Justus-Liebig-Universität Gießen

II. Physikalisches Institut

Classification of low momentum π^0 Mesons at the Belle II experiment by use of deep learning algorithms

Klassifizierung von π^0 Mesonen mit kleinem Impuls am Belle II Experiment unter Nutzung von Deep Learning Algorithmen

Bachelorthesis

Author:

LUKAS HOLLER

Supervisor:

APL. PROF. JENS SÖREN LANGE

Gießen, September 30, 2020

JUSTUS-LIEBIG-UNIVERSITÄT GIESSEN

Experimental Physics II

Abstract

This study aims at identifying low momentum π^0 mesons with machine learning algorithms at the Belle II experiment.

To find π^0 mesons, reconstruction software tries to match photon pairs created by the $\pi^0 \rightarrow \gamma\gamma$ decay. Conventional methods can identify π^0 mesons with a momentum greater than 230 MeV/c reasonably well. But in the lower momentum range, the background exceeds the signal by far, and the identification becomes more difficult. This study uses random forests, feed-forward neural networks, and siamese neural networks to identify these low momentum pions. Therefore, features from possible photon pairs, reconstructed by the detector software, are selected and passed to the classifiers. The studied low momentum π^0's are created by the $D^{*^0} \rightarrow D^0\,\pi^0$ decay, whereby the D^{*^0} mesons at Belle II are, for example, produced in the $e^+e^- \rightarrow \Upsilon(4S) \rightarrow B^+B^-$ and $B^+ \rightarrow K^+ D^{*^0} \bar{D}^0$ decay [1].

Especially the feed-forward neural networks and siamese neural networks accomplished this task fairly well with an area under the curve (AUC) for the Precision-Recall curve of 0.882. Random forests, with an AUC of 0.812, were primarily used to identify the input feature importance. Despite dealing with pairwise input, causing potential symmetry and information loss problems when using feed-forward neural networks [2], the performances of the tested feed-forward neural networks and siamese neural networks are identical. The found results suggest a pattern underlying the data, which causes an upper limit in performance with the used methods.

Contents

1 Introduction

Experimental results indicate that mesons can build molecules. Above 3.75 GeV/c^2, particle states, which defy the theoretical expectations and therefore are called exotic, have been observed. One theory is that these states are meson molecules [3]. Combination of D and D^* mesons are good candidates for these molecules, due to their combined mass being just above this threshold of 3.75 GeV/c^2 [1]. The D-D^* pairing called X(3872) was found in the year 2003 [4]. Currently, researches try to observe the D^*-D^* molecule called X(4014) at, for example, the Belle II experiment. Because D^* mesons mainly decay into pairs of D^0 and π^0 mesons, the good reconstruction of π^0's is critical for identifying D^{0^*}. Most π^0's originating in the decay of such D^{0^*} mesons have a momentum of less than 230 MeV/c [1]. The π^0 mesons decay into a pair of photons with a low momentum as well. The problem is that these photons, and therefor the π^0's, are harder to detect because the background is significantly higher in the low momentum region.

This thesis aims at identifying these low momentum π^0 mesons with machine learning algorithms like artificial neural networks. The idea of artificial neural networks [5, 6] is to adopt the behavior of human brains. Artificial Neural networks (ANN) are systems of nodes (perceptrons) that are connected. Typically a node in an ANN receives the values of connected nodes multiplied by individual weights corresponding to the connections, computes the sum over all inputs of the node, and might apply a function to scale the sum. The network is trained by altering the weights between nodes to get closer to the desired output. A commonly used architecture for ANN's is the feed-forward structure. Feed-forward artificial neural networks consist of several layers of nodes whereby each node of every layer is connected to each node of the following layer. An ANN can be understood as a function with many input parameters.

The strength of neural networks is that tasks do not need to be solved analytically, which, for some cases, is almost impossible due to the complexity of a problem (for example software for self-driving cars). Furthermore, neural networks can learn to recognize non-trivial patterns in datasets. In the '90s neural networks started to become popular, but the breakthrough of neural networks started around the year 2010 with the increase in computing power [7]. Today neural networks are not limited in their size and play an important role in nearly all aspects of our life. The use cases for neural networks range from speech and image recognition to suggestions on what we may like to buy or watch.

The strength of machine learning algorithms to find patterns in data during a learning process should be used in this thesis to create classifiers that can identify π^0 mesons with a momentum below 230 MeV/c reliably.

The first chapter after the introduction describes the theoretical background, such as the standard model of particle physics, the Belle II experiment, different machine learning approaches, and the evaluation of classification algorithms. In the following chapter, the process of creating the data with a Monte-Carlo simulation, the preparation of the data, and the evaluation and implementation of the different machine learning algorithms is described. Chapter 4 discusses and compares the results and concludes the thesis.

2 Theory

2.1 Standard Model of particle physics

The standard model [1, 8] of particle physics is currently the best theory to describe the fundamental particles and their interactions by the electromagnetic, weak, and strong force. In the standard model, these forces are mediated by quanta called carrier particles. Thereby the fundamental particles are separated into matter and carrier particles, and the Higgs boson, giving the other particles their mass. The standard model explains nearly all experimental results and correctly predicted many phenomenons, like the Higgs boson discovered in 2012.

2.1.1 Matter particles

The fundamental matter particles [1, 8] are divided into different groups based on their properties. First, the particles are separated into elementary fermions with a half-integer spin and elementary bosons with an integer spin. The fermions, which have a spin of $\frac{1}{2}$, are further divided into particles with a color charge called quarks and into leptons without a color charge. The quarks consist of the up, charm, and top quark with a charge of $\frac{2}{3}$ and down, strange, and bottom quark with a charge of $-\frac{1}{3}$. The electron, muon, and tau are the three leptons with a charge of -1 and the electron neutrino, the muon neutrino, and the tau neutrino are the uncharged leptons with very little mass.
The quarks and leptons are grouped into three generations based on their mass. The first generation consists of the lightest elementary particles of each type, being the up, down quark, the electron, and electron neutrino. These light particles are the building block for all stable matter in the universe today. The heavier ones in the second and third-generation (compare table 2.1) are, due to their mass, less-stable, and decay quickly.

Despite the mass, spin, and charge the particles can be identified by several other properties called flavor quantum numbers. These properties, including the spin and the charge, combined give each particle a flavor. For both leptons and quarks, this includes the weak isospin T_3. The electron, muon, and tau have a weak isospin of $\frac{1}{2}$ like the up, charm, and top quarks. The remainings leptons and quarks have a weak isospin of $-\frac{1}{2}$.

Table 2.1: The fundamental particles of the standard model (translated from [8]).

Elementary fermions					Elementary bosons				
	3 generations				3 interactions (gauge bosons)				higgs-boson
	light	medium	heavy		strong	em.	weak		
6 leptons	electron-neutrino ν_e	muon-neutrino ν_μ	tau-neutrino ν_τ				bosons W^+ W^- Z^0		H
	electron e^-	muon μ^-	tau τ^-			photon γ			
6 quarks	*down d* *up u*	*strange s* *charme c*	*bottom b* *top t*		gluon g				
for fermions: additionally the same table for antiparticles					for bosons: antiparticles already included				

Leptons are distinguished by quantum numbers for each generation. The electron and the corresponding neutrino have an electronic lepton number L_e of $+1$, the second generation leptons a muonic lepton number L_μ of $+1$, and the third generation leptons a tauonic lepton number L_τ of $+1$.

Quarks carry a baryon number $B = \frac{1}{3}$ instead of a lepton number and differ in their quark flavor quantum numbers. The quarks quantum numbers include the third component of the isospin I_3, the strangeness, the charm, the bottomness, and the topness. Despite the isospin, the other quantum numbers are defined by the number of anti strange, charm, anti bottom, or top quarks. For all the quarks, the quark flavor quantum numbers are 0 despite one entry. Up quarks have a I_3 of $\frac{1}{2}$, down quarks of $-\frac{1}{2}$, strange quarks a strangeness of -1, charm quarks a charm of 1, bottom quarks a bottomness of -1, and

top quarks a topness of 1. Further quarks carry a color-charge of red, green, or blue. Under normal conditions, these colors cannot be isolated and observed. The quarks have to form hadrons, which are color neutral. This phenomenon is called confinement. A hadron is color neutral when, for example, a color and its anti-color bond together in mesons or when a red, a green, and a blue quark (or three antiparticles with antired, antigreen, and antiblue) form a baryon. When dealing with antiparticles the signs of all flavor quantum numbers are swapped.

2.1.2 Carrier particles

The electromagnetic, weak, and strong force is mediated by carrier particles. These carrier particles [1, 8], also called force particles, are gauge bosons with a spin of 1. When particles interact with a certain force, the corresponding carrier particles are exchanged. The electromagnetic force is carried out by photons (γ). Because photons are massless, the electromagnetic force is effective over a long-range.

The strong force is mediated by eight gluons and is responsible for binding quarks and thereby forming hadrons. These gluons interact between color-charged particles, each consisting of a combination of color-charge and anticolor-charge. Because gluons themselves carry a color-charge, they interact among themselves. Due to this the strong force is short-ranged.

The weak force is mediated by the W^+, W^-, and the Z boson and interacts with all particles. Because the $W^\pm$ and Z bosons are relatively heavy (about 80 times the mass of a proton), the weak interaction is very short-ranged. The Z boson is its own antiparticle, and thus all charges and flavor quantum numbers are zero. Therefore the Z boson can only change the spin and momentum of particles. $W^\pm$ are responsible for the decay and conversion of particles, whereby the weak force is the only force that can interact with neutrinos. A typical example is the β^--decay when a down quark decays into an up quark emitting a W^- that further decays into an electron and an electron neutrino.

2.1.3 Limits of the standard model

The standard model [8] is currently the best theory to describe all known particles and their interactions. All found particles till today are consistent with this model, and many phenomenons can be explained with it. Still, the standard model is not a final theory and leads to some fundamental physical questions regarding the model: Why do these particles have their particular mass, and what causes the large difference between the generations? Is there a carrier particle for the gravitational force? What is dark matter? Further, the standard model has 25 fundamental parameters, like several coupling parameters for the interactions and the masses of the leptons and quarks. This is not necessarily a problem but suggests that there could be undiscovered correlations like a unification of the fundamental forces implied in some experimental results.

2.2 The Belle II experiment

2.2.1 Overview

The Belle II experiment and the SuperKEKB accelerator (Fig. 2.1), at which the experiment is performed, are upgrades of the Belle experiment and the KEKB accelerator located in Japan [9, 10, 11, 12].

In the Belle experiment, the CP-violation (violation of CP-symmetry) in the neutral B meson system was successfully measured. CP symmetry states that when a particle is swapped with its antiparticle and the coordinates are inverted, the physics should be the same. CP-violation is one reason why the universe today is dominated by matter over antimatter. Otherwise, the amount of matter and antimatter would be equal because antimatter annihilates in contact with matter, and the Big Bang created equal amounts of both. But the found CP-violation in the neutral B meson system is not enough to explain the whole asymmetry. The Belle II experiment tries to study the related phenomena further and tries to find physics beyond the Standard Model of particle physics (often referred to as "new physics").

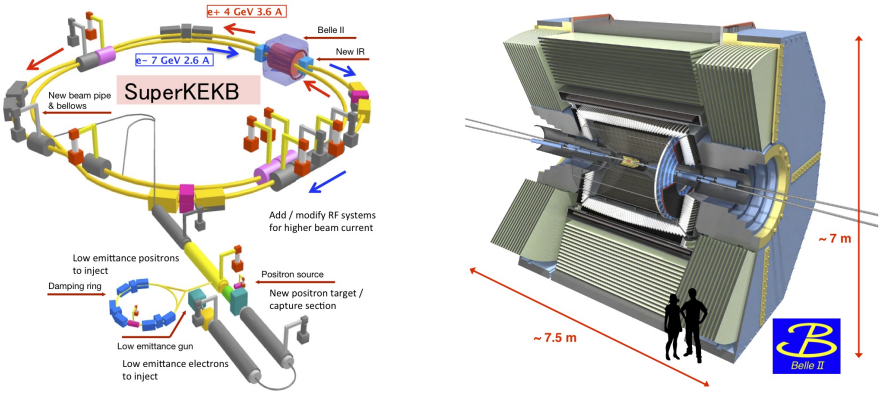


Figure 2.1: The SuperKEKB accelerator (left) and an inside look into the Belle II detector (right) [12].

The SuperKEKB accelerator is an asymmetric electron-positron collider. In the low-energy ring (LER) positrons with an energy of 4 GeV are stored and in the high-energy

ring (HER) electrons with 7 GeV are stored (Fig. 2.1 (left)). The two beams collide with a center of mass energy of $\sqrt{s} = 10.58$ GeV. This energy is in the region of the $\Upsilon(4S)$ resonance, which then decays mostly into $B^0\overline{B}^0$ and B^+B^- pairs. Due to this, the SuperKEKB accelerator is a B-factory.

Another important feature of the SuperKEKB accelerator is its planned very high luminosity of $8 \cdot 10^{35}\text{cm}^{-2}\text{s}^{-1}$. This luminosity is needed to very precisely measure rare decays. On the other hand, the high luminosity creates more data and thus more background. The Belle II detector system and software have to cope with this higher background. In particular the detectors closer to the interaction point, such as the pixel detectors 14 mm away from the center of the barrel, are affected.

2.2.2 Detectors

The Belle II detector setup [9, 10, 11, 13] is depicted in Figure 2.2. Closest to the interaction point is the Vertex detector (VXD), followed by the Central Drift Chamber (CDC). The Vertex detector provides information about decay vertices and information about low momentum tracks. The Central Drift Chamber is used to reconstruct momenta and charges of tracks. These two detectors are surrounded by a particle identification system (PID) and the electromagnetic calorimeter (ECL). Both detectors are placed in the barrel of the Belle II detector and as well in the forward endcap in case of the PID or in both endcaps like the ECL. As the name states, the particle identification system is designed to identify charged particles. With the electromagnetic calorimeter electrons and photons can be detected.

These detectors are placed in a homogeneous 1.5 T magnetic field, which is provided by a superconducting coil placed in the barrel after the ECL. The above-mentioned detectors are further surrounded by a K_L^0 and muon detector (KLM). The KLM detector can distinguish between muons and hadrons and can identify K_L^0 in combination with the previous detectors.

In the used coordinate system, the z-axis is pointing in the direction of the electron beam (fig. 2.2 from left to right), the y-axis points upwards and the x-axis points away from the center of the accelerator.

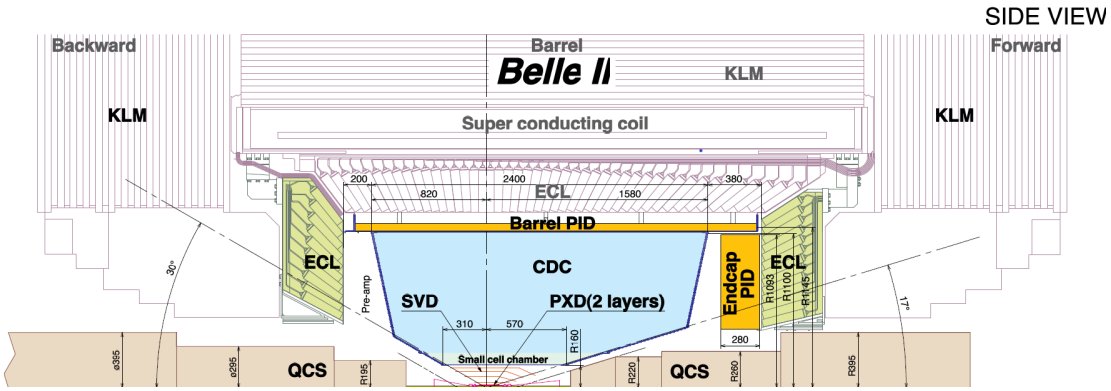


Figure 2.2: Side view on the top half of the Belle II detector [9].

2.2.2.1 Vertex detector (VXD)

The Belle II vertex detector is made up of two different detectors, a silicon Pixel Detector (PXD) and a Silicon Vertex Detector (SVD). Combined these two detectors have six layers. The two innermost layers are the silicon Pixel Detector, and the four outer layers are the Silicon Vertex Detector. Due to the high luminosity of SuperKEKB, and the resulting higher background, the detectors, especially the innermost, need to be very precise. Therefor for the two inner layers, with radii at 14 mm and 22 mm, pixel sensors are used, with a total of eight million pixels. The four other layers with radii ranging from 38 mm to 140 mm are far enough away from the interaction point that strip detectors can be used.

The vertex detector is used to reconstruct the trajectory of charged particles.

