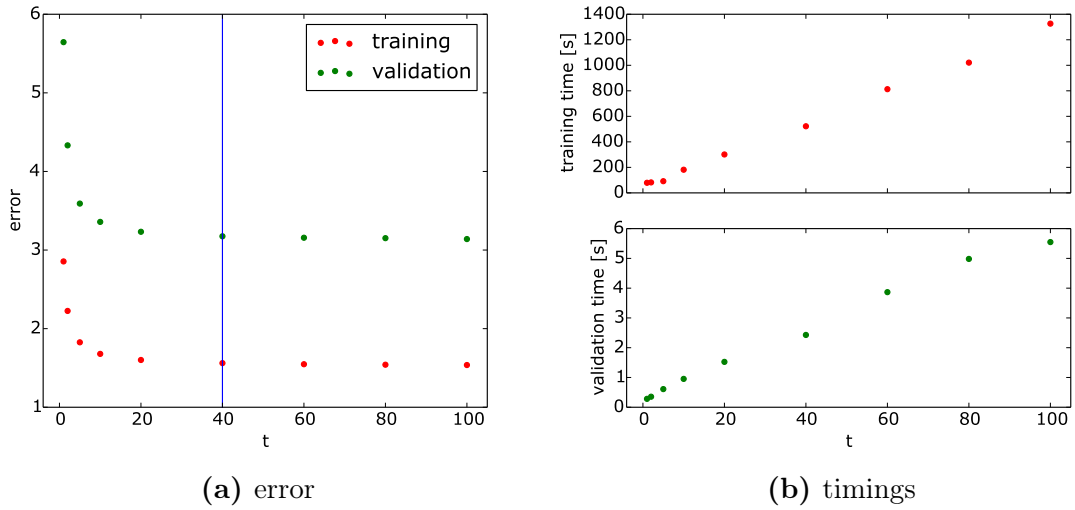


Figure 4.8: random forest: error and timings as function of the number of trees

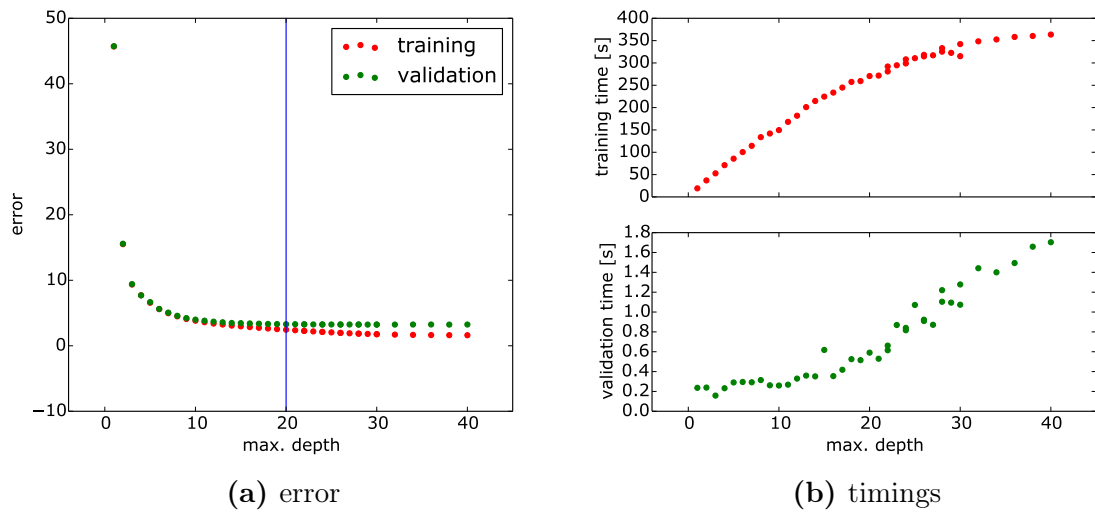


Figure 4.9: random forest: error and timings as function of the maximal depth of each tree

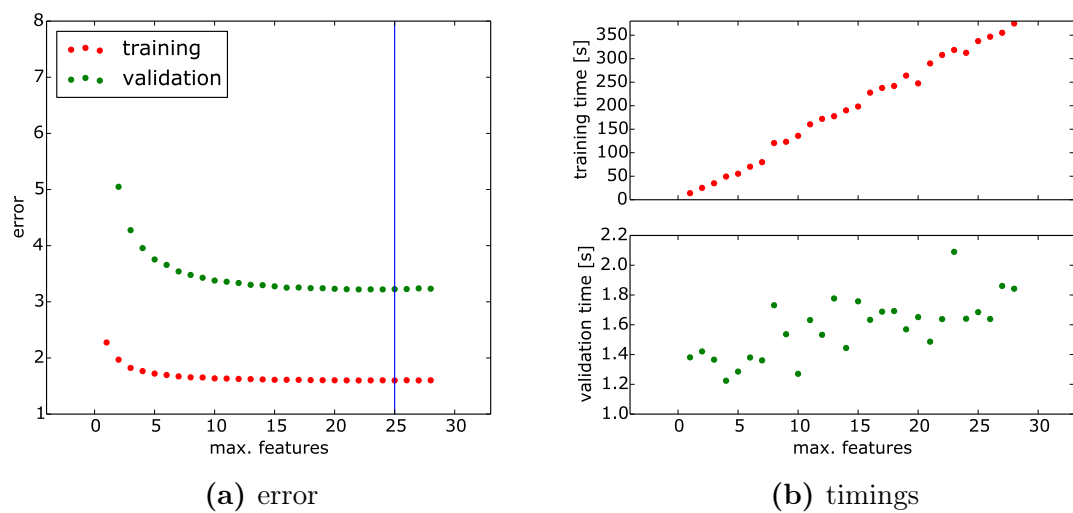


Figure 4.10: random forest: error and timings as function of the maximal number of used features

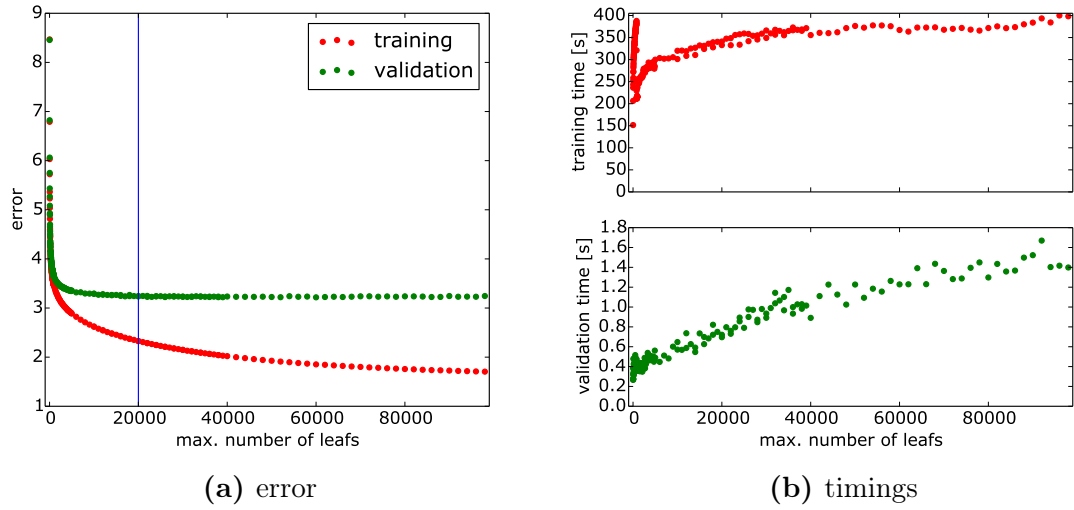


Figure 4.11: random forest: error and timings as function of the maximal number of leaves

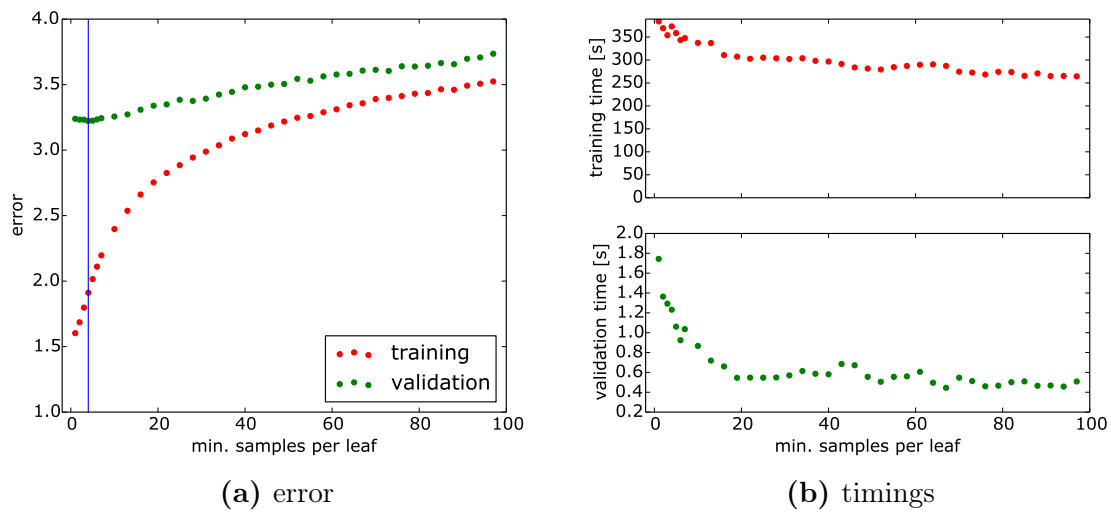


Figure 4.12: random forest: error and timings as function of the minimal number of samples per leaf

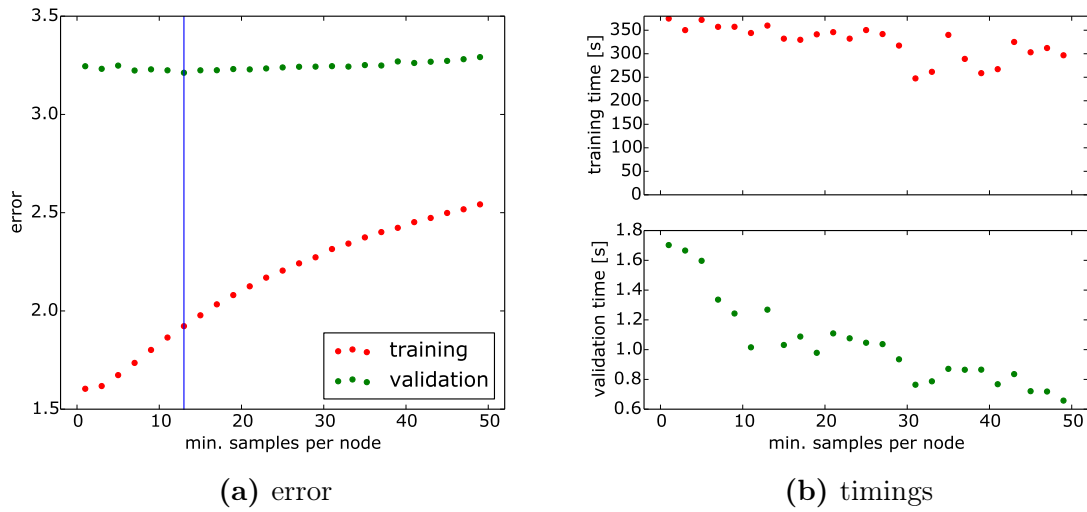


Figure 4.13: random forest: error and timings as function of the minimal number of samples per splitting / node

4.4 Optimization of Neural Networks

As motivated by the previous section, neural networks will be studied in more detail, too. They have many parameters to optimize, in order to find the best configuration the parameters provided by AutoWeka were used to begin with. They will be presented in the first subsection. The following parts discuss how these parameters can be adjusted to improve the prediction.

4.4.1 Configuration by AutoWeka

As mentioned in Section 4.3.2, AutoWeka suggested the ‘MultilayerPerceptron’ of Weka as the best method. This method supports only the BFGS-algorithm (details e.g. in [46]), a sigmoid activation function for the hidden layer and a linear activation function for the output layer. AutoWeka also prints the hyper-parameters found by the internal optimization. These are a learning rate of 0.14 and a momentum of 0.7, furthermore it suggests one hidden layer with 27 neurons. This configuration is summarized in Table 4.5

This network was now roughly reproduced by the FANN-library as the beginning of all further optimizations.

algorithm:	BFGS
activation function:	sigmoid, linear
learning rate:	0.14
momentum:	0.7
hidden layer:	1
neurons per layer:	27

Table 4.5: network configuration as obtained by AutoWeka

4.4.2 Choice of Algorithms

Although FANN offers several different optimization algorithms, the BFGS-algorithm of Weka is not supported. However, in FANN there are five other algorithms available. An overview of the detailed differences and specific advantages / disadvantages of four of them is given in the online manual of FANN [48], the missing ‘SARPROP’-algorithm is presented in [49]. None of these algorithms works always perfectly, their actual performance depends on the specific problem. The default algorithm is RPROP. In order to study which algorithm is suitable for the regression of weights, all settings were adopted from AutoWeka but only the algorithm is changed.

The results are shown in Table 4.6. Batch, QuickPROP and SARPROP seem to converge very slowly, whereas the RPROP algorithm performs already better. The best result is achieved by the incremental algorithm, where the error is smaller than the one obtained by AutoWeka. For this reason the incremental algorithm was chosen for the further studies.

4.4.3 Activation Function and Target Normalization

When working with neural networks, activation functions – sometimes also called transfer functions – have a big importance. They transform the weighted sum of all inputs to one output and also affect the backpropagation of errors. Backpropagation uses gradient methods to minimize the final error as function of all weights, the derivation of the activation function contributes due to the chain rule of differentiation. For this reason, the curvature of the chosen function is important for the actual performance but also the stability of neural networks. An often used activation function is the sigmoid function, but also other functions such as the Gaussian or Elliot function can give good results. They are drawn in Figure 4.14, mathematical descriptions of the functions implemented in FANN are given in [50].

In Weka’s ‘MultilayerPerceptron’ the activation function of all hidden neurons is a sigmoid function, only the output neuron has a linear activation function. This linear function enables the usage of unscaled target data, but limits the training performance due to its constant derivation.

