JUSTUS-LIEBIG-
UNIVERSITÄT
GIESSEN

Employing Deep Learning to Find Slow Pions in the Pixel Detector in the Belle II Experiment

Verwendung von Deep Learning um langsame Pionen im Pixel Detektor des Belle II Experiments zu finden

Master Thesis

by

Johannes Bilk

submitted at
PD Dr. Sören Lange AR
and
Prof. Dr. Claudia Höhne

II. Physics Institute
Faculty 07
Justus-Liebig-Universität Gießen

Abstract (English)

In the context of this thesis it was investigated whether the classification of Slow Pions at the pixel-detector of the Belle II experiment is feasible with methods of artificial intelligence. These Slow Pions experience a large energy loss in the pixel-detector and therefore do not reach the outer detector layers, consequently no charged tracks are available. In this thesis we try to identify these Slow Pions only on the basis of pattern recognition of pixel structures. 78% of all slow pions are found against a large background of other particles. The number of mislabels is 22%.

Abstract (Deutsch)

Im Rahmen dieser Thesis wurde untersucht ob mit Methoden der künstlichen Intelligenz die Klassifizierung von langsamen Pionen am Pixel-Detektor des Belle II Experiments durchführbar ist. Diese langsamen Pionen erfahren im Pixel-Detektor einen großen Energieverlust und erreichen daher nicht die äußeren Detektoren, folglich stehen dann keine geladenen spuren zur Verfügung. Im Rahmen dieser Thesis wird daher versucht die Identifikation dieser langsamen Pionen nur auf Basis von Mustererkennung von Pixelstrukturen durchzuführen. Es werden 78% aller langsamen Pionen gegen einen großen Hintergrund anderer Teilchen gefunden. Der Anteil an Mislabels liegt bei 22%.

Statement of Authorship

I hereby declare that I am the sole author of this master thesis and that I have not used any sources other than those listed in the bibliography and identified as references. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

________________________ ________________________

(Place, Date) (Signature)

Dedication

This is to those who preceded me in my efforts, because without them I would have failed in every regard and to Alfred Wegner[1], who stood up for his scientific findings against the consensus of the wider scientific community.

A dwarf standing on the shoulders of a giant may see farther than a giant himself.

This is to my family bound to me by blood, friendship and love.

I don't know half of you half as well as I should like; and I like less than half of you half as well as you deserve.

[1] Alfred Lothar Wegener (1 November 1880 – November 1930)

Table of Contents

1 Introduction

Nothing surprises me; I'm a scientist.

Dr. Henry Walton Jones Jr.

The Standard Model (SM) is the current gold standard in particle physics, containing the electromagnetic, weak and strong forces and describing the interaction of all elementary particles. Still, it does not give us a complete picture of nature (1). For instance, it does not contain gravity, the weakest of the four fundamental forces, governing the cosmos. General Relativity, the theory describing gravity, was put in a mathematical framework by people like Albert Einstein[2] and Georges Lemaître[3]. The latter corrected Einstein's, self-admitted, biggest mistakes, which gave rise to the theory of the so-called Big Bang[4].

According to the SM matter and antimatter should have been created in equal amounts during the Big Bang, meaning that matter and antimatter is always created and annihilated in pairs. Manifestly the observable universe consists of 5% matter, 27% dark matter and 68% dark energy. This leads to the questions, where did all the antimatter go (1) (2) (3)?

One possible explanation for this discrepancy is an asymmetric decay of matter and antimatter. This problem is related to conversation laws in physics and their violation, namely flavor universality violation, the baryon asymmetry problem and charge-parity (CP) violation. In order to understand these phenomena, we need to investigate them and search for new physics (NP) beyond the SM.

The current frontier of NP is the heavy flavor sector of the SM, in particular B-physics (4) (5) (6). B mesons occur in pairs (7), measuring life time asymmetries in B-decays will give insights into the CP asymmetry (8) and the Higgs[5] sector (9). Together, in a cordial

[2] Albert Einstein (14 March 1879 - 18 April 1955)

[3] Georges Henri Joseph Édouard Lemaître (17 July 1894 - 20 June 1966)

[4] Fred Hoyle: "for it's an irrational process, and can't be described in scientific terms"

[5] Peter Ware Higgs (29 May 1929)

rivalry, BaBar and Belle showed the existence of CP violation in B systems (10). B-physics is currently researched at SLAC, SuperKEKB and LHCb.

There are numerus ways to study B-mesons and the one done here is through so called charged Slow Pions, which exhibit a low transversal momentum. These Slow Pions come from decays of charged excited D mesons called D^* (11) (12), which come from beauty-anti-beauty quark pairs. CP violation in this sector has been predicted to be minuscule (>O(0.01)), CP violation above this level would lead to NP (13).

Having an effective algorithm to find Slow Pions in a large data set will enable us to reconstruct B-decays more proficiently, thus it will give insights into NP beyond the SM. It is not humanly possible to analyze all the data and to look at every data point to infer the rules that created this very data point. We have no way of understanding and deducing conclusions from the data at hand, since we do not have nor do we need to have the ability to look at everything in its entirety. We let computers do the heavy lifting, in not just looking at the data and sorting it, which is the traditional way of analyzing data. We also rely on computers to figure out the rules distinguishing between different kind of events. This approach of computers figuring out the rules themselves is called Machine Learning (ML) and I employed this approach to find Slow Pions in a large data set of different kinds of particles.

2 B-Physics at Belle II

If my calculations are correct, when this baby hits 88 miles per hour, you're gonna see some serious stuff.

Dr. Emmett Brown

The discovery of the positron 1932 by Anderson[6] (14) (15) is considered the beginning of particle physics. Anderson noticed tracks of the same curvature as of electrons in photo emulsion, while tracking cosmic rays. But these track bend in the other direction. The radius and direction of the tracks lead him to the conclusion that there must be other elementary particles of the same weight as electrons, but with the opposite charge.

2.1 SuperKEKB Facility

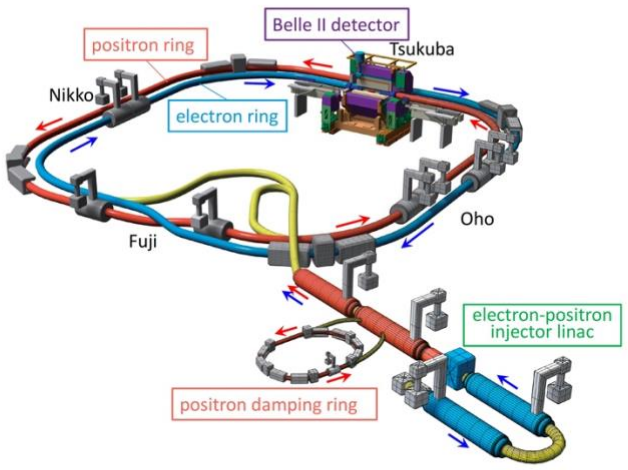


Figure 1: SuperKEKB Facility, consisting of a Positron- and Electron-ring, a damping ring and a linear accelerator and the Belle II decetor system

Figure 1 shows a schematic layout of the SuperKEKB facility with every part labeled. This section is based on *Accelerator design at SuperKEKB* (15), *Report on SuperKEKB phase 2 commissioning* (16) and SuperKEKB Collider (17). SuperKEKB and its

[6] Carl David Anderson (3 September 1905 – 11 January 1991)

predecessor are asymmetric electron-positron ring collider. The KEKB accelerator ran from 1998 until 2010.

The ring is 3 km long, the beam crossing angle is 83mrad in order to keep the beams separated. The blue ring in Figure 1 is the electron ring and has an energy of 7 GeV with a current of 2.6 A. It is also called high energy ring (HER). The Positron ring has an energy of 4 GeV, red in the figure, it has a beam current of 3.6 A. It is called low energy ring (LER). Thus, the center of mass energy is at the $\Upsilon(4S)$ resonance at 10.58 GeV. The Υ are a series of resonances for electron-positron annihilation, it is shown in Figure 2. The difference in energies creates a Lorentz[7] boost of $\beta\gamma = 0.28$, which allows the measurement of decay vertices, precise lifetimes and mixing parameters giving insight into CP violation. Furthermore, there is a linear accelerator (linac) and a damping ring (DR) for the positrons. The boost factor is just two thirds that of KEKB, but the beam pipe of SuperKEKB has a radius of 10 mm around the collision point. This is just two thirds of the beam pipe radius of KEKB. Addtionally, the first two layers of the detector are closer to the beam. These two facts compensate for the smaller boost.

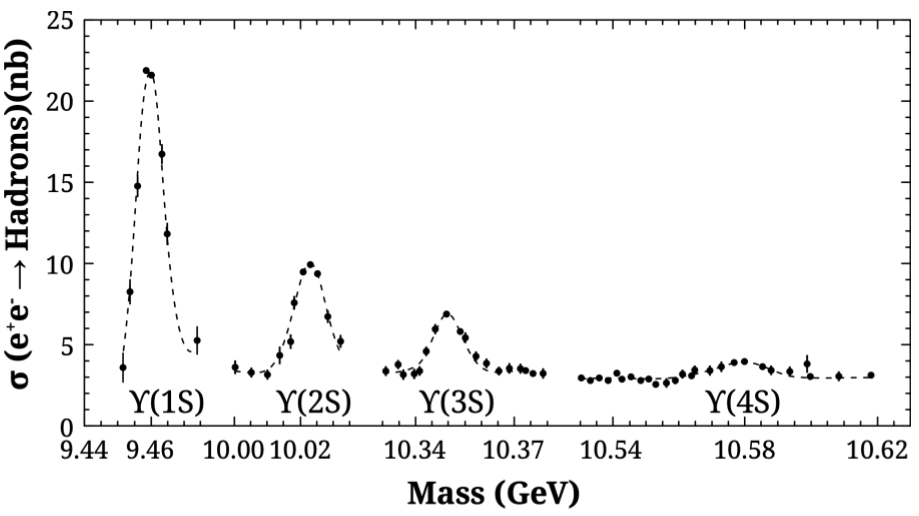


Figure 2: Cross section for electron-positron annihilation (19)

The goal of the upgrade is a luminosity of $8 \times 10^{35} cm^{-2} s^{-1}$, which is forty times the peak luminosity of KEKB. The higher luminosity leads to twenty times larger backgrounds,

[7] Hendrik Lorentz (18 July 1853 – 4 February 1928)

which makes data analysis harder. Luminosity describes the ratio of frequency of detected events to cross-section.

The increased luminosity will be attained by the nano-beam scheme, as was proposed by Raimondi, a depiction is shown in Figure 3. This means the beam emittance is decreased and the current is slightly raised, while having a large crossing angle ϑ at the collusion point.

Given the Lorentz[8] factors $\gamma_\pm$, the beam sizes $\sigma^*_{x,y}$ at the collision point, the beam currents $I_\pm$, a beam-beam tune shift $\xi^*_{y\pm}$ and two geometry correction factors R_L and R_{ξ_y} one can calculate the luminosity by:

$$L = \frac{\gamma_\pm}{2er_e}\left(1+\frac{\sigma^*_y}{\sigma^*_x}\right)\left(\frac{I_\pm \xi_{y\pm}}{\beta^*_y}\right)\left(\frac{R_L}{R_{\xi_y}}\right)$$

The + denotes positrons and – denotes electrons in the equation above. The vertical beta function β^*_y, describes the thickness of the beam. The bulk of increase in luminosity will be achieved by minimizing this factor.

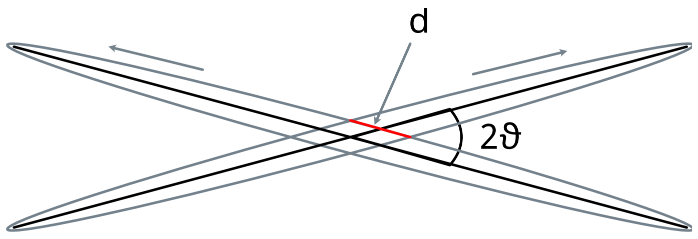


Figure 3: The Nano-Beam Scheme

2.2 Belle II & Its Subdetectors

2.2.1 The Subdetectors

This section is taken from the third chapter of *The Belle II Physics Book* (18) and from *Belle II Technical Design Report* (19). The Belle II detector is a system of multiply subdetectors. Figure 4 shows the Belle II detector system and how each part are situated in the whole gestalt.

[8] Hendrik Antoon Lorentz (18 July 1853 – 4 February 1928)

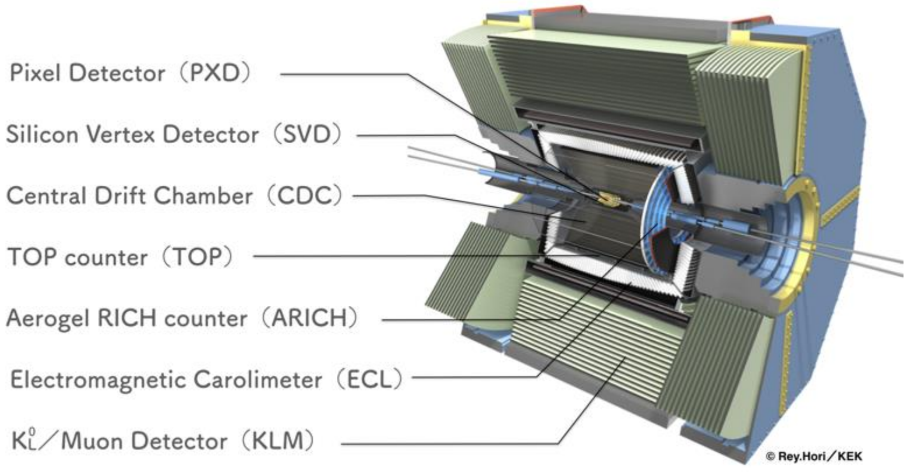

Pixel Detector (PXD)
Silicon Vertex Detector (SVD)
Central Drift Chamber (CDC)
TOP counter (TOP)
Aerogel RICH counter (ARICH)
Electromagnetic Carolimeter (ECL)
K$_L^0$/Muon Detector (KLM)
© Rey.Hori/KEK

Figure 4: Belle II with Subdetector Labels (18)

Now following the subdetectors from the inner most to the outer most, as seen in Figure 4, I will describe each very briefly.

PXD	The pixel detector (PXD) is the inner most detector and it consists of two layers of pixelated silicon sensors with 14 mm and 22mm radii around the beam pipe. It has 10M readout channels.
SVD	The silicon vertex detector (SVD) is made up of four layers of double-sided silicon strips with 39 mm, 88 mm, 104 mm and 135 mm radii. It has 224k readout channels and 1902 readout chips with a fast-shaping time of $\mathcal{O}(50ns)$. Apart from measuring B decay vertices, it looks at decay channels containing D mesons and τ leptons.
CDC	The central drift chamber (CDC) has three main tasks, it reconstructs the 3D helix paths of charged particles and identifies them and uses this information to issue data taking triggers for the other detectors. Its inner radius is 160 mm and the outer radius is 1130 mm. It has 14k readout channels for 14.336 sense wires, made of tungsten and 42,240 field wires, made of aluminum. The chamber is filled with He-C_2H_6 gas.
TOP	The time-of-propagation (TOP) detector consists of 16 quartz glass bars and it has a time resolution of 100 ps. Each quartz bar is about 260 cm × 45 cm × 2 cm big. It has 8k readout channels. Its purpose is to identify charged

particles and separate Kaons from Pions using Cherenkov[9] radiation in the barrel region, the part surrounding CDC.

ARICH The aerogel ring-imaging Cherenkov (ARICH) detector sits at the end of CDC in direction of the electron beam. Like TOP it is used to identify charged particles and to separate Kaons from Pions with an energy resolution of 0.4 GeV up to 4 GeV. This detector has 78k readout channels.

ECL The electromagnatic calorimeter (ECL) detects gamma rays and mainly separates Electrons from Pions and other hadronic matter. It is made up of 8736 thallium doped cesium iodide crystals with a total weight of 43 tons. ECL is 3 m long and has an inner radius of 1.25 m. It has 8.7k read out channels. It measures the luminosity.

KLM The outer most detector identifies K_{Long} and Muons (KLM). It is made up of alternating 14 iron plates and 15 active detector plats, each with a thickness of 4.7 cm. In this manner it can precisely measure hadronic showers. It has 17k readout channels.

The trigger (TRG) and data acquisition (DAQ) system are of further importance, especially in regards to the topic of this work. TRG needs to be efficient in order to fulfill limitations imposed by technical constraints of the DAQ.

The trigger system is built up of several sub-triggers and a final-decision logic. If CDC measures at least three tacks and ECL sees larger energy deposition, their sub-triggers actuate, than the global decision logic makes a decision to issue a global trigger or not. In this manner background events, which are characterized by two or less tracks in CDC, will be suppressed. This system can actuate at a rate of 30 kHz.

Upon an issued trigger by this hardware trigger (L1-trigger), the DAQ takes in the data and a software-based trigger system (HLT) will reduce the trigger rate down to 10 kHz in order to store the data.

[9] Pavel Alekseyevich Cherenkov (28 July 1904 – 6 January 1990)

2.2.2 PXD – Pixel detector

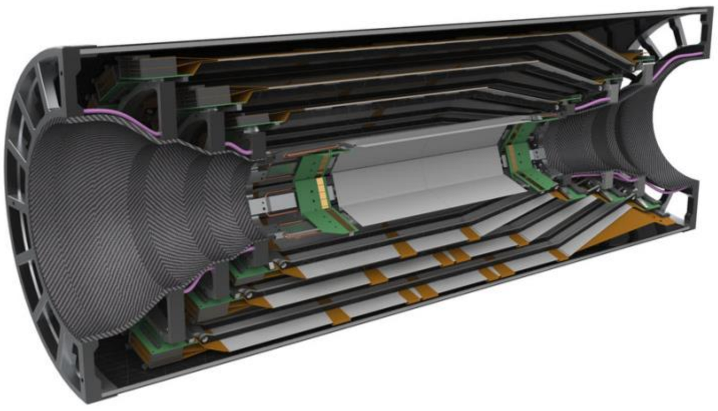


Figure 5: SVD and PXD Subdetectors (20)

This section is based on chapter three of *The Belle II Physics Book* (18), the fourth chapter from *Belle II Technical Design Report* (19) and *Online-analysis of hits in the Belle-II pixeldetector for separation of slow pions from background* (7). Figure 5 shows how PXD is nested in SVD. The PXD is the closest to the beam and it is not included in the trigger system, this is due to the large number of pixels.

The PXD consists of two depleted field effect transistors (DEPFET) silicon layers with a thickness of 75 µm. The inner layers, called modules, have a size of 1.5×6.8 cm^2 and the outer ones have a size of 1.5×8.5 cm^2, each pixel has a size of 0.0025 mm^2. How the PXD models are arranged is depicted in Figure 6. The modules themselves have very little power draw and can be easily air cooled, but the readout electronics, will need to be actively cooled.

The higher luminosity will lead to higher occupancy within the system. In order to deal with this, the number of pixels needs to be high. Every module has a resolution of 256×768 pixels. The readout happens in a rolling shutter manner with 100 ns per pixel row. The total readout time is 20 microseconds.

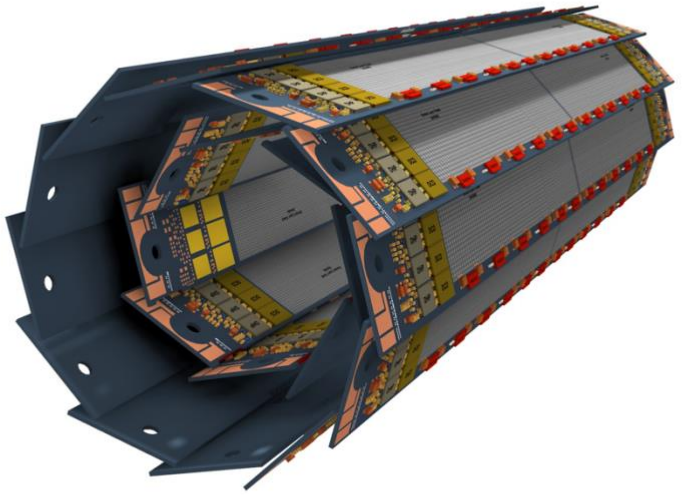


Figure 6: The PXD Modules (40)

The data generated by PXD can reach up to 28 Gbit/s overwhelming DAQ and the trigger system. The data rate coming from PXD is 1 MByte/event it is needs to be reduced by a factor of ten down to 100 kByte/event, this is in part a technical limitation of gigabit ethernet. What happens is that HLT extrapolates from the other detectors to issue a region of interest (ROI) for PXD.

2.3 B-Physics at Belle II

Красота спасёт мир

The Idiot

The b-quark was discovered 1977 at Fermilab by a group led by Leon Lederman[10] (16) (17). They were studying muon-anti-muon pairs. The group found the so called Υ (Upsilon)[11] resonance, which was made up of a new kind of quark-anti-quark pair. This new quark was dubbed 'beauty' or 'b-quark'. The Υ resonance was already mentioned in an earlier section. It is shown in Figure 2.

The B meson was discovered 1983 at CLEO (25). CLEO was an electron-positron accelerator operating at the $\Upsilon(4S)$ resonance, the last peak in Figure 2. The research

[10] Leon Max Lederman (15 July 1922 – 3 October 2018)

[11] Internally they called it 'Oops-Leon'

team was looking for simple decay modes with D mesons and one or two charged Pions. They observed decays of B mesons, namely:

$$B^- \to D^0\pi^-$$
$$\bar{B}^0 \to D^0\pi^+\pi^-$$
$$\bar{B}^0 \to D^{*+}\pi^-$$
$$B^- \to D^{*+}\pi^-\pi^-$$

This section is based on the chapters two, seven, eight and seventeen of *The physics of the B factories* (10) and chapters eight and nine of *The Belle II Physics Book* (18).

The $\Upsilon(4S)$ resonance produces B meson pairs without fragmentation particles, creating clean samples. The $B^0\bar{B}^0$ and B^+B^- pairs are in the quantum state 1^{--}, thus the initial state is well known, allowing analysis methods like missing mass. Which we can calculate by this formular:

$$M_{miss}^2 = \left(E_{\Upsilon(4S)} - \sum_{n=1}^{N_t} E_n\right)^2 - \sum_{n=1}^{N_t} |p_n|^2$$

Belle II looks at different kinds of decays. Here follow a few example decays. Figure 7 shows a purely hadronic B decay.

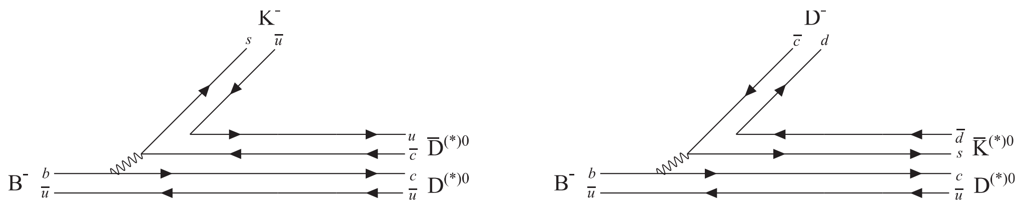


Figure 7: Example Feynman diagrams for hadronic B decays (26)

Fully hadronic means, as can be seen in the figure, that all decay products are of hadronic matter. Figure 8 shows a fully leptonic decay. In these decays we have only leptons in the final states.

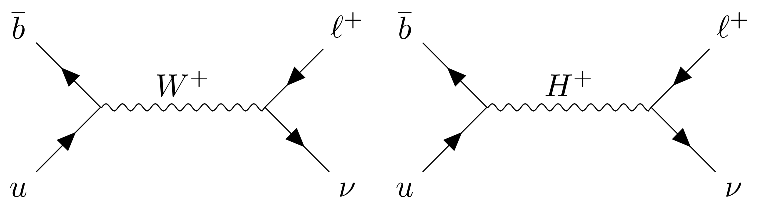


Figure 8: Example Feynman diagrams for leptonic B decays *(10)*

Figure 9 shows a semi-leptonic decay, which decay through first order weak interaction and are governed by W bosons. These decays include hadrons and leptons in their final states.

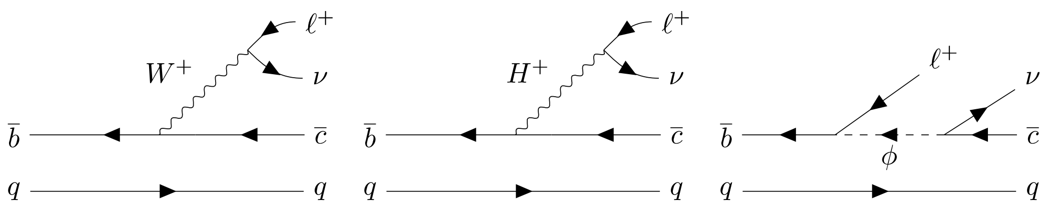


Figure 9: Example Feynman diagrams for semi-leptonic B decays (10)

Finding new physics (NP) in leptonic and semi-leptonic decays will be hard, since it is heavily suppressed within the SM. Thus, is makes sense to look for these decays involving tau leptons, which might give insights into process outside of the SM.
Figure 10 shows a Feynman diagram for BB oscillation. The boosted topology allows to measure the oscillation frequency of neutral B mesons with d-quarks. But the asymmetry is not enough for B mesons with s-quarks, which oscillate at higher frequencies. These kinds of decays are of interest, because of an asymmetry in oscillation.

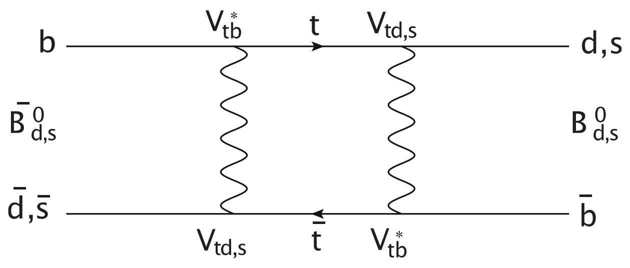


Figure 10: BB oscillation at lowest order diagram (10)

Transitions of the kind $b \rightarrow s$ or $b \rightarrow d$ are called flavor changing neutral current (FCNC). It can be seen in the figure above. These decays proceed through so called penguin or box diagrams and currently only Belle II can measure these processes.
Table 1 shows the for this work relevant particles, it gives some of the important characteristics.

Table 1: Tabulation of the for this work important particles

	Symbols		*Quarks*		*Mass / MeV*	*Isospin*	*Parity*	*Lifetime / s*
Upsilon 4S	$\Upsilon(4S)$		$b\bar{b}$		9460.30	0	1^-	1.218×10^{-20}
Neutral B	B^0	$\bar{B}^0$	$d\bar{b}$	$\bar{d}b$	5279.61	½	0^-	1.520×10^{-12}
Neutral D	D^0	$\bar{D}^0$	$c\bar{u}$	$\bar{c}u$	1864.84	½	0^-	4.101×10^{-13}
Excited D	D^{*+}	D^{*-}	$c\bar{d}$	$\bar{c}d$	2010.27	½	1^-	6.9×10^{-21}
Pion	π^+	π^-	$u\bar{d}$	$\bar{u}d$	139.57	1^-	1^-	2.603×10^{-8}

2.3.1 Reconstructing B mesons

This section is based on the eighth chapter of *The Belle II Physics Book* (18) and sevens chapter of *The physics of the B factories* (10). In this section I want to give some example decay chains for each of the aforementioned decays. Reconstruction happens through summation of all momenta of all final decay products. Thus, it is only possible thanks to momentum conversation. As was already mentioned $\Upsilon(4S)$ predominately decays into two B mesons. We can take from Table 1, that these B mesons will always have the same mass. This means we need only to reconstruct one of the B mesons, since it makes up half of the center of mass energy. Which in turn is measured through the momenta of the final decay products.

An example of a fully hadronic B meson reconstruction can be:

$$B^0 \to D^{*-}\pi^+$$
$$\hookrightarrow \bar{D}^0\pi^-$$
$$\hookrightarrow K^+\pi^-\pi^0$$
$$\hookrightarrow \gamma\gamma$$

In leptonic and semi-leptonic reconstruction additional constraints are necessary, since they include neutrinos, which do not interact with the detector. An example decay is the following:

$$B^0 \to D^{*-}\ell^+\nu$$
$$\hookrightarrow \bar{D}^0\pi^-$$

$$\hookrightarrow K^+\pi^-\pi^0$$

$$\hookrightarrow \gamma\gamma$$

One uses the missing mass method, since the initial state is known and the final states are all measured. Given the signal B (B_{sig}) and a tag side B (B_{tag}) we can calculate the missing mass:

$$M_{miss}^2 = \left(p_{e^+e^-} - p_{B_{signal}} - p_{B_{tag}}\right)^2$$

Under the assumption, that neutrinos are the only missing mass, one can exploit this method. There is one further method of reconstruction called partial B meson reconstruction, where not all final decay products need to be detected. This increases the reconstruction efficiency manifold. Decays where partial reconstruction is possible involve D^* mesons.

2.3.2 Slow Pions

This section is based on the sixth and eighth chapter of *The Belle II Physics Book* (18), chapter eight of *The physics of the B factories* (10) and *Online-analysis of hits in the Belle-II pixeldetector for separation of slow pions from background* (7). Pions where discovered 1947 by a group led by Cesar Lattes[12] (29) (30).

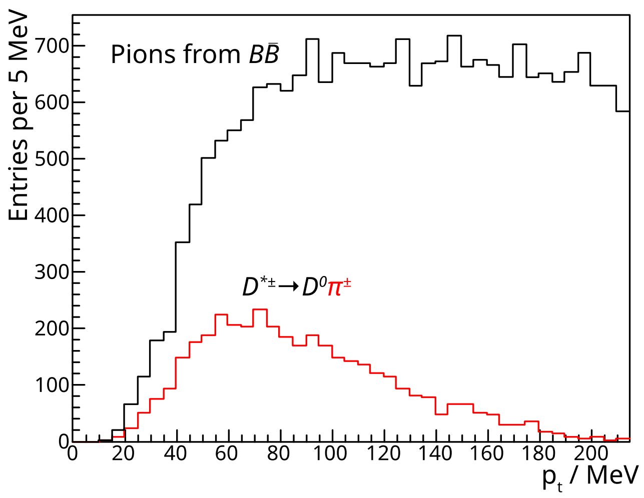


*Figure 11: Slow Pions coming decays of BB pairs vs. D**

[12] Cesare Mansueto Giulio Lattes (11 July 1924 – 8 March 2005)

I mentioned earlier that some of the most important decays of B mesons include D mesons, especially D^*. These are excited D mesons. It is critical to properly tag so called Slow Pions, which is one of their decay products. Slow Pions can come from none D^* decays, but they are regarded as background and thus I am not concerned about them. Slow Pions $\pi^{\pm}_{slow}$ are characterized by a slow transverse momentum, see Figure 11. They are created nearly at rest in the D^* frame, thus they continue its direction together with the D^0. The decay chains I am looking at are:

$$B^0 \rightarrow D^{*-}X^+$$
$$\hookrightarrow \bar{D}^0\pi^-_{slow}$$
$$\bar{B}^0 \rightarrow D^{*+}X^-$$
$$\hookrightarrow D^0\pi^+_{slow}$$

$X^{\pm}$ can stand for any charged meson or a charged lepton and corresponding neutrino. A D^* that decays below 60 MeV mainly decays into a D^0 and a $\pi^{\pm}_{slow}$. This means that most Slow Pions will not reach the outer layers. Figure 12 shows the first five layers of SVD and PXD and how far Pions with different energies reach into the detector. We see that most Pions get stuck in the lower layers. As was mentioned in the section about the Belle II detector, if CDC has two or less tracks for charge particles it will not trigger. It has been suggested to employ an artificial neural network as an online triggering system for PXD in order to find Slow Pions.

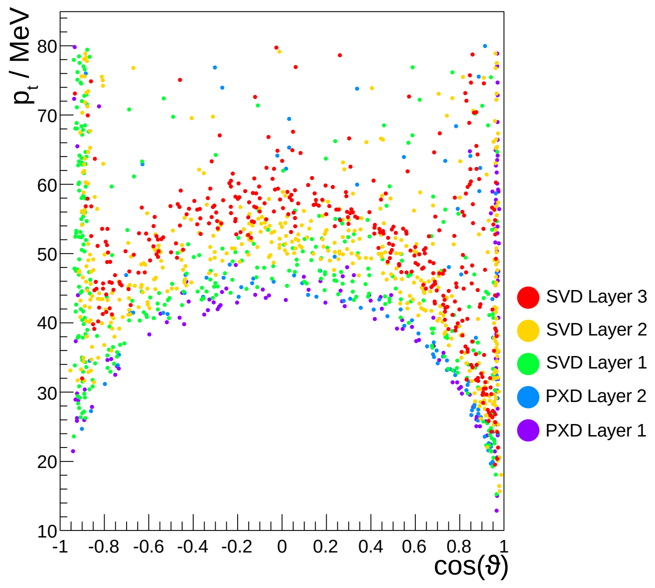


Figure 12: Slow Pions in the first five layers of VXD

It is possible that the D^0 decays futher into a Kaon. In that case one can correlate both mesons and improve the background suppression or rather the reconstruction of the decay channel. Another possibility is $B^0 \to D^* W^\pm$. The W boson than hadronizes into a fast Pion. The angle between the two Pions is large.

2.3.3 Tagging of B mesons

Here I want to make a simplified example of showing how to tag a B meson. Figure 13 shows a semi-leptonic decay of a $B^-(b, \bar{u})$ into a negative lepton, the corresponding anti-neutrino and a placeholder X meson consisting of $\bar{u}$ and q quark.

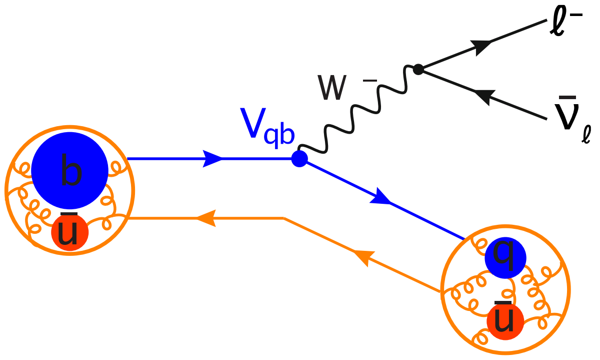


Figure 13: Semi-leptonic decay: $B^- \to X\ell^-\bar{\nu}_\ell$ (10)

The $\bar{u}$ quark stays the same, while the b quark decays into a q quark through radiating a W^- boson, which than decays into the leptons. We can detect the leptons and the X meson and then infer the flavor of the original B meson.

2.4 Simulated data

The data coming from the PXD are represented by 9×9 matrices, which can be interpreted as small pictures, which are considerably smaller than the full PXD module resolution. These are the ROIs that were mentioned earlier; their coordinates are contained within every simulated event. Figure 14 shows the coordinate distribution for all Slow Pions events. The distributions for all other categories can be found in Appendix A. On the left of Figure 14 we see the norm, the two-layer structure of PXD can be easily seen. In the middle we see relatively even distribution of all angles. On the right we see the height distribution and can make out where the interaction point is in relation to PXD.

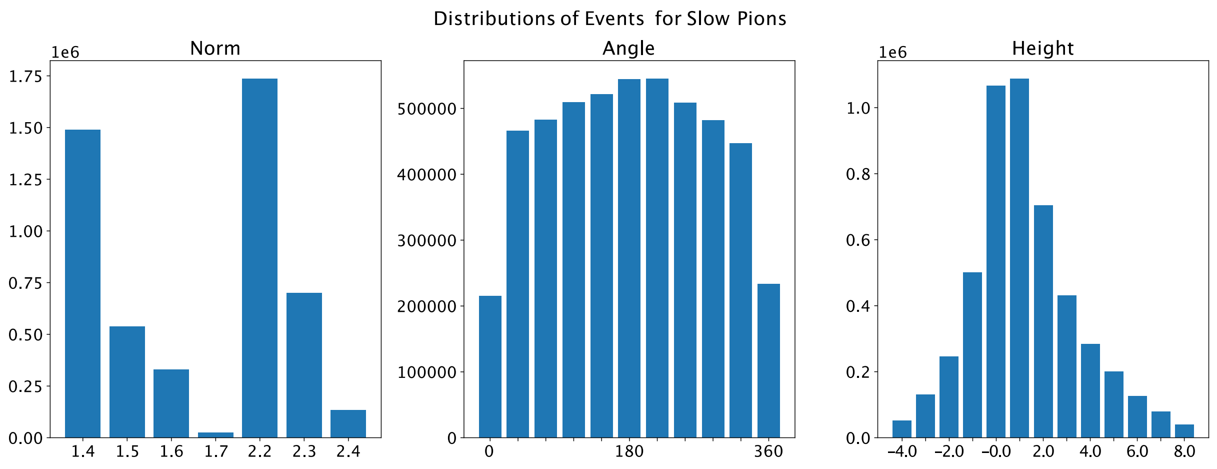


Figure 14: Coordinate Distributions for PXD events of Slow Pions

In Figure 15 example events are shown for different particles and beam background. The data analyzed in this work has been created using Monte-Carlo simulation.

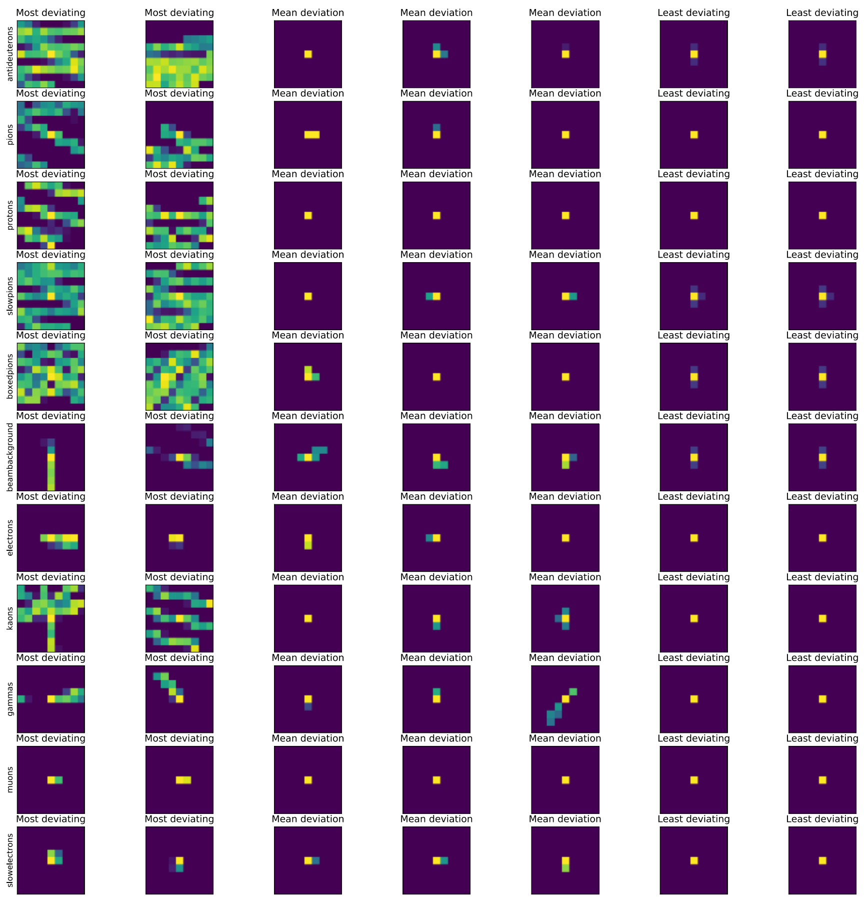


Figure 15: PXD event data, the two left most columns are the most deviating events form the mean of each set, the three columns in the middle deviate from the mean event somewhat and the two right most are the least deviating from the mean event. Rows from the top: Anti-Deuterons, Pions, Protons, Slow Pions, box generated Slow Pions, Beam Background, Electrons, Kaons, Gammas, Muons and Slow Electrons.

The number of events per data set are:

- Slow Pions (SP): 4.957.071 (1.757.348)
- Pions (PI): 911.318 (484.946)
- Anti-Deuterons (DD): 907.168 (365.706)
- Beam Background (BB): 633.283 (142.011)
- Protons (PP): 897.467 (437.956)
- Boxed Slow Pions (BP): 2.911.598 (724.666)
- Electrons (EL): 900.292 (516.605)
- Kaons (KK): 891.969 (516.987)
- Gammas (GA): 13.784 (5.990)
- Muons (MM): 896.921 (527.039)
- Slow Electrons (SL): 1.133.544 (627.185)

with the number of one-pixel events in brackets. This amounts to 15.044.415 simulated events in total and 6.106.439 one-pixel event, where one-pixel events are events where only one pixel of the 81 pixels per events has a non-zero value. Figure 16 shows the total amount of data points per category and the amount of one-pixel events. It is quite obvious, that nearly half of the all normal Pions are one-pixel events and that Slow Pions make up one fifth of the entire data set, if we exclude Boxed Slow Pions. The data set for gammas is about only 1.5% in size of the others, this is due to the fact that gammas rarely interact with the pixel detector.

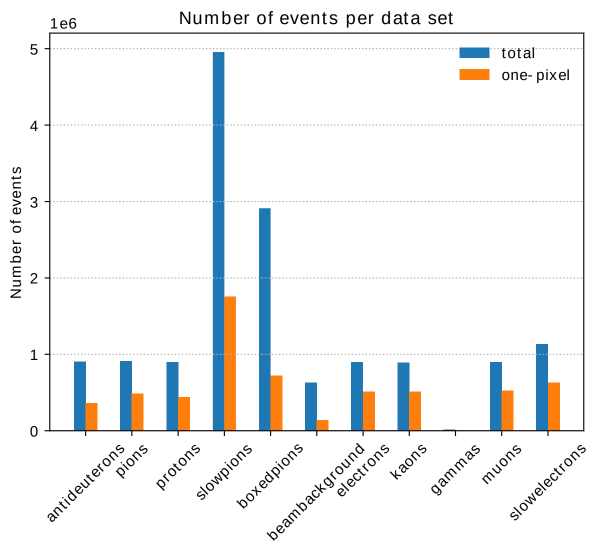


Figure 16: Number of events per data set in total and one-pixel events

I combined these data sets together into four bigger data sets, one combines everything, except Slow Pions, Boxed Slow Pions and Slow Electrons, aptly called Everything (EV). Then there is a data set called Heavy Background (HB), consisting of all the particles made up of quarks, namely Pions, Anti-Deuterons, Protons and Kaons. The next one is called Medium or Meson Background (MB), consisting of Kaons and Pions and the last one is called Light or Lepton Background (LB). This last one is containing Electrons, Muons and Gammas, which are not leptons.

3 The Mathematics Section

You keep using that word. I do not think it means what you think it means.

Inigo Montoya

3.1 Gentle Introduction to Linear Algebra

This section is based on the lecture notes by Max Horn for *Grundlagen der Algebra* by Bernhard Mühlherr and on the book *Mathematische Methoden in der Physik* (24).

Definition 1 (Scalar):

A scalar is a single number. There are natural numbers, which start at zero and are infinitely countable. We can define Integers by introducing the concept of negative numbers. If we then take two integers and divide them by each other we get the rational numbers. And finally, there are real numbers, they are important, because without them there would be no Pi. I only mention this for completeness's sake.

Definition 2 (Vector):

A vector is not a point in space, it is simply an element of a vector space. Vectors contain directional information and obey certain rules about vector operations and scalar multiplications and they are base depended.

In this work it is sufficient to understand a vector as a column:

$$\vec{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

and its dual vector as a row:

$$\vec{y} = \begin{pmatrix} y_1 & \cdots & y_n \end{pmatrix}$$

Definition 3 (Scalar Product):

Given a dual vector, we can define the scalar product, where a vector is mapped to a scalar:

$$\langle \vec{y} | \vec{x} \rangle = \left\langle \begin{pmatrix} y_1 & \cdots & \boldsymbol{y_n} \end{pmatrix} \middle| \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \right\rangle = \vec{y} \cdot \vec{x} = \sum_{i=1}^{n} y_i x_i$$

Scalar products are bilinear, meaning they are linear in both components.

Definition 4 (Norm):

Every scalar product induces also a norm, which just means if we have a scalar product, getting a norm is an easy matter of transposing a vector and scalar producing it with itself:

$$\|\vec{x}\| \overset{\text{def}}{=} \langle \vec{x}^T | \vec{x} \rangle$$

A norm is understood to be a measure of length for a vector.

Definition 5 (Matrix):

The mathematical term here is a linear map, meaning there is a linear correspondence between input and output. A matrix is an element of the so-called linear group. Since we are only interested in concrete representations, a matrix is a quasi-table with n rows and m columns, it takes the form:

$$M = \begin{pmatrix} M_{11} & \cdots & M_{1n} \\ \vdots & \ddots & \vdots \\ M_{m1} & \cdots & M_{mn} \end{pmatrix}$$

Special kinds of matrices are diagonal matrices:

$$M_{diag} = \begin{pmatrix} M_{11} & 0 & \dots & 0 \\ 0 & M_{22} & \dots & \\ \vdots & \vdots & \ddots & \\ 0 & & & M_{mn} \end{pmatrix}$$

The identity matrix is a special diagonal matrix, with only ones:

$$Id = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix}$$

A unitary matrix is a square matrix with a size of $n \times n$ and is defined by:

$$U^T U = Id$$

In the last equation we would first need to define matrix multiplication.

Definition 6 (Vector-Matrix Multiplication):

Here we define how a matrix and a vector can be combined and what the resulting product is. A matrix stands on the left, a vector on the right. The length of the vector must be equal to the number of columns of the matrix. The result will again be a column vector. If the matrix stands right, then the vectors length must be equal to the number of rows, then we will get a row vector.

An example will illustrate this:

$$M\vec{x} = \begin{pmatrix} M_{11} & \cdots & M_{1n} \\ \vdots & \ddots & \vdots \\ M_{m1} & \cdots & M_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^{n} M_{1i} x_i \\ \vdots \\ \sum_{i=1}^{n} M_{mi} x_i \end{pmatrix}$$

Definition 7 (Eigenvalue):

The question is, can we simplify a matrix into a form where only the diagonal elements are non-zero. The answer is, for special matrices we can do that under the condition that all columns and rows are linear independent and that it is a square matrix, meaning $n = m$. The defining equation for an eigenvalue λ then is:

$$M\vec{x} = \lambda\vec{x}$$

Definition 8 (Condition Number):

This is a scalar given by the quotient of the smallest and largest eigenvalue:

$$\kappa(M) = \frac{\lambda_{max}}{\lambda_{min}}$$

if κ is close to one, it means the matrix is well conditioned, if it is a large number it means the matrix is ill conditioned. For dynamic systems it means, that small changes to the system result is large changes in the outcome.

Definition 9 (Matrix-Matrix Multiplication):

Given two matrices A and B of sizes $r \times m$ and $m \times n$ and the resulting matrix C of size $r \times n$. Matrix vector multiplication is a special kind of matrix-matrix multiplication:

$$\begin{pmatrix} a_{11} & \dots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{r1} & \dots & a_{rm} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{m1} & \dots & b_{mn} \end{pmatrix} = \begin{pmatrix} c_{11} = \sum_{i=1}^{m} a_{1i} b_{i1} & \dots & c_{1n} = \sum_{i=1}^{m} a_{1i} b_{in} \\ \vdots & \ddots & \vdots \\ c_{r1} = \sum_{i=1}^{m} a_{ri} b_{i1} & \dots & c_{rn} = \sum_{i=1}^{m} a_{ri} b_{in} \end{pmatrix}$$

We notice that every element of the product is a scalar product of each row times column of matrices A and B.

Definition 10 (Singular Value Decomposition):

Singular Value Decomposition can be done with any matrix, where a single matrix is decomposed into three matrices:

$$M = U\Sigma V^T$$

Where M is a $m \times n$ matrix and Σ is a square diagonal matrix of size $r \times r \leq min(m, n)$. U and V are unitary matrices of size $m \times r$ and $n \times r$ respectively.

Definition 11 (Matrix Inversion):

Matrix Inversion can only be done with square matrices where every column and row are linear independent. Given a $n \times n$ matrix M, the inverse matrix M^{-1} is given by:

$$M \cdot M^{-1} = Id = M^{-1}M$$

This gives us some understanding what a unitary matrix is, it is a matrix where the transposed matrix is equal to its inverse.

Definition 12 (Tensor):

A tensor is a multi linear mapping. In this work it is enough to understand them as multi-dimensional matrices with several indices[13], like T_{ijk} or T^{ijk}; I just wanted to make it clear, that more is a play, mathematically speaking. In Figure 17 is an illustration of a tensor to given intuition what a tensor is, this is a special tensor called full anti symmetric tensor.

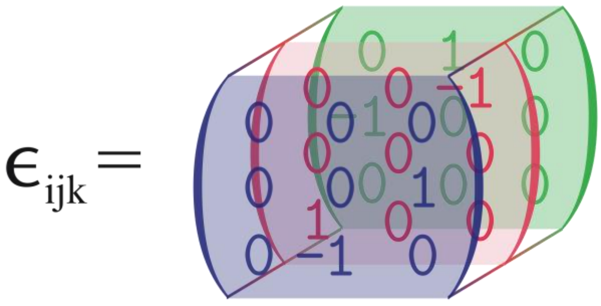


Figure 17: Illustration of a tensor (25)

[13] For some inexplicable reason physics love indices.

3.2 What's so important about derivatives?

The dark side of the Force is a pathway to many abilities some consider to be unnatural.

Chancellor Palpatine

This section is based on the afformentioned book *Mathematische Methoden in der Physik* and *Mathematik für Physiker* (26).

Definition 13 (Continues Function):

A continues function is colloquial speaking a function without gaps. The technical definition is: a function *f* is called continuous, if and only if for every $\varepsilon > 0$ exits a $\delta > 0$, such that:

$$|f(x) - f(a)| < \varepsilon \;\forall\; x \in I \; with \; |x - a| < \delta$$

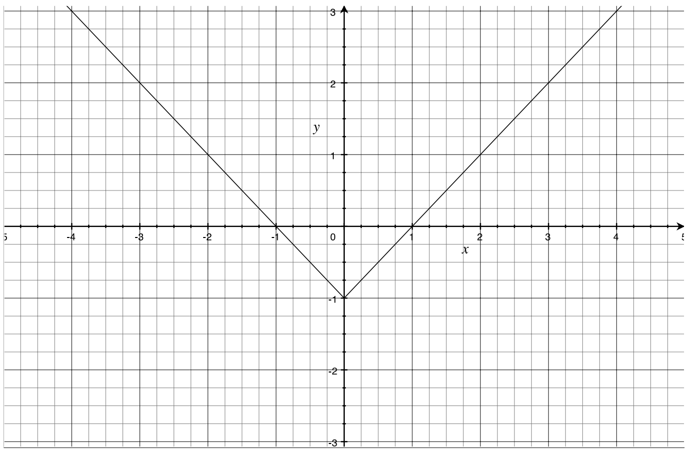


Figure 18: A continues function

Definition 14 (Derivative):

A derivative is defined point wise. The derivative of f in x_0 is given by:

$$f'(x_0) \coloneqq \lim_{I \ni x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

We call *f'(x₀)* the derivative of *f* in *x₀*, we write:

$$f'(x_0) = \frac{d}{dx} f(x_0)$$

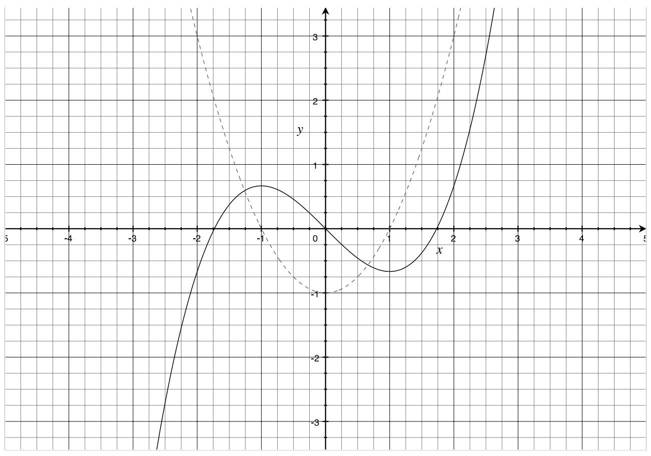


Figure 19: A differentiable function

Figure 18 shows a continues but not a differentiable function and Figure 19 shows a differentiable and continues function. Left of 0 in Figure 18 the function can be characterized as monotonously decreasing function and right of 0 it is a monotonously increasing function.

What happens at the roots of derivatives? The black, solid function Figure 19 has two curious points, at -1 and +1. These points coincide with the roots of the dashed, grey curve in the figure. What happens here is that the dashed function is the derivative of the solid curve, as was defined above and roots of derivatives mark out special points on their respective function. These points are called extrema. Taking the derivative of the derivative will further characterize these points. If this second derivative is negative, meaning the first derivative is continuously decreasing, then the function has a maximum. If the inverse is true, then the function has a minimum. If neither is true, we speak of a saddle point.

What is the Chain Rule? Given two differentiable functions $g: I \to J$ and $f: J \to R$ with $y_0 = g(x_0)$, then the derivative of composite function is:

$$(f \circ g)'(x_0) = f' \underbrace{(y_0)}_{g(x_o)} \cdot g'(x_0)$$

we write:

$$\frac{d}{dx} f\big(g(x)\big) = f'\big(g(x)\big) \cdot g'(x)$$

$g'(x)$ is called the inner derivative and $f'(y_0)$ is called the outer derivative.

Definition 15 (Partial Derivative):

Let *f(x_1, x_2, ..., x_i, ..., x_n)* be multivariable function, then we define that partial derivative with respect to x_i, *i* between 1 and n, as:

$$f_{x_i}(x_1, x_2, \dots, x_i, \dots, x_n) = \frac{\partial f}{\partial x_i}$$

Definition 16 (Nabla):

Nabla is multi-dimensional differential operator, usually represented as a vector:

$$\vec{\nabla} \overset{\text{def}}{=} \begin{pmatrix} \frac{\partial}{\partial x_1} \\ \vdots \\ \frac{\partial}{\partial x_n} \end{pmatrix}$$

What is a gradient? Given a function *f*: $\mathbb{R}^n \to \mathbb{R}$, the *gradient* of it is defined as:

$$\nabla f(x_1, x_2, \dots, x_n) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

Definition 17 (Jacobi Matrix):

Given a function *f*: $\mathbb{R}^n \to \mathbb{R}^m$, the *Jacobi matrix* of it is defined as:

$$Jf \overset{\text{def}}{=} \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & & \ddots & \vdots \\ \vdots & & & \\ \frac{\partial f_m}{\partial x_1} & & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix}$$

Definition 18 (Hessian Matrix):

Given a function *f*: ℝⁿ → ℝ, the *Hessian matrix* of it is defined as:

$$Hf \overset{\text{def}}{=} \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & & \ddots & \vdots \\ \vdots & & & \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

Thanks to Schwarzes Theorem we only need to calculate the diagonal and either the lower left or upper write half of the Hessian, since second derivatives are symmetric, meaning:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial^2 f}{\partial x_j \partial x_i}$$

What is a Lagrangian[14]? The idea is to minimize the action or energy while moving from a fixed starting point to a fixed end point. This is done infinitesimal variation on a path between the two points until a minimum is found. The definition is given by:

$$S[x] = \int_{t_1}^{t_2} \mathcal{L}(x(t), \dot{x}(t), t) dt$$

where $\mathcal{L}(x(t), \dot{x}(t), t)$ is the Lagrangian. The $x(t)$ which minimizes $S[x]$ is given by:

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{x}}\right) - \frac{\partial \mathcal{L}}{\partial x} = 0$$

The dot indicates a time derivative.

[14] Giuseppe Luigi Lagrangia (25 January 1736 – 10 April 1813)

4 Artificial Intelligence

Þægiattv, vǫlva!
þik vil ek fregna,
vnz alkvnna,
vil ec ænn vita
Óðinn

4.1 About Machine Learning

Machine Learning (ML) is a general and broad term applied to all kinds of computer analysis application in which computers learn the analysis parameters themselves. Figure 20 is a depiction of how to think about ML, above we see the traditional approach to computer analysis of data, below the ML approach. In the traditional approach data and rules are used as input to get answers. In ML we try to extract the rules, but only implicitly and not explicitly, since they are not of interest. (27) (28)

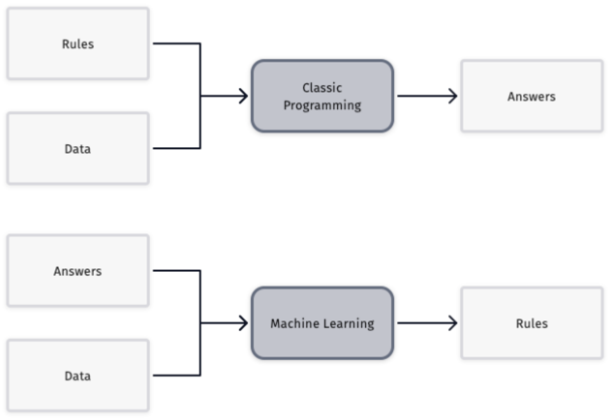


Figure 20: Illustration on machine learning (27)

Machine Learning has been employed in particle and high energy physics since the 1990s (29) (30). It has been applied as a real time, online trigging system and successfully in offline reconstruction of data (31).

There are several subdivisions within Machine Learning. Canonically there are three branches and sometimes a fourth one, namely supervised, unsupervised, reinforced and self-supervised machine learning (28) (27) (32). Relative recently appeared also a new variant called *inverse reinforcement learning* (33). I will summarize their characteristics here based on these three books *Deep learning with Python* by Francois Chollet, *Deep Learning* by Ian Goodfellow and *grokking Deep Learning* by Andrew W. Trask.

4.1.1 Deep Learning

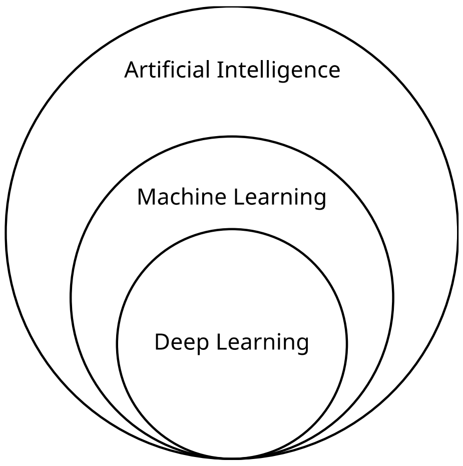


Figure 21: Nestedness of Machine Learning (27), (32), (28)

Deep Learning is the newest and current term for a specific kind of Artificial Intelligence or Machine Learning systems. The descriptor *deep* is a reference to the numerous amounts of parameters this kind of Machine Learning exibits. Figure 21 shows the relation between the different fields of computer implemented Artificial Intelligence. Where Deep Learning is a special case of Artificial Intelligence and ML.

4.1.2 Historical Overview

Deep Learning, or as it was known during its early years *Cybernetics*, has its beginning in the 1940s. Ever since then it waxed and waned in popularity over the decades (28). It was inspired by biological neurons (34) and based on mathematical models to describe how neurological systems learn (28). Machine Learning exists in its modern form since the late 1970s and had a resurgence in recent decades, starting in the early 2000s (28) as computational power became strong enough.

The architecture of the first networks was similar to what we have nowadays, but the weights of each neuron had to be adjusted by hand. It only functioned as a binary categorizer. Just a decade later in the late 1950s the perceptron became the first self-adjusting model. An integral part of Machine Learning known as *stochastic gradient decent* was what made the weight adjustment of a model called *adaptive linear element* (ADALINE) possible. (28)

The second wave of Deep Learning came about in the 1980s due to the advancements in computational power. Its name then changed to *connectionism* or *parallel distributed processing*. A key insight was to break up each problem into small parts, which together can solve more complex problems. This trend lasted until predictions made by researches about Machine Learning did not come true in the mid 1990s. (28)

Today neuroscience remains an inspiration for the development of artificial neural networks and led to the development of convolutional neural networks for image recognition (27).

4.1.3 Supervised Learning

This is probably the most common form of Machine Learning and its prime example are artificial neural networks, such as the one I employ in this work. In supervised learning we have a data set with labels or targets, the algorithm is running over this prelabeled dataset trying to make a prediction. This prediction then is compared to the target and based on how correct or wrong the machine performed, it adjusts the parameters of the algorithm to make better predictions. This form of Machine Learning is employed in written or spoken language recognition, language translation and image classification.

4.1.4 Unsupervised Learning

The classic example of unsupervised learning is a self-organizing map. Here a computer is not given any labels or targets, but instead is supposed to find the topology of the input itself. Meaning the algorithm tries to find clusters or groupings based on the features of a given data set. It is mainly employed in compression, reduction of dimensionality or image denoising.