2.2.2.2 Central Drift Chamber (CDC)

The Central Drift Chamber (CDC) is a large volume drift chamber filled with 50% helium, 50% ethane gas, containing sense and field wires. It surrounds the vertex detector and has an inner radius of 160 mm and an outer radius of 1130 mm.
When charged particles pass through the detector, they create electron-ion pairs. Due to an applied electric field between the field wires and the sense wires, the electrons are accelerated and create a shower of electrons by electron ionization. This shower can

be measured with the sense wires. This way it is possible to reconstruct the tracks and momenta of charged particles. Further, this provides some information for particle identification by measuring the energy loss of particles. This is especially important for low momentum particles, which may not reach detectors designed for particle identification more outwards.

The CDC used in Belle II contains 14336 sense wires arranged in 56 layers. To perform a 3D reconstruction of the helix tracks, the layers are grouped into nine superlayers, containing six layers each (eight for the innermost). The superlayers orientation alternates between being aligned parallel to the beampipe/z-axis (axial layers) and at a non-zero angle to the beampipe (stereo layers).

To cope with the high background, the layer density is higher closer to the interaction point.

2.2.2.3 Particle Identification (PID)

The particle identification system at Belle II has a Time-Of-Propagation (TOP) detector, which is located in the barrel region of the detector and an Aerogel Ring-Imaging Cherenkov (ARICH) detector located in the forward endcap.

Both detectors use the Cherenkov effect. When a charged particle passes through a medium with a higher speed than the speed of light in the given medium, it produces light, Cherenkov light. The angle Θ between the particles trajectory and the emitted light is given by $\cos\Theta = (n\beta)^{-1}$ with the ratio between the particle speed and the speed of light (in vacuum) $\beta = v_p/c$ and the refraction index of the medium n. The created light cone can be measured and be used for particle identification.

The Time-Of-Propagation detector is placed around the Central-Drift-Chamber and consists of 16 quartz modules with a thickness of 20 mm. When charged particles pass through the detector, they produce Cherenkov photons with different angles Θ. The photons are reflected in the quartz radiator and measured with a micro-channel plate (MCP). The MCP only provides two spatial coordinates, but with very precise timing (30 ps), it is possible to obtain the 3-dimensional information. The conceptual overview is shown in fig. 2.3.

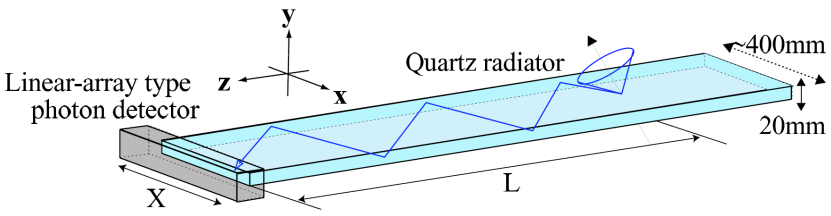


Figure 2.3: Conceptual overview of TOP counter [9].

For the forward endcap a Aerogel Ring-Imaging Cherenkov (ARICH) detector is used. In an aerogel radiator, consisting of two different aerogel layers ($n_1 = 1.046$ upstream and $n_2 = 1.056$ downstream), Cherenkov photons are created. The light cones are detected with a photon detector (MCP) (fig. 2.4). The used design provides a good separation of kaons and pions, and pions, muons, and electrons below 1 GeV/c.

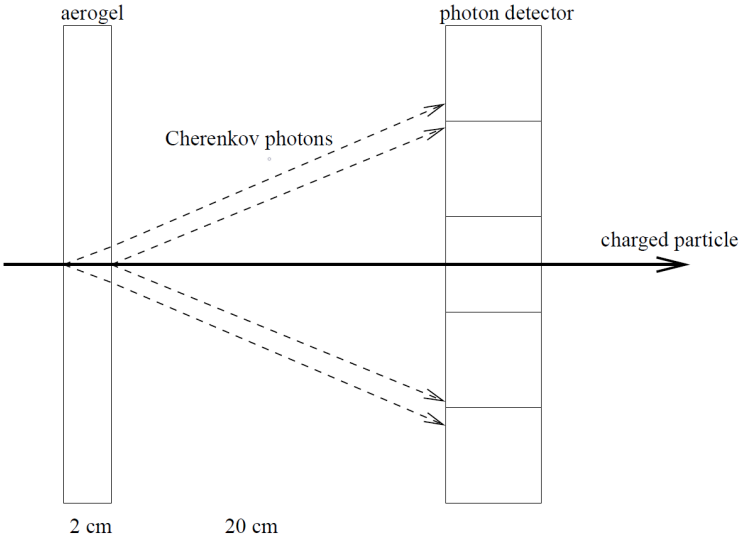


Figure 2.4: Proximity focusing ARICH - principle [9].
Note: The used aerogel radiator for Belle II is, in contrast to the figure, two layers with different refractive indices are used.

2.2.2.4 Electromagnetic calorimeter (ECL)

The main purpose of the electromagnetic calorimeter (ECL) is to detect the energy and angular coordinates of photons and to identify electrons. This is especially important for the Belle II experiment because, during the decay of B-Mesons, about one-third of the created particles are neutral. In order to reconstruct the B mesons, it is very important to precisely detect photons in a wide energy range.

The electromagnetic calorimeter consists of a 3 m long barrel structure surrounding the Time-Of-Propagation detector and two endcaps with a total of 8736 CsI(Tl) (caesium iodide doped with thallium) crystals. This way, the electromagnetic calorimeter covers about 90% of the solid angle in the center-of-mass system.

The effect used in the electromagnetic calorimeter is that when electrons and photons travel through the scintillation crystals, they produce very similar electromagnetic cascades. Photons produce secondary electrons (pair production, photoelectric effect, ...). The electrons, for example, free other electrons or radiate photons when interacting with the crystal material. This way, the energy of the particle is deposited in the calorimeter. With a photomultiplier at the end of each crystal, the amount of photons from a particle is measured. The number can be converted to the absorbed particle energy. Electrons and photons can be distinguished by checking if a charged track from the previous detectors can be matched to the events measured in the ECL.

2.2.2.5 K_L^0 and muon detector (KLM)

The K_L^0 and muon detector (KLM) consists of alternating layers of 4.7 cm thick iron plates and detector elements. In the barrel, the KLM has 15 detector layers and 14 iron plates, and in the endcaps, the KLM consists of 14 detector layers and 14 iron plates. Due to the high background, scintillators are used for the two innermost detector plates of the barrel and all layers of the endcaps. For the other detector layers of the barrel, two resistive plate chambers (RPCs) per layer are used. These RPCs consist of two glass sheets (high voltage electrodes) with a thin gas volume in between them. Charged particles passing through the RPC ionize the gas molecules and create a spark that can be measured.

Due to the higher penetration power of muons compared to hadrons, muons can be distinguished from hadrons. The combined 4.7 interaction lengths of the KLM and the ECL are enough that the hadrons energy gets dissipated, and the hadrons are stopped. Neutral K_L^0 mesons can be identified because they can create clusters in both ECL and KLM, but they cannot produce charged tracks in the detectors in front.

2.2.3 The Belle II Analysis Software Framework (BASF2)

The Belle II Analysis Software Framework BASF2 [10, 13, 14, 15] is a crucial part of the Belle II experiment. It provides the complete software and tools needed for the Belle II experiment. This includes collecting the data from the detector or otherwise generating and simulating data with a Monte Carlo simulation, then reconstruct the particles and their properties from the acquired data and analyze the reconstructed data. The process of how to generate, simulate, and analyze data is explained in more detail in subsection 3.1.

BASF2 is written in C++14, but a Python interface is provided with many Python scripts (*steering files*). This makes BASF2 accessible for more users, and scripting is easier because users can write their own modules in Python. The data is handled with ROOT objects and files.

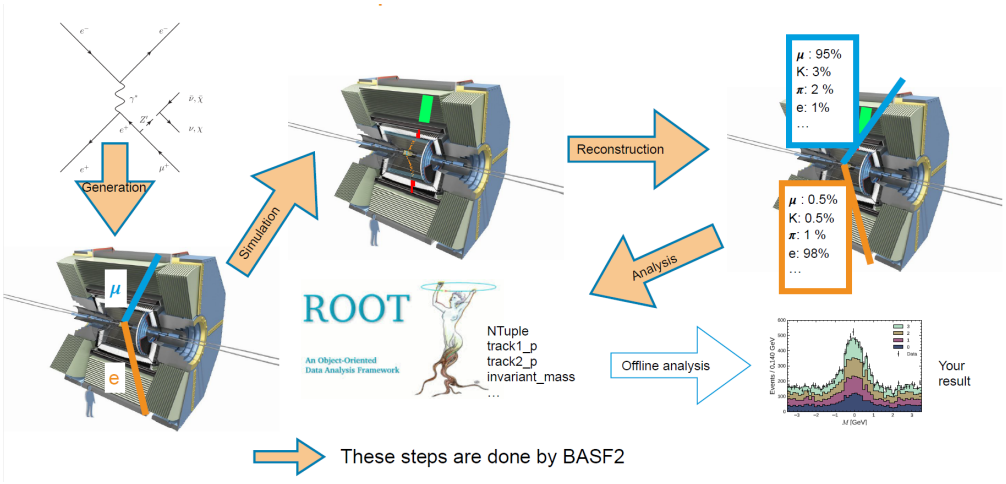


Figure 2.5: Overview of the tasks the BASF2 framework can perform [14].

2.3 Classification methods

Classification methods [16], as the name states, are methods that decide, based on a given object (input), in which category an object most likely belongs or how to label this object. Classification methods apply to a wide variety of tasks, ranging from medical tasks, classify if a person has cancer based on a ct scan, email providers checking if an email is 'spam' or not, or for classifying road signs for autonomous driving. Due to the importance of classification tasks, many different methods are available today. In this thesis Decision Trees, Random Forests, and different kinds of Neural Networks will be discussed and used.

The steps when creating a classifier are to prepare the data, tune the classifier settings, train the classifier, and then measure its performance on some test and validation data. The test data is mostly data close to the training data but still new data for the classifier and is used to measure the performance of the classifier. Validation data is data that is completely new and independent from the training data and should be used because some classifiers tend to overfit data, and to receive less biased performance feedback. Overfitting means that the classifier does not learn to recognize a pattern in the dataset but remembers the tuples from the training set. This can be identified when the performance on the training data is improving, but the performance on the validation data starts to decline. When the performance of the classifier is not satisfying, the process goes back to step one, two, or three.

Programming classifiers can be done in python with the scikit-learn library [17], which provides a wide range of tools for data analysis and machine learning. Other important python libraries are Tensorflow [18] and Keras (Tensorflow API) [19], which provide tools to create neural networks. Keras further provides the option to run on the GPU (NVidia), which results in significantly better run times compared to the CPU. Overall phyton provides many libraries for handling and manipulating large datasets and visualizing data. For this thesis, Jupyter Notebooks [20] have been used as they offer a lot of interactivity and code readability.

A subset of classification tasks is binary classification. Binary classification is used for tasks where only two possible outputs exist (for example, 0/1, true/false, 'spam'/'non-spam', ...). Within this thesis, the objective for the binary classifiers is to decide if a

given pair of photons originate from a low momentum π^0 created by a $D^{0*} \rightarrow D^0\pi^0$ decay or not.

Many classifiers do not directly output 0 or 1, typically the output is a value between 0 and 1. This output can be interpreted as a probability, whereby a value closer to 0 or 1 indicates that a given tuple is more likely a 0 or 1. To perform a prediction based on this continuous value, a threshold is used. All values above the threshold are labeled as 1, and all below, or equal to the threshold, as 0. The standard threshold is 0.5. Changing this threshold can change the performance of a classifier significantly, as shown in figure 2.6 where a higher and lower threshold both performs better than the standard threshold of 0.5.

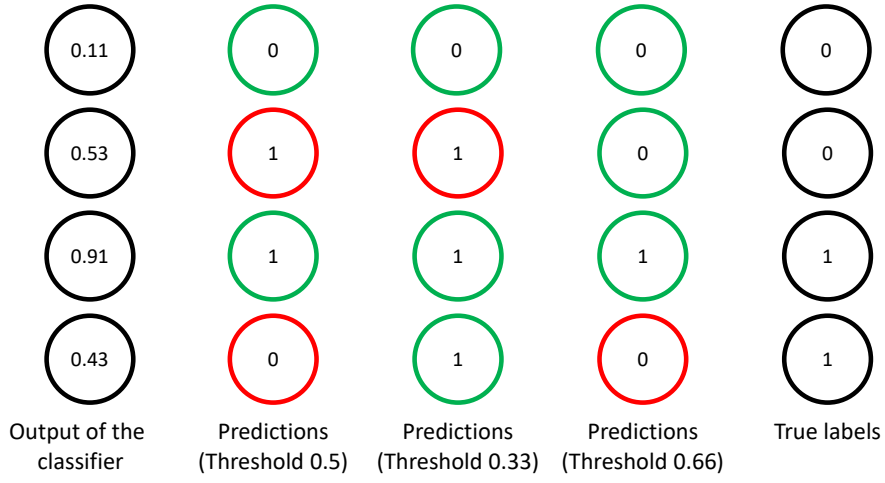


Figure 2.6: Example of how different output scores lead to different prediction in dependency of the chosen threshold and thus to different recall and precision scores.

2.3.1 Measuring performance

When testing a binary classifier (Decision Tree, Random Forest, Neural Network, ...) several parameters are used to measure the performance [21, 22, 23]. The most basic parameter used for all further measurements originates from distinguishing how the data was classified in comparison to the true label. A tuple is categorized as true positive (TP) or true negative (TN) when the prediction of the classifier was correctly positive

or correctly negative. When the classifier wrongly classified a given tuple as positive or negative, the outcome is false positive (FP) or false negative (FN). This categorization can be illustrated as a confusion matrix shown in Table 2.2.

Table 2.2: Example of a confusion matrix.

	Actual class: Positive	Actual class: Negative
Predicted class: Positive	True Positive (TP)	False Positive (FP)
Predicted class: Negative	False Negative (FN)	True Negative (TN)

The most intuitive approach to use these parameters to measure the performance is the accuracy given by the equation:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} = \frac{\textit{\# of correct classifications}}{\textit{\# of classifications}} \tag{2.1}$$

The downside of the accuracy is that it does not give any information about how well a certain label is classified. For example, a test dataset is made up of 90% 'true' tuples and 10% 'false' tuples. Even when a classifier wrongly classifies all tuples as 'true', the overall performance is still 90%. This is not only a problem during the evaluation but also during training. If the training data is heavily unbalanced, the classifier may learn to classify the whole datasets as the over-represented label. This happens because even if a few tuples are wrongly classified, the over-represented label is correctly classified, and due to this, the classifier still performs well regarding the accuracy.

To take this into account the performance can be measured by the precision, the recall, and the F_1-score. All these scores focus on the performance of the positive label because, for this thesis and most classifiers, the attention is on classifying correctly the positive label, not the negative one.

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

$$F_1\ score = \frac{2TP}{2TP + FN + FP} \tag{2.4}$$

Precision is the probability that a positive prediction is truly positive. On the other hand, the probability that an actual positive class is correctly classified as positive is called recall. The F_1 score is a combination of the precision and recall score.
To decide which parameter is most relevant, the purpose of a classifier has to be considered. If a classifier should, for example, detect a tumor on a ct-scan, a high recall is wanted. Because otherwise some tumors, would not be found and cause a serious health risk. Contrarily, precision should be favored when it is important that a positive outcome is truly positive. This may be the case when police investigators use face recognition to find criminals. With a high precision, it can be prevented that the officers arrest the wrong person. In most cases, it is important to find a balance between these parameters. This can be done by using the F_1 score. Another option is the Precision-Recall curve.

For every used prediction threshold, a precision-recall pair exists. By varying the threshold, either precision or recall can be maximized. Hereby, a higher precision normally results in a lower recall and vice versa. For every use-case, a good trade-off between the parameters has to be found.

2.3.2 Precision Recall Curve

A Precision-Recall Curve (PRC) [22, 23] depicts this tradeoff between precision and recall, and provides a visual interpretation of how good a classifier performs independently of the used threshold. In a PCR, the precision recall points for different decision thresholds are plotted and connected. Figure 2.7 depicts the PRCs of a dataset with 33% positive labels with a random, a perfect, and a sample classifier. A random classifier has a constant precision corresponding to the percentage of positive labels independently of the recall. All curves above this line perform better than a random classifier. The best classifier has a precision of 1 for all recall values despite the case where the recall is 1. When the recall is 1, the precision can have all values above the precision of a random

classifier. The performance of a typical classifier is normally between those curves.

Another parameter to measure the performance obtained from the Precision-Recall Curve is the area under the curve (AUC) with a perfect AUC of 1. The strength of the AUC score is that it measures the performance of the classifiers for different thresholds in comparison to the F_1-score. The F_1-score takes the recall and precision into account as well, but only at a given threshold.