The influence of several activation functions was studied in combination with different target scalings. In FANN all activations functions, except the linear function, are bounded to the

algorithm	error
RPROP	9.8
incremental	3.4
batch	538
QuickPROP	534
SARPROP	525

Table 4.6: errors obtained by FANN with the parameters suggested by AutoWeka

interval $[0,1]$ or $[-1,1]$. The latter class of functions are the same as the one of the first, but stretched to the larger interval. In this thesis four activation functions were compared. They are the sigmoid function, the Gaussian function, Elliot's function and the linear function. All were used as well for the hidden but also for the output layer. Following this, the target was scaled to the interval $[0,1]$ or $[-1,1]$, respectively. The training was done with 200 epochs¹⁷, the results are summarized in Table 4.7. Learning rate, momentum and number of neurons were set as suggested by AutoWeka.

The best result was achieved by scaling the target into the interval $[0,1]$ and using the sigmoid function for activation. As expected, the linear function performed worst.

4.4.4 Feature Normalization

Up to now, all features were always scaled to the interval $[0,1]$. Further common used approaches are the interval $[-1,1]$ or transformations to zero mean and unit variance. All these transformations can easily be done using scikit-learn's 'preprocessing'-module. As before, the remaining parameters were set as suggested by AutoWeka. The results are given in Table 4.8. They are very close, therefore the average of five iteration is stated. Nevertheless, the results do not differ significantly. For all further studies the transformation method with zero mean and unit variance was chosen.

4.4.5 Learning Rate and Momentum

The learning rate and momentum belong to the most important parameters of the backpropagation algorithm. Both parameters define the behavior of the algorithm when looking for minima of the error function. The learning rate defines the step size. If it is too small, the convergence is very slow, but when the learning rate gets too high the algorithm may jump over the minima. This dilemma is attempted to be solved by adding a momentum term. If the function seems to be flat, the momentum increases the learning rate, otherwise it decreases[36]. Both learning rate and momentum are in the interval $[0,1]$. Usually, the momentum is higher than the learning rate.

Their influence on the performance of the incremental backpropagation algorithm was studied

¹⁷Epochs means the number of iterations used for the backpropagation.

activation function	target interval $[0,1]$	target interval $[-1,1]$
Elliot	3.41	3.41
sigmoid	3.13	4.00
gaussian	3.69	3.25
linear	21.6	10.8

Table 4.7: errors obtained by FANN using different activation functions

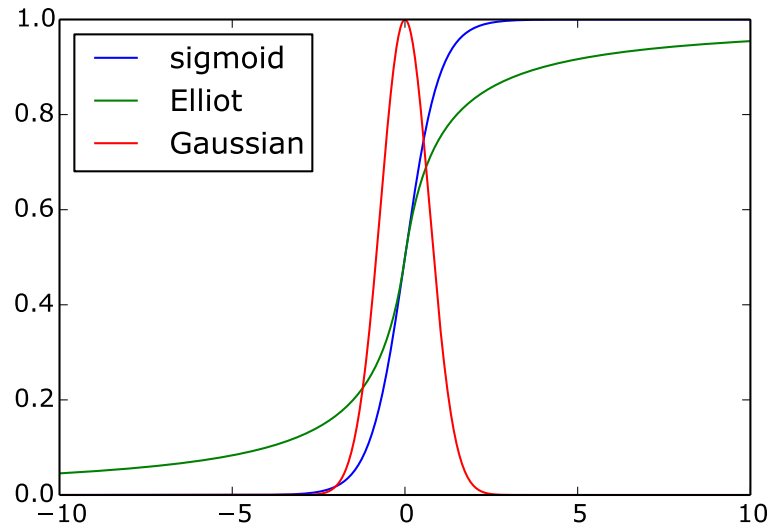


Figure 4.14: comparison of different activation functions implemented in FANN

scaling method	error
interval [0,1]	3.10 ± 0.01
interval [-1,1]	3.08 ± 0.01
mean=0, variance=1	3.08 ± 0.02

Table 4.8: Comparisons of errors, obtained by different methods of feature scaling. The errors are statistically, obtained by a new trained network with different training / validation data.

by a grid of possible combinations. All further parameters were set as determined in the previous sections, 5 iterations were performed and the average is taken. Learning rate and momentum were varied between 0.1 and 0.9, the results are shown in Figure 4.15. The best result was achieved by a learning rate of 0.1 and a momentum of 0.3, the resulting error is 3.01. When looking at Figure 4.15 one can think of setting the learning rate even lower than 0.1, this was tested but increased the error.

Several parameters were changed with respect to the last comparison of algorithms in Section 4.4.2. For this reason, the RPROP algorithm was tested again with the following configuration:

- feature scaling: zero mean, unit variance,
- target scaling: interval [0,1],
- activation function: sigmoid,
- structure: one hidden layer, 27 neurons.

The RPROP algorithm does not have a learning rate or momentum, but some other parameters which were not modified. It gives an error of 4.03, this is better than before but still significantly

worse than the incremental backpropagation. Of course there are many more configurations of algorithms and corresponding parameters possible, but testing all of them is not feasible. However, the incremental backpropagation algorithm, configured with a learning rate of 0.1 and momentum of 0.3, performs with one hidden layer and 27 neurons already better than a random forest of 100 trees.

For this reason the next section deals with the optimal number of events for training, afterwards the structure of the network will be optimized.

4.4.6 Optimal Number of Training Points

In the previous section 700 thousand events were used for training, motivated by the results achieved with the random forest. However, there is no guarantee this holds for neural networks, too. In order to study whether this number should be changed, a new experiment was started. The number of training events was varied between 1000 and 3 millions. Figure 4.16 shows the dependencies of the validation error and the training time of this number, the validation time is not influenced. The error decreases rapidly when increasing the number of training events to some 100k samples, but also the training time grows linearly. 700k training events seem to be a good compromise and are therefore used in what follows.

4.4.7 The Network's Structure

Several parameters were already adjusted in order to improve the network's performance. The best tested configuration uses the 'incremental backpropagation' algorithm and sigmoid activation functions. The target is scaled to $[0,1]$ and the features to zero mean and unit variance, furthermore the learning rate was set to 0.1 and the momentum to 0.3. Using these settings, the neural network performs already better than an optimized random forest, consisting of 100 trees.

However, up to now only small networks with 27 neurons were studied. More complex networks are expected to predict better and will therefore be studied in this section.

One Hidden Layer

There are two principal possibilities to extend the actual network. On the one hand, one can add additional neurons to a single layer, but one can also add further layers and make the network deeper. This section deals with the optimal number of neurons when using one layer only. All hyper-parameters were set as described in Section 4.4.7, the number of neurons was varied between 2 and 500. More complex networks are often slow to train, therefore the number of epochs was increased from 200 to 1000.

The errors and timings are shown in Figure 4.17. Both the validation and training time increase linearly with a increasing number of neurons. The validation error decreases rapidly

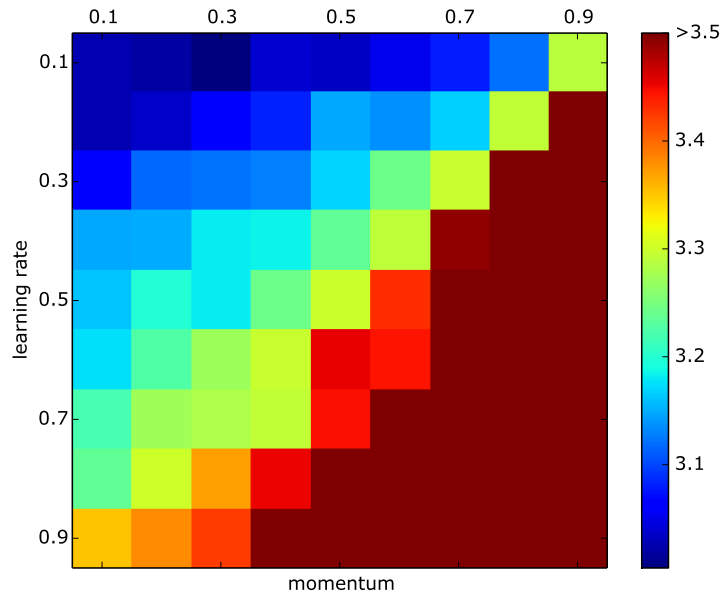


Figure 4.15: Validation error after 200 training epochs as function of the learning rate and the momentum for the incremental backpropagation algorithm.

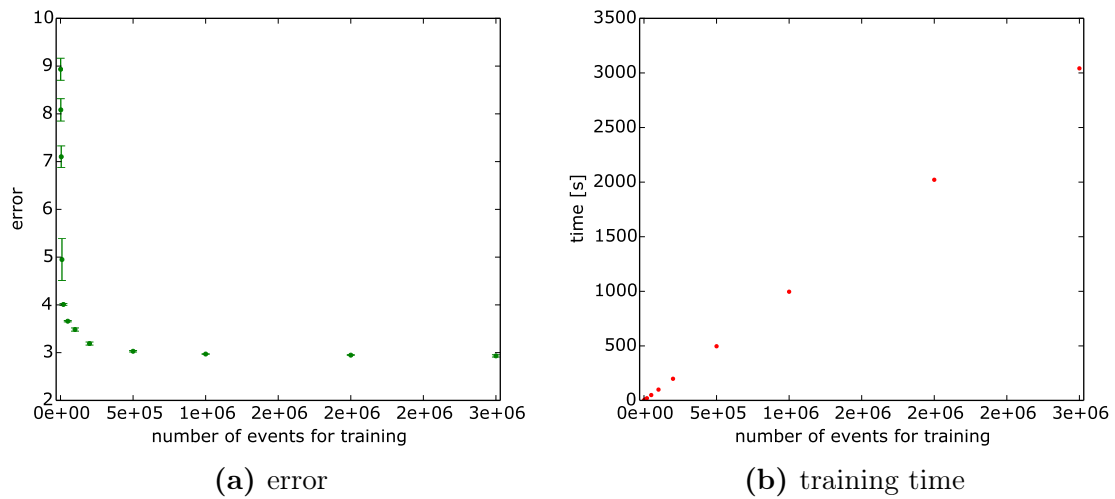


Figure 4.16: Error and training time for a network of 27 neurons, the errors are averaged over 4 iterations.

until the number of neurons reaches 100, using more neurons does not influence the error much. In order to exclude this stagnation to be caused by an insufficient number of epochs, the internal error used during training as function of the number of epochs is compared for different numbers of neurons. These plots are often called 'learning curves' and are shown in Figure 4.18. Indeed, more epochs could help improving the performance of a network with 500 neurons, but nevertheless the curves are very close. The possible gain does not justify the much higher training and validation times.

A further possibility which may make it obsolete to use so many neurons is the usage of far more than one layer. Such networks will be explored in the following sections.

Two Hidden Layers

In order to study neural networks with more than one hidden layer, a second one was added to begin with. The previous network performed best when using around 100 neurons, but this does not necessary hold when using more layers. For this reason, two types of two-hidden-layer networks were studied, one with 40 and the other with 100 neurons in the first layer. The size of the second layer was varied between 5 and 80. All further parameters were set as before, 1000 epochs were used for training.

In Figure 4.19 and Figure 4.20 the results, timings and learning curves are shown for a neural network, consisting of 40 neurons in the first and 5 - 80 neurons in the second hidden layer. The influence of the second layer is limited. Indeed the error is lower when comparing to a single-layer network of 40 neurons, but when considering a total number of 100 neurons, putting them into one layer performs better than 40 in the first and 60 in the second.

This situation changes when increasing the first layer to 100 neurons and then adding a second one. Using five neurons in the second layer gives already a smaller error than a network with 500 neurons in one, increasing the size of the second layer to 80 neurons reduces the error even from 2.74 (100 neurons in only one layer) to 2.58. The corresponding timings, errors and learning curves are shown in Figure 4.21 and Figure 4.22.

Deep Networks

Extending the simple one-layer network with a second layer of roughly the same size could significantly reduce the error for both, 40 and 100 neurons per layer. It is obvious to continue with deeper networks by adding further layers. Regarding the results obtained by two-layer networks one could think of constructing network as kind of pyramids, where the first hidden layer has many neurons and the last only a few. However, examining and comparing such networks in a systematic way is complicated. In order to study the possible error reduction with deeper networks they were studied only with 30 or 100 neurons in each layer. All hyper-parameters were set as before, but the number of epochs was increased to 3000.

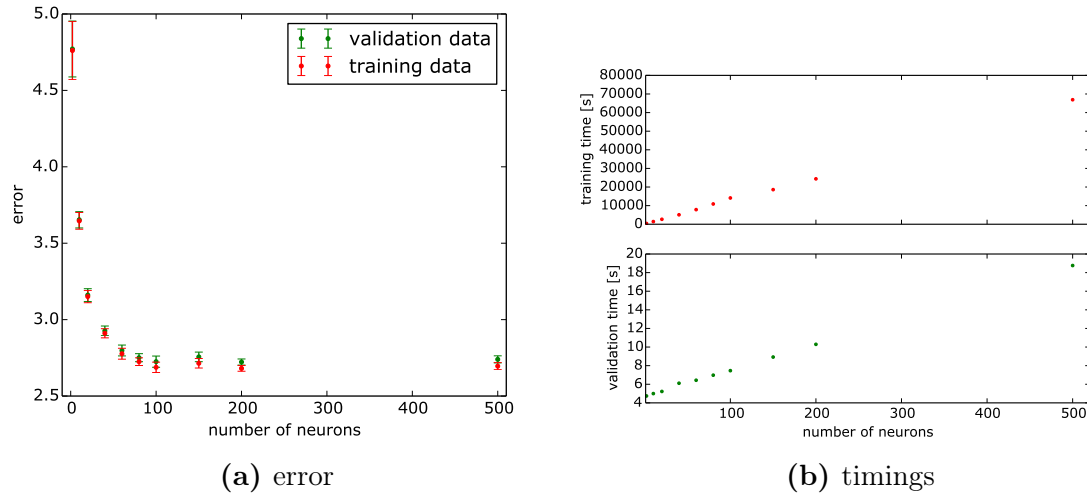


Figure 4.17: Errors and timings of neural networks with one hidden layer as function of the number of used neurons. All values are averaged over 4 iterations.