4.1.5 Self-Supervised Learning

This category is often subsumed by either supervised or unsupervised learning, since it is very similar to both of them. Here a computer is given a data set without any labels, but it generates the labels itself. It is a kind of supervised learning, but without human intervention, still it retains the characteristics of supervised learning, such as making a prediction based on past data. An example for self-supervised lerarning are autodecoders.

4.1.6 Reinforced Learning

This is still a developing branch of Machine Learning. Here we put an agent in an arena and set up rewards and punishments for certain actions. It is easier to understand with an example where it is employed. Reinforced learning is used to train computers at playing games, be it video games, Chess or Go and other board games. So far, its scope of application is rather limited.

On a side note, I would like to mention the Frame Problem in this context. The easiest description is, that it is not possible to write out in closed form all equations governing the non-consequences of actions by an agent in an arena. Furthermore, the search space of all possible interactions, for example threads to the agent, is combinatorial explosive. Meaning an agent would have to check the probability of an infinitude of possible problems at every step. This is closely related to the subject of relevance realization. (35) (36)

4.1.7 Inverse Reinforcement Learning

Inverse reinforcement learning takes inspiration from how children learn. Instead of directly instilling behavior, rules or rewards and punishments, machines are supposed to observe humans and imitate their behavior. (33)

4.2 About Artificial Neuronal Networks

4.2.1 Overfitting & Underfitting

In ML it is important to solve a given problem sufficiently and not to *optimize* a network too stringent on the data set in order not to lose its *generality*. Optimal in the sense of performing an analysis on the training data set without making mistakes and general in the sense of making the least number of mistakes on a test data set (27).

We can define this with understanding two terms central to evaluate a neural network. Given the error on a training set, the error on the test set and the difference between these error values, we can define **underfitting** as (28):

not minimizing the error on the training set

and **overfitting** as (28):

not minimizing the difference between training and test error.

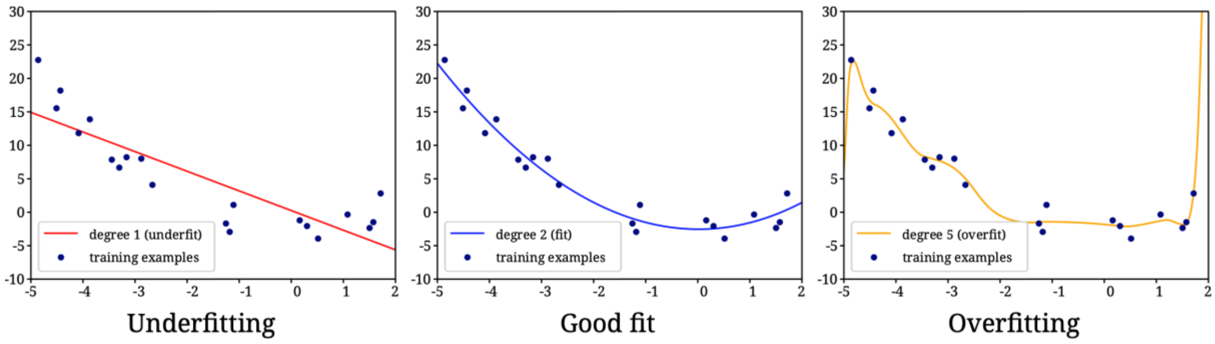


Figure 22: Illustration of under- and overfitting (37)

The processes of under- and overfitting can be understood from Figure 22, on the left we see underfitting in action, where we try to fit a sample with a linear function. On the right we see overfitting in action, where every data point is hit relatively well, but this fit will not be able to accommodate more data points, as they will fall far away from the fit curve. In the middle we see an optimal fit, as the error is relatively low and still new points will fall relatively close to the curve.

Overfitting happens when one trains on a too small data set, whereas underfitting happens, when the the network is trained too short and/or on a too small data set (28) (27).

4.2.2 The Black Box Problem

As already mentioned above, we only know the rules of analysis implicitly, if at all in any capacity. Hence the black box problem refers to the fact, that we do not actually know how a neural network solves a concrete problem. That does not mean that we do not understand the principles at work, but the whole process of learning is happening in a completely transparent manner. Put in another manner, we can understand every step, but understanding the neural network taken together becomes impossible, because of the number of parameters. (33)

4.2.3 The Alignement Problem

Und nun komm, du alter Besen!
Nimm die schlechten Lumpenhüllen;
Bist schon lange Knecht gewesen;
Nun erfülle meinen Willen!
Auf zwei Beinen stehe,
Oben sei ein Kopf,
Eile nun und gehe
mit dem Wassertopf!
The Sorcerer's Apprentice

As Machine Learning grows in power, complexity and applicability, one inherent problem is becoming more and more apparent. We give a machine a set of problems, or an arena to act in, and a set of instructions with rewards and punishments. These machines start tackling the problems with our guidelines and after a while we will discover, that despite the directives we have given it, the machine is not doing what we wanted it to do. There occurs a mismatch between our proposed goals and the consequences of the incentive structure we set up. This problem is called *The Alignment Problem* and it has far reaching implications in many fields of Machine Learning. (33)

4.3 The Ingredients of a Neural Network

In this section it will become clear why I introduced some mathematical basics. We needed the background information in order to understand the basics of artificial neural networks. Most of this information here is based on the books *Deep Learning with Python* by Francois Chollet (27), *grokking Deep Learning* by Andrew W. Trask (32), *Deep Learning* by Ian Goodfellow, Yoshua Bengoi and Aaron Courville (28), *The hundred-page machine learning book* by Andriy Burkov (37) and two MIT lectures by Lex Fridman. Any supplementary sources will be mentioned explicitly.

4.3.1 Activation Functions

An activation function is at the end of each neuron and maps the value of the neuron onto another real number. Activation functions have to be continuous, but not differentiable and they have to be monotonic, but not strictly. These terms have been discussed in *The Mathematics Section*.

The need for activation functions arises out of necessity to analyze data with non-linear correlations. Each neuron in a linear layer is a simple linear mapping from input to output and thus can only represent linear dependencies, this does not change with adding more layers. An activation function introduces this needed non-linearity.

Figure 23 shows the activation functions, which I tested. The formulae are as following:

- LeakyReLU: $LeakyReLU(x) = \begin{cases} x & if\ x \geq 0 \\ 0.01 \times x & otherwise \end{cases}$
- ReLU: $ReLU(x) = \max(0, x)$
- Sigmoid: $Sigmoid(x) = \frac{1}{1+\exp(-x)}$
- Tangent Hyperbolic: $Tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$
- Softmax: $Softmax(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$
- Identity just leaves the value untouched

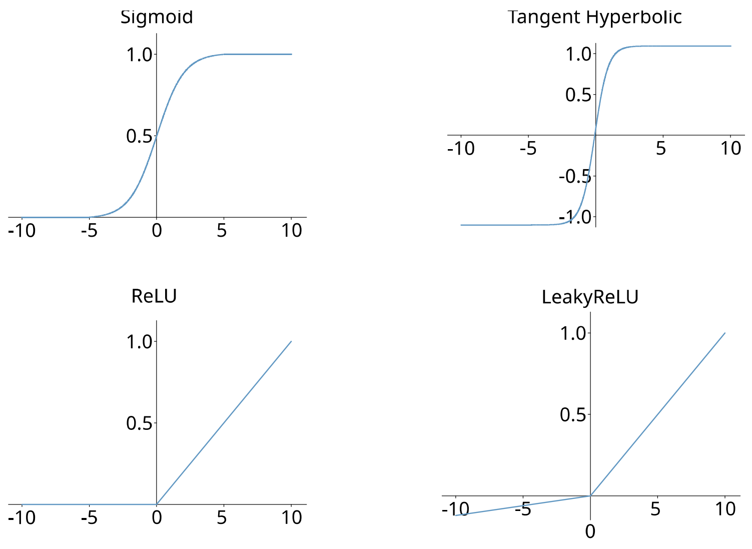


Figure 23: Different Activation Functions (38)

A full list of Activation Functions contained in PyTorch can be found in the PyTorch Documentation (39).

4.3.2 Loss Function

A neural network takes all input data, PyTorch calls these tensors and runs it through all neurons and at the end it makes a prediction. This prediction then is compared to the target for that data and a loss is calculated how far the prediction is from the target. The function determining this distance is called a *Loss Function*. This is where the learning part in Machine Learning comes into play, now the machine tries to adjust all weights and biases to minimize this loss or error. A single run over all data points is called an epoch, training a network successfully takes several epochs. How many can only be determined through training. (27) (32) (40)

In this work I only used the categorical *Crossentropyloss* function and the information about it are lifted from the PyTorch Documentation (41). *Crossentropyloss* is a combination of logarithmic softmax and function called negative log likelihood loss. For every category *i* the final loss function is:

$$CE = -\sum_{i}^{\overbrace{C}^{classes}} \underbrace{t_i}_{target} \log \underbrace{s_i}_{prediction}$$

Another typical, but not suitable for my proposes, loss function is Mean Squared Error:

$$MSE = \frac{1}{N}\sum(t_i - s_i)^2$$

4.3.3 Optimizer & Gradient Descent

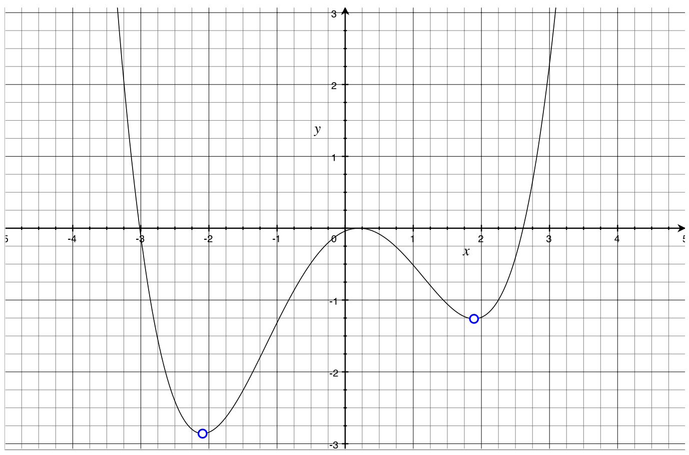


Figure 24: Showing the difference between a local and global minimum

For the equations and mathematics, I consulted the following articles:

(42) (43) (44) (45) (46)

Just calculating the error made on a data set is not enough for learning to have an effect. The optimizers task is it to update the weights and biases of the network based on the loss value calculated through the loss function. One has to be aware of the difference between a local and global minimum. This is shown in Figure 24, where the point on the right is only a local minimum and not a global. It can always happen that the optimizer ends up in the local minimum and it will stop optimzing. (47)

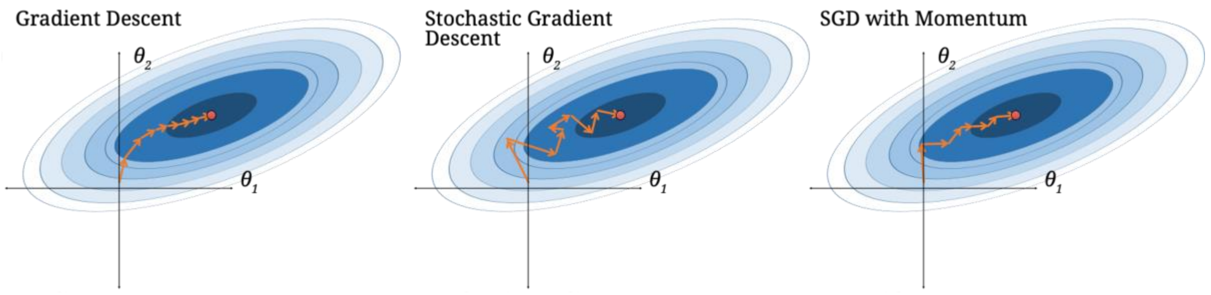


Figure 25: Comparision of Gradient Descent, Stochastic Gradient Descent and SGD with Momentum (44)

There are several optimizers, the most popular are (48):

Gradient Descent (GD)

This is the simplest optimizer, it updates every paratmer θ just by the error-/loss-gradient times a scaling factor μ called the learning rate:

$$\theta_{t+1} = \theta_t - \mu \cdot \nabla_\theta Error(\theta)$$

The memory requirements are high, since all data points are taken into account and updates are infrequent. This means it will ploddingly converge into the minimum. Also they have the same learning rate for all parameters, which causes problems with sparse data sets or data with a wide range of differing frequencies or where the weight matrix is ill conditioned. Further the learning rate is fixed, which can be compensated by using a scheduler to adjust the learning rate. These usually adjust the learning rate depending on the epoch and not based on convergence. How this algorithm reaches a loss minimum is shown on the left in Figure 25. This algorithm was suggested long before ML was conceived by a mathematician by the name Augustin-Louis Cauchy[15].

Stochastic Gradient Descent (SGD)

Here we only test for a smaller sample x_i, y_i of the data set, which reduces the amount of memory needed, while at the same time increasing the update frequency. The size of these samples is called batch size. But this means the optimizer will oscillate. This can be seen in the middle in Figure 25. The oscillation might lead to overshooting the minimum.

$$\theta_{t+1} = \theta_t - \mu \cdot \nabla_\theta Error(\theta, x_j, y_j)$$

As with Gradient Descent we have only a single and fixed learning rate for all parameters.

Stochastic Gradient Descent with Momentum (SGD with Momentum)

In order to rectify the oscillation of *SGD* a momentum γ was introduced:

$$v_t = \gamma v_{t-1} + \mu \cdot \nabla_\theta Error(\theta, x_i, y_i)$$
$$\theta_{t+1} = \theta_t - v_t$$

[15] Augustin-Louis Cauchy (21 August 1789 – 23 May 1857)

This additional hyperparameter leads to faster convergence, but it is one more value that needs to be fiddled with. With this optimizer the learning rate also stays fixed. A comparison with the previous two optimizers is shown in Figure 25, SGD with Momentum is in the right.

Adam

Here a first and second order momentum m_t and v_t are introduced based on the gradient g_t, the parameters β_1 and β_2 lay between zero and one:

$$\begin{aligned}
g_t &= \nabla_\theta Error(\theta_t) \\
m_t &= \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t \\
v_t &= \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2 \\
\widehat{m}_t &= \frac{m_t}{(1-\beta_1^t)} \\
\hat{v}_t &= \frac{v_t}{(1-\beta_2^t)} \\
\theta_t &= \theta_{t-1} - \mu \cdot \frac{\widehat{m}_t}{(\sqrt{\hat{v}_t} + \epsilon)}
\end{aligned}$$

the parameters m_t and v_t are set to zero initially and they tend to stay close to zero, this is why $\widehat{m}_t$ and $\hat{v}_t$ are used to compensate for that. This optimizer converges fast, but at the cost of computational intensity.

AdaGrad

Gradient Descent and all modifications of it have the problems that the learning rate is fixed and that there is a single learning rate for all parameters. This is compensated in this optimizer by scaling the learning rate by G_t and a tiny stability constant ϵ:

$$\begin{aligned}
g_{t,i} &= \nabla_\theta Error(\theta_{t,i}) \\
\theta_{t+1,i} &= \theta_{t,i} - \frac{\mu}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}
\end{aligned}$$

G_t is diagonal matrix, containing the square sums of previous gradients, thus the learning rate is scaled according to the gradient. The learning rate falls too fast and the network stops learning, because the square sums coalesce to a too large sum over time.

AdaDelta

AdaDelta introduces a moving average of the square root sums $E[g(w)^2]$, it has a cut off w for how far into the past it goes and it has a scaling factor γ similar to *SGD* and *Adam*:

$$E[g(w)^2](t) = \gamma E[g(w)^2](t-1) + (1-\gamma)g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\mu}{\sqrt{E[g(w)^2](t) + \epsilon}} \cdot g_t$$

This prevents the accumulation of gradient values and thus the learning rate does not fall too fast. This optimizer is computationally intensive.

Root Mean Square Proverbialities (RMSprop)

The last of the popular optimizers resembles *AdaDelta* insofar that it uses a moving average of square sums of previous gradients.

$$E[g^2](t) = \gamma E[g^2](t-1) + (1-\gamma)\left(\frac{\partial Error}{\partial \theta}\right)^2$$

$$\theta_{ij}(t) = \theta_{ij}(t-1) - \frac{\mu}{\sqrt{E[g^2](t)}} \frac{\partial Error}{\partial \theta_{ij}}$$

In Figure 25 is a comparison between *GD*, *SGD* and *SGD with Momentum* and how they each try to find the minium of the loss function. *RMSprop* is a generally recommended optimizer for a wide range of problems (27).

A more advanced optimizer has been proposed, which takes the Hessian of the loss function into account and can theoretically find the global minimum of the loss function in fewer steps, but at the costs of higher computational requirements. It is called *AdaHessian* and employs an approximation of the eigenvalues of the Hessian Matrix (49).

4.3.4 What is an artificial neuron?

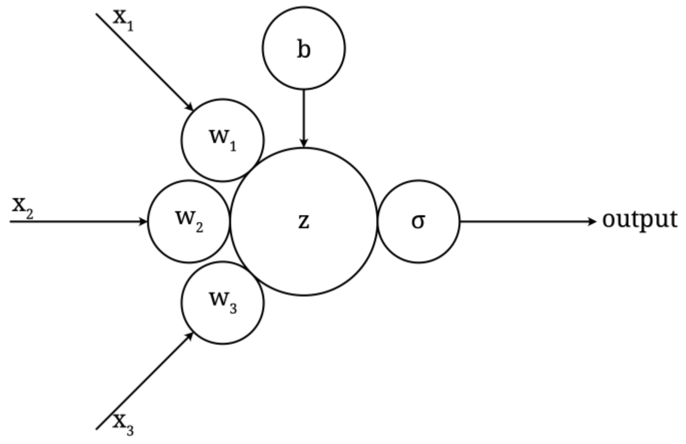


Figure 26: An Artificial Neuron (34)

In Figure 26 is a schema of an artificial neuron. In the figure we see three inputs x_i, which are just numbers. They are multiplied by a weight w_i and all of this is summed up in z, where we also add or subtract a bias b and send that to the activation function σ. The final output is then calculated as:

$$output = \sigma\left(b + \sum_{i=1} x_i w_i\right)$$

4.3.5 Linear layer

A neuron has one output, hence a layer has as many outputs as it has neurons and each neuron has as many inputs as the previous layer has neurons. The first layer has as many inputs as there are features in the data set and the last layer has as many neurons as there are categories.

Handling a data set with widely varying features can be difficult. There are strategies, called regularization that can help with getting a grip on the data. I will talk about two of these techniques since they are the two that I tested and used. They are called out *dropout rate* and the *batchnorm* and third one is called *L1 & L2 regularization*.

The easiest way to prevent overfitting is to introduce *dropout rates*, where random neuros are set to 0. This has the effect that for each input only a subset of the network will be trained and overfitting will be prevented since smaller networks cannot capture as many details of a data set. An illustration of this is shown in Figure 27, where random neurons are set to 0. (28) (27) (32)

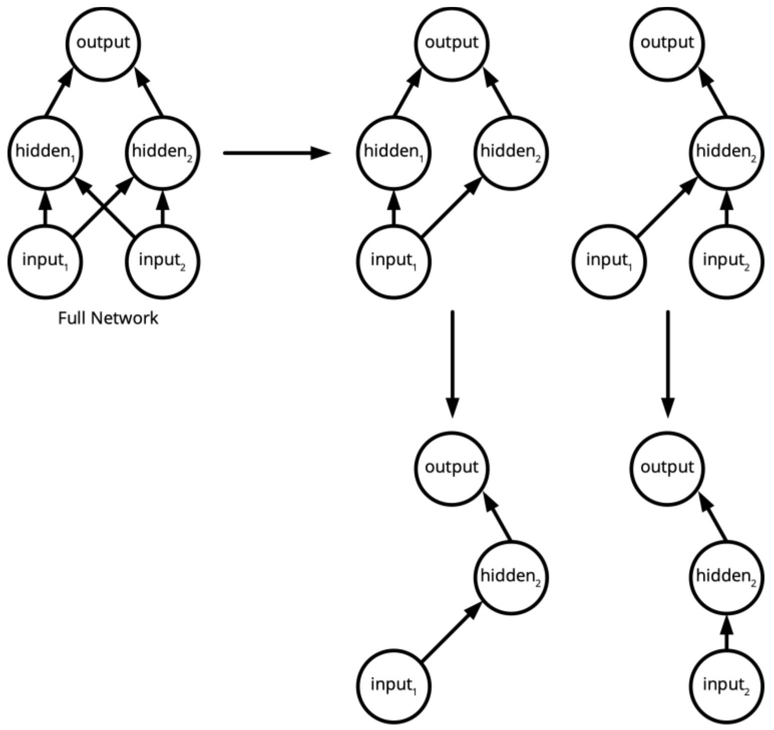


Figure 27: Illustration of dropout, on the left without dropout and on the right different examples of subnetworks (28)

The second strategy is to normalize each batch in each layer. This helps already in smaller networks with the gradient and finding the loss minimum, but for larger networks it is even necessary in order to train them. It has been empirically shown, that batch norm helps with convergence. (27)

Let us turn our attention to what happens in a linear layer. Given a layer of m neurons and n neurons on the previous layer, we can write all weights W of one layer into a matrix and all biases B of the same layer into a vector:

$$W = \begin{pmatrix} W_{11} & W_{12} & \dots & W_{1n-1} & W_{1n} \\ W_{21} & W_{22} & \dots & & \\ & & & & \\ \vdots & & & \ddots & \vdots \\ W_{m1} & & \dots & & W_{mn} \end{pmatrix} \qquad B = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{m-1} \\ b_m \end{pmatrix}$$

Together with an input vector we can calculate the output as:

$$output = weights \cdot input + bias$$

$$= \begin{pmatrix} W_{11} & W_{12} & \dots & W_{1n-1} & W_{1n} \\ W_{21} & W_{22} & \dots & & \\ \vdots & & & \ddots & \vdots \\ W_{m1} & & \dots & & W_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{m-1} \\ x_m \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{m-1} \\ b_m \end{pmatrix}$$

$$= \begin{pmatrix} b_1 + \sum_{i=1}^{n} W_{1i}x_i \\ b_2 + \sum_{i=1}^{n} W_{2i}x_i \\ \vdots \\ b_{m-1} + \sum_{i=1}^{n} W_{m-1i}x_i \\ b_m + \sum_{i=1}^{n} W_{mi}x_i \end{pmatrix}$$

Something curios happened here. Our input is not a matrix, which we would expect if given an image but a vector. What happened here is that the matrix was reshaped into vector $n \times m \to 1 \times n \cdot m$, in this transformation no information was lost, since all values and their relations were kept. Here we have just matrix-vector multiplication. This output will be ran through an activation function and used as input for the next layer.

4.3.6 Convolutional layer

The introduction of convolutional layers lead to breakthroughs in digital image recognition. These layers look at parts of an image with a running filter or kernel and map this filter into one pixel of a target image. The best way to understand what they are doing is to look at the illustration in Figure 28, where we have an input on the left. In the middle we see the filter, sometimes called a kernel, and an output on the right side. This has two effects, first is that it denoises images and second they learn local patterns within images unlike linear layer which only learn global characteristics (27) (28). A convolutional layer has the following arguments (50):

- Input size
- Output size
- Filter or kernel size
- Stride
- Padding
- Pooling

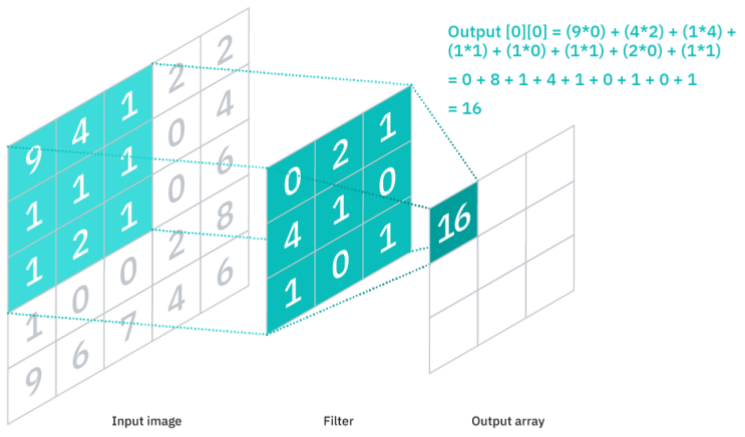

Output [0][0] = (9*0) + (4*2) + (1*4) + (1*1) + (1*0) + (1*1) + (2*0) + (1*1)
= 0 + 8 + 1 + 4 + 1 + 0 + 1 + 0 + 1
= 16
Input image
Filter
Output array

Figure 28: How a convolutional layer works (51)

Input size are the dimensions and number of channels of the input images, output size is in how many channels the output image should be decomposed. The filter size determines how many pixels will be enrolled into one. Larger filters allow looking at larger features, but they lose the ability to abstract features out of their position. The inverse is true for smaller filters and one has to balance the filter size in accordance to the input image. The number of channels can be interpreted as with how many filters the layer is looking at a given input and thus more channels will find more shapes within the image. Stride is the step size of the filter or how many pixels the filter jumps if set to one every pixel will be looked at. Padding is the amount of pixel padding around the image in order to maintain image size. One can pad the image with just zeros or simply extent the border pixels further. Finally pooling averages several pixels into one. (50)

A beautiful way of showing how convolutional layers work is shown in Figure 29. We see how each layer breaks down the bicycle into parts. On the top we have the bike, then one step down we get the frame, a wheel, the saddle and finally on the button each single component, that makes up a bike.

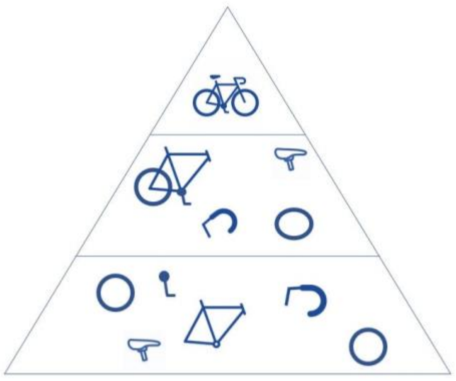


Figure 29: The breaking down of a bike into its components (51)

Convolution in mathematics is defined as (28):

$$s(t) = \int x(a)w(t-a)da = (x * w)(t)$$

This is one dimensional and continues and since we are concerned with two dimensional bitmaps, we will change to a discrete sum and convolute in two directions (28):

$$S(i,j) = (I * K)(i,j) = \sum_{m,n} I(m,n)K(i-m,j-n)$$

If we want to maintain images size, we can employ padding, which creates a border around the image. This can just be the outer values mirrored or just filling in zeros. The padding size p is then, given a kernel size f (52):

$$p = \frac{f-1}{2}$$

Given an input size n_{in}, a kernel size f, padding p and stride s, one can calculate the output size n_{out} (52):

$$n_{out} = \frac{n_{in} + 2p - f}{s} + 1$$

Stride is the speed or step size at which the filter moves over the image. Here is an example of how convolution works. It is taken from (37):

$$\left(\underbrace{\begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}}_{input\ image} * \underbrace{\begin{bmatrix} -1 & 2 \\ 4 & -2 \end{bmatrix}}_{filter} \right) + \underbrace{1}_{bias} = \underbrace{\begin{bmatrix} 4 & -1 & 7 \\ 2 & 7 & 0 \\ 0 & 4 & -1 \end{bmatrix}}_{output\ image}$$

$$\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} -1 & 2 \\ 4 & -2 \end{bmatrix} \rightarrow 1 \cdot (-1) + 0 \cdot 2 + 1 \cdot 4 + 0 \cdot (-2) + 1 = 4$$

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} -1 & 2 \\ 4 & -2 \end{bmatrix} \rightarrow 0 \cdot (-1) + 0 \cdot 2 + 0 \cdot 4 + 1 \cdot (-2) + 1 = -1$$

$$\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix} * \begin{bmatrix}-1 & 2\\4 & -2\end{bmatrix} \rightarrow 0\cdot(-1)+1\cdot 2+1\cdot 4+0\cdot(-2)+1=7$$
$$\begin{bmatrix}1 & 0\\1 & 1\end{bmatrix} * \begin{bmatrix}-1 & 2\\4 & -2\end{bmatrix} \rightarrow 1\cdot(-1)+0\cdot 2+1\cdot 4+1\cdot(-2)+1=2$$
$$\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix} * \begin{bmatrix}-1 & 2\\4 & -2\end{bmatrix} \rightarrow 0\cdot(-1)+1\cdot 2+1\cdot 4+0\cdot(-2)+1=0$$
$$\begin{bmatrix}1 & 0\\0 & 0\end{bmatrix} * \begin{bmatrix}-1 & 2\\4 & -2\end{bmatrix} \rightarrow 1\cdot(-1)+0\cdot 2+0\cdot 4+0\cdot(-2)+1=7$$
$$\begin{bmatrix}1 & 1\\0 & 1\end{bmatrix} * \begin{bmatrix}-1 & 2\\4 & -2\end{bmatrix} \rightarrow 1\cdot(-1)+1\cdot 2+0\cdot 4+1\cdot(-2)+1=0$$
$$\begin{bmatrix}1 & 0\\1 & 0\end{bmatrix} * \begin{bmatrix}-1 & 2\\4 & -2\end{bmatrix} \rightarrow 1\cdot(-1)+0\cdot 2+1\cdot 4+0\cdot(-2)+1=4$$
$$\begin{bmatrix}0 & 0\\0 & 1\end{bmatrix} * \begin{bmatrix}-1 & 2\\4 & -2\end{bmatrix} \rightarrow 0\cdot(-1)+0\cdot 2+0\cdot 4+1\cdot(-2)+1=-1$$

in our example the equation for the output size is as follows:

$$n_{out} = \frac{4+2\cdot 0-2}{2}+1=3$$

4.3.7 Transposed Convolutional Layer

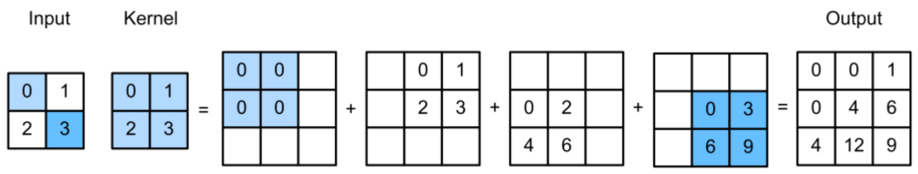


Figure 30: How Transposed Convolution works (53)

A method of upscaling an image is called transposed convolutional layer. The name is a bit misleading, since it is not a convolution at all. In a convolution several pixels are multiplied by a matrix and summed into a single pixel. In principle it works similar to a convolutional layer, but instead of compressing the image size, it increases the image size. But it is important to understand that it is not the inverse of a convolution, hence it is called transposed convolution and not deconvolution. It still retains all the parameters of a convolutional layer, but the filter is applied in such a manner, that it scales up the image and not down. (54)

With transposed convolution we take a single pixel and multiply it with a matrix. Then we patch these matrices together by summing the overlaps. In Figure 30 is an example how this process works. We do this to up-sample the image, but unlike other up-sampling algorithms, transposed convolution has learnable parameters. (55)

4.3.8 Other Layer Types

There are other layer types, like *recurrent neural network (RNN)* and *Long short-term memory (LSTM)*, but I did not implement them, since they are geared towards sequential data types like language or video. It could still be a subject for further research, if these types of layers can be employed in data analysis for the PXD. (27) (28)

4.3.9 The Basic Working Principle of a Neural Network

For the mathematics and the equations of backpropagation I reference (56) (57). Backpropagation is the process of adjusting the weights of the network to minimize the errors. This task is taken care of by optimizers. While I already hinted at how a neural network learns, I will summarize the principle at this point. The basic workflow is shown in Figure 31 and one can follow along each step laid out there.

The first step is initializing the network. Each weight and bias can be initialized either with zeros, randomly or through modified random manners called He or Xavier. The latter two take the sizes of different layers into account (58).

Xavier and He initializes the weights through a uniform distribution U and scaled by the size of the previous layer and biases are set to 0 (59):

$$W_{ij} \propto U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$$

It was empirically shown that this initialization leads to better training results (59).

We can imagine a layer as a function $f(x) = y$ which maps an input x onto an output y. The concept of deep learning is then introduced by chaining several layers or functions together (28):

$$f(x) = f^3\big(f^2(f^1(x)\,)\big) = y = f^1 \circ f^2 \circ f^3(x)$$

By now it should be clear why we talked about the chain rule in the mathematics section, since for backpropagation we need to apply it here:

$$f'(x) = f^{3'}\big(f^2(f^1(x)\,)\big) \cdot f^{2'}(f^1(x)\,) \cdot f^{1'}(x)$$

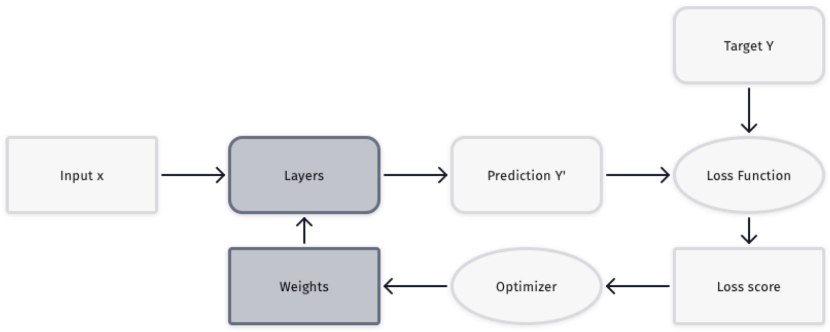


Figure 31: Basic work principle of neural network (27)

The network naïvely reads in the first data batch in form of a series of vectors:

$$x^{(1)} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

the index *m* is the input size. The input layer has *m* inputs and *n* neurons:

$$x^{(2)} = \sigma\left(\sum_{j=1}^{n}\left(b_j^{(1)} + \sum_{i=1}^{m} w_{ji}^{(1)} \cdot x_i^{(1)}\right)\right)$$

The first hidden layer has m inputs and l neurons:

$$x^{(3)} = \sigma\left(\sum_{j=1}^{l}\left(b_j^{(2)} + \sum_{i=1}^{n} w_{ji}^{(2)} \cdot x_i^{(2)}\right)\right)$$

We can generalize this step:

$$x^{(k)} = \sigma\left(\underbrace{\sum_{i=1}^{m^k}\left(b_i^{(k-1)} \sum_{j=1}^{m^{(k-1)}} w_{ij}^{(k-1)} x^{(k-1)}\right)}_{=z^k}\right) = \sigma\left(z^k\right)$$

This prediction will in all likelihood be completely off, but this is just the starting point. In my project I will employ a *softmax* activation function on the last layer. It means the norm of the output will be equal to one, thus making it a probability for each category. The prediction is compared to the target in the loss function:

$$E(x) = \sum_{i=1}^{classes} t_i \log_{10} x_i$$

Its score is then given to the optimizer, which then adjusts the weights and biases in order to minimize the loss for the next batch. This step is called backpropagation, because the error is propagating back through the network. Now we want to know the amount of error caused by every weight and bias, which is given by:

$$\frac{\partial E}{\partial \theta_{ij}^{(k)}} = \frac{\partial E}{\partial z_i^{(k)}} \cdot \frac{\partial z_i^{(k)}}{\partial \theta_{ij}^{(k)}}$$

For that we need something called the local gradient of layer k:

$$\delta^{(k)} = \nabla_{x^{(k)}} E$$

Our task is to calculate the local gradient of each neuron i in layer, but we will start with the last layer:

$$\delta_i^{(L)} = \frac{\partial E}{\partial z_i^{(L)}} \overset{\substack{Chain\\ rule}}{\hat{=}} \frac{\partial E}{\partial x_i^{(L)}} \cdot \frac{\partial a_i^{(L)}}{\partial z_i^{(L)}} = \frac{\partial E}{\partial x_i^{(L)}} \cdot \sigma'\left(z_i^{(L)}\right)$$

The next step is to go one layer deeper. The error here is determined by ${}^{\partial E}\!\big/_{\partial x_i^{(k)}}$ which depends on all previous outputs; hence we get in a similar manner as above:

$$\delta_i^{(k)} = \sum_j^{n^{(n+1)}} \underbrace{\frac{\partial E}{\partial z_j^{(k+1)}}}_{=\delta_j^{(k+1)}} \cdot \underbrace{\frac{\partial z_j^{(k+1)}}{\partial z_i^{(k)}}}_{\theta_{ji}^{(k+1)} \cdot \sigma'\left(z_i^{(k)}\right)} = \sum_j^{n^{(n+1)}} \theta_{ji}^{(n+1)} \sigma'\left(z_i^{(k)}\right) \delta_j^{(k+1)}$$

This gives us finally:

$$\frac{\partial E}{\partial w_{ij}^{(k)}} = \sigma_j^{(k-1)} \delta_i^{(k)}$$

Now we know how much we have to change the weights and biases to make a smaller error in the next epoch. The new values for each weight and bias are given by:

$$\theta_{t+1} = \theta_t - \mu \nabla_{\theta_t} E$$

This is then repeated until all batches have been processed and then again until all training cycles, called epochs, have been run. In the last step it is important to stress, that this is a simple gradient decent. We could have used another and more complex algorithm. (28) (27) (32)

4.3.10 A List of Hyperparameters

Table 2 contains a list of hyperparameters concerning neural networks. These are the numerical values, that fully characterize a neural network and its learning process. It is based on a table from (28) which I extended a bit to the best of my understanding.

Table 2: List of hyperparameters to tune (28)

Hyperparameter	*To increase capacity*	*Reason*	*Premonition*
Number of hidden layers	increase	more details can be encoded, enables non-linear analysis	increases calculation time and needs longer training
Number of Neurons per Layer	increase	more details can be encoded	increases calculation time
Dropout rate	decrease	finer details can be captured	learning rate needs to be adjusted to compensate for overfitting
Learning rate	tune	lower learning rates stave off over fitting, too low learning rates lead to underfitting	lots of testing needs to be done to find a proper learning rate
Decay of learning rate	tune	allows higher learning rates, while preventing overfitting in the long run	lots of testing needs to be done to find a proper decay rate
Filter size of convolutional layers	increase	the right filter size can capture image features perfectly	too large filter compress images and lead to huge information loss

Number of channels	increase	more patterns can be found in input	increases the computational load
Image padding	increase	compsates for loss in case of larger filters	it only mitigates information loss and not curtail it
Momentum	tune	larger momentum leads to faster convergence	with too large momentum the optimizer will overshoot minima
Batch size	tune	smaller batch sizes increase convergence, larger batch sizes allow more generalization	small batch sizes can lead to overfitting, while larger one lead to underfitting
Epochs	depends		

4.4 Python and PyTorch

Python is a high-level scripting language, still maintaining object orientation and is aimed at non-computer-science scientists. Its main goal thus is to be easily learnable, shedding the complexity of the more traditional languages and producing readable code. One big advantage of Python over other high-level languages is its extensibility; modules for Python can be written in Python or in C and interact directly with Python, this makes extending Python easy, while maintaining speed and reliability. Because of the high-level character, its intended target audience, namely scientists and its extensibility make it suitable for this project.
Python was first released 1991, the next verion (2.0) was released 2000 and the current release (3.0) is from 2008. (60) (61) (62)

PyTorch is an open-source framework for developing neural networks and more general for Machine Learning. It was developed by Facebook and first released in 2016 (63).
The syntax and language design of PyTorch is similar to Python, which makes it highly approachable to newcomers, if they already have some knowledge in Python. Additionally, PyTorch has a large community, which can help in case of problems or issues. It has a stronger recommendation for scientists over its alternative Tensorflow. PyTorch is regarded as faster and it allows more control over the neural network than Tensorflow (64) (65) (66) (67).
The reasons why I choose Python over any other language, for example something like C++, which is tremendously faster, was that I could develop and test my code quicker. In other words, Python and PyTorch allowed me to iterate in shorter cycles. Also it is more adaptable and easier to read. Especially the employment of PyTorch to develop the artificial neural networks made it incredible easy to do so. Networks coded in PyTorch performe very well, since Python is only used to create the architecture, while underneath Nvidia CUDA and C++ are running (68). This enables the use of GPUs, massive parallelization and distribution over several computing nodes.

5 Statistics

Never tell me the odds

Han Solo

We do not need much, but I still want them to be defined and explained, so let us talk about the important definitions.

Definition 1 (Confusion Matrix):

Let n be the number of classes, then the confusion matrix is a $n \times n$ matrix. Each column contains the assigned class and each row contains the actual class. Let $0 \geq i \geq n \in \mathbb{N}$, then row i of the confusion matrix contains all elements of class i and how often it was assigned to each class. The sum of row i is the number of elements in class i. Column i corresponds to the number of guesses per class and its sum is the number of total guesses per class i. (69) (70) (71)

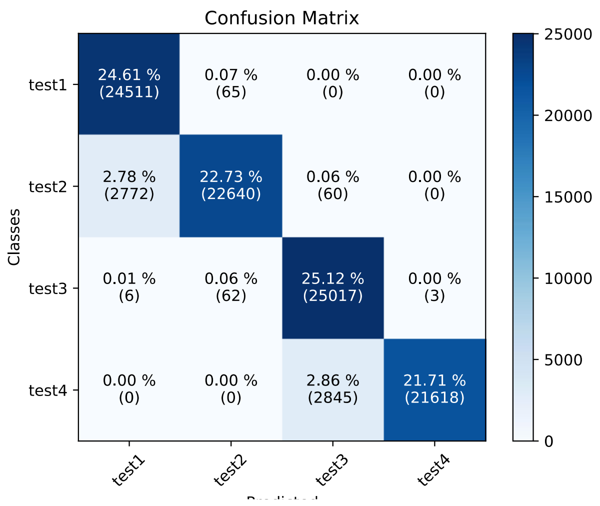


Figure 32: An Example Confusion Matrix based on simple Test Data

Figure 32 shows a depiction of the confusion matrix, with some results from test data generated for this work. The test data are simple nine by nine matrices with either one or two horizontal or vertical lines. This gives us four classes, but sometimes two lines fall together and an item from a class of two lines looks like one from the class of a single line.

Now to understand this matrix, the class *test1* was correctly predicted 24.511 times and 65 times items of this class were predicted to be of class *test2*. Class *test2* was correctly guessed 22.640 times and 2.772 times items of this class were thought to be of class *test1* and 60 of *test3*.

Remark:

With a perfect neural network the confusion matrix would simply be a diagonal matrix with the number of elements per class along the diagonal. Furthermore the elements of the confusion matrix are natural numbers.

Definition 2 (True Positive):

Let *n* be the number of classes and $0 \geq i \geq n \in \mathbb{N}$ and let *M* be a confusion matrix as defined by Definition 1, then *true positive* (TP) for class *i* is given by (72):

$$\mathrm{TP_i} = \mathrm{M_{ii}}$$

Definition 3 (True Negative):

Let *n* be the number of classes and $0 \geq i \geq n \in \mathbb{N}$ and let *M* be a confusion matrix as defined by Definition 1, then *true negative* (TN) for class *i* is given by (72):

$$\mathrm{TN_i} = \sum_{j \neq i}^{n} \sum_{k \neq i}^{n} \mathrm{M_{jk}}$$

Definition 4 (False Positive):

Let *n* be the number of classes and $0 \geq i \geq n \in \mathbb{N}$ and let M be a confusion matrix as defined by Definition 1, then *false positive* (FP) for class *i* is given by (72):

$$\mathrm{FP_i} = \sum_{j \neq i}^{n} \mathrm{M_{ji}}$$

Definition 5 (False Negative):

Let *n* be the number of classes and $0 \geq i \geq n \in \mathbb{N}$, and let M be a confusion matrix as defined by Definition 1, then *false negative* (FN) for class *i* is given by (72):

$$\mathrm{FN_i} = \sum_{j \neq i}^{n} \mathrm{M_{ij}}$$

Remark:

The above definitions are given for a multiclass case. In a binary case, meaning just two classes, the confusion matrix will be 2×2 matrix and the sums above will just be single numbers. Since the confusion matrix consists only of natural numbers, the four definitions from above will all be natural numbers as well. Figure 33 shows how to read the above given definitions from a confusion matrix.

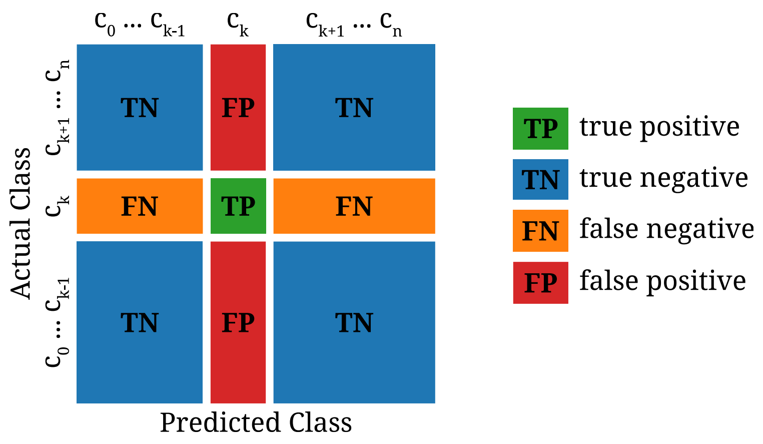


Figure 33: How to read TP, TN, FN & FP from a Confusion Matrix (73)

Definition 6 (Accuracy):

Let TP, TN, FP and FN be a true positive, true negative, false positive and false negative respectively and as defined above, then *accuracy a* is given by (69) (71):

$$a = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}$$

Definition 7 (Precision):

Let TP, TN and FP be a true positive, true negative and false positive respectively and as defined above, then *precision p* is given by (69) (71):

$$p = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

Definition 8 (Recall):

Let TP, TN and FN be a true positive, true negative and false negative respectively and as defined above, then *recall r* is given by (74) (71):

$$\mathrm{r} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

Definition 9 (Weighted F metric):

Let *p* be a precision and *r* be a recall as defined above and β be a number, then the *weighted F metric* f_β is given by:

$$\mathrm{f}_\beta = (1 + \beta^2) \times \frac{\mathrm{p} \times \mathrm{r}}{\beta^2 \mathrm{p} + \mathrm{r}}$$

Remark:

If we set $\beta = 1$ in the weighted F metric we get the F_1 score (74).

Definition 10 (Matthew Correlation Coefficient):

Let TP, TN, FP and FN be a true positive, true negative, false positive and false negative respectively and as defined above, then the *Matthew Correlation Coefficient MCC* is given by (69) (75):

$$\mathrm{MCC} = \frac{\mathrm{TP} \times \mathrm{TN} - \mathrm{FP} \times \mathrm{FN}}{\sqrt{(\mathrm{TP} + \mathrm{FP})(\mathrm{TP} + \mathrm{FN})(\mathrm{TN} + \mathrm{FP})(\mathrm{TN} + \mathrm{FN})}}$$

Remark:

The Matthew Correlation Coefficient as defined above will be a real number between -1 and +1, with -1 meaning that every guess was total wrong, 0 meaning that all guesses were random and +1 meaning, that the network worked perfectly. The -1 case is not the worst situation, one would flip the assumptions to get to +1. Only in case of MCC=0 we would be in a bad position, since it means there is no correlation between input and guess. The advantage of Matthew Correlation Coefficient over other scores is, that it works good with multilabel systems and it suits even unbalanced data sets, meaning data sets with vastly different amounts of data per label (75).

6 Theoretical Background

The miracle of the appropriateness of the language of mathematics for the formulation of the laws of physics is a wonderful gift which we neither understand nor deserve.

Eugene Wigner

6.1 The Standard Model

All models are wrong, but some are useful

George E. P. Box

The SM is the crown achievement of modern-day physics, combing Maxwell's[16] theory of electrodynamics with Einstein's Special Relativity. The former combined electronic forces with magnetic forces and the latter is an extension of Newtonian[17] Mechanics, postulating a four dimensional space, combining space and time into spacetime and fixing the speed of light to an absolute.

In Figure 34 we see a depiction of the standard model with the three generations of the quark and lepton families as outer circle, the gauge bosons[18] in the middle, governing the fundamental forces and at the center is the elusive Higgs.

The basis for the SM is Quantum Field Theory (QFT), where particles are described as excitations in field equations. In QFT forces are carried by exchange particles, which are virtual particles or fluctuations in the field equations. The reach or life times of these exchange particles are determined by Heisenberg's[19] uncertainty principle (76). The field equations can be calculated by using Lagragian equations. The gauge group of the SM Lagragian is $SU(3)_C \times SU(2)_L \times U(1)_Y$, the generators for *SU(2)* and *SU(3)* the three Pauli[20]- and Gell-Mann[21]-matrices respectively (77).

[16] James Clerk Maxwell (13 June 1831 – 5 November 1879)

[17] Sir Isaac Newton (25 December 1642 – 20 March 1727)

[18] Satyendra Nath Bose (1 January 1894 – 4 February 1974)

[19] Werner Karl Heisenberg (5 December 1901 – 1 February 1976)
Allegedly Bohr made Heisenberg cry (127).

[20] Wolfgang Ernst Pauli (25 April 1900 – 15 December 1958)

[21] Murray Gell-Mann (15 September 1929 – 24 May 2019)

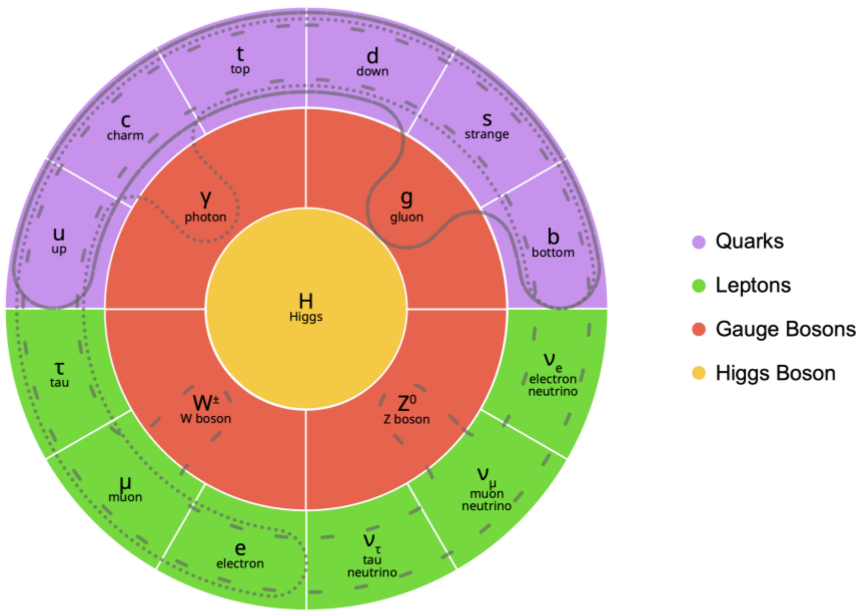


Figure 34: Standard Model (78)

For further details, one should check Table 3 for information on the forces and Table 4 for details about each particle of the SM.

Table 3: The forces of the SM (and gravity) with their exchange particles (79) (80) (81)

	Strong	*Weak*	*Electromagnetic*	*Gravitation*
Current Theory	Quantum Chroma Dynamics (QCD)	Electro Weak Theory	Quantum Electro Dynamics (QED)	General Relativity
Charge	Color	Weak Charge	Electric Charge	
Exchange Particle	8 Gluons (g)	$W^{\pm}$, Z^0	Photons (γ)	Graviton (hypothetical)
Mass / GeV	0	80, 90	0	0
Long-distance behavior	$\sim r$	$\frac{1}{r}e^{-m_{W,Z}r}$	$\frac{1}{r}$	$\frac{1}{r}$
Range / m	2×10^{-15}	2×10^{-18}	∞	∞
Coupling Parameter	$\alpha_{strong} = \frac{1}{2} \cdots \frac{1}{10}$	$\alpha_{weak} = \frac{1}{30}$	$\alpha_{em} = \frac{1}{137}$	$\alpha_g = \frac{1}{10^{45}} \cdots \frac{1}{10^{38}}$

Relative Strength	1	10^{-15}	10^{-2}	10^{-41}
J^P	1^-	1^-	1	2

Table 4: The three generations of leptons and quarks (79)

Fermion	*1st Gen.*	*2nd Gen.*	*3rd Gen.*	*Charge*	*Color*	*Lefthanded Isospin*	*Righthanded Isospin*	*Spin*
Leptons	ν_e	ν_μ	ν_τ	0	--	½	--	½
	e	μ	τ	-1			0	
Quarks	u	c	t	+⅔	r, g, b	½	0	½
	d	s	b	-⅓			0	

But let us have a look at all parts of the SM Lagragian (82):

$$\mathcal{L}_{SM} = \underbrace{\mathcal{L}_{Yang-Mills}}_{\mathcal{L}_{QCD}+\mathcal{L}_{IW}+\mathcal{L}_{Y}} + \mathcal{L}_{Weyl-Dirac} + \mathcal{L}_{Yukawa} + \mathcal{L}_{Higgs}$$

The Yang[22]-Mills[23] sector compromises of QCD, weak isospin field strength and the hypercharge, sometimes $\mathcal{L}_{QCD}$ and $\mathcal{L}_{Y}$ and combined to a gauge Lagrangian and the Weyl[24]-Dirac[25] sector is sometimes called fermion sector (77). Other times the Yang-Mills and Weyle-Dirac sector are combined and expressed as (83):

$$\mathcal{L}_{kinetic} = i\bar{\psi}(D^\mu\gamma_\mu)\psi$$

which is eerily similar to the Dirac equation without mass term. The other terms are then hidden inside the covariant derivative D^μ (83). I will not write the full Lagragian, because it is not necessary for this work and a bit excessive and instead I will focus on the Yukawa sector of the SM. This is the sector which gives rise to the CKM matrix, which governs CP violation in the SM. The gauge groups and the sectors of SM are

[22] Chen-Ning Yang (1 October 1922)

[23] Robert Laurence Mills (15 April 1927 – 27 October 1999)

[24] Hermann Klaus Hugo Weyl (9 November 1885 – 8 December 1955)

[25] Wolfgang Pauli: »There is no God and Dirac is His prophet«

tabulated in Table 5 (77). Sometimes there is an additional ghost Lagrangian to compensate for too many degrees of freedom.

Table 5: The gauge groups of the SM (77)

Group	*Lagrangian fields*	*After electroweak symmetry breaking*
SU(3)	gluons	gluons
SU(2)	$W_\mu^{1,2,3}$	$W_\mu^\pm, Z_\mu$
U(1)	B_μ	A_μ

The Yukawa[26] Lagragian density belongs to $SU(2)_L \times U(1)_Y$ gauge groups (77) and its symmetry is spontaneously broken by the presence of the Higgs mechanism and this is where the mass of W and Z bonsons comes from. This Largangian is the source of quark mixing (77) (82) (83). The Yukawa Largangian reads as follows (77) (84) (85):

$$\mathcal{L}_{Yukawa} = -\left\{\begin{pmatrix}\bar{e}\\ \bar{\mu}\\ \bar{\tau}\end{pmatrix}^T \cdot \begin{pmatrix}m_e & 0 & 0\\ 0 & m_\mu & 0\\ 0 & 0 & m_\tau\end{pmatrix} \cdot \begin{pmatrix}e\\ \mu\\ \tau\end{pmatrix} + \begin{pmatrix}\bar{u}\\ \bar{c}\\ \bar{t}\end{pmatrix}^T \cdot \begin{pmatrix}m_u & 0 & 0\\ 0 & m_c & 0\\ 0 & 0 & m_t\end{pmatrix} \cdot \begin{pmatrix}u\\ c\\ t\end{pmatrix} + \begin{pmatrix}\bar{d}\\ \bar{s}\\ \bar{b}\end{pmatrix}^T \cdot \begin{pmatrix}m_d & 0 & 0\\ 0 & m_s & 0\\ 0 & 0 & m_t\end{pmatrix} \cdot \begin{pmatrix}d\\ s\\ b\end{pmatrix}\right\} \cdot \left(1 + \frac{H}{v}\right)$$

H is a scalar Higgs field and *v* is the vacuum expectation value (85) and *dsb* are linear combinations of electroweak eigenstates (77):

$$\underbrace{\begin{pmatrix}d\\ s\\ b\end{pmatrix}}_{\substack{mass\\ eigenstates}} = V_{CKM} \cdot \underbrace{\begin{pmatrix}d'\\ s'\\ b'\end{pmatrix}}_{\substack{electroweak\\ eigenstates}}$$

This is where the famous CKM matrix comes in, but more on it in a later section. It is a matrix product of two unitary matrices, which allow generation mixing, and is thus also unitary. One can show, that the meson octet and baryon decuplet/octet are generated through SU(N) gauge symmetries and the use of a quark Lagrangian (77):

[26] Hideki Yukawa (23 January Meiji 40 – 8 September Showa 56)

$$\mathcal{L}_{quark} = \sum_{f\in\{u,d,s\}} \bar{\psi}_f\left(i\gamma^{\mu}D_{\mu} - m_f\right)$$

The ansatz here is to use three-by-three ladder matrices (77).

6.1.1 What is Gauge Theory

The basic idea of gauge theories are transformations of fields and potentials that leave the underlying equations of motion invariant. In other words, we transform a mathematical construct, that cannot be measured, that does not change the mathematical object describing what can be measured. Given a potential ϕ we calculate the force by taking the derivative:

$$E = -\nabla\phi - \partial_t A$$

Now we can add any terms that vanish when taking the gradient of the potential. For example:

$$\phi' = \phi + \partial_t\Lambda \qquad A' = A - \nabla\Lambda$$

These transformations form a mathematical group that is called gauge group in physics. This basic idea was first introduced by Clark Maxwell in the theory of electrodynamics. The above example can be easily extended to a four-dimensional potential:

$$A_\mu \rightarrow A'_\mu = A_\mu + \partial_\mu f$$

This kind of transformation is central to the quantum field theories and the theory of relativity. The same principle can be applied to a Lagragian. Adding additional terms to the Lagragian that do not change the resulting equations of motion. (86) (87) (88)

6.1.2 Left- and Right-Handedness

There is an intrinsic difference between particles whose spin is aligned colinear and antilinear to their momentum. In physics this called helicity or left- and right-handedness, this is shown in Figure 35. This is important to understand, because the four fundamental forces act differently on left- and right-handed particles. The weak force for instance acts only on left-handed particles and all neutrinos are left-handed. (89) (79)

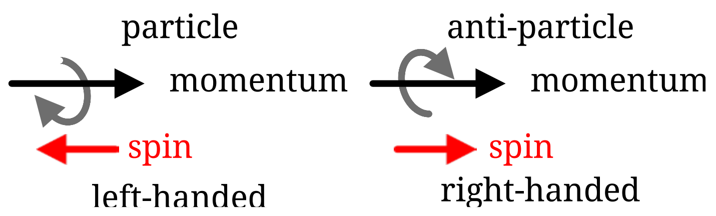


Figure 35: Helicity of particles and anti-particles (90)

Helicity is expressed by (83):

$$h = \frac{1}{2}\vec{\sigma} \cdot \hat{p}$$

CP violation manifests itself by not conserving helicity (83). Left-handed particles have a helicity of $h = -1$ and right-handed $h = +1$ (90).

6.1.3 CPT Theorem

This section is mostly based on the lecture notes for N Tunings lecture on CP violation (83) and on the article *CP violation in the B system* by T. Gershon and V.V. Gligorov (91). If I used other sources to supplement the information presented here, I will mention it explicitly.

CPT stands for charge conjugate, parity and time and consists of three operators. The first in the list flips all quantum numbers, hence it can transform between particle and anti-particle. The second mirrors all spatial coordinates and the last is the time reversal operator, which turns time around.

In order to get a better understanding, let us look at the Dirac equation for QED:

$$\left(i\gamma^\mu \partial_\mu - \gamma^\mu e A_\mu - m\right)\psi(\vec{x}, t) = 0$$

Here γ^μ are the Dirac matrices, A_μ is the four potential and ψ is a four-spinor. Now we will test how each operator acts on our spinor defined by the equation above. First the parity operator:

$$P\colon \psi(\vec{x}, t) \to \gamma^0 \psi(-\vec{x}, t) = P\psi(-\vec{x}, t)$$

then we check charge conjugate:

$$C\colon \psi(\vec{x}, t) \to \; i\gamma^2 \psi^*(\vec{x}, t) = i\gamma^2\gamma^0 \bar{\psi}^T(\vec{x}, t) = C\bar{\psi}^T(\vec{x}, t)$$

and time reversal:

$$T\colon \psi(\vec{x}, t) \to i\gamma^1\gamma^3 \psi^*(\vec{x}, -t) = T\psi^*(\vec{x}, -t)$$

There are two more transformation we have to look at. First the CP operation:

$$CP\psi(\vec{x}, t) = ie^{i\Phi}\gamma^2\gamma^0\psi^*(\vec{x}, t)$$

and second CPT:

$$CPT\psi(\vec{x},t) = e^{i\phi}\gamma^5\psi(-\vec{x},-t)$$

The results additionally with a scalar field and an axial vector field is shown in Table 6.

