

UNIVERSITY OF GENEVA

MASTER THESIS

---

# Track Reconstruction for a High-Luminosity LHC Detector using Deep Neural Networks

---

*Author:*

Luiza Adelina CIUCU

*Supervisor:*

Dr. Tobias GOLLING

*A thesis submitted in fulfilment of the requirements  
for the degree of Master of Particle Physics*

*in the*

Prof. Golling Research Group  
Department of Physics

March 6, 2021



UNIVERSITY OF GENEVA

*Abstract*

Faculty of Science  
Department of Physics

Master of Particle Physics

**Track Reconstruction for a High-Luminosity LHC Detector using Deep Neural Networks**

by Luiza Adelina CIUCU

In the context of the ATLAS detector upgrade for the Run-4, the number of collisions per bunch crossing (pile-up) is increased to about 200. The overlapping particle tracks make it very difficult to reconstruct all particle tracks. Alternative reconstruction methods become necessary. In this thesis a deep neural network (DNN) is used on simulated data from LHC experiments. Tracks are reconstructed from groups of 20 particle hits in the detector, called buckets. The DNN input is the hit spatial coordinates  $x$ ,  $y$ ,  $z$ . In every bucket hits belonging to the particle with the largest number of hits in the bucket have an output label of 1, the rest an output label of -1. This situation corresponds to a machine learning multi-label problem, namely to predict for each hit in the bucket an output value of 1 or -1. The model hyper-parameters are optimised. Various models are compared with various figures of merit. Overall, an efficiency of particle reconstruction of 71.3% is obtained.



## *Acknowledgements*

This project would not have been possible without the continuous support, encouragement and feedback of my supervisor Prof. Tobias Golling and my senior colleague and Ph.D. student Sabrina Amrouche. I would also like to express my gratitude to the ATLAS group of the University of Geneva for having welcomed me, including Prof. Giuseppe Iacobucci and Anna Sfyrla. I would also like to thank the professors from my lectures of the University of Geneva, who offered me a strong theoretical and practical foundation in particle physics.



# Contents

<b>Abstract</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Standard Model . . . . .	1
1.2 Limitations of the Standard Model . . . . .	2
1.3 Experimental Setup . . . . .	3
1.4 Proton Collisions . . . . .	4
1.5 Tracking . . . . .	5
<b>2 TrackML</b>	<b>7</b>
2.1 TrackML Challenge . . . . .	7
2.2 Dataset . . . . .	7
2.3 Simulated Truth Particles and Reconstructed Hits . . . . .	9
2.4 Detector Components . . . . .	10
<b>3 Problem Setting</b>	<b>13</b>
3.1 Question . . . . .	13
3.2 Approximate Nearest Neighbours . . . . .	13
3.3 Buckets . . . . .	14
3.4 Multi-Label Binary Classification . . . . .	15
<b>4 Machine Learning</b>	<b>17</b>
4.1 Introduction . . . . .	17
4.2 Neural Network Architecture . . . . .	17
4.3 Hyper-Parameters . . . . .	19
4.4 Learning Methods . . . . .	22
4.5 Train and Test Split and k-fold . . . . .	22
4.6 Balancing Datasets . . . . .	23
4.7 Hyper-Parameter Tuning . . . . .	23
4.7.1 DNN Architecture . . . . .	24
4.7.2 DNN Learning . . . . .	26
4.7.3 Best Model . . . . .	27
4.8 Predicting or Inference and Figures of Merit . . . . .	28
4.9 Software Used . . . . .	29
<b>5 Model Performance</b>	<b>31</b>
5.1 Training Metrics . . . . .	31
5.2 Hit-Level Metrics . . . . .	31
5.3 Particle-Level Metrics . . . . .	33

<b>6 Conclusions</b>	<b>37</b>
6.1 Conclusions . . . . .	37
6.2 Future Plans . . . . .	38
<b>7 Pseudo-Code</b>	<b>39</b>
7.1 Input and Output Preparation . . . . .	39
7.1.1 Algorithm 1 . . . . .	39
7.1.2 Algorithm 2 . . . . .	41
7.1.3 Algorithm 3 . . . . .	43
7.2 Model Evaluation Metrics . . . . .	43

# List of Figures

1.1	The elementary particles of the Standard Model. . . . .	2
1.2	The LHC Accelerator Complex System. . . . .	3
1.3	The ATLAS detector and its sub-detectors. . . . .	4
1.4	The LHC pile-up ( $\mu$ ) increased during Run-2 [3]. . . . .	4
1.5	Reconstructing one track from hits in a general detector. In green a good reconstruction, in orange two bad reconstructions [4]. . . . .	5
1.6	Tracking for a general detector illustrated by reconstructing tracks (orange lines to the right) from hits (black individual 3D points to the left) [5] [6]. . . . .	5
1.7	Left: Diagram of the collision of proton bunches producing several vertices: one primary vertex from a rare interesting particle, and several pile-up vertices [7]. Right: example of several vertices reconstructed in the ATLAS detector [4]. . . . .	6
1.8	The CPU consumptions increases more than quadratically with both pile-up (left) and years (right) [5] [6]. . . . .	6
2.1	Tracks are reconstructed in the inner detector, which is simulated formed of three types of silicon detectors. From beam pipe outwards the pixels, short strips and long strips [4]. . . . .	8
2.2	The three detectors are organised in volumes, layers and modules [4]. . . . .	9
2.3	Distribution of the number of hits in truth particles in event 99. . . . .	9
2.4	Four truth particle tracks from event 99 travelling in all possible direction shown in the $r - z$ plane. The points are fit with a straight line of equation $r = a_0 + a_1 \cdot x$ . $a_0$ is the intercept and $a_1$ is the slope. $z_0$ is the $z$ coordinate when the radius $r$ is zero. . . . .	10
2.5	Distributions of reconstructed minus truth values for $x, y, z$ in mm for all truth particles in event 99. . . . .	10
2.6	$x, y, r, z$ distributions in mm for all truth particles in event 99. . . . .	11
2.7	The percentage of hits from 100 events in each volume ID and layer ID. . . . .	11
2.8	Distributions of reconstructed hits in the various volume ID, layer ID and module ID, for all truth particles in event 99. . . . .	12
2.9	2D scatter plots of the reconstructed hits between the volume ID, module ID versus $r$ and $z$ , for all truth particles in event 99. . . . .	12
3.1	There are two steps in the ApproximateNearestNeighbours. First create an index from the positions of all hits in the event. Then query hits (even in parallel) to find the nearest neighbours to each hit. . . . .	14
3.2	The distribution of the majority particle size, or nbPositiveHit. . . . .	15
3.3	Overlay of the number of positive hits per bucket by default (Min00) and after moving all hits to negative for buckets with less than 10 number of positive hits (Min10). Train (left) vs Test (right). . . . .	15
4.1	Diagram of a general architecture of a DNN. Credit image: O'Reilly. [19]. . . . .	18

4.2	Diagram of a neuron or node in a NN, with its weight inputs, its bias and the output via the activation function. Credit image: The Fork [20]. . . . .	18
4.3	Overlaid potential activation functions for the last layer. Left: hyperbolic tangent (tanh) and logistic regression (sigmoid). Right: tanh, square non linear and soft sign. Tanh is chosen as our output labels are -1.0 and 1.0. . . . .	20
4.4	Overlaid loss functions of (regular) hinge and squared hinge for varying predicted y, for a fixed true y of -1.0 (left) and 1.0 (right). . . . .	20
4.5	Overlaid activation functions of ReLU and ELU, with $\alpha = 1.0$ . . . . .	21
4.6	Comparison of different numbers of hidden layers. 3 hidden layer is best. Binary accuracy and loss in Train and Test balanced samples. . . . .	24
4.7	Comparison of the ratio of the different number of nodes on the hidden layers divided by the number of nodes on the last layer. k=10, or 200 nodes on each hidden layer, is best. Binary accuracy and loss in Train and Test balanced samples. . . . .	24
4.8	Comparison of ReLU and ELU activation functions on the hidden layers. ReLU is chosen for the final model. Binary accuracy and loss in Train and Test balanced samples. . . . .	25
4.9	Comparison without and with a dropout layer for regularisation. A dropout layer is used in the final model. Binary accuracy and loss in Train and Test balanced samples. . . . .	25
4.10	Comparison of TANH, SQNL and SOSI as activation functions on the last layer. TANH is used in the final model. Binary accuracy and loss in Train and Test balanced samples. . . . .	26
4.11	Comparison of Adam and AdaDelta optimizers for the learning method. Adam is used in the final model. Binary accuracy and loss in Train and Test balanced samples. . . . .	26
4.12	Comparison of regular and square hinge loss functions for DNN learning. Regular hinge is used in the final model. Binary accuracy and loss in Train and Test balanced samples. . . . .	27
4.13	Comparison of various batch sizes for DNN learning. A batch size of 50000 is used in the final model. Binary accuracy and loss in Train and Test balanced samples. . . . .	27
5.1	Binary accuracy and loss, as resulted directly from training in Keras and Tensorflow. Note that while Train is balanced, Test is also balanced, as it would be too slow to have it unbalanced. . . . .	32
5.2	Output and output predicted 1D. Older ones have a sharp peak, the new ones are more flat. . . . .	33
5.3	Output and output predicted 2D. In Train balanced, a diagonal is seen. In Test unbalanced it's harder to see. . . . .	34
5.4	Accuracy, precision, recall, predicted output negative, true negative rate. Train (left) and Test (right). . . . .	35

# List of Tables

4.1 Confusion Matrix . . . . .	28
5.1 Particle reconstruction efficiency results . . . . .	33



## Chapter 1

# Introduction

Elementary particle physics is the field of physics that describes the elementary particles and their interactions. The theoretical model denoted the Standard Model was created about 50 years ago and is a very successful theory. All its predictions have been confirmed experimentally, including the discovery of the Higgs boson in 2012 at CERN. But theorists believe there are phenomena predicted by theories beyond the Standard Model that may be discovered at CERN in the next decade in the future data taking runs, Run-3 and Run-4, at the Large Hadron Collider (LHC). There are four major experiments, two of which are generalistic: ATLAS and CMS. These experiments hope to discover new phenomena like Super-Symmetry and Dark Matter. These detectors are huge digital cameras in three dimensions (3D) made of three different layers. Inner detectors reconstruct tracks of charged particles and measure their momenta via ionization energy. Calorimeters measure the total energy of both charged and neutral particles in a destructive process. The last layer is the muon detector measuring tracks of muons as minimum ionization particles.

To maximize the probability to see rare processes, the LHC collides bunches of protons at ever increasing number of proton pairs that collide in one bunch crossing ( $\mu$ ). The higher  $\mu$ , the busier the collision (event) and the harder to reconstruct the particles. The latest value of  $\mu$  in Run-2 ended in 2018 is about 36 and in Run-4 a value of 200 is expected. The potential combinatorics of the event requires a significantly larger computing power than the allowed budget. The solution is to reconstruct tracks via new machine learning techniques. This is the problem addressed in this thesis. A public simulated dataset for a general purpose detector at  $\mu=200$  produced for the TrackML Challenge is analyzed in this project via a deep neural network and an approximate nearest neighbour technique.

The Sections 1.1 and 1.2 describe the Standard Model and its limitations. The Section 1.3 describes the experimental setup, the Section 1.4 describes the proton collisions, and the Section 1.5 describes the tracking reconstruction.

### 1.1 Standard Model

The Standard Model (SM) is a theory that describes the elementary particle (fermions and bosons) and their interactions via three fundamental forces. The SM elementary particles are illustrated in Figure 1.1 and are described in more detail below.

There are 12 fermions representing the constituents of matter. They have a semi-integer spin (1/2) and they can be divided into two categories: quarks and leptons. The quarks have fractional electric charges and form baryons and mesons. The six quarks are up ( $u^{+2/3}$ ), down ( $d^{-1/3}$ ), charm ( $c^{+2/3}$ ), strange ( $s^{-1/3}$ ), top ( $t^{+2/3}$ ), and bottom ( $b^{-1/3}$ ). The three electrically-charged leptons have a negative charge: the electron ( $e^-$ ), the muon ( $\mu^-$ ), and the tau lepton ( $\tau^-$ ). These have corresponding neutrally-charged leptons called neutrinos

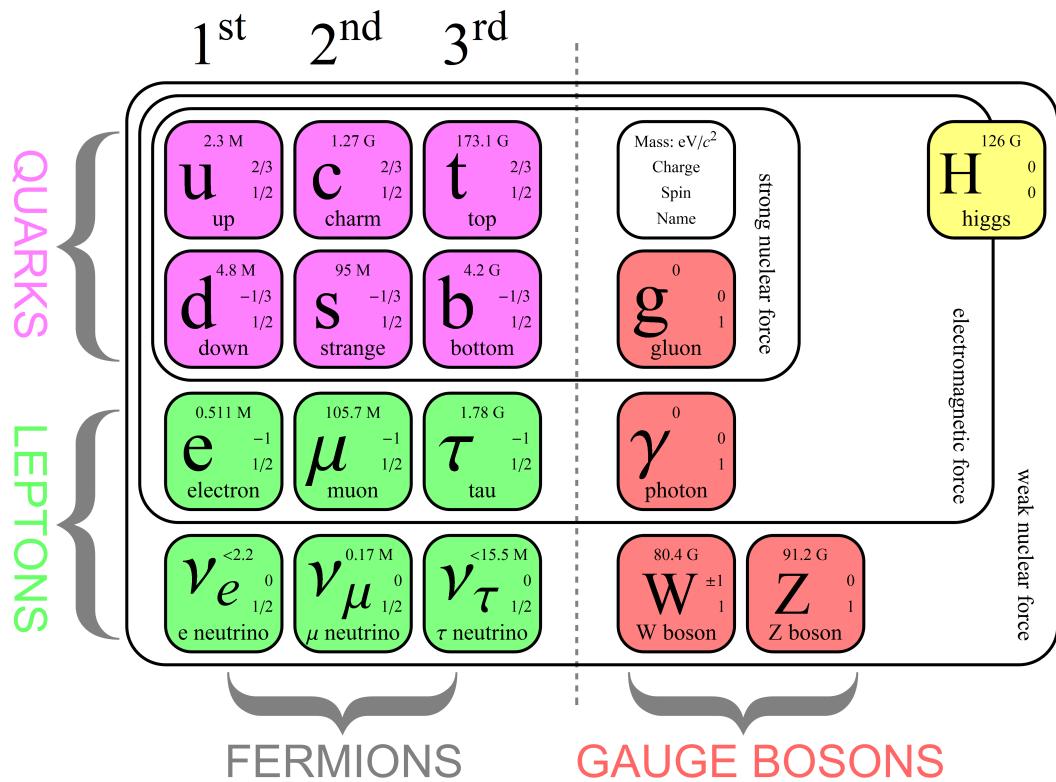


FIGURE 1.1: The elementary particles of the Standard Model.

( $\nu_e^0, \nu_\mu^0, \nu_\tau^0$ ). These particles form matter. For each matter particle there is an anti-matter particle that has an opposite electric charge.

Fermions interact with each other via the exchange of other elementary particles called bosons. Bosons are carriers of the elementary forces and have an integer spin (1). There are eight type of gluons ( $g$ ) that carry the strong force. The photon ( $\gamma$ ) is responsible for the electromagnetic force. The  $W^+, W^-$  and the  $Z^0$  bosons carry the weak force.

There is also another type of boson, a scalar elementary particle of spin zero (0), called the Higgs boson. The Higgs boson is the latest elementary particle discovered, in 2012, by the ATLAS and CMS collaborations at CERN. It is predicted by the mechanism that explains how the elementary particles acquire mass.

## 1.2 Limitations of the Standard Model

Although all particles predicted by the Standard Model (SM) have been discovered, there are phenomena not yet explained by the SM.

The matter-antimatter asymmetry is not yet understood. It is believed in the Big Bang equal quantities of matter and antimatter were produced. But the observable Universe seems to be created only of matter.