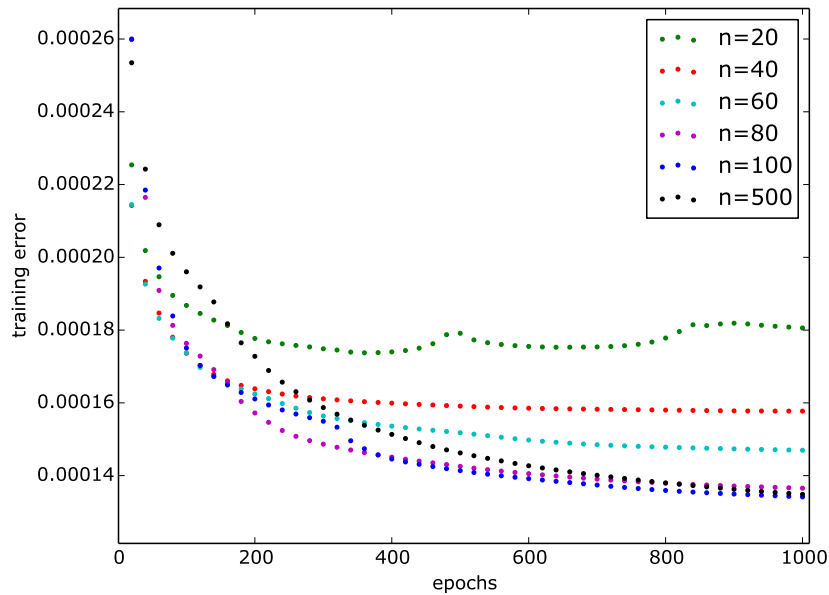


Figure 4.18: comparison of learning curves for different number of neurons with one hidden layer

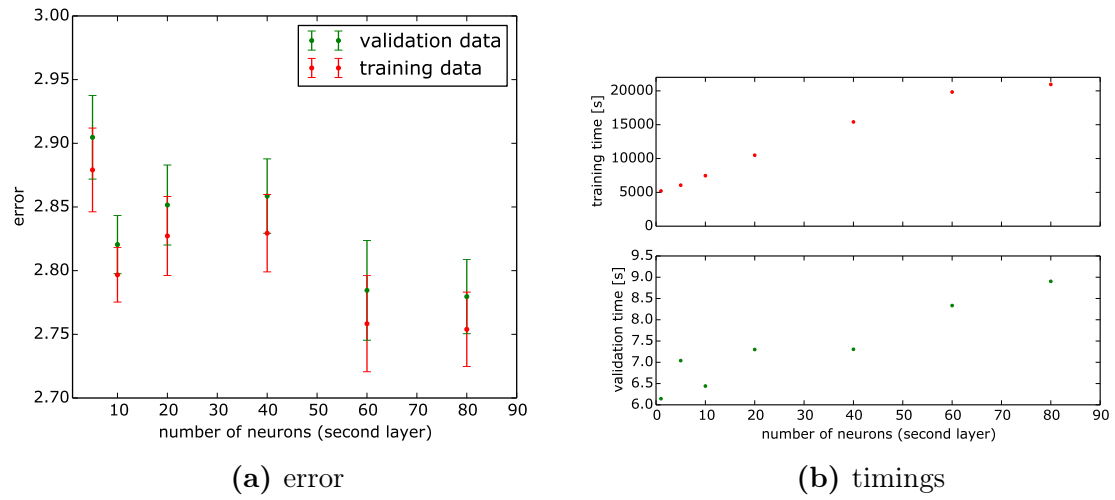


Figure 4.19: Errors and timings of neural networks with two hidden layers. The first layer has 40 neurons, error and timings are plotted as function of the second layer's size. All values are averaged over 4 iterations.

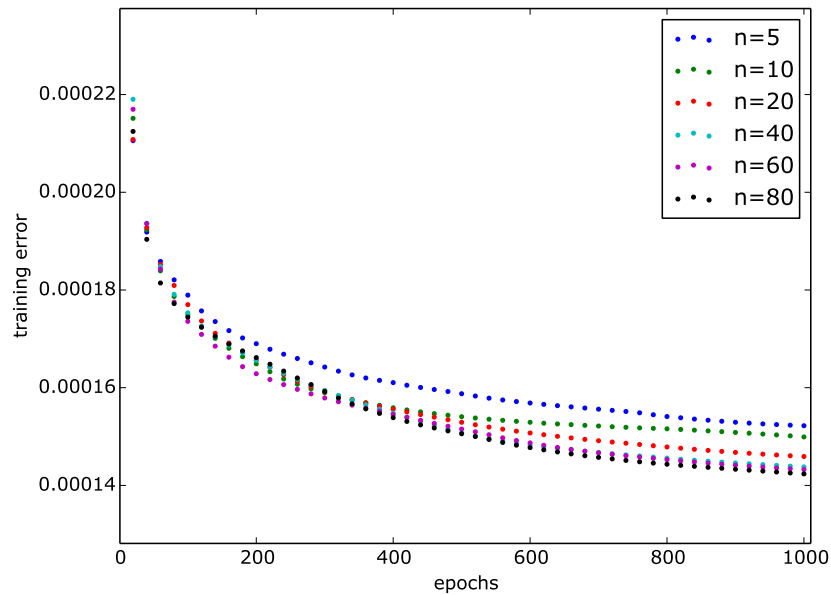


Figure 4.20: Comparison of learning curves for neural networks with two hidden layers. The first layer has 40 neurons, the size of the second one was varied.

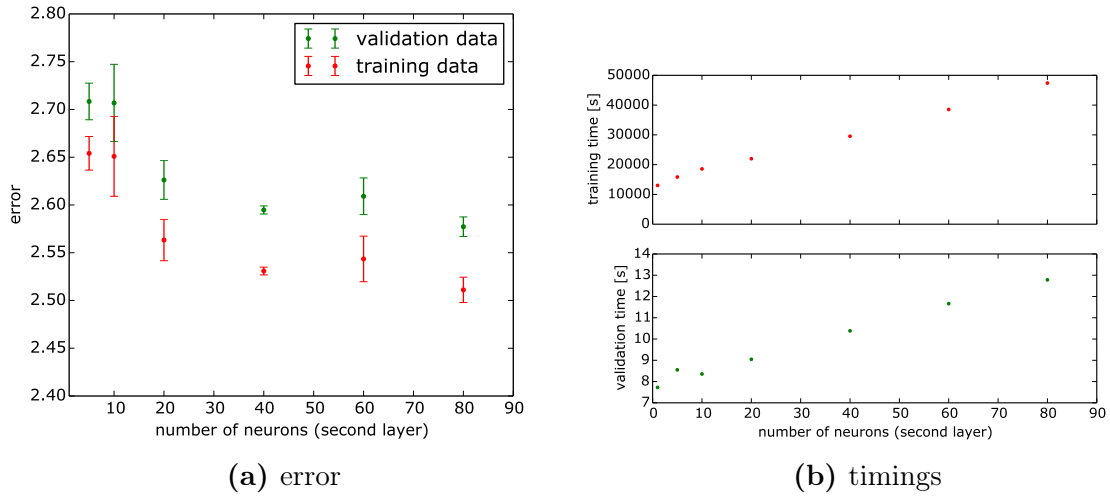


Figure 4.21: Errors and timings of neural networks with two hidden layers. The first layer has 100 neurons, error and timings are plotted as function of the second layer's size. All values are averaged over 4 iterations,

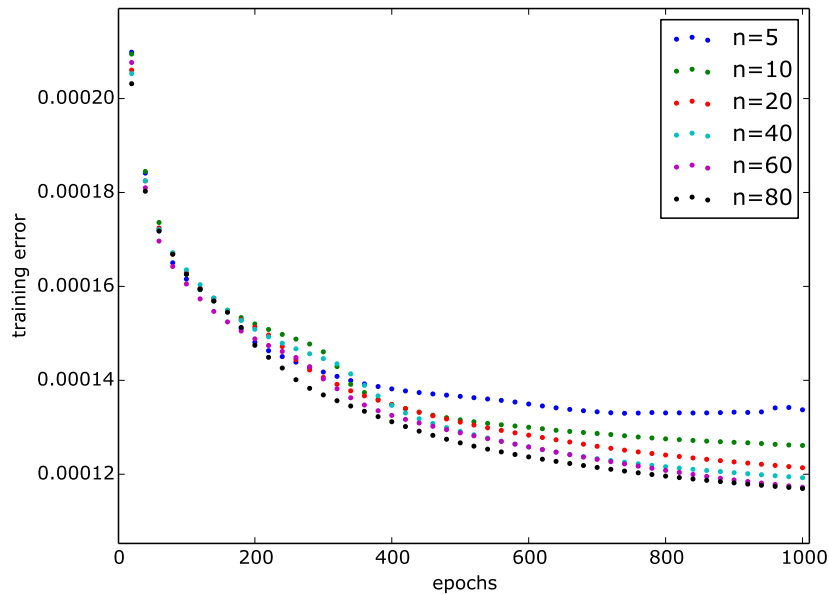


Figure 4.22: Comparison of learning curves for neural networks with two hidden layers. The first layer has 100 neurons, the size of the second one was varied.

In Figure 4.23 and Figure 4.24 the results for the first case of 30 neurons per layer are shown. A minimal error of 2.72 is obtained when using three layers, but the results are still not competitive with a network of 100 neurons in only one layer. This seems not to be an effect of an insufficient number of epochs, the learning curve for three layers looks already well.

When looking at networks with 100 neurons per layer, the three-layer network gives an validation error of 2.50. Deeper networks do not perform much better. The errors and timings are shown in Figure 4.25. The learning curves are drawn in Figure 4.26, it seems networks with more than three layers do not perform better even when trained with more epochs. Nevertheless, one can think of using more epochs to improve the prediction of the 3-layer network, but this would also increase the training time. Training this network with 3000 epochs took already 84 hours, the training time increases linearly with the number of epochs.

For this reason the optimization was stopped here, more complex networks are not assumed to give a significantly lower error but will increase both the training and prediction time.

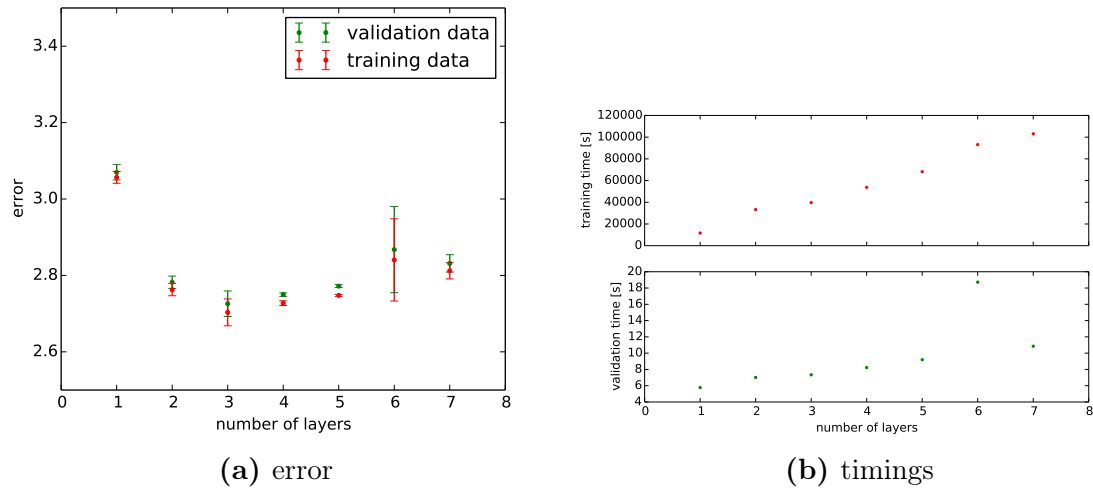


Figure 4.23: Errors and timings of neural networks with 30 neurons per layer. Error and timings are plotted as function of the number of layers. All values are averaged over 4 iterations.

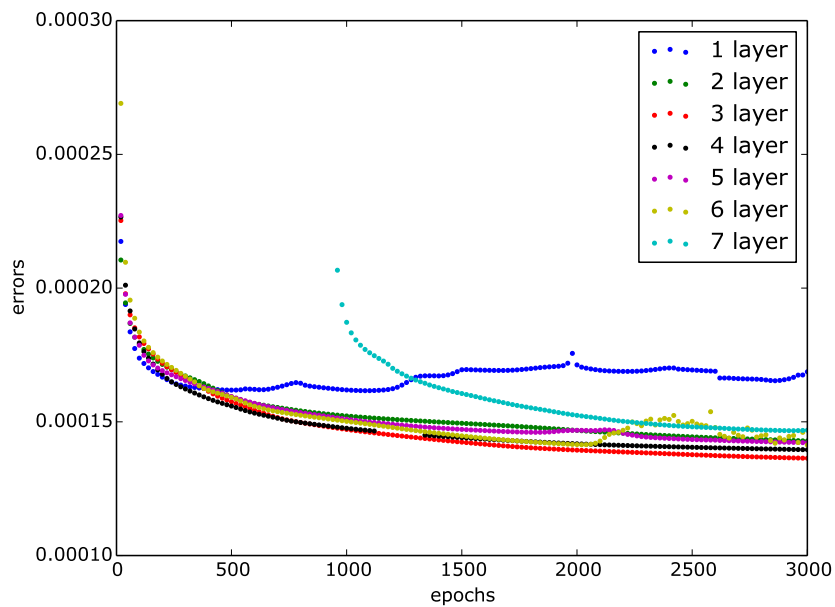


Figure 4.24: comparison of learning curves for neural networks with 30 neurons per layer

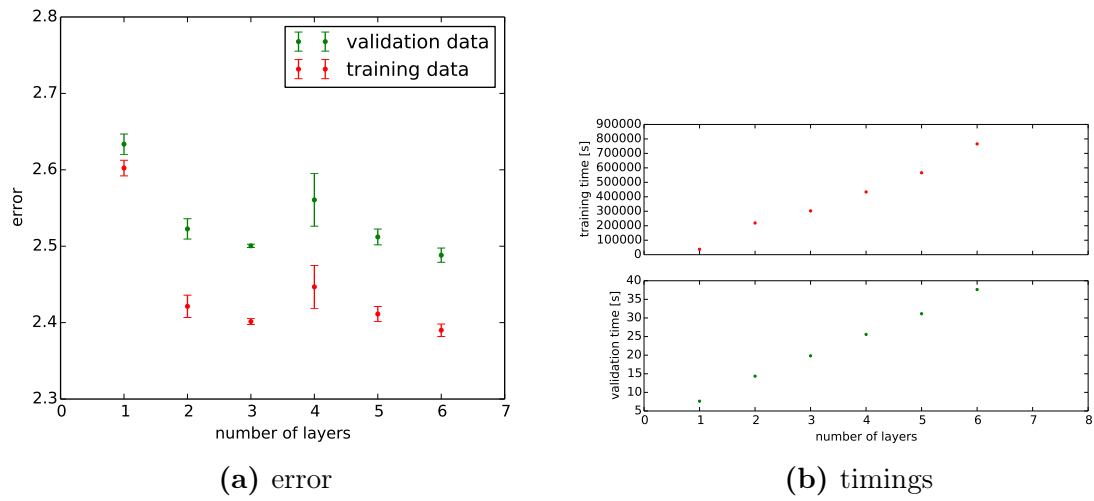


Figure 4.25: Errors and timings of neural networks with 100 neurons per layer. Error and timings are plotted as function of the number of layers. All values are averaged over 4 iterations.