Table 6: C and P operations on different field types (83)

Field		*P*	*C*
Scalar Field	$\phi(\vec{x},t)$	$\phi(-\vec{x},t)$	$\phi^\dagger(\vec{x},t)$
Dirac Spinor	$\psi(\vec{x},t)$	$\gamma^0\psi(-\vec{x},t)$	$i\gamma^2\gamma^0\bar{\psi}^T(\vec{x},t)$
	$\tilde{\psi}(\vec{x},t)$	$\bar{\psi}(-\vec{x},t)\gamma^0$	$-\psi^T(\vec{x},t)C^{-1}$
Axial Vector Field	$A_\mu(\vec{x},t)$	$-A^\mu(-\vec{x},t)$	$A_\mu^\dagger(\vec{x},t)$

Due to the requirement of having Lorentz invariance, we want spinors to be Lorentz scalars and the way to achieve this is to use bilinear forms. Think of scalar products, where a dual vector maps a vector to a number. For this we will look at bilinear forms in Table 7.

Table 7: C, P & T transformations of bilinear forms (83)

	Bilinear	*P*	*C*	*T*	*CP*	*CPT*
scalar	$\bar{\psi}_1\psi_2$	$\bar{\psi}_1\psi_2$	$\bar{\psi}_2\psi_1$	$\bar{\psi}_1\psi_2$	$\bar{\psi}_2\psi_1$	$\bar{\psi}_2\psi_1$
pseudo scalar	$\bar{\psi}_1\gamma_5\psi_2$	$-\bar{\psi}_1\gamma_5\psi_2$	$\bar{\psi}_2\gamma_5\psi_1$	$-\bar{\psi}_1\gamma_5\psi_2$	$-\bar{\psi}_2\gamma_5\psi_1$	$\bar{\psi}_2\gamma_5\psi_1$
vector	$\bar{\psi}_1\gamma_\mu\psi_2$	$\bar{\psi}_1\gamma^\mu\psi_2$	$-\bar{\psi}_2\gamma_\mu\psi_1$	$\bar{\psi}_1\gamma^\mu\psi_2$	$-\bar{\psi}_2\gamma^\mu\psi_1$	$-\bar{\psi}_2\gamma_\mu\psi_1$
axial vector	$\bar{\psi}_1\gamma_\mu\gamma_5\psi_2$	$-\bar{\psi}_1\gamma^\mu\gamma_5\psi_2$	$\bar{\psi}_2\gamma_\mu\gamma_5\psi_1$	$\bar{\psi}_1\gamma^\mu\gamma_5\psi_2$	$-\bar{\psi}_2\gamma^\mu\gamma_5\psi_1$	$-\bar{\psi}_2\gamma_\mu\gamma_5\psi_1$
tensor	$\bar{\psi}_1\sigma_{\mu\nu}\psi_2$	$\bar{\psi}_1\sigma^{\mu\nu}\psi_2$	$-\bar{\psi}_2\sigma_{\mu\nu}\psi_1$	$-\bar{\psi}_1\sigma^{\mu\nu}\psi_2$	$\bar{\psi}_2\sigma^{\mu\nu}\psi_1$	$\bar{\psi}_2\sigma_{\mu\nu}\psi_1$

6.1.4 Weak Force

The weak force only acts on left-handed particles (92). It is only observable when the electromagnetic and strong forces are suppressed. But the weirdness does not stop here (77):

- from Dirac's equations we conclude, that C should hold

- P should still be a sound symmetry
- from Newtonian's mechanics we know, that T is reversable
- combined violations in CPT would also break Lorentz invariance

The W boson was discovered 1983 (93). The weak force is hard to observe, because it has only a tiny cross section (79) (93). The weakness of the Weak Force is is due to W and Z bosons having such a high mass (79), which comes from the breaking of symmetry by the presence of a Higgs field. The weak force is only observable when quark flavors are changed or when neutrinos are involved in the process (93). To gain some sense of the weakness of the weak force, we can look at the two decays and their half-life (93):

$$\Sigma^{+}(1189) \rightarrow p\pi^{0} \qquad \tau \approx 10^{-10}s$$
$$\Sigma^{0}(1192) \rightarrow \Lambda\gamma \qquad \tau \approx 10^{-19}s$$

We can compare them to get an understanding of their relative strength (93):

$$\frac{g}{e} \approx \sqrt{\frac{10^{-19}}{10^{-10}}} \approx 10^{-5}$$

This means, that the weak force is several orders of magnitude weaker than the electromagnetic force. From experiments we know the reason is, that the gauge bosons of weak forces possess a large mass (93). This fact was already mentioned several times throughout this work.

6.2 CP Violation

6.2.1 Some History

CP violation was first observed 1964 in decays of K_{short} and K_{long} by James Cronin[27] and Val Fitsch[28] (94). The former should always decay into two pions, whereas the latter should always decay into three poins (94). But it was observed that K_{long} decayed into

[27] James Watson Cronin (29 September 1929 – 25 Agust 2016)

[28] Val Logsdon Fitch (10 March 1923 – 5 February 2015)

two poins, which happened at a rate of 0.1% (95). CP violation happens due to the weak forces in the quark-mixing matrix (11), which we saw in the Yukawa Lagragian. The parity of the Kaon system is not well defined, if we look at the parity of these decays (93):

$$K^+ \to \pi^+\pi^0 \quad P(\pi^+\pi^0) = (-1)(-1) = +1$$
$$K^+ \to \pi^+\pi^+\pi^- \quad P(\pi^+\pi^+\pi^-) = (-1)^3 = -1$$

From this we can conclude, that the weak force can change parity, which is not true for strong and electromagnetic forces (93). CP violation is accounted for by the CKM matrix in the SM. This matrix comes from the Yukawa sector of the SM Lagragian as was discussed in *The Standard Model*. CP violation is thus quantum mechanical interference.

6.2.2 A general approach to CP violation

The source of CP violation can easily be shown by this form of Yukawa Lagrangian (82) (83) (89):

$$\mathcal{L}_{Yukawa} = Y\psi_L\chi_L H - Y^*\psi_L^\dagger\sigma_2\chi_L^* H^*$$

with Y beging the Yukawa coupling matrices, ψ und χ arbitrary spinors, H is again a Higgs doublet and σ_2 is the second Pauli matrix. Now if we apply a CP transformation, the source of CP violation should reveal itself (82) (83):

$$CP\mathcal{L}_{Yukawa} = -Y\psi_L^\dagger\sigma_2\chi_L^* H^{CP} + Y^*\psi_L^T\sigma_2\chi_L H^{*CP}$$

since we know what the Higgs spinor looks like and that is it simply a complex conjugate we conclude, that CP is violated if and only if $Y \neq Y^*$. We see, that CP violation arises from the Yukawa coupling matrice, if they are not purely real, but contain any kind of complex numbers. (82) (83)

6.2.3 Three classes of CP violation

This section is based on the lecture by N. Tuning (83) and the report by Greshon and Gligorov (91). We start with a generic, neutral Meson P, which decays into a final state f. We can have two eigenstates:

$$P_1 = pP^0 - q\bar{P}^0$$
$$P_2 = pP^0 + q\bar{P}^0$$

For the decay we will employ the the Wigner[29]-Weisskopf[30] Hamiltonian[31] (77):

$$i\frac{\partial}{\partial t}\begin{pmatrix}p\\q\end{pmatrix} = \mathcal{H}\begin{pmatrix}p\\q\end{pmatrix} = \begin{pmatrix} m_{11} - \frac{i}{2}\Gamma_{11} & m_{12} - \frac{i}{2}\Gamma_{12} \\ m_{21} - \frac{i}{2}\Gamma_{21} & m_{22} - \frac{i}{2}\Gamma_{22} \end{pmatrix}\begin{pmatrix}p\\q\end{pmatrix}$$

Since $\mathcal{H}$ is hermitian, this implies $m_{21} = m_{12}^*$ and $\Gamma_{21} = \Gamma_{12}^*$.

CPT conservation implies $m_{11} = m_{22} = m$ and $\Gamma_{11} = \Gamma_{22}^* = \Gamma$:

$$i\frac{\partial}{\partial t}\begin{pmatrix}p\\q\end{pmatrix} = \mathcal{H}\begin{pmatrix}p\\q\end{pmatrix} = \begin{pmatrix} m - \frac{i}{2}\Gamma & m_{12} - \frac{i}{2}\Gamma_{12} \\ m_{12}^* + \frac{i}{2}\Gamma_{12}^* & m - \frac{i}{2}\Gamma \end{pmatrix}\begin{pmatrix}p\\q\end{pmatrix}$$

Now we know that there is a relative phase between m_{12} and Γ_{12} and if time reversal holds, we can use this relative phase to make m_{12} and Γ_{12} real. We can calculate the eigenvalues:

$$\lambda_\pm = \left(m - \frac{i}{2}\Gamma\right) \pm \sqrt{\left(m_{12} - \frac{i}{2}\Gamma_{12}\right)\left(m_{12}^* + \frac{i}{2}\Gamma_{12}^*\right)}$$

and plugging them into:

$$\begin{pmatrix} m - \frac{i}{2}\Gamma & m_{12} - \frac{i}{2}\Gamma_{12} \\ m_{12}^* + \frac{i}{2}\Gamma_{12}^* & m - \frac{i}{2}\Gamma \end{pmatrix}\begin{pmatrix}p\\q\end{pmatrix} = \lambda_\pm \begin{pmatrix}p\\q\end{pmatrix}$$

which yields, after we make the choice that P_2 is the heavier eigenstate:

$$\frac{q}{p} = \sqrt{\frac{m_{12}^* - \frac{i}{2}\Gamma_{12}^*}{m_{12} - \frac{i}{2}\Gamma_{12}}}$$

In the case of decays into final states f und $\bar{f}$ we will have four amplitutes:

$$A(f) = \langle f|T|P^0\rangle$$
$$A(\bar{f}) = \langle \bar{f}|T|P^0\rangle$$
$$\bar{A}(f) = \langle f|T|\bar{P}^0\rangle$$
$$\bar{A}(\bar{f}) = \langle \bar{f}|T|\bar{P}^0\rangle$$

[29] Eugene Paul Wigner (17 November 1902 – 1 January 1995)

[30] Victor Frederick Weisskopf (19 September 1908 – 22 April 2002)

[31] Sir William Rowan Hamilton (3 August 1805 – 2 September 1865)

In these cases, T is the transition operator and not the time reversal operator. We also have the four decay rates, $\Gamma_{P^0\to f}$, $\Gamma_{P^0\to \bar{f}}$, $\Gamma_{\bar{P}^0\to f}$ and $\Gamma_{\bar{P}^0\to \bar{f}}$. With these prerequisites at hand we can understand the three kinds of CP violations.

CP violation in decay

It is also called tree-dominated CP violation, it happens when

$$\Gamma_{P^0\to f} \neq \Gamma_{\bar{P}^0\to \bar{f}}$$

which is the case if:

$$\left|\frac{A(f)}{\bar{A}(\bar{f})}\right| \neq 1$$

This happens for example in semileptonic decays, such as $B^0 \to D^-\mu^+\nu\_\mu$.

CP violation in mixing:

Here CP violation is done through loop-diagrams, like Penguin and box diagrams, it happens in radiative, semileptonic decays $b \to (s,d)(\gamma, \ell^+\ell^-, \nu\bar{\nu})$ and hadronic decays such as $b \to s\bar{s}s, d\bar{d}s, s\bar{s}d$. This happens when

$$Prob(P^0 \to \bar{P}^0) \neq Prob(\bar{P}^0 \to P^0)$$

which occurs when:

$$\left|\frac{q}{p}\right| \neq 1$$

This form of CP violation only occurs occasionally.

CP violation in interference

This happens during oscillation between particle and anti-particle or with $b \to u\bar{u}s, u\bar{u}d$ transitions. The defining relation is:

$$\Gamma\left(P^0_{(\rightsquigarrow \bar{P}^0)} \to f\right) \neq \Gamma(\bar{P}^0_{(\rightsquigarrow P^0)} \to f)$$

This happens, only when the two mesons decay into a common eigenstate. This means $f = \bar{f}$, which then implies:

$$Im\left(\frac{q\bar{A}_f}{pA_f}\right) \neq 0$$

One can think of a double slit experiment, where there are two possible paths to take:

$$\begin{tikzcd} & \bar{P}^0 \arrow[dr] & \\ P^0 \arrow[ur] \arrow[rr] & & f \end{tikzcd}$$

Either P^0 can directly decay into f or first into the anti-P and then into f.

Observed CP violation in the following cases (91):

Table 8: Observed CP violation in B systems

violation in	*K^0*	*K^+*	*Λ*	*D^0*	*D^+*	*D^+_s*	*Λ^+_c*	*B^0*	*B^+*	*B^0_s*	*Λ^0_b*
mixing	✓	-	-	✗	-	-	-	✗	-	✗	-
decay	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗
mixing/decay	✓	-	-	✗	-	-	-	✓	-	✗	-

6.3 CKM and Triangles

6.3.1 CKM Matrix

The Cabibbo[32]–Kobayashi[33]–Maskawa[34] (CKM) matrix is given (96) in the case of three generations by:

$$\underbrace{\begin{pmatrix} d \\ s \\ b \end{pmatrix}}_{\substack{mass \\ eigenstates}} = \begin{pmatrix} V_{ud} & V_{us} & V_{ub} \\ V_{cd} & V_{cs} & V_{cb} \\ V_{td} & V_{ts} & V_{tb} \end{pmatrix} \underbrace{\begin{pmatrix} d' \\ s' \\ b' \end{pmatrix}}_{\substack{electro-weak \\ eigenstates}}$$

The initial idea came from Cabibbo, but he came up with a two-by-two matrix and only connected two of the three generations. Kobayashi and Maskawa extended the idea to the three-by-three matrix we know today (92). It contains nine masses, three angles and a complex phase. This very phase is responsible for CP violation (83), which we saw already in *A general approach to CP violation*. Both, Kobayashi and Maskawa, won a Nobel prize for their contribution to physics (1), Cabbibo was left out. (83)

We now want to figure out some characteristics of this matrix. The CKM matrix is unitary, which means (93) (91):

[32] Nicola Cabibbo (10 April 1935 – 16 August 2010)

[33] Makoto Kobayashi (7 April Showa 19 – 23 July Reiwa 3)

[34] Toshihide Maskawa (7 February Showa 15)

$$|V_{ud}|^2 + |V_{us}|^2 + |V_{ub}|^2 = 1$$
$$|V_{cd}|^2 + |V_{cs}|^2 + |V_{cb}|^2 = 1$$
$$|V_{td}|^2 + |V_{ts}|^2 + |V_{tb}|^2 = 1$$
$$|V_{ud}|^2 + |V_{cd}|^2 + |V_{td}|^2 = 1$$
$$|V_{us}|^2 + |V_{cs}|^2 + |V_{ts}|^2 = 1$$
$$|V_{ub}|^2 + |V_{cb}|^2 + |V_{tb}|^2 = 1$$

It is handy to reparametrize this matrix as a rotation matrix using cosine and sine with three special angles (θ_1, θ_2, θ_3) and a CP violating phase factor (δ). It is customary to abbreviate some parts $c_k \coloneqq \cos(\theta_k)$ and $s_k \coloneqq \sin(\theta_k)$, then this matrix becomes (77):

$$\begin{pmatrix} c_1 & -s_1 c_3 & -s_1 s_3 \\ s_1 c_2 & c_1 c_2 c_3 - s_2 s_3 e^{i\delta} & c_1 c_2 s_3 + s_2 c_3 e^{i\delta} \\ s_1 s_2 & c_1 s_2 c_3 + c_2 s_3 e^{i\delta} & c_1 s_2 s_3 - c_2 c_3 e^{i\delta} \end{pmatrix}$$

This can be rewritten in terms of Euler[35] angels θ_{12}, θ_{23}, θ_{13} and a CP violating phase δ_{13} (97) (83):

$$\begin{pmatrix} c_{12} c_{13} & s_{12} c_{13} & s_{13} e^{-i\delta_{13}} \\ -s_{13} c_{23} - c_{12} s_{23} s_{13} e^{i\delta_{13}} & c_{12} c_{23} - s_{13} s_{23} s_{13} e^{i\delta_{13}} & s_{23} c_{13} \\ s_{12} s_{23} - c_{12} c_{23} s_{13} e^{i\delta_{13}} & -c_{12} s_{23} - s_{12} c_{23} s_{13} e^{i\delta_{13}} & c_{23} s_{13} \end{pmatrix}$$

If we now substitute $\lambda = s_{12}, A\lambda^2 = s_{23}$ and $A\lambda^3(\rho - i\eta) = s_{13} e^{-i\delta}$ we get the Wolfenstein[36] parametrization of the CKM matrix, which looks like the following (97) (98) (83):

$$\begin{pmatrix} 1 - \frac{1}{2}\lambda^2 & \lambda & A\lambda^3(\rho - i\eta) \\ -\lambda & 1 - \frac{1}{2}\lambda^2 & A\lambda^2 \\ A\lambda^3(1 - \rho - i\eta) & -A\lambda^2 & 1 \end{pmatrix} + \mathcal{O}(\lambda^4)$$

[35] Leonhard Euler (15 April 1707 – 18 September 1783)

[36] Lincoln Wolfenstein (10 February 1923 – 27 March 2015)

This form is practical, since most data to date, with exception of sin(2$\boldsymbol{\beta}$), come from CP conserving measurements (99). A b-quark decays most likely into a c-quark i.e., charmed decay, because of $|V_{cb}|^2 \gg |V_{ub}|^2$ (100).

The values for each element of the CKM matrix are (83) (96):

$$V_{CKM} = \begin{pmatrix} |V_{ud}| & |V_{us}| & |V_{ub}| \\ |V_{cd}| & |V_{cs}| & |V_{cb}| \\ |V_{td}| & |V_{ts}| & |V_{tb}| \end{pmatrix}$$

$$= \begin{pmatrix} 0.97446 & 0.22452 & 0.00365 \\ 0.22438 & 0.97359 & 0.04214 \\ 0.00896 & 0.04133 & 0.99911 \end{pmatrix} \pm \begin{pmatrix} 0.00010 & 0.00044 & 0.00012 \\ 0.00044 & 0.00011 & 0.00076 \\ 0.00024 & 0.00974 & 0.00003 \end{pmatrix}$$

A special role is played by the elements V_{us}, V_{cs} and V_{ub}, they are involved in K and B decays, through which CP violation is currently studied (96). The element V_{ub} contains a complex phase and is therefore responsible for CP violation (93). It is this complex phase which was hinted at in the section *A general approach to CP violation*.

6.3.2 The Unitary Triangle

There are six unitary triangles, that can be derivate from the CKM matrix (97) and they can be written as following (83):

$$\underbrace{V_{ud}V_{us}^*}_{of\ order\ \lambda} + \underbrace{V_{cd}V_{cs}^*}_{of\ order\ \lambda} + \underbrace{V_{td}V_{ts}^*}_{of\ order\ \lambda^5} = 0$$

$$\underbrace{V_{ud}V_{ub}^*}_{of\ order\ \lambda^3} + \underbrace{V_{cd}V_{cb}^*}_{of\ order\ \lambda^3} + \underbrace{V_{td}V_{tb}^*}_{of\ order\ \lambda^3} = 0$$

$$\underbrace{V_{us}V_{ub}^*}_{of\ order\ \lambda^4} + \underbrace{V_{cs}V_{cb}^*}_{of\ order\ \lambda^2} + \underbrace{V_{ts}V_{tb}^*}_{of\ order\ \lambda^2} = 0$$

$$\underbrace{V_{us}V_{cs}^*}_{of\ order\ \lambda} + \underbrace{V_{ud}V_{cd}^*}_{of\ order\ \lambda} + \underbrace{V_{ub}V_{cb}^*}_{of\ order\ \lambda} = 0$$

$$\underbrace{V_{tb}V_{ub}^*}_{of\ order\ \lambda^3} + \underbrace{V_{td}V_{ud}^*}_{of\ order\ \lambda^3} + \underbrace{V_{ts}V_{us}^*}_{of\ order\ \lambda^3} = 0$$

$$\underbrace{V_{cd}V_{td}^*}_{of\ order\ \lambda^4} + \underbrace{V_{cs}V_{ts}^*}_{of\ order\ \lambda^2} + \underbrace{V_{cb}V_{tb}^*}_{of\ order\ \lambda^2} = 0$$

The canonical[37] triangle is the second from the top, which is mainly investigated by the Belle group (6). When people speak of *the* unitary triangle, they are talking about this one. It is customary to normalize the baseline to one, a depiction of it can be seen in Figure 36.

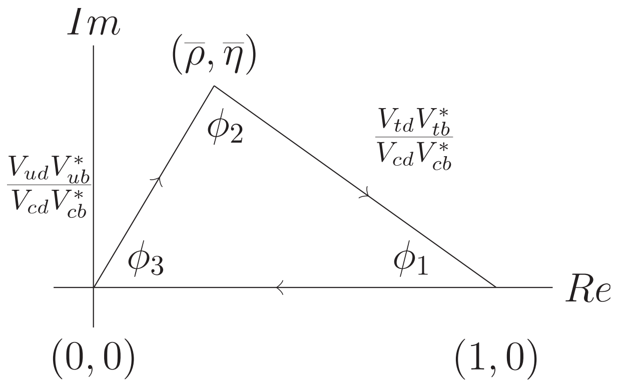

Im
$(\bar{\rho},\bar{\eta})$
$\frac{V_{td}V_{tb}^*}{V_{cd}V_{cb}^*}$
ϕ_2
$\frac{V_{ud}V_{ub}^*}{V_{cd}V_{cb}^*}$
ϕ_3
ϕ_1
Re
$(0,0)$
$(1,0)$

Figure 36: Unitary triangle (18)

The relation to the CKM matrix and the Wolfenstein parameters is given by these equations:

$$R_u = \left|\frac{V_{ud}V_{ub}^*}{V_{cd}V_{cb}^*}\right| = \sqrt{\rho^2 + \eta^2}$$
$$R_t = \left|\frac{V_{td}V_{tb}^*}{V_{cd}V_{cb}^*}\right| = \sqrt{(1-\rho)^2 + \eta^2}$$

and for the angles we have this easy relation:

$$\alpha = \theta_2 \quad \beta = \theta_1 \quad \gamma = \theta_3$$

At Belle II we are measuring $\sin 2\theta_1 = \sin 2\beta$, one of the inner angles of the unitary triangle, and the Wolfenstein parameter λ (11). The angle β is a measure of CP violation and if $\eta = 0$ then there is no CP violation (93).

We can measure the angles:

$$\sin 2\alpha = -Im\left(\frac{V_{td}^*V_{ub}^*}{V_{td}V_{ub}}\right)$$

[37] From Greek: κανών [kanon], meaning “straight measuring rod, ruler”.

an example is $B_d^0\bar{B}_d^0 \to \pi^+\pi^-$, but this is CKM suppressed. The other angle, which is also CKM suppressed, can be measured by:

$$\sin 2\gamma = Im\left(\frac{V_{ub}^*}{V_{ub}}\right)$$

the example here is $B_s^0\bar{B}_s^0 \to \rho K_{short}$. The third angle is CKM allowed:

$$\sin 2\beta = Im\left(\frac{V_{td}^*}{V_{td}}\right)$$

an example decay is $B_d^0\bar{B}_d^0 \to \psi K_{short}$. (101)

Alternatively, they can be measured with (6):

$$\theta_1 = \pi - \arg\left(\frac{-V_{td}V_{tb}^*}{-V_{cd}V_{cb}^*}\right) = \beta$$

$$\theta_2 = \arg\left(\frac{V_{td}V_{tb}^*}{-V_{ud}V_{ub}^*}\right) = \gamma$$

$$\theta_3 = \arg\left(\frac{V_{ud}V_{ub}^*}{-V_{cd}V_{cb}^*}\right) = \alpha$$

The angles α and β are harder to measure because of interference from so-called penguin diagrams (95). We can measure these angles by the following decays (93):

$$\alpha: B^0 \to \pi\pi, \rho\pi$$

$$\beta: B^0 \to J/\psi K_s$$

$$\gamma: B^0 \to DK$$

It is important to note here, that it is not sufficient to measure two angles and then deduce the third one, measuring all three and seeing that they add up to 180 degree is an essential test of the SM.

7 Analysis

Do. Or do not. There is no try.

Yoda

7.1 Methodology

Here I want to explain the method I used to find a suitable neural network for the task at hand. I was not looking for the best neural network, because it is not feasible to run this supposedly best network. By best network I mean the network, that will have a Matthew Correlation Coefficient (MCC) of plus one in every category. Let alone of finding the or one of the configurations which allow this result, this kind of network will take enormous amounts of time to be trained. This leaves us with a suitable network, which is optimal under the conditions, that one can find its parameters and also train it in a reasonable time.

For a linear problem no hidden layer is needed and a simple input and output layer will suffice (28) (47). According to (47) one or two hidden layers should be enough to solve any problem in which neural networks can be applied. The size of each hidden layer should be between the number of inputs and outputs to prevent over- or underfitting (47). It has been empirically shown that deeper networks perform better and are better at generalizing than wide networks; the issue that comes with deeper networks is, that optimizing them will be harder (28). There is still no method to find the optimal architecture of a neural network (102).

The Loss Function and the last Activation Function are dictated by the objective. In my case it is a multiclass, single-label classification and thus requires *crossentropy* as a Loss Function and on the last layer should be a *softmax* Activation Function (27). As for the hidden layer, it is recommended to use *tanh* (32). Picking a not appropriate loss function for the problem will lead to an alignment problem.

With this knowledge, I could start building the network. It is generally advised to experiment around with the architecture, it is only important to pick one metric which will be the measure of success (27). Then one has to create a baseline, with a very simple network, against which all changes are measured (27).

All these tests ran on a smaller subset of the data of about 35%. I did some preliminary tests on how much the amount of data affects the results and I ended up with 35%, since this gave me a good balance of size and speed, which again allowed me to iterate faster and test more setups. I only trained on Slow Pions and Beam Background at first, since the goal of this work is to find Slow Poins against a larger background of other events. Later I added the other data sets as standalone sets and as additional background.

7.2 Process

7.2.1 Finding an Optimizer

I started with finding an optimizer. For now, we will look at training and validation loss curves, where we have on the x-axis the epoch and on the y-axis the error value as calculated by the loss function. The first objective is to get smooth, hyperbolically falling curves. When finding an optimizer, I used the same network architecture of three layers with a width of 81, with ReLU as an activation function and a learning rate of $\mu = 0.001$, all tests were done with a batch size of 64 and ran for 100 epochs. For SGD I used a momentum of $\gamma = 0.15$, Adam ran on the recommended parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$ and an $\epsilon = 1 \times 10^{-8}$.

In order to have a better understanding, while making the loss curves quantitively comparable to other runs, I will use two numbers indicating the decay speed and amount of oscillation exhibited by the curves. Assessing the oscillation is done by curve fitting a rectangular hyperbola and taking the norm of the differences of the fit and the loss curve. I will call this the oscillation value. The smaller this number, the less oscillation a curve will have. The decay speed is characterized by a weighted mean of the loss curve, with bias towards earlier epochs e.g., the first epochs are weighted heavier than the later. The weights are distributed linearly. The smaller this number, the faster the curve decays. I will call this number the baseline. Just for one or two curves, these numbers will not make much sense, but as I start comparing with other curves, they will become handy. Figure 37 shows how fitting two loss curves looks like. On the left we see the baseline at 0.547 and this curve has an oscillation value of 0.004.

On the right the baseline is at 0.540, the baseline gap is 0.007 and its oscillation is valued at 0.023. This oscillation is nearly two and a half times larger, than for the training curve.

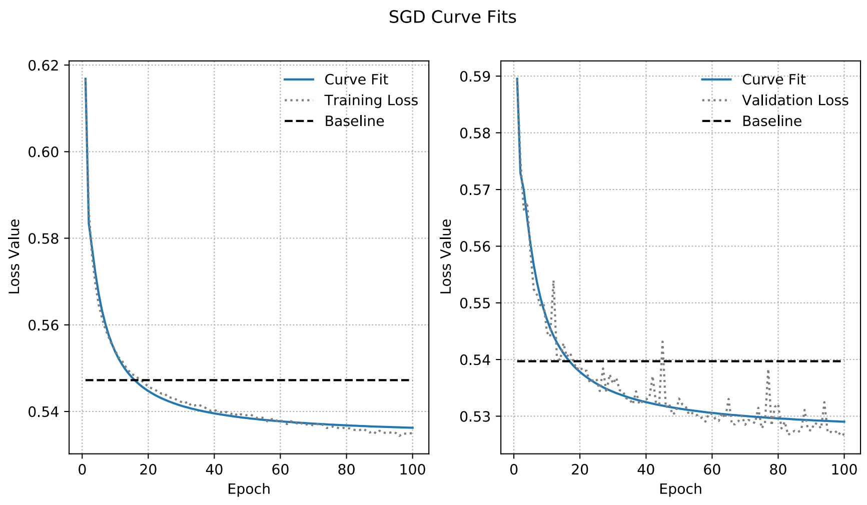


Figure 37: Illustration of Fitting the Loss Curves

One crucial point to understand here is that in order to have a good and generalizing model both baselines for training and validation should be close to each other. A validation baseline too low indicates, that the validation set is too small. A baseline too high, as compared to training, shows us that the network cannot generalize or that it is overfitting. (103) (104)

All test runs here are done for 100 epochs, because I wanted to keep them comparable and I will do longer and shorter test runs later on. There is no prior way of knowing for how long to train a network, but there are markers, that can tell us when to stop. The validation loss curve, if it shows an uptick, will tell us when we should stop the training.

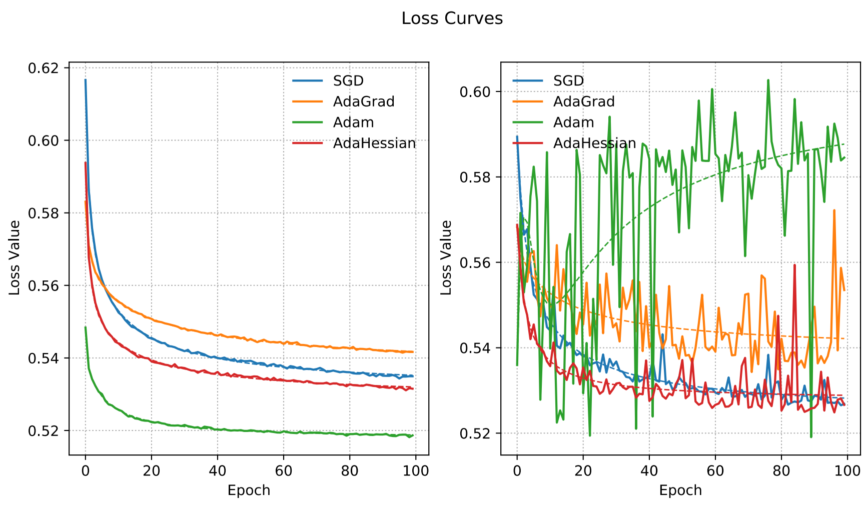


Figure 38: Test Runs for Optimizer

The loss curves for the optimizer test run are in Figure 38, on the left we see the training loss and on the right the validation loss for each epoch. The actual loss curves are solid, while the corresponding fit, on which the oscillation score is based, is plotted as dashed line of the same color.

The results for these runs are tabulated in Table 9 and Figure 39 shows the same results graphically, furthermore the confusion matrices for all four runs, on which the scores are based, are given in Appendix A. What we see is, that the training loss for all four runs are smooth, falling quickly and they all have a baseline around 0.52 and 0.55, with AdaGrad having the highest value and Adam the lowest. All four loss curves have an oscillation value below 0.1. The validation losses look only good for SGD and AdaHessian, their validation baselines only differ about 0.06 points from the training losses. This means, that the network did not over specialize on the dataset. Validation loss curves for AdaHessian and SGD look good, their oscillation values are below 0.05. AdaGrad, orange in Figure 38, is slowly falling. The baselines between training and validation loss differ only for 0.002. The validation loss is oscillating strongly, still it has a small oscillation value of 0.066. Adam was the worst performing optimizer in these runs. The training loss has the lowest baseline and validation has the highest baseline. This is still not the worst problem; it has by far the highest oscillation value.

Looking at accuracy, precision, Matthew Correlation Coefficient (MCC), the correctly and wrongly labled Slow Poins in Figure 39, we see a similar trend to what the loss curves already show. Adam achieves the lowest score in all, but in mislabels, the one we want to minimize. AdaGrad is a little better with a MCC of 0.463 as compared to Adams 0.433. SGD and AdaHessian performed comparably, with a MCC of 0.549 and 0.555 respectively. Ultimately, I decided against AdaHessian, because it more than doubled the training time as compared to the other three, but at the very end of this work I will get back to AdaHessian. From here onward I took Adam as my optimizer of choice. My thinking here is, that this three-layer network is too simple and that it has not enough parameters to adopt to the problem, as seen with Adams's validation loss curve. The training loss curve of Adam was the lowest, meaning it found a minimum on the loss surface the fastest. Going forward, with this reasoning in mind, I kept using Adam.

Table 9: Summary for Optimizer Test Runs

	SGD	*AdaGrad*	*Adam*	*AdaHessian*
Training Oscillation	0.004	0.002	0.002	0.004
Training Baseline	0.547	0.551	0.523	0.54
Validation Oscillation	0.023	0.065	0.17	0.045
Validation Baseline	0.54	0.549	0.568	0.534
Baseline Gap	0.008	0.002	0.044	0.006
Accuracy	0.775	0.731	0.717	0.777
Precision	0.766	0.725	0.715	0.772
MCC	0.549	0.463	0.433	0.555
Labeled Correctly	0.782	0.736	0.708	0.783
Mislabels	0.234	0.275	0.285	0.228

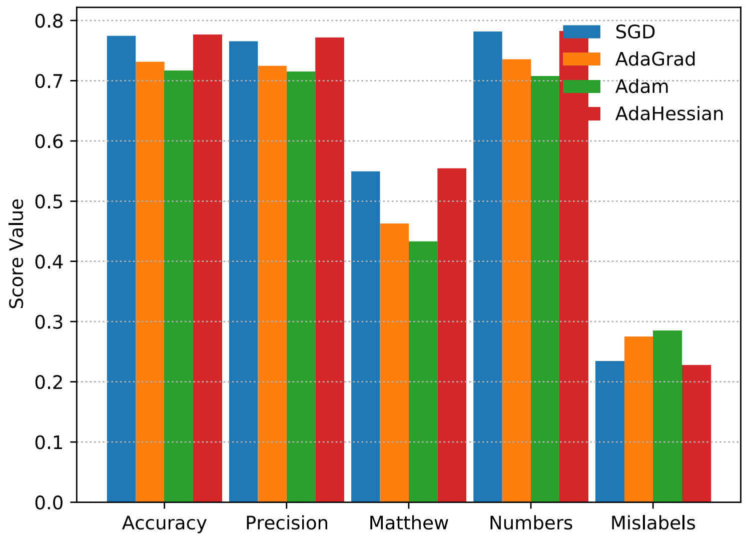


Figure 39: Summary for Optimizer Test Runs

7.2.2 Adjusting Learning Rate

In the next two runs, I tried to adjust the learning rate, in order to get Adam's validation loss to look smoother. In the last run, I used the same learning rate for all optimizers, in order to be able to compare them, but it was obviously too large for Adam.

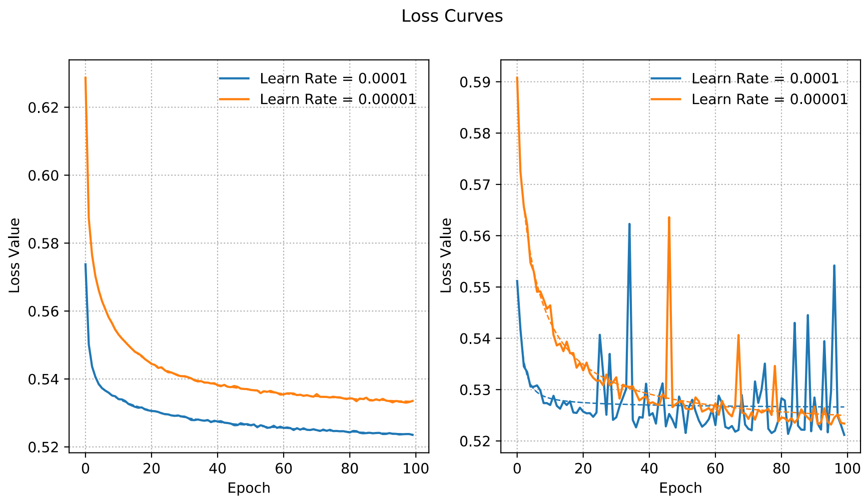


Figure 40: Adjusting the learning rate for Adam

The loss curves for the two tests are in Figure 40 and the summary with all scores are in Table 10 and as always, the confusion matrices will be in the Appendix A, this time together with the visual summary. We see that in both cases the baselines for training and validation loss are much closer to each other, previously the difference was 0.045, now it is 0.003 and 0.01, indicating, that the network could generalize better over the data set. The oscillation value now is just one third for a learning rate of one tenth as what we had before and it is even lower for a learning rate of one hundredth. The validation loss curves are now falling hyperbolas with some peaks during later epochs.

Table 10: Summary for learning rate Test Runs

	Learning Rate = 0.0001	*Learning Rate = 0.00001*
Training Oscillation	0.002	0.002
Training Baseline	0.531	0.546
Validation Oscillation	0.063	0.041
Validation Baseline	0.528	0.536
Baseline Gap	0.003	0.01
Accuracy	0.778	0.778
Precision	0.773	0.764
MCC	0.556	0.557
Labeled Correctly	0.785	0.817
Mislabels	0.227	0.236

7.2.3 Regularization through Drop Rates

In order to get a handle on these peaks in the validation loss curve I started with regularization. As I discussed earlier, there are methods to regularize over- and underfitting. Since all batches are normalized by this network, I am left with L1-, L2-regularization and dropout rates.

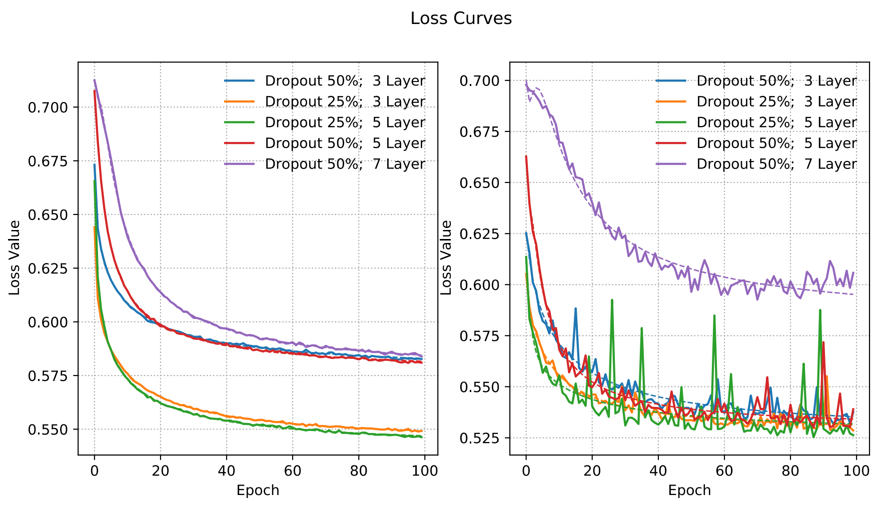


Figure 41: Regularization and Adding more Layer

We see the results in Figure 41 and the scores are in Table 11. I tested five setups with three- and five-layers combined with a dropout rate of 25 and 50% and seven-layers with 50% dropout rate. One striking result is, that the training loss baselines for 25% fall together at around 0.56 and for 50% at around 0.6. The baselines for validation loss for 25% are at 0.54 and for 50% at 0.557. The difference between the baselines for training and validation is a bit smaller for 25% and for 50% the baseline distances are at 0.046, which was the same as in the previous runs. All five curves have a hyperbolic shape, which was achieved by lowering the learning rate, but we still have a higher oscillation value. The seven-layer network has a similar shaped training loss curve as the three- and five-layer setups. The validation loss lies quite a lot higher, with its baseline at 0.634, 0.09 to 0.08 points higher as compared to the other four setups and the validation losses baseline is also 0.017 points higher than the training loss baseline. This leads to the conclusion that it generalized worse than the others.

Now looking at the scores in Table 11, we see, that the accuracy of all four setups come close to each other, hovering around 77%. The precision is a bit higher for the five-layer setups, but both the MCC and the number of all Slow Pions found, is lower for the five-layer setups.

Judging from this, I should go with the three-layer 25% setup. It found nearly 80% of all Slow Pions and it has the second best MCC, only being edged out by the three-layer 50% setup. It has the lowest oscillation score and the baselines are the closest together. Still, I chose the five-layer 50% setup, because it's MCC was not far from that of the three-layer 25% setup and it had fewer mislabels. The seven-layer setup had the fewest mislabels, but I decide against for the reasons given above and additionally because it has the lowest MCC. I will come back to this setup later again at the very end.

Table 11: Summary for Regularization

	Dropout 50%; 3 Layer	*Dropout 25%; 3 Layer*	*Dropout 25%; 5 Layer*	*Dropout 50%; 5 Layer*	*Dropout 50%; 7 Layer*
Training Oscillation	0.004	0.003	0.004	0.005	0.007
Training Baseline	0.599	0.566	0.565	0.603	0.617
Validation Oscillation	0.045	0.034	0.116	0.054	0.05
Validation Baseline	0.557	0.546	0.545	0.557	0.634
Baseline Gap	0.042	0.02	0.02	0.046	0.016
Accuracy	0.778	0.773	0.773	0.767	0.702
Precision	0.765	0.766	0.774	0.794	0.838
MCC	0.557	0.547	0.546	0.536	0.443
Labeled Correctly	0.795	0.784	0.761	0.72	0.512
Mislabels	0.235	0.234	0.226	0.206	0.162

7.2.4 Testing for Batch Size

Here I wanted to figure out, if increasing the batch size had any negative effects. Batch sizes are chosen in accordance to base two in the range of 32 to 256 (28).

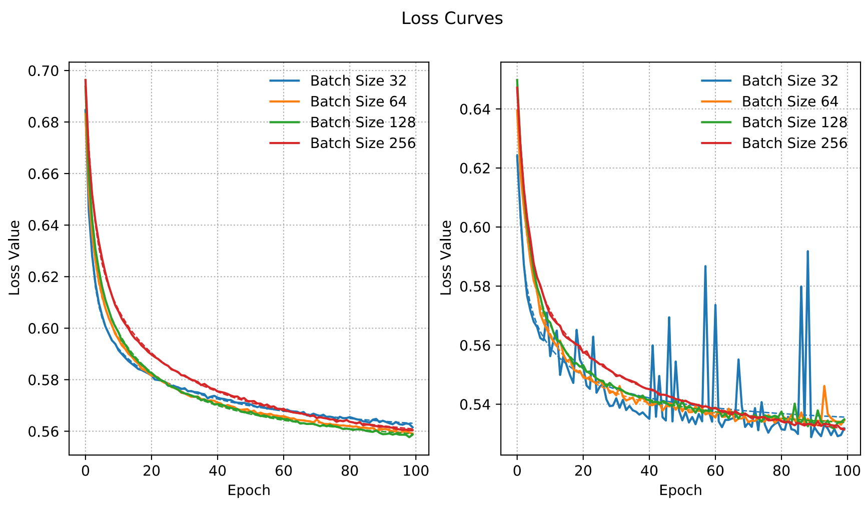


Figure 42: Different Batch Sizes

The loss curves are in Figure 42. The training losses all fall neatly together, with oscillation values below 0.008 and baselines around 0.58, only batch size 256 has a bit higher value. The higher baseline is due to the fact, that it falls slower. The validation loss shows something similar, with baselines a bit lower at 0.55. The oscillation values for batch size 64, 128 and 256 are all at 0.017, 0.011 and 0.006. Only batch size 32 had a large value, it grew by a factor of 18, from 0.006 up to 0.112.

From Table 12 we see, that the accuracy is around 77%, so is the precision, except for batch size 256. Batch sizes 32, 64 and 128 had similar number of mislabels of around 23% and a precision of 76.5%. All had a MCC of about 0.53 up to 0.54. Taking everything together one finds, that batch size 64 and 128 performed comparably, with 128 being nearly twice as fast. Going forward all runs will work with a batch size of 128.

Table 12: Summary for Batch Size Test Runs

	Batch Size 32	*Batch Size 64*	*Batch Size 128*	*Batch Size 256*
Training Oscillation	0.006	0.006	0.006	0.007
Training Baseline	0.583	0.584	0.584	0.591
Validation Oscillation	0.112	0.017	0.011	0.006

Validation Baseline	0.551	0.553	0.555	0.558
Baseline Gap	0.032	0.031	0.029	0.033
Accuracy	0.77	0.766	0.764	0.768
Precision	0.767	0.77	0.764	0.753
MCC	0.539	0.532	0.529	0.537
Labeled Correctly	0.788	0.768	0.76	0.796
Mislabels	0.233	0.23	0.236	0.247

7.2.5 Convolutional Layer – Finding Kernel Size

In the next few sections I will be looking at different setups for convolutional networks that are frontloaded to the linear layers. The objective for the test runs in this section is to find a suitable kernel size. Given the image size there are only two possibilities. PXD events are nine-by-nine matrices, as was discussed in *PXD – Pixel detector*, this leaves us at a kernel size of either three or five.

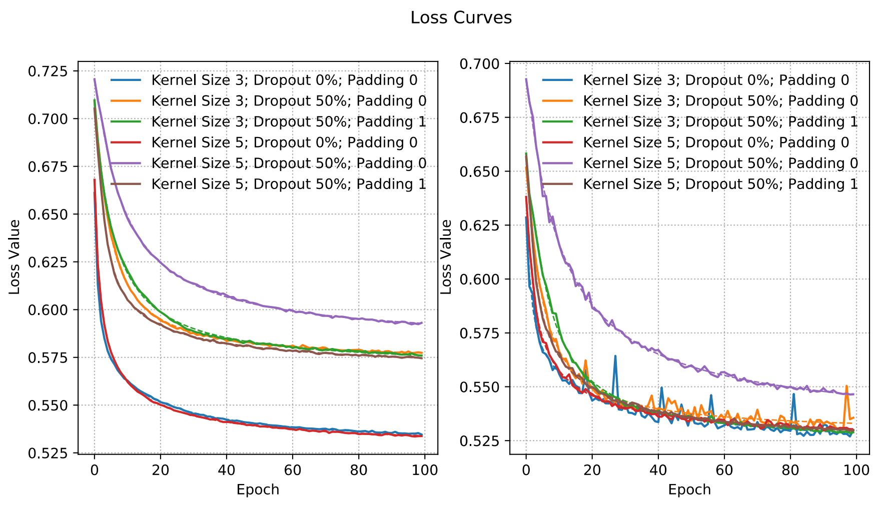


Figure 43: Different Setups for first Convolution

Figure 43 are the loss curves for the first test runs for the convolutional network and Figure 44 with Table 13 show the scores of these test runs. The training losses look good for all runs, the baselines and oscillations can be taken from the aforementioned table. The only striking thing, with zero padding and zero dropouts the training loss has a much lower baseline. This means it can adapt to the training data better, but generally this will lead to overfitting. As for the validation loss curves, all had a baseline between 0.544 and 0.56 and feeble oscillation, only with zero dropouts we got a bit bigger oscillation. This means all setups with dropouts, regardless of padding or no padding, could generalize well, if we only look at the loss curves.

If we want to use a kernel size of five, it is advisable to use padding, because all scores, MCC, accuracy, precision, number of Slow Pions and mislabels, got improved. As for kernel size three, in some instances padding improves the scores. MCC and accuracy got better scores. With number of Slow Pions found we have a jump from 72% to 79%. In other instances, padding reduces the score, the precision lowers from 78% down to 76% and the mislabels rise from 22% up to 24%.

Table 13: Summary for Finding a Kernel Size

	Kernel Size 3; Dropout 0%; Padding 0	*Kernel Size 3; Dropout 50%; Padding 0*	*Kernel Size 3; Dropout 50%; Padding 1*	*Kernel Size 5; Dropout 0%; Padding 0*	*Kernel Size 5; Dropout 50%; Padding 0*	*Kernel Size 5; Dropout 50%; Padding 1*
Training Oscillation	0.003	0.005	0.007	0.003	0.005	0.004
Training Baseline	0.553	0.6	0.602	0.553	0.626	0.596
Validation Oscillation	0.039	0.029	0.01	0.01	0.014	0.007
Validation Baseline	0.547	0.554	0.556	0.548	0.587	0.553
Baseline Gap	0.006	0.046	0.046	0.005	0.038	0.044
Accuracy	0.769	0.765	0.773	0.771	0.758	0.772
Precision	0.787	0.779	0.759	0.754	0.749	0.767

MCC	0.538	0.53	0.547	0.543	0.516	0.543
Labeled Correctly	0.751	0.719	0.792	0.803	0.79	0.803
Mislabels	0.213	0.221	0.241	0.246	0.251	0.233

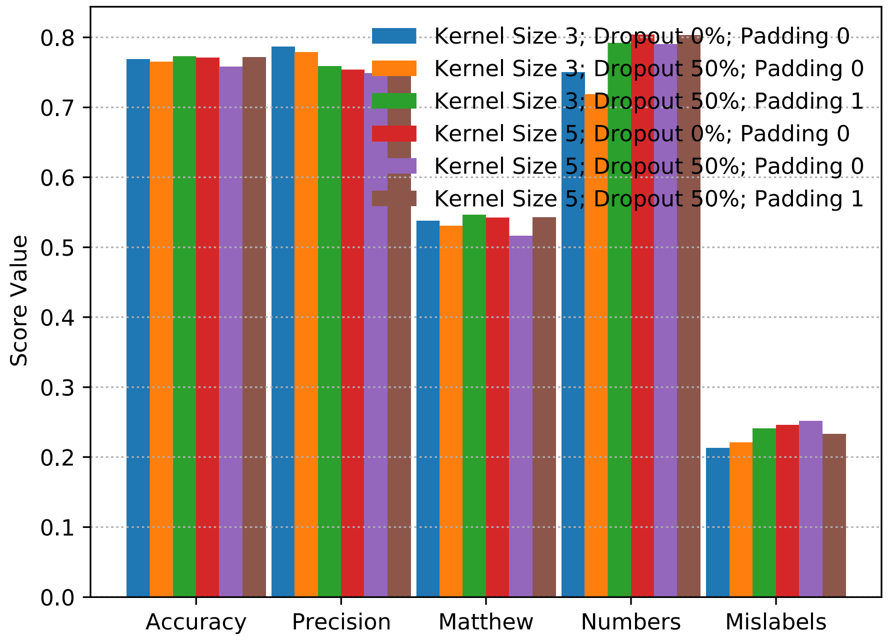


Figure 44: Summary for Finding a Kernel Size

Figure 44 shows the scores in form of a bar chart. Here we see again, that the MCC for the five setups are all above 0.51, but below 0.55, thus all performed similarly. The same goes for accuracy, which is in the upper mid-seventies. If we want to minimize mislabels, then we should go with the blue chart. It has the highest precision and found even more Slow Pions, then the second-best performer in terms of mislabels. As I said, the blue setup is not viable, because the gap between training and validation loss are too big. Hence, we should use the orange setup.

7.2.6 Convolutional Layer – Finding a Channel Width

The next step was to find how many convolutional layers should be used. It makes again sense to think about an upper bound based on the input data. Every convolution shrinks or compresses the image and thus it loses some information. This means that at most three convolutional layers make sense and one should rather increase the amounts of channels. Taking the formular for calculating the output image size after a single convolution, as was given in the section Convolutional layer, we can calculate:

$$kernel\ size\ 3\colon \frac{n_{in} + 2 \cdot padding - kernel\ size}{stride} + 1 = \frac{9 + 2 \cdot 0 - 3}{1} + 1 = 7 \rightarrow 49$$

$$kernel\ size\ 5\colon \frac{9 - 5}{1} + 1 = 5 \rightarrow 25$$

So, we see, if we use zero padding, we lose 32 pixels per convolution with kernel size three and 56 with kernel size five. Thus, is makes only sense to use at most three layers, since padding can only mitigate this loss of pixels and not solve the issue. As for the question of how many channels, I will get back to this question in this section.

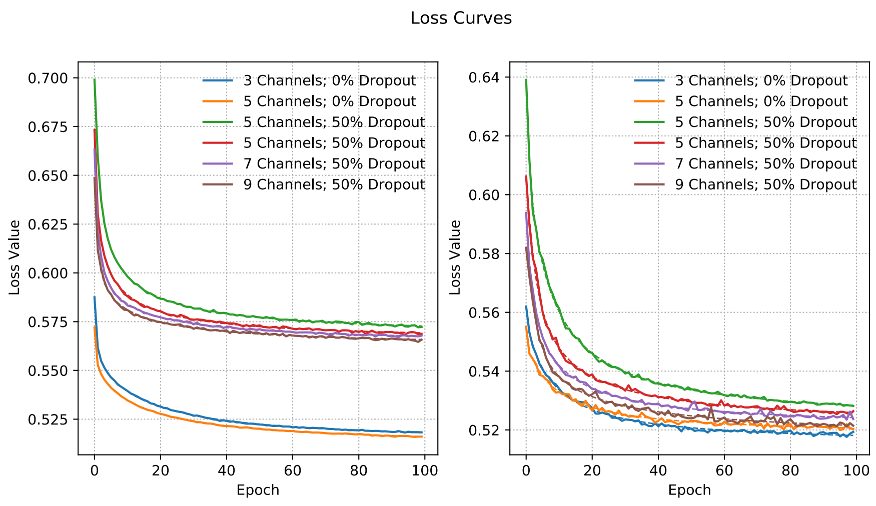


Figure 45: First Test Runs for Amount of Channels

All test runs in this section ran with kernel size three and zero padding on the first convolutional layer, otherwise the image would lose too many pixels after the second layer. The loss curves are in Figure 45. The summaries are in Table 14 and Figure 46. Generally, we see again the pattern, that zero dropout has lower baselines, but in all six cases the oscillations are around the same values as we have seen in previous sections. As for the scores, accuracy and precision are in all cases in the upper seventies. There is a pattern of finding more Slow Pions, but making more mistakes. It showed up earlier already and will continue to show up. Also judging from Figure 46 all setups performed comparably, thus it makes sense to go forward with just three channels, since using five or seven channels did not improve the results much, while increasing the calculation time slightly.

Another observation is, that the curves for three, five, seven and nine channels with 50% have successively lower final loss values in both training and validation.

If we look at three channels vs. just one channel, we already see, that the amount of mislabels increased, but at the same time also the number of Slow Pions found increased by the same amount. Furthermore accuracy rose, while precision fell about the same amount. In other words, we are left with balancing between finding more of what we want, while increasing our error rate or reducing the error rate with losing out on valuable data. Put another way we have to decide between quantity of data and purity of data, while having less to work with for future analyses. This is the pattern I already alluded to in earlier sections.

Table 14: Summary for first Channels Test Runs

	3 Channels; 0% Dropout	*5 Channels; 0% Dropout*	*5 Channels; 50% Dropout*	*5 Channels; 50% Dropout*	*7 Channels; 50% Dropout*	*9 Channels; 50% Dropout*
Training Oscillation	0.002	0.001	0.004	0.005	0.004	0.005
Training Baseline	0.531	0.528	0.59	0.583	0.579	0.576
Validation Oscillation	0.006	0.006	0.006	0.007	0.006	0.008
Validation Baseline	0.527	0.528	0.548	0.54	0.536	0.533
Baseline Gap	0.004	0.0	0.042	0.042	0.043	0.044
Accuracy	0.78	0.78	0.772	0.774	0.778	0.778
Precision	0.763	0.767	0.754	0.755	0.758	0.774

MCC	0.561	0.56	0.546	0.549	0.556	0.555
Labeled Correctly	0.816	0.804	0.812	0.813	0.802	0.798
Mislabels	0.237	0.233	0.246	0.245	0.242	0.226

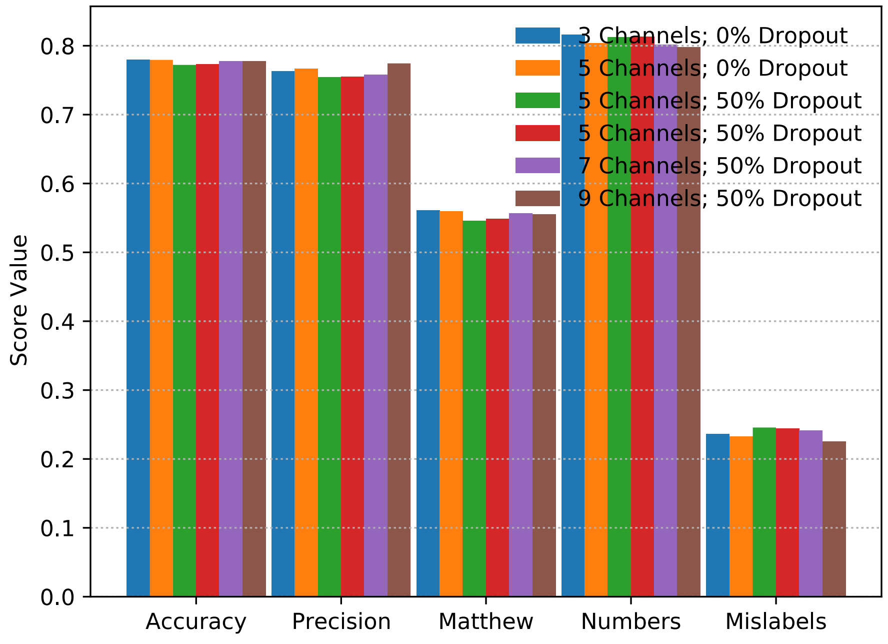


Figure 46: Summary for first Channels Test Runs

Judging from Figure 46, the introduction of evermore channels only marginally improves the performance of the network. In the charts we see, that precision and accuracy stay in the upper seventy percent and that MCC for all five runs is in roughly 0.55 and the mislabels are in the lower twenties. The runs with seven and nine channels ran roughly 40% to 60% longer than compared to the runs with three channels.

7.2.7 Convolutional Layer – How many Convolutions?

As I already mentioned it does not make sense to go beyond three convolutions, because then nothing is left of the original image. In this section I will compare two- to three-layers with just three and five channels and a kernel size of three. Furthermore, I will compare 50% dropout rate to 0% dropout rate in the linear layers of this network. This means there are eight different setups to this network.

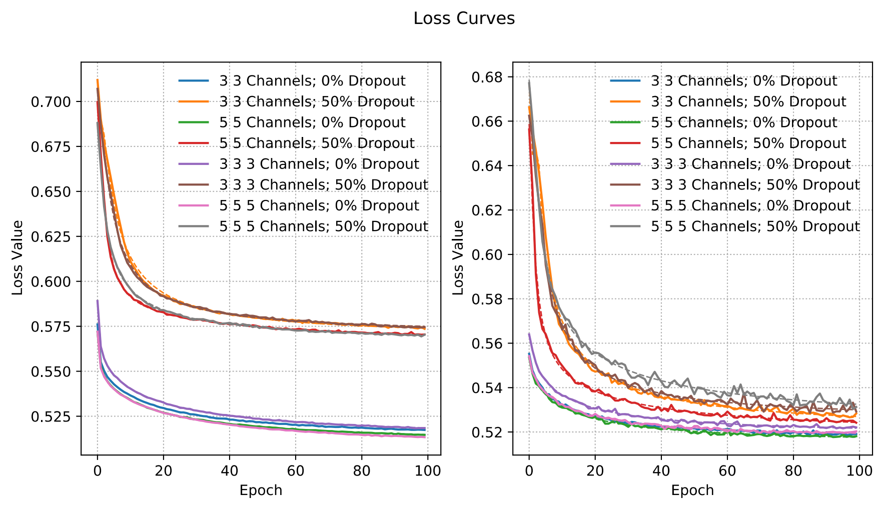


Figure 47: First Test Runs for Number of Convolutions

Figure 47 shows the loss curves and we see the usual, the training baselines for the zero dropout runs at around 0.53, while for the runs with 50% dropout close to 0.6. The validation baselines are at 0.53 for the zero dropouts and at 0.55 for those with dropouts. In all eight cases the oscillation values are below 0.03, which means the runs were good. These numbers were all taken from the summaries in Table 15 and Table 16.