The content of matter-energy of the Universe is formed by only 5% of regular baryonic matter. About 25% is represented by dark matter, an unknown form of matter that interacts only very weakly with regular matter. It is thought this matter is key to the evolution of the Universe.

Overall, it is believed that the Standard Model is in fact only a low-energy approximation of a higher-energy theory. New particles and interactions are predicted by a variety of models of physics beyond the Standard Model (BSM). Such models have already been excluded by current searches at CERN. The search for new physics will continue in Run-3 and Run-4 at the LHC.

### 1.3 Experimental Setup

At the moment, the Large Hadron Collider (LHC), which is situated at CERN, is the most powerful proton-proton collider in the world. After the Tevatron and the Large Electron-Positron Collider (LEP) era, a new machine was needed for new discoveries in particle physics. The LHC was designed to achieve a center-of-mass energy  $\sqrt{s} = 14$  TeV. Two of the biggest goals of the LHC are to study and test the SM and to search for new physics beyond the SM (BSM). The ATLAS is the largest of several detectors of the LHC accelerator complex, as illustrated in Figure 1.2.

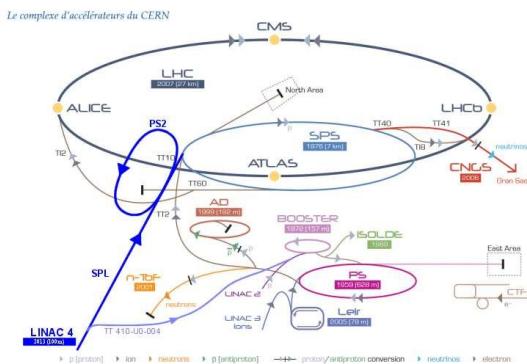


FIGURE 1.2: The LHC Accelerator Complex System.

This project is affiliated to the international collaboration of the A Toroidal LHC ApparatuS (ATLAS) [1][2]. ATLAS is the largest of the four LHC detectors. ATLAS is one of the two general-purpose particle physics detectors at the LHC, the other being CMS. The two detectors and collaborations perform a similar research program. A new discovery must be observed by both detectors to be believed as true.

ATLAS is a cylindrical detector around the beam that collides protons beams. It is formed of four main subdetectors. The closest to the beam is the inner detector (ID). It measures the momentum vector of charged particles. These ionise the gas of the ID, an electric current is measured giving the position of a particle in the detector, also called a *hit*. A collection of hits forms a *track*. The particle travels further in the calorimeter, which measures the energy of the particle in a destructive way. It is formed of two parts. First there is the electromagnetic calorimeter (EMCal), which measures the energy of electrons, positrons and photons. Then the hadronic calorimeter (HadCal), which measures the energy of hadrons, originated from quarks and gluons. Several hadrons travelling together after having originated in the same particle are called *jets*. Muons deposit very little energy in the calorimeters, being minimum ionising particles. The fourth subdetector detects energy depositions of muons and thus measures their momentum. These subdetectors of ATLAS are illustrated in Figure 1.3.

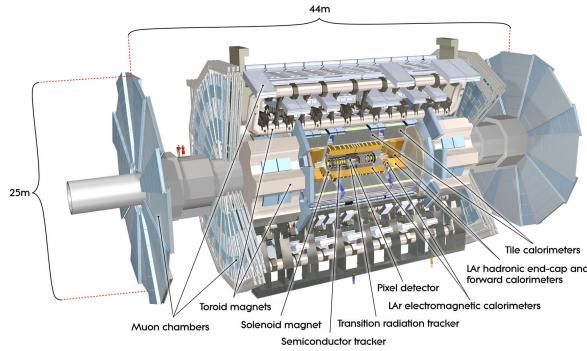


FIGURE 1.3: The ATLAS detector and its sub-detectors.

## 1.4 Proton Collisions

The LHC collides proton beams head-on. The beam is not continuous. Instead, the protons are grouped into up to 2808 bunches. Each bunch contains about 115 billion protons. Proton bunches collide every 25 nanoseconds (ns), at a bunch collision rate of 40 MHz. The number of proton collisions during a bunch crossing is called *pile-up* and is denoted by  $\mu$ . The data from the proton collisions form a bunch crossings are recorded together in an *event*. Usually only one of the collisions produces a rare interesting particle, such as a top quark, a W, Z, or Higgs boson, or BSM particles like supersymmetric candidates or dark matter. The rest of the collisions represent a background for the rare one. The rare collision is called *hard scatter* (HS) and the rest of the collisions are called *pile-up* collisions (PU). The event reconstruction in general and charged-particle track reconstruction in particular becomes harder with increasing  $\mu$ .

Yet increasing  $\mu$  is exactly the strategy employed at the LHC Run-1 and Run-2, in order to increase the collision luminosity and thus the probability to produce rare particles. The  $\mu$  average values during Run-2 were about 13, 25, 38, 36 in 2015, 2016, 2017 and 2018, respectively, as seen in Figure 1.4 [3]. In Run-3 and Run-4, the pile-up will be increased even further to  $\mu = 200$ .

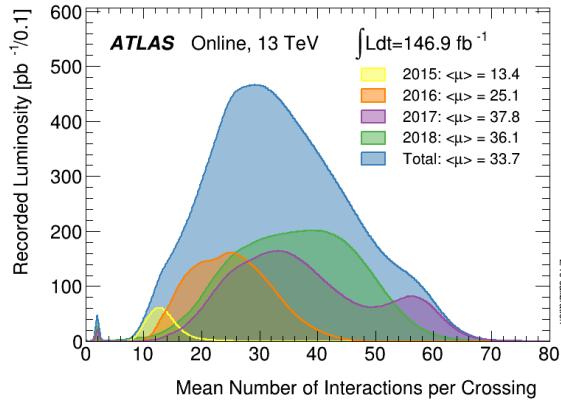


FIGURE 1.4: The LHC pile-up ( $\mu$ ) increased during Run-2 [3].

## 1.5 Tracking

Reconstructing charged particles in the inner detector (ID) by grouping together recorded hits that were produced by the same particle is called *tracking*. The group of hits belonging to the same particle is called a *track*. Tracks are produced only by charged particles that ionize the gas in the ID. The ID is held in a magnetic field, so that positively and negatively charged particles are curved in opposite directions. The radius of the curvature allows the measurement of the particle momentum. Neutral particles, such as photons, neutrinos, neutrons and other neutral hadrons, do not produce tracks. Because of the magnetic field, tracks are geometrically in an ideal case helices pointing approximately to the origin of the primary proton-proton interaction. Reconstructiong one track from hits is illustrated for a general particle detector simulated in the TrackML dataset and used in this project in Figure 1.5 and overall all the hits to tracks in Figure 1.6.

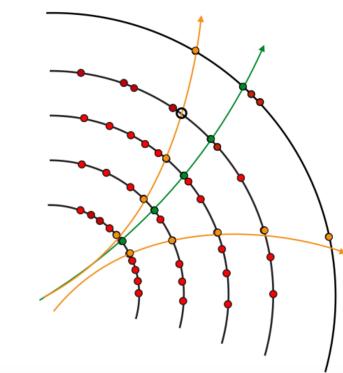


FIGURE 1.5: Reconstructing one track from hits in a general detector. In green a good reconstruction, in orange two bad reconstructions [4].

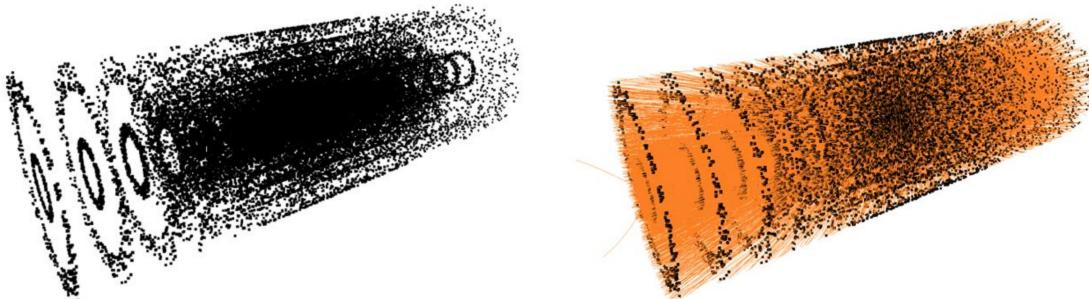


FIGURE 1.6: Tracking for a general detector illustrated by reconstructing tracks (orange lines to the right) from hits (black individual 3D points to the left) [5] [6].

Tracking is a key component of the event reconstruction and is used in several of its steps. First, to reconstruct vertices. Every proton collision in a bunch crossing produces its own particles, out of which the charged ones produce tracks. These tracks are clustered in a vertex. From the several vertices in the event, the most interesting one (usually at higher energy for a rare particle) is called the *primary vertex*, the rest being *pile-up vertices*, as illustrated in Figure 1.7 for the ATLAS detector.

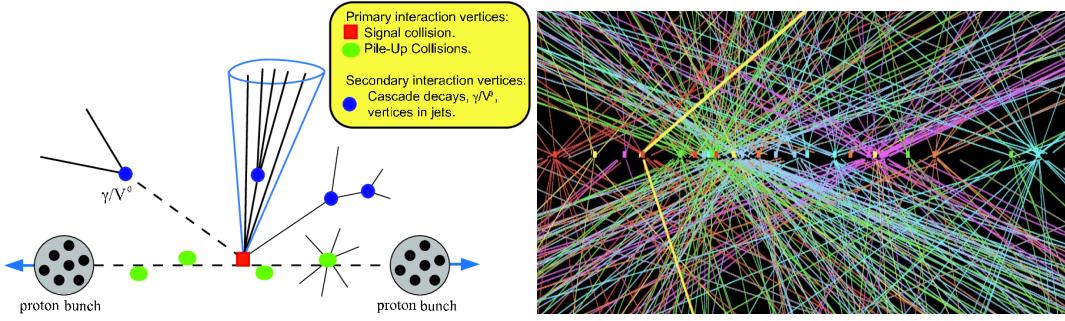


FIGURE 1.7: Left: Diagram of the collision of proton bunches producing several vertices: one primary vertex from a rare interesting particle, and several pile-up vertices [7]. Right: example of several vertices reconstructed in the ATLAS detector [4].

Tracks are essential also to reconstruct charged particles. For example, an electron is reconstructed as a track in the ID, and an electromagnetic shower in the EMCal. A muon is reconstructed as a track in the ID and energy deposits in the muon detector. A charged hadron is reconstructed as a track in the ID and a hadronic shower in HadCal.

Current tracking algorithms employed at ATLAS [8] and CMS [9] use a combinatorial approach, where first a track seed is found and later the track is computed. These algorithms require many calculations and thus a high CPU consumption. There is a stringent need to find algorithms with reduced CPU consumption to be able to scale from an LHC pile-up of about  $\mu = 36$  in Run-2 to  $\mu = 200$  in Run-3 and beyond. As illustrated in Figure 1.8 (left), the CPU consumption increases more than quadratic with  $\mu$ , and tracking represents the largest part of the event reconstruction CPU. Figure 1.8 (right) illustrates how the CPU consumption increases even more for Run-4 and Run-5. Especially for Run-4 in 2026 significant algorithm improvements are needed to improve tracking CPU requirements by a factor of approximately 10 [4].

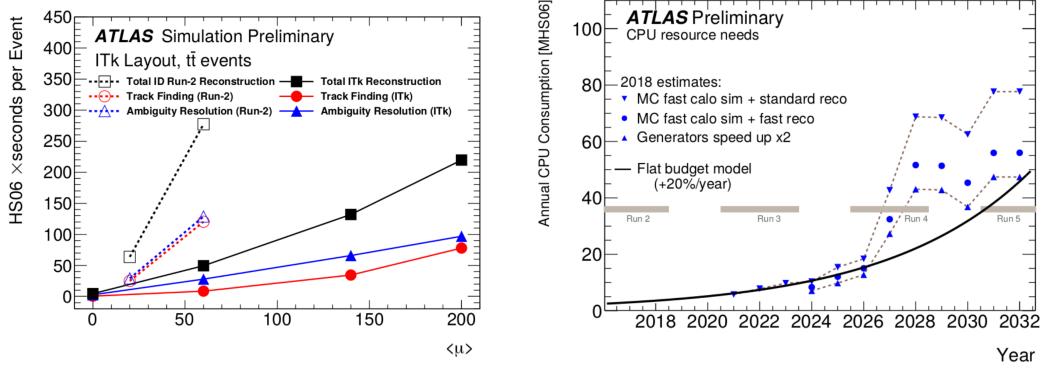


FIGURE 1.8: The CPU consumptions increases more than quadratically with both pile-up (left) and years (right) [5] [6].

A promising avenue are machine learning algorithms. Such algorithms require a lot of training data and need a long time to train (learn), but are usually fast to predict (infer). The field of machine learning has experienced a large growth in last few years. In this thesis a deep neural network algorithm is studied.

## Chapter 2

# TrackML

### 2.1 TrackML Challenge

To bring expertise for particle tracking for Run-3 to CERN from the computer science and machine learning communities, several LHC experiments have worked together to invite machine learning teams from outside CERN to compete in 2018 in the Kaggle challenge called the TrackML Particle Tracking Challenge [10] [4] [5] [6] and a follow-up challenge in 2019 [11].

The candidates were offered a dataset of simulated charged particles in a generic detector, representative of ATLAS and CMS at CERN. The simulation contains both the true and reconstructed position of the track hits, allowing for a labelled dataset on which learning can be performed. The dataset was obtained using a common tracking software framework at CERN called ACTS [12] [13]. Events of top quark production ( $t\bar{t}$  events) were simulated at  $\mu = 200$ , leading to about 10 thousand tracks per event.  $t\bar{t}$  events are known to produce many particles and thus also many tracks.

Tracks are reconstructed in the inner detector, which is simulated formed of three types of silicon detectors, to be representative of both the ATLAS and CMS at the future High Luminosity LHC (HL-LHC). As illustrated in Figure 2.1, they are the pixels, short strips and long strips, in the order of increasing radius.

The coordinate system is right-handed and cartesian. The z-axis points along the beam axis (longitudinal axis). The x-y plane is the transverse plane. The azimuthal angle  $\phi$  with values within  $[0, 2\pi]$  is the angle in the transverse plane to the x-axis. The polar angle  $\theta$  with values within  $[0, \pi]$  is the angle to the z-axis. In particle physics, instead of the angle  $\theta$  often the pseudo-rapidity  $\eta$  is used, where  $\eta = -\ln \tan(\theta/2)$ .

In this coordinate system, the three silicon detectors are presented in the longitudinal plane in the left side of Figure 2.2. The horizontal lines represent the barrel, and the vertical lines represent the two end-caps. The pixel detector is shown in the transverse plane in the right side of 2.2.

### 2.2 Dataset

Ten thousand events were simulated with collisions in the centre of the detector, leading to about 0.1 billion tracks. Each track has about 10 hits, or 3D points, in the simulated detector, leading to a total of 1 billion points, and about 100 GBytes of data [6]. The dataset from TrackML is described in detail in [14] [15]. A subset of 100 events of the TrackML dataset is used in this study.

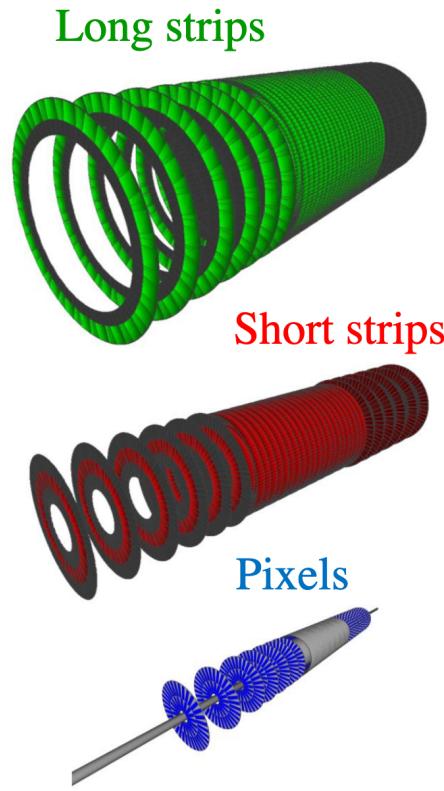


FIGURE 2.1: Tracks are reconstructed in the inner detector, which is simulated formed of three types of silicon detectors. From beam pipe outwards the pixels, short strips and long strips [4].