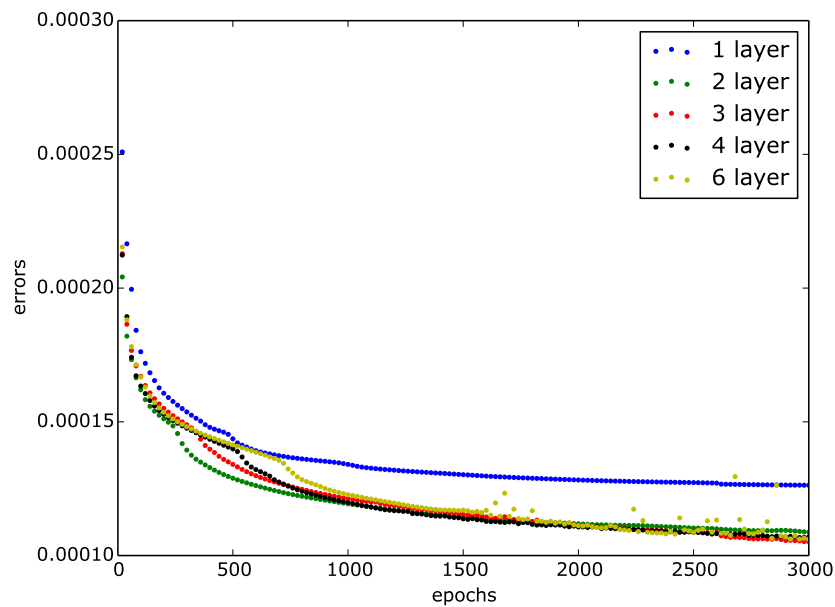


Figure 4.26: comparison of learning curves for neural networks with 100 neurons per layer

5 Evaluation

In Section 4.1 several possible applications of an unweighting based on machine learning were introduced. The quality of prediction was afterwards measured by a newly developed error definition which quantifies the variance of the new weight, but this number does not say anything about the quality of this approach in comparison to the currently implemented methods.

The most rigorous idea of doing this would be an full implementation of the machine learning approach in SHERPA. This would require a big effort and was not feasible due to the time limitations of the master thesis. However, even if this was doable, the estimation of the possible gain can prevent a lot of unnecessary work.

For this reason, this chapter is divided into two main parts, one which describes the used methods for evaluation and one where they are applied for different processes.

In the first part methods are developed to estimate the performance gain by using a set of truth-prediction pairs in combination with the timings of exact weight calculation and approximation. In contrast to the error definition used for optimization, here also the probabilities for surviving the acceptance condition have to be included. This can be done by means of the weighted mean and the weighted standard deviation, which are defined as follows:

$$\bar{t} = \frac{\sum_{i=1}^N p_i t_i}{\sum_{i=1}^N p_i}, \quad (5.1)$$

$$\sigma(t) = \sqrt{\frac{\sum_{i=1}^N p_i t_i^2}{\sum_{i=1}^N p_i} - \left(\frac{\sum_{i=1}^N p_i t_i}{\sum_{i=1}^N p_i} \right)^2}. \quad (5.2)$$

Here, the p_i are the weights, when setting them to $\frac{1}{N}$ all t_i have the same probability and one obtains the known expressions of mean and standard deviation.

These methods were used in the second part to evaluate first the training process and afterwards some others, too.

5.1 Direct Use of Machine-Learning based Unweighting

As mentioned in Section 4.1, in the machine learning based unweighting procedure the resulting events still need weights to compensate errors by the acceptance condition. Nevertheless, it can be useful to use them directly because two effects are interfering: indeed non-uniform weights

increase the error, but using many of them can decrease it by statistics. Which of these effects dominates depends on the machine learning performance, how this can be estimated is shown in this section.

5.1.1 Error Estimation

When creating differential cross-sections with weighted events, all events are entered with their weights into the histogram. The crucial point is the error of sum of weights. In case of many events or a low variance the sum can be approximated with the product $s = k \cdot \bar{t}$, given by the ‘number of events’ (k) times the ‘average weight’ ($\bar{t}$). The relative error of this product can be expressed for all methods as a function of k , the exact weights w and the predictions g , using the weighted mean and standard deviation as defined above.

k is assumed to be Poisson distributed, leading to an error of $\sqrt{k}$. $\bar{t}$ is a mean, its error can be written as ratio of the standard deviation of t and $\sqrt{k}$. Furthermore, $\bar{t}$ and k are uncorrelated. Using this, the error of s can be expressed as follows:

$$\Delta(s) = \sqrt{\left(\sqrt{k} \cdot \bar{t}\right)^2 + \left(k \cdot \frac{\sigma(t)}{\sqrt{k}}\right)^2}. \quad (5.3)$$

The relative error of s is the ratio of $\sigma(s)$ and s and can be calculated directly:

$$\frac{\Delta(s)}{s} = \frac{\sqrt{\left(\sqrt{k} \cdot \bar{t}\right)^2 + \left(k \cdot \frac{\sigma(t)}{\sqrt{k}}\right)^2}}{k \cdot \bar{t}} = \sqrt{\frac{1}{k}} \cdot \frac{\sqrt{\bar{t}^2 + \sigma(t)^2}}{\bar{t}} = \sqrt{\frac{1}{k}} \cdot \sqrt{1 + \frac{\sigma(t)^2}{\bar{t}^2}}. \quad (5.4)$$

Equation (5.4) can now be evaluated for all methods. Henceforth, the exact weights as calculated by SHERPA are written as w , the prediction by the algorithm as g and the final weight as obtained when using the machine-learning based unweighting procedure as x .

Unweighting by SHERPA

When using the classic unweighting procedure as currently implemented in SHERPA, all remaining events have the same weight. Their variance is therefore zero and the relative error depends only on k .

$$\frac{\Delta(s)}{s}_{(uw)} = \sqrt{\frac{1}{k}} \quad (5.5)$$

Weighted Events

When using events with weight w , their mean is $\bar{w}$ and their standard deviation can be expressed as $\sigma(w) = \sqrt{\bar{w}^2 - \bar{w}^2}$. This can be inserted in Equation (5.4) to receive an expression

for the relative error by using weighted events:

$$\frac{\Delta(s)}{s}_{(w)} = \frac{1}{\sqrt{k}} \cdot \sqrt{1 + \frac{\sigma(x)^2}{\bar{x}^2}} = \frac{1}{\sqrt{k}} \cdot \sqrt{1 + \frac{\bar{w}^2 - \bar{w}^2}{\bar{w}^2}} = \frac{1}{\sqrt{k}} \cdot \sqrt{\frac{\bar{w}^2}{\bar{w}^2}}. \quad (5.6)$$

Machine Learning based Unweighting

In this approach the new weights x are given by $x = \frac{w}{g}$, where w are the true weights as used above and g the corresponding predictions. However, in order to obtain an expression for $\frac{\sigma(s)}{s}_{(ml)}$ one also has to include the probability for each event to survive the acceptance condition. An event is kept with weight $\frac{w}{g}$ if the ratio $\frac{g}{w_{\max}}$ is larger than a randomly chosen number of the interval $[0,1]$. Following this, the mean and standard deviation can be obtained by weighting the new weight x with $p = \frac{g}{w_{\max}}$:

$$\bar{x} = \frac{\sum_i p_i \cdot x_i}{\sum_i p_i} = \frac{\sum_i \frac{g_i}{w_{\max}} \cdot \frac{w_i}{g_i}}{\sum_i \frac{g_i}{w_{\max}}} = \frac{\sum_i w_i}{\sum_i g_i} = \frac{\bar{w}}{\bar{g}} \quad (5.7)$$

$$\sigma(x) = \sqrt{\frac{\sum_i p_i x_i^2}{\sum_i p_i} - \left(\frac{\sum_i p_i x_i}{\sum_i p_i} \right)^2} = \sqrt{\frac{\sum_i \frac{g_i}{w_{\max}} \left(\frac{w_i}{g_i} \right)^2}{\sum_i \frac{g_i}{w_{\max}}} - \left(\frac{\bar{w}}{\bar{g}} \right)^2} = \sqrt{\frac{\left(\frac{\bar{w}^2}{\bar{g}} \right)}{\bar{g}} - \left(\frac{\bar{w}}{\bar{g}} \right)^2} \quad (5.8)$$

Inserting these into Equation (5.4) makes it possible to obtain an expression for $\frac{\sigma(s)}{s}_{(ml)}$:

$$\frac{\Delta(s)}{s}_{(ml)} = \sqrt{\frac{1}{k}} \cdot \sqrt{1 + \frac{\sigma(x)^2}{\bar{x}^2}} = \sqrt{\frac{1}{k}} \cdot \sqrt{\left(\frac{\bar{w}^2}{\bar{g}} \right) \cdot \frac{\bar{g}}{\bar{w}^2}}. \quad (5.9)$$

This result is compatible with $\frac{\Delta(s)}{s}_{(uw)}$ and $\frac{\Delta(s)}{s}_{(w)}$. In case of an perfect prediction w is equal to g and second square root becomes equal to one. If the prediction is very bad and predicts always to a constant (e.g. the mean weight), g cancels out and $\frac{\Delta(s)}{s}_{(ml)}$ and $\frac{\Delta(s)}{s}_{(w)}$ become equal. Furthermore, all relative errors decrease by $\frac{1}{\sqrt{k}}$ with an increasing number of events, which is the expected behavior due to statistics.

5.1.2 Comparisons of Timings

In the previous section the relative error for the different methods was expressed in dependence of the number of events k . In order to achieve the same relative error with e.g. weighted and unweighted events, one needs much more weighted events than unweighted ones. However, on the other side generating one weighted event is much faster then the generation of one unweighted one.

The best way of event generation is therefore the method which minimizes the overall time under the condition of having a constant relative error. This was studied for two different scenarios. In the first one, the events generated by SHERPA were used directly without any further manipulation, here it is called the ‘theoretical scenario’. The second ‘experimental scenario’ deals with a typical experimental issue, where all events have to pass a second very time consuming process, the simulation of their behavior in detectors. In both cases the time for training the algorithm is not considered because it can – at least if it does not take too long – be included into the integration phase.

The ‘Theoretical Scenario’

In this scenario only the timings for event generation contribute to the overall time. They can be written for all methods as a function of the final number of events (k), the time for the exact calculation of one weight (t_{gen}) and the time for the approximated prediction of one weight (t_{pred}).

In case of generating weighted events, the weights have to be calculated exactly k times.

$$t_w = k \cdot t_{\text{gen}} \quad (5.10)$$

When generating unweighted events, the unweighting efficiency $\epsilon_{\text{uw}} = \frac{\bar{w}}{w_{\text{max}}}$ determines how many weights have to be calculated exactly to obtain one unweighted event. The overall time can therefore be written as follows:

$$t_{\text{uw}} = \frac{k}{\epsilon_{\text{uw}}} \cdot t_{\text{gen}}. \quad (5.11)$$

A more complicated situation is given when using the machine-learning approach. The number of weights which have to be predicted is determined by a modified unweighting efficiency $\epsilon_{\text{ml}} = \frac{\bar{g}}{w_{\text{max}}}$, but for all final events the weight has still to be calculated exactly:

$$t_{\text{ml}} = \left(\frac{1}{\epsilon_{\text{ml}}} - 1 \right) \cdot k \cdot t_{\text{pred}} + k \cdot t_{\text{gen}} \approx k \cdot \left(\frac{t_{\text{pred}}}{\epsilon_{\text{ml}}} + t_{\text{gen}} \right). \quad (5.12)$$

These equations make it possible to compare the performance of the different ways of event generation. The relative errors and overall timings can be shortly written by introducing method-specific constants a_{method} and b_{method} :

$$\frac{\Delta(s)}{s}_{\text{method}} = \sqrt{\frac{1}{k}} \cdot a_{\text{method}} \quad (5.13)$$

$$t_{\text{method}} = k \cdot b_{\text{method}}. \quad (5.14)$$

Under the condition of having a predefined relative error, the first equation can be transformed to k and inserted into the second. Only the ratios of the overall timings are important to

compare the different methods. The relative error is required to be equal for all methods and therefore cancels out.

$$\frac{t_1}{t_2} = \frac{a_1^2 b_1}{a_2^2 b_2} \quad (5.15)$$

For this scenario the weighted events were chosen as the reference model to compare the performance with.

The ‘Experimental Scenario’

In this scenario all events pass after their generation a further treatment, e.g. a detector simulation. This increases all timings by k times the simulation time for one event, t_{sim} :

$$t_{\text{uw}} = \frac{k}{\epsilon_{\text{uw}}} \cdot t_{\text{gen}} + k \cdot t_{\text{sim}} = k \cdot \left(\frac{t_{\text{gen}}}{\epsilon_{\text{uw}}} + t_{\text{sim}} \right) \quad (5.16)$$

$$t_{\text{w}} = k \cdot t_{\text{gen}} + k \cdot t_{\text{sim}} = k \cdot (t_{\text{gen}} + t_{\text{sim}}) \quad (5.17)$$

$$t_{\text{ml}} = k \cdot \left(\frac{t_{\text{pred}}}{\epsilon_{\text{ml}}} + t_{\text{gen}} \right) + k \cdot t_{\text{sim}} = k \cdot \left(\frac{t_{\text{pred}}}{\epsilon_{\text{ml}}} + t_{\text{gen}} + t_{\text{sim}} \right). \quad (5.18)$$

Currently, unweighted events obtained by the classic unweighting procedure are most often used for this scenario, they are therefore the reference. t_{sim} is not known exactly, but usually much higher than t_{gen} . Therefore, it is not possible to calculate the possible performance gain directly. However, in order to evaluate this scenario, the ratio $\frac{t_{\text{method}}}{t_{\text{uw}}}$ can be plotted over the ratio $\frac{t_{\text{sim}}}{t_{\text{gen}}}$ for all methods.