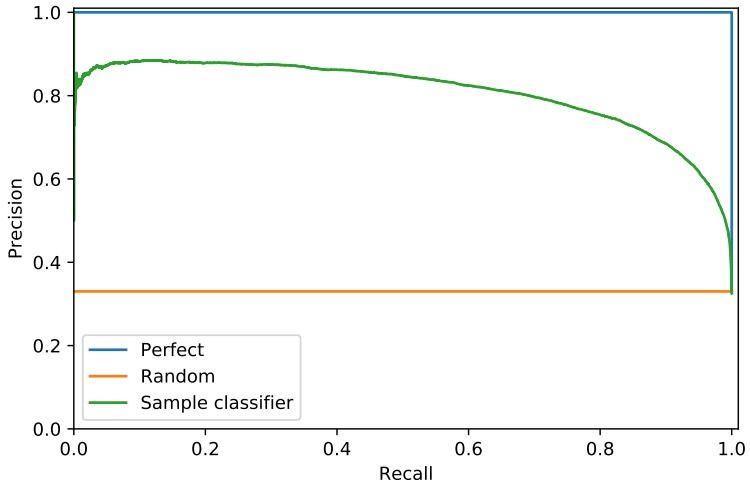


Figure 2.7: The Precision Recall Curve for a random classifier, the perfect classier and a sample classifier. The used dataset contains 33% positive labels.

Compared to the often used ROC plot, the precision-recall curve should be used when the dataset is imbalanced, meaning one label is being overrepresented [22]. In the case of this thesis, as shown later, the negative label outweighs the positive.

2.3.3 Decision trees

Decision trees [24, 25, 26, 27] are a pretty simple approach to classification (and regression) problems. Decision trees classify data by starting at the root node and splitting the dataset. The split is done by a decision function (decision node). The decision function is, for example, if a given feature x is above or below a certain value or in a certain range of values. A decision node is not limited to one feature, it can split the dataset by

asking multiple 'questions' at once. After the root node, this process continues with the internal nodes and ends when a leaf node is reached. The given tuple of the dataset is assigned the label of the leaf node it ends in. Figure 2.8 depicts a simple example of a use-case for decision trees.

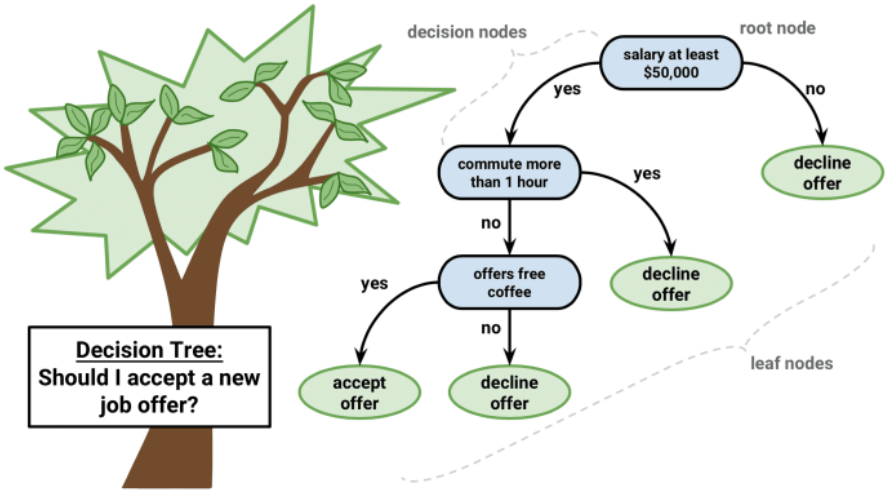


Figure 2.8: Simple example of a decision tree [26].

The main parameters of decision trees are the depth of the tree, the maximum number of leaves, and how the splits are done. The depth of a decision tree is the length of the longest path or, in other words, the number of layers of internal nodes and leaves. Figure 2.8 has, for example, a depth of 3. Another way to prevent the tree from becoming too complex is to set a maximum number of leaves. This may be helpful if a certain classification (extracting a specific label) is easily done by a few decision nodes, but another classification in the same tree needs more steps and would be cut off by a maximum depth.

The actual training of a decision tree is the optimization of the split criteria at each node. Commonly used measures to describe the goodness of a split are the Gini Index and Entropy. The Gini Index is defined by:

$$1 - \sum_i (p_i)^2 \tag{2.5}$$

where p_i is the probability that an object is being classified to a certain class at some node or leaf [27]. The Gini Index (also known as Gini coefficient) measures how pure, regarding the labels, the data is or how well the labels have been separated by an applied split. The optimal score is a Gini Index of 0.

Another function, used to measure how well decision nodes perform, is Entropy:

$$\sum_i (-p_i \log_2(p_i)) \tag{2.6}$$

with an optimum (or max purity) of 0 [27]. To obtain the 'best' split for a given node the weighted Gini index or the weighted Entropy of each branch is subtracted from the Gini index or the Entropy of this particular node. The split with the highest score is considered the 'best' split. In practice, the difference between the two functions is, for the most dataset, insignificantly small.

The strength of decision trees lies in their simplicity [25]. The idea behind them is very simple and the data requires little to no preparation. Decision trees can be visualized very well. This can lead to a better understanding of the dataset and give information about how classes are distinguishable. This includes that decision trees can give information on how important which input variable is (feature importance) for classifying the input [25]. This information, about how capable a given feature is to differentiate the dataset, can, for example, be used to decide which features a neural network is trained on.

A downside of decision trees is that trees cannot handle noise or small changes in the training data very well. These small changes often result in completely different trees with strongly varying performances. Another problem with decision trees is overfitting. Especially large trees tend to overfit the data because nearly every tuple of the training dataset has created a single leaf during the training. This can be reduced by, for example, limiting the max depth of a tree, set a maximum number of nodes, or by setting a minimum number of samples for a node to be split further.

2.3.4 Random forest

A random forest [28, 29] is a collection of uncorrelated decision trees and, therefore, an ensemble method. The classification is done by a majority vote of all trees. The combination of many different trees should boost performance and reduce statistical errors.

The most important task, when creating the trees, is to create uncorrelated trees. This can be obtained in several ways. One option is to provide different training data for each tree. This can be done by using a different subset of the training data for each tree, using feature selection and train each tree only on a subset of all features, or by using linear combinations of the input. Another way is to use random splits when creating trees.

The strength of random forests is that they are not as susceptible to overfit as single decision trees. This is due to the Strong Law of Large numbers reducing the overall error when using uncorrelated trees. The other big strength of random forests is that a large number of relatively weak trees can together form a strong classifier. For example, take three decision trees, which alone are correct 80% of the time. If these trees are combined to a random forest and decide via majority vote, the probability that the ensemble is correct is

$$\mathrm{P}(correct) = 1 - \left(0.2^3 + 3 \cdot 0.2^2\right) = 0.88 \tag{2.7}$$

The accuracy increases with the number of used trees in the ensemble. To guarantee that the ensemble performs better than some single trees, it is important that the trees are uncorrelated. Otherwise, all three trees would classify more or less the same 80% correct, and thus the overall performance would be roughly 80% as well. Furthermore, random forests are capable of giving an even more detailed look into the feature importance.

2.3.5 Neural network

Artificial neural networks or simply neural networks [5, 6, 11] are an adaptation of the neural networks in our brains and the try to mimic their behavior. Our brain consists of neurons that are connected. The electric signals in our brains travel through these

networks, and the individual neurons may block a signal coming from another neuron, transmit it, or even intensify the signal before passing it on to some other neurons. Artificial neural networks are build in a comparable manner. The neurons in the artificial neural network are called nodes and are connected. The decision of how these nodes are arranged is an important task when designing a neural network and influences the performance of a neural network drastically.

A simple approach for a neural network is a feed-forward neural network. In this network, several layers of neurons are connected, and an input is fed forward through the layers (fig. 2.9).

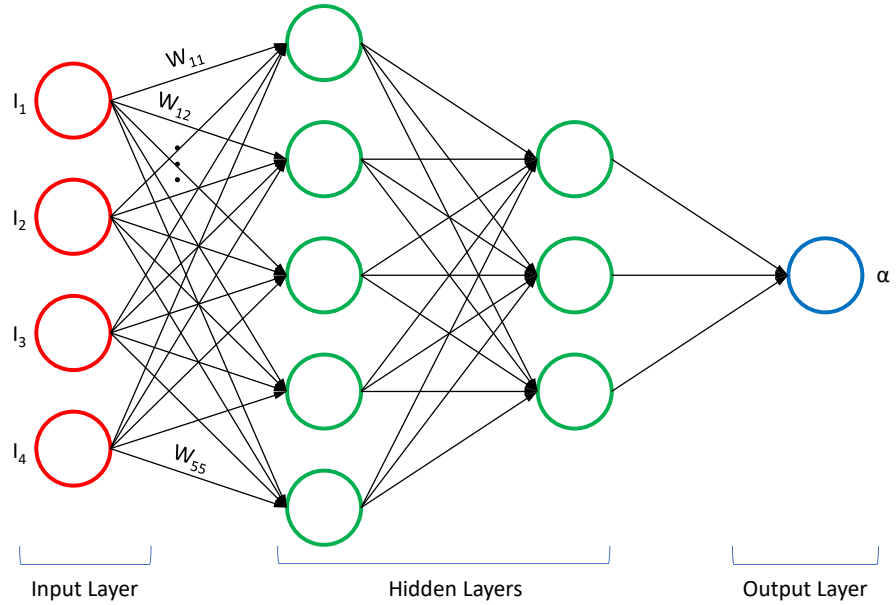


Figure 2.9: Illustration of a feed-forward artificial neural network with four input nodes, two hidden layers with five and three nodes and a single output node.

The first layer is the input layer and the nodes have the values of the input features. Typically all nodes of one layer are connected with all nodes of the next layer. A connection between two nodes is called a weight (W_{11}, W_{12}, ...). When data is passed through the network the value of a node is computed as the sum h over all previous nodes times the weights of the corresponding connection. Additionally, a bias can be added to this sum. Then this sum is typically passed through a function to limit its range. Typical functions

are the tangens hyperbolicus with a range from -1 to 1, the RELU function given by the equation $relu(h) = \max(0, h)$, or the Sigmoid function with output values between 0 and 1. When using, for example, the tangens hyperbolicus the value x_i of the i-th node in a layer is calculated by

$$x_i = \tanh\left(b_i + \sum_j \omega_{i,j}\, u_j\right) \tag{2.8}$$

with the bias b_i, the nodes u_j of the previous layer, and the corresponding weights $\omega_{i,j}$ connecting the two layers [6].

The weights are containing the entire information of the neural network. During the training process the target is to find the values with the best performance. This is done by providing the neural network with training data. The data is used to adjust the weights to reduce the deviation between the prediction and the true labels. Methods used to decrease this output error are called optimizer and try to find minimum for the overall output error of a neural network (loss) in the multi-dimensional weight space. The main differences between the optimizers are how fast they alter weights (learning rate), how they change the learning rate over time, and how do they decide what weights to change how. Backpropagation (short for backward propagation) is an algorithm to calculate the gradient for the neural weights in regards to the loss. This is done backwards through the model. Beginning with the last layer going backwards to the first, the weights are adjusted along the gradient with the learning rate defining the magnitude of change. Due to performance reasons, back-propagation is normally not applied after every training sample, mostly this is done after a certain number of samples combined in a so-called batch.

The loss is calculated with a loss function. For binary classification a typical loss function is binary cross-entropy:

$$Loss = \frac{1}{\text{output}\quad\text{size}} \sum_{i=1}^{\text{output}\quad\text{size}} y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \tag{2.9}$$

with the true label y_i and the predicted label $\hat{y}_i$ [30].

When creating a neural network the main challenges are to find the right amount of layers, nodes per layer, and activation functions for these layers. A small number of layers and nodes per layer can be computed faster but may not provide enough degrees of freedom to find the underlying pattern. A neural network with more layers and nodes per layer is significantly slower to compute but may be more capable to solve a given task. Another problem that comes with more degrees of freedom is the risk of overfitting. This can be partially prevented when during the training random weights between nodes are set to 0. This acts similar to adding noise to training data because the network is forced to adapt to slightly change situations and is called dropout.

The strengths of a neural network are that they are capable of finding patterns in data that humans can not recognize. They can handle high-dimensional data and find important features without any prior knowledge. Furthermore, they can solve tasks where it is nearly impossible to write algorithms by hand, for example, to recognize traffic signs in a video.

2.3.6 Siamese neural network

The input of these artificial feed-forward neural network is, in this thesis, a pair of photons each with an individual feature vector. One decision that needs to be made is how to create the input vector based on this pair. The option used in this study is to concatenate the vectors of each photon in a pair. Other options [2] are the Hadamard product where the i-th feature of each photon is multiplied to create the input vector or to use the Cartesian product and create a vector containing each possible product between the two single-photon features.
Hereby symmetry is the main concern. Symmetry in this context means that a pair of photons create the same result regardless of their order of input. When noting the neural network as a function f with the two photons $\overrightarrow{x}_1$ and $\overrightarrow{x}_2$ as an input pair, symmetry can be noted as:

$$f(\overrightarrow{x}_1, \overrightarrow{x}_2) = f(\overrightarrow{x}_2, \overrightarrow{x}_1) \tag{2.10}$$

But regardless of how the vectors are concatenated information is lost or symmetry is not guaranteed for a feed-forward neural network [2]. Due to these problems, a feed-forward neural network mostly performs worse than so-called pairwise neural networks or Siamese neural networks which are designed to guarantee symmetry.

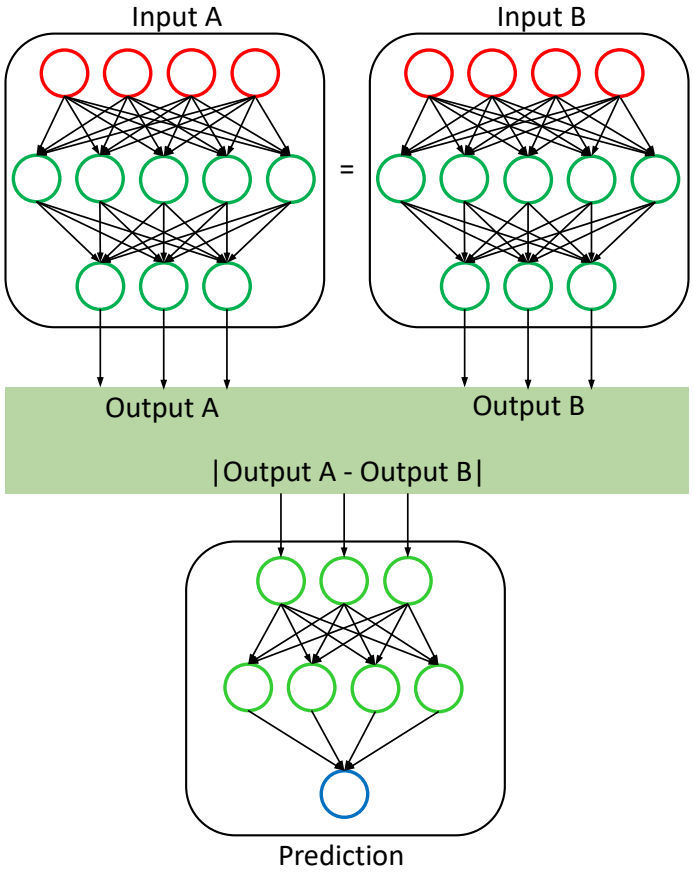


Figure 2.10: Illustration of a siamese neural network.

The structure of a Siamese neural network [2, 31, 32] is depicted in figure 2.10. First, the two features vectors of each photon in a pair are propagated through a feed-forward neural network. To ensure symmetry the two neural networks share their weights, thus being the same network. This ensures the same output for a certain photon regardless if it is propagated through the first or the second network. The two outputs than need to be merged. The function used to merge needs to be symmetric otherwise the order of photons would alter the result. For this thesis, the absolute distance between the

output vectors is used. This output is then once again propagated through a different feed-forward neural network which outputs the prediction.

Siamese neural networks have typically, compared to feed-forward neural networks, a better performance when pairwise input is used. Further, it ensures symmetry and prevents information loss compared to merged input vectors.

3 Analysis

3.1 Data generation

As previously mentioned, the basf2 framework [14, 15, 33, 34] is not only capable of collecting the data from the Belle II experiment, it can simulate data as well. The advantage with this is that the decay channels of the starting $\Upsilon(4S)$ and for the decay products can be modified. This gives the possibility to create exactly the particles which are needed for certain analysis. In the case of this thesis, π^0 mesons with a low momentum are needed. These low momentum pions are created during the $D^{*0} \rightarrow D^0\,\pi^0$ decay [1]. The π^0 mesons decay into a pair of photons that are then detected.