The detector volumes for barrel and end-caps is divided into unique `volume_id`. Each volume is divided into layers described by `layer_id`, which for technical reasons have only even values. Each layer is divided into modules identified by `module_id`. For each event, four files are provided [14].

The `hit` file contains the reconstructed hit information: `hit_id`, the numerical identifier of the hit within the event; the measured coordinates  $x, y, z$  of the hit in mm, the `volume_id`, the `layer_id`, and the `module_id`.

The `truth` files contains the generated (also called truth) hit information: the `hit_id`, the `particle_id` of the particle that generated this hit, the truth coordinates  $tx, ty, tz$ , and the truth momenta  $tpx, tpy, tpz$ .

The `particles` file contains for each `particle_id` the particle type, its velocities and momenta, the electric charge and the number of hits.

The `cells` file contains for each `hit_id` the cell that recorded the hit. A cell is the smallest unit in a particle detector. A cell is identified uniquely by two channel numbers, similarly to two coordinates of a pixel in an image.

In this thesis only the `hit` and `truth` files of 100 events simulated with  $\mu = 200$  are studied. They are read in as data frames, are concatenated by columns. As a result, for each

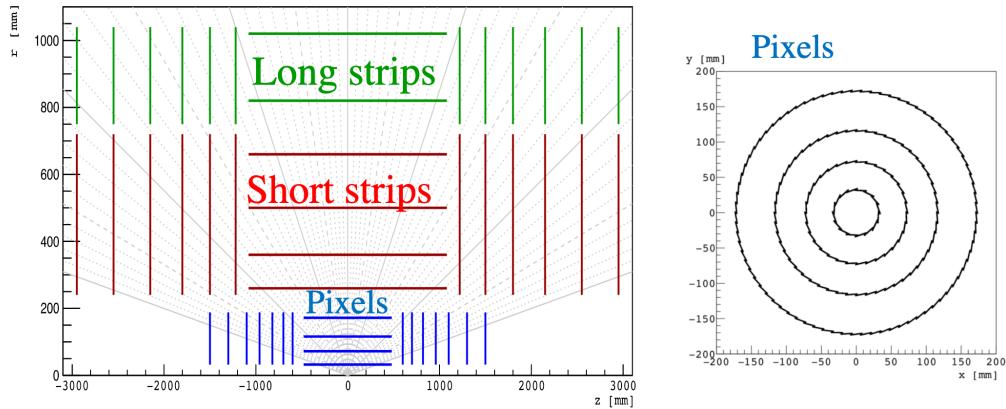


FIGURE 2.2: The three detectors are organised in volumes, layers and modules [4].

`hit_id` one knows the reconstructed coordinates, the truth coordinates, truth momenta and to what `particle_id` it belongs to.

## 2.3 Simulated Truth Particles and Reconstructed Hits

One event with index 99 is studied. Its behaviour is however representative for all events. A typical event produces about 10 thousand simulated particles, with a distribution illustrated in Figure 2.3, with a mean of 10.8 and a standard deviation (rms) of 3.3.

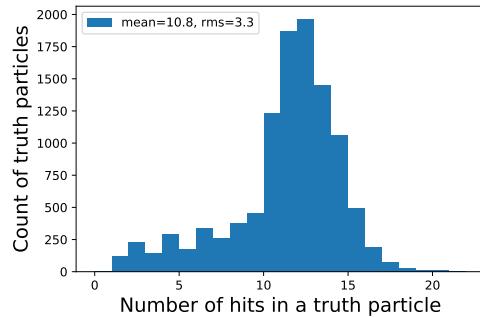


FIGURE 2.3: Distribution of the number of hits in truth particles in event 99.

Particles produced in the collisions in the centre of the detector travel indeed in all possible directions. Four tracks made of hits for truth particles, going to both left and right, at an angle closer to the transverse plane, or to the z axis, are shown in Figure 2.4. The images confirm that the hits for each particle are grouped a line. The z and radius  $r$  coordinates are fit to a line of equation ( $r = a_0 + a_1 \cdot x$ ).  $a_0$  represents the intercept, or the radius when  $z = 0$ , at the centre of the detector. The values of  $a_0$  that are close to zero are consistent with the particles being emitted from the centre of the detector.  $a_1$  represents the slope of the line. The fact that  $a_1$  have different values, both positive and negative, show that particles are emitted in all directions.  $z_0$  represents the z position when the radius  $r = 0$ , meaning when the particle line intersect the z axis. The fact that the  $z_0$  values are close to zero is consistent again with the particles being emitted in the centre of the detector.

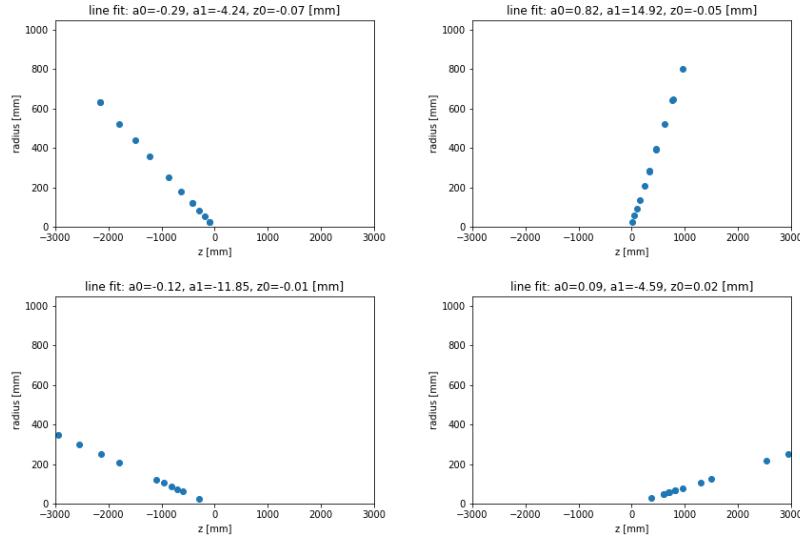


FIGURE 2.4: Four truth particle tracks from event 99 travelling in all possible direction shown in the  $r - z$  plane. The points are fit with a straight line of equation  $r = a_0 + a_1 \cdot x$ .  $a_0$  is the intercept and  $a_1$  is the slope.  $z_0$  is the  $z$  coordinate when the radius  $r$  is zero.

The reconstructed hit coordinates  $x, y, z$  are very close to the corresponding truth values, as illustrated in Figure 2.5. The reconstructed scale and resolution are therefore very good.

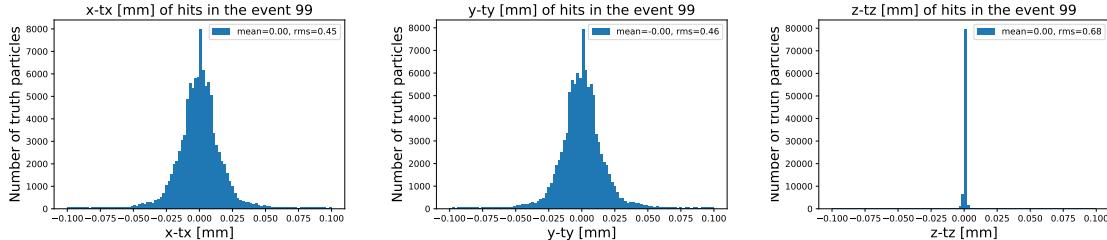
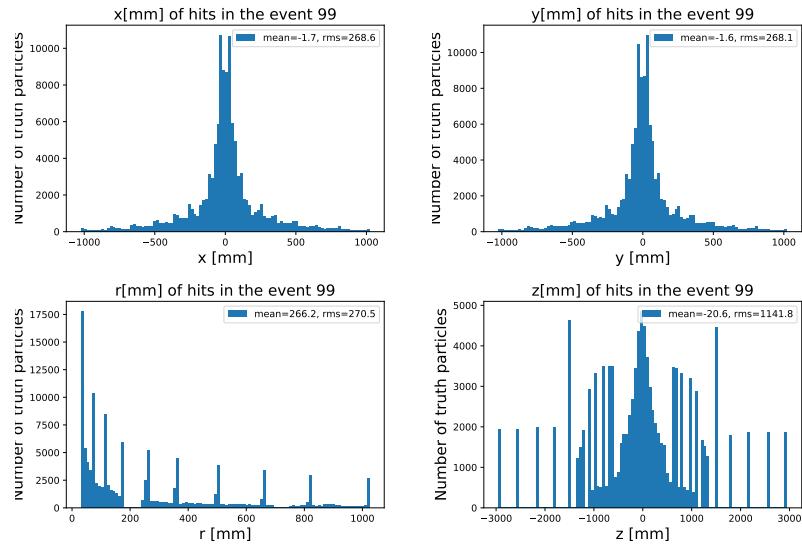


FIGURE 2.5: Distributions of reconstructed minus truth values for  $x, y, z$  in mm for all truth particles in event 99.

Their coordinates, plus the  $r$  coordinate, are illustrated in Figure 2.6.

## 2.4 Detector Components

The grouping of the hits inside the detector is used later in this study to evaluate the performance of the model in each detector sub-volume called `volume_id`. The detector is formed by a central barrel and two end-caps. There are three layers of volumes from the beam onwards. The volumes from the barrel (end-caps) have the modules aligned horizontally (vertically). Most hits are in the volumes of the inner layer, and in those of the barrel. This is consistent with hard scatter collisions emitted mostly at high  $p_T$ , so close to the transverse plane. The `volume_id` numbers are visualised in Figure 2.7, along with the percentage of the hits in each `volume_id`, as measured in 100 simulated event in this study. Also Figure 2.8 illustrates the clustering of hits by `volume_id`, `layer_id`, and `module_id`. The 2D scatter plots

FIGURE 2.6:  $x, y, r, z$  distributions in mm for all truth particles in event 99.

between the `volume_id` and the `module_id` on one side and the  $r$  and  $z$  on the other side are illustrated in Figure 2.9.

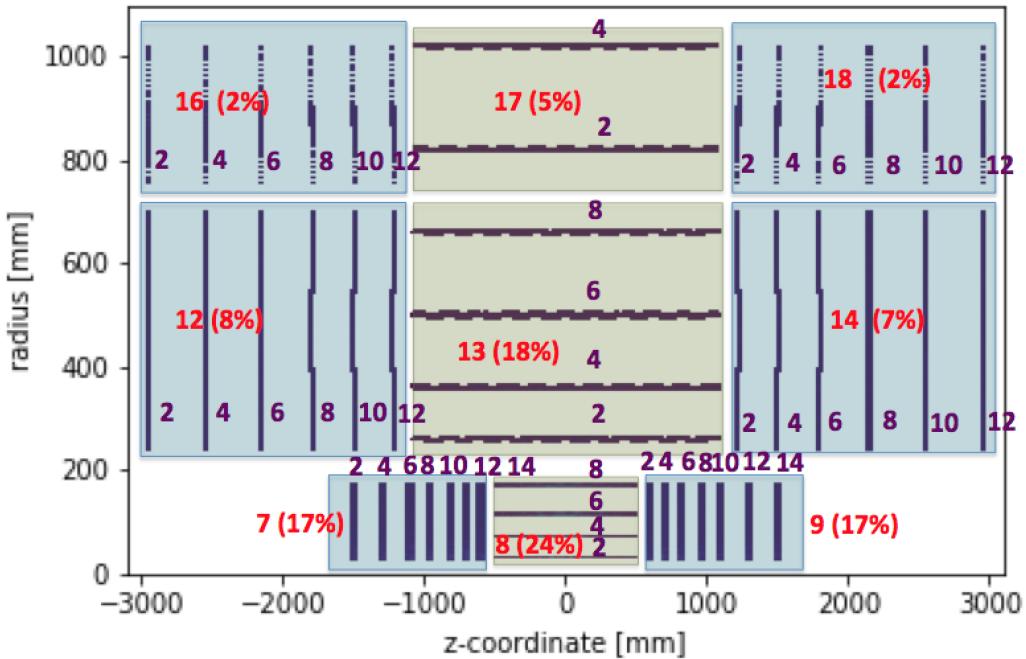


FIGURE 2.7: The percentage of hits from 100 events in each volume ID and layer ID.

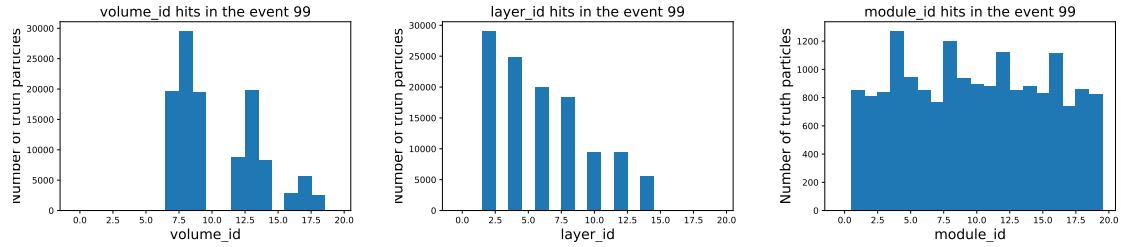


FIGURE 2.8: Distributions of reconstructed hits in the various volume ID, layer ID and module ID, for all truth particles in event 99.

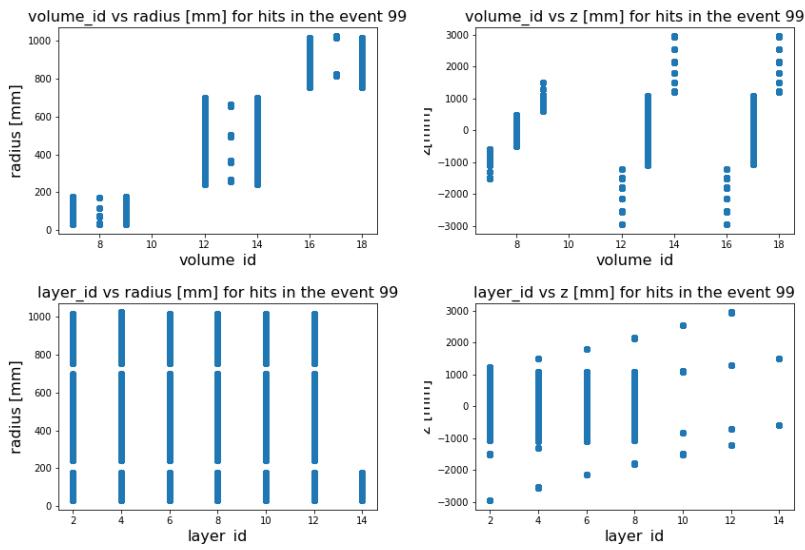


FIGURE 2.9: 2D scatter plots of the reconstructed hits between the volume ID, module ID versus  $r$  and  $z$ , for all truth particles in event 99.

## Chapter 3

# Problem Setting

In this study a machine learning algorithm is developed to reconstruct tracks from reconstructed hits in an event. The dataset used is comprised of 100 events from the TrackML challenge, described in Chapter 2. This chapter describes what is processed for a given event, how are the input and output datasets formed, and how is the question formulated.

### 3.1 Question

Ideally, one would want like to take as input the totality of hits reconstructed in the detector in one event and return all the reconstructed particle tracks, each track with its own set of hits. This problem is too hard to solve due to the very large combinatorics.

The problem may be simplified by asking to identify all the particle tracks in a given group of hits, also each particle with its own hits. It turns out this is still too hard. So an even simpler problem is attempted.