5.2 Second Unweighting

Even if the events obtained by the machine-learning approach would perform better than the real unweighted ones in the experimental scenario, too, there can still be some reasons why only events with the same weight are desired. These reasons can be e.g. motivated by memory limitations or just to have an easier handling.

A possible way to generate such events by means of machine learning is to reweight the weighted events obtained by the machine-approach using the same procedure as for the classic unweighting. The full method is introduced in Figure 5.1. Two unweighting efficiencies are now decisive for the success of this method. The first is equal to the efficiency used for the pure machine-learning approach and given by $\epsilon_{\text{ml}} = \frac{\bar{g}}{w_{\text{max}}}$, it defines the fraction of events which survive the first unweighting, respectively the first loop in Figure 5.1. The second efficiency $\epsilon_{\text{suw}} = \frac{\bar{x}}{x_{\text{max}}}$ describes the ratio of the number of final remaining events to the number of events which passed the first machine-learning unweighting procedure.

Using this expressions one can compare the classic unweighting procedure with the new ‘second unweighting’ method. In order to generate one unweighted event using the ‘second unweighting’ method, on average $\frac{1}{\epsilon_{\text{ml}} \cdot \epsilon_{\text{suw}}}$ weights have to be predicted, but only $\frac{1}{\epsilon_{\text{suw}}}$ weights to be calculated exactly. By contrast, in the classic unweighting procedure $\frac{1}{\epsilon_{\text{uw}}}$ weights have to be calculated exactly to obtain one unweighted event. These relations allow to calculate the ratio $\frac{t_{\text{suw}}}{t_{\text{uw}}}$, the ‘second unweighting’ performs better than the ‘classic unweighting’ if this ratio is smaller than unity:

$$\frac{t_{\text{suw}}}{t_{\text{uw}}} = \frac{\left(\frac{t_{\text{pred}}}{\epsilon_{\text{ml}} \cdot \epsilon_{\text{suw}}} + \frac{t_{\text{gen}}}{\epsilon_{\text{suw}}} \right)}{\frac{t_{\text{gen}}}{\epsilon_{\text{uw}}}} = \frac{\epsilon_{\text{uw}}}{\epsilon_{\text{ml}} \cdot \epsilon_{\text{suw}}} \cdot \frac{t_{\text{pred}}}{t_{\text{gen}}} + \frac{\epsilon_{\text{uw}}}{\epsilon_{\text{suw}}}. \quad (5.19)$$

The second term in the sum does only depend on the prediction’s quality, to minimize the first term the prediction has furthermore to be fast.

5.2.1 Determination of the Second Unweighting Efficiency

In order to evaluate the time ratio in Equation (5.19), all unweighting efficiencies have to be determined by using the available sets of prediction / truth pairs of around 5 millions events¹. For ϵ_{uw} and ϵ_{ml} the corresponding mean and maxima can be estimated directly because all 5 million points can be used and should give enough statistics.

The estimation of x_{max} and $\bar{x}$ is more difficult. When using the available data set to do the first machine-learning unweighting explicitly, only some thousand events will remain and may cause high errors when estimating x_{max} . This can be avoided when using the simulated x -distribution as done for the error estimation in Section 5.1.1. $\bar{x}$ is then directly given by $\frac{\bar{w}}{\bar{g}}$. The determination of the maximum is not as easy. Weights which will be underestimated by the machine-learning algorithm will result in a high x , but also be suppressed due to the acceptance condition. This means there can be very high x , but their probability to be accepted is very low. However, in Equation (5.19) only the ratio of $\frac{\epsilon_{\text{uw}}}{\epsilon_{\text{suw}}}$ is important. This ratio can be approximated by $\frac{\epsilon_{\text{uw}}^*}{\epsilon_{\text{suw}}^*}$, where ϵ^* is given by a maximum which only covers 99.9% of the total cross section². The maxima used for this thesis were obtained by filling all events with their weights into histograms and integrating them until 99.9% of the total sum is covered. This method can directly be applied to determine w_{max}^* . In order to calculate x_{max}^* , all events have to be weighted with both, the weight x but also the probability to receive this weight ($\frac{g}{w_{\text{max}}}$). For example, both histograms are plotted for the training process in the following section, where all results are summarized for this process.

¹In order to increase the accuracy, the validation data set was increased to 5 million sample points.

²A similar method can be used in SHERPA to improve the unweighting performance, it is called ‘partial unweighting’. The maximum weight to unweight against will be reduced, only a small predefined part of the total cross section is allowed to still be weighted afterwards[51].

```

1 repeat
2   repeat
3     generate event  $E$ ;
4     calculate the corresponding approximated weight  $g$ ;
5     generate uniform distributed random number  $R \in (0, 1)$ ;
6   until  $g > R \cdot w_{\max}$ ;
7   calculate exact weight  $w$  for event  $E$ ;
8    $x = \frac{w}{g}$ ;
9   generate uniform distributed random number  $R \in (0, 1)$ ;
10 until  $x > R \cdot x_{\max}$ ;
11 keep event  $E$  with weight = 1;

```

Figure 5.1: Algorithm for the generation of one unweighted event using the ‘second-unweighting’ method.

5.3 Evaluation of the Training Process

Three different applications were introduced, where machine learning may help to improve SHERPA’s performance. As seen in the previous sections, not only a minimal error but also a fast prediction is important, especially for the ‘theoretical scenario’ and the ‘second unweighting’. In order to verify which of the identified configurations performs best, four different configurations of neural networks and the performance-optimized random forest were compared. The networks’ configurations are two one-layer networks with 40 or 100 neurons, one two-layer network with 100 neurons per layer and one three-layer network, again with 100 neurons per layer. The small network of 30 neurons is the fastest and the three-layer network the most precise one. All timings, errors and results are summarized in Table 5.1.

5.3.1 Direct Methods

When studying the ‘theoretical scenario’ as described in Section 5.1.2, the ratio $t_{\text{ml}}/t_{\text{w}}$ is for all tested configurations smaller than one. The greatest performance gain was achieved by using a neural network with 100 neurons in one layer, the machine-learning based approach with this configuration reduces the time to generate events with a given error to 41%. Further, it gets also clear why weighted events are the reference for this scenario. If one uses unweighted events obtained by the classic unweighting procedure, one would indeed need less of them to receive the same relative error. However, generating them would nevertheless take more than 130 times longer.

In the ‘experimental scenario’ the situation is different. In Figure 5.2 the time ratio $t_{\text{method}}/t_{\text{uw}}$ is drawn as a function of the ratio $t_{\text{sim}}/t_{\text{gen}}$ for different configurations. t_{method} and t_{uw} are the compared overall timings of the selected method and the classic unweighting, t_{gen} is the measured time needed for the generation of one weighted event and t_{sim} is the assumed time

for the detector simulation of one event. The second is usually much higher than the first, here t_{gen} is 0.05 seconds whereas t_{sim} can be on the order of some minutes, resulting in a time ratio larger than 1000.

Figure 5.2 shows that the prediction time has only a small influence. The configuration with the lowest error gives the best result, too³. However, no configuration was found which can perform better than the real unweighted events in this scenario.

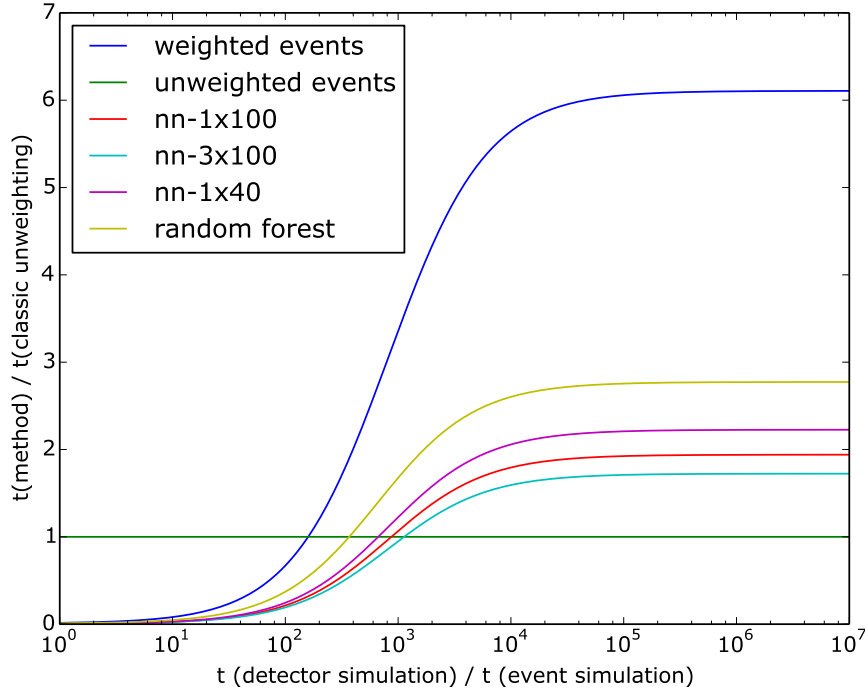


Figure 5.2: Comparison of the expected time ratios for several configurations and methods in the ‘experimental scenario’.

³This can be seen as the subsequent confirmation of the definition of f in Section 4.2.3, indeed the lowest error gives the best result.

	general results		theoretical scenario	second unweighting
	t_{pred} [s/events]	error f	$t_{\text{ml}}/t_{\text{w}}$	$t_{\text{suw}}/t_{\text{uw}}$
rf	1.48e-5	3.19	0.63	$0.68 = 0.15 + 0.53$
nn-1x40	9.8e-6	2.91	0.44	$0.34 = \mathbf{0.05} + 0.29$
nn-1x100	1.58e-5	2.61	0.41	0.29 = 0.07 + 0.22
nn-2x100	3.39e-5	2.52	0.49	$0.30 = 0.13 + 0.17$
nn-3x100	5.48e-5	2.48	0.61	$0.32 = 0.17 + \mathbf{0.15}$

Table 5.1: Evaluation of the ‘theoretical scenario’ and the ‘second unweighting’ for different algorithms and configurations. ‘rf’ means the random forest in the performance optimized configuration, ‘nn-2x100’ means e.g. a neural network with two layer and 100 neurons per layer. Independent of the chosen algorithm, t_{gen} is equal to 0.05 [s/events] and $t_{\text{uw}}/t_{\text{w}}$ equal to 134.9 in the theoretical scenario.

5.3.2 Second Unweighting

As described in Section 5.2, the ‘second unweighting’ approach is a method to obtain real unweighted events by means of machine learning. The reference model is therefore the classic unweighting method, their performance can be compared by the ratio $t_{\text{suw}}/t_{\text{uw}}$. These ratios are given for all methods in Table 5.1. As well as in the ‘theoretical scenario’, here both a precise and fast prediction are required. The influence of them can be estimated by comparing both terms of Eq. (5.19). The first is dominated by the timings and minimal for the fastest algorithm, a neural net with only 40 neurons. The second term does only depend on the quality of the prediction and is minimal for the complex network with three layers, each with 100 neurons. The best compromise is given by a one-layer network of 100 neurons. Using this configuration, unweighted events can be generated more than three times faster as with the current unweighting procedure.

The unweighting efficiencies were determined as described in Section 5.2.1. In Figure 5.3 the distribution of weights is shown, first for the weighted events produced by SHERPA (weight w) and second as expected for the machine-learning approach (by weighting all new weights $x = \frac{w}{g}$ with their probability $\frac{g}{w_{\text{max}}}$). Their means are drawn as vertical green lines. They seem to be shifted, but this is an effect of the logarithmic binning. The bins do not have the same width, large bins contribute more to the mean. For comparison, the distribution of around 6000 weighted events, obtained by the machine-learning based unweighting by using 5 million events of SHERPA, is given in Figure 5.5. It agrees well with Figure 5.3b. Finally, the cumulative histograms as used for the determination of w_{max}^* and x_{max}^* are drawn in Figure 5.4. Here all events are filled in with their weight (as they contribute to the cross section), the chosen maxima which cover 99.9% of the total cross section are drawn as red vertical lines.

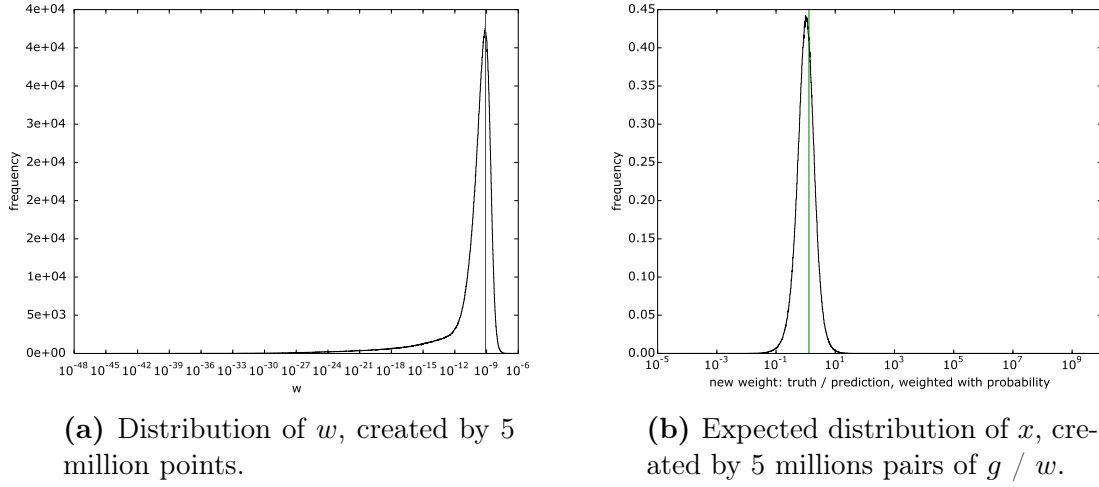


Figure 5.3: Distribution of weights for both methods, weighted events as generated by Sherpa (w) and weighted events as generated by the machine-learning based unweighting procedure (x). The mean values are drawn as green vertical lines.