The decay channels for the simulation are given in a decay file. Listing 3.1 is a section of the whole decay file used in this thesis. The complete decay file can be found in the appendix (listing 4.1).

Listing 3.1: Section of the decay.dec file.

```
1 Decay D*0
2 0.65 D0 pi0 PHOTOS VSS;
3 0.35 D0 gamma PHOTOS VSP_PWAVE;
4 Enddecay
5 CDecay anti-D*0
```

First, the particle to decay is specified in line one. In the two following lines, two different decay channels are specified. The first value is the branching ratio, followed by the daughters of the particle specified in line one. The 'PHOTOS' parameter enables final state radiation for this decay [34]. 'VSS' and 'VSP_ PWAVE' specify the decay model. Each model respects specific angular distributions of the particles. 'VSS' stands for a decay of a vector to a pair of scalars and 'VSP_ PWAVE' means that a vector decays to a scalar and a photon. 'Enddecay' tells the interpreter that no more decay

channels for the D^{*^0} will follow. 'CDecay anti-D*0' creates analog the same decay channels with the anti-particles. The decays defined for D^{*^0} in this section can (without the anti-particle decays) be written as:

$$\begin{aligned} 65\% \quad & D^{*^0} \rightarrow D^0\, \pi^0 \\ 35\% \quad & D^{*^0} \rightarrow D^0\, \gamma \end{aligned} \tag{3.1}$$

If a decay of a certain particle is not specified in the decay file, the normal decay channels will be used. This also includes particles that were created via specified decays like the π^0's create in the decay channel of the D^{*^0}.

Table 3.1 contains all decay channel defined in the complete decay file 4.1 in the appendix, despite the antiparticle decays of D^{*^0} and D^0. Besides the decay channel $D^{*^0} \rightarrow D^0\, \pi^0$, to create low momentum π^0, several other π^0 are created in other specified decays with the target to create a few extra π^0 mesons. These extra π^0's should help the classifier to learn to differentiate not only between background and real π^0's but also between the π^0's from the wanted decay and other π^0's.

Table 3.1: Decays defined in the decay.dec file. Note that for the anti-particles of D^{*^0} and D^0 the decay channels are defined analog to the here given channels.

Decay channel	Branching Ratio
$\Upsilon(4S) \rightarrow B^+\, B^-$	100%
$B^+ \rightarrow K^+\, D^{*^0}\, \overline{D}^0$	50%
$B^+ \rightarrow K^+\, D^0\, \overline{D}^{*^0}$	50%
$D^{*^0} \rightarrow D^0\, \pi^0$	65%
$D^{*^0} \rightarrow D^0\, \gamma$	35%
$D^0 \rightarrow K^-\, \pi^+$	13%
$D^0 \rightarrow K^-\, \pi^+\, \pi^0$	47%
$D^0 \rightarrow K^-\, \pi^+\, \pi^0\, \pi^-$	27%
$D^0 \rightarrow K^0_S\, \pi^+\, \pi^-$	13%
$D^0 \rightarrow K^-\, K^+$	13%

The next step is to initialize the simulation with a given number of events (number of $\Upsilon(4S)$ mesons). In the appendix in listing 4.2 the corresponding source code ($steering\ file$) can be found. The generator receives the number of events and the custom decay file that should be used. The outcome is saved as a ROOT file [33] and is passed to the simulation.

The simulation is done with the code in listing 4.3. In this file, the previously created $\Upsilon(4S)$ mesons are first simulated (Monte Carlo), which includes their decays, the radiation they produce, and how all these particles pass through and interact with the detectors. The resulting data from the detectors is then reconstructed the same way data from a real experiment would be. The only difference is that for a Monte Carlo simulation, the experimental results can be matched to the previous simulation. Thereby the information if a measured track or particle corresponds to a simulated particle or background noise is obtained. The result is stored in a ROOT file in the mDST (mini data summary table) format and contains several classes of information [15]. One, for example, to store the particle tracks and one storing the reconstructed clusters in the ECL. When using data from a Monte Carlo simulation another class called MCParticle contains the information about the previously simulated particles like their momentum, decay products, and so on, to later match them with the reconstructed particles. Despite that, the reconstructed data now contains the particles that can be reconstructed directly from detectors, like π^+ mesons and photons. The other particles, like π^0 mesons, can be reconstructed with another $steering\ file$.

The listing 4.4 (appendix) contains the script used to process all 100 simulations at once, with the core part of this script displayed in listing 3.2. In the first line, a path is created in which the modules are arranged [13, 15]. In line three the previously created data is loaded from a given path ('args.inputfile'). As mentioned, this mDST format contains the particles that can be directly reconstructed, like photons. In the fifth line, these photons are loaded in a list called 'all'. With 'matchMCTruth', the reconstructed photons are matched with the corresponding particles during the simulation. Then the $\pi^0 \to \gamma\gamma$ decay is reconstructed, and the π^0 mesons are saved in a list called 'pi0', which is again matched with the π^0's during the simulation. In the next line, the reconstructed 'pi0' list is saved as a ROOT file with the name given by the input variable 'args.ntupel' appended by '_pi0.root'. 'var_mother' and 'var_pi0', defined in listing 4.5, contain a list

of variables that are saved. This includes, for example, the momentum in the z-direction for both daughters and the angle phi of both daughters. Further, the momentum p of the π^0's, the difference dM between the reconstructed mass and the π^0 mass of about 135 MeV/c^2 [1], and if a reconstructed π^0 matches a π^0 from the simulation, are saved. Why these variables have been chosen we will be disused in the next section 3.2.1. To execute the previous statements, 'process(xxx)' is called. This executes the modules which have been added to the path created in line one. The created data is again stored in a ROOT file.

Listing 3.2: The core part of the python script (listing 4.4) for reconstructing the $\pi^0 \rightarrow \gamma\,\gamma$ decay.

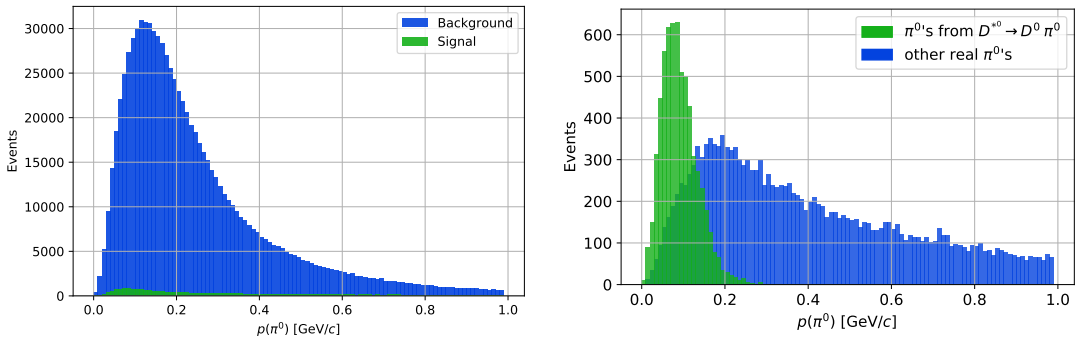
```
1  xxx = create_path()
2
3  inputMdst("default", args.inputfile, path=xxx)
4
5  fillParticleLists([('gamma:all', '')], path=xxx)
6
7  matchMCTruth('gamma:all', path=xxx)
8
9  reconstructDecay('pi0:pi0 -> gamma:all gamma:all', '', path=xxx)
10 matchMCTruth('pi0:pi0', path=xxx)
11
12 variablesToNtuple('pi0:pi0', var_mother + var_pi0 + ['p', 'dM', 'isSignal'], filename=
       args.ntuple + "_pi0.root", path=xxx)
13
14 process(xxx)
```

A more detailed overview of the basf2 framework can be found in the "The Belle II Core Software" paper [15] from the Belle II Framework Software Group.

The generation and the simulation have been done a total of 100 times on the computing system provided by Belle II with 1,000,000 events per run. The data now consists of 100 files, each containing the chosen variables from all possible π^0 mesons reconstructed by the software from the measured photons. This includes correctly paired π^0's and π^0's from wrongly paired photons or background photons.
The left plot in figure 3.1 depicts the difference between signal π^0's and the background as a function of the momentum. The background is peaking around a momentum of 120 MeV/c, this is why often a threshold of about 230 MeV/c is used when analyzing π^0's.

Only about 2.7% of the reconstructed photon combinations correspond to signal π^0 mesons. This could be due to photons that are matched incorrectly, background photons that are matched with each other, or real photons being matched with background photons. Many phenomenons are causing this background. Part of the background beam is produced by synchrotron radiation caused by the e^+e^--beam. Other reasons are, for example, QED processes or impurities within the detectors, such as gas atoms, which can cause background radiation when struck by other particles or photons.

The right histogram in figure 3.1 compares the π^0's originating from the $D^{*0} \rightarrow D^0\,\pi^0$ decay and the other π^0 mesons which are not part of the background but from various other decay channels . The π^0's created by the D^{*0} decay are mostly below the 230 MeV/c threshold and are the subject of this thesis. These π^0's are labeled as 1 (signal) and used for training. The other π^0 mesons makeup about 75 % of all π^0's and are from other decay channels, such as two of the defined D^0 decay channels and from the natural decay channels of $K^\pm$ mesons. These other π^0's are labeled as 0 as well as the beam background. The classifier should learn to discriminate between the π^0 mesons from the wanted decay and every other reconstructed π^0, whether it was reconstructed correctly or not.



(a) (b)

Figure 3.1: (a) Comparison of the Background to signal ratio. (b) Histogram of the signal π^0 mesons and other π^0's that are not part of the background.

3.2 Data preparation

Preparing the data consists of two main steps. First, the features that should be used, need to be chosen. As second the data needs to be prepared and modified to achieve the best performance with the classifiers.

3.2.1 Data selection

To achieve the best performance with the classifiers, collinearity between variables should be avoided. To do so, first all available variables were collected and then reduced during multiple steps, to obtain the features used as input for the classifiers. The predefined steering file *variables.py*, which can be run by in the basf2 environment with the 'basf2 variables.py' command, contains all available variables.
In the first step, all available variables were collected, despite obviously dependent variables like the energy and the momentum of a photon, where only one was selected. To check for further collinearity the Variance inflation factor (VIF) is used [35, 36]. This factor provides information about how much correlated a variable is compared to the others. Hereby a higher VIF indicating more dependencies. The VIF can be computed for every variable, and then variables with a high VIF can be dropped. The threshold used was 10, which indicates high correlations.

After the preparation 16 features per photon remain. In total, the classifiers have an input of 32 features due to the comparison of two photons. The used features can be found in table 3.2 together with the VIF-scores. In the appendix the full list including the description from the *variables.py* steering file can be found (tab. 4.2 and 4.3). Later the feature importance is measured with random forests to study if the number of features used can be reduced further.

Three of the input variables are depicted in figure 3.2. Hereby the first and the second photon is plotted separately, and the background is reduced by 95% to make the signal more visible. The order of the photons is given by the reconstruction software. No difference can be seen between the momenta of the photons. Only between the first and second photon of the consAngleBetweenMomentumAndVertexVector variable, a significant difference can be found. For the error of the momentum in the z-direction,

a small difference is observed. Plots of the distributions from all input variables can be found in the appendix (fig. 4.2, 4.3, 4.4, and 4.5).

Table 3.2: Variables used for training and VIF scores.

Kinematic variables		Cluster variables	
Variable name	VIF	Variable name	VIF
pz	1.97	cosAngleBetweenMomentumAndVertexVector	2.06
pxErr	3.69	cosAngleBetweenMomentumAndVertexVectorInXYPlane	4.29
pyErr	3.77	clusterTiming	1.01
pzErr	3.90	clusterErrorTiming	2.95
phi	2.49	minC2TDist	3.72
b2bPhi	2.46	clusterZernikeMVA	6.04
pRecoilPhi	1.56	clusterLAT	3.21
		clusterAbsZernikeMoment40	4.59
		clusterAbsZernikeMoment51	4.93

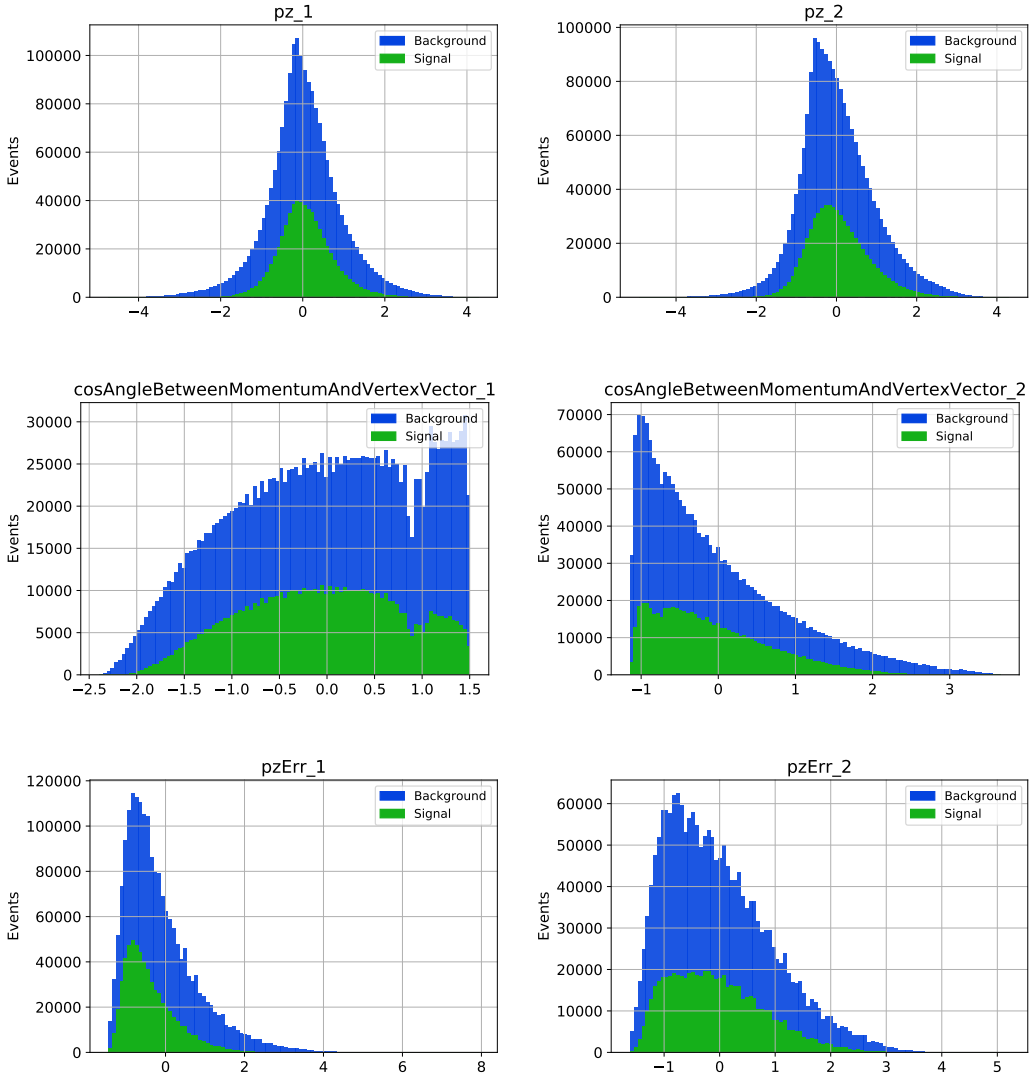


Figure 3.2: Distribution of the momentum in z-direction, the error of the momentum in z-direction, and the cosAngleBetweenMomentumAndVertexVector input variable. Hereby the first (..._ 1) and second photons (..._ 2) are separated. The plots for all input variables can be found in the appendix (fig. 4.2, 4.3, 4.4, and 4.5).

3.2.2 Data scaling

The data preparation consists of multiple steps. Initially, the overall amount of data is reduced with two cuts. First, the π^0 mesons with a momentum above 230 MeV/c are cut off because the pions below this threshold are the interesting ones for this thesis. The second cut is done to eliminate some background. When plotting the difference dM of the reconstructed mass from the known π^0 mass of the signal (the π^0's from the D^{*0} decay) and the background (fig. 3.3), it can be observed that the mass difference dM of the background is spread more widely. This can be used to eliminate background by applying a cut where only the data with an invariant mass between -0.075 and 0.02 GeV/c^2 is kept. With this cut, the background is reduced by about 70% with a loss of only 1% of the signal π^0's.