In each group of reconstructed hits, there is one truth particle that has the largest number of hits in the group. This particle is denoted the *majority particle* or the *leading particle*. If a hit belongs to the majority particle, then it is assigned a label of 1 and is considered a positive hit (signal). If not, it is assigned a label of 0 or -1. In this study a label of -1 is used and it is considered a negative hit (background). The number of positive hits in the group represents physically the *majority particle size* and is denoted by `nbPositiveHit`. The question then becomes for each hit in the group if it belongs or not to the particle with the largest number of hits in the group (the majority particle). In other words, the question is given the  $x$ ,  $y$ ,  $z$  coordinates of all the hits in the group to predict the label of each hit in the group (1 or -1).

### 3.2 Approximate Nearest Neighbours

A preliminary step is therefore to select groups of hits from the event that are close to each other in such a way as to be likely that the group contains at least one real particle inside. We employ an algorithm denoted *approximate nearest neighbours*, a form of unsupervised learning, described in detail in Reference [16]. The collection of hits with  $x$ ,  $y$ ,  $z$  coordinates is represented mathematically by a 3D point cloud.

Usually the distance considered to measure the closeness is the euclidean distance in 3D ( $x$ ,  $y$ ,  $z$ ). However, in this case, it is best to select hits not closest in space, but closest to the same direction of travel from the centre of the detector onwards, as illustrated in Figure 2.4 for several truth particles. Nearest neighbours can be computed exactly via brute force techniques, but they consume too many resources in terms of memory, CPU and time. Given the large number of hits in one event, it is preferred to use a more efficient algorithm, even if it

may not return the exact result at every query. Such methods are called *approximate nearest neighbour* methods. The algorithm contains two main steps, illustrated in Figure 3.1.

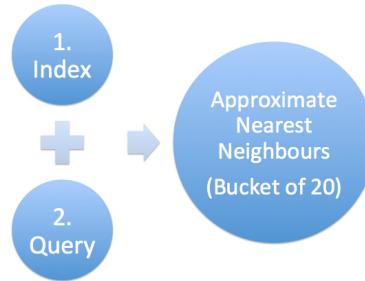


FIGURE 3.1: There are two steps in the ApproximateNearestNeighbours. First create an index from the positions of all hits in the event. Then query hits (even in parallel) to find the nearest neighbours to each hit.

The first step takes as input the  $x$ ,  $y$ ,  $z$  coordinates of all the reconstructed hits in an event and returns as output a tree that groups the hits along their direction relative to the centre of the detector. The tree is built using random projections. At every node, a random sample of two hits is chosen and the hyper-plane equidistant to both hits is chosen to divide into two further subspaces. This is done  $k$  times, to obtain an ensemble method of a forest of trees.  $k$  is a hyper-parameter to tune and is set to 10 in our study. The result is called an index.

In the second step, a query hit is given and the  $N$ -nearest-neighbour hits along the direction are returned (including the query hit), where  $N$  is a parameter chosen by the user. The resulting group of hits is called a *bucket* or a *hash* of hits. The operation is done for every hit in the event, resulting in as many buckets as hits in the event. The procedure is then repeated for each event in the sample.

There are several code implementations of the Approximate Nearest Neighbors algorithm. In this project the implementation in Annoy (Approximate Nearest Neighbors Oh Yeah) [17] is used. Annoy is fast, since it is coded in C++, but easy to use from Python as it provides a Python wrapper (or bindings). Annoy is an efficient library as a static read-only file is produced for the index, allowing it to be queried simultaneously (in parallel) by many threads, like when running on several CPU cores, or a GPU, or in a real production environment. The Annoy implementation supports the two step process described above: building one tree and querying in parallel for several points.

The number  $N$  of hits in a bucket is chosen to be 20, since in Figure 2.3 the count of truth particles by the number of hits tails off just before 20. The pseudo-code used in this project to produce a bucket for each hit in each of the 100 events described in Appendix 7.1.

### 3.3 Buckets

With the procedure above the distribution of the size of the majority particle, defined in Section 3.1 and denoted `nbPositiveHit`, is obtained and illustrated in Figure 3.2 for the Train and Test samples that are described later in Section 4.5. The mean and rms of the histograms are 8.5 and 2.8, respectively. The interpretation is that the average bucket contains fewer than 10 hits belonging to the same particle.

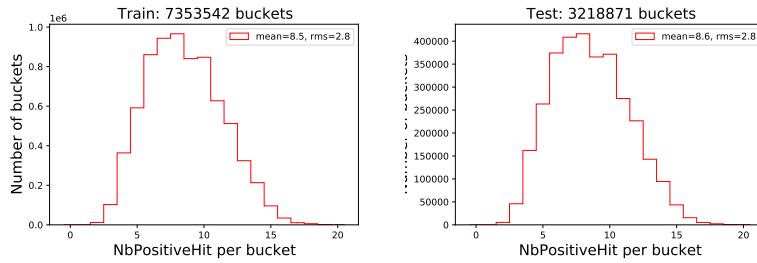


FIGURE 3.2: The distribution of the majority particle size, or nbPositiveHit.

Similarly as to the reconstructed hits above, another aspect realised from the Figure 2.3 is that only a fraction of truth particles have a number of hits smaller than 10. For both reasons, it is desired to reconstruct a particle with a significant number of hits in the bucket. A threshold of 10 is chosen. If a bucket has  $\text{nbPositiveHit} < 10$ , then all its hit labels are set artificially to -1, leading to its  $\text{nbPositiveHit} = 0$ . In Figure 3.3 the default threshold of 0 (Min00) and the chosen threshold of 10 (Min10) are overlaid for both the Train and Test samples. The Train and Test samples represent 70% and 30% of the events, respectively, as detailed in Section 4.5, and behave similarly. For each sample, the histograms for  $\text{nbPositiveHit} \geq 10$  are identical (seen as an overlapped colour). The values in red with  $\text{nbPositiveHit} < 10$  appear in blue at  $\text{nbPositiveHit} = 10$ . After this change the unbalance between the fraction of positive and negative hits is increased in the favor of the negative hits. A further balancing is needed, as described in Section 4.6.

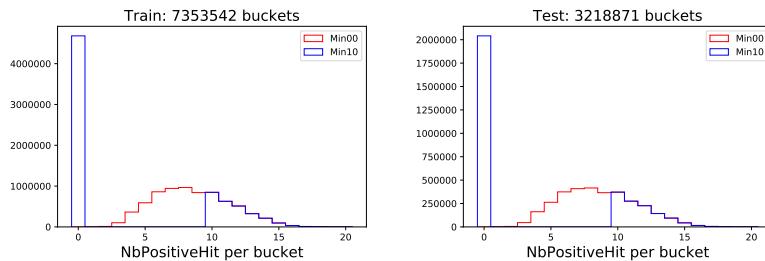


FIGURE 3.3: Overlay of the number of positive hits per bucket by default (Min00) and after moving all hits to negative for buckets with less than 10 number of positive hits (Min10). Train (left) vs Test (right).

## 3.4 Multi-Label Binary Classification

To formulate mathematically the track reconstruction, for each bucket one must ask as many questions as there are hits in the bucket. Does the hit belong to the majority particle, or does it have the label +1 (signal) or -1 (background)? For one hit, it is a classification problem. For the entire bucket, it is a multi-label classification problem. This can be answered via a supervised machine learning model. It is a supervised problem, as the labels are known. It is a classification problem, as the answers are categorical (yes or no, +1 or -1). There are only two possible answers, so it is a binary classification. It is a multi-label, as for each data sample (bucket), several questions are asked (one for each hit). The exact procedure is described in detail in Chapter 4.



## Chapter 4

# Machine Learning

### 4.1 Introduction

As described in Section 3.4, the reconstruct tracks from the reconstructed hits in an event, one must solve a multi-label binary classification problem. This chapter describes how Machine Learning answers this question.

Since the labels are known, it is a supervised problem. Supervised learning can be either classification or regression. The possible answers are finite, or categorical, so it is a classification problem. There are several types of classification problems.

Multi-class is a classification with more than two classes. In this category each sample is assigned to one and only one label. For example, a color can have only one label from several choices: red, green, blue. In other words, there is only one question, and each answer can be one of three or more labels.

Multi-label is a classification where each sample is mapped to a set of labels, each being binary. It is as if asking several questions, each with an answer of yes or no. In this case, the question asked for each hit is if the output is -1 or 1 for this hit. So this problem is a multi-label one.

The general case is a multi-class multi-label classification problem, when there are many questions, and for each the answer can be chosen from several labels.

### 4.2 Neural Network Architecture

Several machine learning algorithms can be used to perform a classification task. The most common are decision trees and neural networks [18]. Several decision trees are trained and grouped together into an ensemble method like random forests and boosted decision trees. In general, to find a multidimensional highly non-linear function representing a non-linear relation between input and output, it is efficient to train a neural network (NN). NNs are statistical models inspired by biological neural networks in the brain. The brain contains millions of neuron cells forming a network where electro-chemical impulses are being passed between them. An artificial neural network is formed by a number of interconnected artificial *neurons*, or *nodes*. In this project NNs are used.

One of the NN characteristics is that they contain weights along paths between neurons. The weights can be tuned with algorithms that learn from observed data to improve the model. The NN learns through optimisation techniques, like the gradient descent. A NN is represented by an architecture formed by layers of artificial neurons, which are able

to receive several inputs, which are processed by an activation function to determine the output. A simple model is formed by an input layer followed by a hidden layer and then an output layer. Each layer may contain one or more neurons. A NN with more than one hidden layer is called a deep neural network (DNN). For the best DNN performance, the model has to be designed according to the problem to solve, and then hyper-parameters are tuned. A general structure of a fully connected DNN is presented in Figure 4.1.

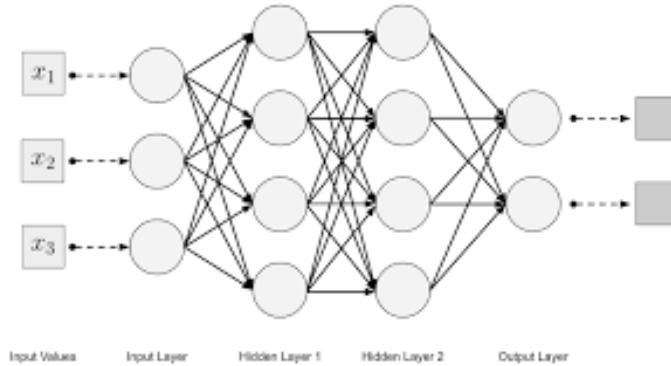


FIGURE 4.1: Diagram of a general architecture of a DNN. Credit image: O'Reilly. [19].

The *Universal Approximation Theorem* states that a neural network with one *hidden layer* can in principle approximate any N-dimensional function to an arbitrary degree of accuracy, given a sufficiently large (though finite) number of nodes. In practice however it is more suitable to use multiple hidden layers connected in series [18].

In a fully connected NN, each node takes a weighted linear combination of the outputs from nodes of the previous layer, adds its own bias value, and applies an *activation function*, then outputs the result to the neurons of the next layer, as illustrated in Figure 4.2. The activation function is chosen via optimisation for each neuron when the architecture of the NN is defined.

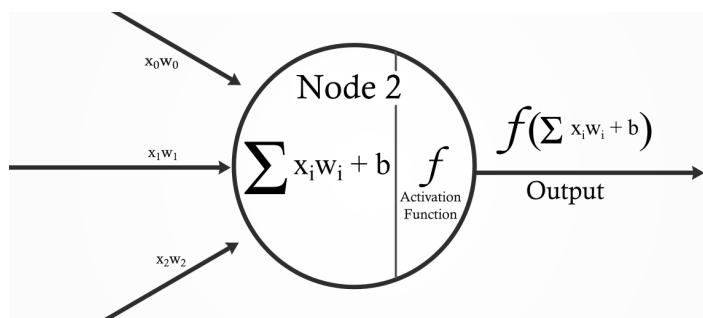


FIGURE 4.2: Diagram of a neuron or node in a NN, with its weight inputs, its bias and the output via the activation function. Credit image: The Fork [20].

Once the NN architecture is set, the layers, nodes and activation functions, the total function of the NN is parametrized by all the weights for the connections between nodes plus the biases of each node. Training the NN means learning these weights and biases so that the NN function can predict well the output when a new data not seen before is taken as input.

## 4.3 Hyper-Parameters

Several hyper-parameter choices can be made depending on the question being asked, regarding the number of hidden layers, number of nodes on a hidden layer, the activation function of nodes in the hidden layers, the activation function of nodes in the output layer, the learning optimizer, the loss function, the batch size. In the plots of this section the hyper-parameters that are retained for the best performing model are colored in red, to already illustrate their behaviour relative to the other possible hyper-parameters.

The problem is a multi-label binary classification. Given a collection (bucket) of 20 hits, for each hit there is a question of a binary classification (if yes or no) the hit belongs to the particle with the largest number of hits in the bucket.

The output labels of yes and no may be encoded as 1.0 and 0.0, or as 1.0 and -1.0, respectively. A preliminary study suggests that the latter option provides better results. With this choice, there remains only a limited choice of activation functions of the output layer and of loss functions.

Firstly, the output value of the NN prediction fixes the choice of the activation function on the last layer to the hyperbolic tangent (TANH), which has values between -1.0 and 1.0, and not the logistic regression, also called sigmoid, which has values between 0.0 and 1.0, illustrated in Equations

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1} \quad (4.1)$$

and

$$S(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}. \quad (4.2)$$

and on the left-hand side of Figure 4.3. Two more activation functions are possible for values between -1.0 and 1.0. The square non linear (SQNL) is described by Equation

$$\text{SQNL}(x) = \begin{cases} -1, & x < -2.0 \\ x + \frac{x^2}{4}, & -2.0 \leq x < 0 \\ x - \frac{x^2}{4}, & 0 \leq x < 2.0 \\ 1, & x > 2.0 \end{cases}. \quad (4.3)$$

and the soft sign (SOSI) function by Equation

$$\text{SOSI}(x) = \frac{x}{1 + |x|}. \quad (4.4)$$

All three functions reach the values of -1.0 and 1.0, but for different values of x. SQNL, TANH and SOSI reach the value of 1.0 (-1.0) for the x values of exactly 2.0 (-2.0), of around  $\pi \sim 3.14$  ( $-\pi \sim -3.14$ ) and for  $\infty$  ( $-\infty$ ), respectively, as illustrated in the right-hand side of Figure 4.3.

The appropriate loss functions for the target values of -1.0 and 1.0 are only the (regular)

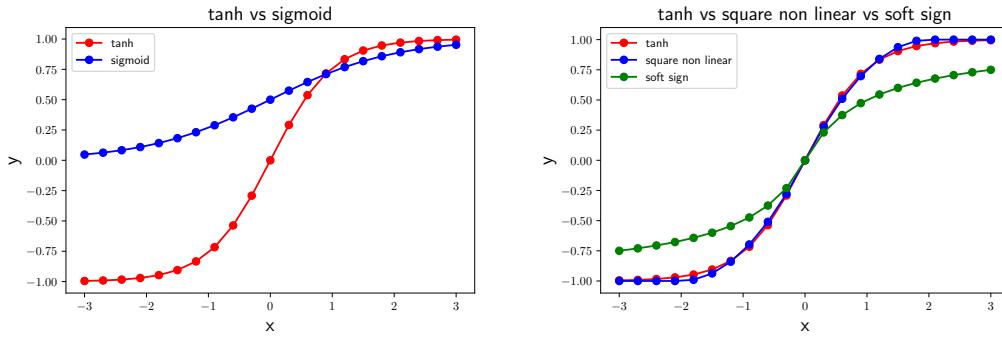


FIGURE 4.3: Overlaid potential activation functions for the last layer. Left: hyperbolic tangent (tanh) and logistic regression (sigmoid). Right: tanh, square non linear and soft sign. Tanh is chosen as our output labels are -1.0 and 1.0.

hinge function and the squared hinge function. Denoting  $y$  the predicted output and  $t$  the true target output (label), the hinge function is given by the Equation

$$\text{Loss function hinge : } l(y) = \max(0, 1 - t \cdot y), \quad (4.5)$$

and the squared hinge by the Equation

$$\text{Loss function squared hinge : } l(y) = [\max(0, 1 - t \cdot y)]^2. \quad (4.6)$$

Their relative behaviour is illustrated in Figure 4.4 for  $t=-1.0$  (left) and  $t=1.0$  (right). These loss functions never become negative. For  $t=1.0$ , for  $y \geq t$ , the loss function is exactly zero. For  $y < t$ , the loss function gradually increases. The squared hinge does not have a discontinuity at  $y = 1$  and at high values increases more than the regular hinge, penalizing large deviations more. The same is valid for  $y=-1.0$ , but in the opposite direction.