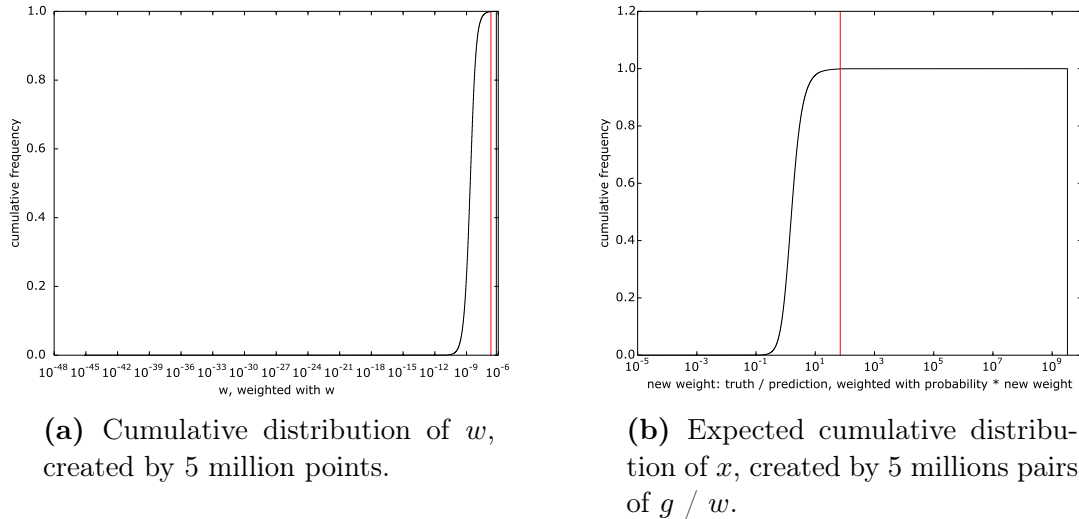


Figure 5.4: Cumulative distribution of weights for both methods, weighted events as generated by Sherpa (w) and weighted events as generated by the machine-learning based unweighting procedure (x). All events are entered with their weights, the maxima which cover 99.9% of the total cross section are drawn as red vertical lines.

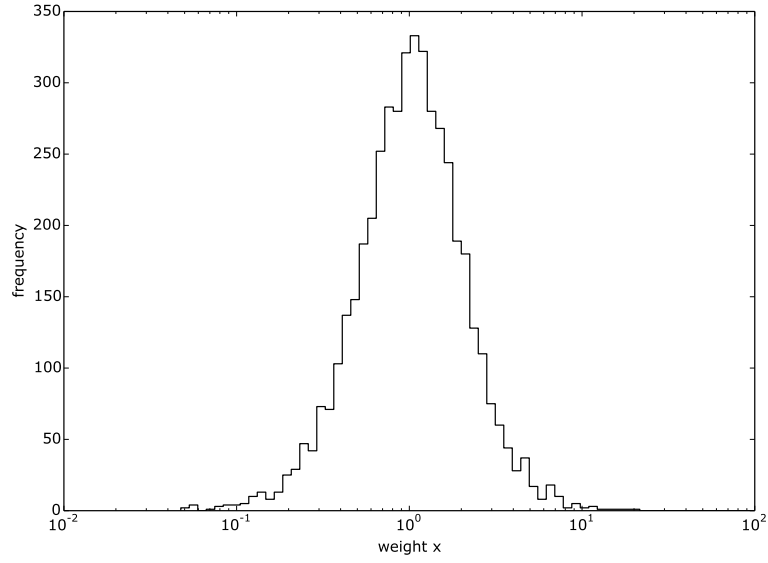


Figure 5.5: Distribution of around 6000 weights x , generated by explicit application of the machine-learning based unweighting procedure on 5 million weighted events of SHERPA.

5.4 Further Processes

In the previous evaluation only the process $gg \rightarrow e^+e^- \bar{d}dgg$ was probed. It was chosen for parameter optimization in the hope to be representative for a large number of interesting processes. It will not be feasible to adjust all parameters for each process new. For this reason, it is important to study whether the identified configuration of a one-layer neural network with 100 neurons can speed up the event generation of other processes, too. In the following sections only the ‘theoretical scenario’ and the ‘second unweighting’ are studied, the weighted events obtained by the machine-learning based approach will not be able to perform better than weightless events obtained by the ‘second unweighting’ in the ‘experimental scenario’.

5.4.1 $Z + \text{jets}$

The training process is part of the ‘ $Z + \text{jets}$ ’ production, the analysis is therefore continued with similar processes which also contribute to this group. The following processes were studied:

1. $d\bar{d} \rightarrow e^+e^- ggg$,
2. $u\bar{u} \rightarrow e^+e^- gggg$,
3. $d\bar{d} \rightarrow e^+e^- ugg$.

Some example Feynman diagrams representing these processes are drawn in Figure 5.6. All results for both the ‘theoretical scenario’ ($t_{\text{ml}}/t_{\text{w}}$) and the ‘second unweighting’ ($t_{\text{suw}}/t_{\text{uw}}$) are given in Table 5.2.

The first process, $d\bar{d} \rightarrow e^+e^-ggg$, is a $2 \rightarrow 5$ process and therefore expected to be easier to predict. Indeed the error is with 2.31 lower than the one of the training process, but also the calculation of weights is 20 times faster. This time ratio seems to have a big influence. Although the error is lower, the ratios $t_{\text{ml}}/t_{\text{w}}$ and $t_{\text{suw}}/t_{\text{uw}}$ are both larger than one, meaning the new method will not be able to speed up the event generation for this process.

The remaining processes are $2 \rightarrow 6$ processes. For $d\bar{d} \rightarrow e^+e^-uugg$ an error of 2.90 is obtained, but t_{gen} is relative low, too. The ratio $t_{\text{ml}}/t_{\text{w}}$, considered for the ‘theoretical scenario’, is still larger than one. Nevertheless the process would profit from the ‘second unweighting’ approach, the performance benefit is expected to be around 10%.

Finally, the process $d\bar{d} \rightarrow e^+e^-uugg$ was studied. Its error is 3.99 and with $0.055 \frac{\text{s}}{\text{event}}$ the exact weight calculation takes quite long. Despite the large error both $t_{\text{ml}}/t_{\text{w}}$ and $t_{\text{suw}}/t_{\text{uw}}$ are smaller than one. In the ‘theoretical scenario’ the expected time saving is around 40% and with the ‘second unweighting’ still 20%.

Of course, these three processes are not extensive, but nevertheless, they allow a small outlook. The new unweighting approach can in principle accelerate the generation of both weighted and unweighted events. It seems not to be beneficial for ‘fast’ processes, in this case e.g. processes with five or less final-state particles.

The next step would be the examination of more complicated processes with e.g. seven final-state particles. Doing this with the currently used approach is difficult. Up to now the matrix elements of all processes were calculated by ‘AMEGIC’[52] which is Sherpa’s original matrix-element generator. This generator is stretched to its limit in case of processes with many final states. The generation of $gg \rightarrow eeddg$ events was tried with AMEGIC, but it already failed to generate all necessary libraries because there are too many contributing diagrams. When using SHERPA for event generation, this problem can be avoided by using the newer ‘COMIX’[53] generator as an alternative generator for these processes. However, the color charges are not summed over in matrix elements calculated with COMIX. In order to use this generator for a machine learning based unweighting procedure, there would be additional steps necessary which are beyond the scope of this thesis. A second possibility for further study are other processes as ‘ $pp \rightarrow Z + \text{jets}$ ’, e.g. ‘ $pp \rightarrow W + \text{jets}$ ’ or ‘ $pp \rightarrow tt + \text{jets}$ ’. Doing this would probably help to establish whether the presented new unweighting procedure can be generalized, but it is likely that the interesting processes with a long generation time will be challenging for AMEGIC, too. Nevertheless, some processes of the group ‘ $W + \text{jets}$ ’ were studied additionally.

5.4.2 $W + \text{jets}$

The possible performance gain for the following processes was studied:

1. $d\bar{u} \rightarrow e^-\bar{\nu}_e ggg$

$$2. \ d\bar{u} \rightarrow e^- \bar{\nu}_e gggg$$

$$3. \ d\bar{u} \rightarrow e^- \bar{\nu}_e uggg.$$

Example Feynman diagrams of these processes are given in Figure 5.7. All expected time ratios for the ‘theoretical scenario’ and the ‘second unweighting’ are summarized in Table 5.3. The generation of weighted events with the machine-learning approach seems not to be beneficial, all ratios $t_{\text{ml}}/t_{\text{w}}$ are larger than one. The situation does change when considering the expected time ratios of the ‘second unweighting’ approach. For both $2 \rightarrow 6$ processes, weightless events are expected to be producible with time savings of 17%, respectively 46%, in comparison to the currently used unweighting approach.

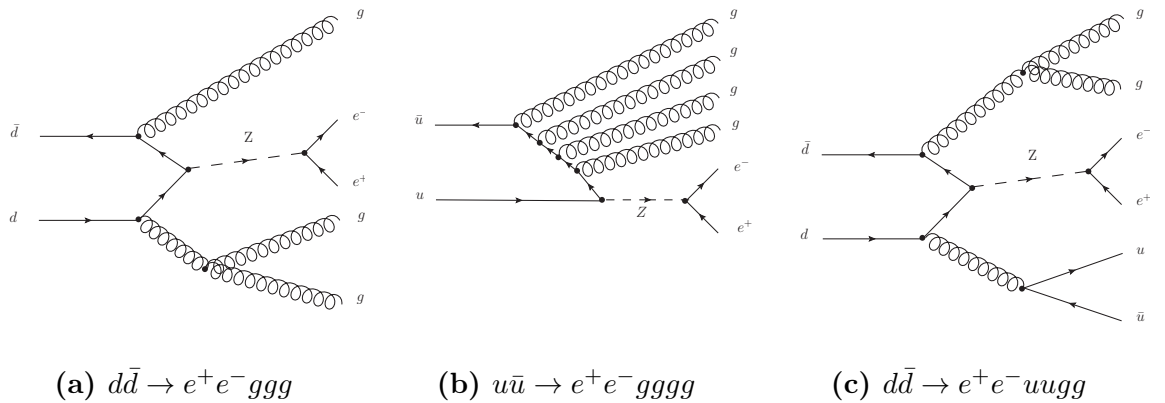


Figure 5.6: Example Feynman diagrams, representing further processes of the group ‘ $Z + \text{jets}$ ’ which were probed in the evaluation.

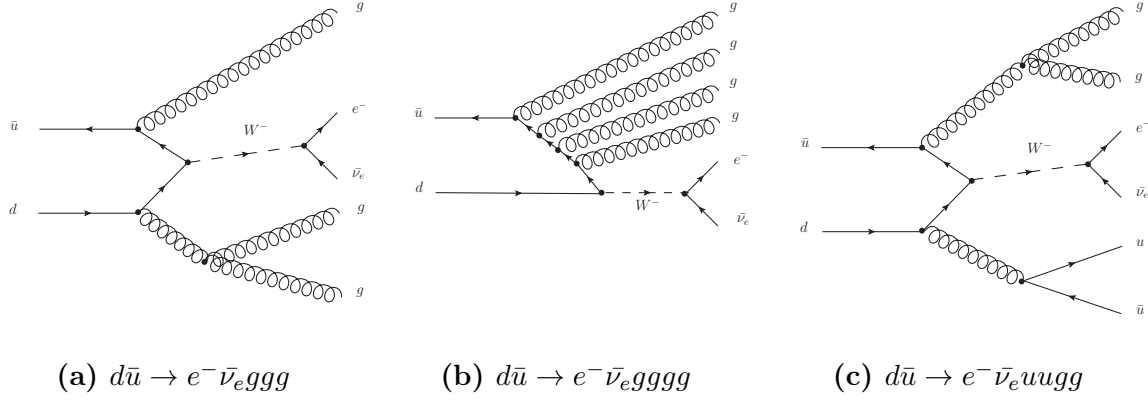


Figure 5.7: Example Feynman diagrams, representing further processes of the group ‘ W + jets’ which were probed in the evaluation.

process	error	t_{gen} [s]	t_{pred} [s]	$t_{\text{ml}}/t_{\text{w}}$	$t_{\text{suw}}/t_{\text{uw}}$
$dd \rightarrow e^+ e^- ggg$	2.31	0.0025	1.44e-5	2.31	$1.05 = 0.90 + 0.15$
$d\bar{d} \rightarrow e^+ e^- uugg$	2.90	0.014	1.46e-5	1.21	$0.92 = 0.67 + 0.25$
$u\bar{u} \rightarrow e^+ e^- gggg$	3.99	0.055	1.52e-5	0.59	$0.79 = 0.27 + 0.52$
$gg \rightarrow eeddg$	2.61	0.050	1.58e-5	0.41	$0.29 = 0.07 + 0.22$

Table 5.2: Results of the evaluation of further processes, belonging to the group ‘ Z + jets’. The results for the training process are given for comparison below.

process	error	t_{gen} [s]	t_{pred} [s]	$t_{\text{ml}}/t_{\text{w}}$	$t_{\text{suw}}/t_{\text{uw}}$
$d\bar{u} \rightarrow e^- \bar{\nu}_e ggg$	2.18	0.0027	1.41e-5	1.67	$1.21 = 0.97 + 0.24$
$d\bar{u} \rightarrow e^- \bar{\nu}_e gggg$	4.17	0.0399	1.49e-5	1.01	$0.83 = 0.44 + 0.39$
$d\bar{u} \rightarrow e^- \bar{\nu}_e uugg$	3.60	0.0099	1.46e-5	1.23	$0.54 = 0.46 + 0.08$

Table 5.3: Results of the evaluation of further processes, belonging to the group ‘ W + jets’.

6 Conclusion and Outlook

In particle physics, event generators are an essential tool to compare theoretical predictions with experimental results. There are many applications where the generated events are desired to be weightless, meaning they should follow their underlying physical probability distribution. In Sherpa such events are currently generated by a ‘hit-or-miss’ method, called ‘unweighting’. This method becomes very inefficient with increasing complexity of the processes considered. In order to prepare for the analyses of current and upcoming experiments, new methods for event generation are needed.

In this thesis a new approach of event generation is introduced. It combines the currently used unweighting method with the power of machine-learning algorithms by using fast, approximated weights for the acceptance condition of the ‘hit-or-miss’ procedure. The arising error will be compensated by assigning a new weight to all remaining events. The resulting events can be used directly as weighted events with a lower variance or can be unweighted again to obtain weightless events in a shorter time.