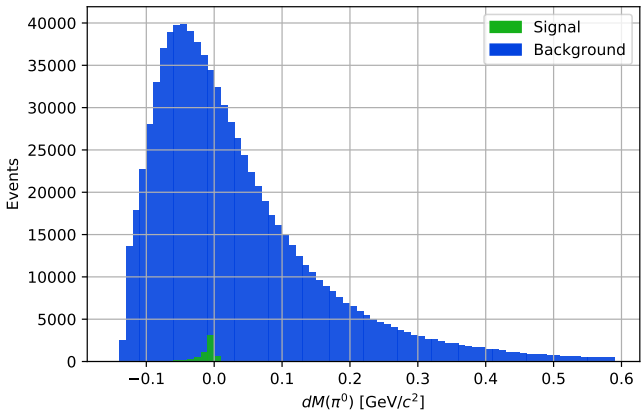


Figure 3.3: Histogram of the π^0 mesons invariant mass of the signal and background.

The next step is to create the labels. The link to jupyter notebook containing the corresponding code can be found in the appendix 4.2. A label of 1 represents the wanted low momentum π^0's created during the decay of the D^{*0}. This is done by checking if 'isSignal' equals 1, which means the reconstructed particle is not part of the background, and by making sure that the π^0 originates from the desired decay. This is done by checking if 'genMotherPDG' equals 423 or -423, which is the PDG code for D^{*0} and the corresponding anti-particle. The remaining data is labeled with 0.

As seen before, the background exceeds the signal by far. The datasets of each simulation contain only about 2.5% tuples labeled as 1. To create more balanced datasets, 95% of the background π^0's are dropped. As a result, the created datasets contain about 33% data labeled 1.
The last modification done to the data, is to scale the data featurewise to a mean value of 0 and a standard deviation of 1. This transformation is not mandatory but can speed up the training process of neural networks. This is done with the $StandardScaler$ method provided by the sklearn.preprocessing library. To scale the data from each simulation equally, the StandardScaler is fitted on a certain number of simulations (in this case 50) and then applied on all datasets. This way all datasets are transformed the same way and the transformed datasets are still comparable. This should ensure that the trained classifiers perform well even on never before seen datasets.

In the end, the datasets from all 100 runs (simulations) are concatenated, whereby 4 runs are separated as validation data and shuffled. The remaining 96 runs are shuffled and split up into training and test data. From the 96 runs, 75% of the tuples are used for training, the remaining ones for testing. In total, all datasets contain 1,914,246 events of which 625,729 are labeled with a 1. Table 3.3 summarizes all key parameters and contains the number of events in the different datasets and the ratio of data labeled 1 in comparison to the whole dataset. Note that the exact number of events in the different datasets is not consistent even when creating the dataset from the same simulations. This is caused by the dropping of tuples labeled as 0, which is done randomly. But due to the largeness of the dataset, the overall alterations are small.

A link to the GitHub repository containing the code can be found in the appendix 4.2.

Table 3.3: Summary of the key parameters of the simulated and prepared data.

Total number of runs		100
Number of runs used for validation		4
Number of events per run		1,000,000
Total length of the dataset		1,914,246
Number of low momentum π^0 combinations		625,729
Length of the dataset	training	1,378,943
	test	459,648
	validation	75,655
Percentage of low momentum π^0 combinations	training	0.327
	test	0.326
	validation	0.326
Number of features		32 (16 per daughter)

3.3 Evaluation

To measure the performance, the area under the curve (AUC), the F_1-score on the test and validation data, and the precision on the validation data is used. The area under the curve provides information about the overall performance of the classifier [1]. The F_1-scores for the training and validation data can be compared to check for overfitting. The precision of the validation data is important because this parameter should be as high as possible to ensure that the predicted low momentum pions are correctly classified as such. Hereby the F_1-scores and the precision are calculated with a decision threshold of 0.5 for the whole evaluation.

3.3.1 Random Forest

First random forests are used to determine the feature importance and as an first attempt to create a valid classifier. In the beginning, two decision trees are compared before several random forests of different sizes are used.

The difference between the two trained decision trees is the criterion used to identify the best splits. For the trees, the Gini and the Entropy methods are used. Table 3.4 contains the results from both trees. Despite no restrictions on the trees regarding depth, the maximum number of leaves or nodes, and creating only pure leaves containing only a single label, both trees do not overfit. Overfitting can be detected by a significantly higher F_1-score on the training data compared to the validation data. The precision of both trees is similar with about 62% of positive predictions being truly positive. This is about double the precision of a random classifier, which would have a 33% precision score due to a balance of 1:2 positive labels to negative labels in the training set.

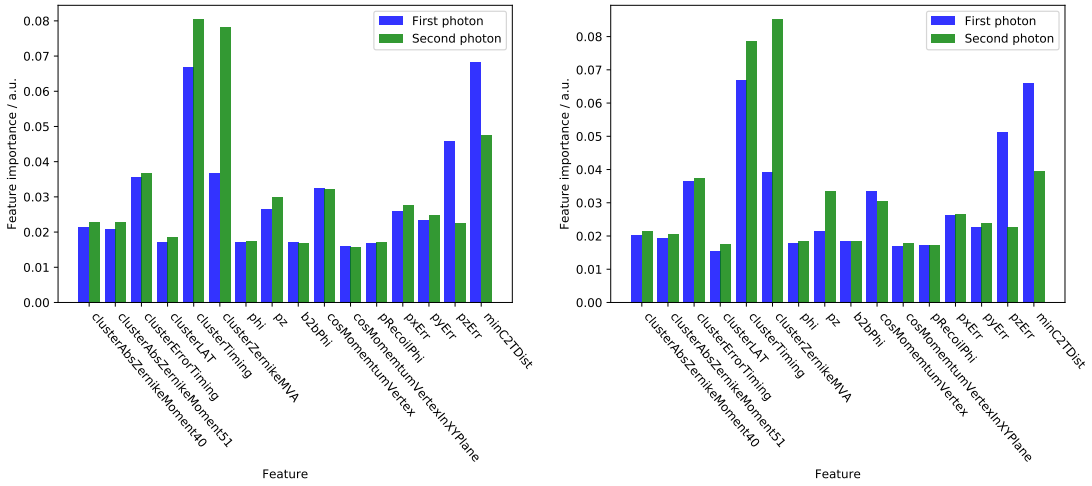
The feature importance of both trees is depicted in figure 3.4. The importance is plotted for each variable and each photon. As mentioned for the input vectors of the decision trees, the two feature vectors for each photon are concatenated. In the figure, the first and second photon is distinguished to identify differences between those. Both plots are already pretty consistent and indicate clear differences in feature importance. Strange is that there is a significant difference between the feature importance of the first and the

[1] The AUC score is not used for decision trees, only for random forests and the different neural networks.

Table 3.4: Comparison of two decision trees with a different splitter criterion. The used decision threshold for the F_1-scores is 0.5.

Split criterion	F_1-score (trainings)	F_1-score (validation)	Precision (validation)
Gini	0.628	0.628	0.623
Entropy	0.633	0.629	0.626

second photon for some variables. Intuitively the feature importance should be equal for both photons because the order of a pair should be irrelevant. Only for the *cosAngleBetweenMomentumAndVertexVector* and the *pzErr* variable, a difference would be expected due to the previously found differences in the distributions of the first and second photon for these variables. The plots of the other variables showed no differences between the two photons.



(a) (b)

Figure 3.4: Feature importance for the two decision trees. (a) Entropy, (b) Gini.

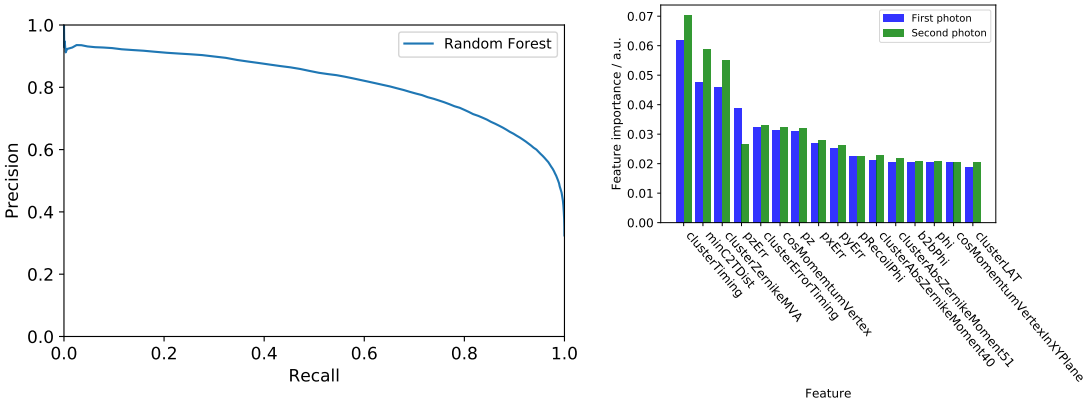
To improve the performance of decision trees and get an averaged feature importance, random forests are used. Due to overfitting being no issue for decision trees, the trees in the random forests are not restricted either. The differences between the forests are the number of estimators used, and again the split criterion during training. Table 3.5

contains forests with 20, 40, and 100 estimators for both Gini and Entropy, and one large random forest with a total of 250 trees with Entropy as the split criterion. Again the differences in performance between Gini and Entropy are insignificant. Overall the performance of the forests is pretty similar. Larger forests tend to have a better AUC and F_1-score than smaller forests, but the precision is, despite some small fluctuations, consistent. Figure 3.5 (a) depicts the precision-recall curve for the forest with 100 estimators and Gini as the split criterion. As expected, random forests perform better than single decision trees. The F_1-scores are about 0.1-0.14 points higher, and the precision is about 14% higher. Still, a higher precision would be desirable. Furthermore, with a growing forest size, the computing times start to increase significantly.

Table 3.5: Performance of different random forests. The used decision threshold for the F_1-scores is 0.5.

Number of estimator	Split criterion	AUC	F_1-score (trainings)	F_1-score (validation)	Precison (validation)
20	Gini	0.791	0.720	0.721	0.763
20	Entropy	0.792	0.727	0.729	0.760
40	Gini	0.804	0.734	0.735	0.766
40	Entropy	0.807	0.741	0.743	0.761
100	Gini	0.812	0.743	0.744	0.767
100	Entropy	0.817	0.749	0.751	0.763
250	Entropy	0.821	0.754	0.754	0.765

Figure 3.5 (b) confirms the feature importance obtained from the single trees. There are still discrepancies in some variables between the 'first photon' and the 'second photon'. Regarding the feature importance *clusterTiming*, *clusterZernikeMVA*, and *minC2TDist* have the highest importance, followed by *clusterErrorTiming*, *pz cosAngleBetweenMomentumAndVertexVector* (*cosMomemtumVertex*), and *pzErr*. *pxErr* and *pyErr* have a lower feature importance but still slightly higher than the remaining features, which all have a similar importance. Despite some features being certainly more important, even the lowest feature importance score is only a third of the highest. That implies that all features contribute to the classification to a certain extend. Still, the idea is to take



(a) (b)

Figure 3.5: (a) Precision-Recall curve for the random forest with 100 estimator (Gini) and (b) the feature importance for the same classifier sorted by the first photon.

the most important features, including *pxErr*, *pyErr*, and higher, and train the neural network only with these features to create a less complicated and faster learning neural network. But due to all features contributing to the classification in the random forest, this smaller input may not be sufficient enough. This selective input needs to be tested against the full input to ensure that a potential information loss does not cause a decrease in performance.

3.3.2 Neural network

In the next step, feed-forward neural networks are used for the classification. First a network with the same architecture, despite the input layer size, will be trained with all and with the most important features to compare the performance. Subsequent three networks, each consisting of four layers but different amounts of nodes, will be trained and compared. The batch size used for all feed-forward neural networks and Siamese neural networks is 1024 with the 'Adam' optimizer and binary corssentropy as a loss function.

To spot a potential difference between the used input features, two feed-forward neural networks with identical nodes despite the input nodes are tested (table 3.6). Figure 3.6 (a)

contains the accuracy during each training epoch. The performance of the network with all features is significantly better. This applies not only to the accuracy but the AUC, the F_1-scores, and the precision as well (see table 3.8). This result is consistent for other tested networks. Due to this, for the further feed-forward neural network, all features are used.

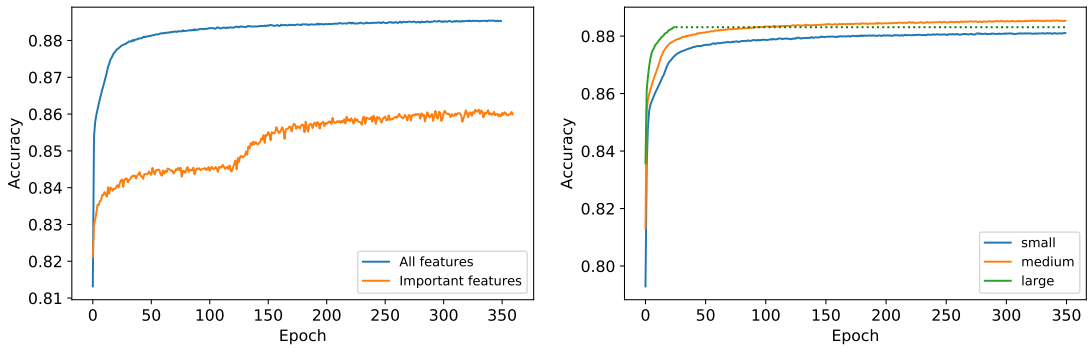
Table 3.6: Design parameter of the feed-forward neural network.

Layer	-	Activation	Nodes
0 (Input)	-	-	32 / 18
1	Dense	tanh	128
2	Dense	tanh	32
3	Dense	tanh	8
4 (Output)	Dense	sigmoid	1

To compare the performance of different feed-forward neural networks and find an optimal configuration, three networks will be compared. The amount of layers is four for all networks. The difference is in the number of nodes per layer. One network has a small number of nodes ranging from 4 to 40, one has a medium amount with 8 to 128, and one ranging from 32 to 512 is significantly larger compared to the first ones. The detailed configurations can be found in table 3.8.

Table 3.7: The feed-forward neural networks used for comparison. The type and activation function for each layer resembles the ones from the network given in table 3.6.

Layer	Nodes		
	small	medium	large
0 (Input)	32	32	32
1	40	128	512
2	20	32	128
3	4	8	32
4 (Output)	1	1	1

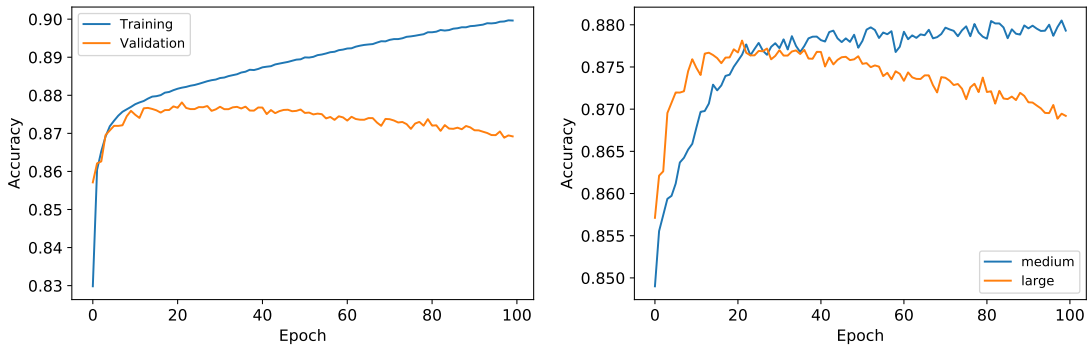


(a) (b)

Figure 3.6: (a) Comparison of the accuracy when using all features or only the important ones. (b) Comparison of the accuracy for the neural network. The training for the large network was stopped before overfitting occurred (25 steps).

The medium-sized network and the small network start to plateau towards the end of the 350 epochs (fig. 3.6). These two networks have fewer parameters that can be adapted during training than there are training examples, and therefore overfitting is less likely, and the accuracy evens out. The large network reaches the same accuracy as the smaller ones in fewer steps, but overfitting starts to become an issue. Figure 3.7 depicts the training and validation accuracy for the large network during training. After about 20 to 25 steps the validation accuracy peaks and starts to decrease despite the model improving on the training data. When comparing the validation accuracy during the training of the large and the medium model, the accuracy of the medium model increases the whole plotted 100 epochs. Despite this rapid overfitting, the large model reaches a similar accuracy as the medium model in fewer epochs before decreasing again.

To compare the overall performance of all three models, early stopping was used to stop the training process as soon as the validation accuracy started to decrease. With early stopping, the accuracy during training for all three models is almost identical. This is confirmed when comparing the AUC, the F_1-scores, and the precision (table 3.8). The medium network seems to be slightly better with, for example, the precision being 0.5-0.7% percentage points higher. But due to certain randomness, when creating neural networks (for example, random starting weights, random batches, ...), the performance can change slightly. Therefore all three models perform equally well.