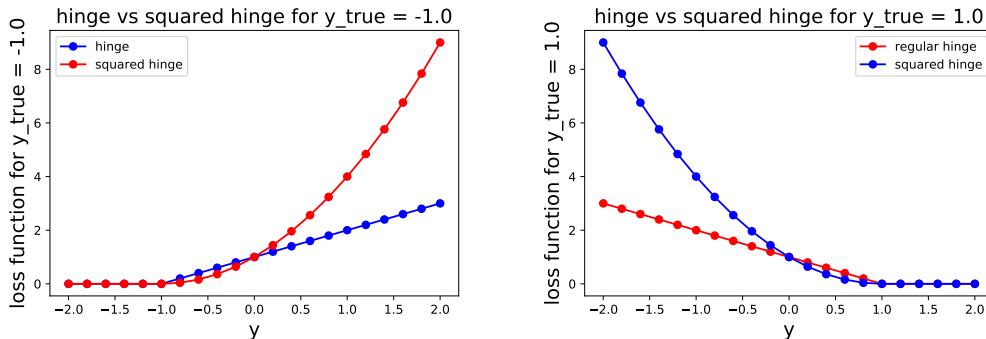


FIGURE 4.4: Overlaid loss functions of (regular) hinge and squared hinge for varying predicted  $y$ , for a fixed true  $y$  of -1.0 (left) and 1.0 (right).

The functions above apply to a pair of predicted  $y$  and true  $t$  values. In our problem it refers to each hit. The final loss function for the entire sample represents a sum over all the buckets in all events in the train or test sample, and for each bucket the sum over each of the 20 hits, as exemplified in Equation

$$\text{Loss function hinge : } l(y) = \sum_{\text{bucket}} \sum_{\text{hit}} \max(0, 1 - t_{\text{hit}} \cdot y_{\text{hit}}) \quad (4.7)$$

and

$$\text{Loss function squared hinge : } l(y) = \sum_{\text{bucket}} \sum_{\text{hit}} [\max(0, 1 - t_{\text{hit}} \cdot y_{\text{hit}})]^2. \quad (4.8)$$

Another choice to make is that of the activation functions for the nodes of the hidden layers. Besides the already-mentioned sigmoid and hyperbolic tangent functions, the Rectified Linear Unit (ReLU) is introduced. ReLU is the mostly common activation function for neural networks nowadays, including for more advanced neural networks such as convolutional neural networks (CNN) or deep neural networks (DNN). ReLU is *rectified* from the bottom, meaning its values are zero for negative inputs and return the same value as the input for positive inputs. ReLU can be summarized by Equation

$$\text{ReLU} : R(x) = \max(0, x). \quad (4.9)$$

While both the function  $R(x)$  and its derivative are monotonous, the function also has some drawbacks. It is not differentiable at zero. Since for all negative values the input is exactly zero, for methods learning with gradient descent, the ability to learn is reduced. To address this problem, a variation of ReLU is introduced, namely the Exponential Linear Function (ELU), described by Equation

$$\text{ELU} : E(x) = \begin{cases} \alpha(e^x - 1), & x < 0 \\ x, & x \geq 0 \end{cases}. \quad (4.10)$$

For positive values, the function remains the same. But for the negative values, an exponential curve appears, tending smoothly to a constant value  $\alpha$ . ELU has the advantage over ReLU to be fully continuous and differentiable, and not having a vanishing gradient problem for gradient descent learning. Its main disadvantage is that it is slower to compute for negative values. But it may be worth it for a more precise result. The comparison of the ReLU and ELU functions is illustrated in Figure 4.5.

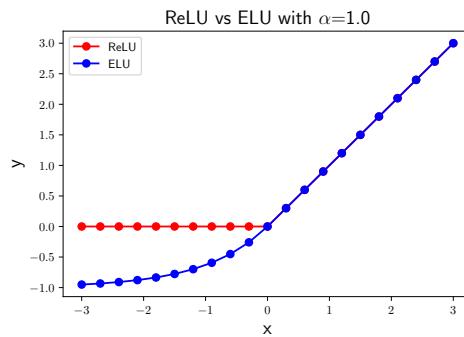


FIGURE 4.5: Overlaid activation functions of ReLU and ELU, with  $\alpha = 1.0$ .

Another hyper-parameter to tune is the number of hidden layers and the number of nodes on each hidden layer. The *Universal Approximation Theorem* suggests that one layer with a very large number of nodes is enough to learn any arbitrary function. But in practice it is worth having several consecutive layers of fewer nodes per layer. That forms a deep neural network.

Another option in the architecture of the NN is whether to use or not to add a regularisation layer, in particular a dropout layer [21]. Sometimes the resulting model is too complex relative to the quantity of input data, leading to overfitting during training. Overfitting is similar to memorisation of the input data, leading to not be able to predict correctly on new data. To avoid overfitting, regularisation techniques are used. Typical methods add a new term to the loss functions. Other techniques add a dropout layer. The dropout method sets randomly some of the inputs to 0 with a frequency  $f$ , and reweights the other inputs by  $1/(1 - f)$ , so that the total sum of inputs remains constant. The value of  $f$  is a hyper-parameter to be optimised. The location of the dropout layer is usually between the hidden layers and the last layer. Dropout is applied only during training, and not during testing or inferring on new data.

Another choice related to the learning method is the learning optimiser. Two algorithms based on stochastic gradient descent are compared, namely Adam [22] and AdaDelta [23]. For both their default parameters are used, the most important being the learning rate of 0.001. Adam is very computationally efficient, while AdaDelta uses adaptive learning rate.

## 4.4 Learning Methods

NN training is done learning on the training dataset and testing on the testing dataset. Running once over all the data from training and testing represents an *epoch*. During an epoch, events are analysed in groups called *batches*. The number of epochs to run on and the number of events in each batch can be optimised.

Let's take a look at how the NN training happens. At first the NN has random values for the weights and biases. For each of the events in the first batch, data comes in, and the NN predicts an output values, that are at first very different from the true desired output value. To evaluate how far away are the predicted output from the desired output, a *loss* function is defined that can be chosen from several formulas, but have the generic form of a sum over the absolute values of these differences. The goal of the NN training is to update the values of the weights and biases so that the loss function is minimized. After the first batch, the NN changes the weights via a back-propagation algorithm using the optimiser algorithm. For each new batch, the weights and biases change, and become continuously closer to the correct values, as the loss function becomes gradually smaller. When all the training events are used, the first epoch is finished. The NN function at this point is then applied to the new dataset from the testing dataset, which is not split in batches, and a loss function is also calculated. The entire procedure repeats for the number of epochs chosen. At the end, the final weights and biases define the final NN model that has been learned.

## 4.5 Train and Test Split and k-fold

The k-fold validation is a procedure used to test the effectiveness of a machine learning model. It is used especially if the data are limited. Normally, the data are split in two equal parts (train and test), corresponding to  $k=2$ . For a general  $k$ , the data are split randomly into  $k$  groups.  $k-1$  groups are used in training and the last one in testing. The operation is repeated by permuting the groups, so that each group is used only once in testing. The final result is obtained by averaging out the permutations.

It is also possible to split into two unequal parts. In this project the split is done with  $k=2$ ,

70% in train and 30% in test. Although each data sample is a bucket, it is convenient to use that events have about the same number of buckets. The split of 100 events is done with 70% of events in train and 30% of events in test. One can split the events randomly, as by the laws of particle physics events simulate independent particle collisions. NN training takes a significant amount of time. It is useful to train on an increasing data sample size and check if the performance reaches a plateau. For convenience, to study easily with a step of 10 events, events are grouped in consecutive order by 10, with the first 7 events being used for training (Train sample) and the following 3 events are used for testing (Test sample). The pseudo-code is described in Appendix 7.1.

## 4.6 Balancing Datasets

It is common practice in ML classification problems that the signal target is much rarer than the backgrounds. It is said the dataset is unbalanced. The solution is to balance the dataset by increasing the weights of signal events such that the total sum of weights of signal equals the total sum of weights for background. The NN learning uses for each data samples its own specific weight. A balanced dataset is used for training. It is not used in testing, in order to represent a real-world situation when one tries to predict the proportion of signal to background, and does not know the balancing ratio.

This project involves a multi-label classification problem, where there are 20 hits in a bucket, and for each hit a classification question is asked, if it is a signal (1) or background (0) relative to being part of the majority particle in the bucket. From physics studies in Chapter 2, it is decided that all buckets with less than 10 hits with output = 1, have all their hits set to output = -1. The dataset becomes even more unbalanced. Balancing is done in two steps.

First, buckets are removed so that there are equal number of buckets for each of nbPositiveHit between 10 and 17. Values 18-20 are left unchanged, as they are too small. The goal is to learn uniformly and not be biased towards a particular category. nbPositiveHit between 1 and 9 are zero by construction, as all these values are moved to nbPositiveHit of zero, which originally did not have any entries. Then the number of buckets of nbPositiveHit=0 is reduced so that overall, there are exactly equal numbers of positive and negative hits in the sample. That ensures the most general way for learning, without a bias towards one category or the other. The training set is balanced, keeping about 130k buckets. The testing set remains unbalanced, with roughly 3.2M buckets.

## 4.7 Hyper-Parameter Tuning

The hyper-parameters are tuned by choosing the models that perform best in the test sample over 300 epochs, looking at the accuracy and loss that result directly from Keras/TensorFlow after the training. In the plots of this section, the best model summarised in Section 4.7.3 is compared with alternative models where all hyper-parameters are kept constant, except one that is changed. Since training on 1200 epochs and an unbalanced test dataset takes too long, 300 epochs and the balanced test dataset are used. The balanced train dataset are used both in this study and for the final result with 1200 epochs, described in Chapter 5. From this study the best performing hyper-parameters are chosen. When the performance of several hyper-parameters is similar, the simplest or most commonly used hyper-parameter is chosen. The resulting final model is presented in Section 4.7.3.

### 4.7.1 DNN Architecture

A comparison of the number of hidden layers is studied. The performance is similar for different values, and 3 hidden layers is retained for the final model, as it provides a slightly better performance, as illustrated in Figure 4.6.

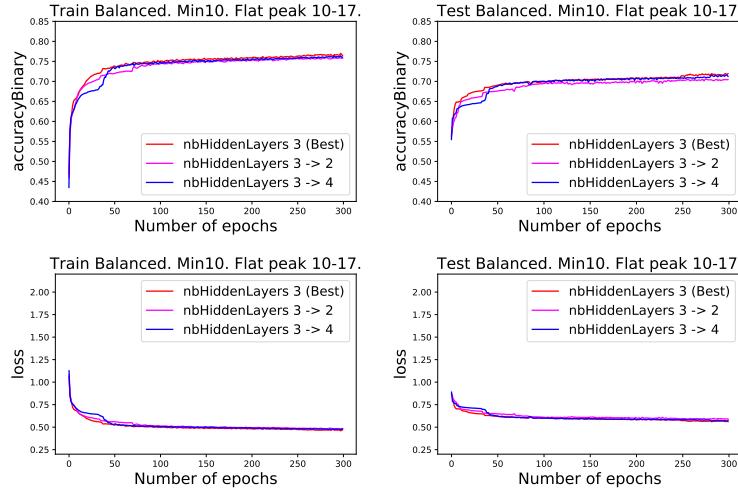


FIGURE 4.6: Comparison of different numbers of hidden layers. 3 hidden layer is best. Binary accuracy and loss in Train and Test balanced samples.

A comparison of the number of nodes on the hidden layers is studied. For simplicity, in this study it is considered the same number of nodes on each hidden layer. The performance is similar for different values, and 200 nodes on the hidden layers is retained for the final model, as illustrated in Figure 4.7. This represents 10 times more than the nodes on the output layer.

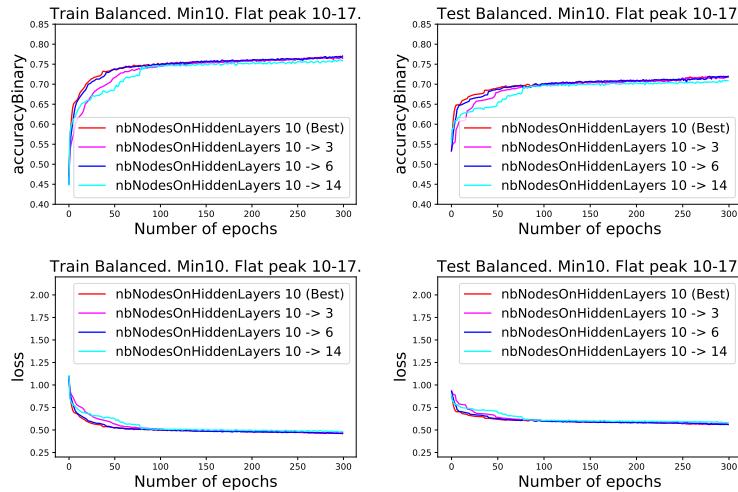


FIGURE 4.7: Comparison of the ratio of the different number of nodes on the hidden layers divided by the number of nodes on the last layer. k=10, or 200 nodes on each hidden layer, is best. Binary accuracy and loss in Train and Test balanced samples.

A comparison of the activation functions for the hidden layers, namely ReLU and ELU, is

performed. For simplicity, in this study it is considered that all nodes of all hidden layers have the same activation function. The performance is similar between the two options, so the standard and mostly used ReLU is retained for the final model, as illustrated in Figure 4.8.

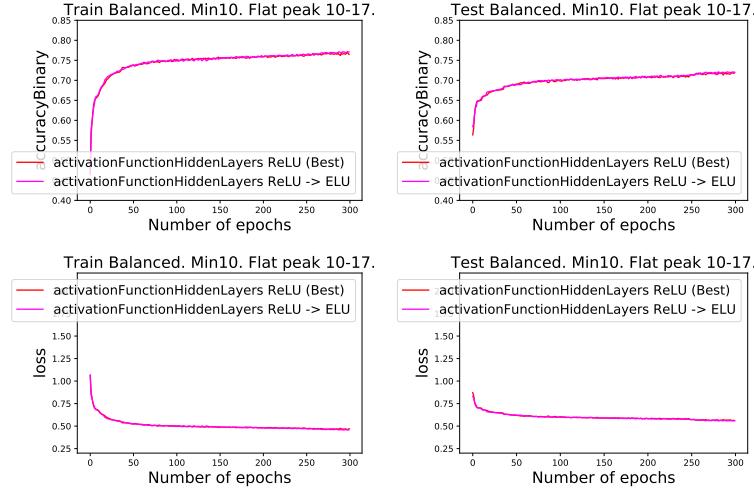


FIGURE 4.8: Comparison of ReLU and ELU activation functions on the hidden layers. ReLU is chosen for the final model. Binary accuracy and loss in Train and Test balanced samples.

A comparison of adding or not adding a regularisation function via the dropout layer at the end of the hidden layers is studied. The performance is better by having a dropout layer, as illustrated in Figure 4.9.