Examining whether this new approach works and estimating its possible performance gain were further parts of this thesis. In order to do so, several algorithms and configurations were studied to optimize the prediction for a chosen example process. Furthermore, methods were developed to compare the new approach with the currently used methods of event generation without the need of an implementation.

The best configuration uses a neural network with one hidden layer and 100 neurons for the prediction, the possible performance gain was determined for the training process and three similar ones, all part of ‘ $Z + \text{jets}$ ’ production, and three processes of ‘ $W + \text{jets}$ ’ production.

Using the new method, weighted events with the same relative error as for weighted events using the old method can be produced with time savings of around 50% for two processes of the ‘ $Z + \text{jet}$ ’ group. Weighted events of all tested processes of the ‘ $W + \text{jets}$ ’ group are not expected to profit by this method.

More promising results are obtained when considering weightless events, for all $2 \rightarrow 6$ processes of both tested groups they are expected to be producible with time saving between 8% and 70%. However, the new unweighting method is not suitable for processes which are already relatively fast, in this case e.g. processes with five or less final-state particles.

All expected performance gains are only rough estimates and have to be treated with caution. First of all, they were obtained only by statistical considerations and not by an implementation.

Errors are also to be expected due to the measured timings, necessary for an exact weight calculation and prediction. All calculations were done on a local cluster, interferences due to other running processes can not be excluded. Furthermore, all times needed for scaling the data, transforming the data or throwing the random numbers were assumed to be negligible in contrast to the time needed for an exact calculation or the prediction, respectively.

Within the scope of this thesis only the study of some easy processes was feasible, there are many more steps necessary before one can think of implementing this method in Sherpa. Open questions are e.g. how this new approach can be combined with the ‘COMIX’ matrix-element generator to study processes with more final-state particles and if the identified configuration can be applied to other processes, too.

A Run Cards

A.1 Z + jets

Run.dat

```
(run){
  EVENTS = 1000;
  BEAM_1 = 2212; BEAM_ENERGY_1 = 4000;
  BEAM_2 = 2212; BEAM_ENERGY_2 = 4000;

  FRAGMENTATION = Off;
  MI_HANDLER = None;
  ME_SIGNAL_GENERATOR=Amegic;
  CSS_MAXEM=0;
  ME_QED=0;
  EVENT_GENERATION_MODE=Weighted;
}(run)

(processes){
# one process was selected, all other were uncommented
  Process 21 21 -> 11 -11 1 -1 21 21;      # g g -> e^- e^+ d bd g g
  Process 1 -1 -> 11 -11 21 21 21;          # d bd -> e^- e^+ g g g
  Process 1 -1 -> 11 -11 2 -2 21 21;         # d db -> e^- e^+ u bu g g
  Process 2 -2 -> 11 -11 21 21 21 21;       # u bu -> e^- e^+ g g g g

  Order_EW 2;
  CKKW sqr(30/E_CMS);
  Cut_Core 1;
  End process;
}(processes)

(selector){
  Mass 11 -11 66 116
}(selector)
```

A.2 $W + \text{jets}$

Run.dat

```
(run){
  EVENTS = 400000
  BEAM_1 = 2212; BEAM_ENERGY_1 = 4000;
  BEAM_2 = 2212; BEAM_ENERGY_2 = 4000;

  FRAGMENTATION = Off
  MI_HANDLER = None
  ME_SIGNAL_GENERATOR=Amegic
  CSS_MAXEM=0
  ME_QED=0
  EVENT_GENERATION_MODE=Weighted
  #PSH_TRAINING_FILE=TPoints
}(run)

(processes){
# one process was selected, all other were uncommented
  Process 1 -2 -> 11 -12 21 21 21 # d bu -> e bv_e g g g g
  Process 1 -2 -> 11 -12 21 21      # d bu -> e bv_e g g g
  Process 1 -2 -> 11 -12 21 2 -2 # d bu -> e bv_e g g u bu

  Order_EW 2;
  CKKW sqr(30/E_CMS)
  Cut_Core 1
  Print_Graphs graphs/;
  End process;
}(processes)

(selector){
  Mass 11 -11 66 116
}(selector)
```

B Plots

B.1 $gg \rightarrow e^+e^- \bar{d}d gg$

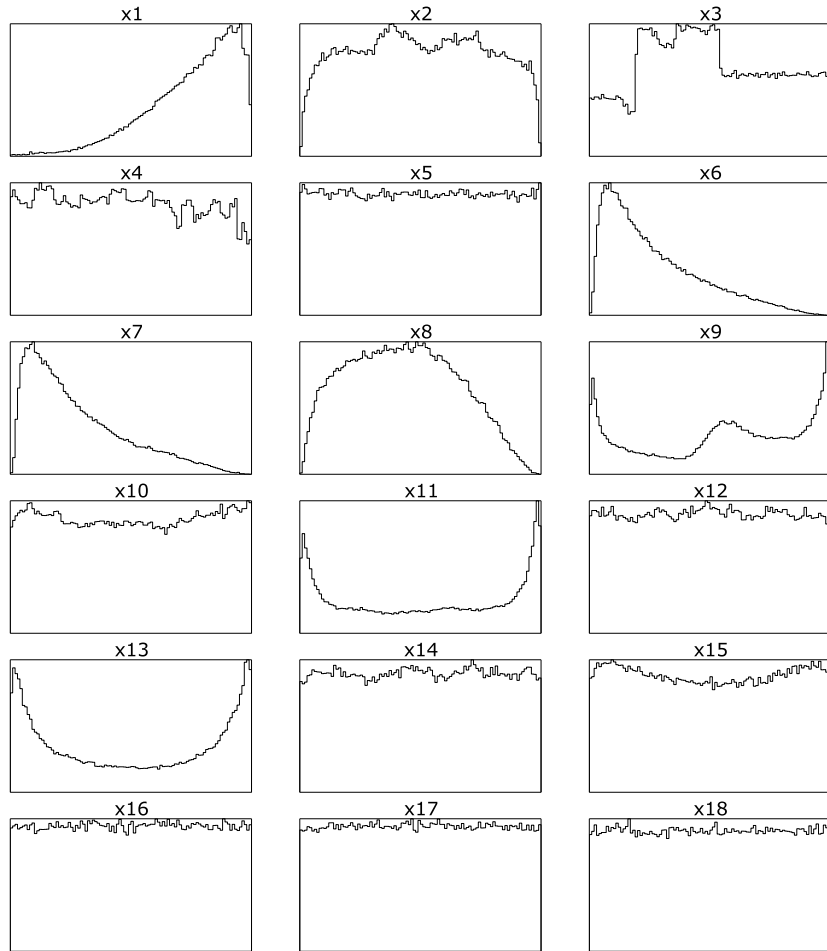


Figure B.1: Histograms of SHERPA's random numbers, weighted with the weight w . Axis labels are removed for clarity.

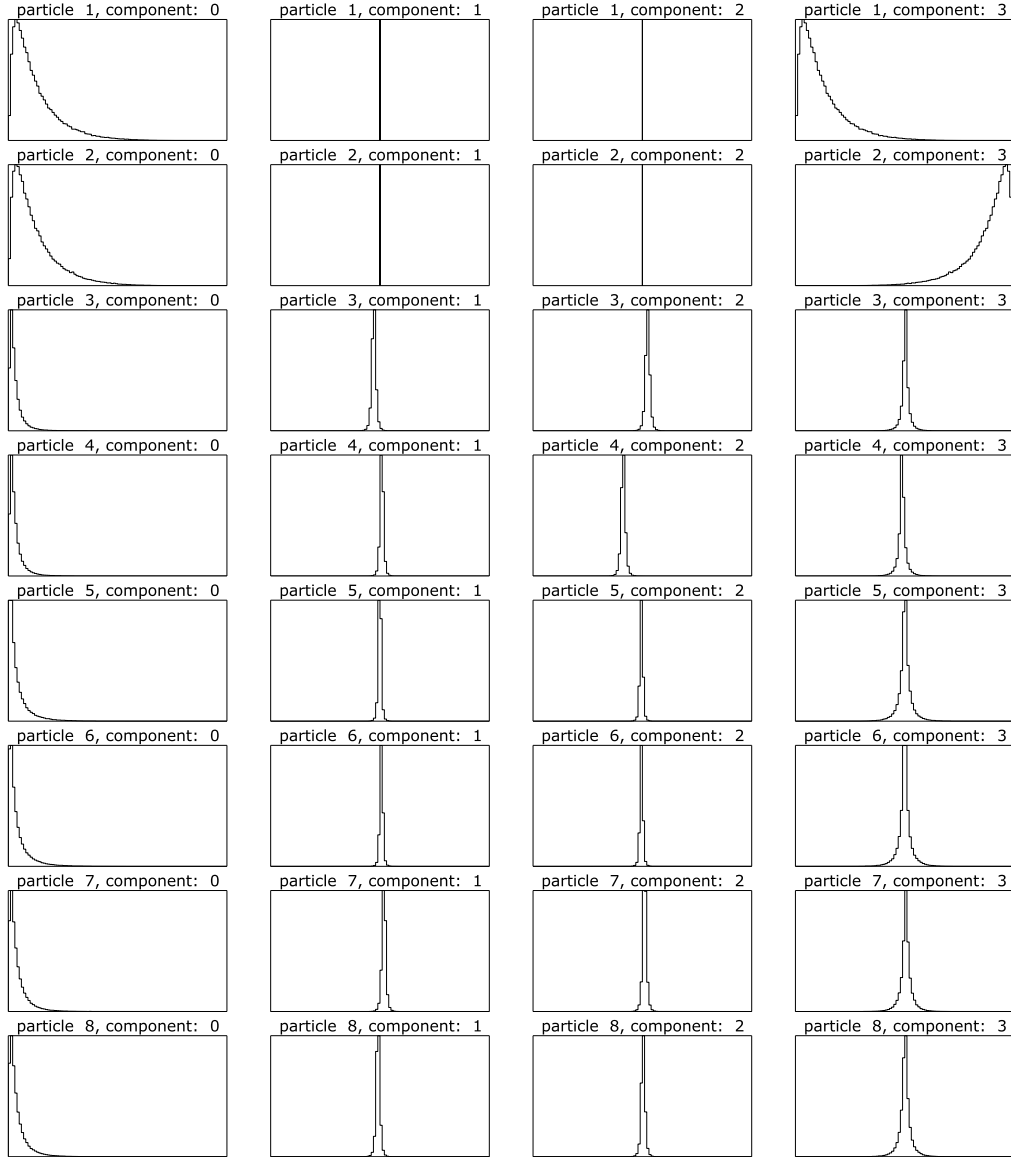


Figure B.2: Histograms of momenta, weighted with the weight w . Particles 1 and 2 are the incoming partons. Axis labels are removed for clarity.

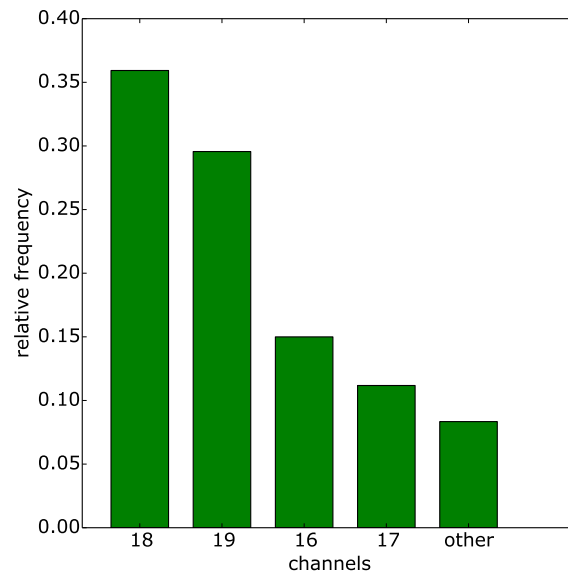


Figure B.3: frequency distribution of initial channels

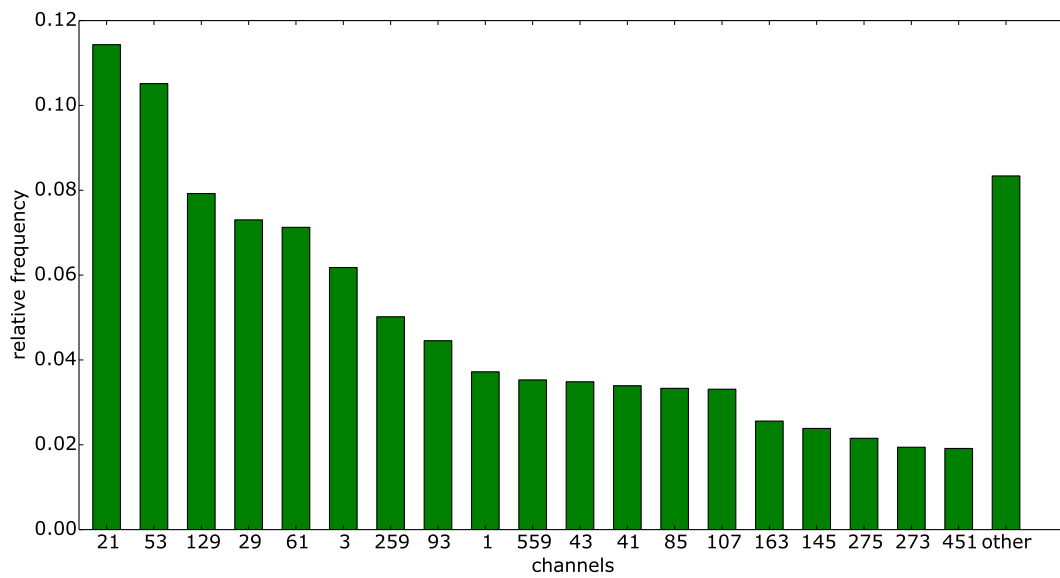


Figure B.4: frequency distribution of final channels

List of Figures

2.1	Picture of the LHC tunnel and schematic of the ATLAS detector.	10
2.2	Feynman diagrams of Z production where the Z boson decays into two quarks. . .	12
2.3	Steps in the simulation of the production of a Higgs boson and two top quarks. . .	14
3.1	Monte Carlo approximation of π	16
3.2	classification of machine learning methods	20
3.3	example: decision tree	22
3.4	single-layer perceptron	24
3.5	multilayer perceptron	25
4.1	comparison of the classic and machine-learning based unweighting attempts	28
4.2	two Feynman diagrams, representing the $gg \rightarrow d\bar{d}e^+e^-gg$ process	32
4.3	Histogram of the weights, generated by 1 million sample points. Process: $gg \rightarrow d\bar{d}e^+e^-gg$	32
4.4	example of a x -distribution for the process $dd \rightarrow eeg$	34
4.5	comparison of some of the most popular algorithms implemented in scikit-learn . .	36
4.6	feature importances of an unoptimized random forest, trained with SHERPA's random numbers	37
4.7	random forest: error and timings as function of the number of samples for training	42
4.8	random forest: error and timings as function of the number of trees	42
4.9	random forest: error and timings as function of the maximal depth of each tree . .	43
4.10	random forest: error and timings as function of the maximal number of used features	43
4.11	random forest: error and timings as function of the maximal number of leafs . . .	44
4.12	random forest: error and timings as function of the minimal number of samples per leaf	44
4.13	random forest: error and timings as function of the minimal number of samples per splitting / node	45
4.14	comparison of different activation functions implemented in FANN	48
4.15	validation error as function of learning rate and momentum	50
4.16	neural network: error and training time as function of the number of training sample points	50
4.17	neural network: errors and timings as function of the number of used neurons when using one layer	52