(a) (b)

Figure 3.7: (a) Overfitting in the large feed-forward neural network, (b) Comparison of the validation accuracy of the medium and large network.

Compared to random forests, the performance increased significantly. For example, the precision gained about 3.5 % percentage points. But, as previously discussed, the pairwise input creating a potential symmetry and information loss problem is still mostly neglected. To consider this in the next step, siamese neural networks will be used.

Table 3.8: Performance of the feed-forward neural networks from table 3.8. The used decision threshold for the F_1-scores is 0.5.

Neural Network size	AUC	F_1-score (trainings)	F_1-score (validation)	Precison (validation)
small	0.880	0.823	0.822	0.797
medium (all features)	0.882	0.824	0.822	0.802
medium (important features)	0.851	0.795	0.794	0.760
large	0.874	0.819	0.816	0.795

3.3.3 Siamese neural network

Again several combinations of nodes per layer are tested and compared. The design parameters of the used networks can be found in table 3.10. The first feed-forward neural network of the siamese neural network consists of 2 layers and the second one of three layers. The first siamese neural networks, of the three tested networks, all have 96 output nodes that are merged, the other node numbers are altered. The last two models in table 3.10 have the same weights as network 1, but the number of output nodes of the first network is greater. The idea is to test if the higher amount of information of each photon (more output nodes of the first feed-forward neural network), provided for comparison, results in a better performance. Table 3.9 shows the type of the layer and the activation function used for all networks, and exemplary the nodes of the first neural network. The basic conditions as batch size, optimizer, and loss, for testing, are adopted from the feed-forward neural networks in the previous section. To prevent overfitting, early stopping was used as soon as the validation loss decreased over 20 epochs.

Table 3.9: Design parameter of the siamese neural network (network 1 in table 3.10).

Layer	Type	Activation	Nodes	
A0 (Input)	-	-	128	128
A1	Dense	tanh	64	64
A2	Dense	tanh	96	96
B0	absolute difference	-	96	
B1	Dense	tanh	32	
B2	Dense	tanh	8	
B3 (Output)	Dense	sigmoid	1	

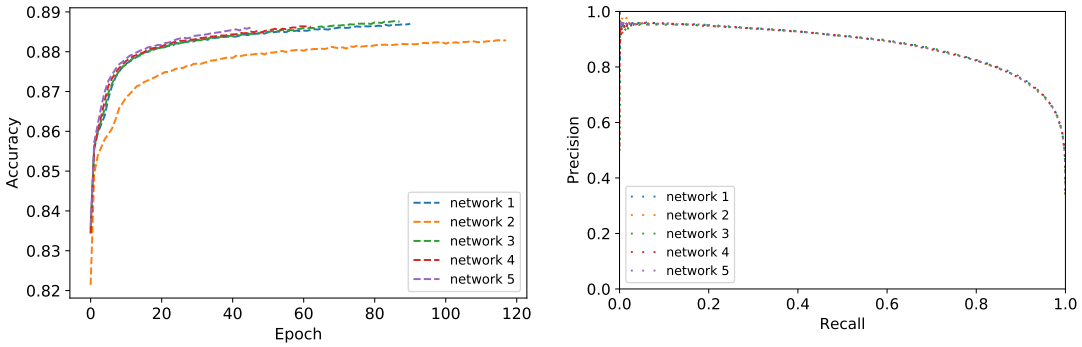
As previously, the AUC, the F_1-scores, and the precision are compared. Table 3.11 contains these parameters for the five networks. The found results are pretty similar for all models. Small differences can be seen in the accuracy during training (figure 3.9 (a)), but the Precision-Recall curves are, despite some fluctuation near a recall of 0, indistinguishable.

As done with the feed-forward neural networks, the performance of a siamese neural network with all features as input is compared to the performance of a network, with

Table 3.10: Siamese neural networks used for comparison. The type and activation function for each layer resembles the ones from the neural network given in table 3.9.

Layer	Nodes				
	network 1	network 2	network 3	network 4	network 5
A0 (Input)	128	48	128	128	128
A1	64	16	64	64	64
A2	96	96	96	160	250
B0	96	96	96	160	250
B1	32	32	128	32	32
B2	8	8	16	8	8
B3 (Output)	1	1	1	1	1

the same architecture, despite the input vector consisting of only the important features. For a single network, the performance was significantly worse with only the important features, and therefore the previous siamese neural networks were trained on all features. For comparison, the first siamese neural network was again trained, with only the most important features. When comparing the performance (table 3.11), the network with only the important features still performs worse. This is confirmed when comparing the validation accuracy during training and the Precision-Recall curve in figure 3.9 (a) and (b). But when comparing the PRC, the network trained with fewer input features has still a decent performance.

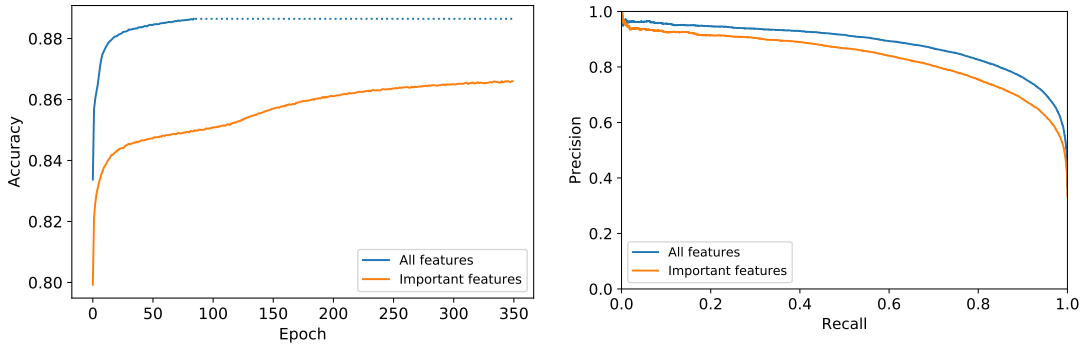


(a) (b)

Figure 3.8: Comparison of all networks from table 3.10: (a) accuracy during training (the training was stopped when overfitting started) (b) Precision-Recall curves.

Table 3.11: Performance of the siamese neural networks from table 3.10 and the first network training only with the most important features. The used decision threshold for the F_1-scores is 0.5.

Network	AUC	F_1-score (training)	F_1-score (validation)	Precison (validation)
1	0.881	0.825	0.825	0.795
2	0.881	0.824	0.822	0.798
3	0.881	0.824	0.822	0.798
4	0.880	0.821	0.819	0.806
5	0.880	0.822	0.819	0.801
1 (important features)	0.834	0.779	0.775	0.762



(a) (b)

Figure 3.9: Comparison of the first network (compare table 3.10) with all input features and only the important input features: (a) training accuracy, (b) Precision-Recall curves.

4 Discussion and Conclusion

4.1 Discussion

The three classification methods, especially the feed-forward neural network and the siamese neural network, have an acceptable performance. In table 4.1 the performances of the three methods are summarized. The random forests perform slightly worse than the neural networks and have significantly longer computing time, especially with more trees. The performances of both the Siamese neural networks and the feed-forward neural networks are almost identical. The same result can be found when comparing the Precision-Recall curves (fig. 4.1 (a)). Thereby all models differ significantly in their weights after the training. Reasons for this equal performance could be that all networks find the same pattern in the data, or that the upper bound regarding the accuracy is reached due to a certain noise in the data. Otherwise, higher fluctuations between the models and even between networks with the same architecture trained multiple times, would be expected. These fluctuations are mostly caused by certain random parameters during training, such as the starting weights of the connections.

When plotting the momentum distribution of all signal π^0 mesons and the momentum of all true positive predictions (fig. 4.1 (b)), slower pions seem to be predicted more accurately, but no distinct pattern can be found with confidence.

To study the performance a bit more in-depth, figure 4.1 (c) depicts all outputs made by a siamese neural network for three simulations with the positive labels marked in green. The classifier seems to be pretty confident when labeling most of the negative cases, as they have output values very close to zero. Labeling the positive cases seems to be harder since the output scores for positive labels are more spread, and the highest value is only 0.965. This difference is partially due to the imbalance in the dataset

Table 4.1: Comparison of the performance of the random forest, feed-forward neural network, and siamese neural network with the highest precision.

Classifier	AUC	F_1-score (training)	F_1-score (validation)	Precison (validation)
Random forest	0.812	0.743	0.744	0.767
Neural network	0.882	0.824	0.822	0.802
Siamese neural network	0.880	0.821	0.819	0.806

(67% negative labels, 33% positive labels), but the peak just above 0 is still significantly higher compared to what the imbalance would cause.

This plot, as well as the PRC's, are suggesting that the used threshold should be increased, especially when higher precision is wanted. For example with a threshold of about 0.86, the precision reaches 91.2% with a recall, the probability that a positive class is labeled as such, of still 50%.

When comparing the performance of the tested neural networks with all input features and only the important features. The performance with only the important features is worse. But the classifiers are still capable of achieving decent AUC-scores of about 0.84 and good PRC's (compare figure 3.9). This concludes that the important features, and maybe even fewer, are providing enough information to classify many tuples correctly. Still giving more features enables the classifiers to get even more tuples correctly. It is possible that feed-forward neural networks or siamese neural networks with only the important features would be able to this as well, but considerably more steps or training data would be needed (compare figure 3.6 and 3.9).

Despite the differences in the architecture, the feed-forward neural networks do not seem to suffer any information loss or symmetry problems due to the merged single-photon vectors. Especially interesting is that symmetry does not cause a problem, because the training data contains every pair only in one combination. This could be because the π^0's can be distinguished by certain patterns that the feed-forward neural network can learn because the training data is sufficiently large enough, containing

enough pairs in both orders.

To research the performance of a trained classifier with π^0 mesons from other decay channels than the used one, the first feed-forward neural network was used to classify other π^0's. In figure 4.1 (d) all other real π^0's are plotted as a function of the momentum. These π^0's are partially misclassified as positive by the classifiers, but only in the momentum range where the training examples are located. The false-positive predictions are marked in green in the figure. Overall, the classifiers seem to have no particular difficulties differentiating between the wanted π^0 mesons and other real π^0 mesons.

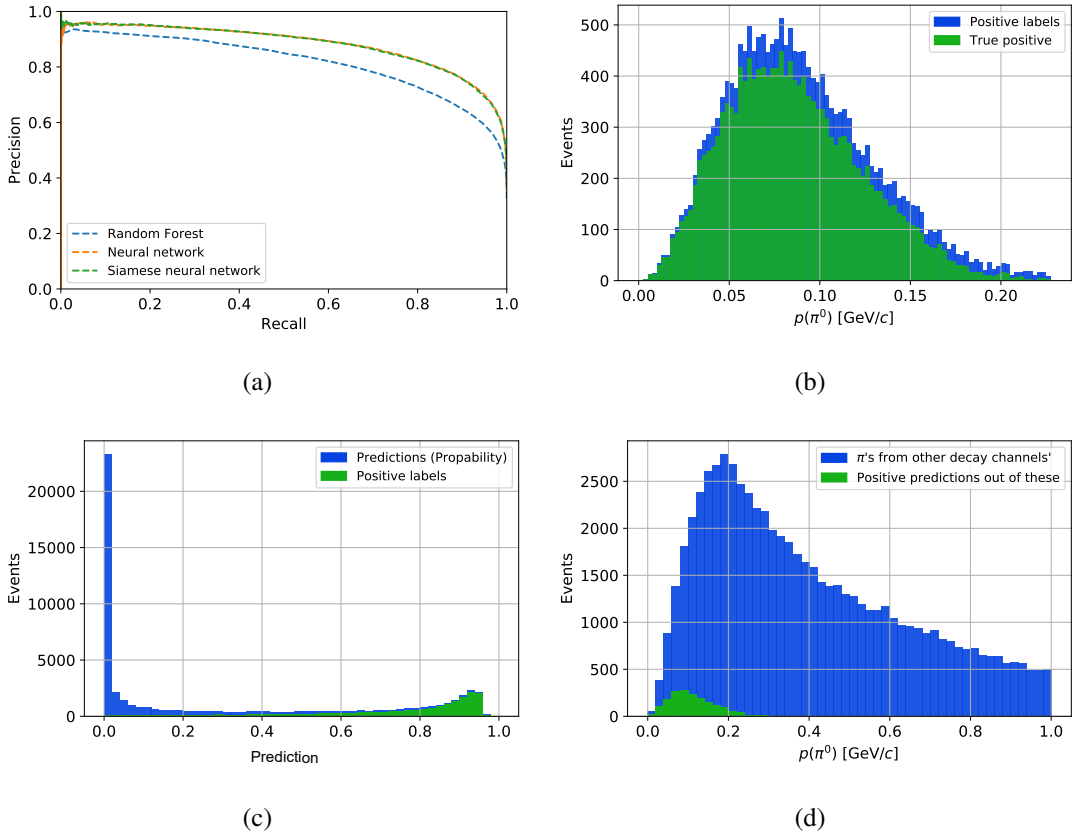


Figure 4.1: (a) Comparison of the best Precision-Recall curve of a random forest, a feed-forward neural network, and a siamese neural network. (b) Histogram of all positive labels and the correctly predicted positive labels. (c) Histogram of all prediction probability's for a siamese neural network, with the positive labels marked in green. (d) Histogram of the momentum of π^0 mesons from different decay channels than the desired one, and out of these the π^0 mesons that were falsely predicted positive marked in green. For (b), (c), and (d) three simulation runs were used.

4.2 Conclusion

In conclusion, random forests, feed-forward neural networks, and siamese neural networks are capable of identifying low momentum π^0 mesons from the background reasonably well. Hereby feed-forward neural networks with all features and siamese neural networks with all features perform the best. Depending on the decision threshold, a precision of over 90% can be achieved with a recall still above 50% .
The results indicate that the data has an underlying pattern that makes π^0's distinguishable from the especially high background in the low momentum spectrum. Further research could be done to find this potentially underlying patter, which may apply to different particles as well, and regarding this, finding the core features needed for the classification.

Appendix

Code - Classifiers and analysis

The Jupiter notebooks used for preparing the data, building the classifiers, and analyzing the performance can be found in the GitHub repository hosted at `https://github.com/LukasHoller/Bachelorthesis-Code`.

The source code is divided into a single notebook for each classification method, one notebook 'data_ preparation' used for preparing the data, and two notebooks used for analyzing and plotting the data as well as the results.