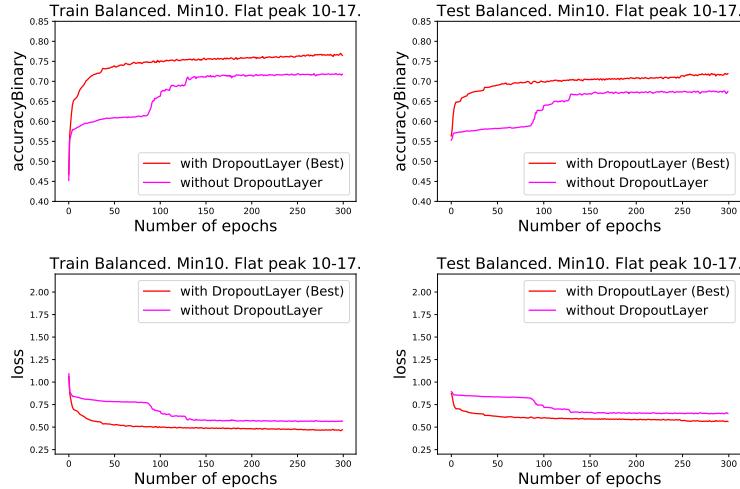


FIGURE 4.9: Comparison without and with a dropout layer for regularisation. A dropout layer is used in the final model. Binary accuracy and loss in Train and Test balanced samples.

A comparison of the activation functions for the last layer, namely TANH, SQNL and SOSI, is studied. The performance is similar for the three options, so the standard and mostly used TANH is retained for the final model, as illustrated in Figure 4.10.

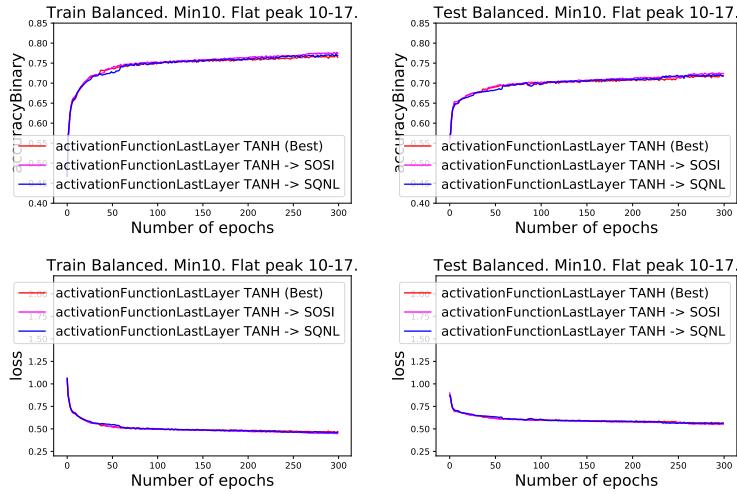


FIGURE 4.10: Comparison of TANH, SQNL and SOSI as activation functions on the last layer. TANH is used in the final model. Binary accuracy and loss in Train and Test balanced samples.

#### 4.7.2 DNN Learning

Moving on from the hyper-parameters that define the geometry of the deep neural network to those defining its learning method, a comparison of the optimizers, Adam and AdaDelta, each with its default parameters, is studied. The performance of Adam is significantly better than that of AdaDelta, so Adam is retained for the final model, as illustrated in Figure 4.11.

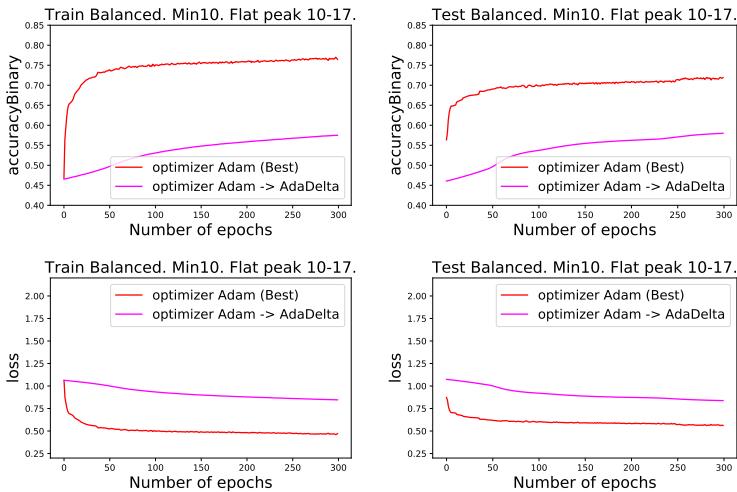


FIGURE 4.11: Comparison of Adam and AdaDelta optimizers for the learning method. Adam is used in the final model. Binary accuracy and loss in Train and Test balanced samples.

A comparison of the loss functions used to learn the weights and biases of the DNN via gradient descent, regular hinge and squared hinge, is studied. Their performance is similar, so the standard and mostly used regular hinge is retained for the final model, as illustrated in Figure 4.12.

The conclusion is that the learning part of the hyper-parameters tuning by comparing various batch sizes. The best performance is obtained for 50000, which is retained for the final

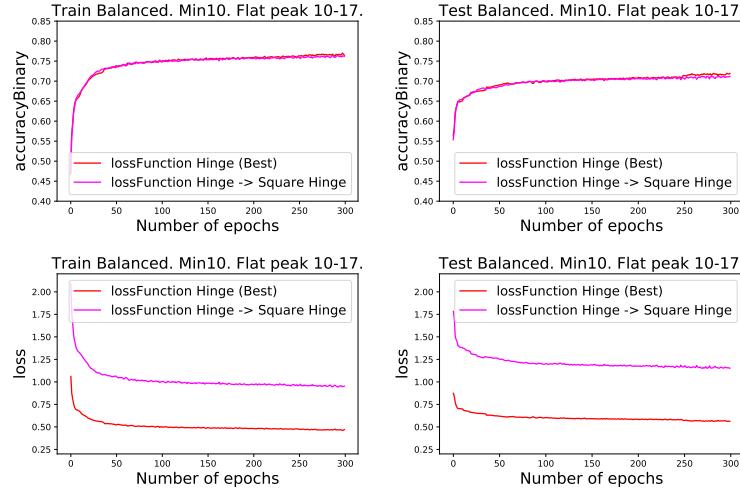


FIGURE 4.12: Comparison of regular and square hinge loss functions for DNN learning. Regular hinge is used in the final model. Binary accuracy and loss in Train and Test balanced samples.

model, as illustrated in Figure 4.13.

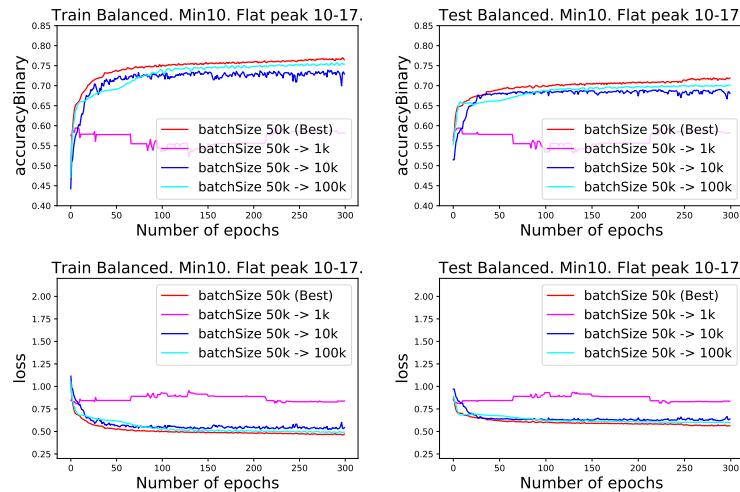


FIGURE 4.13: Comparison of various batch sizes for DNN learning. A batch size of 50000 is used in the final model. Binary accuracy and loss in Train and Test balanced samples.

### 4.7.3 Best Model

The problem structure fixes the number of nodes on the input layer to 60 (3 coordinates for 20 hits), and of the output layer to 20 (1 boolean for 20 hits). The DNN architecture and learning methods are optimised as hyper-parameters, whose options are limited by the choice of representing the answers yes and no by 1.0 and -1.0, respectively. The hyper-parameters that describe the best model are summarised as follows.

There are 3 hidden layers, each with 200 nodes, or 10 times more than the number of nodes in the output layer. A reminder is that the input layer has 60 nodes (3 coordinates x, y, z for each of the 20 hits in the bucket) and the output layer has 20 nodes (an output of -1.0 or 1.0

for each of the 20 hits in the bucket). A dropout layer (0.2) is added at the end of the hidden layers. The activation function for the hidden layers is the rectified linear unit (ReLU). The activation function for the last layer is hyperbolic tangent (tanh). The loss function is the (regular) hinge function. The batch size is 50000.

While the choice of hyper-parameters is done using 300 epochs and the balanced test dataset, the final result uses 1200 epochs and the unbalanced test dataset.

## 4.8 Predicting or Inference and Figures of Merit

Once the model is trained, it can be applied to a new dataset to infer or make a prediction.

Besides the value of the loss function across the entire dataset, there is also another figure of merit of how well does a NN perform in training and testing. It is called *accuracy* and is related to the number of true positives or false negatives. The larger the accuracy, the better.

For the training dataset, the loss and accuracy values always improve. But in the testing dataset they can start to degrade if we train for too many epochs. By degrading it means that the loss value starts to grow, and the accuracy value starts to decrease. That is called over-training and consists of memorizing the inputs, and thus not being able to predict correctly any more for new inputs.

The confusion matrix is a table that summarises the performance of a binary classification model, as illustrated in Table 4.1.

TP	FP
FN	TN

TABLE 4.1: Confusion Matrix.

The four metrics presented in the table are true positive (TP), false positive (FP), false negative (FN), and true negative (TN). Based on these four numbers further figures of merit are derived.

The accuracy defines what percentage of the total predictions are classified correctly, either as TP or TN, as defined by the Equation

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}. \quad (4.11)$$

The precision defines the percentage of the predicted positive that are actually positive, as defined by Equation

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \quad (4.12)$$

The recall defines the percentage of actual positive that are correctly predicted, as defined by the Equation

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \quad (4.13)$$

The equivalents of the precision and recall can be constructed also for the negative values, as if the negative values are the sought target. The negative predicted value is the negative equivalent of precision and it represents the percentage of the predicted negative that in reality are also negative, as defined by the Equation

$$\text{Negative Predicted Value} = \frac{\text{TN}}{\text{TN} + \text{FN}}. \quad (4.14)$$

The true negative rate is the negative equivalent of the recall. It defines the percentage of the real negative that are also predicted as negative, as defined by the Equation

$$\text{True Negative Rate} = \frac{\text{TN}}{\text{TN} + \text{FP}}. \quad (4.15)$$

The pseudo-code to calculate the metrics is described in Appendix 7.2.

## 4.9 Software Used

Several software Python libraries are used to perform this study, including data manipulation, the NN training, the figure of merit evaluation and plotting.

Numpy is a Python programming library, which is a coding support for large multi-dimensional arrays and matrices [24].

PANDAS is a data manipulation and analysis library [25]. It is an open source library that is made mainly for working with relational or labeled data. It provides various data structures and operations for manipulating numerical data and time series. It is fast, has high performance and productivity. Data frames is a two-dimensional data structure with labeled data (rows and columns). Pandas uses numpy behind the hood.

Jupyter Notebook is an important tool in the data science field. It is a web-based interactive computing platform [26]. The Notebook combines code, equations, text, visualisation, interactive dashboard and other media. It works in line code, using blocks.

Matplotlib is used for producing the plots in this thesis, both 1D and 2D [27].



## Chapter 5

# Model Performance

In this chapter the performance metrics of the best deep neural network model are described. The hyper-parameters values for the best model are enumerated in Section 4.7.3.

## 5.1 Training Metrics

In Figure 5.1 the binary accuracy and loss function are presented as resulted directly from training in Keras and Tensorflow. Note that while Train is balanced and in general Test is unbalanced, for these plots Test is also balanced. The reason is that it would be too slow to train the NN with 1200 epochs with Test unbalanced. Training is done on the balanced Train dataset. Prediction is done on the unbalanced Test dataset.

## 5.2 Hit-Level Metrics

In this section the metrics at the hit-level are studied. A bucket is made of 20 hits. Figure 5.2 presents at the top the distribution of the number of positive hits (`nbPositiveHit`), at the bottom the distribution of the number of hits predicted to be positive, with Train on the left and Test on the right. The distributions are normalised, so their shapes can be compared. The number of events used in Train relative to Test is 7 to 3. Furthermore, the Train dataset is balanced, keeping only a tiny subset of the buckets, while the Test dataset is unbalanced, keeping all of its buckets. In both datasets all values of  $\text{nbPositiveHit} < 10$  are set artificially to 0, considering there was no hit belonging to an interesting particle. Furthermore, the Train dataset is balanced, so that buckets with  $\text{nbPositiveHit}$  between 10 and 17 are eliminated, so that equal numbers of buckets for each  $\text{nbPositiveHit}$  remain. Since the distribution is falling (as seen in the Test plot), all values take the lower value of the bin 17. Values 18-20 are left unchanged, as they are too small. Values 18-20 are left unchanged, as they are too small.

Tests were done with values varying from 14 to 20, but 17 had by far the best outcome. Lower than 17 would not have a training distribution flat enough. Higher than 17 would leave very few buckets to train on.

At the bottom of Figure 5.2 there are the equivalent plots after the prediction using our model. The training is quite solid. There is a tall bin at  $\text{nbPositiveHit} = 0$ , followed by virtually empty bins, and the distribution starts to grow at  $\text{nbPositiveHit} = 10$ , just as in the desired output distribution. For the higher bins, the relative shape does not agree fully to the desired output. For Train, instead of being flat, the distribution increases with larger  $\text{nbPositiveHit}$ . For Test it decreases as expected, but not nearly as fast. Overall, however, the model behaves quite well, relative to many other hyper-parameter settings tried.

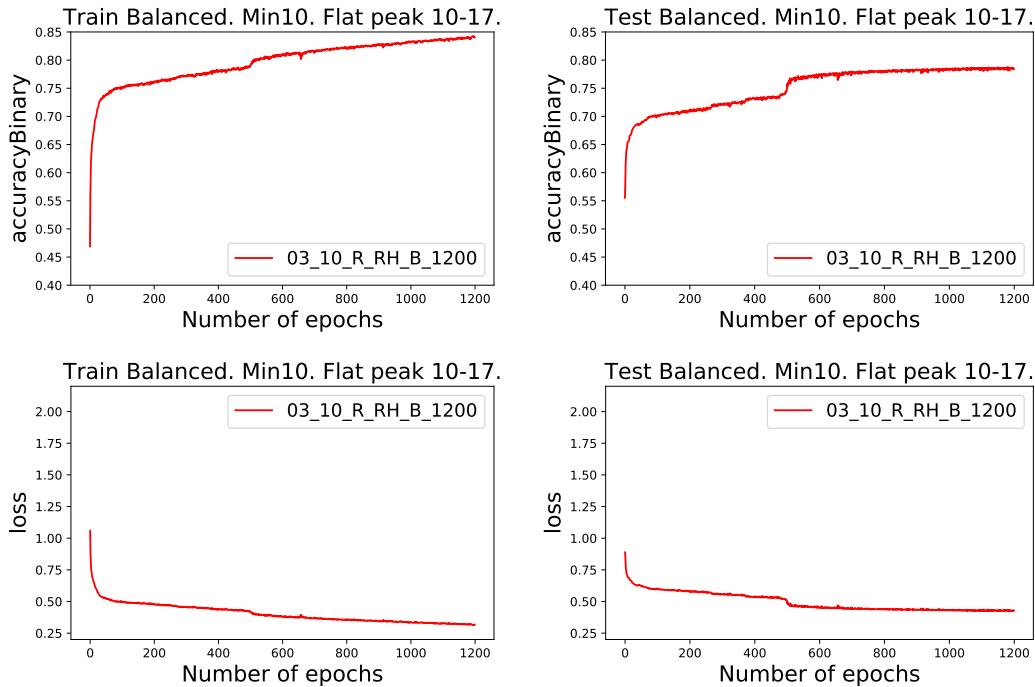


FIGURE 5.1: Binary accuracy and loss, as resulted directly from training in Keras and Tensorflow. Note that while Train is balanced, Test is also balanced, as it would be too slow to have it unbalanced.