Table 15: Summary for Number of Convolutions Test Runs, First Part

	3 3 Channels; 0% Dropout	*3 3 Channels; 50% Dropout*	*5 5 Channels; 0% Dropout*	*5 5 Channels; 50% Dropout*
Training Oscillation	0.001	0.014	0.001	0.005
Training Baseline	0.529	0.599	0.527	0.587
Validation Oscillation	0.004	0.026	0.005	0.015
Validation Baseline	0.527	0.555	0.526	0.542
Baseline Gap	0.002	0.044	0.001	0.045
Accuracy	0.781	0.773	0.782	0.777

Precision	0.776	0.778	0.767	0.759
MCC	0.561	0.547	0.564	0.554
Labeled Correctly	0.806	0.775	0.811	0.803
Mislabels	0.224	0.222	0.233	0.241

Table 16: Summary for Number of Convolutions Test Runs, Second Part

	3 3 3 Channels; 0% Dropout	*3 3 3 Channels; 50% Dropout*	*5 5 5 Channels; 0% Dropout*	*5 5 5 Channels; 50% Dropout*
Training Oscillation	0.002	0.009	0.001	0.004
Training Baseline	0.532	0.597	0.526	0.588
Validation Oscillation	0.005	0.016	0.005	0.016
Validation Baseline	0.531	0.555	0.527	0.56
Baseline Gap	0.001	0.043	0.001	0.028
Accuracy	0.777	0.771	0.779	0.762
Precision	0.762	0.79	0.755	0.815
MCC	0.555	0.542	0.56	0.531
Labeled Correctly	0.797	0.737	0.805	0.697
Mislabels	0.238	0.21	0.245	0.185

Figure 48 shows the same pattern, that I mention with every test run, the higher the precision, the lower the number of mislabels, but this inevitably leads to lower total number of Slow Pions found. If we compare now two to one convolution, we see that there is not much of a difference in performance. One convolution yields a MCC of 0.54 to 0.56 and two 0.53 to 0.555. With just one-layer dropouts increase the number of mislabels slightly from 23.5% to 24.5%, while the total number of Slow Pions remained stable at around 81%. With two layers there is no correlation between dropouts or no dropouts, we see the number of mislabels stays roughly between 22% and 24% and total number of Slow Pions is around 80%.

Something happens when using three layers. The two runs with zero dropouts are similar to the other six runs from this section and five runs from the previous section. They have a MCC of about 0.56 and mislabels of around 24% with total number of Slow Pions of 80%. With 50% dropout rate in the linear layers, we lose about 1% in accuracy, but we gain 3% to nearly 5% in precision and thus the mislabels fall to 21% for three channels and 18.5% for 5 channels. Interestingly enough the MCC falls a bit, down to 0.54 for three channels and 0.53 for five channels.

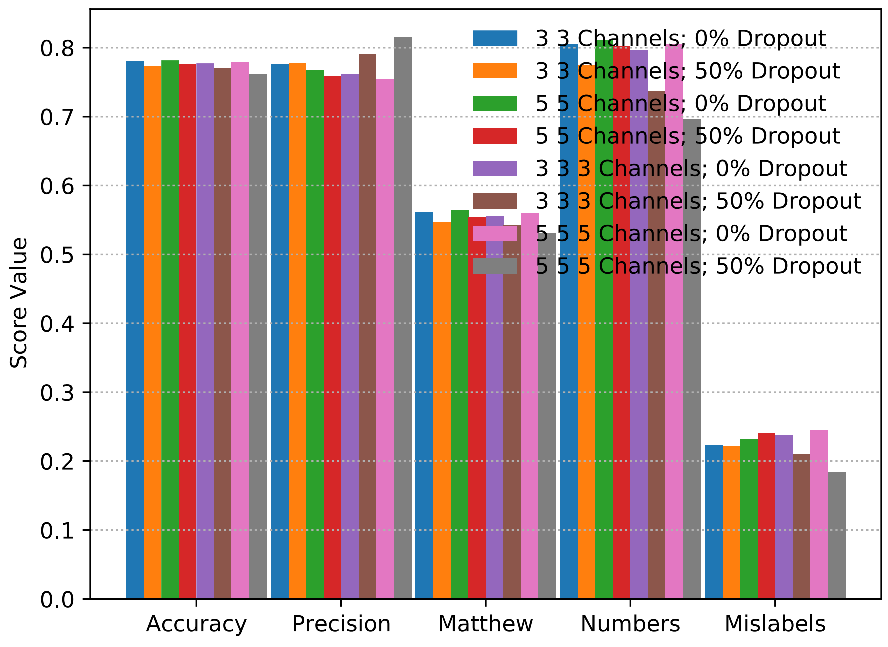


Figure 48: Summary for Number of Convolutions Test Runs

7.2.8 Transposed Convolutional Layer

Earlier we calculated, that given an image size of nine-by-nine pixels, it does not make much sense to go beyond three convolutions. There are possible remedies to this, one was padding, but this, as was already said, only offsets the loss of information. The other is using transposed convolutions, they were already discussed in Transposed

Convolutional Layer. One can think of something like up sampling an image, with the added benefit of having learnable parameters. This comes with its own caveats, like having more parameters, makes finding a minimum on the loss surface harder. I tested a simple three-layer, three channels setup and five linear layers with 50% dropout rates. I did four runs, one without a transposed layer and then three with a transposed layer at each possible point.

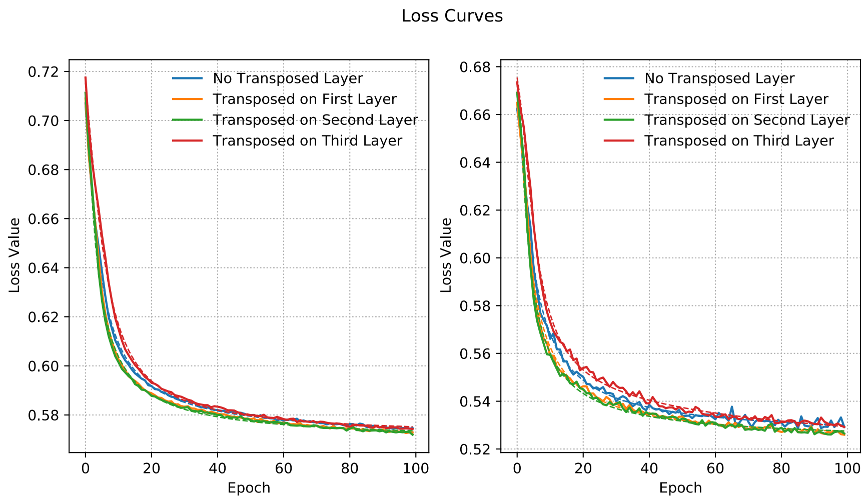


Figure 49: Loss curves for Transposed Convolutional Layer

Figure 49 shows the loss curve for the transposed convolution test runs. There is not much to say. They all have a baseline of around 0.6 for the training loss and 0.55 for the validation loss and an oscillation value below 0.02.

The accuracy for all of them is at 77% and it is the highest for the second layer convolution. The precision is a bit more fluctuating; the second layer is the highest of the networks with a transposed layer, with 78% and only the pure convolutional network is better here with 79%. Conversely, the pure convolutional network has the fewest mislabels and the network with the highest mislabels found the most Slow Pions. The accuracy and precision of a two-layer, three channels convolution is at 77% and 78% and 22% of mislabels. This means a transposed convolution brings neither an improvement over a three layer nor a two-layer setup.

Table 17: Summary for the Transposed Convolutional Test Runs

	No Transposed Layer	*Transposed on First Layer*	*Transposed on Second Layer*	*Transposed on Third Layer*
Training Oscillation	0.009	0.007	0.012	0.011
Training Baseline	0.597	0.595	0.594	0.601
Validation Oscillation	0.016	0.016	0.018	0.016
Validation Baseline	0.555	0.551	0.55	0.56
Baseline Gap	0.043	0.044	0.044	0.041
Accuracy	0.771	0.774	0.775	0.771
Precision	0.79	0.744	0.78	0.768
MCC	0.542	0.552	0.549	0.541
Labeled Correctly	0.737	0.822	0.765	0.767
Mislabels	0.21	0.256	0.22	0.232

The graphical summary is seen in Figure 50. Most notably is that the network without and transposed convolutional layer has the fewest number of mislabels and the network with a transposed layer as the first layer had the most mislabels, found the most Slow Pions, but had the lowest precision and the highest MCC.

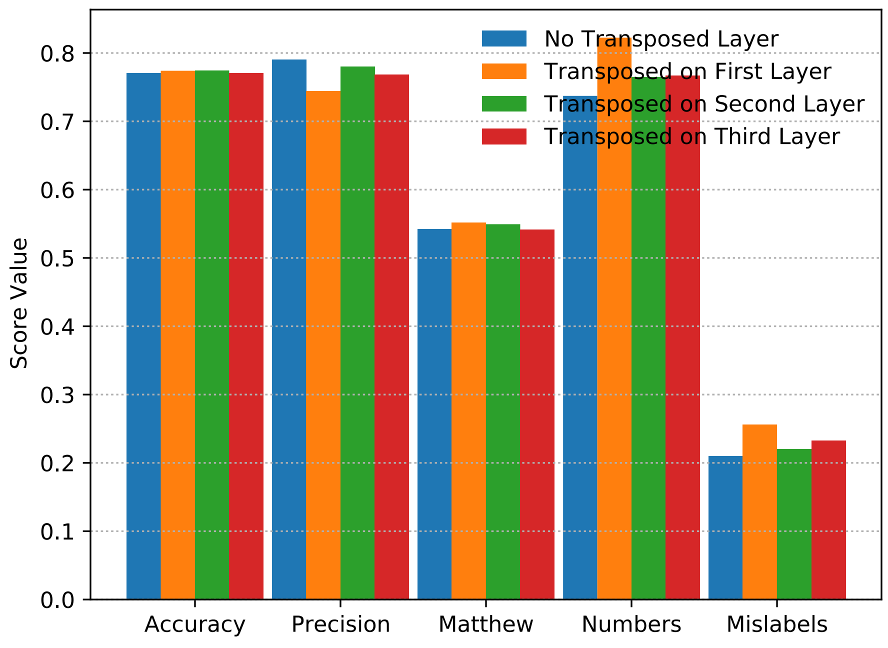


Figure 50: Summary for the Transposed Convolutional Test Runs

7.2.9 Learning Rate Schedulers

The next few tests will be less than exhaustive and there is a far larger range of experiments possible. This is probably true for most tests I made here, especially for the optimizers, but with learning rate schedulers I have a wide range of choices and parameters for each scheduler, plus I can combine several schedulers. In this sense I understand these tests only as a survey into learning rate scheduler.

In this and the immediately following sections I used a network with five linear layers, like with all convolutional test runs, with a dropout rate of 50% and one convolution with three channels and a kernel size of three. I used ReLU as activation function throughout all tests so far.

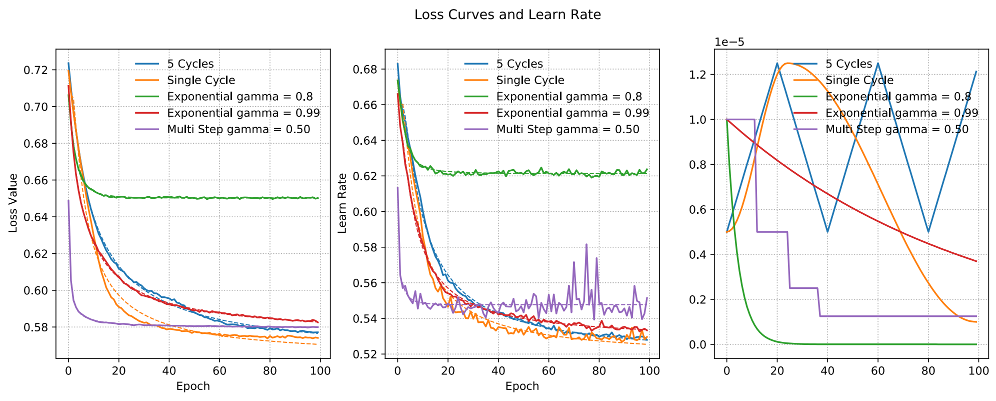


Figure 51: Loss Curves and Learning Rate for Scheduler Tests

Figure 51 shows the loss curves for different learning rate scheduler, on the left are the training losses, in the middle are validation losses and on the right are the different learning rates. So far this was not necessary, because I employed flat learning rates.

The first thing that one notices in the loss curves is, that the green curve, an exponential decay by 20%, the loss levels out early. This essentially means, that the network stopped learning. I will exclude this run from my further analysis, but I wanted to keep it, in order to illustrate this point. Looking at Figure 51 we see on the left, that the MultiStepper falls the fastest and indeed it has the lowest baseline of 0.584 and the lowest oscillation value at 0.001, the cyclical schedulers have stronger oscillations and higher baselines at 0.013 and 0.037.

Comparing training and validation baselines reveal a gap of about 0.04. With 0.048 at the high end is the Exponential scheduler and 0.034 at the low end is the MultiStepper. For validation oscillation the MultiStepper has the strongest at 0.062, but this probably due to the peaks after epoch 60. These peaks could be fixed through one additional step, but the step height would need to be adjusted, otherwise we will get the same issue, which we already have with the green curve.

One sees, that the MultiSteppers Training loss has a steep fall until epoch 12, this is where the learning rate decays for the first time. Like the green curves, the MultiSteppers loss bottoms out and remains flat.

The fewest mislabels were achieved by the Single Cycle scheduler with just 21.4% and the highest got the Exponential with 24.5%. We see again, that the fewer mislabels, the fewer Slow Pions in total, with exception of the MultiStepper, which came close in mislabels to the Single Cycle scheduler, but the number of Slow Pions correctly tagged correlated negatively with mislabels.

Table 18: Summary for Learning Rate Scheduler

	5 Cycles	*Single Cycle*	*Exponential* γ *= 0.8*	*Exponential* γ *= 0.99*	*Multi Step* γ *= 0.50*
Training Oscillation	0.013	0.037	0.004	0.006	0.001
Training Baseline	0.614	0.602	0.654	0.609	0.584
Validation Oscillation	0.017	0.042	0.012	0.014	0.062
Validation Baseline	0.567	0.559	0.626	0.561	0.55
Baseline Gap	0.047	0.043	0.029	0.048	0.034
Accuracy	0.775	0.772	0.673	0.769	0.743
Precision	0.764	0.786	0.742	0.755	0.784
MCC	0.55	0.545	0.362	0.54	0.49
Labeled Correctly	0.782	0.762	0.536	0.791	0.671
Mislabels	0.236	0.214	0.258	0.245	0.216

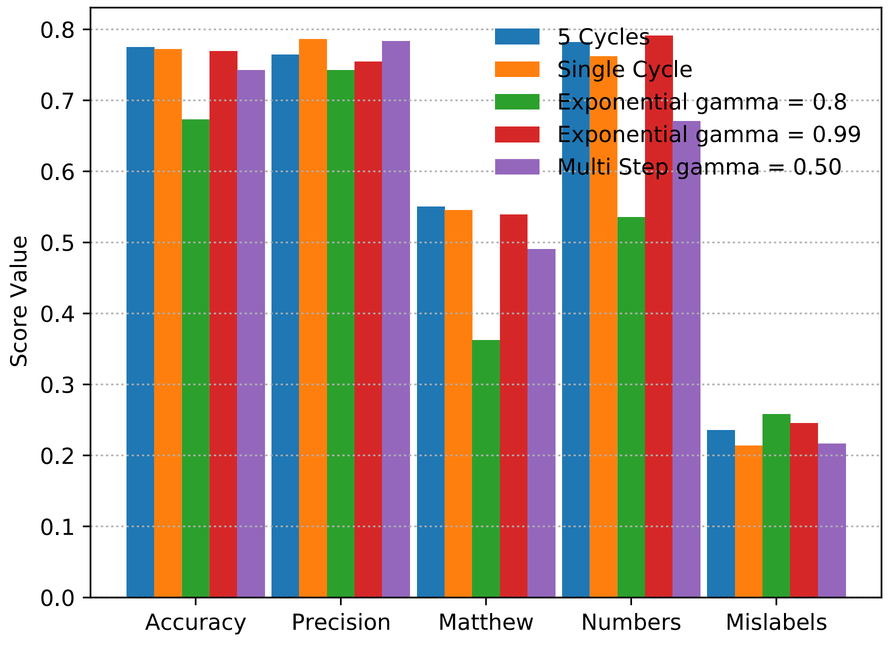


Figure 52: Summary for Learning Rate Scheduler

It is instructive to not just look at number, but also at a visual presentation, Figure 52 shows this. Here the Exponential scheduler with a decay of about 20% per epoch, the green bar, is obviously the worst performer. In every score it is falling far behind. Another thing becomes pretty clear, the MultiStepper found a lot less Slow Pions, than the rest. Looking at the learning rate over time on the right in Figure 51 I think it becomes obvious, that the MultiSteppers learning rate fell to fast and that could be one reason, why it found less Slow Pions, than the others.

The conclusion one can draw from these test runs is, that the learning rate not only has an upper bound, but also a lower bound, below which a network stops working. And the last point is, that the decay speed of the learning rate is of utmost importance.

7.2.10 Activation Functions

In this section I tested different activation functions, I refer to *Activation Functions* in order not to plot them here again. I tested ReLU, LeakyReLU, Sigmoid, Softplus and Tangent Hyperbolic. There are a few more, but these are sufficiently different from one another, that made them of interest, just to cover a wider ground.

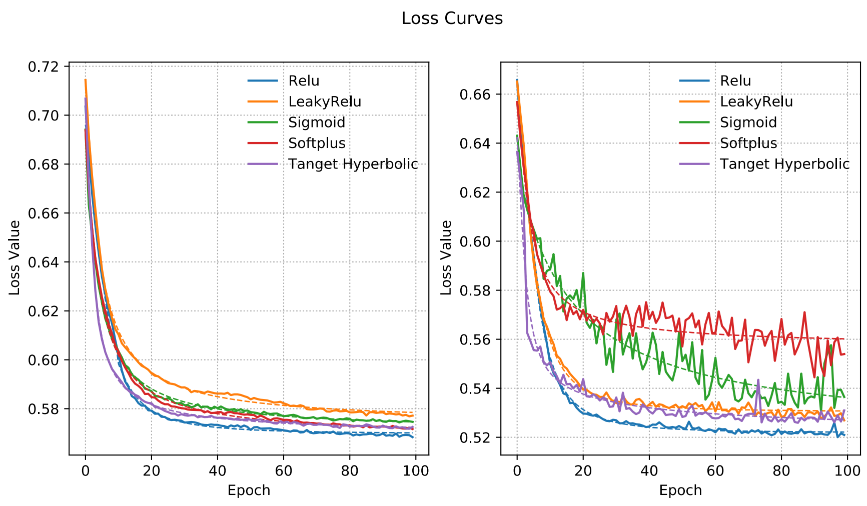


Figure 53: Loss Curve for Activator Test Runs

The loss curves are in Figure 53, the training losses for all five curves have a small oscillation value with ReLU having the highest at 0.015 and Tangent Hyperbolic the lowest at 0.005. The baselines are all between 0.588 and 0.601, with Tangend Hyperbolic having the smallest and LeakyReLU the highest.

With validation loss it is similar, the oscillation is the smallest with ReLU at 0.013 and the highest with Sigmoid. The lowest Baselines have Tangent Hyperbolic and ReLU together at 0.542 and the highest has Softplus and thus the gaps between the training and validation loss baselines are between 0.051 for LeakyReLU and 0.016 for Softplus. The accuracies are all at 77% give or take, only Softplus is worse at 72.5%. With precision it is again the other way around, Softplus has the highest at 78% and ReLU the lowest at 76%.

Table 19: Summary for Activation Test Runs

	ReLU	*LeakyReLU*	*Sigmoid*	*Softplus*	*Tanget Hyperbolic*
Training Oscillation	0.015	0.011	0.007	0.011	0.005
Training Baseline	0.589	0.601	0.593	0.592	0.588

Validation Oscillation	0.013	0.015	0.064	0.055	0.037
Validation Baseline	0.542	0.55	0.568	0.576	0.542
Baseline Gap	0.047	0.05	0.025	0.016	0.045
Accuracy	0.779	0.774	0.771	0.725	0.77
Precision	0.762	0.766	0.777	0.781	0.769
MCC	0.559	0.548	0.543	0.459	0.54
Labeled Correctly	0.798	0.794	0.75	0.629	0.774
Mislabels	0.238	0.234	0.223	0.219	0.231

Figure 54 shows the results visually and one notices, that all five, but Softplus, performed similarly. Softplus has by far the lowest MCC and found nearly only 60% of all Slow Pions. It is barely the best performer for mislabels. The conclusion of this test can only be, that four out of five activation functions worked well and that Softplus should remain only as the activation function for the very last layer, as it was discussed in Methodology.

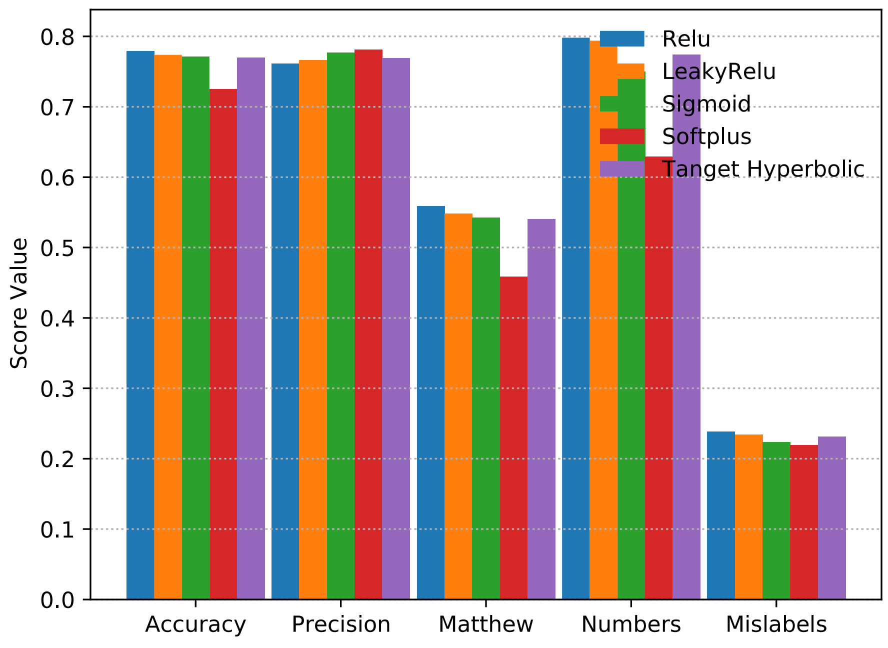


Figure 54: Summary for Activation Test Runs

7.3 Results

誰にも運命はかえられないだが、ただ待つかみずからおもむくかは決められる。

ヒイサマ

In the previous section I presented different test runs to figure out which network would be suitable for the purposes of finding Slow Pions. The results were, that the simplest network will suffice or at least, that bigger networks will not improve the results substantially. My approach to this now is to run the baseline network, the smallest network with a single convolutional layer and the biggest network configuration from the previous section, for 25, 50, 100, 150 and 200 epochs. They all ran with Adam as the optimizer and a 5 cycles learning rate scheduler. The network configurations are in Table 20: Network Setups. Unlike the previous tests, I ran these tests with Slow Pions against all other categories, as they were discussed in Simulated data, combined into one. Then I will compare the results and take training and validation time into account in assessing the viability of the bigger network.

Table 20: Network Setups

Network	*Small Network*	*Medium Network*	*Large Network*
Convolutional Layer	0	1	3
Output Channels	-	3	5
Kernel Size	-	3	3
Padding	-	1	1
Transposed Convolution	-	-	no transposed convolution
Activation Function	-	ReLU	ReLU
Linear Layer	5	5	5
Layer Width	81	243	405
Dropout Rate	50%	50%	50%
Activation Function	ReLU	ReLU	ReLU

7.3.1 Long-Term Tests

The first network I ran was the Small Network, see Table 20, the loss curves as in Figure 55 and the scores are summarized in Table 21 and in Figure 56.

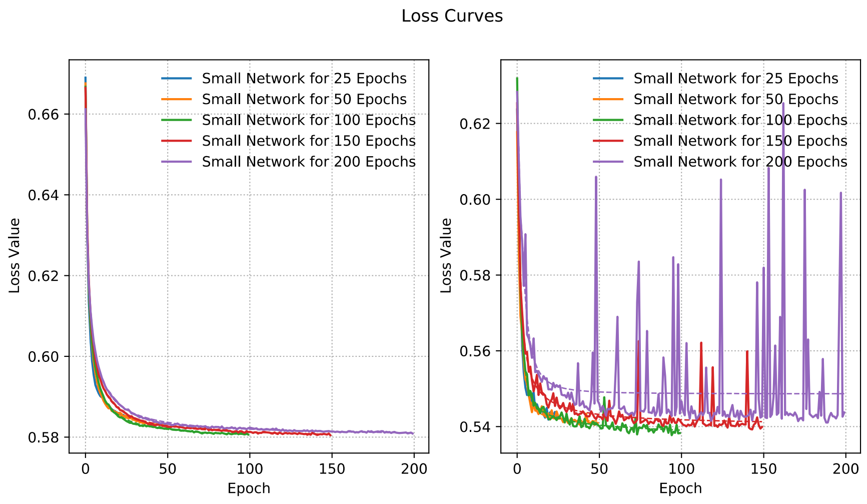


Figure 55: Loss Curve for Small Network

The training loss curve for the Small Network level all out at around 0.59, but the baselines for the shorter runs lay a bit higher, for 25 epochs it is at 0.602 and there seems to be a convergence between 150 and 200 epochs at 0.587. There is nothing to say about the training oscillations, which are low for all five runs at 0.001 and 0.003 in case of 200 epochs.

The validation baselines lay around 0.55, give or take and thus the gap is about 0.04, 25 and 50 epochs runs lay bit higher. Only the gap for 200 epochs is smaller with 0.034. The oscillation values are equal or smaller than 0.012 up to 100 epochs. Then there is a jump at 150 epochs to 0.04, which is still acceptable and due to the peaks. This could be solved by tweaking the learning rate. The 200 epochs run has the strongest oscillations with 0.186. This is probably due to the cyclical learning rate scheduler, which maintained five cycles and I should probably adjust it, but I wanted to keep it comparable to the two other setups, which I have tested in this section.

Table 21: Summary for Small Network

	Small Network for 25 Epochs	*Small Network for 50 Epochs*	*Small Network for 100 Epochs*	*Small Network for 150 Epochs*	*Small Network for 200 Epochs*
Training Oscillation	0.001	0.002	0.002	0.002	0.003
Training Baseline	0.602	0.595	0.589	0.587	0.587
Validation Oscillation	0.007	0.007	0.012	0.04	0.186
Validation Baseline	0.559	0.551	0.547	0.548	0.553
Baseline Gap	0.043	0.044	0.042	0.04	0.034
Accuracy	0.76	0.762	0.765	0.764	0.758
Precision	0.741	0.748	0.74	0.742	0.752
MCC	0.522	0.526	0.533	0.53	0.515

Labeled Correctly	0.8	0.788	0.834	0.804	0.749
Mislabels	0.259	0.252	0.26	0.258	0.248

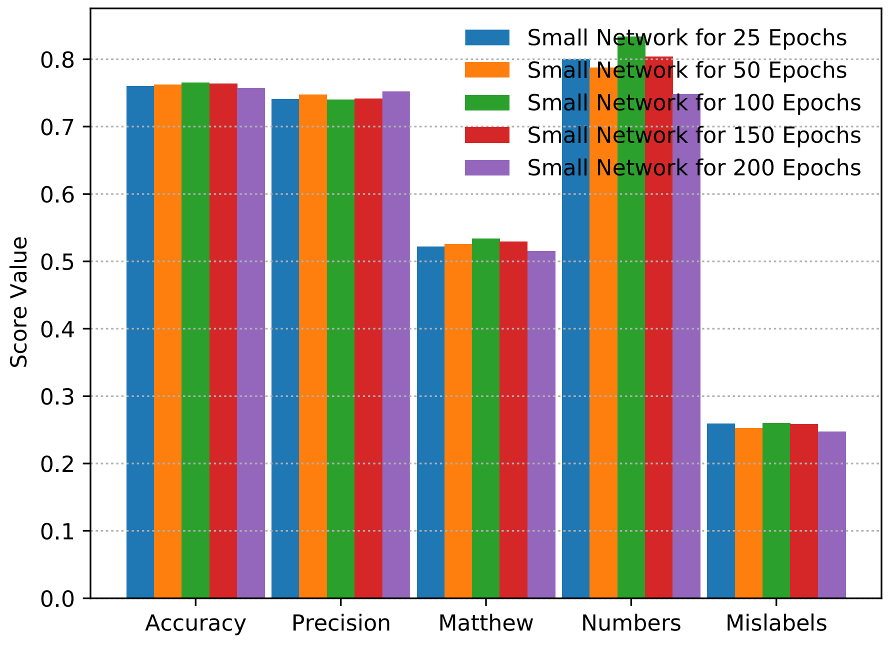


Figure 56: Summary for Small Network

The scores are graphically summarized in Figure 56. All five runs had similar scores for accuracy and precision, both are in the mid-seventies, all MCCs are above 0.5 and no run falls below 20% or above 30% mislabels. Interesting is only that the 100 epochs run found above 83% of all Slow Pions and is thus 3% points ahead of the others.

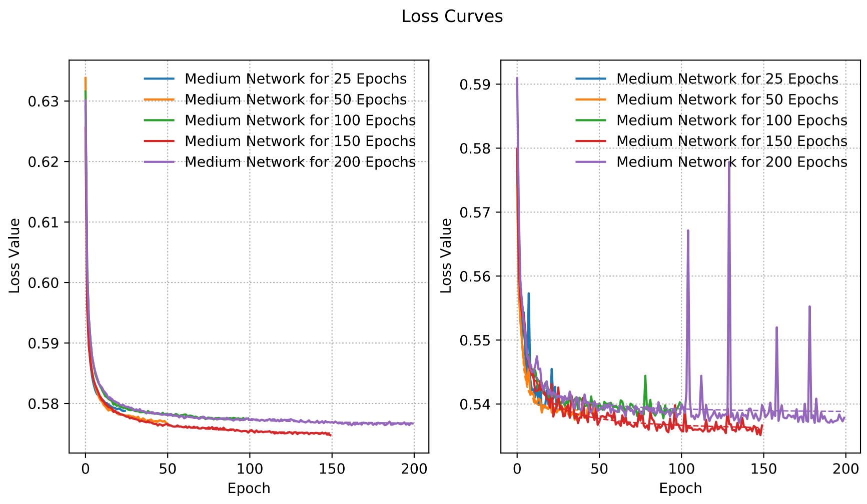


Figure 57: Loss Curves for Medium Network

Figure 57 shows the results for the Medium Network and the scores are summarized in Table 22. Only the validation curve for 200 epochs exhibits some peaks, which occur past epoch 100. The oscillation values for training loss are all less or equal to 0.002 and the baselines are near 0.58 with a low point for 150 epochs. The validation loss curves have oscillation values less or equal to 0.012, with the exception of the 200 epochs run. There the oscillation is 0.054, this is due to the aforementioned peaks. The baseline gap is relatively constant with 0.038.

Table 22: Summary for Medium Network

	Medium Network for 25 Epochs	*Medium Network for 50 Epochs*	*Medium Network for 100 Epochs*	*Medium Network for 150 Epochs*	*Medium Network for 200 Epochs*
Training Oscillation	0.001	0.001	0.001	0.002	0.002
Training Baseline	0.587	0.583	0.582	0.578	0.58
Validation Oscillation	0.012	0.008	0.009	0.012	0.054
Validation Baseline	0.549	0.543	0.543	0.54	0.542

Baseline Gap	0.038	0.04	0.038	0.038	0.038
Accuracy	0.763	0.762	0.761	0.767	0.764
Precision	0.756	0.758	0.751	0.754	0.737
MCC	0.526	0.524	0.521	0.534	0.531
Labeled Correctly	0.77	0.775	0.76	0.798	0.813
Mislabels	0.244	0.242	0.249	0.246	0.263

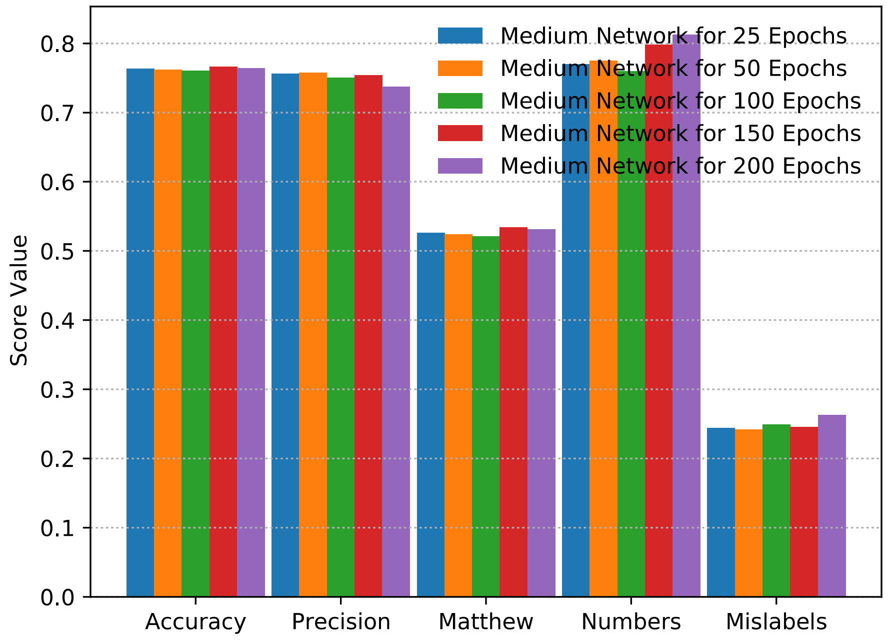


Figure 58: Summary for Medium Network

Figure 58 shows the visual representation of the scores for the Medium Network. All scores again fall to in the ballpark and they are similar to what we had before. The accuracy and precision are again in the mid-seventies. Generally, the Medium Network got lower scores for number of Slow Pions found, with an uptick for longer training time. It had slightly less mislabelled, with the exception for the 200 epochs run. Whereas the Small Network got around 25% mislabels, the Medium Network gained half a percentage.

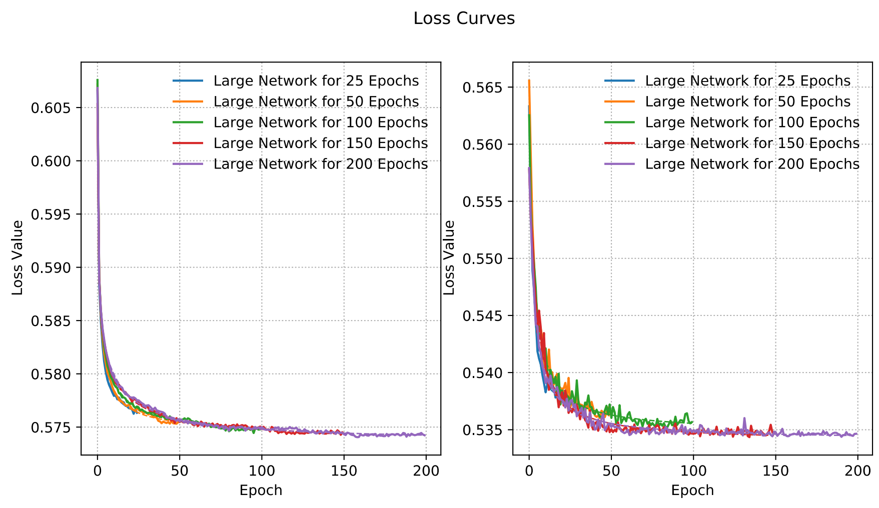


Figure 59: Loss Curves for Large Network

We are coming finally to the Large Network. Its loss curves are in Figure 59 and the scores are summarized in Table 23. Now all oscillations for training loss are equal to or less than 0.004 and the baselines are again at 0.580, with a convergence toward 0.577 for 150 epochs. The validation loss has an oscillation value of 0.006 or less, which is so far the best scores and a baseline at 0.54 with a convergence toward 0.537 for 150 epochs. The baseline gaps stay constant at 0.04.

The accuracy increases with more epochs, starting at 76.3% with 25 epochs and ending at 76.7% for 150 and 200 epochs. The precision declines from 76.1% for 25 epochs down to 74.7% for 200 epochs. The number of Slow Pions found and the mislabels increases from 77.7% and 23.9% for 25 epochs up to 81.1% and 25.3% for 200 epochs.

Table 23: Summary for Large Network

	Large Network for 25 Epochs	*Large Network for 50 Epochs*	*Large Network for 100 Epochs*	*Large Network for 150 Epochs*	*Large Network for 200 Epochs*
Training Oscillation	0.001	0.001	0.002	0.002	0.002
Training Baseline	0.582	0.58	0.578	0.577	0.577

Validation Oscillation	0.002	0.004	0.005	0.006	0.005
Validation Baseline	0.544	0.542	0.539	0.537	0.537
Baseline Gap	0.038	0.037	0.039	0.04	0.04
Accuracy	0.763	0.765	0.765	0.767	0.767
Precision	0.761	0.754	0.754	0.748	0.747
MCC	0.526	0.531	0.531	0.536	0.535
Labeled Correctly	0.777	0.802	0.798	0.807	0.811
Mislabels	0.239	0.246	0.246	0.252	0.253

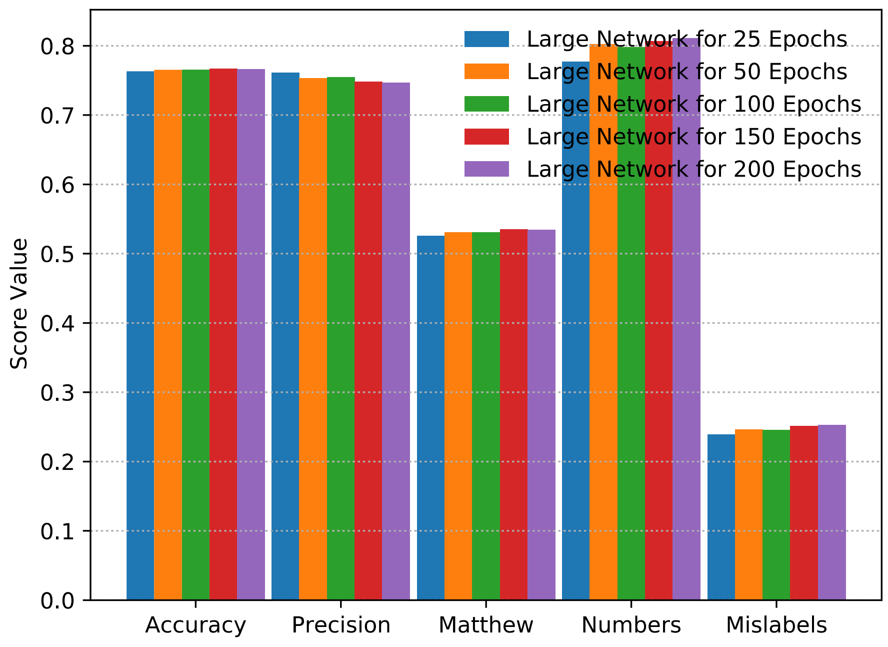


Figure 60: Summary for Large Network

Figure 60 confirms what I have descripted so far. There is a slight uptick in accuracy, MCC, amounts of Slow Pions found and mislabels with an increase in training time and a drop in precision. Overall, the scores are similar.

7.3.2 Tests against Single Particles

Now with previous results at hand, I want to look into how the network can differentiate between Slow Pions and specific other particles. This should give some insight into which events can make problems and from which particles should stem the most mislabels and against which the network losses most Slow Pions.

The long-term tests were done with every category combined into one against Slow Pions. Here I will pick one event at a time and compare it against slow pions. I will keep balanced training sets, meaning, that both categories will be at the same size, but I will utilize the biggest possible set per category. I will test the following:

- Anti-Deuterons (DD)
- Protons (PP)
- Pions (PI)
- Kaon (KK)
- Muon (MM)
- Electrons (EL)
- Beam Background (BB)
- and Gammas (GA)

The second last category was tested already in the section Process in order to build up this network. The last category will not be very representative, since there are not many gamma events, as is shown in Figure 16 in section Simulated data.

The tests in this section will use the Large Network, its configuration is in Table 20. I concluded from the last section, that this network will have highest capability to generalize over the data sets. Determining from the last section I will use 150 epochs, since 200 epochs ran 25% longer, but did not improve a networks performance sufficiently. I plotted the training results from the long-term tests according to number of epochs, comparing network sizes. These plots will be in the Appendix A. They show again, that 150 epochs gave the best performance and that the Large Network could generalize the best.

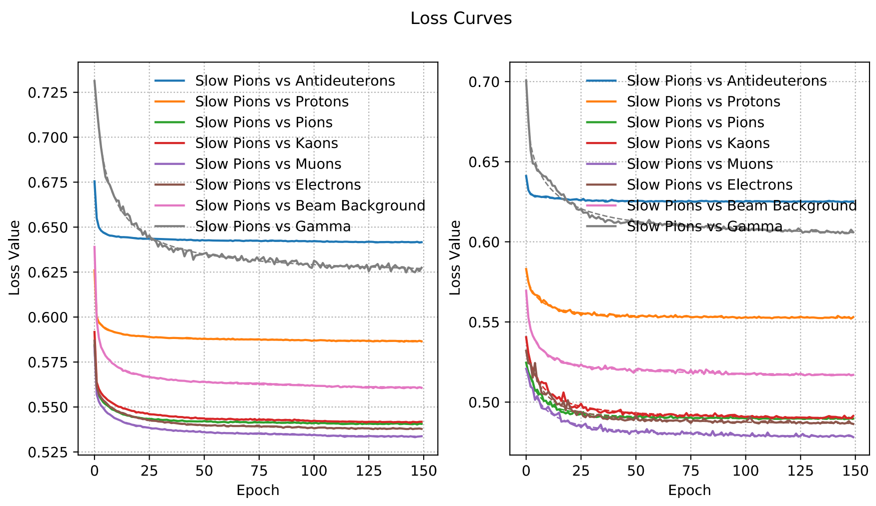


Figure 61: Loss Curves for the Single Particle Tests

As I already mentioned, here I tested the performance against individual particles in order to find out where the network fails. This is assuming a level of independence for identifying these particles. Figure 61 shows the loss curves for these test runs and Table 24 summarizes the results.

The size of the data set for gammas is 1.5% of others, this makes it not a viable option for a comparison to the others. Its loss curve can be seen in grey in Figure 61. This is why I will ignore it for now.

The training oscillations are equal or less than 0.004, the biggest oscillations happened with Beam Background. Validation oscillations are a tad bit larger, but stay below 0.016, this value was achieved against electrons.

The baselines, training and validation, have a large spread. In both cases the sequence is the same, the highest are the Anti-Deuterons baselines at 0.644 and 0.626 with the smallest gap with 0.018. This shows good generalization, but looking at the scores shows, that the Anti-Deuterons show the highest mislabels score of nearly 35% and the smallest number of Slow Pions at 66%. This run has also one of the lowest MCCs of this work with a score of 0.283, showing a weak correlation.

The next baselines are for Protons at 0.589 and 0.556 with a gap of 0.033. Again, this would imply good generalization, but here we have the second worst scores for mislabels and number of Slow Pions at 26.3% and 77.6%. Here the MCC is 0.484, which is more in line of what we have seen so far.

Beam Background has the next baselines at 0.567 and 0.523 with a gap of 0.044, this is close to the gaps of the other particle test runs. It is the third worst in mislabels and number of Slow Pions found with 23.1% and 79%. This is right in between the results for Anti-Deuterons and Protons and the remaining particles. Beam Background achieved a MCC of 0.567, which is a stronger correlation than what we had in the long-term tests.

The remaining particles, Pions, Kaons, Electrons and Muons, have training baselines between 0.539 and 0.546, with Muons having the lowest. Their validation baselines are in the range of 0.498 and 0.486, again with Muons being the lowest. The gaps are around 0.05, with Muons having the biggest gap. In all cases the amount of mislabels was below 20%. Pions had the most mislabels at 19.9% and Muons the fewest at 18.4%. The number of Slow Pions exceeded 82% in these four cases. This together gave MCCs of more than 0.62, showing stronger correlations than any other run so far.

Table 24: Summary for the Single Particle Tests

	DD	*PP*	*PI*	*KK*	*MM*	*EL*	*BB*	*GA*
Training Oscillation	0.002	0.003	0.002	0.003	0.003	0.003	0.004	0.017
Training Baseline	0.644	0.589	0.544	0.546	0.539	0.543	0.567	0.644
Validation Oscillation	0.003	0.007	0.01	0.012	0.011	0.016	0.008	0.02
Validation Baseline	0.626	0.556	0.494	0.498	0.486	0.494	0.523	0.62
Baseline Gap	0.018	0.033	0.05	0.049	0.053	0.049	0.044	0.025
Accuracy	0.642	0.742	0.816	0.812	0.827	0.818	0.784	0.67
Precision	0.652	0.737	0.801	0.813	0.816	0.804	0.769	0.657
MCC	0.283	0.484	0.632	0.624	0.655	0.636	0.567	0.34

Labeled Correctly	0.66	0.776	0.833	0.823	0.842	0.829	0.79	0.747
Mislabels	0.348	0.263	0.199	0.187	0.184	0.196	0.231	0.343

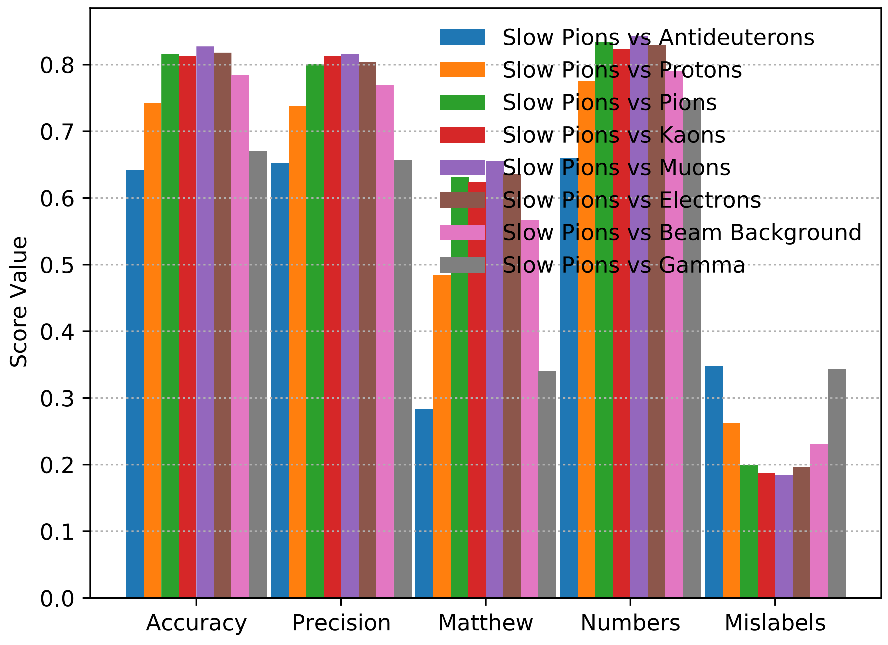


Figure 62: Summary for the Single Particle Tests

Figure 62 shows the graphical representation of the summary from Table 24. From the charts and what we discussed above, we can deduce, that the network has minor issues with the lighter particles, such as Mesons and Leptons. Anti-Deuterons are clearly out of line with the other particles, except for gammas. It has the lowest accuracy, precision and MCC, found the fewest Slow Pions and has the most mislabels. Less pronounced, but still clearly visible is the performance for protons. Beam Background falls close to protons, assessing it purely visually, so I can assume, that the network is also struggling with it. In the previous section the MCC of the Large Network for 150 epochs was at 0.536. The mean value of MCCs from these runs is 0.554, if I include gammas, then the MCC falls down to 0.528. The score for the previous falls right between these two.

7.3.3 Multiclass Tests

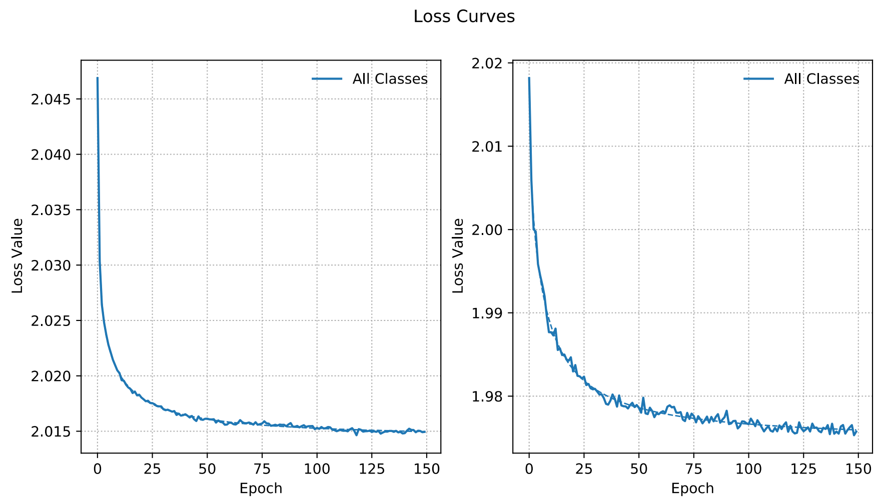


Figure 63: Loss Curves for Multiclass Test Run

Figure 63 shows the loss curves for a multiclass test run. I tested the same categories as I descripted above, but this time all at the same time. The baselines here are at 2.018 for training and 1.982 for validation loss and it has a baseline gap of 0.036. The training loss has an oscillation of 0.002 and the validation has 0.006.

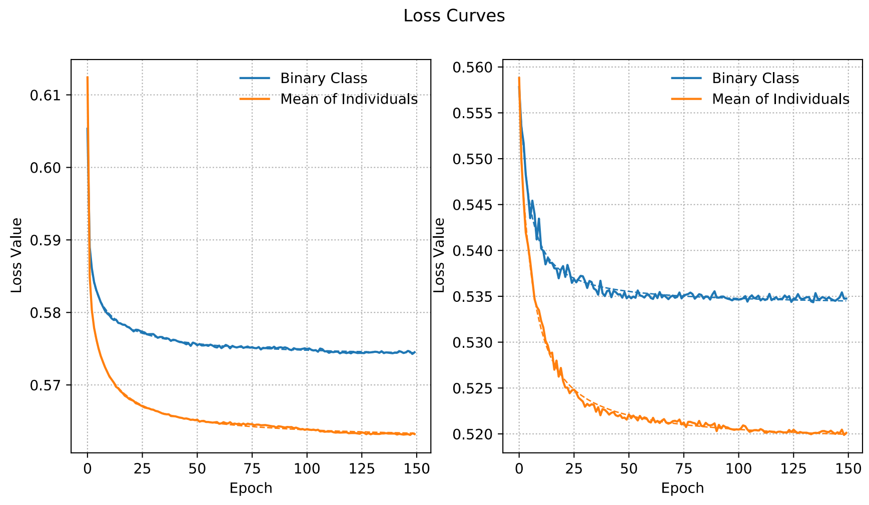


Figure 64: Loss Curves for Binary Test Runs

Figure 64 shows the loss curves for the Large Network for 150 epochs in blue and in orange we see the averaged curves for the individual particle tests from the most recent section. The baselines for the binary test are at 0.577 and 0.537 with a gap of 0.04 and the averaged baselines for the individual particles are at 0.568 and 0.525 with a gap of 0.042. While the baselines are not directly comparable to that of the multiclass test run, the gap can be compared, as it is a difference to be minimized. The gaps for the multiclass run is at 0.036. The gap for the binary run and the average gap for individual particle runs are at around 0.04. The oscillation values are the same as for the multiclass test. Since the loss value scales with number of categories we can compare the multiclass run, with the individual runs and the binary run, but factoring out the number of classes. In order to keep it comparable to the other runs, which were all binary, I will factor out only the multiclass run. This gives us a training baseline at 0.505 and a validation baseline at 0.496. The difference to the binary class and the averaged individual classes are smaller than 0.08 and 0.05 points for training and validation baselines, respectively. Thus, concluding from the loss curves alone the network could handle the multiclass test well.

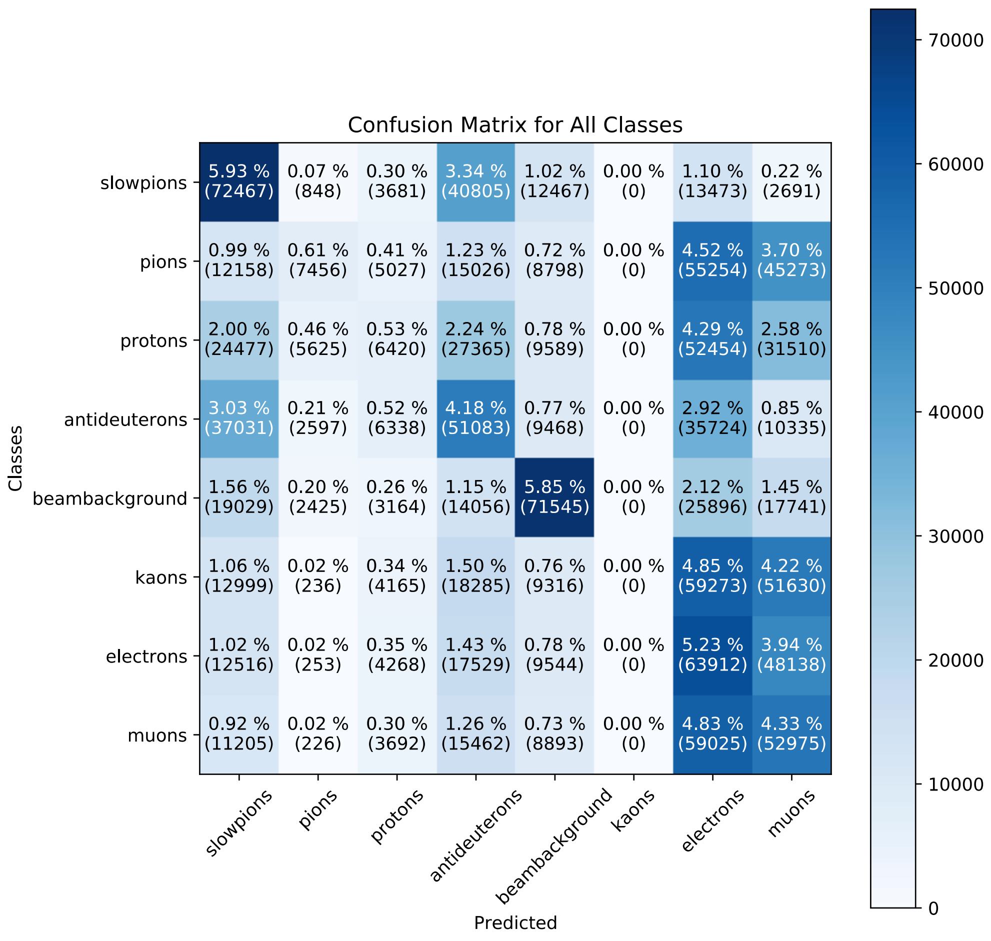


Figure 65: Confusion Matrix for Muliclass Test Run

Each category made up one eighth of the full data set, this means 12.5% should be the goal of in each category. We see in Figure 65 the confusion matrix for the multiclass test run. I am talking about the diagonal elements of the matrix. For Slow Pions there is a score of 5.93%, this is less than half of the ideal score. The MCC for Slow Pions is 0.33, which is indicates a weak correlation.

Protons and Pions are below 1% and their MCC, taken from Figure 70, show that the network was struggling and was more or less guessing randomly. Not many Pions or Protons were guessed to be Slow Pions.

Anti-Deuterons is at 4.18%, that is one third of what would have been perfect and the MCC is 0.18, indication a weak correlation. 3.34% of all Anti-Deuterons were guessed to be Slow Pions, this fits well with the results from the previous section for Anti-Deuterons

Beam Background is the second best at 5.85%, this is less than half of what it should be. Most wrongly labeled Beam Background events were assigned to be Slow Pions. It has the best MCC of 0.42.

The whole line for Kaons is just zero, this means, that noting was guessed to be a Kaons. This gave us the best accuracy at 87%, since most events are not Kaons, but the precision was the lowest at 0%. I cannot tell what happened here.

Electrons were guessed to be anything, but the fewest guessed to be Slow Pions. For Muons we have a similar picture, also here were the fewest guesses for Slow Pions. That is why their MCCs were close to zero, but it still fits in with the individual tests from the previous section.

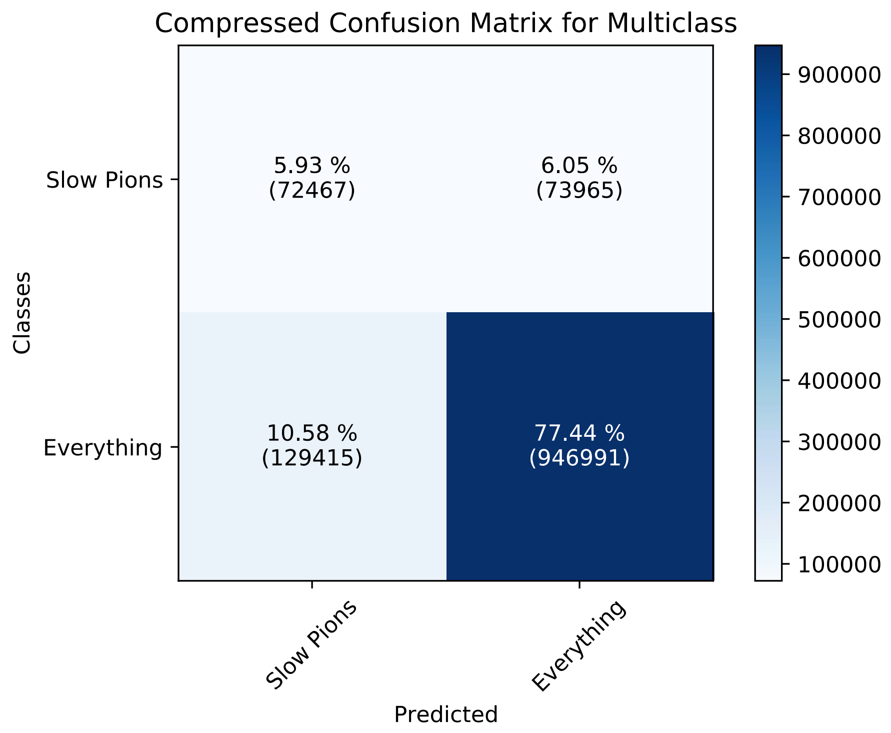


Figure 66: Compressed Confusion Matrix for Binary Test Run

Figure 66 shows a collapsed version of Figure 65. The upper, left corner remained; these are the Slow Pions. The left most column was summed up to a single cell, these are the missed Slow Pions. The upper most row was summed up; these are the mislabels for Slow Pions. The remaining block was summed up to one cell, this is the rest of the data set. We now can compare this to Figure 67. We see, that we have fewer mislabels and missed fewer Slow Pion events in the multiclass test run.

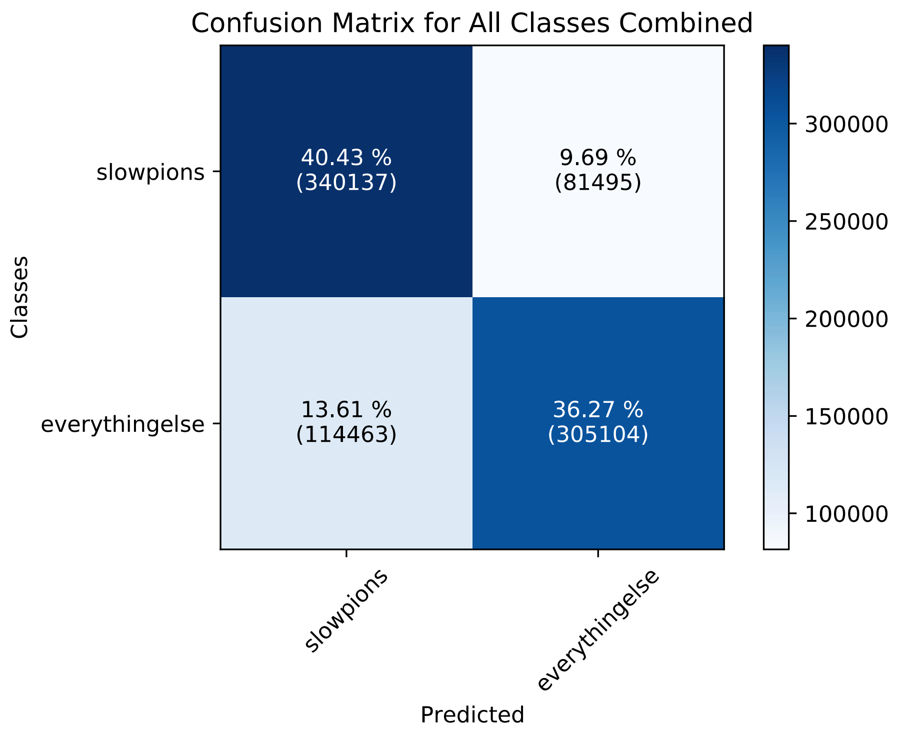


Figure 67: Confusion Matrix for Binary Test Run

I also wanted to compare the averaged confusion matrix of the individual runs, Figure 68, with the run of all categories combined into one, Figure 67. The first thing to notice is, that the amount of data for the individual runs are half of that for the run where all classes were combined. Apart from that, the scores are close and differ for about 1% or 2%, I attribute this small difference to the smaller data set.