Code - Generation, simulation and reconstruction

Listing 4.1: Decy file used for simulation - decay.dec.

```
Decay Upsilon(4S)
1.0 B+ B- VSS;
Enddecay

Decay B+
0.5 K+ D*0 anti-D0 PHOTOS PHSP;
0.5 K+ D0 anti-D*0 PHOTOS PHSP;
Enddecay

Decay D*0
0.65 D0 pi0 PHOTOS VSS;
0.35 D0 gamma PHOTOS VSP_PWAVE;
Enddecay
CDecay anti-D*0

Decay D0
0.13 K- pi+              PHOTOS PHSP;
0.47 K- pi+ pi0          PHOTOS D_DALITZ;
0.27 K- pi+ pi+ pi-      PHOTOS PHSP;
0.13 K_S0 pi+ pi-        PHOTOS D_DALITZ;
0.13 K- K+               PHOTOS PHSP;
Enddecay
CDecay anti-D0

End
```

Listing 4.2: Phyton script for generating 1.000.000 events.

```
import basf2 as b2
import generators as ge
import modularAnalysis as ma

# Defining custom path
my_path = b2.create_path()

# Setting up number of events to generate
ma.setupEventInfo(noEvents=1000000, path=my_path)

# Adding generator
ge.add_evtgen_generator(path=my_path, finalstate='signal',
                        signaldecfile=b2.find_file('../decay_file/decay.dec'))

# If the simulation and reconstruction is not performed in the sam job,
# then the Gearbox needs to be loaded with the loadGearbox() function.
ma.loadGearbox(path=my_path)

# dump generated events in DST format to the output ROOT file
my_path.add_module('RootOutput', outputFileName='out.root')

# process all modules added to the path
b2.process(path=my_path)

# print out the summary
print(b2.statistics)
```

Listing 4.3: Phyton script for the simulation of the given events.

```
import basf2 as b2
import mdst as mdst
import simulation as si
import reconstruction as re
import modularAnalysis as ma

b2.conditions.disable_globaltag_replay()

# create path
my_path = b2.create_path()

# load input ROOT file
ma.inputMdst(environmentType='default', filename='', path=my_path)

# simulation
si.add_simulation(path=my_path)

# reconstruction
re.add_reconstruction(path=my_path)

# dump in MDST format
mdst.add_mdst_output(path=my_path, mc=True)

# Show progress of processing
progress = b2.register_module('ProgressBar')
my_path.add_module(progress)

# Process the events
b2.process(my_path)

# print out the summary
print(b2.statistics)
```

Listing 4.4: Phyton script for the reconstruction of the $\pi^0 \rightarrow \gamma\gamma$ decay for all 100 simulations.

```
from basf2 import *
from modularAnalysis import *
from stdV0s import stdKshorts
from variables import variables
import pdg
import os.path
import sys
from vertex import *
from reconstruction import *
from ROOT import Belle2
from glob import glob
from stdPi0s import *
from stdPhotons import *
from basf2 import use_central_database
import variables
from variables import variables as v
import basf2_mva
from reco_variables import *
import argparse
import numpy as np


parser = argparse.ArgumentParser()
parser.add_argument("inputfile", help="Total path to input file.", type=str)
parser.add_argument("ntuple", help="Total path to output file.", type=str)

args = parser.parse_args()

set_log_level(LogLevel.ERROR)

for i in np.arange(1,101,1):
    print('\n', '='*30, '\n', 'RUN ', str(i), '\n', '='*30, '\n' , sep='')

    xxx = create_path()

    filename = args.inputfile + '' + str(i) + '.root'

    inputMdst("default", filename, path=xxx)

    fillParticleLists([('gamma:all', '')], path=xxx)

    matchMCTruth('gamma:all', path=xxx)

    reconstructDecay('pi0:pi0 -> gamma:all gamma:all', '', path=xxx)
    matchMCTruth('pi0:pi0', path=xxx)

    variablesToNtuple('pi0:pi0', var_mother + var_pi0 + ['p', 'dM', 'isSignal', '
```

```
    daughter(0, mdstSource)', 'daughter(1, mdstSource)'], filename=args.ntuple + '_' +
     str(i) + "_pi0.root", path=xxx)

    progress = register_module('Progress')
    xxx.add_module(progress)

    process(xxx)
    print(statistics)
```

Listing 4.5: Phyton script to add aliases for the used variables and to define a list containing all variables.

```
import variables

variables.variables.addAlias('clusterAbsZernikeMoment40_1', 'daughter(0,
    clusterAbsZernikeMoment40)')
variables.variables.addAlias('clusterAbsZernikeMoment40_2', 'daughter(1,
    clusterAbsZernikeMoment40)')

variables.variables.addAlias('clusterAbsZernikeMoment51_1', 'daughter(0,
    clusterAbsZernikeMoment51)')
variables.variables.addAlias('clusterAbsZernikeMoment51_2', 'daughter(1,
    clusterAbsZernikeMoment51)')


variables.variables.addAlias('clusterTiming_1', 'daughter(0, clusterTiming)')
variables.variables.addAlias('clusterTiming_2', 'daughter(1, clusterTiming)')

variables.variables.addAlias('clusterErrorTiming_1', 'daughter(0, clusterErrorTiming)')
variables.variables.addAlias('clusterErrorTiming_2', 'daughter(1, clusterErrorTiming)')

variables.variables.addAlias('clusterLAT_1', 'daughter(0, clusterLAT)')
variables.variables.addAlias('clusterLAT_2', 'daughter(1, clusterLAT)')

variables.variables.addAlias('clusterZernikeMVA_1', 'daughter(0, clusterZernikeMVA)')
variables.variables.addAlias('clusterZernikeMVA_2', 'daughter(1, clusterZernikeMVA)')


variables.variables.addAlias('phi_1', 'daughter(0, phi)')
variables.variables.addAlias('phi_2', 'daughter(1, phi)')

variables.variables.addAlias('b2bPhi_1', 'daughter(0, b2bPhi)')
```

```
variables.variables.addAlias('b2bPhi_2', 'daughter(1, b2bPhi)')

variables.variables.addAlias('pRecoilPhi_1', 'daughter(0, pRecoilPhi)')
variables.variables.addAlias('pRecoilPhi_2', 'daughter(1, pRecoilPhi)')


variables.variables.addAlias('cosAngleBetweenMomentumAndVertexVector_1', 'daughter(0,
    cosAngleBetweenMomentumAndVertexVector)')
variables.variables.addAlias('cosAngleBetweenMomentumAndVertexVector_2', 'daughter(1,
    cosAngleBetweenMomentumAndVertexVector)')

variables.variables.addAlias('cosAngleBetweenMomentumAndVertexVectorInXYPlane_1', '
    daughter(0, cosAngleBetweenMomentumAndVertexVectorInXYPlane)')
variables.variables.addAlias('cosAngleBetweenMomentumAndVertexVectorInXYPlane_2', '
    daughter(1, cosAngleBetweenMomentumAndVertexVectorInXYPlane)')


variables.variables.addAlias('pz_1', 'daughter(0, pz)')
variables.variables.addAlias('pz_2', 'daughter(1, pz)')

variables.variables.addAlias('pxErr_1', 'daughter(0, pxErr)')
variables.variables.addAlias('pxErr_2', 'daughter(1, pxErr)')

variables.variables.addAlias('pyErr_1', 'daughter(0, pyErr)')
variables.variables.addAlias('pyErr_2', 'daughter(1, pyErr)')

variables.variables.addAlias('pzErr_1', 'daughter(0, pzErr)')
variables.variables.addAlias('pzErr_2', 'daughter(1, pzErr)')


variables.variables.addAlias('minC2TDist_1', 'daughter(0, minC2TDist)')
variables.variables.addAlias('minC2TDist_2', 'daughter(1, minC2TDist)')

# Create a list with all previously defined variables
var_pi0 = ['clusterAbsZernikeMoment40_1', 'clusterAbsZernikeMoment40_2', '
    clusterAbsZernikeMoment51_1', 'clusterAbsZernikeMoment51_2', 'clusterErrorTiming_1
    ', 'clusterErrorTiming_2', 'clusterLAT_1', 'clusterLAT_2', 'clusterTiming_1', '
    clusterTiming_2', 'clusterZernikeMVA_1', 'clusterZernikeMVA_2', 'phi_1', 'phi_2',
    'pz_1', 'pz_2', 'b2bPhi_1', 'b2bPhi_2', 'cosAngleBetweenMomentumAndVertexVector_1'
    , 'cosAngleBetweenMomentumAndVertexVector_2', '
    cosAngleBetweenMomentumAndVertexVectorInXYPlane_1', '
    cosAngleBetweenMomentumAndVertexVectorInXYPlane_2', 'pRecoilPhi_1', 'pRecoilPhi_2'
    , 'pxErr_1', 'pxErr_2', 'pyErr_1', 'pyErr_2', 'pzErr_1', 'pzErr_2', 'minC2TDist_1'
    , 'minC2TDist_2']

var_mother = ['genMotherPDG', 'genMotherPDG(1)']
```

Features

Table 4.2: List of kinematic variables used. The highlighted variables are the more important features.

Variable name	Description	Importance	
		first γ	second γ
pz	momentum component z	0.032	0.032
pxErr	error of momentum component x	0.027	0.029
pyErr	error of momentum component y	0.025	0.027
pzErr	error of momentum component z	0.038	0.026
phi	momentum azimuthal angle in radians	0.020	0.021
b2bPhi	Azimuthal angle in the lab system that is back-to-back to the particle in the CMS	0.021	0.021
pRecoilPhi	Azimutal angle of aparticle's missing momentum in the lab system	0.022	0.023
cosAngleBetweenMomentumAndVertexVector	cosine of the angle between momentum and vertex vector (vector connecting ip and fitted vertex) of this particle	0.032	0.033
cosAngleBetweenMomentumAndVertexVectorInXYPlane	cosine of the angle between momentum and vertex vector (vector connecting ip and fitted vertex) of this particle in xy-plane	0.020	0.021

Table 4.3: List of all used variables regarding the ECL cluster. The highlighted variables are the more important features.

Variable name	Description	Importance	
		first γ	second γ
clusterTiming	Returns ECL cluster's timing. Photon timing is given by the fitted time of the recorded waveform of the highest energetic crystal in a cluster	0.061	0.074
clusterErrorTiming	Returns ECL cluster's timing uncertainty that contains 99% of true photons (dt99)	0.033	0.032
minC2TDist	Returns distance between ECL cluster and nearest track hitting the ECL	0.048	0.055
clusterZernikeMVA	Returns output of a MVA using eleven Zernike moments of the cluster. Zernike moments are calculated per shower in a plane perpendicular to the shower direction	0.045	0.058
clusterLAT	Returns lateral energy distribution (shower variable)	0.020	0.020
clusterAbsZernike-Moment40	Returns absolute value of Zernike moment 40	0.021	0.023
clusterAbsZernike-Moment51	Returns absolute value of Zernike moment 51	0.021	0.022

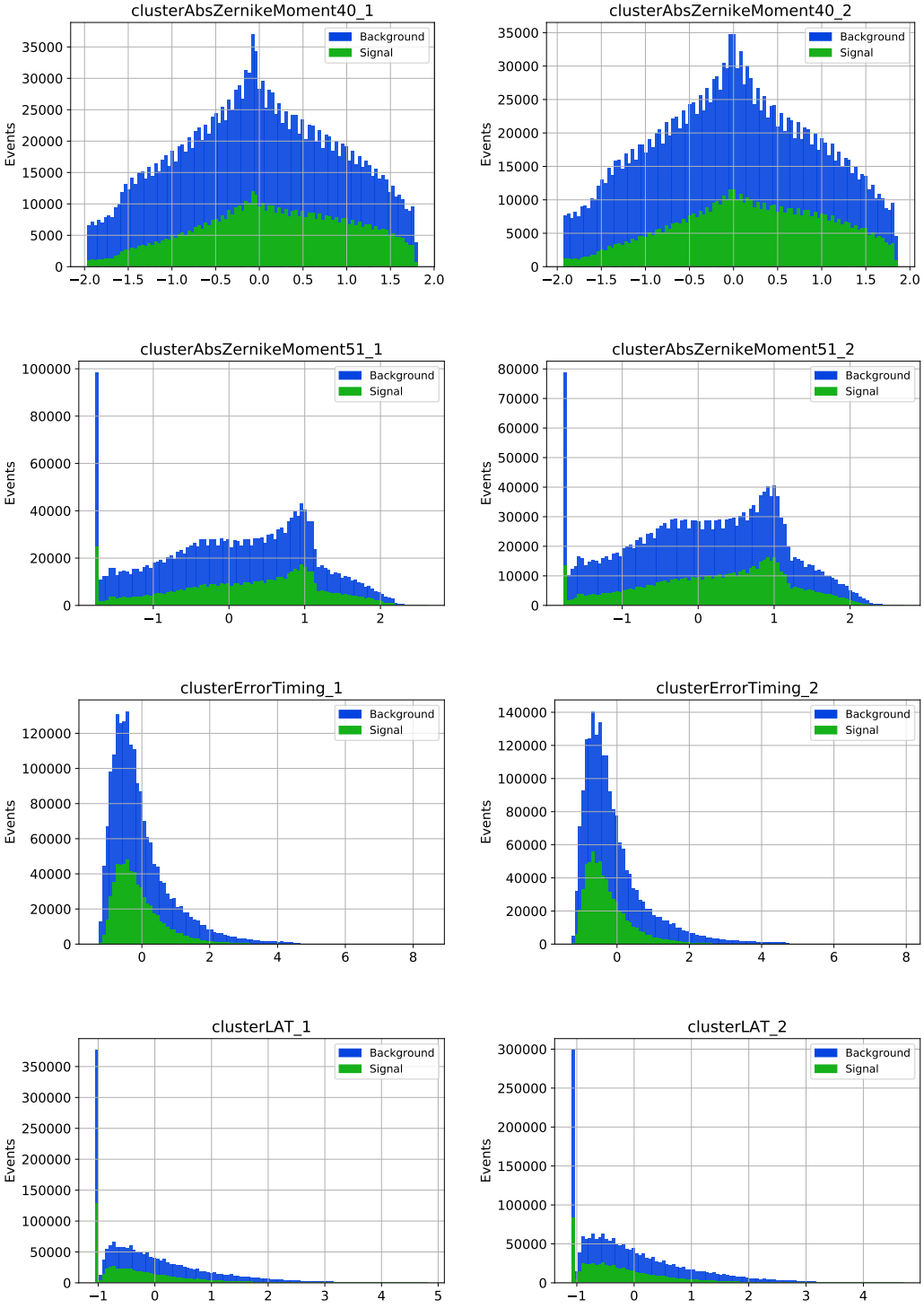


Figure 4.2: Distribution of the used input variable. Hereby the first (..._ 1) and second photons (..._ 2) are seperated.

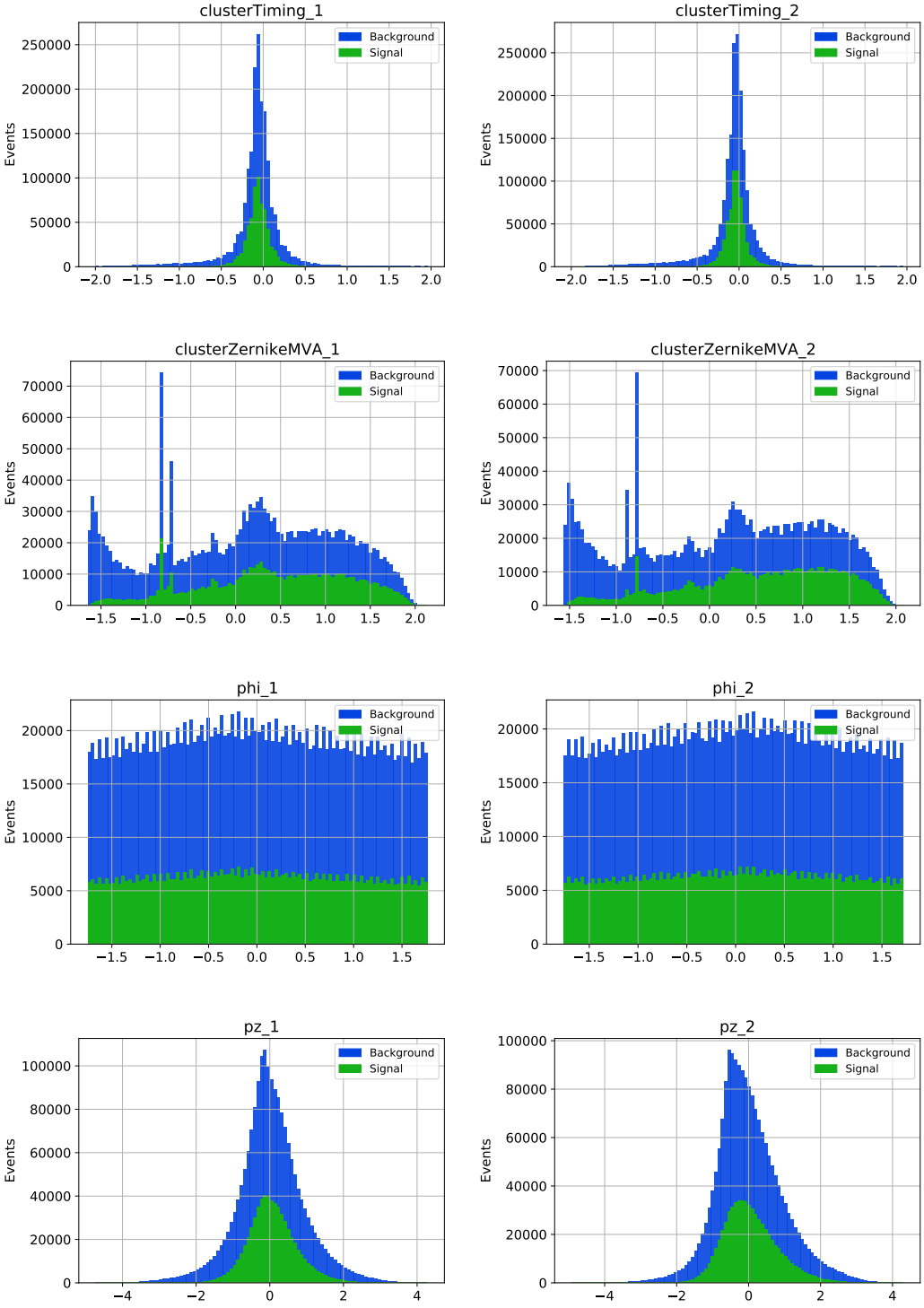


Figure 4.3: Distribution of the used input variable. Hereby the first (..._ 1) and second photons (..._ 2) are seperated.