Figure 5.3 presents the 2D distributions of the output predicted vs output in terms of number of positive hits. The color coding is proportional to the number of entries in the 2D histogram. As expected, there is a peak at (0, 0), meaning events with exactly zero number of positive hits (after the move of all values smaller than 10 to 0), and that are also predicted to have exactly zero. There is nothing between 1 and 9. And for values  $\geq 10$ , there is a diagonal, as expected for the Train balanced. For the Test unbalanced, it is harder to see. But indeed most entries are in the top left corner of this region, as expected.

From these plots it can be concluded that in the balanced dataset (Train and Test) a diagonal can be nicely seen. In the Test unbalanced it is harder to see, but still values look relatively flat in 1D.

Figure 5.4 presents figures of merit at hit level for each volume\_id in the detector. By looping over all buckets and all hits, both the true and predicted output labels for each hit are known. It results if the hit is TP (True Positive), FP (False Positive), FN (False Negative), or TN (True Negative). The number of hits in these four categories in each volume\_id is counted. From these for each volume\_id there are estimated the figures of merit of accuracy, precision, recall, predicted output negative and true negative rate. They all need to be as high as possible, ideally 1. As it is impossible for one model to satisfy all criteria, as known from the bias-variance trade-off, a model with overall good performance across all categories is chosen.

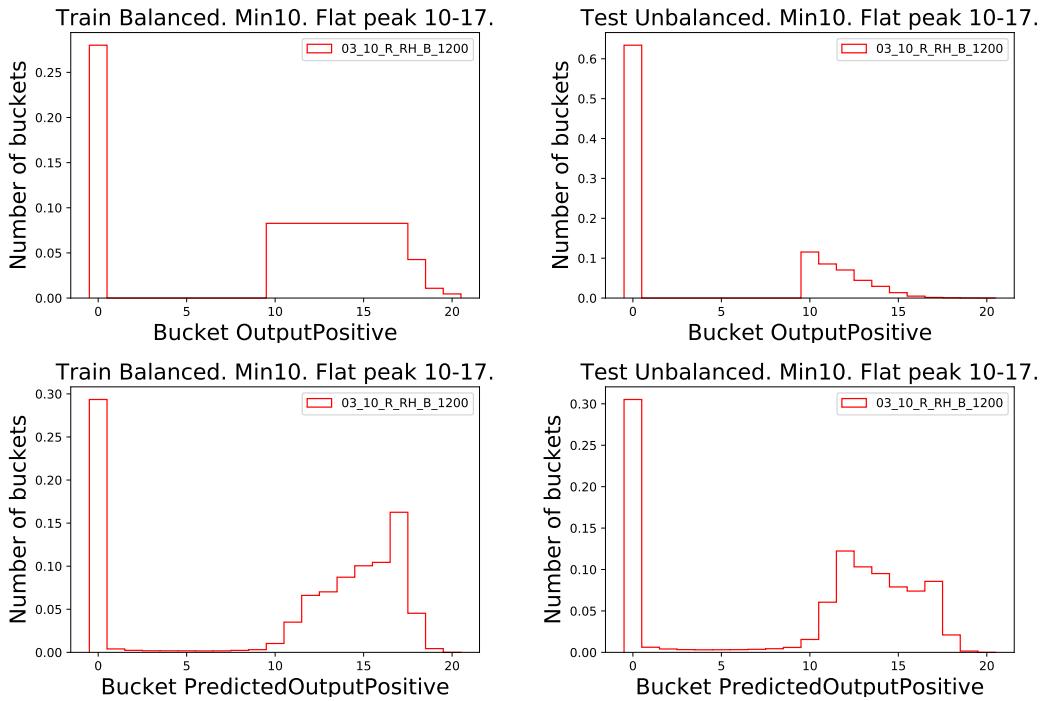


FIGURE 5.2: Output and output predicted 1D. Older ones have a sharp peak, the new ones are more flat.

### 5.3 Particle-Level Metrics

Two numpy arrays of dimension two are used: the true output and the output predicted by the model. Each row represents a bucket. There are 20 columns, representing the 20 hits. A for loop over buckets is made. For each bucket a loop over hits is made. Each hit has a value of -1 or +1 for the output and for the predicted output. For the current bucket, the number of hits that are positive (`nbPositiveHit`) is counted. A count is also made for those that are both positive and in addition predicted to also be positive (`nbTruePositiveHit`). A bucket with  $\text{nbPositiveHit} \geq 10$  is considered to contain a truth particle. If in addition the bucket has also  $\text{nbTruePositiveHit}/\text{nbPositiveHit} > 80\%$ , it is considered to also have reconstructed correctly that particle. A count is made of the buckets that have a truth particle, and then those that both have a truth particle and have reconstructed it. The efficiency to reconstruct a particle ( $\text{eff} = \text{nbParticleReco}/\text{nbParticleTruth}$ ) is 84.2% for Train and 71.3% for Test. Detailed results are summarized in Table 5.1.

Sample	eff	nbBucket	nbParticleTruth	nbParticleReco
Train Balanced	84.2%	130k	94k	79k
Test Balanced	74.9%	62k	45k	34k
Test Unbalanced	71.3%	3219k	1178k	840k

TABLE 5.1: Particle reconstruction efficiency results.

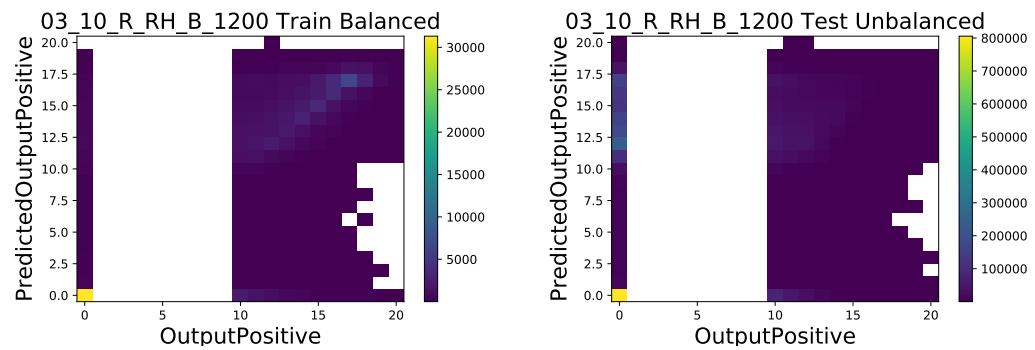


FIGURE 5.3: Output and output predicted 2D. In Train balanced, a diagonal is seen. In Test unbalanced it's harder to see.

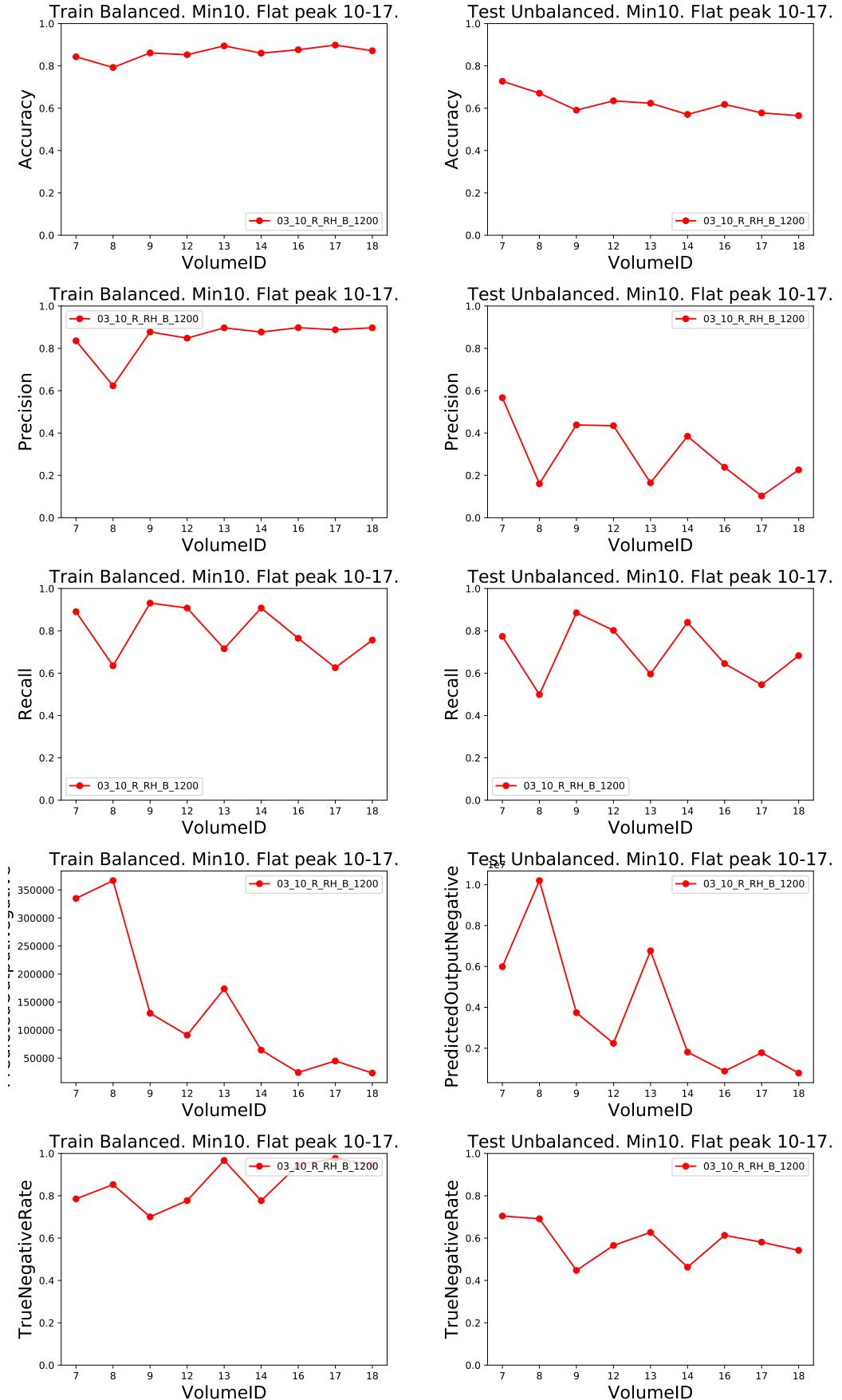


FIGURE 5.4: Accuracy, precision, recall, predicted output negative, true negative rate. Train (left) and Test (right).



## Chapter 6

# Conclusions

### 6.1 Conclusions

CERN experiments at the LHC plan to continue their studies of the building blocks of the Universe by studying proton-proton collisions with ever instantaneous luminosity. This improves the probability that interesting particles are produced, such as the Higgs boson, top quarks, W and Z bosons, and hopefully also particles predicted by models Beyond the Standard Model. It translates to increasing number of collisions per bunch crossing (pile-up  $\mu$ ). For Run-4,  $\mu$  is expected to reach a value of 200, from a maximum around 60 today. The drawback is that the particle events become much busier, with many more particles overlapping. Reconstructiong these particles becomes harder. So much so that computing resources are predicted to not be enough to reconstruct Run-4 detector data and simulation data.

The solution is to improve dramatically particle reconstruction from the algorithm and software side. In this regard, machine learning is a promising avenue studied thoroughly at CERN. In this thesis, a deep neural network (DNN) is studied to reconstruct particles from hit data in the inner detector, using simulations of a general particle physics detector at CERN, provided by the TrackML data challenge. The inner detector is split into subdetectors and regions defined by the `volume_id`, `layer_id` and `module_id`. For each simulated hit, both the reconstructed position and the original (true) position is known. Also the `particle_id` of the particle producing the hit is known. This allows to build a supervised machine learning algorithm, to study groups of hits at a time and try to identify the particle with the largest number of hits in the group. First all hits in an event are organised in an approximate nearest neighbour tree based on their spatial position. For each hit in the detector a group of hits is created consisting of hits that are closest to the direction of the line that connects this hit with the original collision in the center of the detector. Such group is denoted a bucket. Since a particle leaves typically around 10 hits in the detector, the bucket size is chosen to be 20. However, the number of hits belonging to the particle with the largest number of hits is counted. If this number is smaller than 10, then it is considered that none of the hits belongs to this particle, and the labels of all hits are reset to -1. The goal is therefore that in each group of 20 hits, to identify those that belong to a particle that has at least 10 hits. It is a multi-class binary classification problem, as for each hit in the bucket there is a question that can be answered by yes or no, namely if the hit belongs to the particle with the largest number of hits in the bucket.

A deep neural network algorithm is trained, using Keras and TensorFlow in Python. 100 events are used, split 70% in training and 30% in testing. The training data set is balanced. The test sample is studied both in a balanced and unbalanced format. The balanced training set has about 130k buckets, the balanced testing set about 62k buckets, and the unbalanced

testing set about 3.2M buckets. Hyper-parameters are tuned and the resulting structure of the best model is summarised in Section 4.7.3.

The final performance metrics are shown for the unbalanced dataset, ensuring a particle reconstruction efficiency of 71.3%.

## 6.2 Future Plans

Given more time, several improvements or new studies may enhance the current results.

The output labels of the hits of no and yes are represented by -1.0 and 1.0. One can represent them also as 0.0 and 1.0. This leads to the use of other activation functions on the last layer and to other loss functions. A preliminary study suggests that -1.0/1.0 behave better than 0.0/1.0, but a more thorough study may be performed.

Only 100 events have been studied in this project. The entire TrackML dataset is 100 times larger, consisting of 10000 events. Deep learning methods benefit from using large quantities of data. Using the entire dataset should provide a better performing model. The technical challenges will remain the computing power needed for training. Dedicated resources at CERN and member institutes, such as the University of Geneva, using CPU and ideally also GPU, may then be used to improve the training and inference times.

Once the current question is addressed, namely of identifying hits belonging to the particle with the largest number of hits, more complex questions may be tackled, such as identifying several particles at once from a given bucket, probably with using a larger bucket size.

To conclude, this study is a stepping stone towards improving tracking reconstruction for Run-4 at the LHC at pile-up  $\mu = 200$ , for fast real-time inference for trigger and particle offline reconstruction, using advanced machine learning techniques.

## Chapter 7

# Pseudo-Code

In this appendix chapter several pseudo-code algorithms are presented.

### 7.1 Input and Output Preparation

This section presents the pseudo-code for three algorithms for preparing the input and output for the NN training.

#### 7.1.1 Algorithm 1

The Algorithm 1 is the following: for all events in a folder, store in .npy files numpy arrays with each row representing a bucket of 20 hits.

The input variables are the following:

- inputFolderName: contains any number of files for each event in format "-hits.csv", "-truth.csv"
- outputFolderName:
- fileNameNNInputTrainAll: ("input\_train.npy")
- fileNameNNOutputTrainAll: ("input\_test.npy")
- fileNameNNInputTestAll: ("output\_train.npy")
- fileNameNNOutputTestAll: ("output\_test.npy")
- metric: ("angular")
- nrtrees: (10)

There is no output variables. There are only side effects.