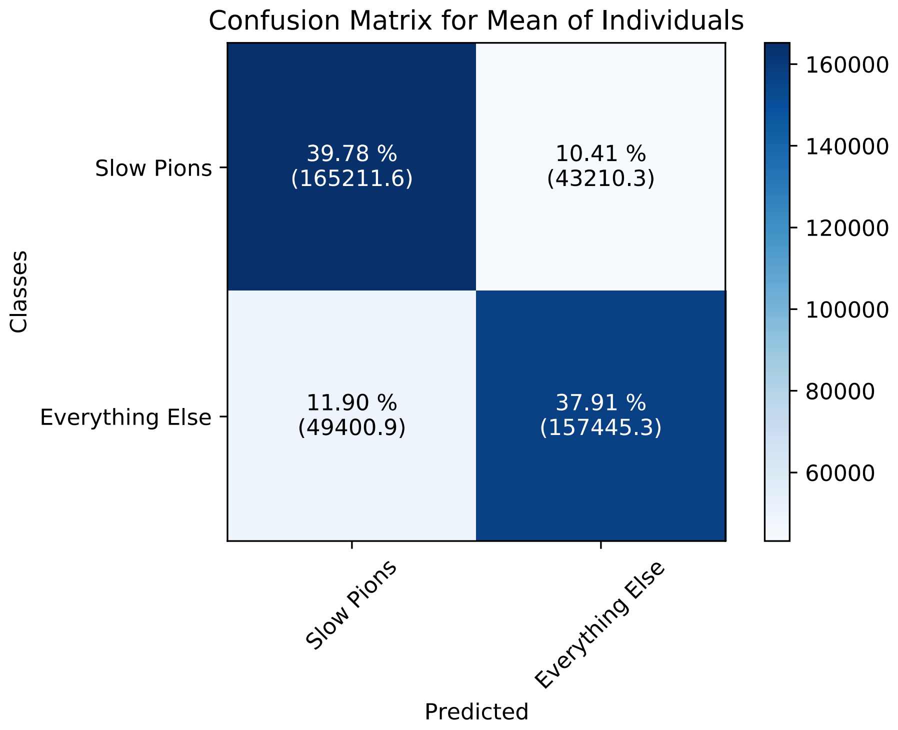


Figure 68: Average Confusion Matrix for Individual Particle Tests

We see all scores summarized in Table 25. I already talked about most of what can be deduced from it, besides I referenced to this table throughout this section many times.

Table 25: Summary for Multiclass and Binary Class Test Runs

	Multiclass	*Binary*	*Means*
Training Oscillation	0.002	0.002	0.002
Training Baseline	2.018	0.577	0.568
Validation Oscillation	0.006	0.006	0.005
Validation Baseline	1.982	0.537	0.525
Baseline Gap	0.036	0.040	0.042
Accuracy	0.834	0.767	0.777
Precision	0.359	0.748	0.770
MCC	0.328	0.536	0.554
Labeled Correctly	0.495	0.807	0.793
Mislabels	0.641	0.252	0.230

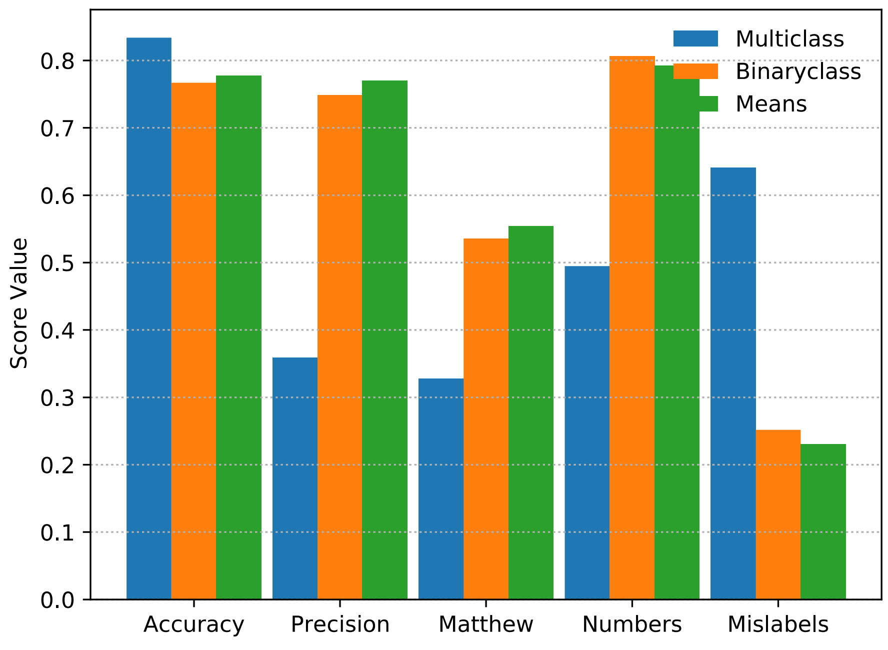


Figure 69: Score Summary for Multiclass and Binary Class Test Runs

Figure 69 shows a visual representation of the scores in regards to Slow Pions only. While accuracy is above 80% for the multiclass test, all other scores are far worse, the precision is below 40%, the MCC is below 0.4, not even half of all Slow Pions were found and more than 60% of what was labeled Slow Pion, was incorrectly labeled. The other two runs performed similarly and to the extent of what we expected.

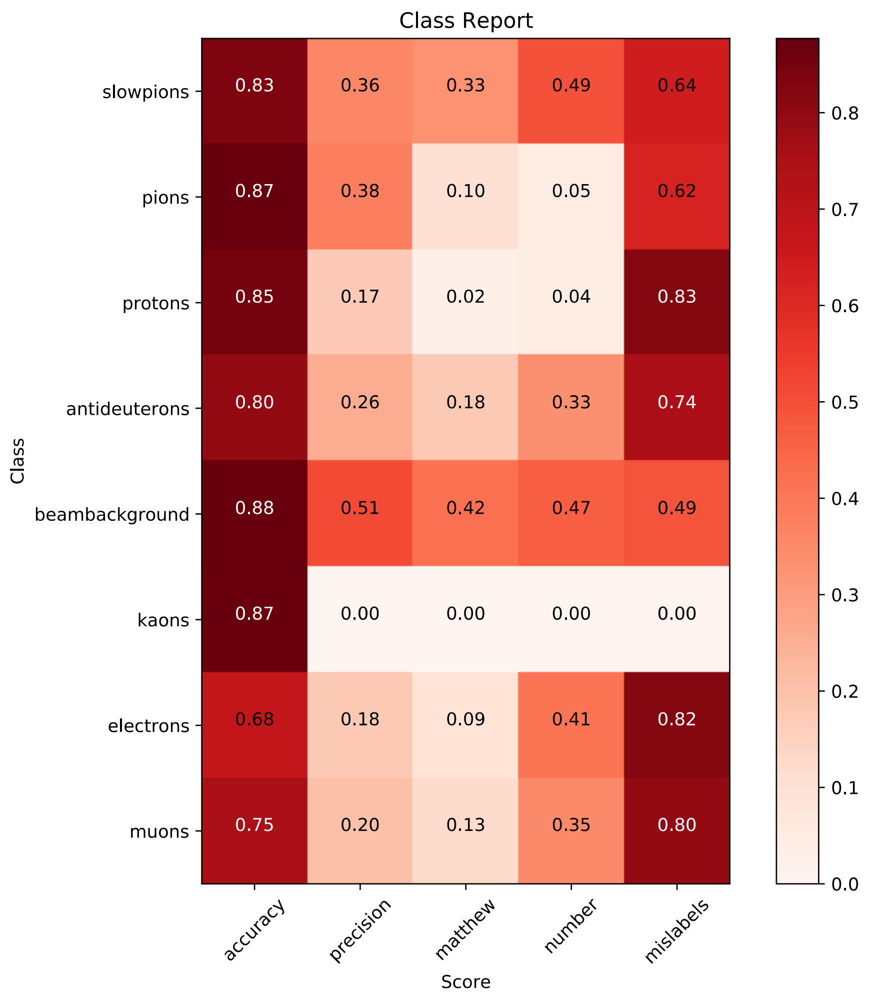


Figure 70: Score Report for Multiclass Test Run

Figure 70 shows all the scores from Figure 69 for every single category. The accuracy is in all cases strong, but the precision is plummeting on all cases. We also notice here how many events were wrongly categorized, especially Electrons, Muons and Protons. The conclusion for these tests has to be, that testing for individual particles and combined backgrounds makes a whole lot more sense and that one should fray away from training every category at once. It is much more helpful to train specifically for the particles one is trying find, than to do it all at once.

7.3.4 Tests against Larger Combinations

In this section I continue the network setup from the previous section. This means the Large Network will be run for 150 epochs. This time it is running on different data set configurations. I combined several particles together into what I call Heavy

Background, containing Protons, Anti-Deuterons, Kaons and Pions. The next combined data set is Kaons and Pions and the last is Electrons, Muons and Gammas, it is called Light Background. Since I am using balanced datasets, all runs will run with roughly thrice as many Slow Pions as compared to the single particle runs.

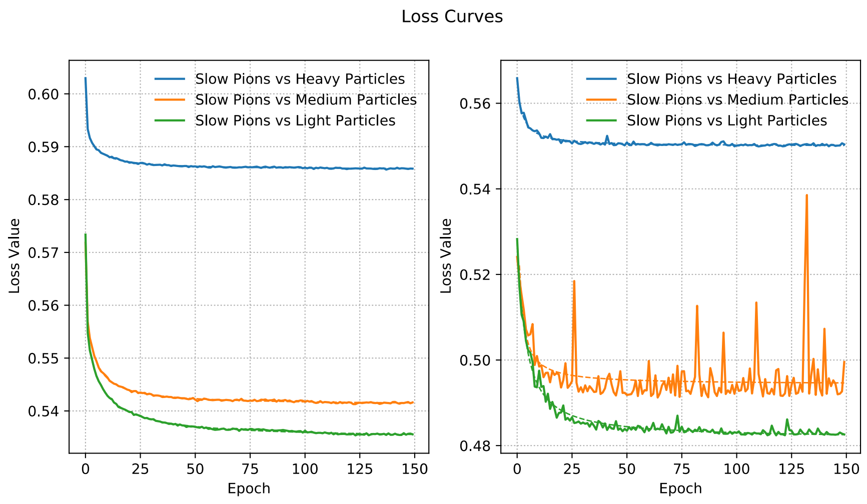


Figure 71: Loss Curves for Grouped Particles

Figure 71 shows what was already to be expected from the previous section, the scores are summarized in Table 26. The Heavy Background, performed weak, in the sense, that it has the highest baselines at 0.587 and 0.551 for training and validation loss, it has the smallest gap at 0.036. The heavy particles achieved a MCC of 0.499, which is just slightly smaller than the mean MCC for the individual particles at 0.505.

The Medium Background particles have their baselines at 0.544 and 0.497 with a gap of 0.047. The validation loss has the only oscillation value out of line with the others at 0.069. There are several peaks throughout the curve. Interestingly the individual particles have half the validation oscillation at 0.012 and 0.011. This run had a MCC of 0.620, which is nearly equal to the mean MCC of the individual particles at 0.628.

The Light Background particles, performed good, they have the lowest baselines at 0.539 and 0.488 with a gap of 0.051. This is the largest gap of the three runs. The validation oscillation is still small, but interestingly even larger than for the Heavy Background particles, at 0.012. The MCC is at 0.644, which is larger than the mean value for the individual particles, which is at 0.544. This score included the Gammas, if we exclude them, the mean MCC is 0.645, which is spot on. It makes sense to exclude Gammas, since their data set is significantly smaller than that for Electrons and Muons.

Table 26: Summary for Grouped Particles

	Heavy Background	*Medium Background*	*Light Background*
Training Oscillation	0.001	0.002	0.002
Training Baseline	0.587	0.544	0.539
Validation Oscillation	0.004	0.069	0.012
Validation Baseline	0.551	0.497	0.488
Baseline Gap	0.036	0.047	0.051
Accuracy	0.748	0.806	0.822
Precision	0.725	0.765	0.821
MCC	0.499	0.62	0.644
Labeled Correctly	0.82	0.883	0.83
Mislabels	0.275	0.235	0.179

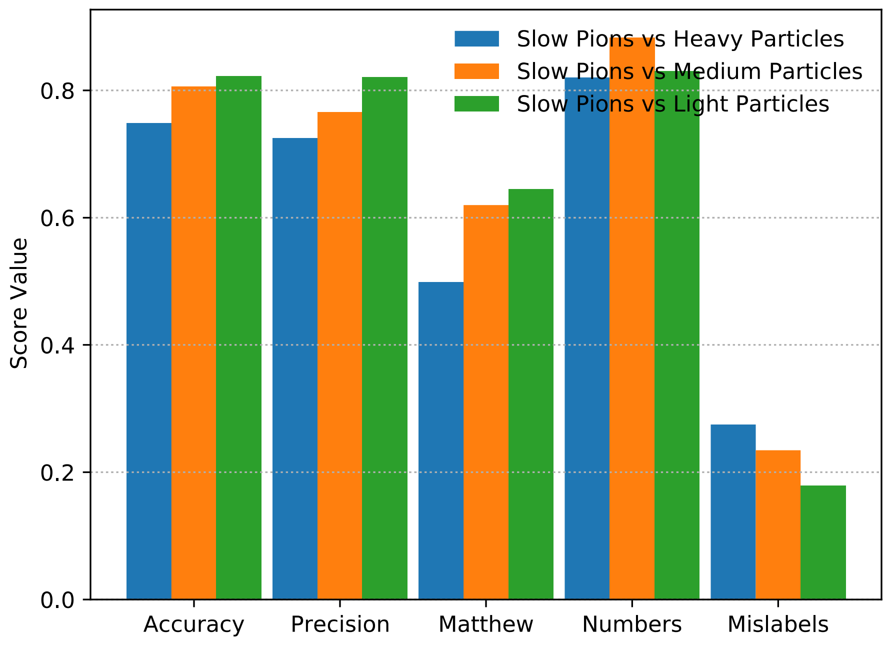


Figure 72: Summary for Grouped Particles

Figure 72 shows the visual summary of the scores for the three grouped runs. The heavy particles had the lowest accuracy and precision at only 74.8% and 72.5%, which is reflected in the fewest numbers of Slow Pions found and the largest mislabels at 82% and 27.5%. The mean value of Slow Pions from the last section is at 77.3%, which is quite a bit smaller. The mean value for mislabels is at 24.9%, which is also reduced.

The Medium Background particles have the second highest accuracy and precision at 80.6% and 76.5%. This run found the most Slow Pions out of these three and it had the second most mislabels at 88.3% and 23.5% respectively. The mean number of Slow Pions is at 82.8 %, again it is a bit less if we test against individual particles. The mean mislabels are at 19.3 % and here we are again at a larger value.

For the light data set I will exclude Gammas from the mean scores for the aforementioned reasons. The accuracy and precision were the highest in this run, at 82.2% and 82.1%. The number of Slow Pions found and the number of mislabels are at 83% and 17.9%. The mean values for these scores are 83.6% and 19%. The number of Slow Pions found is nearly identical and the number of mislabels is lower for the grouped particles.

The conclusion I draw from this is, that testing heavier particles in a grouped manner helps in finding Slow Pions against a larger background of other particles, but it also increases the number of mislabels. This holds true for the heavy and medium background. Lighter particles can be grouped, it even helps decreasing the number of mislabels.

7.3.5 The runs against Slow Electrons

In this section I want to test the Large Network for 150 epochs against so called Slow Electrons. In one run I want to employ all events and, in another run, I want to exclude all events, where only one pixel lights up in the nine-by-nine PXD images. This should help in lowering the ambiguity posed by single pixel events and thus lower mislabels. One should keep in mind, that this also lowers the number of Slow Pions, even if the percentage number is larger, because the overall number of events is lowered. This was shown in the section Simulated data.

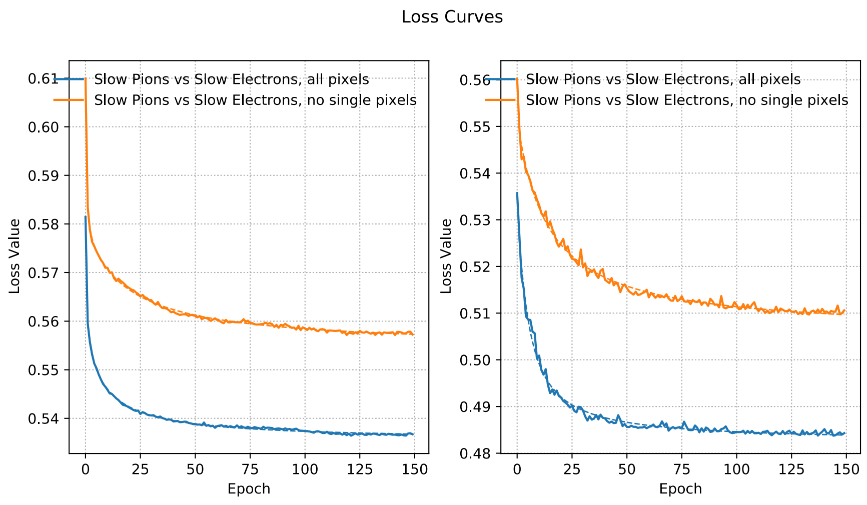


Figure 73: Loss Curves for Slow Electrons

Figure 73 shows us the loss curves for the Slow Pions against Slow Electrons. The scores are summarized in Table 27. The first thing one sees is that the baselines for the full data sets lay quite a bit lower, 0.541 and 0.491 as compared to 0.564 and 0.520 for the no single pixels run. The gaps are at 0.05 for the full data set and 0.044 for the no single pixels run. The oscillation scores are larger for the no single pixles run. All this shows, that it was learning slower, which can entirely be explained by the fact, that the data set is much smaller.

Table 27: Summary for Slow Electrons

	All Events	*No Single Pixel Events*
Training Oscillation	0.003	0.004
Training Baseline	0.541	0.564
Validation Oscillation	0.009	0.01
Validation Baseline	0.491	0.52
Baseline Gap	0.05	0.044
Accuracy	0.82	0.792
Precision	0.82	0.789
MCC	0.641	0.583
Labeled Correctly	0.82	0.796
Mislabels	0.18	0.211

Figure 74 shows the summary visualized. My interpretation from earlier, that the difference is just due to the size of the data sets, is further strengthened. The accuracies are at 82% and 79.2%, this is a gap of 2.8%. The precisions are at 82% and 78.9%, this is a gap of 3.1%. The number of Slow Pions found fell from 82% down to 79.6% by 2.4% and the mislabels rose by 3.1% from 18% up to 21.1%. The biggest changed was in MCC, the full data set had 0.641 and the no single pixels had 0.583. Overall, all scores changed by about 3% and all the charts lay pretty close.

A further comparison to ordinary Electrons shows, that the accuracies are the same, within a margin of error, at 82% for Slow Electrons and 81.8% for ordinary Electrons. The precisions were at 82% and 80.4%. There is not much in change for these data sets.

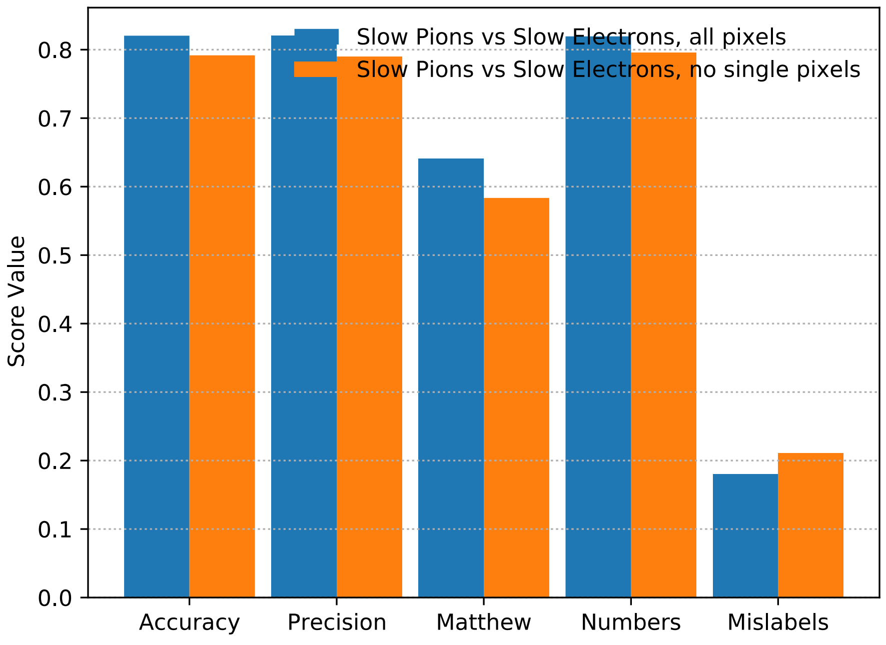


Figure 74: Summary for Slow Electrons

7.3.6 No Single Pixel Runs

Prompted by the results from the tests against Slow Electrons, I wanted to verify them by testing the Large Network with no single pixel events. I will keep the description here as short as possible, since a lot will repeat from earlier sections.

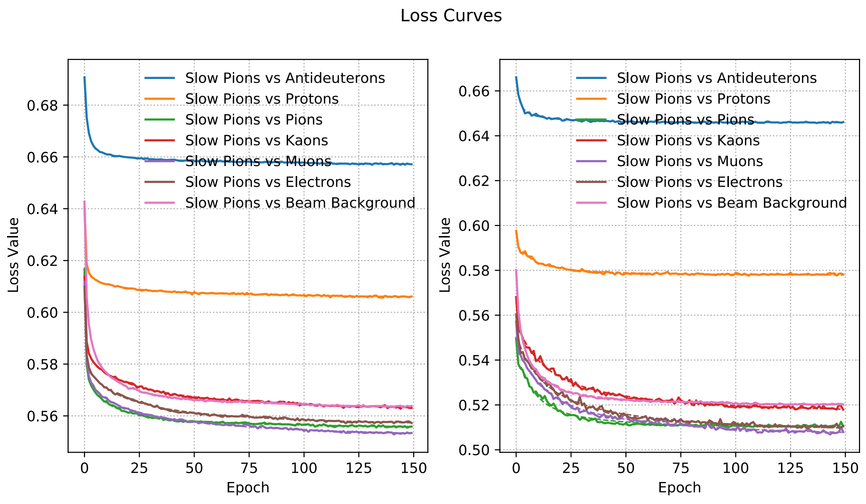


Figure 75: No Single Pixels against Individual Particles

The loss curve in Figure 75 look no different than the results for the full data sets. The scores are summarized in Table 28. The oscillation value for training and loss for all runs is less or equal to 0.01 this results in a mean value of 0.003 for training and 0.007 for validation. Both scores are a negligibly smaller than 0.005 and 0.010 for the full sets. Again, the highest baselines are for Anti-Deuterons, followed by Protons with other particles in these runs laying relatively close to each other. The gaps between baselines decreased, which indicates a better generalization, but this decrease is inconsequential. More interesting is, that the MCC fell from 0.554 down to 0.506, on a mean. Even though I removed every event, that could potentially be ambiguous, the correlation between correct guesses and labels weakened. Overall accuracy and precision fell each by 2.5% and 2.2% and with this 2.7% fewer Slow Pions were found and the mislabels increased for 2.2%. The only test, that profited from the exclusion of single pixel events was beam background, here 2.8% more Slow Pions were found. It is important to remember, that this does not mean, that in total more Slow Pions were found, but relative to the amount data used in these runs. Against beam background was also the lowest increase in mislabels of 0.9%. Protons were neutral in the decrease of Slow Pions found, but here the number of mislabels jumped the strongest with 4.5%. Anti-Deuterons had the biggest drop in Slow Pions found and was close to the mean increase in mislabels with 2.6%.

Table 28: Summary for No Single Pixels Run

	DD	*PP*	*PI*	*KK*	*MM*	*EL*	*BB*
Training Oscillation	0.002	0.003	0.003	0.004	0.004	0.004	0.003
Training Baseline	0.66	0.609	0.561	0.57	0.56	0.564	0.57
Validation Oscillation	0.003	0.005	0.009	0.009	0.009	0.01	0.004
Validation Baseline	0.647	0.58	0.516	0.528	0.518	0.52	0.527

Baseline Gap	0.012	0.028	0.045	0.042	0.042	0.044	0.044
Accuracy	0.613	0.714	0.792	0.784	0.795	0.792	0.779
Precision	0.626	0.692	0.79	0.783	0.796	0.789	0.76
MCC	0.227	0.43	0.584	0.567	0.589	0.583	0.56
Labeled Correctly	0.589	0.776	0.794	0.791	0.798	0.796	0.818
Mislabels	0.374	0.308	0.21	0.217	0.204	0.211	0.24

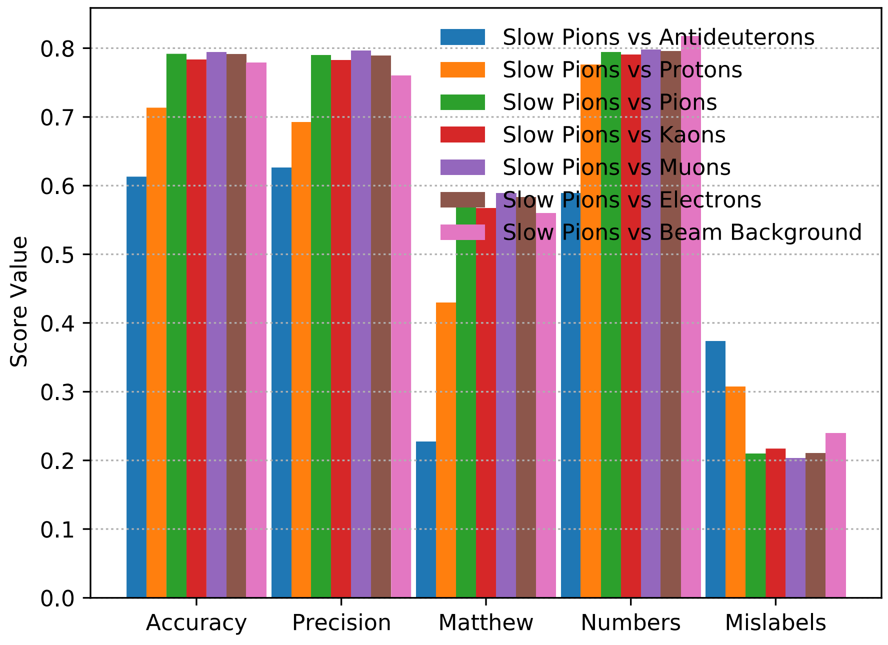


Figure 76: Summary for No Single Pixels Run

Figure 76 paints a very similar pictures to what Figure 62 already draw. These graphs make the worsening of the results more apparent. Anti-Deuterons and Protons fall further behind and the distinction between the heavier and lighter particles becomes stronger. The only improvement becomes clear as well, the distance between Beam Background and the lighter particles vanishes.

The general conclusion from this can only be, that excluding single particle events does not improve the quality of the categorization done by the network. This holds true especially for heavier particles such as Protons and Anti-Deuterons. The slight decrease in score can still be entirely due to the fact, that the training and validation data sets were smaller and thus the network had less for work with. This seems possible due to the fact, that the decreases are only around 2%. In this sense we got similar results as we got with no single pixels for Slow Electrons.

7.3.7 One Last Test Run

Throughout my analysis I alluded to this last experiment, where I use a linear network with 7, 5 and 3 layers and the AdaHessian optimizer and I ran it for 200 epochs. The reasoning for these test runs was, that AdaHessian showed a lot of promise very early on, without tinkering. I wanted to test it on a simpler network, in order to keep the time requirements short.

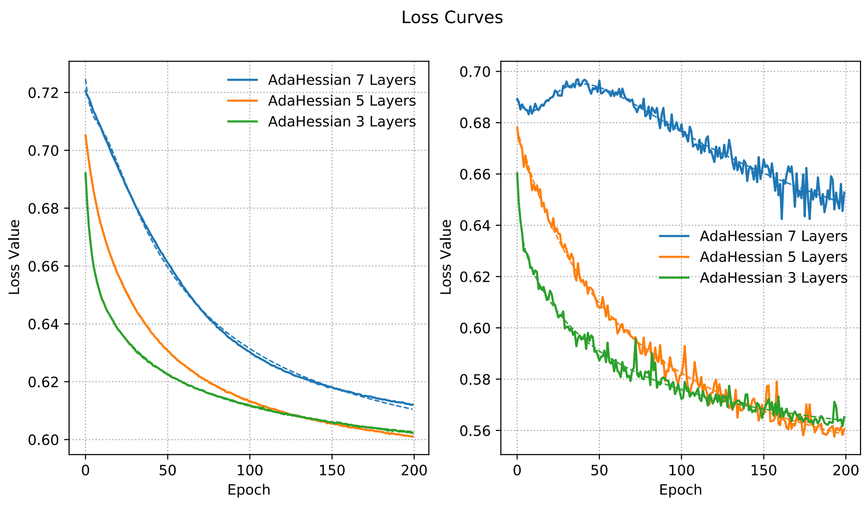


Figure 77: Loss Curves for AdaHessian

Figure 77 shows the loss curves for the 200 epoch test runs with AdaHessian optimizer. The baselines for the 7-layer setup are at 0.659 and 0.683, with a gap of 0.024, which is smaller than the gaps for the Small, Medium and Large Network. Their gaps are all larger than 0.034 for 200 epochs. Still the baselines were lower. The validation oscillation values are at 0.045, 0.043 and 0.035. This is around seven to nine times larger, than 0.005 for the Large Network for 200 epochs.

Table 29: Summary for AdaHessian Test Runs

	7 Layers	*5 Layers*	*3 Layers*
Training Oscillation	0.015	0.003	0.004
Training Baseline	0.659	0.634	0.625
Validation Oscillation	0.045	0.043	0.035
Validation Baseline	0.683	0.609	0.592
Baseline Gap	0.024	0.025	0.033
Accuracy	0.611	0.756	0.758
Precision	0.773	0.745	0.729
MCC	0.276	0.513	0.519
Labeled Correctly	0.315	0.776	0.809
Mislabels	0.227	0.255	0.271

Figure 78 and Table 29 are the summaries for these three runs. Seven layers only achieved an accuracy of 61.1%, which quite a bit lower than 75% for the five- and three-layer setups. The precisions are close at 77.3%, 74.5% and 72.9% for the seven-, five- and three-layer setup.

The three- and five-layer setup performed comparably, they have both a MCC of 0.51, which is spot-on with the long-term tests for all three networks. Both AdaHessian setups found more than 77%, the smaller network found more than 80%, but also had the most mislabels. The seven-layer network had the weakest MCC at 0.276, the MCC for the Small, Medium and Large Network for 200 epochs all were above 0.5, nearly twice as large as this. It found the least number of Slow Pions at 31.5% and the fewest mislabels at 22.7%. There is a large gap in Slow Pions correctly labeled of at least 46.1% and the small gap in mislabels of 2.8%.

Taking this all together, the seven-layer network did not perform well. The five- and three-layer performed similarly, but they did not outperform the three networks from the long-term test, while running longer. The only argument speaking for AdaHessian, would be this. One trains the Large Network with AdaHessian for 200, 300, 400 epochs and so on, of course, after tweaking the learning rate. Then one has a well-trained network. The optimizer is only involved in training and cannot hinder the evaluation performance. Together with the fact, that one has to train only once, it might be worth using AdaHessian going forward.

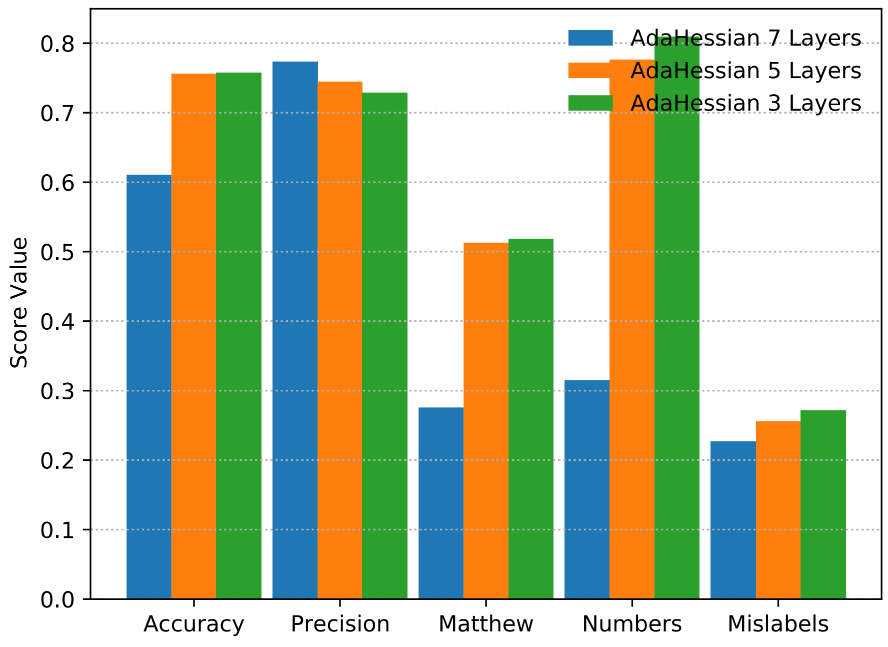


Figure 78: Summary for AdaHessian Test Runs

8 Summary & Concusions

Hooray! A Happy Ending For The Rich People.

Dr. John A. Zoidberg

8.1 What did we achieve?

Throughout this work we saw relative similar performances by larger and smaller network setups, with an accuracy of ranging from 76% for the Small Network and small increase for the Medium Network up to nearly 77% for the Large Network. The precision stayed around 75% for the Large Network, it was a bit lower for the Medium Network and it was around 74% for the Small Network. We saw, that smaller networks require less training, otherwise they will lose generalization. Bigger networks could increase their capabilities given longer training, with the catch, that with more time the number of mislabels also increased.

Judging from the loss curves, the Large and Medium Network could generalize better than the Small Network. This was indicated by the smaller baseline gaps for the larger networks and the higher oscillations for the smaller networks.

When I tested Slow Pions individually against every particle, we saw, that Anti-Deuterons and Protons were revealed to be very challenging. They had the lowest accuracy of all particles at 64% and 74%, while all others were above 80%, with the exception of Beam Background at 78%, which can be explained by the smaller size of the data set. Anti-Deuterons and Protons also produced the lowest precision of 65% and 74%, where all others were above 80%, except for Beam Background, which was at 77%.

These scores were all confirmed by two further tests, one where I combined the smaller sets into three larger sets and one where I excluded single pixel events. The single pixel events were about 2% to 3% worse in every score, as compared to the full sets. This can entirely be due to the smaller size of the sets. The combined sets achieved accuracies of 75%, 80% and 82%, the averaged accuracies for the individual sets corresponding to the combined sets are 75%, 81% and 81%. These scores are matching within a margin of error. The same holds true for precision, with 75%, 77% and 82% for the Heavy, Medium and Light Backgrounds and the respective averaged precisions of 75%, 81% and 80%. Only the Medium Background had a bit worse precision if tested in a combined manner.

The Large Network, with which these tests were done, achieved an accuracy of 77% and a precision of 75% for 150 epochs. The averaged accuracy for the individual tests is 79% and the precision is at 79%. Here running each particle individually slightly improved the scores.

In a realistic setting it is not possible to tests each set individually. If it were, then we would already have the sets separated out and there would be no need for using a neural network in order to sort the particles out.

In conclusion I can say, that the Large Network for 150 epochs performed well and if it would be possible to separate Anti-Deuterons and Protons out before hand, then this network could achieve an accuracy of above 80% with a precision near 80%. Guessing from the individual tests, most mislabels came from Anti-Deuterons and Protons.

I tested Slow Pions against Slow Electrons in much the same manner as did Erwin Do[38]. Unlike him, I only tested full data sets and left out single pixel events, he did more extensive tests here. My Network achieved an accuracy and precision of 82%, excluding single pixel events decreased this number by the familiar 3%. The tests against ordinary Electrons got scores of 82% for accuracy and 80% for precision, thus there was no performance impact for Slow Electrons vs. Slow Pions as compared to Electrons vs. Slow Pions.

The goal of this was it to employ a neural network in order to find Slow Pions within a larger background of other particles and beam-background. This was achieved to the extent, that a simpler network could find the majority of Slow Pions with a high precision of 80%, while keeping training time low. Thus, also validation can be done swiftly, even for enormous data sets and on weaker hardware.

[38] Ludwig-Maximilians-Universität München, 22. June 2020

8.2 What needs to be done?

While I conducted many tests, there is still a lot of room for experimentation. One could do more tests with different dropout rates for each layer, test deeper networks and test narrower layers. Furthermore, I only made a small survey of learning rate scheduler and it might be worthwhile to try more settings and tune the learning rate better. The same goes for activation functions, but to a lesser extent, since they have fewer parameters. Another shortcoming of my network was, that I only employed squared kernel sizes and it might improve the networks performance if it would look for vertical and horizontal patterns. Testing more setups with more layer combinations of convolutional and transposed convolutional layers might also lead to better pattern recognition and increase the networks capabilities.

As was indicated by several of my tests in this work, I do not think that even bigger networks will improve much of the capabilities of differentiating between data sets, especially if we are only looking at one category against everything else. In this case going larger even decreased the performance. Still, it can be of interest to test bigger networks on multilabel tests, since maybe more parameters will enable the network to adapt to more characteristics within the data sets. This again will require more tweaking of all hyper parameters, especially learning rate and epochs.

It could give some insights into what is causing the most mislabels and against which data set the most Slow Pions, if the individual labels for each particle would be retained, even in the case where they are combined to a bigger background. Then one could check the mislabels, from which category they came. So far, I can only make inferences from the individual tests.

In terms of analysis and visualization, there is also all lot more to do. One could compare loss curves not just for each set of runs, like I did, but also more cross comparisons. For example, plotting the loss curves for Small, Medium and Large Network for epochs and not network sizes or comparing individual particles with their respective combined backgrounds. I tried to facilitate these kinds of comparisons by employing what is called oscillation and baselines, I also did not want to exhaust the reader too much, with too many graphs and plots. Additionally, one should look at the

convergences of each of the loss curves fits I made. This might give further indications, which setups will achieve given enough time, without actually running for that amount of time, or at least one can tell for how long it would be wise to train.
I tried linear, convolutional and transposed convolutional layers in my network. There are more kinds of layers, as I already mentioned throughout this work. There are Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, which both belong to the same class and are mostly applied in cases of sequential data, like speech and text. This is not strictly the case with PXD data, which are independent events. The next type of network one could test are Graph Neural Networks (GNNs), which can find relations between objects and events and organize them into graphs (105), which is also not the case for PXD data.
Besides implementing more features in the code, I would restructure it. Training and validation should be done separately, the same holds for plotting and analyzing the output. This should be done with different files, instead of all on just one big file. The output should be more diversified. While I could do all my plots from the log files, it would be tremendously helpful to write out files that can be plotted without any further manipulation.
If PXD data could be processed in sequences and not like independent events, RNNs and LSTMs could yield improvements and maybe within the PXD data are hidden metadata containing graph like relations, which then would make GNNs sensible. This might be the case if one included the event coordinates, with were mentioned in the section Simulated data. This is just speculation in my part.
Lastly, neural networks performance is strongly dependent on the representation of the data. This means, one should preprocess the data in order to find a more optimized representation and eventually find better results.

9 Bibliography

The world is not in your books and maps. It's out there.

Olórin

1. *Recent Anomalies in B Physics.* **Ying Li, Cai-Dian Lü.** s.l. : arXiv, 2018.

2. *Lattice QCD inputs to the CKM unitary triangle analysis.* **Laiho, and al, .** s.l. : Physical Review D, 2010, Vol. 81.

3. *The Matter-Antimatter Asymmetry Problem.* **Robson, Brian Alber.** s.l. : Journal of High Energy Physics, Gravitation and Cosmology, 2018, Vol. 4.

4. *A proposal for B-physics on current lattices.* **Blossier, , et al.** 4, s.l. : Journal of High Energy Physics, 2010, Vol. 49.

5. *Charming new B-physics.* **Jäger, , et al.** 3, s.l. : Journal of High Energy Physics, 2020.

6. *Belle B physics results.* **Collaboration, and Tajima, .** 22, s.l. : International Journal of Modern Physics A , 2002, Vol. 17.

7. *Online-analysis of hits in the Belle-II pixeldetector for separation of slow pions from background.* **Bähr, and al, .** 9, s.l. : Journal of Physics: Conference Series, 2015, Vol. 664.

8. *Impact of tag-side interference on time-dependent CP asymmetry measurements using coherent B0-B0bar paris.* **Long, , et al.** s.l. : Physical Review D, 2003, Vol. 68.

9. **Isgur, and Wise, Mark B.** elationship between form factors in semileptonic B and D decays and exclusive rare B-meson decays. *Physical Review D.* 1990.

10. **Bevan, and al, .** *The physics of the B factories.* s.l. : Springer Nature, 2017.

11. *Improved measurement of CP-violation parameters sin 2 φ 1 and| λ|, B meson lifetimes, and B 0-B¯ 0 mixing parameter Δ m d.* **Abe, and al, .** 7, s.l. : Physical Review D, 2005, Vol. 71.

12. **Münchow, .** *Development of the Online Data Reduction System and Feasibility Studies of 6-Layer Tracking for the Belle II Pixel Detector.* Gießen : Doctoral dissertation, 2015.

13. *Lifetime differences, direct CP violation, and partial widths in D 0 meson decays to K+ K− and π+ π−.* **Csorna, S. E. and al, .** 9, s.l. : Physical Review D, 2002, Vol. 65.

14. **Univeristät Mainz.** First particles circulate in SuperKEKB accelerator. *Phys.org.* [Online] 14 April 2016. https://phys.org/news/2016-04-particles-circulate-superkekb.html.

15. **Ohnishi, , et al.** Accelerator design at SuperKEKB. *Progress of Theoretical and Experimental Physics.* 2013.

16. **Ohnishi, .** Report on SuperKEKB phase 2 commissioning. *Proc. IPAC'18.* 2018.

17. *SuperKEKB collider.* **Akai, , Furukawa, and Koiso, .** s.l. : Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 2018, Vol. 907.

18. **KEK.** Electrons and Positrons Collide for the first time in the SuperKEKB Accelerator. *KEK.* [Online] 26 April 2018. https://www.kek.jp/en/newsroom/2018/04/26/0700/.

19. **Kinoshita, .** Reflections on beauty. *New Astronomy Reviews* . 1998.

20. **Irmler, , et al.** Origami chip-on-sensor design: progress and new developments. *Journal of Instrumentation.* 2013. Vol. 8.

21. *The Belle II pixel vertex detector.* **Furletov, .** Zurich : IOP Publishing, 2011.

22. **Paschen, , et al.** Belle II pixel detector: Performance of final DEPFET modules. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment.* 2020.

23. **Andricek, , et al.** Initial requirements for an upgraded VXD in Belle II. *Belle II.* 2019.

24. **Lang, Christian B. and Pucker, .** Mathematische Methoden in der Physik. s.l. : Springer Spektrum, 2005.

25. **Wikipedia.** Tensor. *Wikipedia.* [Online] 20 April 2021. https://de.wikipedia.org/wiki/Tensor.

26. **Kerner, and Wahl, .** Mathematik für Physiker. s.l. : Springer Berlin Heidelberg, 2013.

27. **Chollet, .** *Deep learning with Python.* New York : Manning, 2018.

28. **Goodfellow, and Courville, Yoshua Bengio and Aaron.** *Deep Learning.* Massachusetts : MIT press, 2017.

29. *Graph neural networks in particle physics.* **Shlomi, , Battaglia, and Vlimant, .** s.l. : Machine Learning: Science and Technology, 2020.

30. *Machine Learning in High Energy Physics Community White Paper.* **Albertsson, , et al.** 2018, Journal of Physics: Conference Series.

31. *Neural networks in high energy physics: a ten year perspective.* **Denby, .** s.l. : Computer Physics Communications, 1999.

32. **Trask, Andrew W.** *Deep Learning.* 2019.

33. **Christian, .** *The Alignment Problem: Machine Learning and Human Values.* s.l. : WW Norton & Company, 2020.

34. *Application of artificial neural networks in particle physics.* **Kolanoski, .** Heidelberg : International Conference on Artificial Neural Networks, Vol. 1996.

35. **Shanahan, .** The Frame Problem. [Online] 23 February 2004. [Cited: 14 April 2021.] https://plato.stanford.edu/entries/frame-problem/.

36. **Vervaeke, , Lillicrap, Timothy P. and Richards, Blake A.** Relevance realization and the emerging framework in cognitive science. *Journal of Logic and Computation.* 2012.

37. **Burkov, .** *The hundred-page machine learning book.* 2019.

38. *Reconstruction of porous media from extremely limited information using conditional generative adversarial networks.* **Feng, , et al.** 2019, Physical Review E.

39. **Torch Contributors.** TORCH.NN. *PyTorch.* [Online] 2019. https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity.

40. **Heaton, .** *Introduction to the Math of Neural Networks.* s.l. : Heaton Research Inc., 2012.

41. **Torch Contributors.** NLLLOSS. *PyTorch.* [Online] 2019. https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html#torch.nn.NLLLoss.

42. **Doshi, .** Various Optimization Algorithms For Training Neural Network. [Online] 13 January 2019. [Cited: 20 April 2021.] https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6.

43. **Ruder, .** An overview of gradient descent optimization algorithms. [Online] 19 January 2016. [Cited: 20 April 2021.] https://ruder.io/optimizing-gradient-descent/.

44. **Price, , et al.** Stochastic gradient descent. [Online] 21 December 2020. [Cited: 20 April 2021.] https://optimization.cbe.cornell.edu/index.php?title=Stochastic_gradient_descent.

45. **Kincaid, .** Adam. [Online] 21 December 2020. [Cited: 20 April 2021.] https://optimization.cbe.cornell.edu/index.php?title=Adam.

46. **Huang, .** RMSProp. [Online] 21 December 2020. [Cited: 20 April 2021.] https://optimization.cbe.cornell.edu/index.php?title=RMSProp.

47. **Heaton, .** *Introduction to neural networks with Java.* s.l. : Heaton Research Inc., 2008.

48. **Kumar, .** Overview of various Optimizers in Neural Networks. [Online] 9 June 2020. [Cited: 17 April 2021.] https://towardsdatascience.com/overview-of-various-optimizers-in-neural-networks-17c1be2df6d5.

49. *ADAHESSIAN: An adaptive second order optimizer for machine learning.* **Yao, and al, .** s.l. : arXiv preprint , 2020.

50. **Dertat, .** Applied Deep Learning - Part 4: Convolutional Neural Networks. [Online] 8 November 2017. [Cited: 18 April 2021.] https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2.

51. **IBM Cloud Education.** Convolutional Neural Networks. [Online] 20 October 2020. [Cited: 20 April 2021.] https://www.ibm.com/cloud/learn/convolutional-neural-networks.

52. **Skalski, .** Gentle Dive into Math Behind Convolutional Neural Networks. [Online] 12 April 2019. [Cited: 20 April 2021.] https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9.

53. **Zhang, , et al.** Transposed Convolution. [Online] 23 April 2021. [Cited: 4 May 2021.] http://d2l.ai/chapter_computer-vision/transposed-conv.html.

54. **Anwar, .** What is Transposed Convolutional Layer? [Online] 6 March 2020. [Cited: 4 May 2021.] https://towardsdatascience.com/what-is-transposed-convolutional-layer-40e5e6e31c11.

55. **Shibuya, .** Up-sampling with Transposed Convolution. [Online] 13 November 2017. [Cited: 4 May 2021.] https://naokishibuya.medium.com/up-sampling-with-transposed-convolution-9ae4f2df52d0.

56. **Fredenslund, .** Introduction To The Basics Of Neural Networks - How Does Backpropagation Work? [Online] [Cited: 20 April 2021.] https://kasperfred.com/series/introduction-to-neural-networks/how-does-backpropagation-work.

57. **Kostadinov, .** Understanding Backpropagation Algorithm. [Online] 8 August 2019. [Cited: 20 April 2021.] https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd.

58. **Yadav, .** Weight Initialization Techniques in Neural Networks. [Online] 9 November 2018. [Cited: 18 April 2021.] https://towardsdatascience.com/weight-initialization-techniques-in-neural-networks-26c649eb3b78.

59. *Understanding the difficulty of training deep feedforward neural networks.* **Glorot, and Bengio, .** s.l. : JMLR Workshop and Conference Proceedings, 2010. Proceedings of the thirteenth international conference on artificial intelligence and statistics . pp. 249-256.

60. **Venners, .** The Making of Python A Conversation with Guido van Rossum. *Artima.* [Online] 13 January 2003. [Cited: 30 March 2021.] https://www.artima.com/articles/the-making-of-python.

61. **Pramanick, .** History of Python. *Geeks for Geeks.* [Online] 6 May 2019. [Cited: 30 March 2021.] https://www.geeksforgeeks.org/history-of-python/.

62. **Stuff, .** A Brief history of the Python Programming Language. *DigitalAdBlog.* [Online] 4 March 2021. [Cited: 30 March 2021.] https://digitaladblog.com/2021/03/04/a-brief-history-of-the-python-programming-language/.

63. **Synced.** Caffe2 Merges With PyTorch. *Medium.* [Online] 2 April 2018. [Cited: 30 March 2021.] https://medium.com/@Synced/caffe2-merges-with-pytorch-a89c70ad9eb7.

64. **Shetty, .** What is PyTorch and how does it work? *Packt.* [Online] 18 September 2018. [Cited: 30 March 2021.] https://hub.packtpub.com/what-is-pytorch-and-how-does-it-work/.

65. **Howard, .** *Deep Learning Frameworks - TensorFlow, PyTorch, fast.ai.* [interv.] Lex Fridman. 6 October 2019.

66. **Dancuk, .** PyTorch vs TensorFlow: In-Depth Comparison. [Online] 23 February 2021. [Cited: 11 May 2021.] https://phoenixnap.com/blog/pytorch-vs-tensorflow.

67. **Boesch, .** Pytorch vs Tensorflow: A Head-to-Head Comparison. [Online] 2 March 2021. [Cited: 11 May 2021.] https://viso.ai/deep-learning/pytorch-vs-tensorflow/.

68. **Nvidia Deep Learning Frameworks Documentation. *PyTorch Release Notes.* [Online] [Cited: 30 March 2021.] https://docs.nvidia.com/deeplearning/frameworks/pytorch-release-notes/overview.html#overview.**

69. *Classification assessment methods." Applied Computing and Informatics.* Tharwat, . 2020.

70. *An introduction to ROC analysis.* Fawcett, . s.l. : Pattern recognition letters, 2006.

71. Burkov, . *The HundredPage Machine Learning Book.* 2019.

72. *Generalized Confusion Matrix for Multiple Classes.* Manliguez, . 2016, www.researchgate.net.

73. Krüger, . Activity, Context, and Plan Recognition with Computational Causal Behaviour Models. 2016.

74. *Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation.* Powers, David MW. 2020.

75. *The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation.* Chicco, and Jurman, . s.l. : BMC genomics, 2020.

76. OpenStax. The Standard Model. *Physics LibreTexts.* [Online] 5 November 2020. https://phys.libretexts.org/Bookshelves/University_Physics/Book%3A_University_Physics_(OpenStax)/Book%3A_University_Physics_III_-_Optics_and_Modern_Physics_(OpenStax)/11%3A_Particle_Physics_and_Cosmology/11.06%3A_The_Standard_Model.

77. Boyle, Peter A. *Standard Model.* Edinburgh : s.n., 2014.

78. Wolchover, . What No New Particles Means for Physics. [Online] 9 August 2019. [Cited: 21 April 2021.] https://www.quantamagazine.org/what-no-new-particles-means-for-physics-20160809.

79. Povh, Bogdan, et al. *eilchen und Kerne: eine Einführung in die physikalischen Konzepte.* s.l. : Springer-Verlag, 2013.

80. LEIFIphysik. [Online] [Cited: 9 April 2021.] https://www.leifiphysik.de/kern-teilchenphysik/teilchenphysik/grundwissen/die-vier-fundamentalen-wechselwirkungen.

81. Wikipedia. Fundamental interaction. [Online] [Cited: 9 April 2021.] https://en.wikipedia.org/wiki/Fundamental_interaction.

82. Ramond, . *Journeys Beyond the Standard Model.* Cambridge : Perseus Books, 1999.

83. Tuning, . *Lecture Notes on CP Violation.* 2020.

84. Gronau, . *A Variety of CP Violating B Decays.* Oxford : s.n., 1995.

85. Pich, . *CP Violation.* Geneva : s.n., 93.

86. The Editors of Encyclopaedia Britannica. Gauge theory . [Online] [Cited: 26 April 2021.] https://www.britannica.com/science/gauge-theory.

87. Hooft, Gerard 't. Gauge theories. [Online] 2008. [Cited: 26 April 2021.] https://en.wikipedia.org/wiki/Gauge_theory.

88. Encyclopedia of Mathematics. Gauge transformation. [Online] [Cited: 26 April 2021.] https://encyclopediaofmath.org/index.php?title=Gauge_transformation.

89. *The Standard Model of Particle Physics.* Romanino, . Trieste : INFN, SISSA/ISAS, 2009, nternational Baikal Summer School on Physics of Elementary Particles and Astrophysics.

90. Nave, . Left-Handed Neutrinos. [Online] [Cited: 26 April 2021.] http://hyperphysics.phy-astr.gsu.edu/hbase/Particles/neutrino3.html.

91. *CP violation in the B system.* Gershon, T., and V. V. Gligorov. s.l. : Reports on Progress in Physics, 2017.

92. *Rare kaon and pion decays: incisive probes for new physics beyond the standard model.* Bryman, and al, . s.l. : Annual Review of Nuclear and Particle Science, 2011, Vol. 61.

93. Lange, . Vorlesung Höhere Teilchenphysik. Gießen : s.n., 2019.

94. Demtröder, . *Experimentalphysik 4.* Heidelberg : Springer Berlin, 2010.

95. Mazur, . Unitary triangle. *symmetry - dimensions of particle physics.* [Online] 1 January 2006. [Cited: 31 March 2021.] https://www.symmetrymagazine.org/article/december-2005january-2006/deconstruction-unitary-triangle.

96. *The CKM matrix and the unitarity triangle.* Battaglia, and al, . s.l. : arXiv preprint hep-ph/0304132, 2003.

97. *Relating CKM Parametrizations and Unitarity Triangles.* Lebed, Richard F. s.l. : Physical Review D, 1997, Vol. 55.

98. Gershon, . A triangle that matters. *Physics World.* [Online] IOP Publishing, 2 April 2007. [Cited: 29 March 2021.] https://physicsworld.com/a/a-triangle-that-matters/.

99. ***B-physics anomalies: a guide to combined explanations.*** **Buttazzo, Dario, Admir Greljo, Isidori, and Marzocca, . 11, s.l. : Journal of High Energy Physics, 2017, Vol. 44.**

100. ***A new inclusive secondary vertex algorithm for b-jet tagging in ATLAS.*** **Piacquadio, and Weiser, . 3, s.l. : Journal of Physics: Conference Series, 2008, Vol. 119.**

101. ***How to determine all the angles of the unitarity triangle from B0d → DKs and B0s → DΦ.*** **Gronau, . 3,4, s.l. : Physics Letters B, 1991, Vol. 253.**

102. ***How many hidden layers and nodes?*** **Stathakis, . s.l. : International Journal of Remote Sensing, 2009.**

103. Brownlee, . How to use Learning Curves to Diagnose Machine Learning Model Performance. [Online] 27 February 2019. [Cited: 5 June 2021.] https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/.

104. Google Developers. Interpreting Loss Curves. [Online] [Cited: 5 June 2021.] https://developers.google.com/machine-learning/testing-debugging/metrics/interpretic.

105. Menzli, . Graph Neural Network and Some of GNN Applications – Everything You Need to Know. [Online] 9 April 2021. [Cited: 21 May 2021.] https://neptune.ai/blog/graph-neural-network-and-some-of-gnn-applications.

106. ***he Belle II silicon vertex detector: Assembly and initial results.*** **Thalmeier, and a, . s.l. : Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 2019, Vol. 936.**

107. Silicon Vertex Detector. ***Belle II Italian collaboration.*** **[Online] [Cited: 30 March 2021.] https://web.infn.it/Belle-II/index.php/detector/svd.**

108. New electronics tested for Belle II central drift chamber. [Online] 23 March 2010. [Cited: 30 March 2021.] https://www2.kek.jp/proffice/archives/feature/2010/BelleIICDCDesign.html.

109. ***Three-dimensional Fast Track for the Central Drift Chamber Based Level-1 Trigger System in the Belle II Experiment.*** **Won, and Kim, J. B. 1, s.l. : Journal of the Korean Physical Society, 2018, Vol. 72.**

110. TOP detector. ***Belle II Italian collaboration.*** **[Online] [Cited: 30 March 2021.] https://web.infn.it/Belle-II/index.php/detector/top/22-top-detector.**

111. Electromagnetic Calorimeter. ***Belle II Italian collaboration.*** **[Online] [Cited: 30 March 2021.] https://web.infn.it/Belle-II/index.php/detector/ecl.**

112. KLM detector. ***Belle II Italian collaboration.*** **[Online] [Cited: 30 March 2021.] https://web.infn.it/Belle-II/index.php/detector/klm.**

113. *Observation of Exclusive Decay Modes of b-Flavored Mesons.* Behrends, and al, . 12, s.l. : Physical Review Letters, 1983, Vol. 50.

114. *Neutral B flavor tagging for the measurement of mixing-induced CP violation at Belle.* Kakuno, and al, . 3, s.l. : Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, 2004, Vol. 533.

115. *b Tagging in Atlas and CMS.* Scodellaro, . s.l. : CMS, 2017, Vol. 225.

116. Lellouch, , Lin, C-J. David and Collaboration, . Standard model matrix elements for neutral B-meson mixing and associated decay constants. *Physical Review D.* 2001.

117. Dorigo, . New Exclusive Upsilon Decays Observed By Belle! [Online] 10 May 2012. [Cited: 13 April 2021.] https://www.science20.com/quantum_diaries_survivor/new_exclusive_upsilon_decays_observed_belle-89925.

118. Marino, Gaetano de. D0 Lifetime Measurment with Belle II Early Data. Pisa : s.n., 2019.

119. Gamsızkan, and Collaboration, . Observation of the rare B0s??+?-decay from the combined analysis of CMS and LHCb data. 2015.

120. *Slow Pion Relative Tracking Efficiency In Belle II.* Souvik Maity, N Sushree Ipsita. s.l. : XXIV Dea-Brns High Energy Physics Symposiun, 2020.

121. Browder, , Oide, and Iijima, . Belle II super-B factory experiment takes shape at KEK. [Online] 12 August 2016. [Cited: 13 April 2021.] https://cerncourier.com/a/belle-ii-super-b-factory-experiment-takes-shape-at-kek/.

122. Kapliy, . Discovery of the Pion. Chicago : s.n., 30 April 2008.

123. Feldman, G. J., et al. Observation of the Decay $D^{*+} \rightarrow D^0 \pi^+$. *Physical Review Letters.* 1977.

124. Jung, Andereas Werner. *Measurement of the D* Meson Cross Section and Extraction of the Charm Contribution, F2c (x, Q2), to the Proton Structure in Deep Inelastic ep Scattering with the H1 Detector at HERA.* Heidelberg : (Doctoral dissertation), 2008.

125. *Charm meson decays.* Artuso, and al, . s.l. : Annual Review of Nuclear and Particle Science, 2008, Vol. 58.