4.18	comparison of learning curves for different number of neurons with one hidden layer	52
4.19	neural network with 40 neurons in the first layer: error and timings as function of the number of neurons in the second hidden layer	53
4.20	neural networks: comparison of learning curves of two-layer networks with 40 neurons in the first	53
4.21	neural network with 100 neurons in the first layer: error and timings as function of the number of neurons in the second hidden layer	54
4.22	neural networks: comparison of learning curves of two-layer networks with 100 neurons in the first	54
4.23	errors and timings of neural networks with 30 neurons per layer	56
4.24	comparison of learning curves for neural networks with 30 neurons per layer . . .	56
4.25	errors and timings of neural networks with 100 neurons per layer	57
4.26	comparison of learning curves for neural networks with 100 neurons per layer . .	57
5.1	algorithm: the ‘second unweighting’	65
5.2	evaluation of the ‘experimental scenario’ for the training process	66
5.3	determination of the mean values for the second unweighting in the example of the training process	68
5.4	determination of the maxima for the second unweighting in the example of the training process	68
5.5	new distribution of weights after the machine-learning based unweighting procedure	69
5.6	example Feynman diagrams of the process group $Z + \text{jets}$	71
5.7	example Feynman diagrams of the process group $W + \text{jets}$	72
B.1	histograms of SHERPA’s random numbers, weighted with the weight w	77
B.2	histograms of momenta, weighted with the weight w	78
B.3	frequency distribution of initial channels	79
B.4	frequency distribution of final channels	79

List of Tables

2.1 Properties of the Standard Model bosons.	7
2.2 Properties of the Standard Model leptons.	8
2.3 Properties of the Standard Model quarks.	8
4.1 comparison of some of the most popular algorithms implemented in scikit-learn . .	35
4.2 transformations to the new coordinate system	38
4.3 comparison of the errors for different methods of prediction	39
4.4 comparison of different configurations of the random forest	41
4.5 network configuration as obtained by AutoWeka	45
4.6 errors obtained by FANN with the parameters suggested by AutoWeka	46
4.7 errors obtained by FANN using different activation functions	47
4.8 comparison of errors for different methods of feature scaling	48
5.1 evaluation of the training process	66
5.2 evaluation of some processes of the process group ‘ $Z + \text{jets}$ ’	72
5.3 evaluation of some processes of the process group ‘ $W + \text{jets}$ ’	72

Bibliography

- [1] T. Gleisberg, S. Höche, F. Krauss, M. Schönherr, S. Schumann, F. Siegert and J. Winter, *Event generation with SHERPA 1.1*, Journal of High Energy Physics **2** (2009), 7, [[arXiv:0811.4622 \[hep-ph\]](#)].
- [2] D. J. Griffiths, *Introduction to elementary particles*, ed. 2., rev. ed., Wiley-VCH, Wiley-VCH, 2008.
- [3] W. N. Cottingham and D. A. Greenwood, *An introduction to the standard model of particle physics*, ed. 1. publ., Cambridge Univ. Press, Cambridge Univ. Press, 1998.
- [4] J. Beringer et al., Particle Data Group collaboration, *Review of Particle Physics*, Phys. Rev. D **86** (2012), 010001.
- [5] ATLAS and CMS Collaborations, *Combined Measurement of the Higgs Boson Mass in pp Collisions at $\sqrt{s} = 7$ and 8 TeV with the ATLAS and CMS Experiments*, Physical Review Letters **114** (2015), 191803, [[1503.07589v1](#)].
- [6] A. R. Steere, *A Timeline of Particle Accelerators*, Master's thesis, 2005.
- [7] C. O'Lunaigh, *First images of collisions at 13 TeV*, <http://home.web.cern.ch/about/updates/2015/05/first-images-collisions-13-tev>, July 2015.
- [8] *What is Atlas?*, http://atlas.ch/what_is_atlas.html, July 2015.
- [9] *ATLAS Photos*, <http://www.atlas.ch/photos/lhc.html>, July 2015.
- [10] *What is Atlas?*, <http://www.atlas.ch/photos/full-detector-cgi.html>, July 2015.
- [11] L. H. Ryder, *Quantum field theory*, ed. 2. ed., repr., Cambridge University Press, Cambridge University Press, 2003.
- [12] C. Anastasiou, C. Duhr, F. Dulat, F. Herzog and B. Mistlberger, *Higgs Boson Gluon-Fusion Production in QCD at Three Loops*, Phys. Rev. Lett. **114** (2015), 212001.
- [13] G. Zanderighi, *Standard Model theory for collider processes*, 2015.
- [14] <https://sherpa.heforge.org>.

-
- [15] *Monte Carlo event generators*, <https://sherpa.hepforge.org/trac/wiki/MonteCarloGenerators>, July 2015.
- [16] S. Catani, F. Krauss, R. Kuhn and B. R. Webber, *QCD matrix elements + parton showers*, JHEP **11** (2001), 063, [[arXiv:hep-ph/0109231](#) [hep-ph]].
- [17] S. Höche, F. Krauss, S. Schumann and F. Siegert, *QCD matrix elements and truncated showers*, Journal of High Energy Physics **5** (2009), 53, [[arXiv:0903.1219](#) [hep-ph]].
- [18] S. Hoeche, F. Krauss, M. Schonherr and F. Siegert, *QCD matrix elements + parton showers: The NLO case*, JHEP **04** (2013), 027, [[arXiv:1207.5030](#) [hep-ph]].
- [19] <http://sherpa.hepforge.org/event.png>, July 2015.
- [20] N. Metropolis, *The beginning of the Monte Carlo method*, Los Alamos Science **15** (1987), 125–130.
- [21] S. Weinzierl, *Introduction to Monte Carlo methods*, [hep-ph/0006269](#).
- [22] R. Kleiss and R. Pittau, *Weight optimization in multichannel Monte Carlo*, Comput. Phys. Commun. **83** (1994), 141–146, [[arXiv:hep-ph/9405257](#) [hep-ph]].
- [23] J. von Neumann, *Various Techniques Used in Connection with Random Digits*, J. Res. Nat. Bur. Stand. **12** (1951), 36–38.
- [24] R. Kleiss, W. J. Stirling and S. D. Ellis, *A new Monte Carlo treatment of multiparticle phase space at high energies*, Computer Physics Communications **40** (1986), no. 2, 359–373.
- [25] E. Alpaydin, *Maschinelles Lernen*, Oldenbourg, 2008.
- [26] S.-B. Cho and H.-H. Won, *Machine Learning in DNA Microarray Analysis for Cancer Classification*, Proceedings of the First Asia-Pacific Bioinformatics Conference on Bioinformatics 2003 - Volume 19 (Darlinghurst, Australia, Australia), APBC '03, Australian Computer Society, Inc., 2003, pp. 189–198.
- [27] R. Caruana and A. Niculescu-Mizil, *An Empirical Comparison of Supervised Learning Algorithms*, Proceedings of the 23rd International Conference on Machine Learning, ICML '06.
- [28] E. Fix and J. J. L. Hodges, *Discriminatory analysis, non-parametric discrimination*, Tech. report, USAF School of Aviation Medicine, Randolph Field, Tex., 1951.
- [29] N. Bhatia and Vandana, *Survey of Nearest Neighbor Techniques*, CoRR **abs/1007.0085** (2010), .

- [30] L. Breiman, J. Friedman, C. J. Stone and R. A. Olshen, *Classification and regression trees*, CRC press, 1984.
- [31] I. H. Witten, E. Frank and M. A. Hall, *Data Mining : Practical Machine Learning Tools and Techniques*, Elsevier Science, 2011.
- [32] A. J. Smola and B. Schoelkopf, *A Tutorial on Support Vector Regression*, Tech. report , STATISTICS AND COMPUTING, 2003.
- [33] A. Bordes, S. Ertekin, J. Weston and L. Bottou, *Fast Kernel Classifiers with Online and Active Learning*, J. Mach. Learn. Res. **6** (2005), 1579–1619.
- [34] F. Rosenblatt, *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*, Psychological Review **65** (1958), no. 6, 386–408.
- [35] D. E. Rumelhart, G. E. Hinton and R. J. Wilson, *Learning representations by back-propagating errors*, Nature **323** (1986), 533–536.
- [36] *Einführung in Neuronale Netze, Backpropagation Learning*, <http://cs.uni-muenster.de/Professoren/Lippe/lehre/skripte/wwwnnscrip/modifikationen1.html>, July 2015.
- [37] B. Lantz, *Machine Learning with R*, Packt Publishing, 2013.
- [38] L. Breiman, *Bagging predictors*, Machine learning **24** (1996), no. 2, 123–140.
- [39] L. Breiman, *Random Forests*, Mach. Learn. **45** (2001), no. 1, 5–32.
- [40] Y. Freund and R. E. Schapire, *A Short Introduction to Boosting*, In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1999, pp. 1401–1406.
- [41] *scikit-learn*, <http://scikit-learn.org/>.
- [42] *Auto-WEKA*, <http://www.cs.ubc.ca/labs/beta/Projects/autoweka/>.
- [43] C. Thornton, F. Hutter, H. H. Hoos and K. Leyton-Brown, *Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms*, Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (New York, NY, USA), KDD '13, ACM, 2013, pp. 847–855.
- [44] *FANN - Fast Artificial Neural Network Library*, <http://leenissen.dk/fann/wp/>.
- [45] R. K. Ellis, J. Stirling and B. R. Webber, *QCD and collider physics*, ed. 1. paperback ed., Cambridge Univ. Press, Cambridge Univ. Press, 2003.

-
- [46] K. P. Murphy, *Machine learning / a probabilistic perspective*, MIT Press, MIT Press, 2012.
 - [47] J. Bergstra and Y. Bengio, *Random Search for Hyper-Parameter Optimization*, Journal of Machine Learning Research **13** (2012), 281–305.
 - [48] *Training Algorithms*, http://leenissen.dk/fann/fann_1_2_0/r1996.html, July 2015.
 - [49] N. K. Treadgold and T. D. Gedeon, *Simulated annealing and weight decay in adaptive learning: The SARPROP algorithm*, IEEE Transactions on Neural Networks **9** (1998), no. 4, 662–668.
 - [50] *FANN Datatypes*, http://leenissen.dk/fann/html/files/fann_data-h.html, July 2015.
 - [51] *Max_Epsilon*, https://sherpa.hepforge.org/doc/SHERPA-MC-2.1.1.html#Max_005fEpsilon, July 2015.
 - [52] F. Krauss, R. Kuhn and G. Soff, *AMEGIC++ 1.0, A Matrix Element Generator In C++*, Journal of High Energy Physics **2** (2002), 44, [[hep-ph/0109036](#)].
 - [53] T. Gleisberg and S. Höche, *Comix, a new matrix element generator*, Journal of High Energy Physics **12** (2008), 39, [[arXiv:0808.3674 \[hep-ph\]](#)].

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich im Laufe meines Studiums und insbesondere während dieser Masterarbeit auf verschiedenste Art und Weise unterstützt haben.

Mein besonderer Dank gilt meinem Betreuer Dr. Frank Siegert. Ihm gelang es während des Hauptseminars, mich für das Thema ‘Monte-Carlo’ zu begeistern und er gab mir die Gelegenheit an diesem spannenden Thema zu arbeiten. Auch hatte er bei allen Problemen stets ein offenes Ohr für mich.

Weiterhin möchte ich mich bei Dr. Christian Gütschow, Stefanie Todt, Felix Socher und Carsten Bittrich für das Korrekturlesen dieser Arbeit und die vielen wertvollen Tipps zum Schreiben bedanken, ebenso bei Dr. Christian Gumpert für die anregenden Diskussionen über Statistik.

Danke sagen möchte ich weiterhin unserer ‘guten Fee’ Kristin Walter sowie unseren Systemadministratoren Robert Mantey und Dr. Wolfgang Mader, alle waren bei Problemen immer zur Stelle und haben meine Arbeit sehr erleichtert.

Ein herzliches Dankeschön auch an alle, die mir geholfen haben am Tischkicker den Kopf freizukriegen, insbesondere an meinen Turnierpartner Lukas Schröder.

Außerdem möchte ich mich bei allen bedanken, die mich im Laufe des Studiums sowohl in der SLUB als auch in der Bierstube unterstützt haben, danke an Melanie, Flo, Florian, Martin, Jonas, Philipp und all die anderen!

Besonderer Dank gilt auch meinen Eltern – sie haben mich von klein auf immer in allem unterstützt und sind nach wie vor Vorbilder für mich.

Ganz besonders bedanken möchte ich mich schlussendlich bei Corinna, die mühsam versuchte mir die englischsprachige Kommasetzung beizubringen, aber vor allem immer an mich glaubte und mir unzählige schöne Momente außerhalb der Physik bescherte. Danke!

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit im Rahmen der Betreuung am Institut für Kern- und Teilchenphysik ohne unzulässige Hilfe Dritter verfasst und alle Quellen als solche gekennzeichnet habe. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Johannes Krause
Dresden, Oktober 2015