There are the following intermediate variables:

- list\_eventNumber: used in the for loop, using the eventNumber and index i
- inputFileName\_hits\_recon
- inputFileName\_hits\_truth
- df\_hits\_recon
- df\_hits\_truth

- df\_hits
- nparray\_position
- numberDimension
- index
- nparray\_input\_all
- nparray\_output\_all
- nrBuckets
- list\_index\_Train
- list\_index\_Test
- nparray\_Input\_Train
- nparray\_Input\_Test
- nparray\_Output\_Train
- nparray\_Output\_Test
- nparray\_Input\_Train\_all\_Events
- nparray\_Input\_Test\_all\_Events
- nparray\_Output\_Train\_all\_Events
- nparray\_Output\_Test\_all\_Events

The method of the algorithm is the following.

```

create empty list: list_eventNumber=[]
for fileName in alphabetical list (sorted) of files in inputFolder:
    if fileName ends in "-hits.csv":
        eventNumber <-- from fileName remove "event" and "-hits.csv"
        append eventNumber to the list_eventNumber

for eventNumber in list_eventNumber also remember index i:
    inputFileName_hits_recon <-- inputFolderName+"/event"+eventNumber+-hits.csv"
    inputFileName_hits_truth <-- inputFolderName+"/event"+eventNumber+-truth.csv"
    df_hits_recon <-- read pandas df from csv file inputFileName_hits_recon
    df_hits_truth <-- read pandas df from csv file inputFileName_hits_truth
    df_hits <-- concatenate df_hits_recon & df_hits_truth on axis 1
    nparray_position <-- df_hits take column "x", "y", "z" and convert to nparray
    numberDimension <-- number of columns in nparray_position (3)
    metric <-- "angular"
    index <-- AnnoyIndex constructor(numberDimension, metric)
    for position in nparray_position with index j:
        add to index the position at index j
    build index with 10 trees

```

```

nparray_input_all, nparray_output_all <-- function with arguments df_hits, index
(see this function implemented in Algorithm 2)

nrBucket <-- number of elements in nparray_input_all
if nrBucket is odd remove last element from nparray_input_all and nparray_output_all
reshape nparray_input_all to have three dim as needed by TensorFlow
nrBucket <-- recount number of elements in nparray_input_all
list_index_Train <-- build list of even bucket indices
list_index_Test <-- build list of odd bucket indices

nparray_Input_Train <-- takes subset of nparray_input_all with list_index_Train
nparray_Input_Test <-- takes subset of nparray_input_all with list_index_Test
nparray_Output_Train <-- takes subset of nparray_output_all with list_index_Train
nparray_Output_Test <-- takes subset of nparray_output_all with list_index_Test

if i is 0:
    nparray_Input_Train_all_Events <-- nparray_Input_Train
    nparray_Input_Test_all_Events <-- nparray_Input_Test
    nparray_Output_Train_all_Events <-- nparray_Output_Train
    nparray_Output_Test_all_Events <-- nparray_Output_Test
else:
    nparray_Input_Train_all_Events <-- concatenate over axis 0
    nparray_Input_Train_all_Events & nparray_Input_Train
    nparray_Input_Test_all_Events <-- concatenate over axis 0
    nparray_Input_Test_all_Events & nparray_Input_Test
    nparray_Output_Train_all_Events <-- concatenate over axis 0
    nparray_Output_Train_all_Events & nparray_Output_Train
    nparray_Output_Test_all_Events <-- concatenate over axis 0
    nparray_Output_Test_all_Events & nparray_Output_Test
# done for loop over all events

write nparray_Input_Train_all_Events to binary .npy file fileNameNNInputTrainAll
write nparray_Input_Test_all_Events to binary .npy file fileNameNNInputTestAll
write nparray_Output_Train_all_Events to binary .npy file fileNameNNOuputTrainAll
write nparray_Output_Test_all_Events to binary .npy file fileNameNNOuputTestAll

```

### 7.1.2 Algorithm 2

The Algorithm 2 is the following. For one event, from df\_hits and Annoy index compute numpy arrays for NN input and output, where each row represents a bucket of 20 hits. For one bucket, the input has 60 elements (20 hits times 3 coordinates (x,y,z)), and the output has 20 elements (-1 or 1 depending if the hit belongs to the particle with largest number of hits in that bucket).

The input variables are the following:

- df\_hits
- index
- minValueOfNrHitsForParticleWithMostHits (0 or 10)
- bucketSize (20)

The output variables are the following:

- nparray\_input\_all
- nparray\_output\_all

The intermediate variables are the following:

- nparray\_volume\_id
- nparray\_layer\_id
- list\_nparray\_input
- list\_nparray\_output
- i index of hit in df\_hits
- list\_index list of indices in df\_hits for 20 nearest neighbors (nns) by angle to one hit
- df\_bucket
- nparray\_input
- nparray\_particleID
- dict\_particleID\_counterParticleID
- particleIDWithMaxHits
- counterParticleIDWithMaxHits
- list\_output
- nparray\_output

The method of the algorithm is the following.

```
Create empty lists list_nparray_input and list_nparray_output

nparray_volume_id <-- df_hits takes column "volume_id" and convert to nparray
nparray_layer_id <-- df_hits takes column "layer_id" and convert to nparray

for i in list of indices of elements in df_hits:
    list_index <-- from annoy index get the nns for hit with index i and bucketSize
    df_bucket <-- subset of df_hits using indices from list_index
    nparray_input <-- df_bucket take column "x", "y", "z" and convert to flat nparray

    nparray_particleID <-- df_bucket take column "particle_id" and convert to nparray
    dict_particleID_counterParticleID <-- is a dictionary for each particleID (in the bucket)
    counterParticleIDWithMaxHits <-- find max counter of the dictionary above
    list_output <-- create empty list

    for particleID in nparray_particleID (loop over hits in bucket):
        if counterParticleIDWithMaxHits < minValueOfNrHitsForParticleWithMostHits:
```

```

        add to list_output -1 (consider no hit belongs to a particle)
    else:
        if articleID==particleIDWithMaxHits:
            add to list_output +1 (consider this hit belongs to the particle)
        else:
            add to list_output -1 (consider this hit not belongs to the particle)
    # done loop over hits in bucket
    nparray_output <-- convert list_output into a numpy array

    Add to list list_nparray_input. the element nparray_input
    Add to list list_nparray_output. the element nparray_output
    # done loop over hits in the event

    nparray_input_all <-- convert from list_nparray_input
    nparray_output_all <-- convert from list_nparray_output

    return nparray_input_all, nparray_output_all

```

### 7.1.3 Algorithm 3

The Algorithm 3 describes how the 100 events are split 70% into the Train sample and 30% in the Test sample.

```

for i, eventNumber in list_eventNumber:
    df_hits_recon <-- pd.read.csv (file ending in “_hits.csv”)
    df_hits_truth <-- pd.read.csv (file ending in “_truth.csv”)
    df_hits <-- concatenate (df_hits_recon, df_hits_truth)

    nparray_position <-- from df_hits take columns “x”, “y”, “z” as numpy arrays
    index <-- use Annoy library to build an index (sorting hits per direction)
    nparray_input_all, nparray_output_all <-- from df_hits and index
    (see this function implemented in Algorithm 1)

    # keep only number of buckets multiple of 10
    nbBucket=nparray_input_all.shape[0]
    rest=nbBucket%10
    if  rest<7:
        add event to Train
    else:
        add event to Test
    # done for loop over event

```

## 7.2 Model Evaluation Metrics

In this section the pseudo code for the metric evaluation of the model is presented.

In the first algorithm, a histogram is built across buckets for multi-label classification metrics at bucket level, across the 20 hits in a bucket: Accuracy, Precision, Recall, Positive, Negative, Predicted Positive, Predicted Negative. The algorithm is implemented in a function that is called for both Train and Test.

The input variables are the following:

- nparray\_Output
- nparray\_PredictedOutput

The output variables are the following:

- nparray\_bucket\_OutputPositive
- nparray\_bucket\_OutputNegative
- nparray\_bucket\_PredictedOutputPositive
- nparray\_bucket\_PredictedOutputNegative
- nparray\_bucket\_TruePositive
- nparray\_bucket\_FalsePositive
- nparray\_bucket\_FalseNegative
- nparray\_bucket\_TrueNegative
- nparray\_bucket\_acc
- nparray\_bucket\_accuracy
- nparray\_bucket\_precision
- nparray\_bucket\_recall

The intermediate variables are the following:

- nparray\_bucket\_Output
- nparray\_bucket\_PredictedOutput
- counter\_hit\_TP (per bucket)
- counter\_hit\_FP (per bucket)
- counter\_hit\_FN (per bucket)
- counter\_hit\_TN (per bucket)
- TP True Positive (per hit)
- FP False Positive (per hit)
- FN False Negative (per hit)
- TN True Negative (per hit)
- bucket\_OutputPositive
- bucket\_OutputNegative

- bucket\_PredictedOutputPositive
- bucket\_PredictedOutputNegative
- bucket\_TruePositive
- bucket\_FalsePositive
- bucket\_FalseNegative
- bucket\_TrueNegative
- bucket\_acc, bucket\_accuracy
- bucket\_precision
- bucket\_recall

The method of the algorithm is the following.

```
Create an empty list for every metric
nbBucket=number of rows in nparray_Output
for i in range(nbBucket): (for loop over buckets):
    nparray_bucket_Output=nparray_Output[i]
    nparray_bucket_PredictedOutput=nparray_PredictedOutput[i]
    # Initialize to zero counters of number of hits for each confusion matrix element (TP,FP,
    counter_hit_TP=0
    counter_hit_FP=0
    counter_hit_FN=0
    counter_hit_TN=0
    for j in range(len(nparray_bucket_Output)): (for loop over hits)
        # Read values for every hit
        hit_Output=nparray_bucket_Output[j]
        hit_PredictedOutput=nparray_bucket_PredictedOutput[j]
        # initialize matrix confusion element to zero for this hit
        TP=0
        FP=0
        FN=0
        TN=0
        if hit_PredictedOutput>0 and hit_Output>0:
            TP=1
        if hit_PredictedOutput>0 and hit_Output<0:
            FP=1
        if hit_PredictedOutput<0 and hit_Output>0:
            FN=1
        if hit_PredictedOutput<0 and hit_Output<0:
            TN=1
        # increment counters for this hit
        counter_hit_TP+=TP
        counter_hit_FP+=FP
        counter_hit_FN+=FN
        counter_hit_TN+=TN
    # done for loop over hits, for each bucket calculate metrics: accuracy, precision, recall
    bucket_accuracy=(counter_hit_TP+counter_hit_TN)/(counter_hit_TP+counter_hit_FP+counter_hit_FN)
```

```
if counter_hit_TP+counter_hit_FP==0:  
    bucket_precision=0  
else:  
    bucket_precision=(counter_hit_TP)/(counter_hit_TP+counter_hit_FP)  
if counter_hit_TP+counter_hit_FN==0:  
    bucket_recall=0  
else:  
    bucket_recall=(counter_hit_TP)/(counter_hit_TP+counter_hit_FN)  
#  
bucket_TruePositive=counter_hit_TP+counter_hit_FP  
bucket_TrueNegative=counter_hit_FN+counter_hit_TN  
bucket_PredictedOutputPositive=counter_hit_TP+counter_hit_FN  
bucket_PredictedOutputNegative=counter_hit_FP+counter_hit_TN  
bucket_acc=counter_hit_TP+counter_hit_TN  
For each metric append the value for this bucket to its corresponding list  
# done for loop over buckets  
For each metric create a numpy array from the corresponding list
```

This function is run twice, one pe Train or Test. This allows to create a histogram from the numpy array and overlay Train and Test. If the histogram with values between 0 and 20 is divided by the number of buckets (20), the x variable becomes the probability density function with values between 0.0 and 1.0.

# Bibliography

- [1] “The ATLAS Experiment at the CERN Large Hadron Collider”. In: *JINST* (2008). URL: <https://iopscience.iop.org/article/10.1088/1748-0221/3/08/S08003>.
- [2] In: (2020). URL: <https://atlas.cern/>.
- [3] “Luminosity Public Results Run 2”. In: (2018). URL: <https://twiki.cern.ch/twiki/bin/view/AtlasPublic/LuminosityPublicResultsRun2>.
- [4] “CERN Seminar on TrackML: Tracking Machine Learning challenge”. In: (2018). URL: [https://indico.cern.ch/event/702054/attachments/1606643/2561698/tr180307\\_davidRousseau\\_CERN\\_trackML-FINAL.pdf](https://indico.cern.ch/event/702054/attachments/1606643/2561698/tr180307_davidRousseau_CERN_trackML-FINAL.pdf).
- [5] “Presentation at EPS-HEP 2019: Conclusions from TrackMLthe HEP Tracking Machine Learning Challenge”. In: (2019). URL: [https://indico.cern.ch/event/577856/contributions/3423349/attachments/1879205/3095755/190712\\_BASARA\\_EPSHEP\\_TrackML.pdf](https://indico.cern.ch/event/577856/contributions/3423349/attachments/1879205/3095755/190712_BASARA_EPSHEP_TrackML.pdf).
- [6] “Presentation at EPS-HEP 2019: TrackML - the roller coasterof organizinga HEP challenge on Kaggleand Codalab”. In: (2019). URL: [https://indico.cern.ch/event/577856/contributions/3423422/attachments/1879294/3095813/tr1907\\_davidRousseau\\_EPSOutreach\\_TrackML\\_final.pdf](https://indico.cern.ch/event/577856/contributions/3423422/attachments/1879294/3095813/tr1907_davidRousseau_EPSOutreach_TrackML_final.pdf).
- [7] E Bouhova-Thacker et al. “A framework for vertex reconstruction in the ATLAS experiment at LHC”. In: *Physics, Computer Science* (2010). URL: <https://www.semanticscholar.org/paper/A-framework-for-vertex-reconstruction-in-the-ATLAS-Bouhova-Thacker-Koffas/b07cdf945ec5b76f2d1714835f06938be5e7e48a>.
- [8] “Concepts, Design and Implementation of the ATLAS New Tracking (NEWT)”. In: *ATL-SOFT-PUB-2007-007* (2007). URL: <http://cds.cern.ch/record/1020106>.
- [9] “Track reconstructionin the CMS tracker”. In: *CMS-NOTE-2006-041* (2005). URL: [http://cds.cern.ch/record/934067/files/NOTE2006\\_041.pdf](http://cds.cern.ch/record/934067/files/NOTE2006_041.pdf).
- [10] “TrackML Particle Tracking Challenge”. In: (2018). URL: <https://www.kaggle.com/c/trackml-particle-identification>.
- [11] “The TrackML Challenge Grand Finale”. In: (2019). URL: <https://indico.cern.ch/event/813759/>.
- [12] “Acts Common Tracking Software”. In: (2020). URL: <http://cern.ch/acts>.
- [13] “Presentation on ACTS”. In: (2019). URL: <https://indico.cern.ch/event/773049/contributions/3474760/attachments/1937534/3213665/2019-11-05-acts-chep-v14.pdf>.
- [14] “Particle Tracking Machine Learning Challenge: Detector and Dataset”. In: () .
- [15] “The TrackML Particle Tracking Challenge”. In: *hal-01680537* (2018). URL: <https://hal.inria.fr/hal-01680537/document>.
- [16] “Hashing and metric learning for charged particle tracking”. In: *Second Workshop on Machine Learning and the Physical Sciences (NeurIPS 2019), Vancouver, Canada* (2019). URL: [https://ml4physicalsciences.github.io/2019/files/NeurIPS\\_ML4PS\\_2019\\_31.pdf](https://ml4physicalsciences.github.io/2019/files/NeurIPS_ML4PS_2019_31.pdf).

- [17] In: (2020). URL: <https://pypi.org/project/annoy/>.
- [18] Andrew Ng. *Machine Learning*. Coursera online course. URL: <https://www.coursera.org/learn/machine-learning>.
- [19] Adam Gibson and Josh Patterson. *Deep Learning*. URL: <https://www.oreilly.com/library/view/deep-learning/9781491924570/ch04.html>.
- [20] In: (2020). URL: <https://theffork.com/activation-functions-in-neural-networks/>.
- [21] In: (2020). URL: [https://keras.io/api/layers/regularization\\_layers/dropout/](https://keras.io/api/layers/regularization_layers/dropout/).
- [22] In: (2020). URL: <https://keras.io/api/optimizers/adam/>.
- [23] In: (2020). URL: <https://keras.io/api/optimizers/adadelta/>.
- [24] In: (2020). URL: <https://numpy.org/>.
- [25] In: (2020). URL: <https://pandas.pydata.org/>.
- [26] In: (2020). URL: <https://jupyter.org/>.
- [27] In: (2020). URL: <https://matplotlib.org/>.