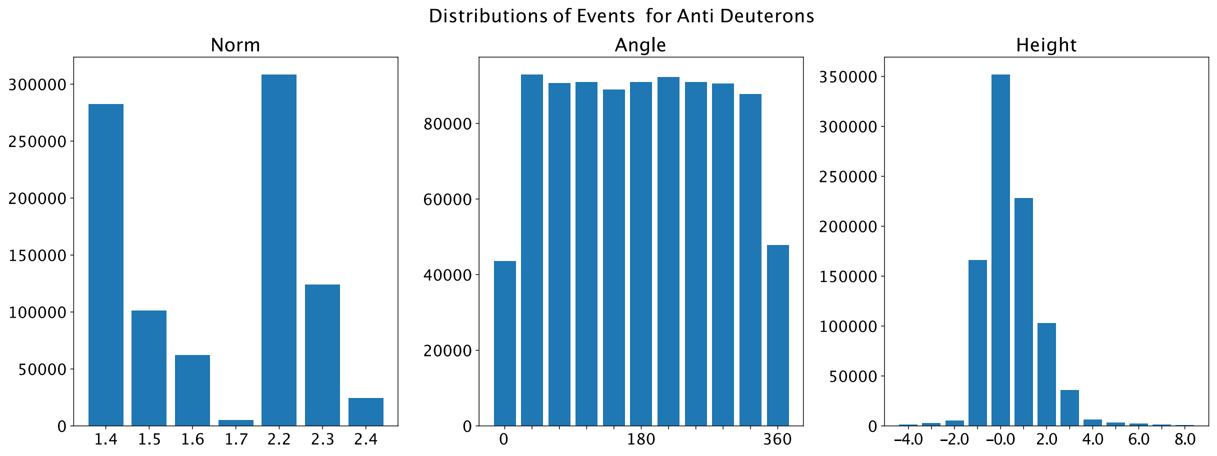
126. Herb, S. W. and al, . Observation of a dimuon resonance at 9.5 GeV in 400-GeV proton-nucleus collisions. *Physical Review Letters.* 1977.

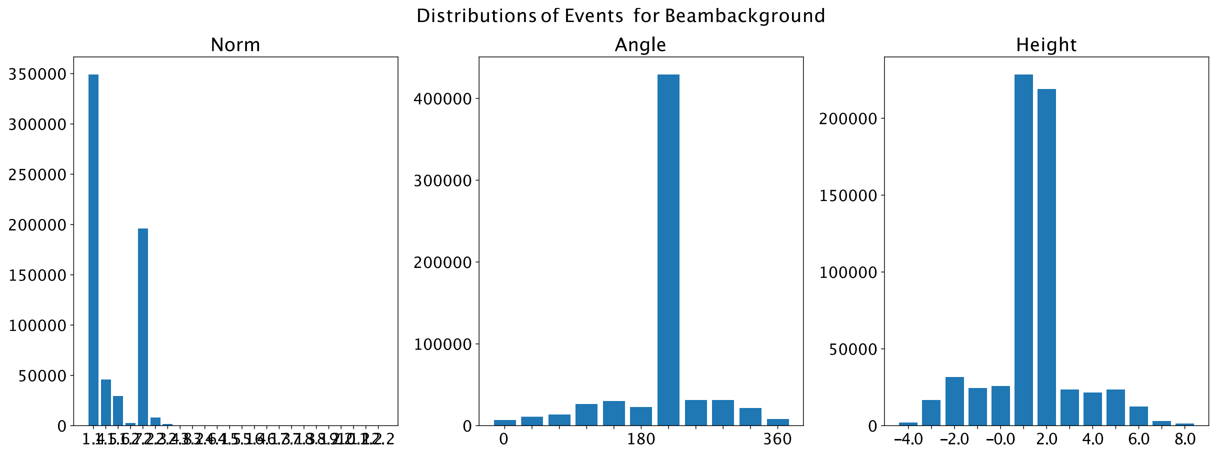
127. Spencer, John H. *he Eternal Law: Ancient Greek Philosophy, Modern Physics and Ultimate Reality.* s.l. : Param Media, 2013.

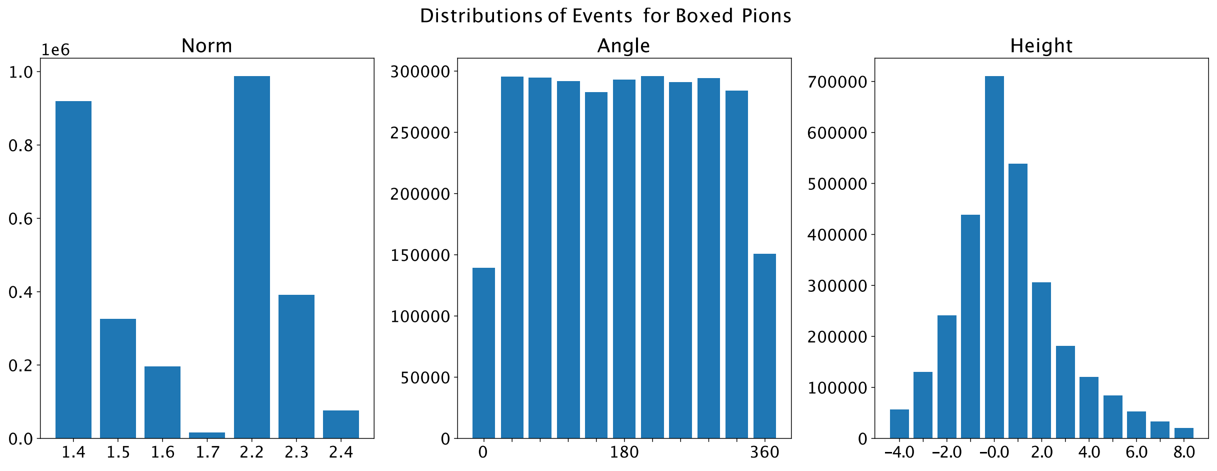
128. *The belle II silicon vertex detector.* Friedl, and al, . s.l. : Physics Procedia, 2012, Vol. 37.

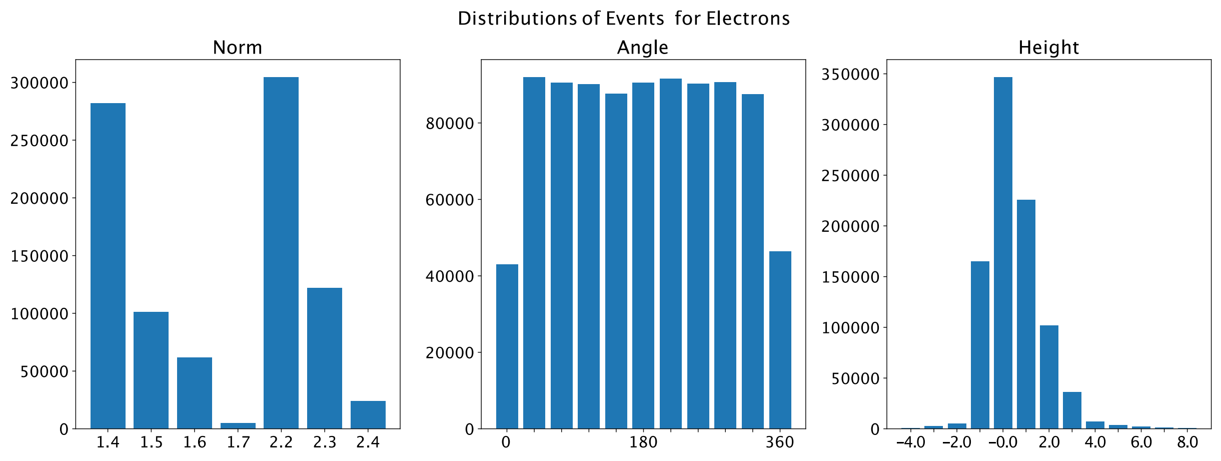
A. Additional Graphs

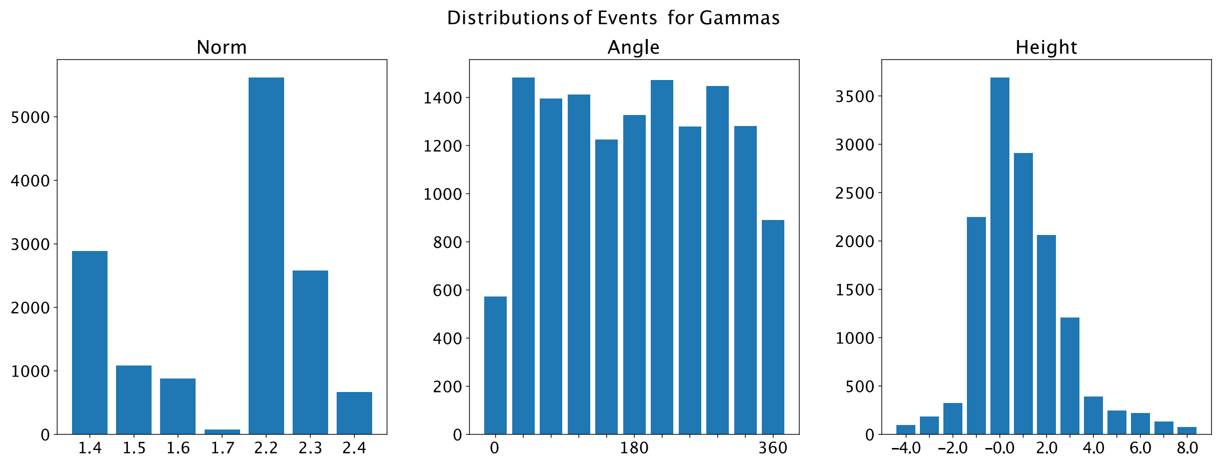
Event Coordinate Distributions

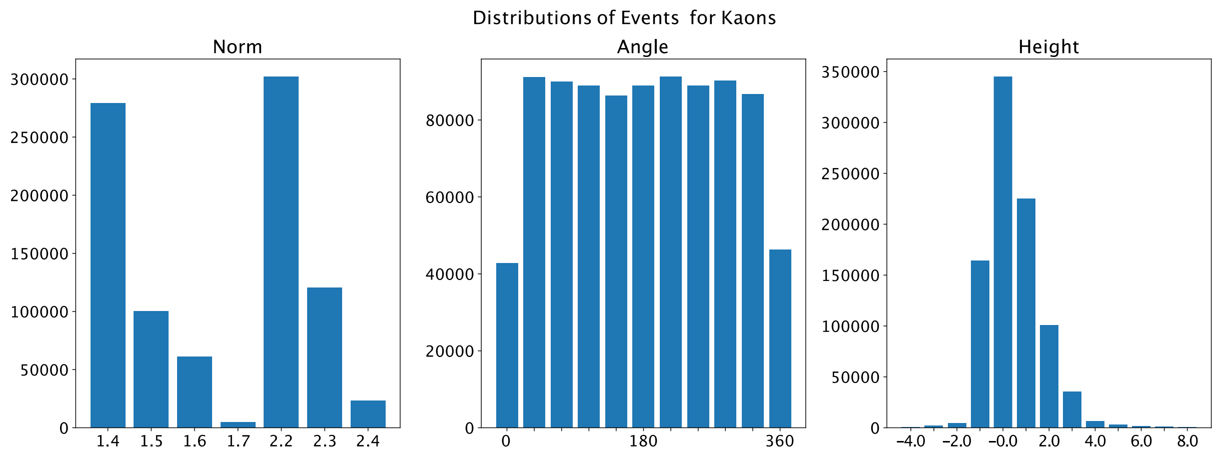


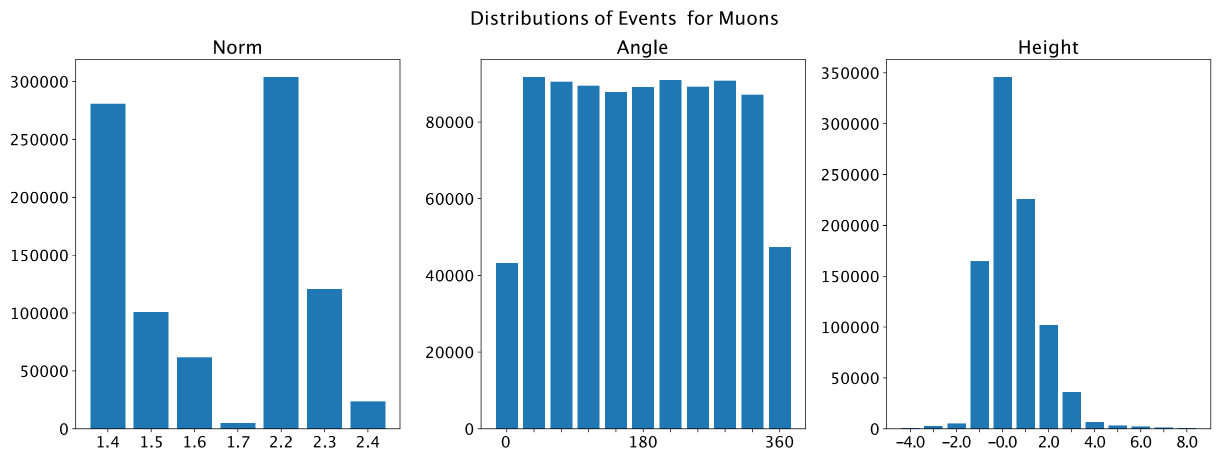


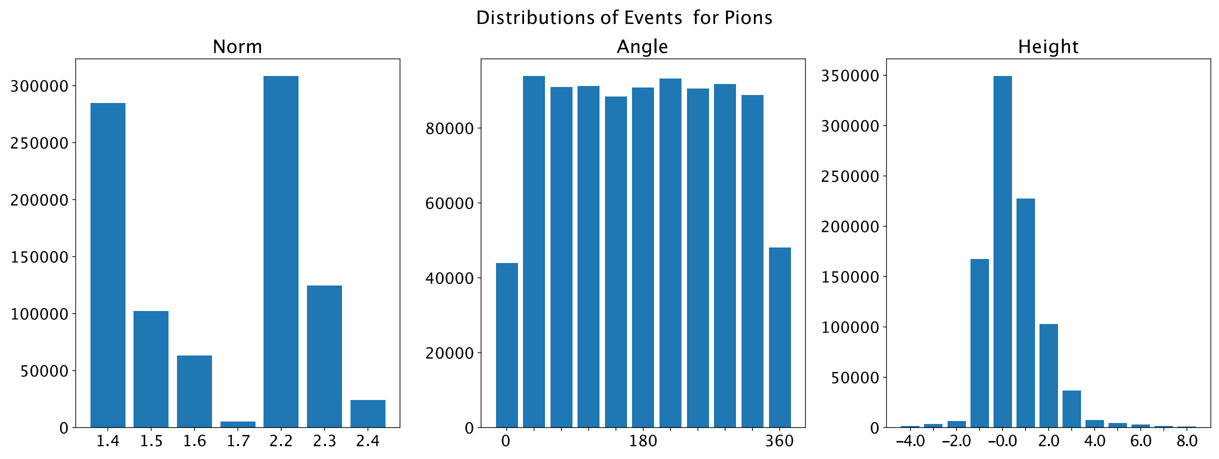


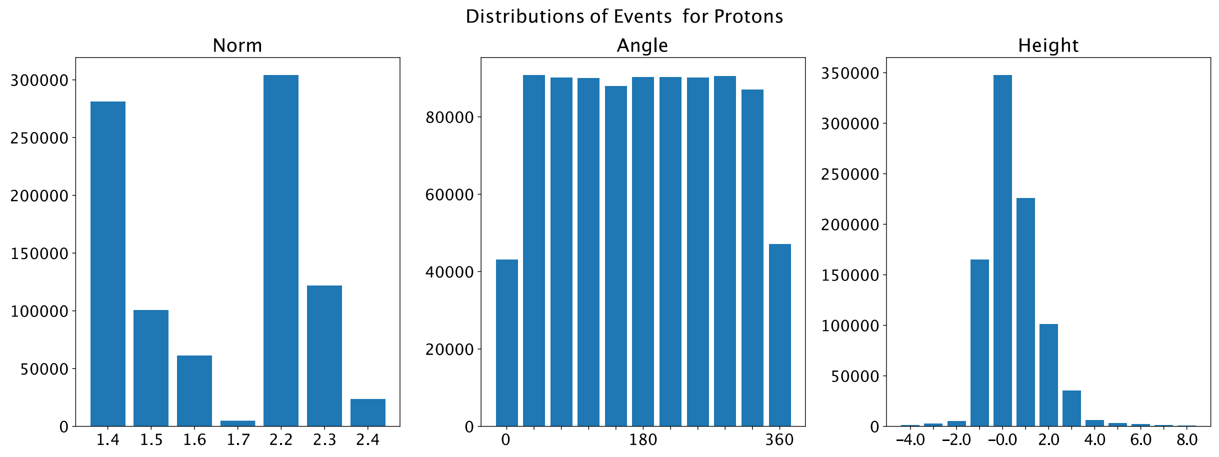

Distributions of Events for Electrons
Norm
Angle
Height

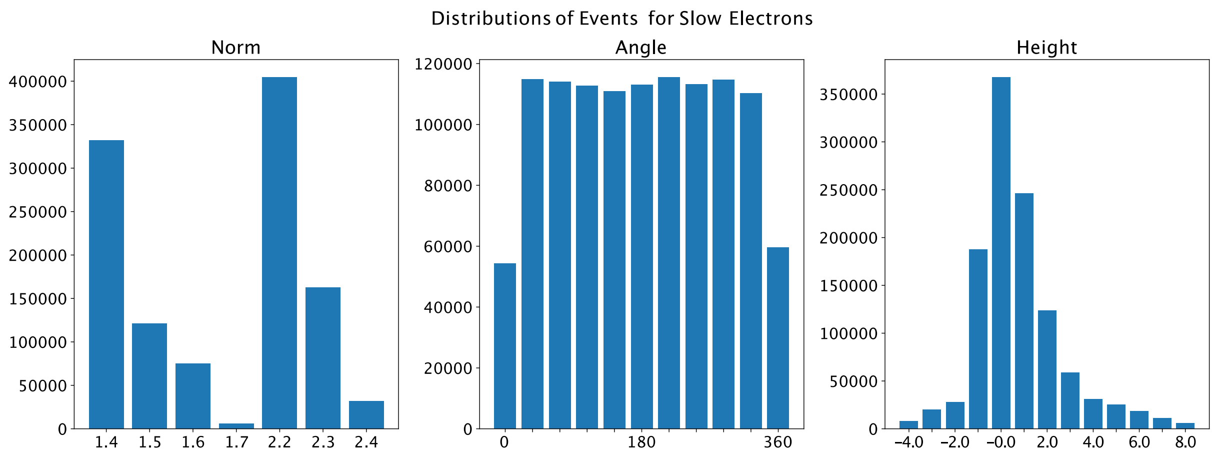

Distributions of Events for Gammas
Norm
Angle
Height


Distributions of Events for Kaons
Norm
Angle
Height

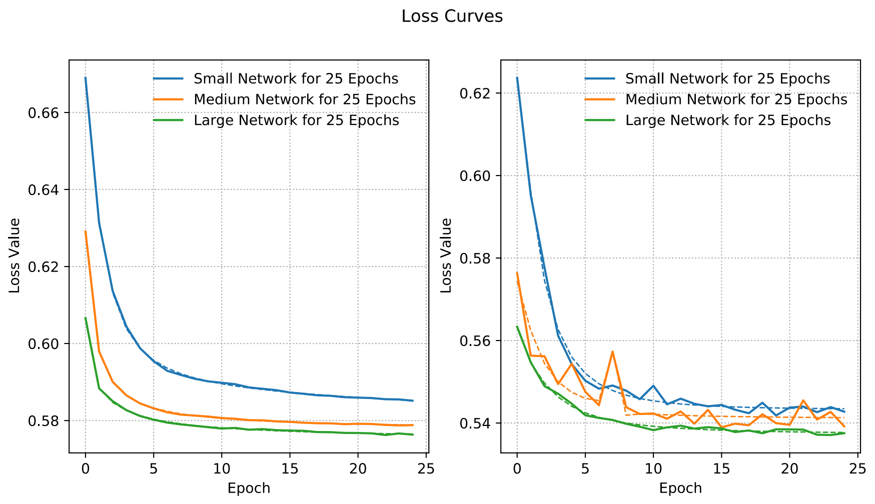

Distributions of Events for Muons
Norm
Angle
Height
300000
250000
200000
150000
100000
50000
0
1.4
1.5
1.6
1.7
2.2
2.3
2.4
80000
60000
40000
20000
0
180
360
350000
−4.0
−2.0
−0.0
2.0
4.0
6.0
8.0

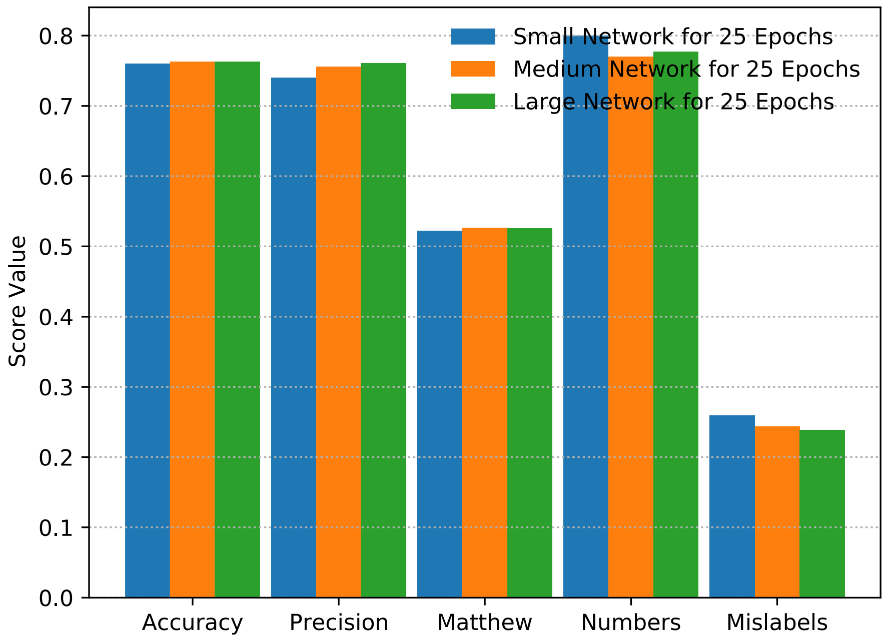

Distributions of Events for Pions
Norm
Angle
Height
300000
250000
200000
150000
100000
50000
0
1.4
1.5
1.6
1.7
2.2
2.3
2.4
80000
60000
40000
20000
0
180
360
350000
−4.0
−2.0
−0.0
2.0
4.0
6.0
8.0

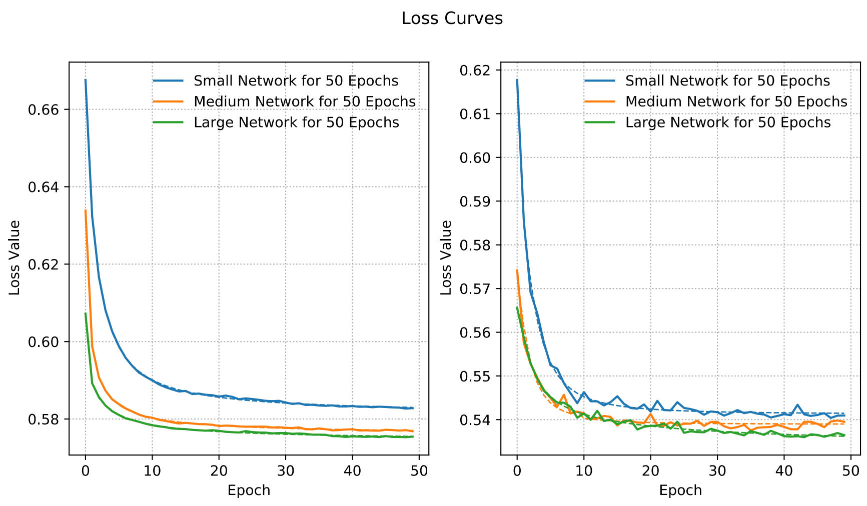

Distributions of Events for Protons
Norm
Angle
Height
300000
250000
200000
150000
100000
50000
0
1.4
1.5
1.6
1.7
2.2
2.3
2.4
80000
60000
40000
20000
0
180
360
350000
−4.0
−2.0
−0.0
2.0
4.0
6.0
8.0

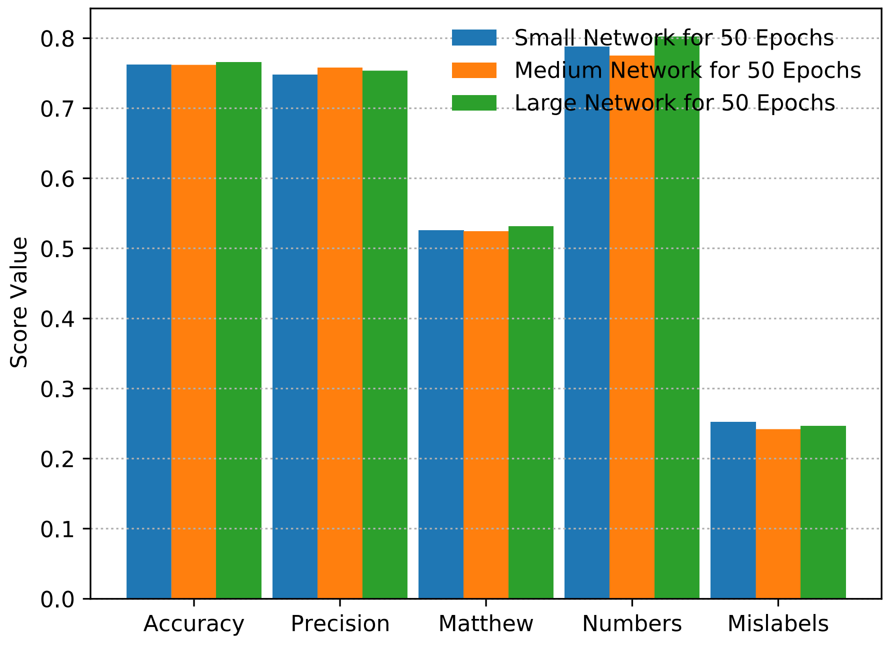

Distributions of Events for Slow Electrons
Norm
Angle
Height
400000
350000
300000
250000
200000
150000
100000
50000
0
1.4
1.5
1.6
1.7
2.2
2.3
2.4
120000
100000
80000
60000
40000
20000
0
0
180
360
350000
300000
250000
200000
150000
100000
50000
0
−4.0
−2.0
−0.0
2.0
4.0
6.0
8.0

Additional Plots for Long-Term Test Runs

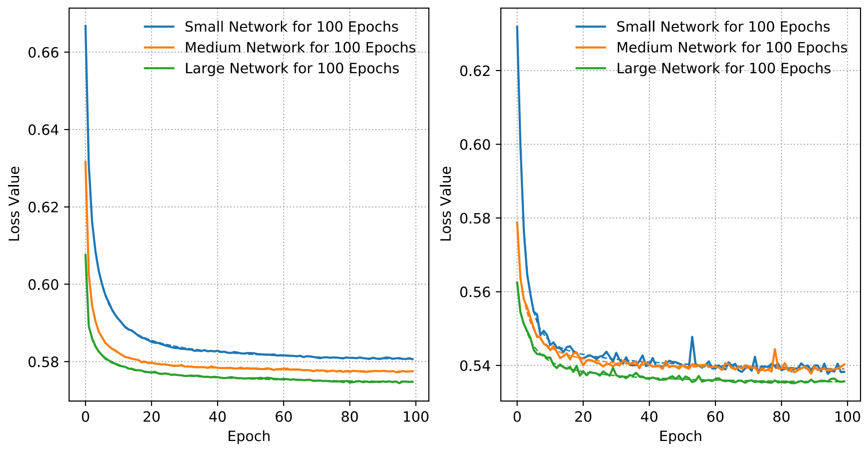




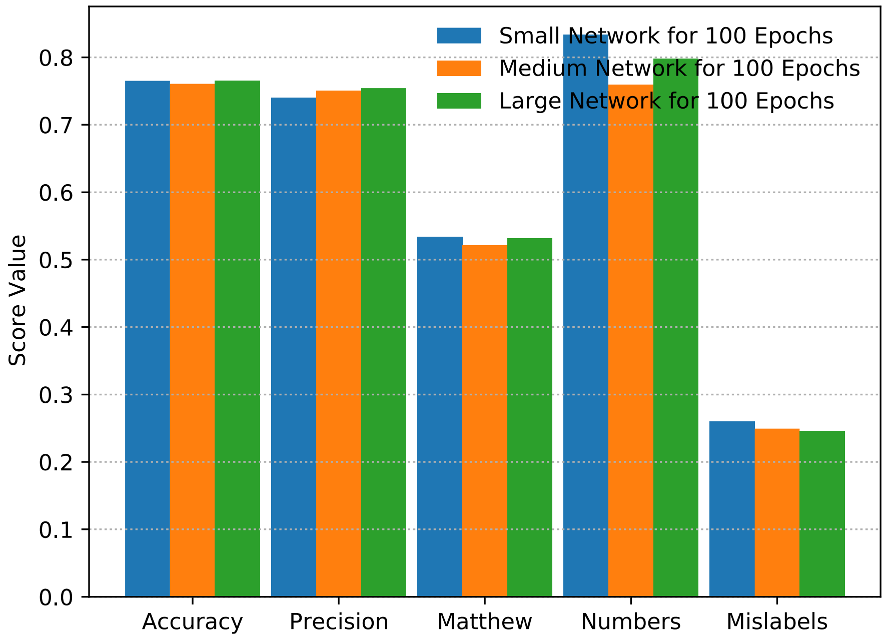

Loss Curves
Small Network for 50 Epochs
Medium Network for 50 Epochs
Large Network for 50 Epochs
Loss Value
Epoch


Small Network for 50 Epochs
Medium Network for 50 Epochs
Large Network for 50 Epochs
Score Value
Accuracy
Precision
Matthew
Numbers
Mislabels

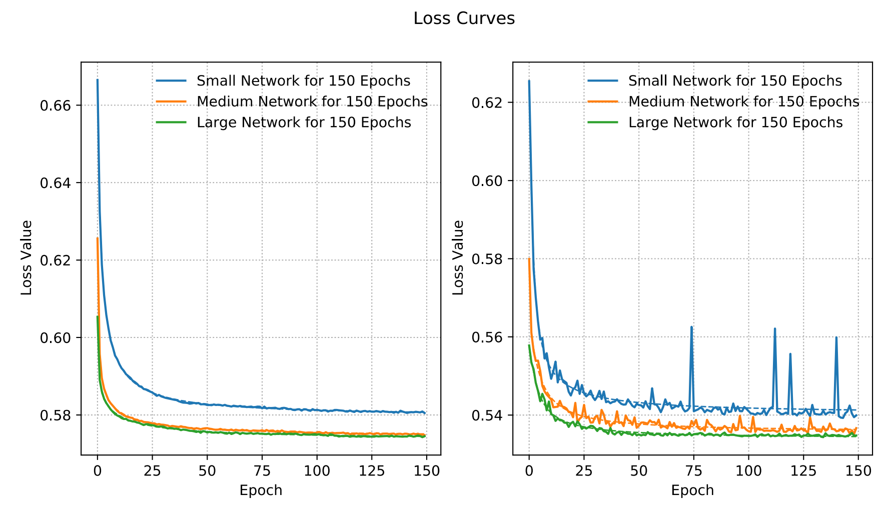
Loss Curves

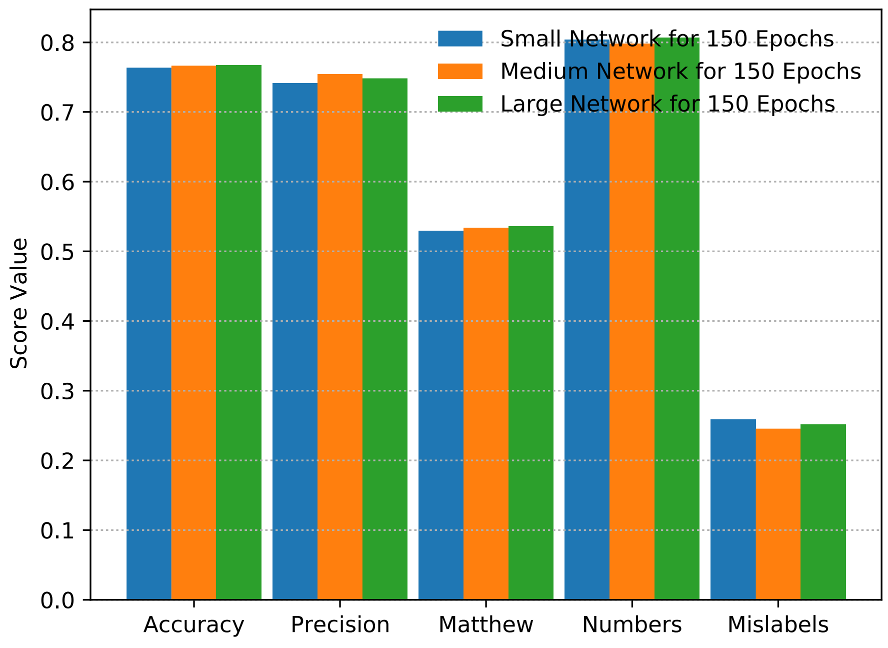


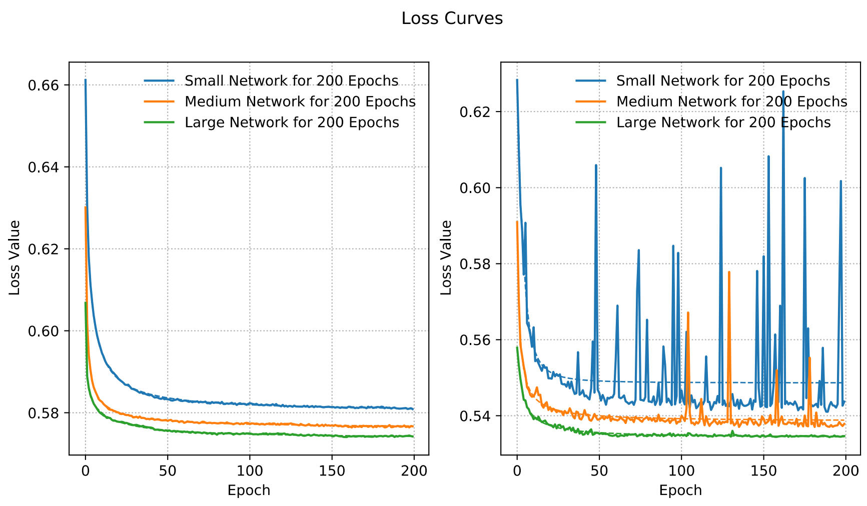
Small Network for 100 Epochs
Medium Network for 100 Epochs
Large Network for 100 Epochs
Loss Value
Epoch

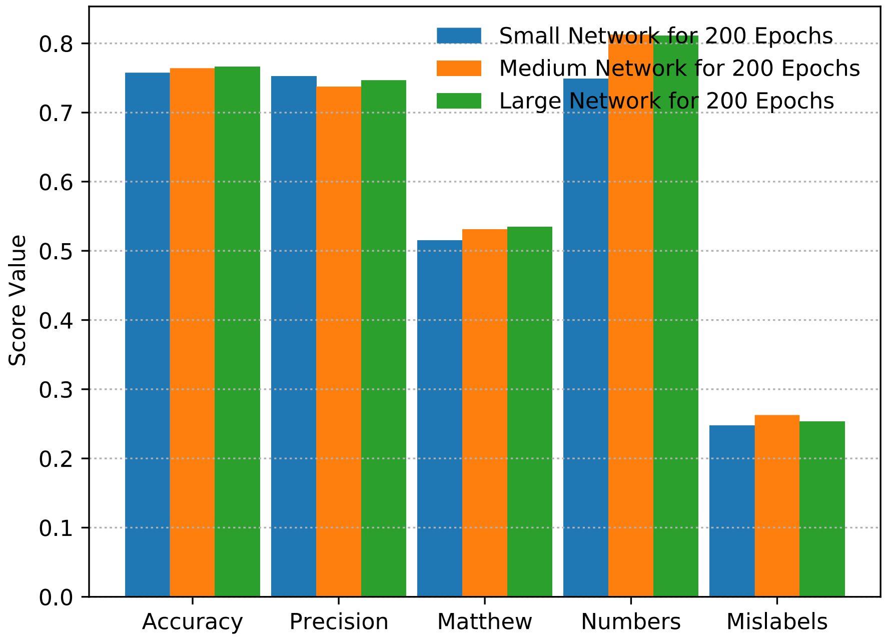


Small Network for 100 Epochs
Medium Network for 100 Epochs
Large Network for 100 Epochs
Score Value
Accuracy
Precision
Matthew
Numbers
Mislabels

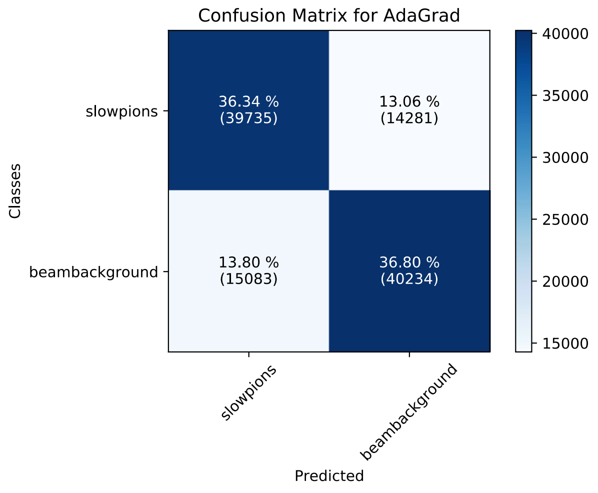

Loss Curves
Small Network for 150 Epochs
Medium Network for 150 Epochs
Large Network for 150 Epochs
Loss Value
Epoch

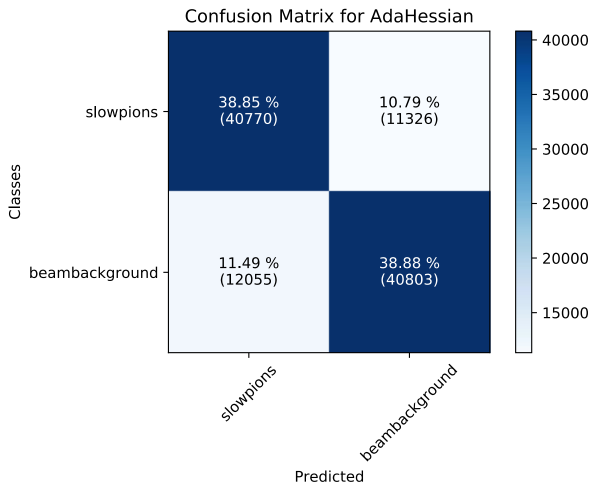

Small Network for 150 Epochs
Medium Network for 150 Epochs
Large Network for 150 Epochs
Score Value
Accuracy
Precision
Matthew
Numbers
Mislabels

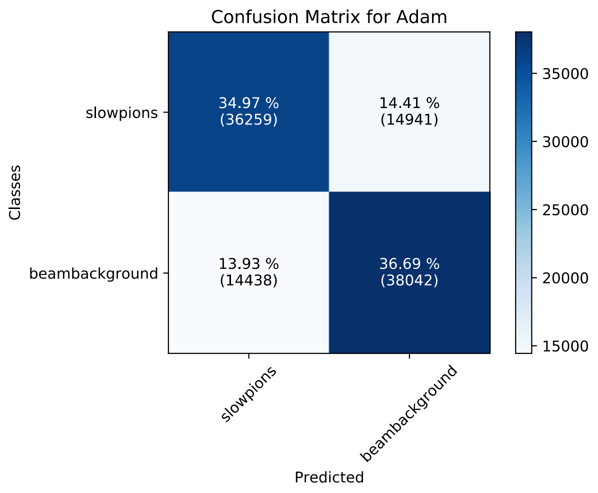

Loss Curves
Small Network for 200 Epochs
Medium Network for 200 Epochs
Large Network for 200 Epochs
Loss Value
Epoch

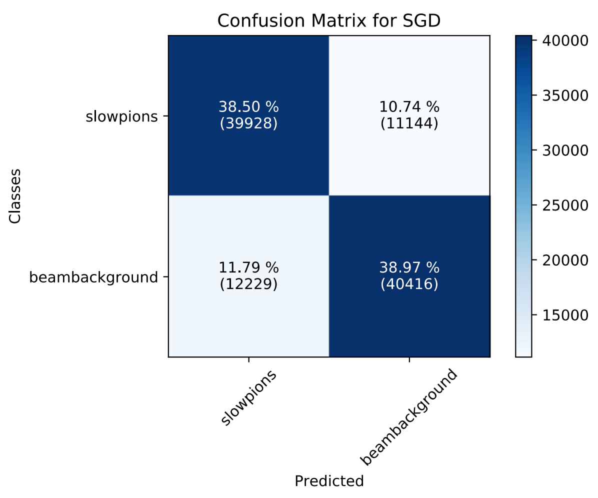

Small Network for 200 Epochs
Medium Network for 200 Epochs
Large Network for 200 Epochs
Score Value
Accuracy
Precision
Matthew
Numbers
Mislabels

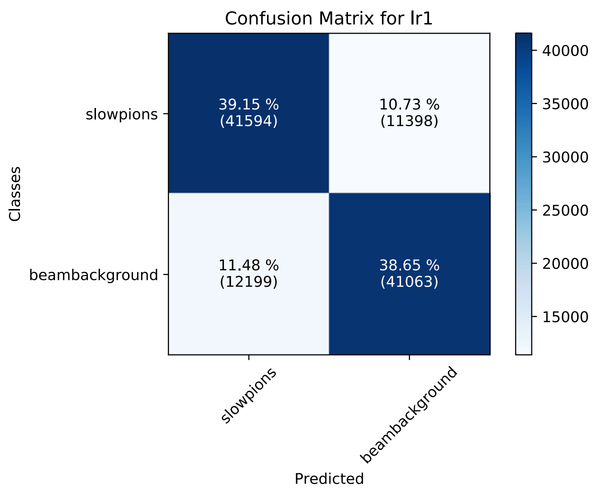
Confusion Matrices for all Runs

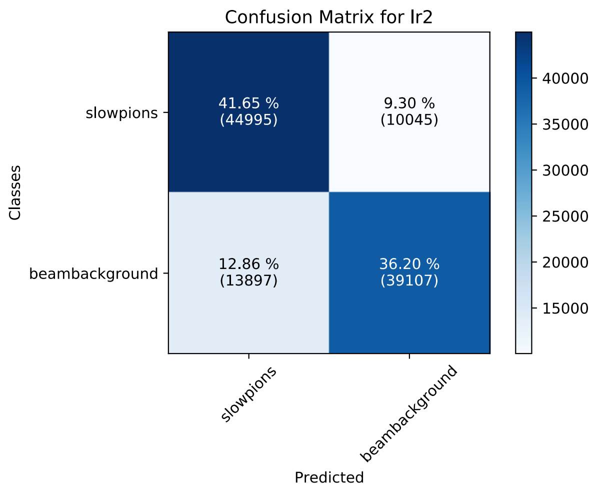


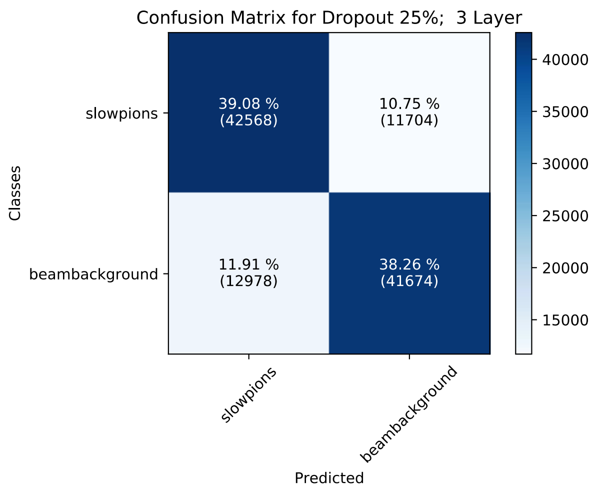


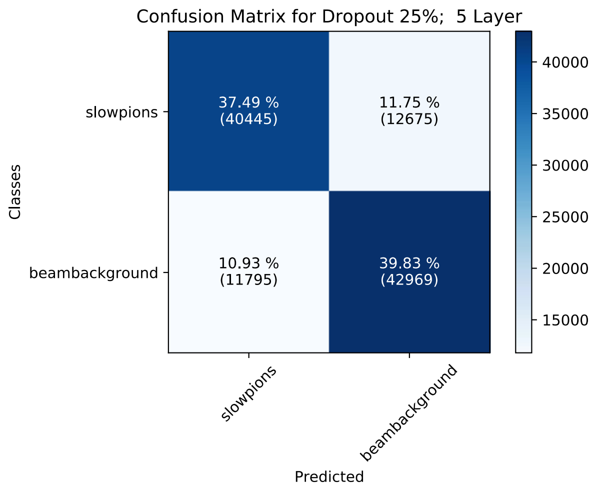


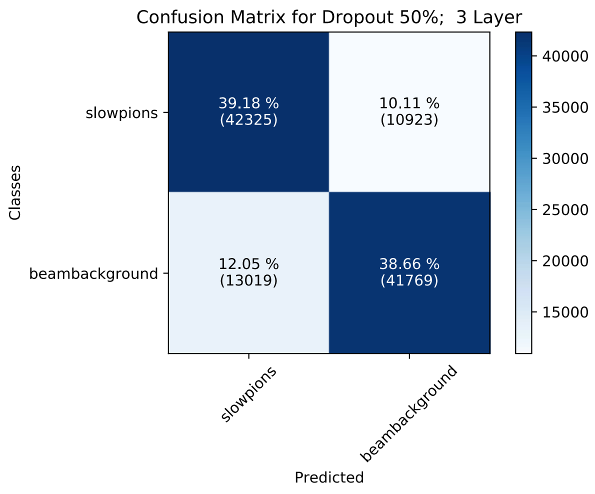

Confusion Matrix for SGD
slowpions
beambackground
Classes
38.50 % (39928)
10.74 % (11144)
11.79 % (12229)
38.97 % (40416)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000

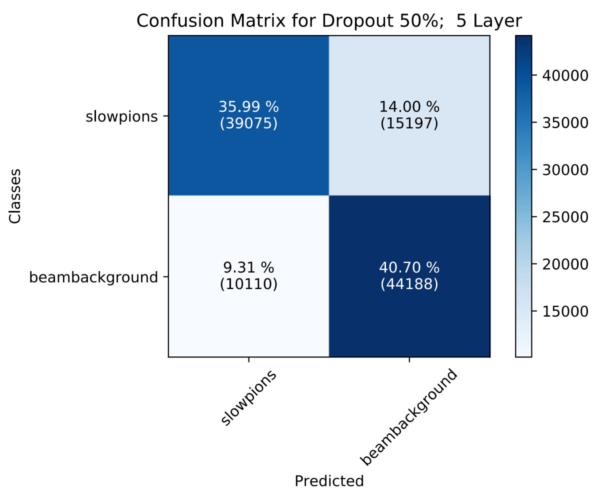

Confusion Matrix for lr1
slowpions
beambackground
Classes
39.15 % (41594)
10.73 % (11398)
11.48 % (12199)
38.65 % (41063)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000

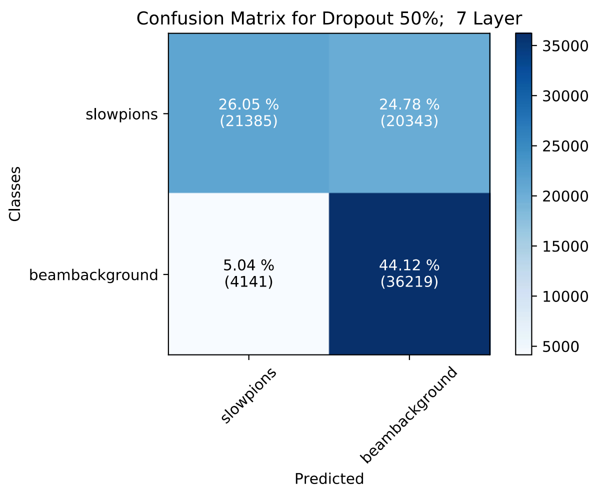

Confusion Matrix for lr2
slowpions
beambackground
Classes
41.65 % (44995)
9.30 % (10045)
12.86 % (13897)
36.20 % (39107)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000

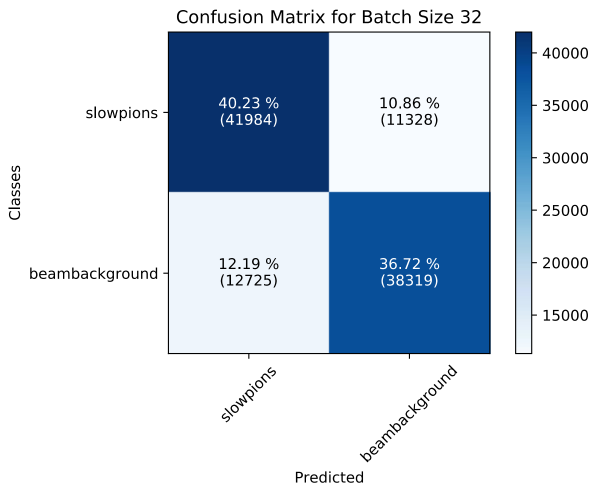

Confusion Matrix for Dropout 25%; 3 Layer
Classes
slowpions
beambackground
39.08 %
(42568)
10.75 %
(11704)
11.91 %
(12978)
38.26 %
(41674)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000

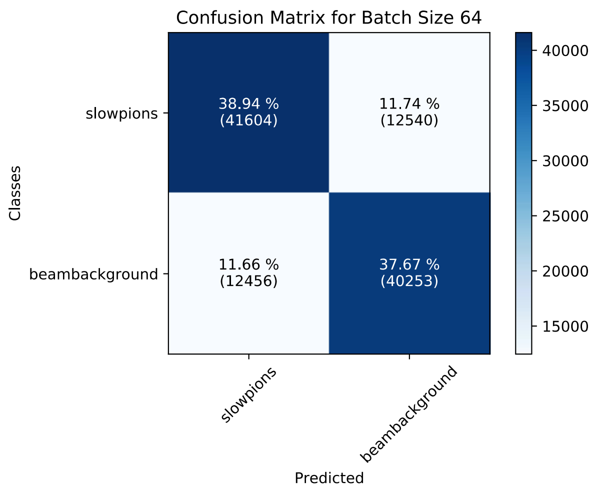

Confusion Matrix for Dropout 25%; 5 Layer
Classes
slowpions
beambackground
37.49 %
(40445)
11.75 %
(12675)
10.93 %
(11795)
39.83 %
(42969)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000

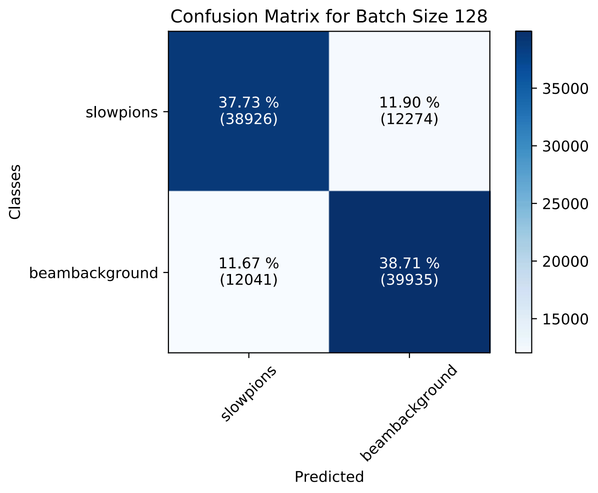

Confusion Matrix for Dropout 50%; 3 Layer
Classes
slowpions
beambackground
39.18 %
(42325)
10.11 %
(10923)
12.05 %
(13019)
38.66 %
(41769)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000

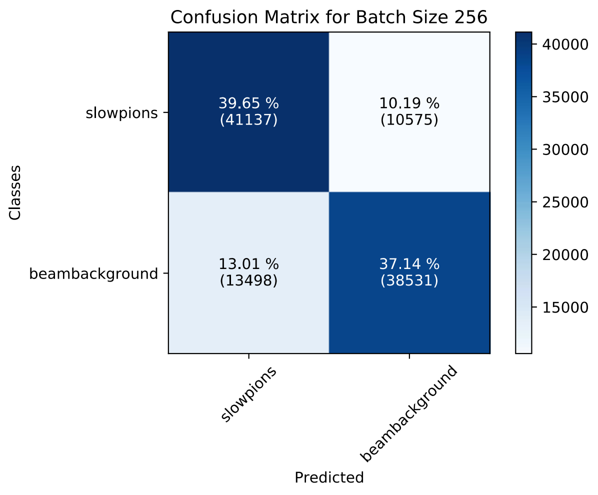

Confusion Matrix for Dropout 50%; 5 Layer
slowpions
beambackground
Classes
35.99 % (39075)
14.00 % (15197)
9.31 % (10110)
40.70 % (44188)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000

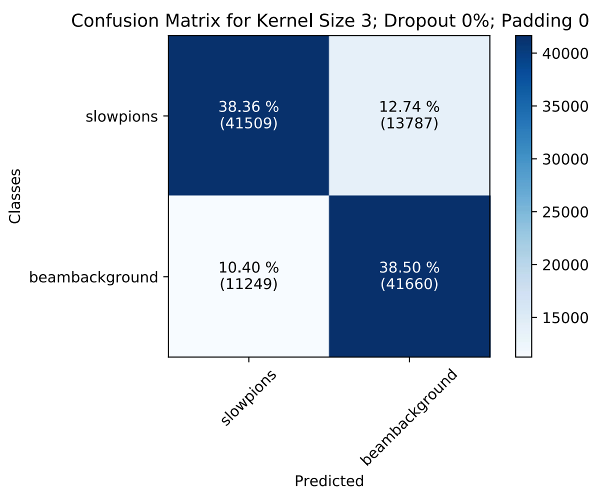

Confusion Matrix for Dropout 50%; 7 Layer
slowpions
beambackground
Classes
26.05 % (21385)
24.78 % (20343)
5.04 % (4141)
44.12 % (36219)
slowpions
beambackground
Predicted
35000
30000
25000
20000
15000
10000
5000

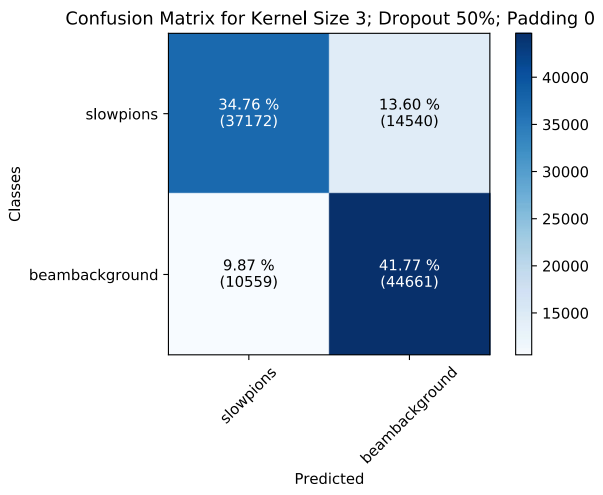

Confusion Matrix for Batch Size 32
slowpions
beambackground
Classes
40.23 % (41984)
10.86 % (11328)
12.19 % (12725)
36.72 % (38319)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000

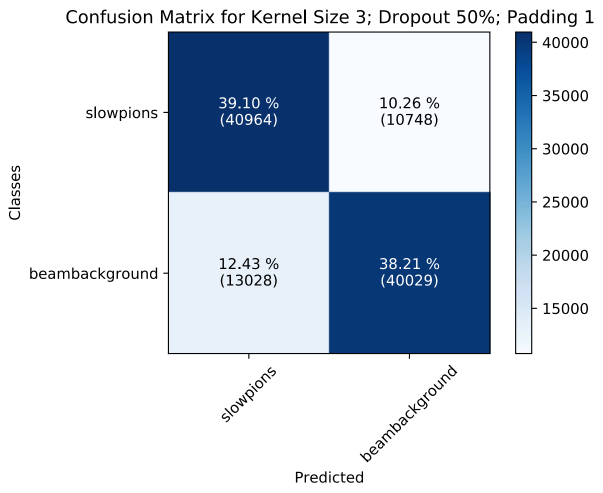

Confusion Matrix for Batch Size 64
slowpions
beambackground
Classes
38.94 %
(41604)
11.74 %
(12540)
11.66 %
(12456)
37.67 %
(40253)
Predicted
40000
35000
30000
25000
20000
15000

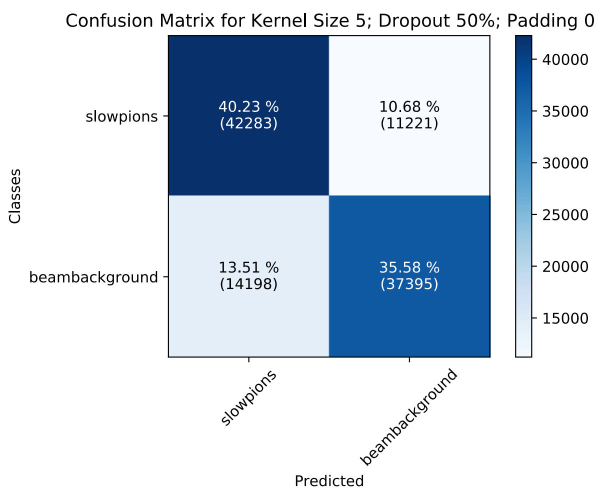

Confusion Matrix for Batch Size 128
slowpions
beambackground
Classes
37.73 %
(38926)
11.90 %
(12274)
11.67 %
(12041)
38.71 %
(39935)
Predicted
35000
30000
25000
20000
15000

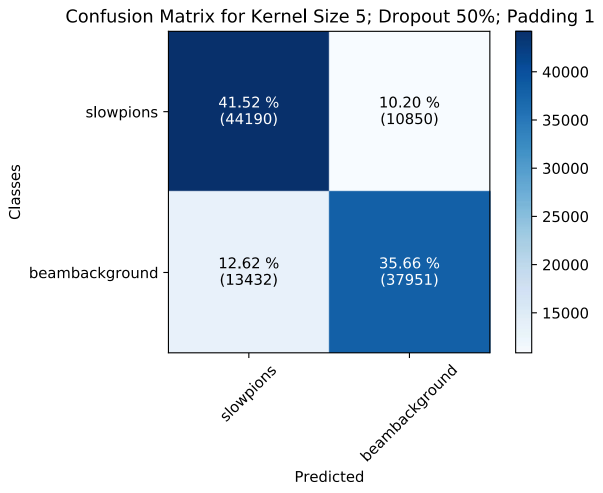

Confusion Matrix for Batch Size 256
slowpions
beambackground
Classes
39.65 %
(41137)
10.19 %
(10575)
13.01 %
(13498)
37.14 %
(38531)
Predicted
40000
35000
30000
25000
20000
15000

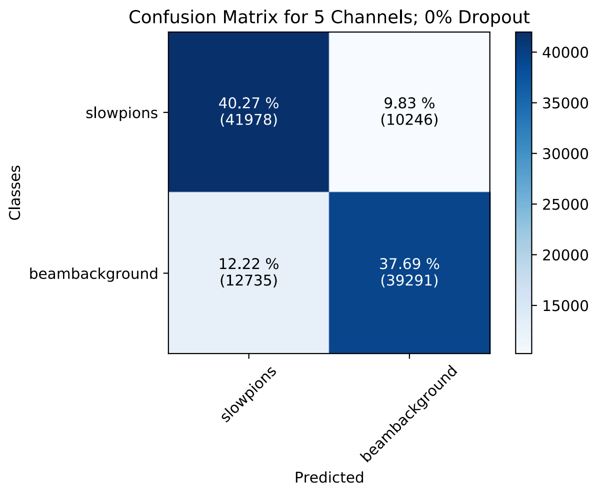

Confusion Matrix for Kernel Size 3; Dropout 0%; Padding 0
Classes
slowpions
beambackground
38.36 % (41509)
12.74 % (13787)
10.40 % (11249)
38.50 % (41660)
Predicted
slowpions
beambackground
40000
35000
30000
25000
20000
15000

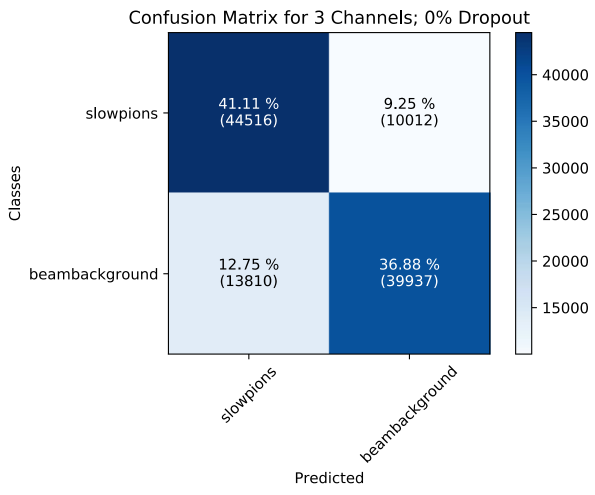

Confusion Matrix for Kernel Size 3; Dropout 50%; Padding 0
Classes
slowpions
beambackground
34.76 % (37172)
13.60 % (14540)
9.87 % (10559)
41.77 % (44661)
Predicted
slowpions
beambackground
40000
35000
30000
25000
20000
15000