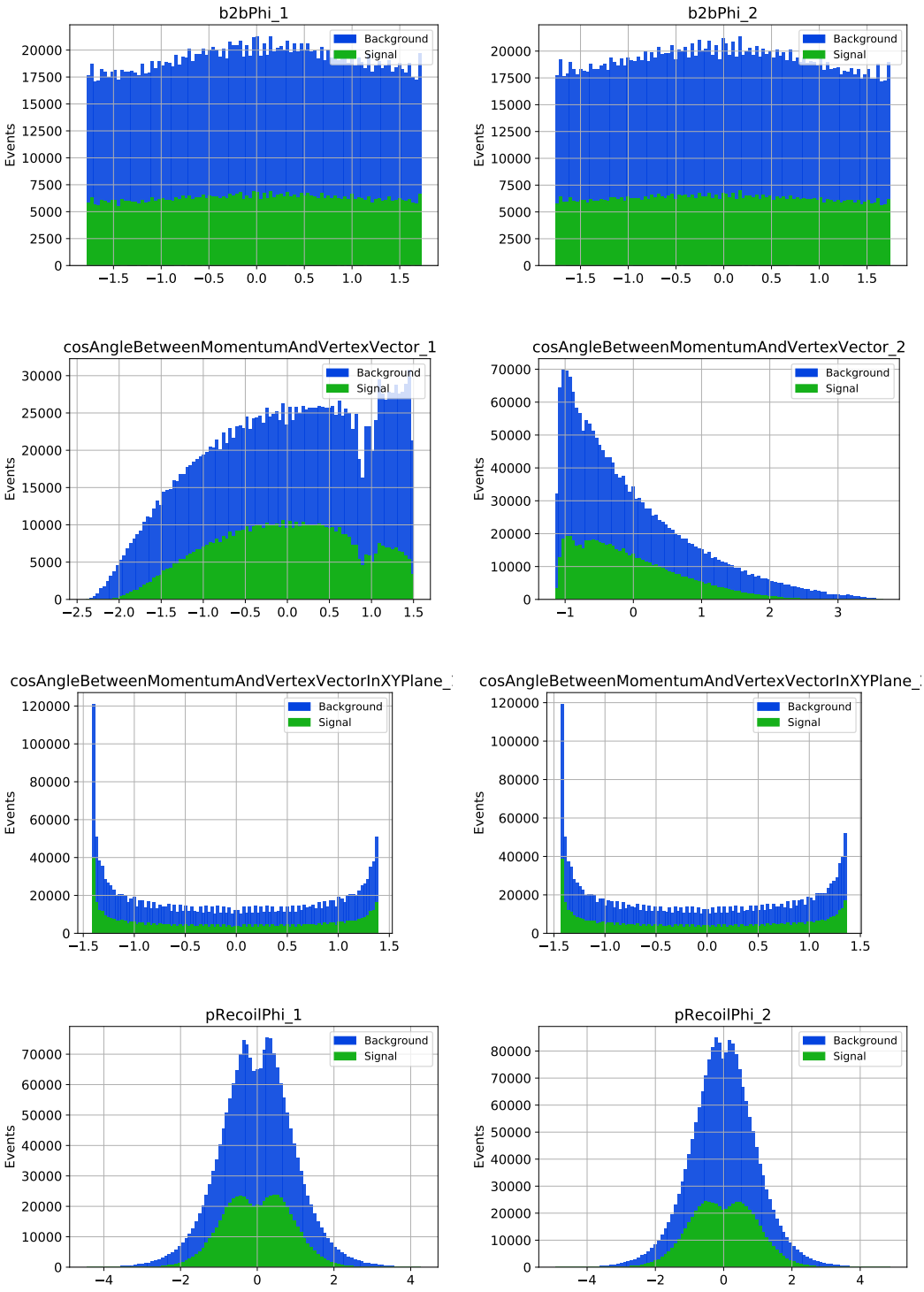


Figure 4.4: Distribution of the used input variable. Hereby the first (..._ 1) and second photons (..._ 2) are seperated.

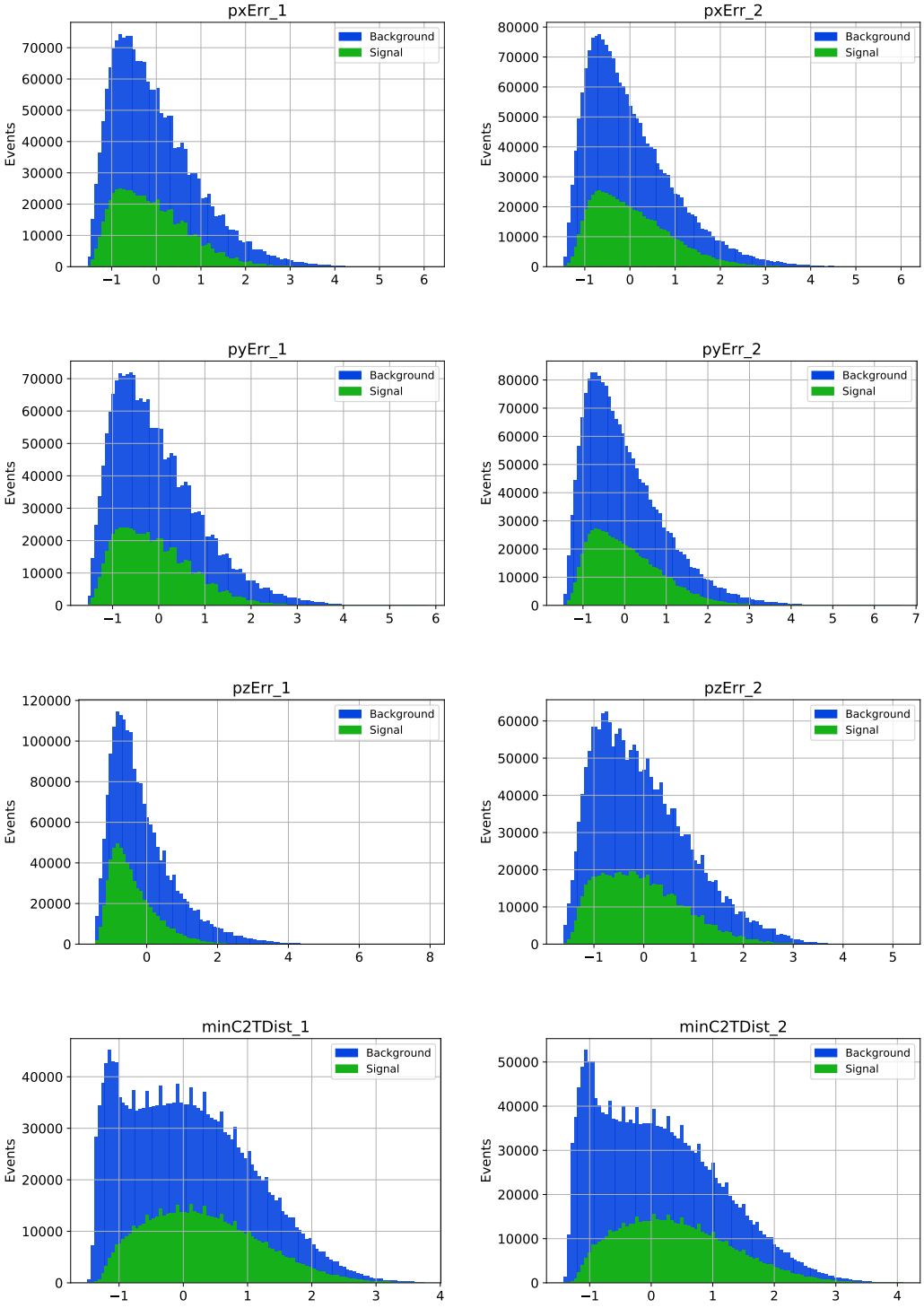


Figure 4.5: Distribution of the used input variable. Hereby the first (..._ 1) and second photons (..._ 2) are seperated.

Bibliography

[1] M. Tanabashi et al. “Review of Particle Physics”. In: *Phys. Rev. D* 98 (3 Aug. 2018), p. 030001. DOI: 10.1103/PhysRevD.98.030001. URL: https://link.aps.org/doi/10.1103/PhysRevD.98.030001.

[2] Kyohei Atarashi et al. “A Deep Neural Network for Pairwise Classification: Enabling Feature Conjunctions and Ensuring Symmetry”. In: Apr. 2017, pp. 83–95. ISBN: 978-3-319-57453-0. DOI: 10.1007/978-3-319-57454-7_7.

[3] Leonard Koch. “Search for Resonant X(3872) Formation in Electron Positron Annihilations and the Development of a Prototype Data Acquisition for the Crystal Zero Degree Detector at BESIII”. eng. PhD thesis. Otto-Behaghel-Str. 8, 35394 Gießen: Justus-Liebig-Universität, 2019. URL: http://geb.uni-giessen.de/geb/volltexte/2019/14685.

[4] S.-K. Choi et al. “Observation of a Narrow Charmoniumlike State in Exclusive $B^{\pm} \to K^{\pm} \pi^{\pm} + \pi^{-}$ J/Ψ Decays”. In: *Physical Review Letters* 91.26 (Dec. 2003). ISSN: 1079-7114. DOI: 10.1103/physrevlett.91.262001. URL: http://dx.doi.org/10.1103/PhysRevLett.91.262001.

[5] J.C. MacKay David. “Information Theory, Inference, and Learning Algorithms”. In: Cambridge University Press, 2003. URL: http://www.inference.org.uk/itprnn/book.pdf.

[6] B. Mehlig. “Artificial Neural Networks”. In: *CoRR* abs/1901.05639 (2019). arXiv: 1901.05639. URL: http://arxiv.org/abs/1901.05639.

[7] John Bishop. “HISTORY AND PHILOSOPHY OF NEURAL NETWORKS”. In: Jan. 2015, pp. 22–96. ISBN: 978-1780215204.

[8] Jörn Bleck-Neuhaus. *Elementare Teilchen*. 2nd ed. Springer Spektrum, 2013. ISBN: 978-3-642-32578-6. DOI: 10.1007/978-3-642-32579-3.

[9] T. Abe et al. *Belle II Technical Design Report*. 2010. arXiv: 1011.0352.

[10] E Kou et al. "The Belle II Physics Book". In: *Progress of Theoretical and Experimental Physics* 2019.12 (Dec. 2019). ISSN: 2050-3911. DOI: 10.1093/ptep/ptz106. URL: http://dx.doi.org/10.1093/ptep/ptz106.

[11] Christian Pulvermacher. "dE/dx particle identification and pixel detector data reduction for the Belle II experiment". PhD thesis. Karlsruhe Institute of Technology (KIT), 2012.

[12] *Super KEKB and Belle II*. Belle II. 2020. URL: https://www.belle2.org/project/super_kekb_and_belle_ii (visited on 07/15/2020).

[13] Christian Pulvermacher. "Analysis Software and Full Event Interpretation for the Belle II Experiment". PhD thesis. 2015. DOI: 10.5445/IR/1000050704.

[14] Ilya Komorov. "Tutorial: Collaborative services 101 & introduction to basf2". 2019 Belle II Summer School. 2019. URL: https://indico.bnl.gov/event/5655/contributions/29864/attachments/23926/35034/Introduction_to_basf2.pdf.

[15] T. Kuhr et al. "The Belle II Core Software". In: *Comput. Softw. Big Sci.* 3.1 (2019), p. 1. DOI: 10.1007/s41781-018-0017-9. arXiv: 1809.04299 [physics.comp-ph].

[16] Peter Zhang. "Neural Networks for Classification: A Survey". In: *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 30 (Dec. 2000), pp. 451–462. DOI: 10.1109/5326.897072.

[17] *scikit-learn*. 2020. URL: scikit-learn.org/ (visited on 08/22/2020).

[18] *Tensorflow*. Google. 2020. URL: https://www.tensorflow.org (visited on 08/22/2020).

[19] *Keras*. 2020. URL: https://keras.io (visited on 08/22/2020).

[20] *Jupyter*. 2020. URL: https://jupyter.org (visited on 08/22/2020).

[21] Cyril Goutte and Eric Gaussier. "A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation". In: vol. 3408. Apr. 2005, pp. 345–359. DOI: 10.1007/978-3-540-31865-1_25.

[22] Saito T and Rehmsmeier M. “The precision-recall plot is more informative than the ROC plot when evaluating binary classifiers on imbalanced datasets”. In: Mar. 2015. ISBN: 978-3-319-57453-0. DOI: doi:10.1371/journal.pone.0118432.

[23] Jesse Davis and Mark Goadrich. “The Relationship Between Precision-Recall and ROC Curves”. In: vol. 06. June 2006. DOI: 10.1145/1143844.1143874.

[24] Sreerama K. Murthy. “Automatic construction of decision trees from data: a multi-disciplinary survey”. In: *Data Mining and Knowledge Discovery* 41.2 (1998), pp. 345–389. URL: https://hal.archives-ouvertes.fr/hal-00442435.

[25] Jalil Kazemitabar et al. “Variable Importance Using Decision Trees”. In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 426–435. URL: http://papers.nips.cc/paper/6646-variable-importance-using-decision-trees.pdf.

[26] *Decision Tree Hugging*. towards data science. June 6, 2019. URL: https://towardsdatascience.com/decision-tree-hugging-b8851f853486 (visited on 07/17/2020).

[27] *Decision-Tree Learning*. Techical University of Darmstadt. URL: https://www.ke.tu-darmstadt.de/lehre/archiv/ws0809/mldm/dt.pdf (visited on 07/18/2020).

[28] Misha Denil, David Matheson, and Nando Freitas. “Narrowing the Gap: Random Forests In Theory and In Practice”. In: *31st International Conference on Machine Learning, ICML 2014* 2 (Oct. 2013).

[29] Leo Breiman. “Random Forests”. In: *Machine Learning* 45 (Oct. 2001). ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324.

[30] Shubair Abdulla and Ahmed Alashoor. “An Artificial Deep Neural Network for the Binary Classification of Network Traffic”. In: *International Journal of Advanced Computer Science and Applications* 11 (Jan. 2020). DOI: 10.14569/IJACSA.2020.0110150.

[31] Luca Bertinetto et al. “Fully-Convolutional Siamese Networks for Object Tracking”. In: *CoRR* abs/1606.09549 (2016). arXiv: 1606.09549. URL: http://arxiv.org/abs/1606.09549.

[32] Kyohei Atarashi et al. “A Deep Neural Network for Pairwise Classification: Enabling Feature Conjunctions and Ensuring Symmetry”. In: *Advances in Knowledge Discovery and Data Mining*. Ed. by Jinho Kim et al. Cham: Springer International Publishing, 2017, pp. 83–95. ISBN: 978-3-319-57454-7.

[33] DorisYangsoo Kim. “The software library of the Belle II experiment”. In: *Nuclear and Particle Physics Proceedings* 273-275 (2016). 37th International Conference on High Energy Physics (ICHEP), pp. 957–962. ISSN: 2405-6014. DOI: https://doi.org/10.1016/j.nuclphysbps.2015.09.149. URL: http://www.sciencedirect.com/science/article/pii/S2405601415006380.

[34] David Lange and Anders Ryd. “EvtGen tutorial”. Decays. 2003. URL: https://indico.cern.ch/event/411269/contributions/1867718/attachments/835829/1159322/tut-all.pdf.

[35] *Faraway, Julian J.* 2002. URL: https://cran.r-project.org/doc/contrib/Faraway-PRA.pdf (visited on 07/18/2020).

[36] M. H. Kutner, C. J. Nachtsheim, and J. Neter. *Applied Linear Regression Models*. 4th ed. McGraw-Hill Irwin, 2004.

Acknowledgments

I would like to thank apl. Prof. Dr. Jens Sören Lange for the opportunity to work in his research group, and together with Klemens Lautenbach and Leonard Koch, for introducing me into the topic, proving data and examples, proofreading my thesis, and giving helpful feedback. The welcoming and supportive atmosphere helped me during writing my thesis and made it very pleasant.

In addition, I would like to thank my family which supported me and helped me to focus on my thesis. Furthermore, I would like to thank Irene Allmansberger. Your support and patience helped me throughout my thesis.

Selbstständigkeitserklärung
(Statement of originality)

Hiermit versichere ich, die vorgelegte Thesis selbstständig und ohne unerlaubte fremde Hilfe und nur mit den Hilfen angefertigt zu haben, die ich in der Thesis angegeben habe. Alle Textstellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen sind, und alle Angaben die auf mündlichen Auskünften beruhen, sind als solche kenntlich gemacht. Bei den von mir durchgeführten und in der Thesis erwähnten Untersuchungen habe ich die Grundsätze guter wissenschaftlicher Praxis, wie sie in der 'Satzung der Justus-Liebig-Universität zur Sicherung guter wissenschaftlicher Praxis' niedergelegt sind, eingehalten. Gemäß §25 Abs. 6 der Allgemeinen Bestimmungen für modularisierte Studiengänge dulde ich eine Überprüfung der Thesis mittels Anti-Plagiatssoftware.

. . . .	. . . .
Datum und Ort	Unterschrift