Confusion Matrix for Kernel Size 3; Dropout 50%; Padding 1
Classes
slowpions
beambackground
39.10 % (40964)
10.26 % (10748)
12.43 % (13028)
38.21 % (40029)
Predicted
slowpions
beambackground
40000
35000
30000
25000
20000
15000

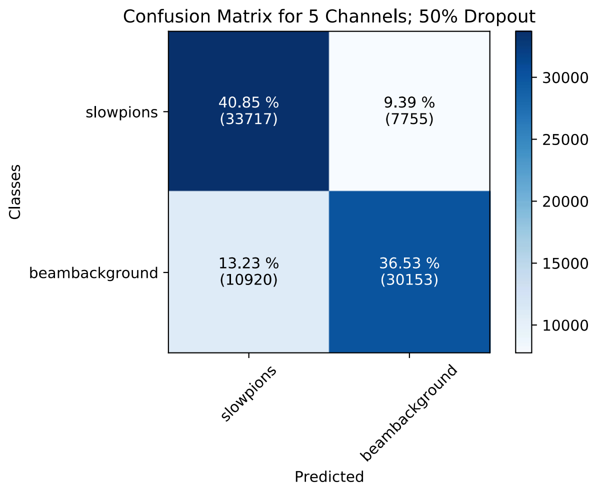

Confusion Matrix for Kernel Size 5; Dropout 50%; Padding 0
slowpions
beambackground
Classes
40.23 %
(42283)
10.68 %
(11221)
13.51 %
(14198)
35.58 %
(37395)
40000
35000
30000
25000
20000
15000
slowpions
beambackground
Predicted


Confusion Matrix for Kernel Size 5; Dropout 50%; Padding 1
slowpions
beambackground
Classes
41.52 %
(44190)
10.20 %
(10850)
12.62 %
(13432)
35.66 %
(37951)
40000
35000
30000
25000
20000
15000
slowpions
beambackground
Predicted

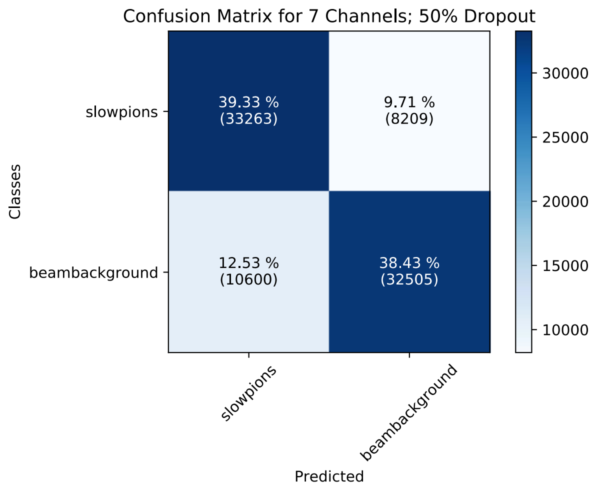

Confusion Matrix for 5 Channels; 0% Dropout
slowpions
beambackground
Classes
40.27 %
(41978)
9.83 %
(10246)
12.22 %
(12735)
37.69 %
(39291)
40000
35000
30000
25000
20000
15000
slowpions
beambackground
Predicted



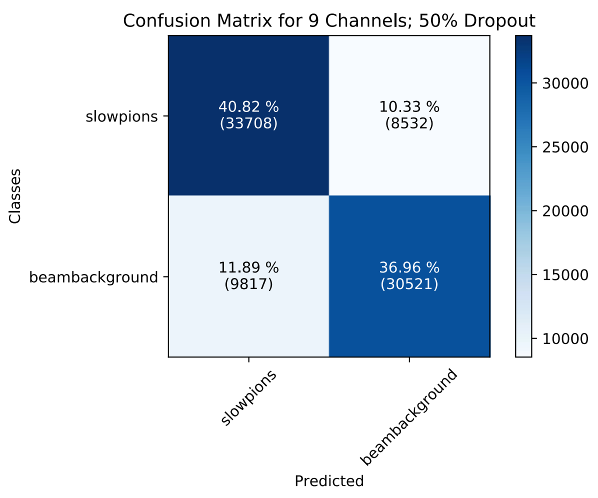
Confusion Matrix for 3 Channels; 0% Dropout
41.11 % (44516)
9.25 % (10012)
12.75 % (13810)
36.88 % (39937)
slowpions
beambackground
Classes
Predicted
40000
35000
30000
25000
20000
15000



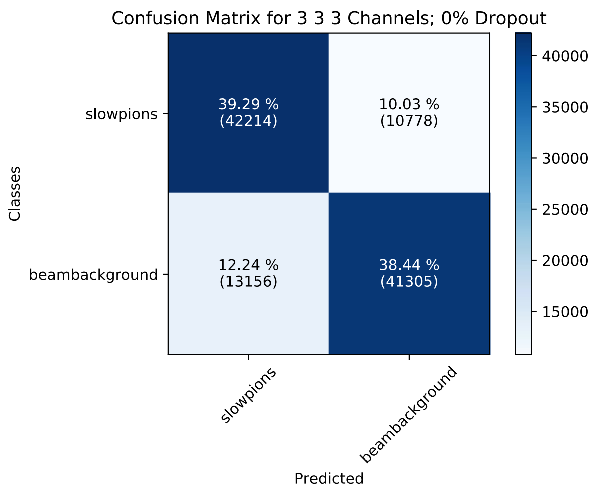
Confusion Matrix for 5 Channels; 50% Dropout
40.85 % (33717)
9.39 % (7755)
13.23 % (10920)
36.53 % (30153)
slowpions
beambackground
Classes
Predicted
30000
25000
20000
15000
10000



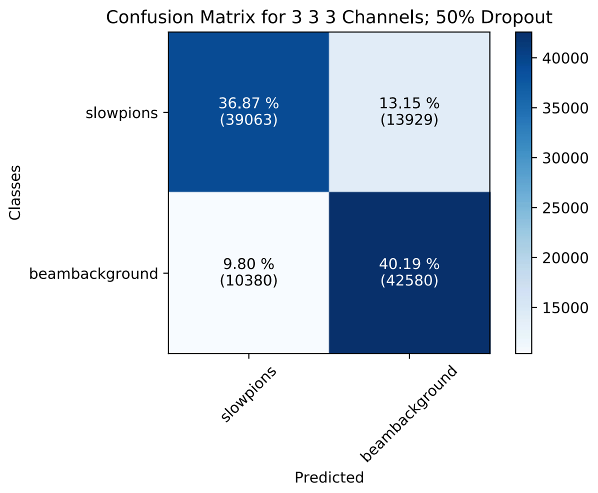
Confusion Matrix for 7 Channels; 50% Dropout
39.33 % (33263)
9.71 % (8209)
12.53 % (10600)
38.43 % (32505)
slowpions
beambackground
Classes
Predicted
30000
25000
20000
15000
10000



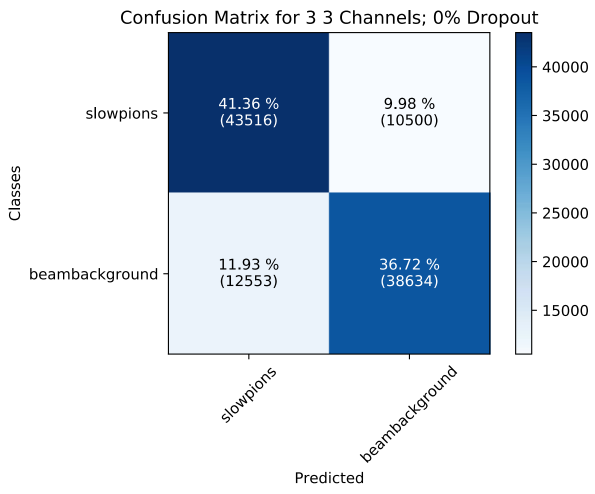
Confusion Matrix for 9 Channels; 50% Dropout
slowpions
beambackground
Classes
40.82 %
(33708)
10.33 %
(8532)
11.89 %
(9817)
36.96 %
(30521)
slowpions
beambackground
Predicted
30000
25000
20000
15000
10000



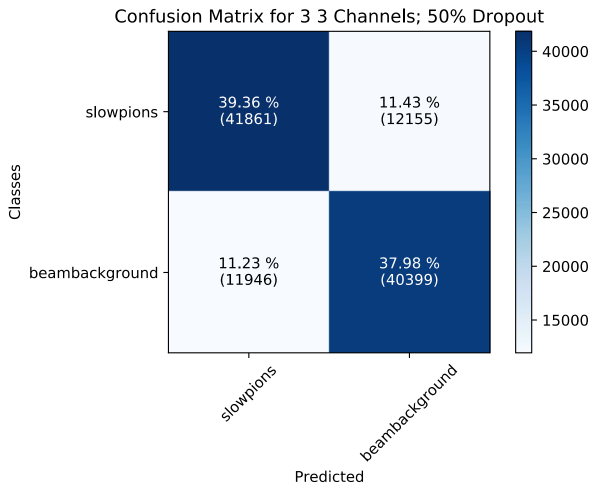
Confusion Matrix for 3 3 3 Channels; 0% Dropout
slowpions
beambackground
Classes
39.29 %
(42214)
10.03 %
(10778)
12.24 %
(13156)
38.44 %
(41305)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000



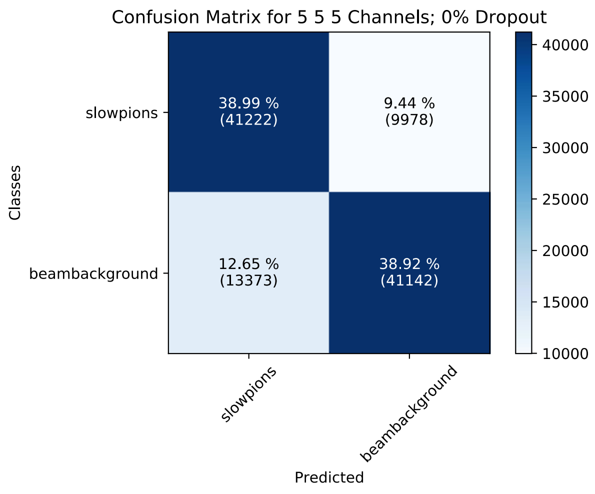
Confusion Matrix for 3 3 3 Channels; 50% Dropout
slowpions
beambackground
Classes
36.87 %
(39063)
13.15 %
(13929)
9.80 %
(10380)
40.19 %
(42580)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000



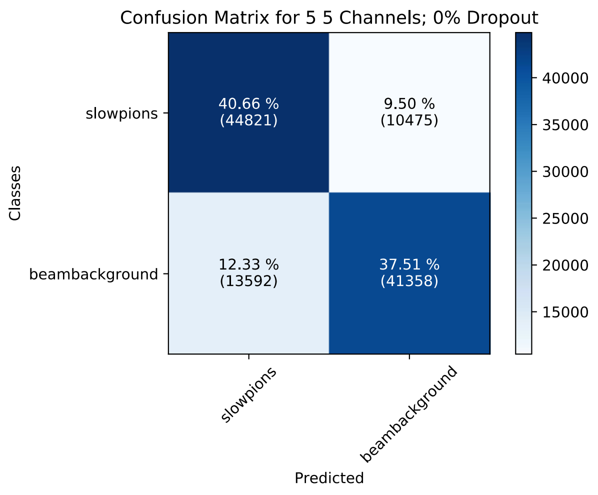
Confusion Matrix for 3 3 Channels; 0% Dropout
slowpions
beambackground
Classes
41.36 % (43516)
9.98 % (10500)
11.93 % (12553)
36.72 % (38634)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000

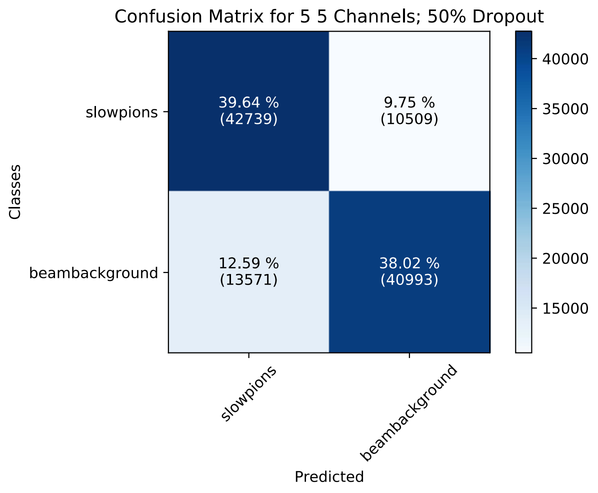


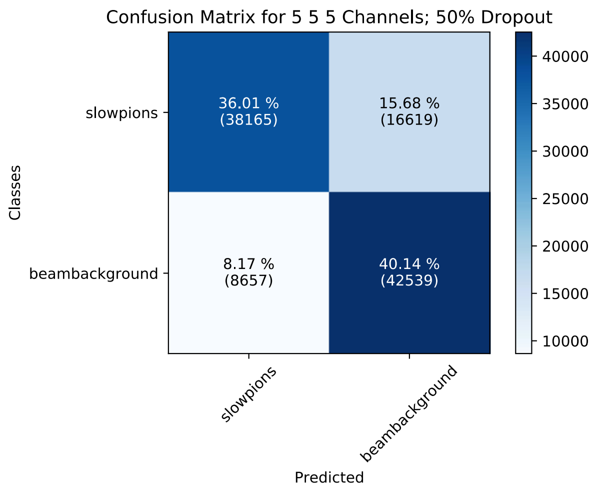
Confusion Matrix for 3 3 Channels; 50% Dropout
slowpions
beambackground
Classes
39.36 % (41861)
11.43 % (12155)
11.23 % (11946)
37.98 % (40399)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000

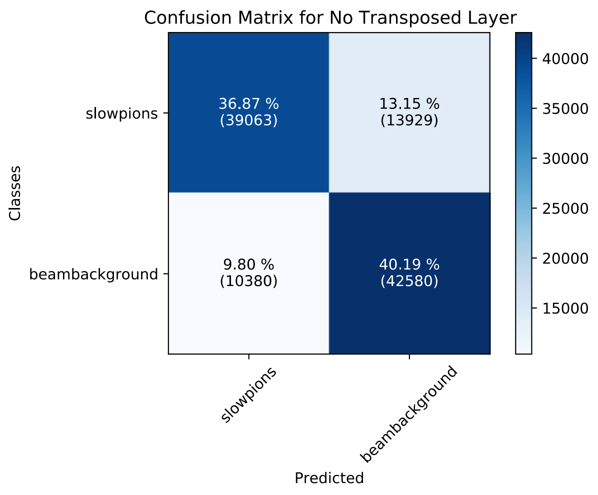


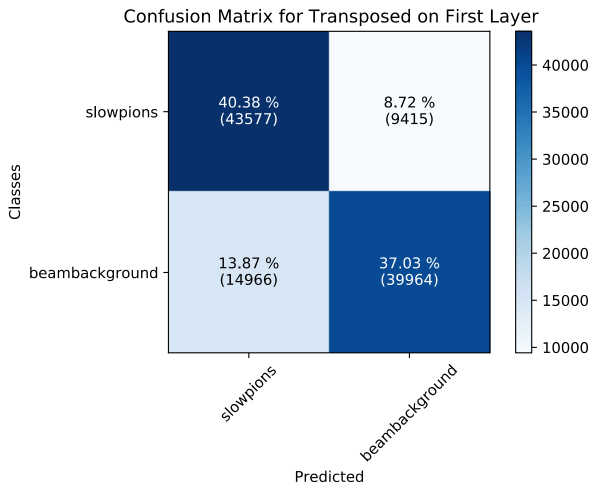
Confusion Matrix for 5 5 5 Channels; 0% Dropout
slowpions
beambackground
Classes
38.99 % (41222)
9.44 % (9978)
12.65 % (13373)
38.92 % (41142)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000
10000

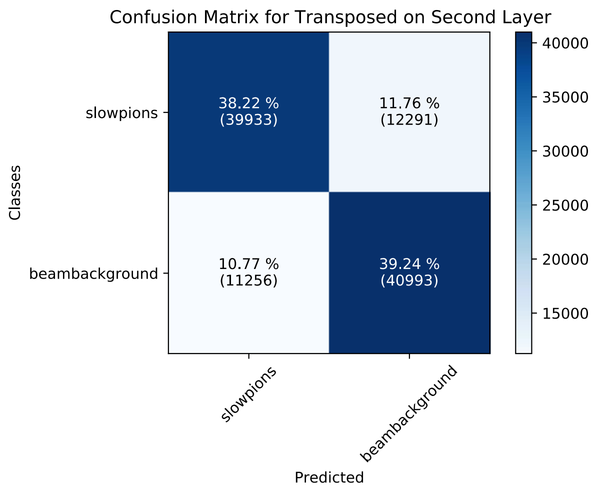

Confusion Matrix for 5 5 Channels; 0% Dropout
Classes
slowpions
beambackground
40.66 % (44821)
9.50 % (10475)
12.33 % (13592)
37.51 % (41358)
Predicted
40000
35000
30000
25000
20000
15000

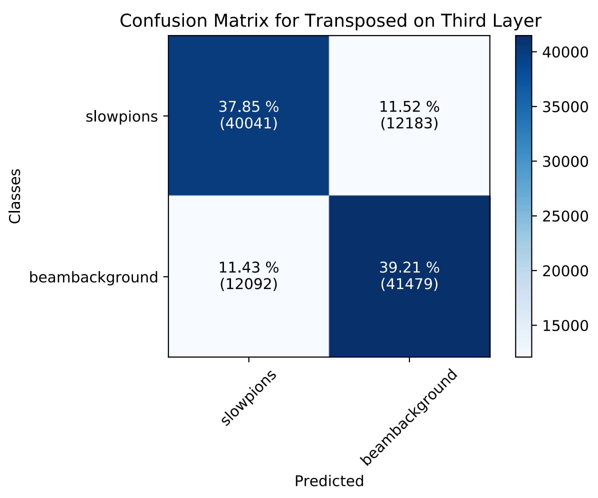

Confusion Matrix for 5 5 Channels; 50% Dropout
Classes
slowpions
beambackground
39.64 % (42739)
9.75 % (10509)
12.59 % (13571)
38.02 % (40993)
Predicted
40000
35000
30000
25000
20000
15000

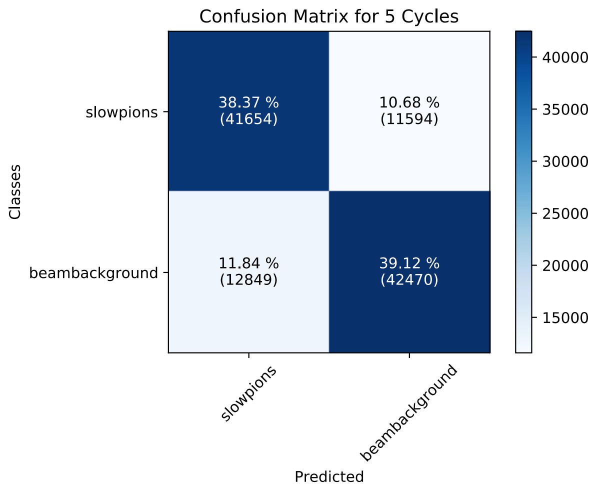

Confusion Matrix for 5 5 5 Channels; 50% Dropout
Classes
slowpions
beambackground
36.01 % (38165)
15.68 % (16619)
8.17 % (8657)
40.14 % (42539)
Predicted
40000
35000
30000
25000
20000
15000
10000

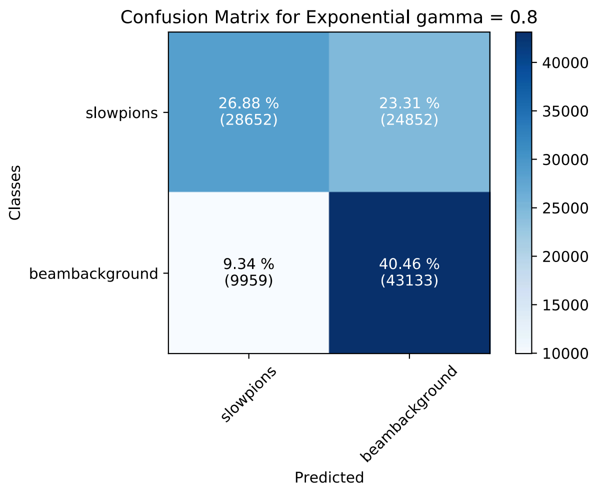

Confusion Matrix for No Transposed Layer
slowpions
beambackground
Classes
36.87 %
(39063)
13.15 %
(13929)
9.80 %
(10380)
40.19 %
(42580)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000

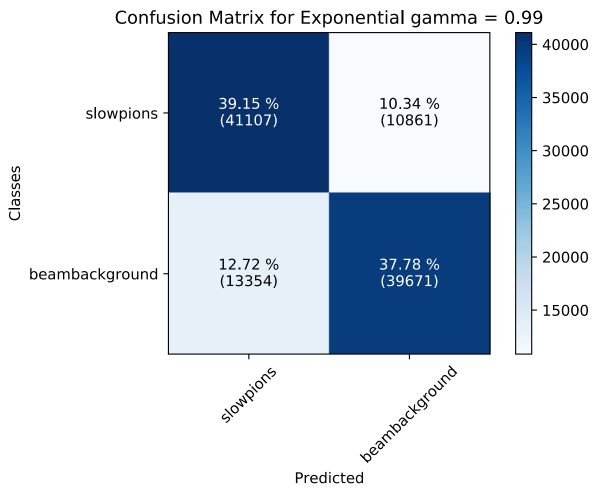

Confusion Matrix for Transposed on First Layer
slowpions
beambackground
Classes
40.38 %
(43577)
8.72 %
(9415)
13.87 %
(14966)
37.03 %
(39964)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000
10000

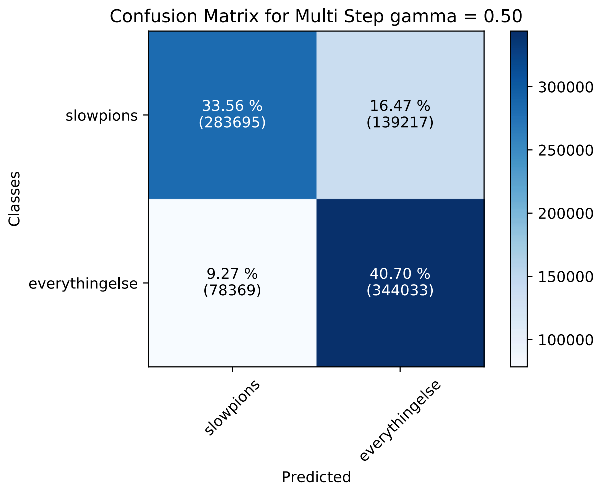

Confusion Matrix for Transposed on Second Layer
slowpions
beambackground
Classes
38.22 %
(39933)
11.76 %
(12291)
10.77 %
(11256)
39.24 %
(40993)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000

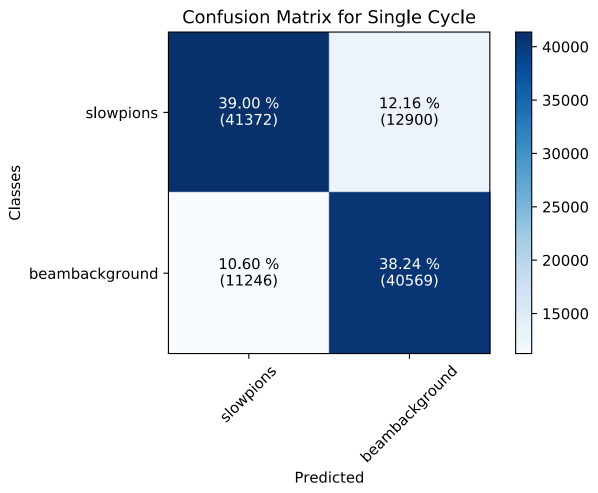

Confusion Matrix for Transposed on Third Layer
slowpions
beambackground
Classes
Predicted
37.85 % (40041)
11.52 % (12183)
11.43 % (12092)
39.21 % (41479)
40000
35000
30000
25000
20000
15000

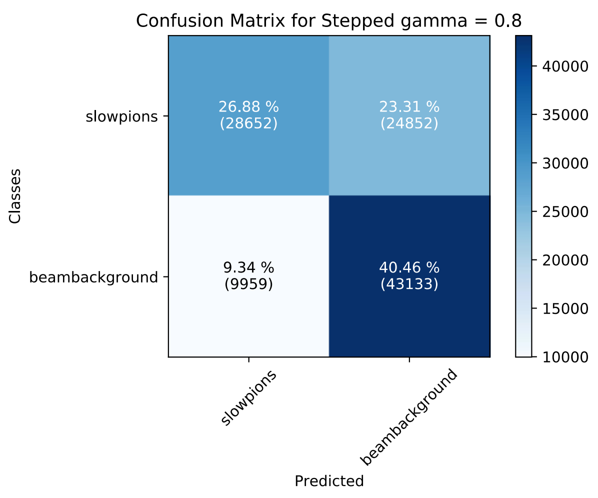

Confusion Matrix for 5 Cycles
slowpions
beambackground
Classes
Predicted
38.37 % (41654)
10.68 % (11594)
11.84 % (12849)
39.12 % (42470)
40000
35000
30000
25000
20000
15000


Confusion Matrix for Exponential gamma = 0.8
slowpions
beambackground
Classes
Predicted
26.88 % (28652)
23.31 % (24852)
9.34 % (9959)
40.46 % (43133)
40000
35000
30000
25000
20000
15000
10000

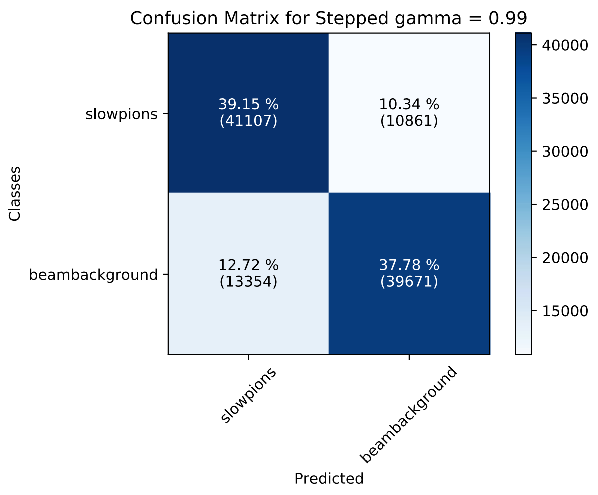

Confusion Matrix for Exponential gamma = 0.99
39.15 % (41107)
10.34 % (10861)
12.72 % (13354)
37.78 % (39671)
slowpions
beambackground
Classes
Predicted
40000
35000
30000
25000
20000
15000


Confusion Matrix for Multi Step gamma = 0.50
33.56 % (283695)
16.47 % (139217)
9.27 % (78369)
40.70 % (344033)
slowpions
everythingelse
Classes
Predicted
300000
250000
200000
150000
100000

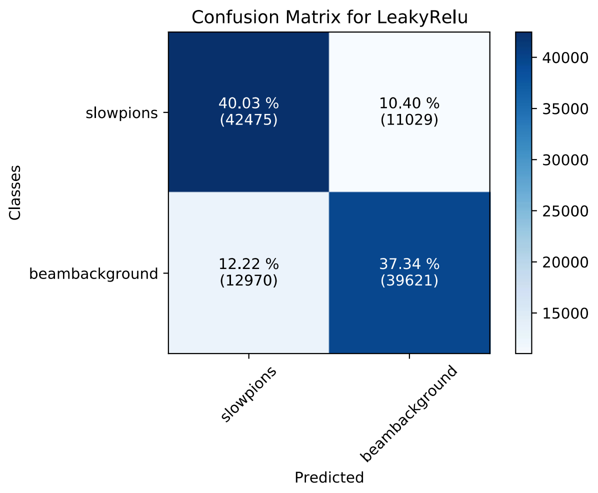

Confusion Matrix for Single Cycle
39.00 % (41372)
12.16 % (12900)
10.60 % (11246)
38.24 % (40569)
slowpions
beambackground
Classes
Predicted
40000
35000
30000
25000
20000
15000



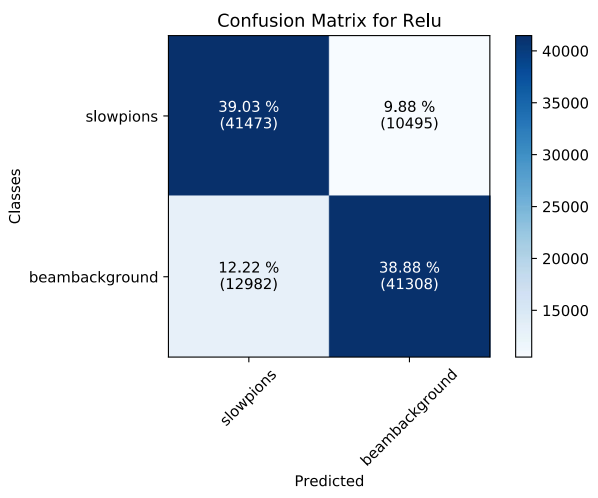
Confusion Matrix for Stepped gamma = 0.8
slowpions
beambackground
Classes
26.88 % (28652)
23.31 % (24852)
9.34 % (9959)
40.46 % (43133)
40000
35000
30000
25000
20000
15000
10000
slowpions
beambackground
Predicted

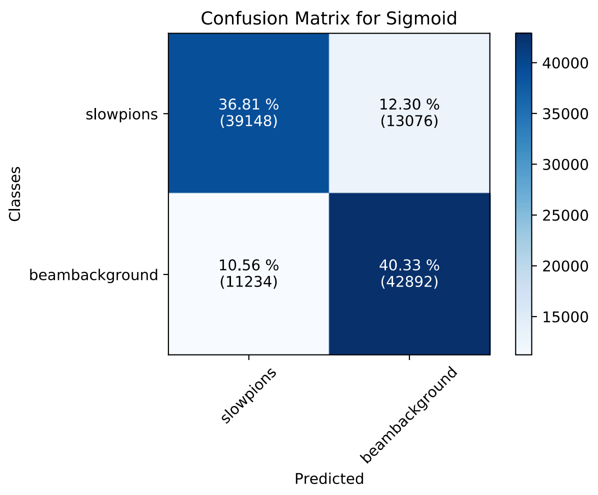


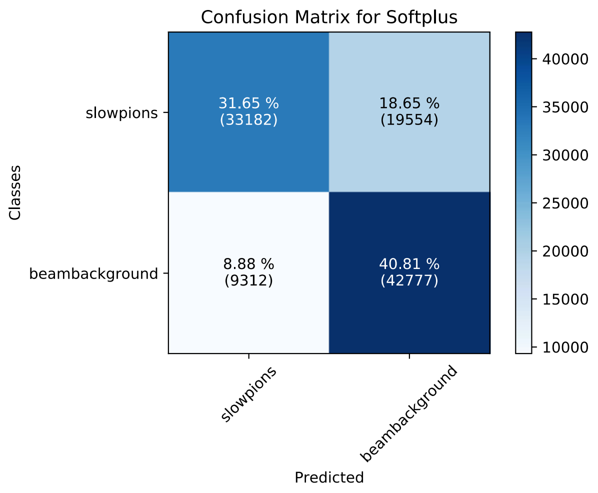
Confusion Matrix for Stepped gamma = 0.99
slowpions
beambackground
Classes
39.15 % (41107)
10.34 % (10861)
12.72 % (13354)
37.78 % (39671)
40000
35000
30000
25000
20000
15000
slowpions
beambackground
Predicted

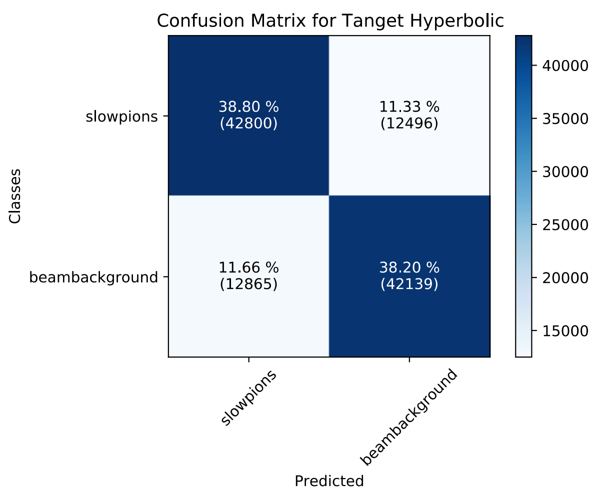


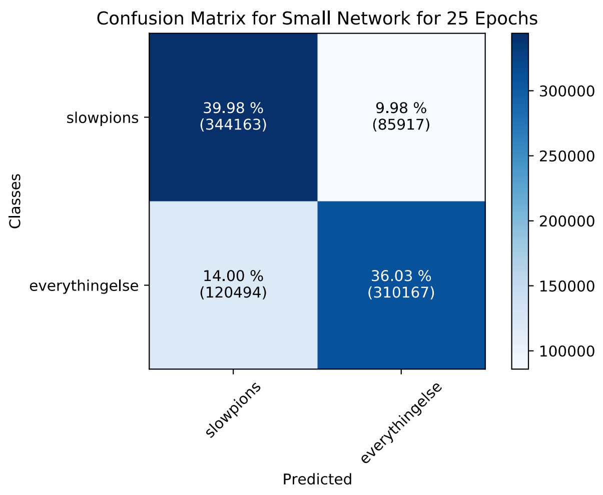
Confusion Matrix for LeakyRelu
slowpions
beambackground
Classes
40.03 % (42475)
10.40 % (11029)
12.22 % (12970)
37.34 % (39621)
40000
35000
30000
25000
20000
15000
slowpions
beambackground
Predicted

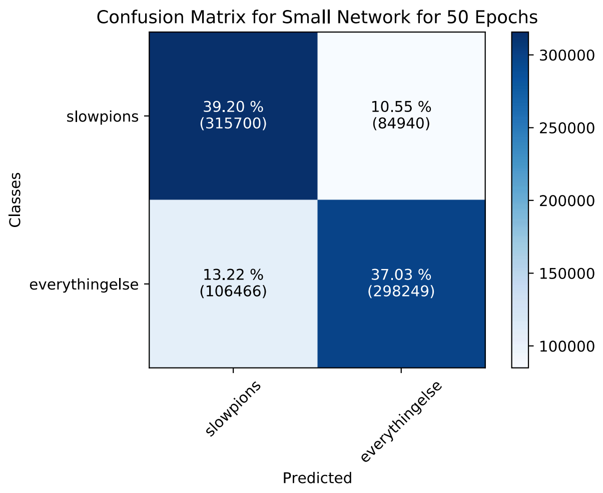

Confusion Matrix for Relu
slowpions
beambackground
Classes
39.03 % (41473)
9.88 % (10495)
12.22 % (12982)
38.88 % (41308)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000

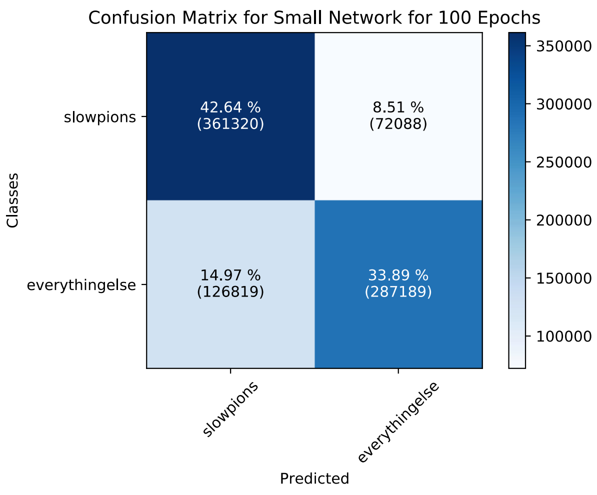

Confusion Matrix for Sigmoid
slowpions
beambackground
Classes
36.81 % (39148)
12.30 % (13076)
10.56 % (11234)
40.33 % (42892)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000


Confusion Matrix for Softplus
slowpions
beambackground
Classes
31.65 % (33182)
18.65 % (19554)
8.88 % (9312)
40.81 % (42777)
slowpions
beambackground
Predicted
40000
35000
30000
25000
20000
15000
10000

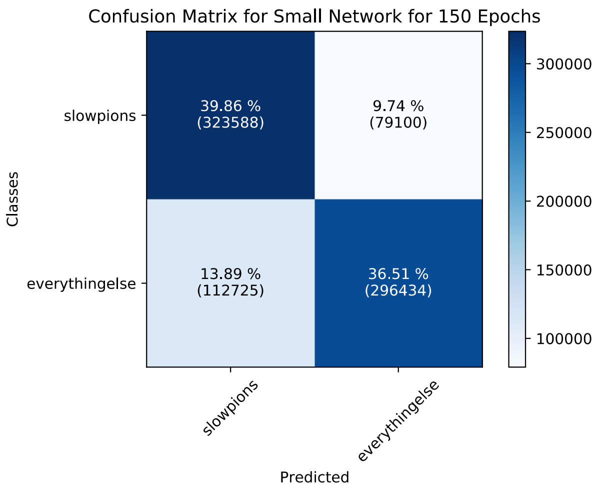

Confusion Matrix for Tanget Hyperbolic
Classes
slowpions
beambackground
38.80 % (42800)
11.33 % (12496)
11.66 % (12865)
38.20 % (42139)
Predicted
slowpions
beambackground
40000
35000
30000
25000
20000
15000


Confusion Matrix for Small Network for 25 Epochs
Classes
slowpions
everythingelse
39.98 % (344163)
9.98 % (85917)
14.00 % (120494)
36.03 % (310167)
Predicted
slowpions
everythingelse
300000
250000
200000
150000
100000

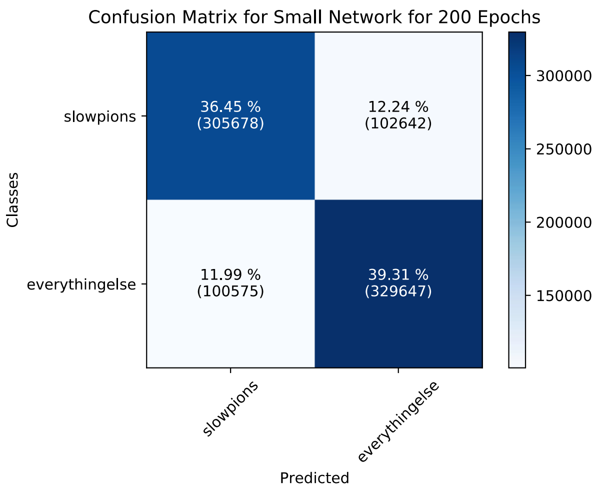

Confusion Matrix for Small Network for 50 Epochs
Classes
slowpions
everythingelse
39.20 % (315700)
10.55 % (84940)
13.22 % (106466)
37.03 % (298249)
Predicted
slowpions
everythingelse
300000
250000
200000
150000
100000



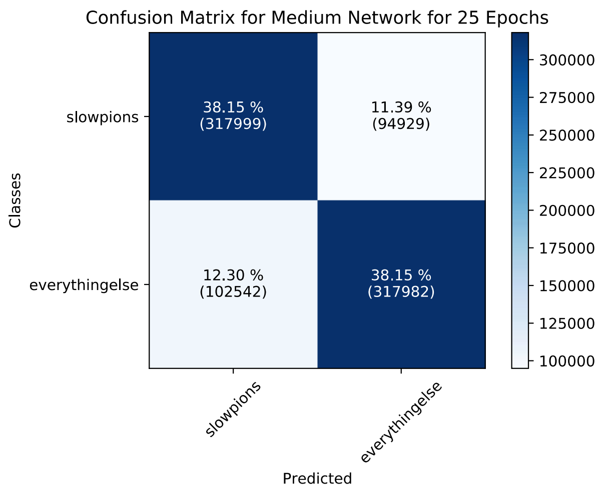
Confusion Matrix for Small Network for 100 Epochs
slowpions
everythingelse
Classes
42.64 %
(361320)
8.51 %
(72088)
14.97 %
(126819)
33.89 %
(287189)
slowpions
everythingelse
Predicted
350000
300000
250000
200000
150000
100000



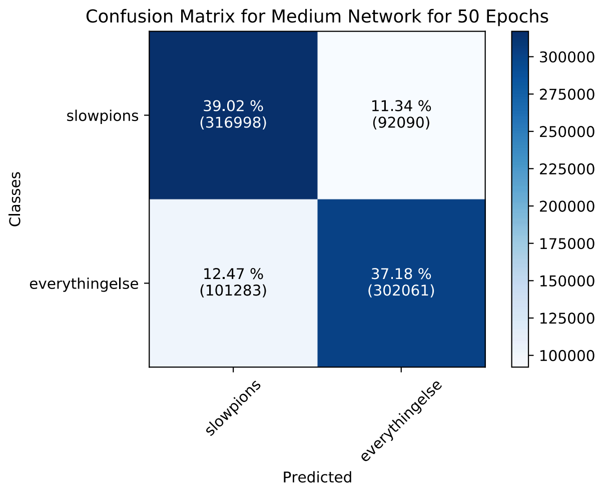
Confusion Matrix for Small Network for 150 Epochs
slowpions
everythingelse
Classes
39.86 %
(323588)
9.74 %
(79100)
13.89 %
(112725)
36.51 %
(296434)
slowpions
everythingelse
Predicted
300000
250000
200000
150000
100000



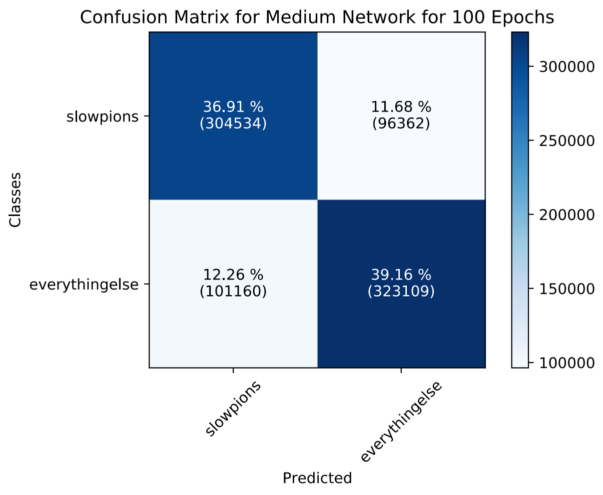
Confusion Matrix for Small Network for 200 Epochs
slowpions
everythingelse
Classes
36.45 %
(305678)
12.24 %
(102642)
11.99 %
(100575)
39.31 %
(329647)
slowpions
everythingelse
Predicted
300000
250000
200000
150000



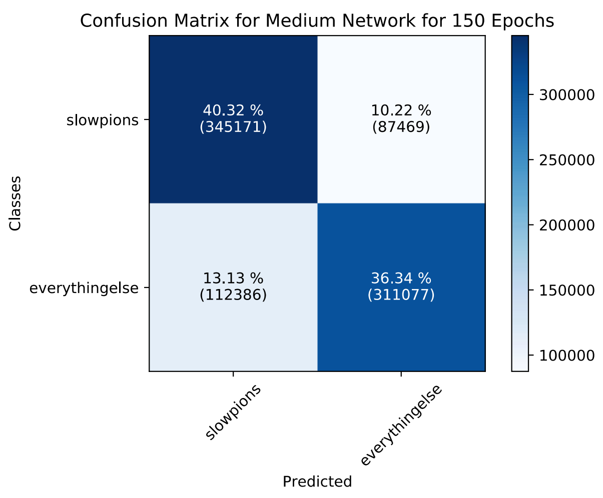
Confusion Matrix for Medium Network for 25 Epochs
slowpions
everythingelse
Classes
38.15 % (317999)
11.39 % (94929)
12.30 % (102542)
38.15 % (317982)
slowpions
everythingelse
Predicted
300000
275000
250000
225000
200000
175000
150000
125000
100000

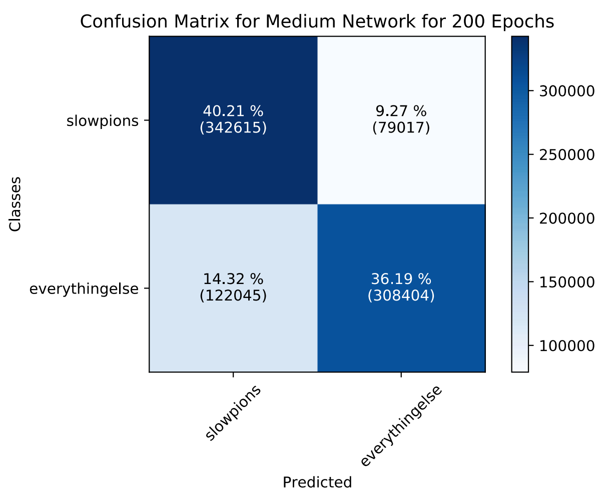


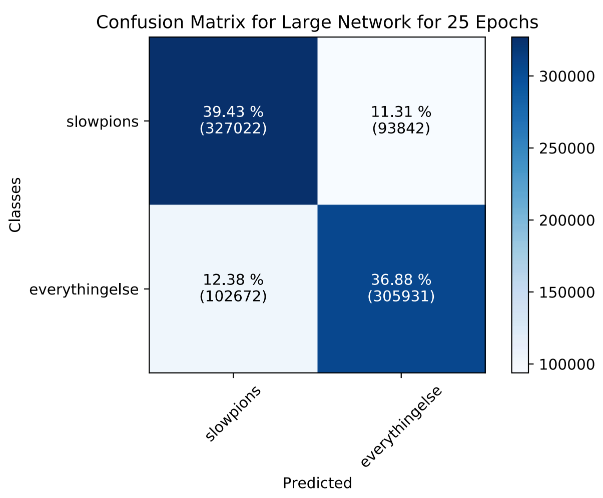
Confusion Matrix for Medium Network for 50 Epochs
slowpions
everythingelse
Classes
39.02 % (316998)
11.34 % (92090)
12.47 % (101283)
37.18 % (302061)
slowpions
everythingelse
Predicted
300000
275000
250000
225000
200000
175000
150000
125000
100000

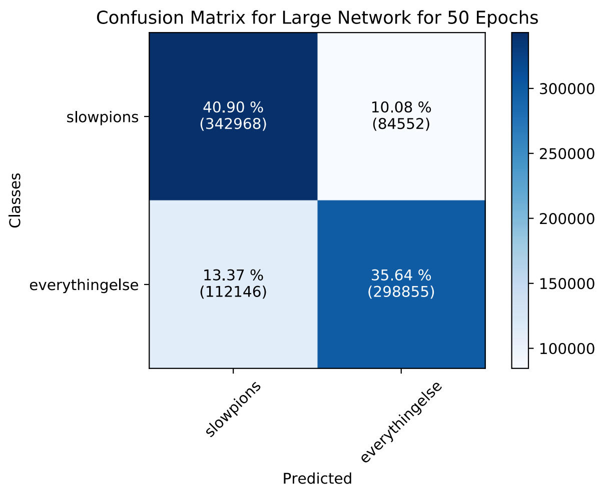


Confusion Matrix for Medium Network for 100 Epochs
slowpions
everythingelse
Classes
36.91 % (304534)
11.68 % (96362)
12.26 % (101160)
39.16 % (323109)
slowpions
everythingelse
Predicted
300000
250000
200000
150000
100000

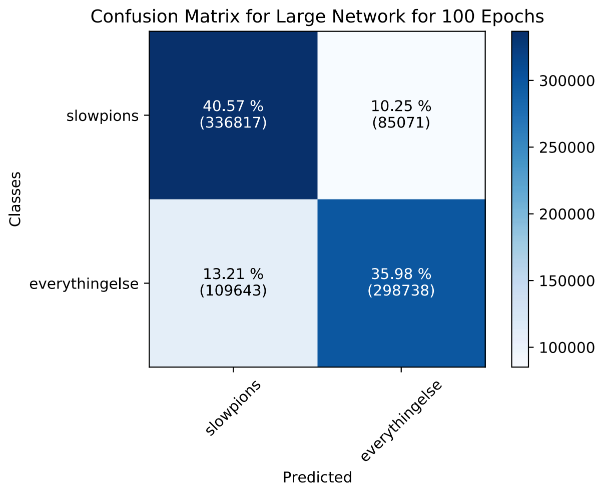

Confusion Matrix for Medium Network for 150 Epochs
Classes
slowpions
everythingelse
40.32 %
(345171)
10.22 %
(87469)
13.13 %
(112386)
36.34 %
(311077)
300000
250000
200000
150000
100000
slowpions
everythingelse
Predicted


Confusion Matrix for Medium Network for 200 Epochs
Classes
slowpions
everythingelse
40.21 %
(342615)
9.27 %
(79017)
14.32 %
(122045)
36.19 %
(308404)
300000
250000
200000
150000
100000
slowpions
everythingelse
Predicted

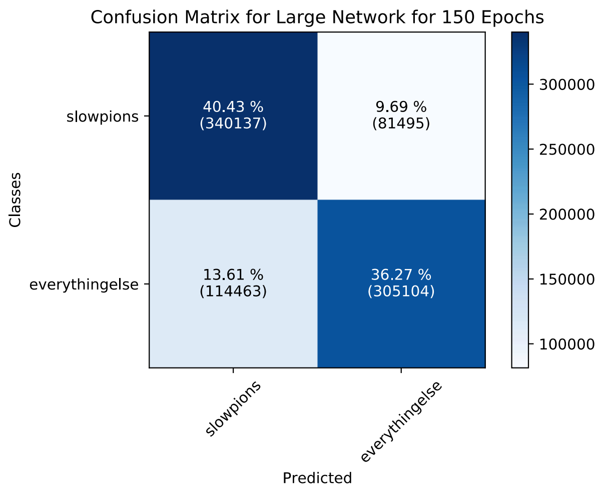

Confusion Matrix for Large Network for 25 Epochs
Classes
slowpions
everythingelse
39.43 %
(327022)
11.31 %
(93842)
12.38 %
(102672)
36.88 %
(305931)
300000
250000
200000
150000
100000
slowpions
everythingelse
Predicted



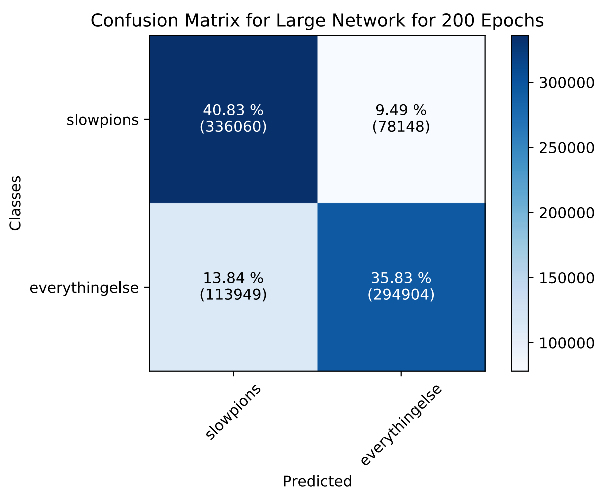
Confusion Matrix for Large Network for 50 Epochs
Classes
slowpions
everythingelse
40.90 %
(342968)
10.08 %
(84552)
13.37 %
(112146)
35.64 %
(298855)
300000
250000
200000
150000
100000
slowpions
everythingelse
Predicted

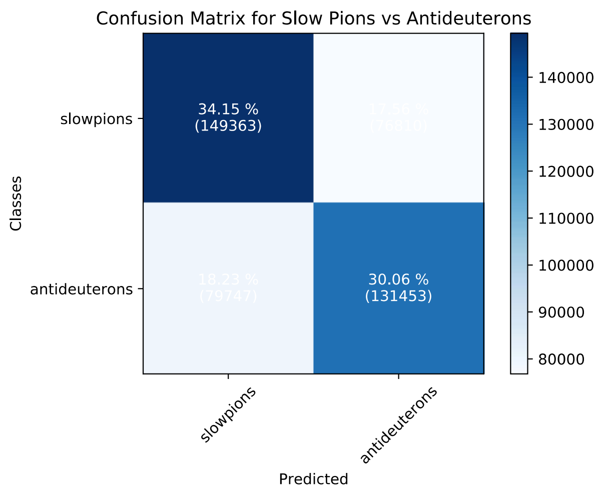


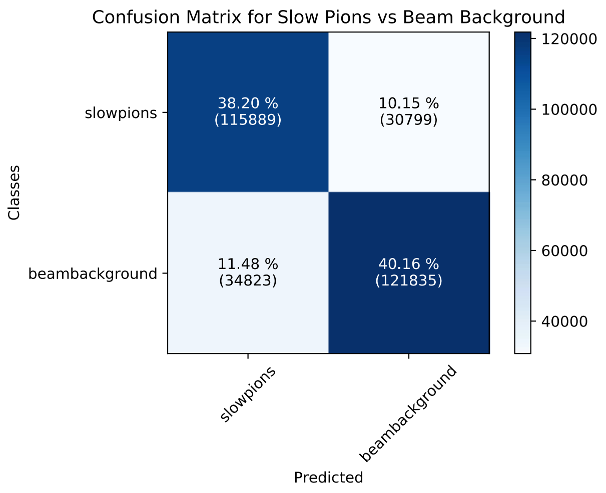
Confusion Matrix for Large Network for 100 Epochs
Classes
slowpions
everythingelse
40.57 %
(336817)
10.25 %
(85071)
13.21 %
(109643)
35.98 %
(298738)
300000
250000
200000
150000
100000
slowpions
everythingelse
Predicted

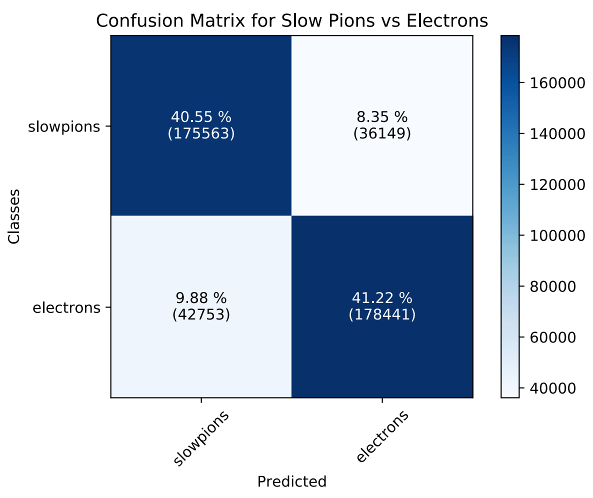


Confusion Matrix for Large Network for 150 Epochs
Classes
slowpions
everythingelse
40.43 %
(340137)
9.69 %
(81495)
13.61 %
(114463)
36.27 %
(305104)
300000
250000
200000
150000
100000
slowpions
everythingelse
Predicted

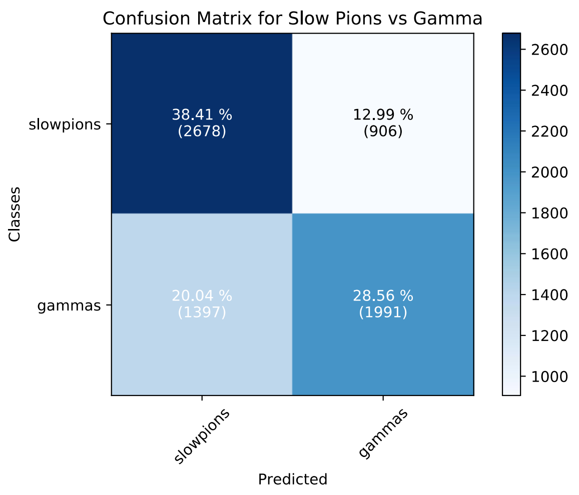

Confusion Matrix for Large Network for 200 Epochs
slowpions
everythingelse
Classes
40.83 %
(336060)
9.49 %
(78148)
13.84 %
(113949)
35.83 %
(294904)
slowpions
everythingelse
Predicted
300000
250000
200000
150000
100000


Confusion Matrix for Slow Pions vs Antideuterons
slowpions
antideuterons
Classes
34.15 %
(149363)
17.56 %
(76810)
18.23 %
(79747)
30.06 %
(131453)
slowpions
antideuterons
Predicted
140000
130000
120000
110000
100000
90000
80000

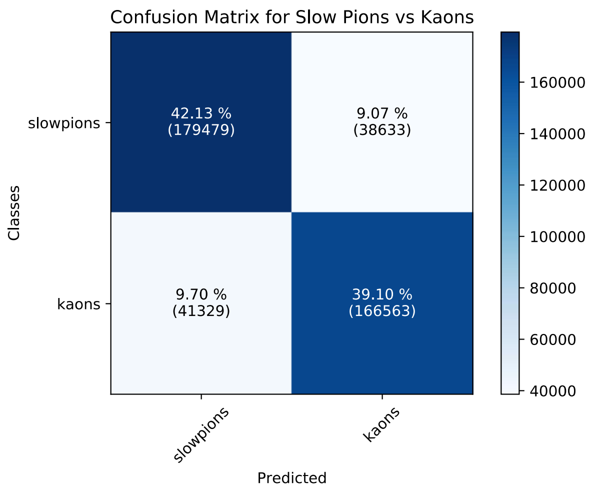

Confusion Matrix for Slow Pions vs Beam Background
slowpions
beambackground
Classes
38.20 %
(115889)
10.15 %
(30799)
11.48 %
(34823)
40.16 %
(121835)
slowpions
beambackground
Predicted
120000
100000
80000
60000
40000



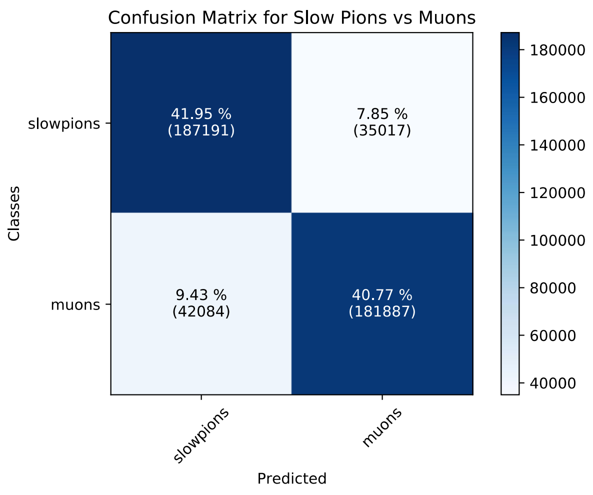
Confusion Matrix for Slow Pions vs Electrons
40.55 % (175563)
8.35 % (36149)
9.88 % (42753)
41.22 % (178441)
slowpions
electrons
Classes
Predicted
160000
140000
120000
100000
80000
60000
40000

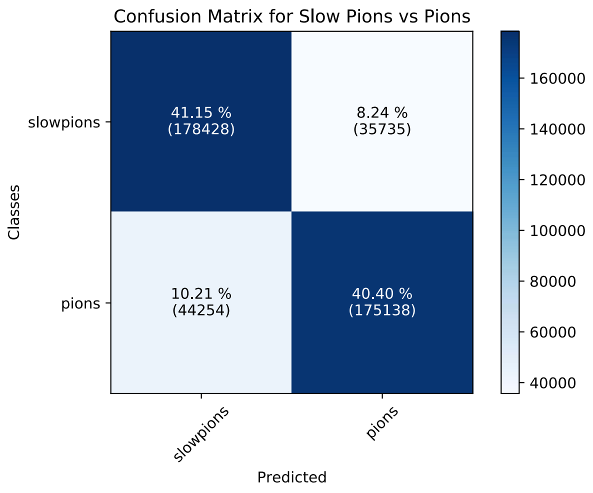


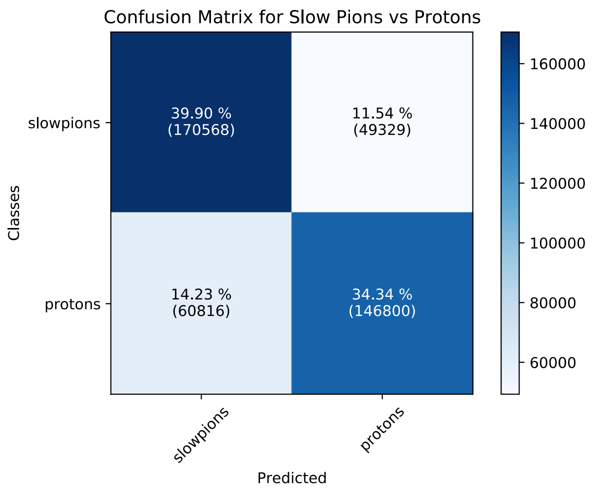
Confusion Matrix for Slow Pions vs Gamma
38.41 % (2678)
12.99 % (906)
20.04 % (1397)
28.56 % (1991)
slowpions
gammas
Classes
Predicted
2600
2400
2200
2000
1800
1600
1400
1200
1000

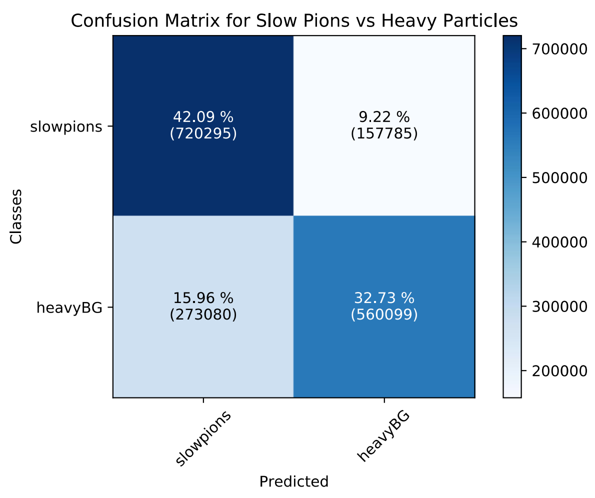


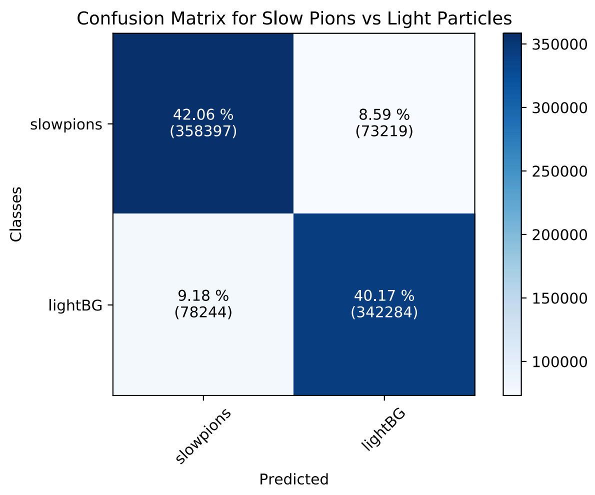
Confusion Matrix for Slow Pions vs Kaons
42.13 % (179479)
9.07 % (38633)
9.70 % (41329)
39.10 % (166563)
slowpions
kaons
Classes
Predicted
160000
140000
120000
100000
80000
60000
40000

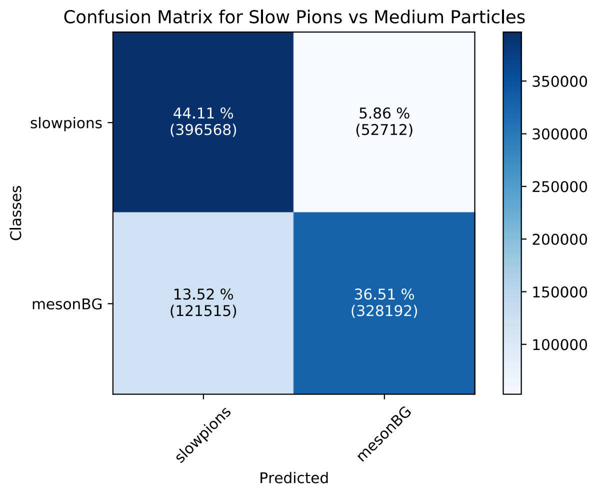

Confusion Matrix for Slow Pions vs Muons
slowpions
muons
Classes
41.95 % (187191)
7.85 % (35017)
9.43 % (42084)
40.77 % (181887)
slowpions
muons
Predicted
180000
160000
140000
120000
100000
80000
60000
40000

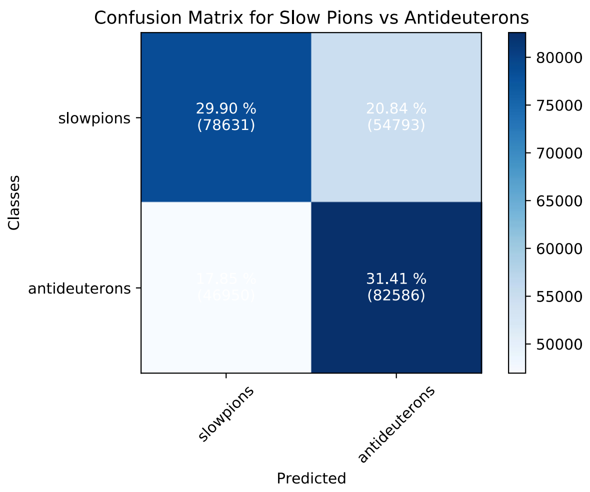

Confusion Matrix for Slow Pions vs Pions
slowpions
pions
Classes
41.15 % (178428)
8.24 % (35735)
10.21 % (44254)
40.40 % (175138)
slowpions
pions
Predicted
160000
140000
120000
100000
80000
60000
40000

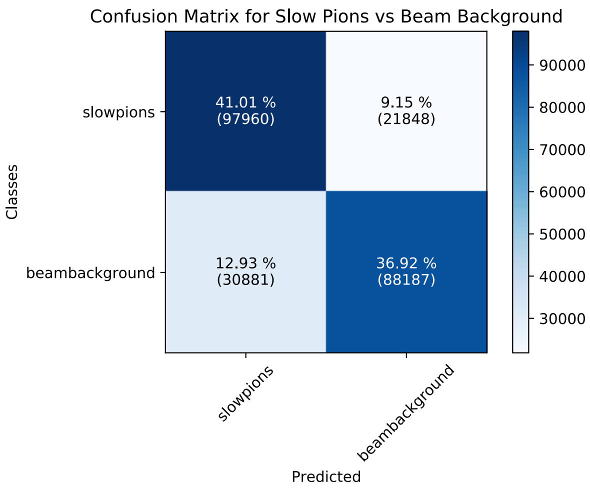

Confusion Matrix for Slow Pions vs Protons
slowpions
protons
Classes
39.90 % (170568)
11.54 % (49329)
14.23 % (60816)
34.34 % (146800)
slowpions
protons
Predicted
160000
140000
120000
100000
80000
60000

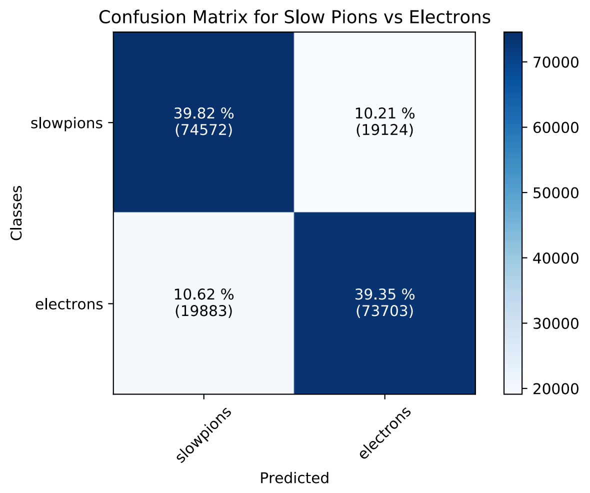

Confusion Matrix for Slow Pions vs Heavy Particles
slowpions
heavyBG
Classes
42.09 % (720295)
9.22 % (157785)
15.96 % (273080)
32.73 % (560099)
slowpions
heavyBG
Predicted
700000
600000
500000
400000
300000
200000

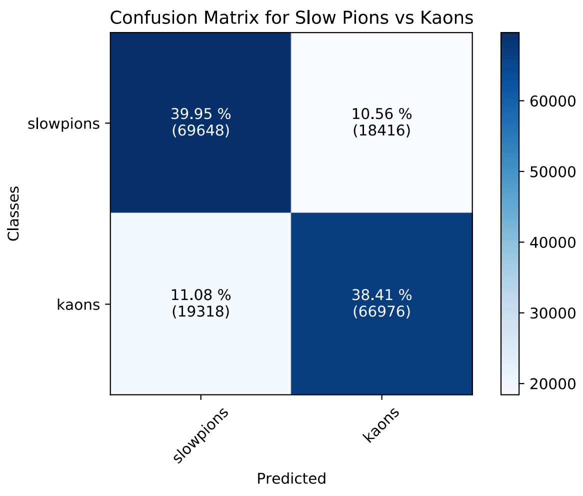

Confusion Matrix for Slow Pions vs Light Particles
slowpions
lightBG
Classes
42.06 % (358397)
8.59 % (73219)
9.18 % (78244)
40.17 % (342284)
slowpions
lightBG
Predicted
350000
300000
250000
200000
150000
100000

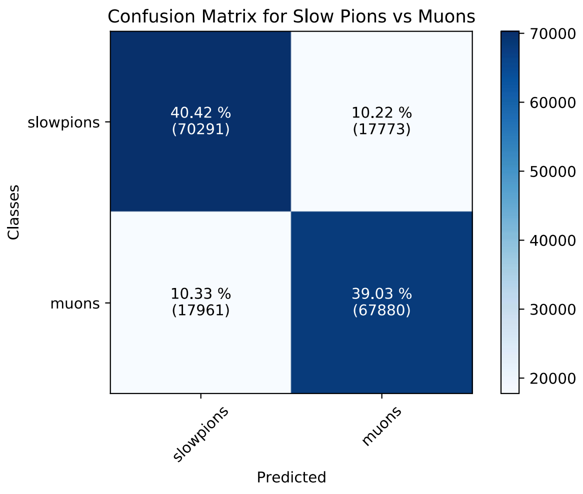

Confusion Matrix for Slow Pions vs Medium Particles
slowpions
mesonBG
Classes
44.11 % (396568)
5.86 % (52712)
13.52 % (121515)
36.51 % (328192)
slowpions
mesonBG
Predicted
350000
300000
250000
200000
150000
100000

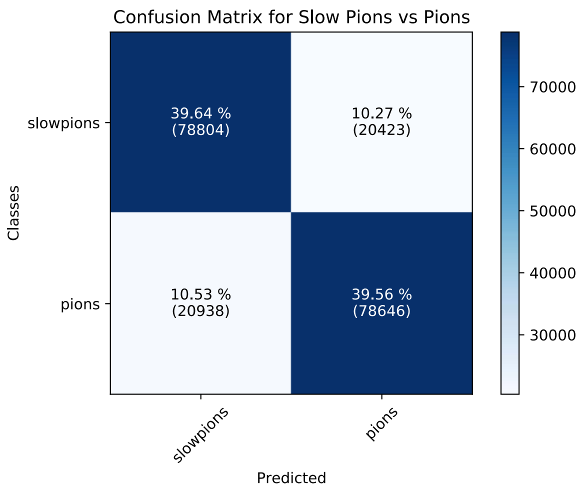

Confusion Matrix for Slow Pions vs Antideuterons
Classes
slowpions
antideuterons
29.90 %
(78631)
20.84 %
(54793)
17.85 %
(46950)
31.41 %
(82586)
slowpions
antideuterons
Predicted
80000
75000
70000
65000
60000
55000
50000

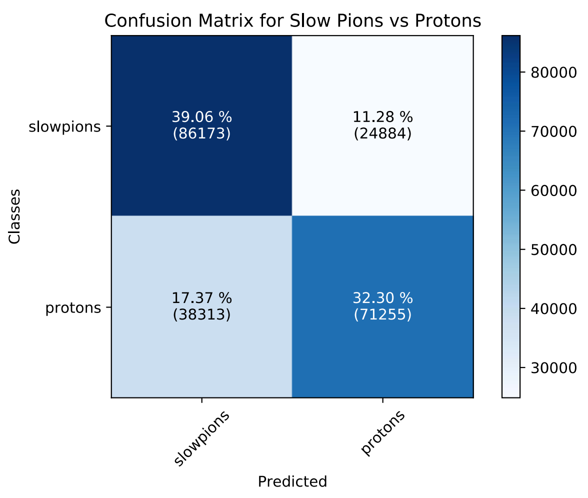

Confusion Matrix for Slow Pions vs Beam Background
Classes
slowpions
beambackground
41.01 %
(97960)
9.15 %
(21848)
12.93 %
(30881)
36.92 %
(88187)
slowpions
beambackground
Predicted
90000
80000
70000
60000
50000
40000
30000


Confusion Matrix for Slow Pions vs Electrons
Classes
slowpions
electrons
39.82 %
(74572)
10.21 %
(19124)
10.62 %
(19883)
39.35 %
(73703)
slowpions
electrons
Predicted
70000
60000
50000
40000
30000
20000

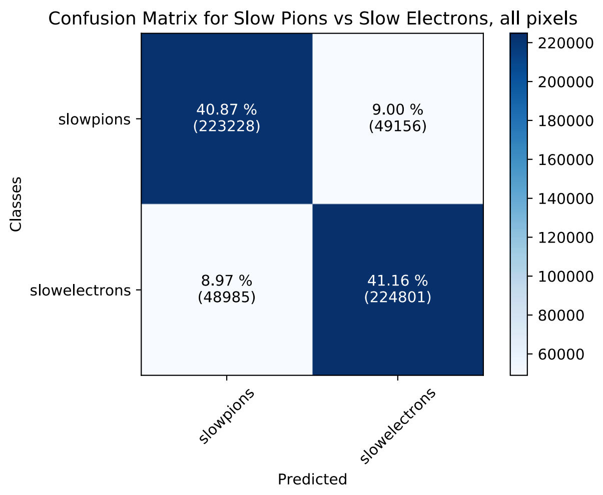

Confusion Matrix for Slow Pions vs Kaons
slowpions
kaons
Classes
39.95 % (69648)
10.56 % (18416)
11.08 % (19318)
38.41 % (66976)
slowpions
kaons
Predicted
60000
50000
40000
30000
20000


Confusion Matrix for Slow Pions vs Muons
slowpions
muons
Classes
40.42 % (70291)
10.22 % (17773)
10.33 % (17961)
39.03 % (67880)
slowpions
muons
Predicted
70000
60000
50000
40000
30000
20000

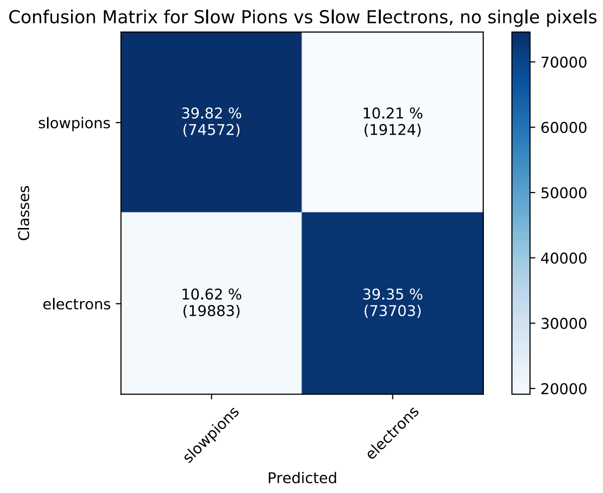

Confusion Matrix for Slow Pions vs Pions
slowpions
pions
Classes
39.64 % (78804)
10.27 % (20423)
10.53 % (20938)
39.56 % (78646)
slowpions
pions
Predicted
70000
60000
50000
40000
30000



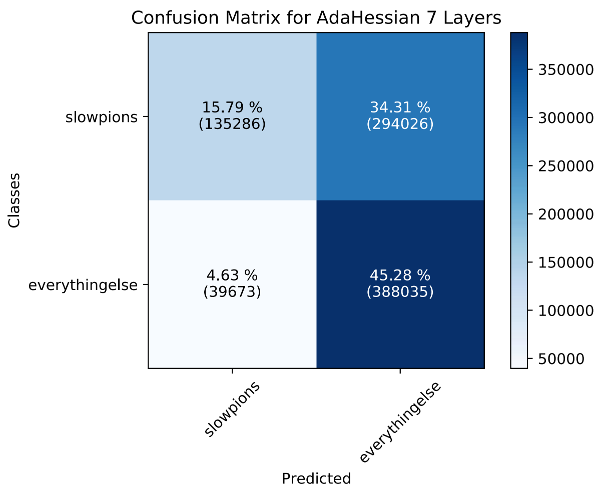
Confusion Matrix for Slow Pions vs Protons
slowpions
protons
Classes
Predicted
39.06 % (86173)
11.28 % (24884)
17.37 % (38313)
32.30 % (71255)
80000
70000
60000
50000
40000
30000



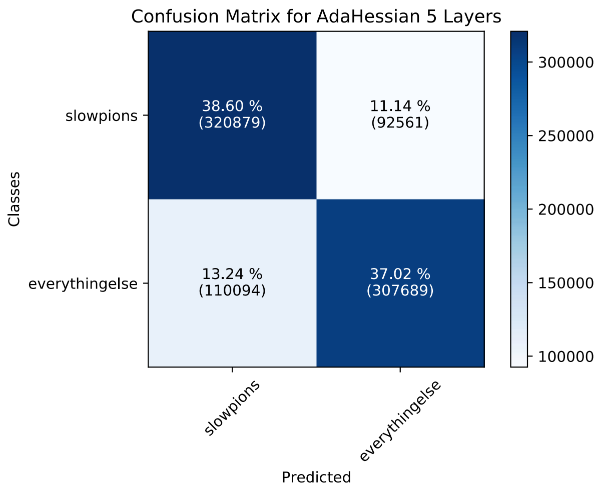
Confusion Matrix for Slow Pions vs Slow Electrons, all pixels
slowpions
slowelectrons
Classes
Predicted
40.87 % (223228)
9.00 % (49156)
8.97 % (48985)
41.16 % (224801)
220000
200000
180000
160000
140000
120000
100000
80000
60000



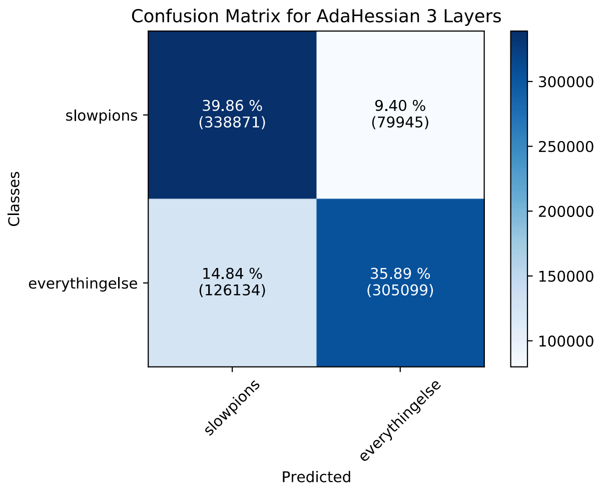
Confusion Matrix for Slow Pions vs Slow Electrons, no single pixels
slowpions
electrons
Classes
Predicted
39.82 % (74572)
10.21 % (19124)
10.62 % (19883)
39.35 % (73703)
70000
60000
50000
40000
30000
20000



Confusion Matrix for AdaHessian 7 Layers
slowpions
everythingelse
Classes
15.79 % (135286)
34.31 % (294026)
4.63 % (39673)
45.28 % (388035)
slowpions
everythingelse
Predicted
350000
300000
250000
200000
150000
100000
50000



Confusion Matrix for AdaHessian 5 Layers
slowpions
everythingelse
Classes
38.60 % (320879)
11.14 % (92561)
13.24 % (110094)
37.02 % (307689)
slowpions
everythingelse
Predicted
300000
250000
200000
150000
100000



Confusion Matrix for AdaHessian 3 Layers
slowpions
everythingelse
Classes
39.86 % (338871)
9.40 % (79945)
14.84 % (126134)
35.89 % (305099)
slowpions
everythingelse
Predicted
300000
250000
200000
150000
100000

B. The Code Base

The Main Code

```
import torch
from torch import nn
from torch import optim
from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader
from matplotlib import pyplot as plt
import numpy as np
import sys
import argparse
import time
import progress.bar as pb
from pathlib import Path
from helper import *
import math
from prettytable import PrettyTable
#import torch_optimizer as Toptimizers
import configparser


""""
 .d8888b.           888                                .d8888b.  8888888b.  888     888
d8b                            888          .d888 888
888
d88P  Y88b          888                               d88P  Y88b 888   Y88b 888     888
Y8P                            888         d88P"  888
888
Y88b.               888                               888    888 888    888 888     888
888         888    888                                              888
 "Y888b.    .d88b.  888888 888  888 88888b.           888        888   d88P 888     888
888 88888b.  88888b.  888  888 888888      888888 888
8888b.   .d88b.  .d8888b            .d88b.  888888 .d8888b
    "Y88b. d8P  Y8b 888    888  888 888 "88b          888  88888 888888P"   888     888
888 888 "88b 888 "88b 888  888 888         888    888     "88b d88P"88b 88K               d8P
Y8b 888   d88P"
      "888 88888888 888    888  888 888  888          888    888 888        888     888
888 888  888 888  888 888  888 888         888    888 .d888888 888  888 "Y8888b.
88888888 888   888
Y88b  d88P Y8b.     Y88b.  Y88b 888 888 d88P d8b      Y88b  d88P 888        Y88b. .d88P d8b
888 888  888 888 d88P Y88b 888 Y88b.       888    888 888  888 Y88b 888      X88 d8b      Y8b.
Y88b. Y88b.    d8b
 "Y8888P"   "Y8888   "Y888  "Y88888 88888P"  88P       "Y8888P88 888         "Y88888P"  88P
888 888  888 88888P"   "Y88888  "Y888      888    888 "Y888888  "Y88888  88888P' 88P
"Y8888   "Y888 "Y8888P Y8P
                                    888      8P                                         8P
888                                                    888          8P
                                    888      "                                          "
888                                               Y8b d88P          "
                                    888
888                                                "Y88P"
"""

startStart = time.time()
# input flags definieren, help sollte alles erklären
parser = argparse.ArgumentParser()
parser.add_argument("-i", "--infile", help="define name of settings file", type = str)
parser.add_argument("-o", "--outfile", help="define name of output files", type = str)
parser.add_argument("-b", "--batchSize", help="sets the batchsize", type = int)
parser.add_argument("-l", "--learnRate", help="sets the learn rate", type = float)
parser.add_argument("-m", "--momentum", help="sets the momentum", type = float)
parser.add_argument("-e", "--epochs", help="sets the number of epochs", type = int)
parser.add_argument("-k", "--kFold", help="sets the of k-folds", type = int)
parser.add_argument("-w", "--weightDecay", help="sets the weight decay for optimizer", type =
float)

parser.add_argument("--scheduler", help="define the scheduler used", type=str)
parser.add_argument("-g", "--gamma", help="factor by which learnRate is reduced", type =
float)
parser.add_argument("-s", "--stepSize", help="step size with which to reduce learnRate", type
= float)
```

```
parser.add_argument("--milestones", help="sets the milestones at which learn rate should
change", nargs='*', type = int)
parser.add_argument("--learnMax", help="the maximum learnRate for lambda scheduler", type =
float)
parser.add_argument("--learnMin", help="the minimum learnRate for lambda scheduler", type =
float)
parser.add_argument("--learnPeak", help="the epoch of learnRate peak for lambda scheduler",
type = int)

parser.add_argument("--nesterov", help="switches SGD to the Nesterov variant",
action='store_true')
parser.add_argument("--rho", help="coefficient for running average of squared gradients", type
= float)
parser.add_argument("--eps", help="numerical stability constant for optimizer", type = float)
parser.add_argument("--alpha", help="smoothing constant for RMSprop", type = float)
parser.add_argument("--learnRateDecay", help="determines the falloff for adagrad learnrate",
type = float)
parser.add_argument("--beta", help="runnung average gradient coefficients for adam",
nargs='*', type = float)

parser.add_argument("--datapath", help="sets where the data to be analyzed are stored", type =
str)
parser.add_argument("--nosinglepixels", help="exclude single pixel events",
action='store_true')

parser.add_argument("-sf", "--setFactor", help="sets a factor for the total amount of data per
set", type = float)
parser.add_argument("--balanced", help="should all data sets be about the same size?",
action='store_false')
parser.add_argument("-d", "--device", help="sets the processing device {cpu, cuda}", type =
str)
parser.add_argument("-t", "--threads", help="sets the number of processes", type = int)
parser.add_argument("-c", "--categories", help="specify the train/valid categories {dd, pi,
pp, sp, bp, bg, test}", type = str)
parser.add_argument("--retrain", help="force to retrain the net", action='store_true')
parser.add_argument("--save", help="save output data", action='store_true')
parser.add_argument("--optim", help="define the optimizer used", type=str)
parser.add_argument("-ll", "--linLayer", nargs='*', type = int)
parser.add_argument("-do", "--dropout", nargs='*', type = int)
parser.add_argument("-ch", "--channels", nargs='*', type = int)
parser.add_argument("-ks", "--kernelSize", nargs='*', type = int)
parser.add_argument("-pd", "--padding", nargs='*', type = int)
parser.add_argument("-al", "--actilin", nargs='*', type = str)
parser.add_argument("-ac", "--acticonv", nargs='*', type = str)
args = parser.parse_args()


# input file file lesen
# input files bestehen aus zwei sections: [SETTINGS] & [NETWORK]
# in diesen sections sind dann unter den schlüsselwörtern die entsprechenden einstellungen zu
hinterlegen
config = configparser.ConfigParser()
if Path("{}".format(args.infile)).is_file():
    config.read(args.infile)
    print("reading input file {}".format(args.infile))
else:
    print("couldn't find the specified file {}".format(args.infile))
    config['SETTINGS'] = {} # um einen keyerror zuverhindern erstelle ich ein leeres dict
    config['NETWORK'] = {}


# parsen der input flags, defaults definieren ...
# ein paar informationen printen
settings = settingsClass()
network = networkClass()
print("programm start...")
print("the following settings will be used:")
for key in settings.list():
    try:
        inputValue = getattr(args, key) # input flag wert lesen
    except AttributeError:
        continue
    value = settings.getValue(key) # default wert laden
    if inputValue != None and inputValue != value:
        settings.setValue(key, inputValue) # input wert setzen
        print("\tset", key, "to", inputValue, "from input flags")
    elif key in config['SETTINGS']:
        readValue = config['SETTINGS'][key] # wert der input file lesen
        readType = getType(readValue) # daten typ bestimmen
        if readType == int:
            settings.setValue(key, int(readValue))
        elif readType == float:
            settings.setValue(key, float(readValue))
        elif readType == bool:
            readBool = config['SETTINGS'].getboolean(key)
            settings.setValue(key, readBool)
        else:
            settings.setValue(key, readValue)
        print("\tset", key, "to", readValue, "from input file: {}".format(args.infile))
    else:
        print("\tlet", key, "at", value, "from default settings")

# hier wird die save flag gesetzt, sobald ein output name angegeben wird
if settings.outfile != "default name":
    settings.save = True


"""
8888888                                         888    d8b
8888888b.           888
  888                                           888    Y8P
888  "Y88b          888
  888                                           888
888    888          888
  888   88888b.d88b.  88888b.   .d88b.  888d888 888888 888  .d88b.  888d888 .d88b.  88888b.
888  888  .d88b.  88888b.       888    888  8888b.  888888 .d88b.  88888b.
  888   888 "888 "88b 888 "88b d88""88b 888P"   888    888 d8P  Y8b 888P"  d8P  Y8b 888 "88b
888  888 d88""88b 888 "88b      888    888     "88b 888   d8P  Y8b 888 "88b
  888   888  888  888 888  888 888  888 888     888    888 8888888 888    8888888 888  888
Y88  88P 888  888 888  888      888    888 .d888888 888   8888888 888  888
  888   888  888  888 888 d88P Y88..88P 888     Y88b.  888 Y8b.     888    Y8b.     888  888
Y8bd8P  Y88..88P 888  888      888  .d88P 888  888 Y88b. Y8b.     888  888
8888888 888  888  888 88888P"   "Y88P"  888      "Y888 888  "Y8888  888     "Y8888  888  888
Y88P    "Y88P"  888  888      8888888P"  "Y888888  "Y888 "Y8888  888  888
                      888
                      888
                      888
"""

# kategorien parsen, namen, kürzel in string arrays schreiben
# background ist antideuterons, pions und protons gestapelt
settings.catNames()
numbers = {} # hier werden alle numerischen parameter gespeichert
numbers["numCategories"] = len(settings.categoryNames)
outputs = outputsClass(numbers["numCategories"], settings.device) # outputs einrichten

start = time.time()
# dateien/datensätze importieren
# es gibt zwei datensätze:
# - der normale pixel daten satz
# - und ein evaluierungs datensatz, zum testen des netzes
imported = importer(settings.categoryNames, numbers, settings.datapath,
settings.nosinglepixels)
train, valid = importToTensor(imported, settings.categoryNames, settings.balanced,
settings.setFactor, settings.kFold, settings.batchSize, numbers)

end = time.time()
print("\nfinished importing, it took {0:.1f} seconds\n".format(end-start))

# numbers dict enthält sämtliche parameter des datensatzes, für das netz und zeiten ...
numbers["importTime"] = end-start


# tabelle erstellen, wie viele daten enthalten sind und für was verwendet werden
tableMetrics = PrettyTable()
tableMetrics.field_names = ["Data Set", "Total", "Training", "Validation"]
for name in settings.categoryNames:
    tableMetrics.add_row([name,
numbers["{}train".format(name)]+numbers["{}valid".format(name)],
numbers["{}train".format(name)], numbers["{}valid".format(name)]])
tableMetrics.add_row(["Total", numbers["total"], numbers["alltrain"], numbers["allvalid"]])
print(tableMetrics)
```

```
"""
888b    888                                                     888
888b    888          888                   .d8888b.           888
8888b   888                                                     888
8888b   888          888                  d88P  Y88b          888
88888b  888                                                     888
88888b  888          888                  Y88b.               888
888Y88b 888  .d88b.  888  888 888d888 .d88b.  88888b.   8888b.  888  .d88b.  .d8888b
888Y88b 888  .d88b.  888888 88888888       "Y888b.    .d88b.  888888 888  888 88888b.
888 Y88b888 d8P  Y8b 888  888 888P"  d88""88b 888 "88b     "88b 888 d8P  Y8b 88K           888
Y88b888 d8P  Y8b 888       d88P            "Y88b. d8P  Y8b 888    888  888 888 "88b
888  Y88888 88888888 888  888 888    888  888 888  888 .d888888 888 88888888 "Y8888b.      888
Y88888 88888888 888      d88P              "888 88888888 888    888  888 888  888
888   Y8888 Y8b.     Y88b 888 888    Y88..88P 888  888 888  888 888 Y8b.          X88       888
Y8888 Y8b.     Y88b.   d88P         Y88b  d88P Y8b.     Y88b.  Y88b 888 888 d88P
888    Y888  "Y8888   "Y88888 888     "Y88P"  888  888 "Y888888 888  "Y8888   88888P'      888
Y888  "Y8888   "Y888 88888888       "Y8888P"   "Y8888   "Y888  "Y88888 88888P"

888

888

888
"""

start = time.time()
print("\nsetting up neural net...")

# print der netzwerk parameter
# parsen der infile und input flags
for key in network.list():
    inputValue = getattr(args, key)
    value = network.getValue(key)
    if inputValue != None and inputValue != value:
        network.setValue(key, inputValue)
        print("\tread", key, "to", inputValue, "from flags")
    elif key in config['NETWORK']:
        readValue = config['NETWORK'][key].split(',') # parsen der infile
        # daten type entsprechend anpassen
        if key == "dropout":
            printList = list(map(float, readValue))
            network.setValue(key, printList)
        elif key == "actilin" or key == 'acticonv':
            newList = []
            printList = readValue
            for word in printList:
              newList.append(word.strip())
            network.setValue(key, newList)
        else:
            printList = list(map(int, readValue))
            network.setValue(key, printList)
        print("\tread", key, "to", printList, "from {}".format(args.infile))
    else:
        print("\tleft", key, "at", value, "from defaults")

network.test(numbers["numCategories"]) # netzwerk paramter anpassen, sodass sie zusammen
passen.
print("the network will have the following parameters:")
for key in network.list():
    print("\t", key, network.getValue(key))

net = pixelNet(linLayers=network.linLayer, dropout=network.dropout, actilin=network.actilin,
channels=network.channels, kernels=network.kernelSize, pads=network.padding,
acticonv=network.acticonv)

# verlustfunktion und optimizer definieren
# scheduler passt die lernrate über die epochen an
lossFunc = nn.CrossEntropyLoss()
print("using CrossEntropyLoss as loss function")

if settings.optim == 'AdaHessian':
  optimizer = Toptimizers.AdaHessian(net.parameters(), lr=settings.learnRate,
weight_decay=settings.weightDecay, eps=settings.eps, betas=(settings.beta[0],
settings.beta[1]), hessian_power=1.0)
```

```
  print("set optimizer to {}".format(optimizer.__class__.__name__))
  correctOptim = True
elif settings.optim == 'SGD' or settings.optim == 'sgd':
  optimizer = optim.SGD(net.parameters(), lr=settings.learnRate, momentum=settings.momentum,
weight_decay=settings.weightDecay, dampening=settings.dampening, nesterov=settings.nesterov)
  print("set optimizer to {}".format(optimizer.__class__.__name__))
  correctOptim = True
elif settings.optim == 'Adagrad' or settings.optim == 'adagrad':
  optimizer = optim.Adagrad(net.parameters(), lr=settings.learnRate,
lr_decay=settings.learnRateDecay, weight_decay=settings.weightDecay, eps=settings.eps)
  print("set optimizer to {}".format(optimizer.__class__.__name__))
  correctOptim = True
elif settings.optim == 'Adadelta' or settings.optim == 'adadelta':
  optimizer = optim.Adadelta(net.parameters(), lr=settings.learnRate,
weight_decay=settings.weightDecay, eps=settings.eps, rho=settings.rho)
  print("set optimizer to {}".format(optimizer.__class__.__name__))
  correctOptim = True
elif settings.optim == 'RMSprop' or settings.optim == 'rmsprop':
  optimizer = optim.RMSprop(net.parameters(), lr=settings.learnRate,
weight_decay=settings.weightDecay, eps=settings.eps, momentum=settings.momentum,
alpha=settings.alpha)
  print("set optimizer to {}".format(optimizer.__class__.__name__))
  correctOptim = True
elif settings.optim == 'Adam' or settings.optim == 'adam':
  optimizer = optim.Adam(net.parameters(), lr=settings.learnRate,
weight_decay=settings.weightDecay, eps=settings.eps, betas=(settings.beta[0],
settings.beta[1]))
  print("set optimizer to {}".format(optimizer.__class__.__name__))
  correctOptim = True
else:
  optimizer = optim.SGD(net.parameters(), lr=settings.learnRate, momentum=settings.momentum,
weight_decay=settings.weightDecay, dampening=settings.dampening, nesterov=settings.nesterov)
  print("could not understand input, so set optimizer to SGD")
  correctOptim = False

if settings.scheduler == 'step':
  scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=settings.stepSize,
gamma=settings.gamma)
  print("set scheduler to {}".format(scheduler.__class__.__name__))
elif settings.scheduler == 'multistep':
  scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=settings.milestones,
gamma=settings.gamma)
  print("set scheduler to {}".format(scheduler.__class__.__name__))
elif settings.scheduler == 'exponential':
  scheduler = optim.lr_scheduler.ExponentialLR(optimizer, gamma=settings.gamma)
  print("set scheduler to {}".format(scheduler.__class__.__name__))
elif settings.scheduler == 'reduce':
  scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer)
  print("set scheduler to {}".format(scheduler.__class__.__name__))
elif settings.scheduler == 'cycle':
  scheduler = optim.lr_scheduler.OneCycleLR(optimizer, settings.learnMax,
total_steps=settings.epochs, pct_start=settings.learnPeak/settings.epochs,
div_factor=settings.learnMax/settings.learnRate,
final_div_factor=settings.learnRate/settings.learnMin)
  print("set scheduler to {}".format(scheduler.__class__.__name__))
elif settings.scheduler == 'cycles':
  if optimizer.__class__.__name__ == 'Adam' or optimizer.__class__.__name__ == 'Adahessian':
    scheduler = optim.lr_scheduler.CyclicLR(optimizer, base_lr=settings.learnRate,
max_lr=settings.learnMax, step_size_up=int(settings.epochs/settings.cycles),
gamma=settings.gamma, cycle_momentum=False)
  else:
    scheduler = optim.lr_scheduler.CyclicLR(optimizer, base_lr=settings.learnRate,
max_lr=settings.learnMax, step_size_up=int(settings.epochs/settings.cycles),
gamma=settings.gamma, cycle_momentum=True)
  print("set scheduler to {}".format(scheduler.__class__.__name__))
else:
  scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=settings.stepSize,
gamma=settings.gamma)
  print("could not understand input, so set scheduler to StepLR")

# netz auf die gpu kopieren, falls eine cuda karte vorhanden ist
if settings.device == "cuda":
    startGPU = time.time()
    print("\ncopying net to gpu...")
    net.to(settings.device)
    endGPU = time.time()
    numbers["gpuTime"] = endGPU-startGPU
numbers["setupTime"] = time.time()-start
torch.set_num_threads(settings.threads)

"""
.d8888b.
888           .d888 d8b          d8b
d88P  Y88b
888          d88P"  Y8P          Y8P
Y88b.
888          888
 "Y888b.    8888b.  888  888  .d88b.       88888b.   8888b.
88888b.d88b.   .d88b.  .d8888b        .d88888  .d88b.  888888 888 88888b.  888  .d88b.
888d888 .d88b.  88888b.
    "Y88b.     "88b 888  888 d8P  Y8b      888 "88b     "88b 888 "888 "88b d8P  Y8b 88K
d88" 888 d8P  Y8b 888    888 888 "88b 888 d8P  Y8b 888P"  d8P  Y8b 888 "88b
      "888 .d888888 Y88  88P 88888888      888  888 .d888888 888  888  888 88888888 "Y8888b.
888  888 88888888 888    888 888  888 888 88888888 888    88888888 888  888
Y88b  d88P 888  888  Y8bd8P  Y8b.          888  888 888  888 888  888  888 Y8b.          X88
Y88b 888 Y8b.     888    888 888  888 888 Y8b.     888    Y8b.     888  888
 "Y8888P"  "Y888888   Y88P    "Y8888       888  888 "Y888888 888  888  888  "Y8888   88888P'
"Y88888  "Y8888  888    888 888  888 888  "Y8888  888     "Y8888  888  888
"""

# den namen erstellen unter welchem das model gespeichert wird
settings.modelName = network.saveName()

# den namen erstellen unter welchem outputs gespeichert werden
if settings.outfile == "default name":
    settings.outfile = settings.saveName()
    if settings.device == "cuda":
        settings.outfile += "_cuda"


"""
88888888888              d8b          d8b
    888                  Y8P          Y8P
    888
    888  888d888 8888b.  888 88888b.  888 88888b.   .d88b.
    888  888P"      "88b 888 888 "88b 888 888 "88b d88P"88b
    888  888    .d888888 888 888  888 888 888  888 888  888
    888  888    888  888 888 888  888 888 888  888 Y88b 888
    888  888    "Y888888 888 888  888 888 888  888  "Y88888
                                                        888
                                                   Y8b d88P
                                                    "Y88P"
"""

# hier wird geprüft ob es bereits ein model für das netzwerk gibt, falls ja, wird es geladen
if settings.retrain == False:
    print("\nlooking for model...")
    if Path("modelstate/{}--{}.pth".format(settings.modelName,
settings.categories)).is_file():
        trainNet = False
        start = time.time()
        print("loading model...")
        net.load_state_dict(torch.load("modelstate/{}--{}.pth".format(settings.modelName,
settings.categories)))
        net.to(settings.device)
        net.eval()
        numbers["loadTime"] = time.time()-start
    else:
        trainNet = True
        print("no model found")
else:
    trainNet = True
    print("\nno model loaded")

# das netzwerk wird trainiert, falls ein statedict vorhanden ist oder falls "--retrain"
geschrieben wurden
if trainNet == True:
    fit(net, lossFunc, optimizer, scheduler, train, valid, settings.epochs,
settings.batchSize, numbers, settings.device)
    #plotTerminal(settings.epochs, numbers["losses"], numbers["confidence"],
numbers["accuracy"], numbers["validation"])
    # speichern des statedicts, mit namen des models, des verwendeten datensatzes
```

```python
    torch.save(net.state_dict(), "modelstate/{}--{}.pth".format(settings.modelName,
settings.categories))


"""
888     888         888 d8b      888 d8b
888     888         888 Y8P      888 Y8P
888     888         888          888
Y88b   d88P 8888b.  888 888  .d88888 888  .d88b.  888d888 888  888 88888b.   .d88b.
 Y88b d88P     "88b 888 888 d88" 888 888 d8P  Y8b 888P"   888  888 888 "88b d88P"88b
  Y88o88P  .d888888 888 888 888  888 888 88888888 888     888  888 888  888 888  888
   Y888P   888  888 888 888 Y88b 888 888 Y8b.     888     Y88b 888 888  888 Y88b 888
    Y8P    "Y888888 888 888  "Y88888 888  "Y8888  888      "Y88888 888  888  "Y88888
                                                                                 888
                                                                            Y8b d88P
                                                                             "Y88P"
"""

print("\nbeginn validation...")
net.eval() # netz in evaliierungs modus setzen
length = numbers["allvalid"]/(2*settings.batchSize)
bar = pb.PixelBar("validation:", max=length+1) # progressbar wird definiert
valstart = time.time()
with torch.no_grad():
    for inputs, labels in valid:
        if settings.device == "cuda":
            startGPU = time.time()
            inputs = inputs.to(settings.device)
            labels = labels.to(settings.device)
            endGPU = time.time()
            numbers["gpuTime"] += endGPU-startGPU
        guesses = net(inputs)
        outputs.allGuesses = torch.vstack((outputs.allGuesses, guesses))
        _, preds = torch.max(guesses, 1) # wie der name sagt passiert hier die vorhersage
        outputs.total += labels.shape[0] # zählen wie viel insgesamt geraten wurde
        outputs.correct += (preds == labels).sum().item() # sämtliche korrekte vorhersagen
summieren
        c = (preds == labels).squeeze()
        # hier erstelle ich die confusion matrix
        for i in range(numbers["numCategories"]):
            outputs.confMatrix[labels,i] += (preds == i).sum().item()
            label = labels[i]
            outputs.classCorrect[label] += c[i].item()
            outputs.classTotal[label] += 1
        bar.next()
outputs.adjustGuesses(numbers["numCategories"])
bar.finish()


"""
       d8888
888                                         .d8888b.  888             888    d8b          888
d8b 888
      d88888                                                      888
d88P  Y88b 888             888    Y8P          888    Y8P 888
     d88P888                                                      888
Y88b.      888             888                 888        888
    d88P 888 888  888 .d8888b  888  888  888  .d88b.  888d888 888888 888  888 88888b.   .d88b.
"Y888b.   888888  8888b.  888888 888 .d8888b  888888 888 888  888
   d88P  888 888  888 88K      888  888  888 d8P  Y8b 888P"   888     888  888 888 "88b
d88P"88b              "Y88b. 888        "88b 888    888 88K       888    888 888 .88P
  d88P   888 888  888 "Y8888b. 888  888  888 88888888 888     888     888  888 888  888 888
888                "888 888    .d888888 888    888 "Y8888b. 888    888 888888K
 d8888888888 Y88b 888      X88 Y88b 888 d88P Y8b.     888     Y88b.  Y88b 888 888  888 Y88b
888 d8b      Y88b  d88P Y88b.  888  888 Y88b.  888      X88 Y88b.  888 888 "88b
d88P     888  "Y88888  88888P'  "Y8888888P"   "Y8888  888      "Y888  "Y88888 888  888
"Y88888 88P       "Y8888P"   "Y888 "Y888888  "Y888 888  88888P'  "Y888 888 888  888

888 8P
                                                                                       Y8b
d88P "
                                                                                        "Y88P"
"""

outputs.confMatrix = outputs.confMatrix.type(torch.LongTensor) # konvertieren des datentypes
des confusion matrix
```

```
numbers["validationTime"] = time.time()-valstart
Time = printTime(numbers["validationTime"])
print("validation finished, it took {}".format(Time))
print("")

# priten der confusion matrix als tabelle
confTabel = plotConfTerminal(outputs.confMatrix, settings.categoryNames)
print(confTabel)
print('\nAccuracy of the network on the {} events: {:.2f}%\n'.format(numbers["allvalid"],
outputs.accuracy()))

# true Positive, false Negative etc.
outputs.tptnfpfn(settings.categoryNames)
# report enthält sensitivity, specificity, precision, f1score und matthew correllation
coeffecient
outputs.classReport(settings.categoryNames)

reportTable = plotReportTerminal(outputs.report, settings.categoryNames, outputs.scores)
print(reportTable)

print("")
for name in settings.categoryNames:
    print("category: ",name)
    print("\t- TP: ",outputs.TP[name].item())
    print("\t- TN: ",outputs.TN[name].item())
    print("\t- FP: ",outputs.FP[name].item())
    print("\t- FN: ",outputs.FN[name].item())
    for score in outputs.scores:
        print("\t- {0}: {1:.2f}".format(score, outputs.report[score][name]))
print("total accuracy:
{0:.2f}".format(outputs.confMatrix.diagonal().sum().item()/outputs.confMatrix.sum().item()))

endEnd = time.time()
if 'slowpions' in settings.categoryNames:
  everyCount = outputs.confMatrix.sum() - outputs.confMatrix[-1,-1]
  print('got {0:.2f}% of the slow pions\n'.format(100 * outputs.confMatrix[0,0]/everyCount))

  if settings.save == True:
    # resultat datei erstellen
    results = open('results', 'a')
    results.write('{}\t'.format(settings.outfile))
    results.write('{}\t'.format(outputs.accuracy()/100))
    results.write('{}\t'.format(outputs.confMatrix[0,0]/everyCount))
    results.write('{}\t'.format(outputs.report['precision']['slowpions']))
    results.write('{}\t'.format(outputs.report['matthew']['slowpions']))
    results.write('{}\n'.format(endEnd-startStart))
    results.close()

numbers["runTime"] = endEnd-startStart
print("the whole run took {}".format(printTime(numbers["runTime"])))
if settings.device == "cuda":
    print("copying to GPU took {}".format(printTime(numbers["gpuTime"])))
print("")


"""
 .d88888b.           888                      888
d88P" "Y88b          888                      888
888     888          888                      888
888     888 888  888 888888 88888b.  888  888 888888 .d8888b
888     888 888  888 888    888 "88b 888  888 888    88K
888     888 888  888 888    888  888 888  888 888    "Y8888b.
Y88b. .d88P Y88b 888 Y88b.  888 d88P Y88b 888 Y88b.       X88
 "Y88888P"   "Y88888  "Y888 88888P"   "Y88888  "Y888  88888P'
                            888
                            888
                            888
"""

# einfache methode und die outputs nicht zu speichern
# das programm wird einfach abgebrochen
if settings.save == False:
    sys.exit()
print("saved all outputs under {}".format(settings.outfile))

# barchats, die ausgeben wie viele daten für training und validierung verwendet werden
```

```
plotbars(settings.categoryNames, numbers, settings.outfile)

# trainings kurven zu verlust und genauigkeit
if trainNet == True:
  plotLoss(["losses", "validation", "confidence", "accuracy"], settings.epochs, numbers,
settings.outfile)
  #plotLRLoss(numbers['learnRates'], numbers['losses'], numbers['validation'],
settings.outfile)

# verschiedene auswertungs plots
plotConfMatrix(outputs.confMatrix, settings.categoryNames, settings.outfile)
plotClassErrors(outputs.confMatrix, settings.categoryNames, numbers, settings.outfile)
plotClasses(outputs.report, settings.categoryNames, outputs.scores, settings.outfile)
plotGuesses(outputs.allGuesses, settings.categoryNames, settings.outfile)

# output log file schreiben, die ich für spätere auswertung verwenden möchte
file = open("outputs/{}.log".format(settings.outfile), "w")
file.write("------ Settings ------")
file.write("\nthis runs settings were:")
for key in settings.list():
    file.write("\n\t{}: {}".format(key, settings.getValue(key)))

file.write("\n\n------ Network ------")
file.write("\nthis run used: {}".format(net.__class__.__name__))
if correctOptim == True:
    file.write("\noptimizer was: {}".format(settings.optim))
else:
    file.write("\noptimizer was: {}".format(SGD))
file.write("\nlearn rate scheduler was: {}".format(settings.scheduler))
file.write("\nit ran on {}".format(settings.device))

file.write("\nthe networks parameters were:")
for key in network.list():
    file.write("\n\t{}: {}".format(key, network.getValue(key)))
if trainNet == True:
    file.write("\n\nthe network was trained from scratch")
else:
    file.write("\n\nit was a validation run")

file.write("\n\n------ Dataset ------")
file.write("\nthere are {} categories, namely:".format(numbers["numCategories"]))
for name in settings.categoryNames:
    file.write("\n- {} with {} data points".format(name,
numbers["{}train".format(name)]+numbers["{}valid".format(name)]))
    file.write("\n\t- {} points for training".format(numbers["{}train".format(name)]))
    file.write("\n\t- {} points for validation".format(numbers["{}valid".format(name)]))
file.write("\n\ntotal number for training:\t{}".format(numbers["alltrain"]))
file.write("\ntotal number for validation:\t{}".format(numbers["allvalid"]))
file.write("\n\n")
file.write(str(tableMetrics))

if trainNet == True:
    file.write("\n\n------ Training ------")
    file.write("\nhere follow traning statistics")
    for epoch in range(settings.epochs):
        file.write("\nepoch {}/{}:".format(epoch+1, settings.epochs))
        file.write("\n\tlearnRate: {}".format(numbers["learnRates"][epoch]))
        file.write("\n\tlosses: {}".format(numbers["losses"][epoch]))
        file.write("\n\tvalidation: {}".format(numbers["validation"][epoch]))
        file.write("\n\taccuracy: {}".format(numbers["accuracy"][epoch]))
        file.write("\n\tconfidence: {}".format(numbers["confidence"][epoch]))

file.write("\n\n------ Time ------")
file.write("\nloading data took \t \t {0:.2f}".format(numbers["importTime"]))
file.write("\nsetting up the network took \t {0:.2f}".format(numbers["setupTime"]))
if trainNet == True:
    file.write("\ntraining took \t \t \t {0:.2f}".format(numbers["trainingTime"]))
else:
    file.write("\nloading the model took \t \t {0:.2f}".format(numbers["loadTime"]))
file.write("\nvalidation took \t \t {0:.2f}".format(numbers["validationTime"]))
file.write("\nthe whole run took \t \t {0:.2f}".format(numbers["runTime"]))
if settings.device == "cuda":
    file.write("\ncopying to GPU took \t {0:.2f}".format(numbers["gpuTime"]))

file.write("\n\n------ Statistics ------")
file.write('\nAccuracy of the network on the {} events:
{:.2f}%'.format(numbers["allvalid"],outputs.accuracy()))
for i in range(numbers["numCategories"]):
    file.write('\n\nfor {}:'.format(settings.categoryNames[i]))
    file.write('\n- correct guesses: \t {}'.format(int(outputs.classCorrect[i])))
    file.write('\n- Accuracy: \t \t {:.2f}%'.format(100 * outputs.classCorrect[i] /
outputs.classTotal[i]))
if 'slowpions' in settings.categoryNames:
  file.write('\ngot {0:.2f}% of the slow pions\n'.format(100 * outputs.confMatrix[-1,-
1]/everyCount))
file.write("\n\n")
file.write(reportTable)

file.write("\n\n------ Confusion Matrix ------\n")
file.write(confTabel)

file.write("\n\n------ True, False, Positive, Negative ------")
for name in settings.categoryNames:
    file.write("\ncategory: {}".format(name))
    file.write("\n\t- TP: {}".format(outputs.TP[name].item()))
    file.write("\n\t- TN: {}".format(outputs.TN[name].item()))
    file.write("\n\t- FP: {}".format(outputs.FP[name].item()))
    file.write("\n\t- FN: {}".format(outputs.FN[name].item()))
    for score in outputs.scores:
        file.write("\n\t- {0}: {1:.2f}".format(score ,outputs.report[score][name]))
file.write("\ntotal accuracy:
{0:.2f}\n".format(outputs.confMatrix.diagonal().sum().item()/outputs.confMatrix.sum().item()))

if Path("{}".format(args.infile)).is_file():
  file.write('\n\n------ Input file ------\n')
  file.write('{}\n'.format(args.infile))
file.close()
```

And the helper code

```
import torch
import numpy as np
from matplotlib import pyplot as plt
from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader
from torch import nn
import torch.nn.functional as F
from torch import optim
import time
import progress.bar as pb
import math
from ast import literal_eval
from pathlib import Path
from prettytable import PrettyTable

"""
8888888888                888      888    d8b
888                       888      888    Y8P
888                       888      888
8888888 888  888 88888b.  888  888 888888 888  .d88b.  88888b.   .d88b.  88888b.
888     888  888 888 "88b 888 .88P 888    888 d88""88b 888 "88b d8P  Y8b 888 "88b
888     888  888 888  888 888888K  888    888 888  888 888  888 88888888 888  888
888     Y88b 888 888  888 888 "88b Y88b.  888 Y88..88P 888  888 Y8b.     888  888
888      "Y88888 888  888 888  888  "Y888 888  "Y88P"  888  888  "Y8888  888  888
"""

# diese funktion dient den daten typ des inputs zu überprüfen
def getType(input):
    try:
        return type(literal_eval(input))
    except (ValueError, SyntaxError):
        return str

# passt die inputs der ersten linearen layers an die umstände an
def firstLinSize(channels, kernels, pads):
    size = 9
    for kern, pad, chan in zip(kernels, pads, channels):
        if kern > 0:
            size = (size-(kern-1)+2*pad)
        else:
            kern = -kern
            size = (size+(kern-1)+2*pad)
    if len(channels) > 0:
        return size**2*channels[-1]
    else:
        return size**2

# netzwerk parameter vergleicheun und anpassen
def layerSize(master, slave, defaultFill):
    # auffüllen mit einem füllwert
    if len(master)-1 > len(slave):
        if len(slave) > 0:
            fillValue = slave[-1]
        else:
            fillValue = defaultFill
        while len(master)-1 > len(slave):
            slave.append(fillValue)
        print("filled {} to fit to {} with {}".format('slaveName', 'masterName', fillValue))
        return slave
    # falls es zu lang ist, wird es einfach abgeschnitten
    elif len(slave) > len(master)-1:
        lengthDiff = len(slave)-(len(master)-1)
        print("truncated {} to fit to {}".format('slave', 'master'))
        return slave[:lengthDiff+1]
    else:
        return slave

# paddings and kernelSizes anpassen
def kernPadSize(kernels, padding):
    if len(kernels) > len(padding):
        while len(kernels) > len(padding):
            index = len(padding)
            kernSize = kernels[index]
            if kernSize > 0:
                padValue = int((kernSize-1)/2)
            else:
                padValue = 0
            padding.append(padValue)
        print("filled padding to fit to kernelSize")
    # kürzen von padding, falls es zu lang ist
    elif len(padding) > len(kernels):
        lengthDiff = len(padding)-len(kernels)
        padding = padding[:lengthDiff+2]
        print("truncated padding to fit to kernelSize")
    return padding

# importier funktion
def importer(names, numbers, path, singlePixels):
    imported = {}
    numbers['allImported'] = []
    suffix = ''
    if singlePixels == True:
        suffix = '-nosinglepixels'
    print("\nimporting data ", end='')
    for name in names:
        if Path("{}/{}.pt".format(path, name+suffix)).is_file():
            imported[name] = torch.load("{}/{}.pt".format(path, name+suffix))
        print(".", end='')

    for name in names:
        numbers['allImported'].append(len(imported[name]))

    return imported


def importToTensor(imported, names, balanced, setFactor, kFold, batchSize, numbers):
    train = {}
    valid = {}
    if balanced == True:
        upperBound = int(min(numbers['allImported'])*setFactor)
        lowerBound = int(0.92*upperBound)
    # hier kürze ich den datensatz, einfach nur um das training zu beschleunigen
    # außerdem balanziere ich die kategorien auf eine ungefähr gleiche länge
    for i, name in enumerate(names):
        if balanced == False:
            upperBound = int(numbers['allImported'][i]*setFactor)
            lowerBound = int(0.92*upperBound)
        index = torch.randperm(len(imported[name]))
        length = torch.LongTensor(1).random_(lowerBound, upperBound).item()
        index = index[:length]
        imported[name] = imported[name][index]
        # den vollen datensatz in trainings-/validierungsdatensat trennen, nach kfold faktoren
        indices = torch.randperm(len(imported[name]))
        split = int(len(imported[name])/kFold)
        train[name], valid[name] = imported[name][split:], imported[name][:split]

    # tensoren für datensätze erstellen
    trainTensor = torch.vstack([train[name].float() for name in names])
    validTensor = torch.vstack([valid[name].float() for name in names])
    # labels für den jeweiligen datensatz erstellen
    trainLabel = torch.vstack([torch.full((len(train[name]),1),i) for i,name in
enumerate(names)]).flatten()
    validLabel = torch.vstack([torch.full((len(valid[name]),1),i) for i,name in
enumerate(names)]).flatten()

    for name in names:
        numbers["{}total".format(name)] = len(train[name])+len(valid[name])
        numbers["{}train".format(name)] = len(train[name])
        numbers["{}valid".format(name)] = len(valid[name])

    numbers["alltrain"] = len(trainTensor)
    numbers["allvalid"] = len(validTensor)
    numbers["total"] = len(trainTensor)+len(validTensor)

    # datensätze mit labels zusammenbringen, trainings datensatz mischen
    train = DataLoader(TensorDataset(trainTensor, trainLabel), batch_size=batchSize,
shuffle=True)
    valid = DataLoader(TensorDataset(validTensor, validLabel), batch_size=batchSize*2)

    return train, valid

# rechnet zeiten in stunden, minuten um und printet es übersichtlicher
def printTime(time):
    hours = math.floor(time/(60*60))
    if hours >= 1:
        time = time % (60*60)
    minutes = math.floor(time/60)
    seconds = time % 60
    if hours >= 1 and minutes > 0:
        return "{} hours, {} minutes & {:.1f} seconds".format(hours,minutes,seconds)
    elif hours >= 1 and minutes == 0:
        return "{} hours, {:.1f} seconds".format(hours,seconds)
    elif minutes >= 1:
        return "{} minutes, {:.1f} seconds".format(minutes,seconds)
    else:
        return "{:.1f} seconds".format(seconds)

# Fuktion zum erstellen des Model Names
def addStrings(seperator, numbers):
    modelSaveName = ""
    for i, number in enumerate(numbers):
        if i == 0:
            modelSaveName += seperator
        else:
            modelSaveName += '+'
        modelSaveName += str(number)
    return str(modelSaveName)

# wie der name schon sagt
def accuracyFunc(outputs, labels):
    preds = torch.argmax(outputs, dim=1)
    return (preds == labels).float().mean()

# nimmt nimmt durchschnitt der erratenen werte für batchsize
def confidenceFunc(outputs):
    m = nn.Softmax(dim=1)
    outputs = m(outputs)
    return outputs.max(1)[0].mean().item()

# die fit funktion, nimmt das model/netzwerk, die verlust- und optimierungsfunktion entgegen
# in numbers speichere ich alle zahlen wie epochs, batchsize, zeiten etc.
# device ist selbst erklärend, cpu oder cuda:0
def fit(model, lossFunc, opt, scheduler, train, valid, epochs, batchSize, numbers, device):
    startTrain = time.time()
    losses = torch.tensor([], device=device)
    validation = torch.tensor([], device=device)
    confidence = torch.tensor([], device=device)
    accuracy = torch.tensor([], device=device)
    learnRates = []

    print("\nbeginn training...")
    timeEpoch = torch.tensor([])
    for epoch in range(epochs):
        startEpoch = time.time()
        length = numbers["alltrain"]/batchSize+numbers["allvalid"]/(2*batchSize)
        bar = pb.PixelBar("epoch {}/{}:".format(epoch+1, epochs), max=length+1) # das ist die
progressbar pro lern epoche

        # lern metrik pro epoche, die werte pro batch werden hier gespeichert und später
gemittelt
        lossepoch = torch.tensor([], device=device)
        vallossepoch = torch.tensor([], device=device)
        confepoch = torch.tensor([], device=device)
        accuepoch = torch.tensor([], device=device)

        model.train() # model/netzwerk in trainings modus setzen
        for inputs, labels in train:
            # daten auf graphikkarte kopieren
            if device == "cuda":
                startGPU = time.time()
                inputs = inputs.to(device)
                labels = labels.to(device)
                endGPU = time.time()
                numbers["gpuTime"] += endGPU-startGPU
            opt.zero_grad() # optimizer gradienten leeren
            guesses = model(inputs) # predictions machen

            # hier werden lehrnmetriken berechnet
            confi = confidenceFunc(guesses)
            confepoch = torch.cat((confepoch, torch.tensor([confi], device=device)))
            loss = lossFunc(guesses, labels)
            lossepoch = torch.cat((lossepoch, torch.tensor([loss.item()], device=device)))
            accu = accuracyFunc(guesses, labels)
            accuepoch = torch.cat((accuepoch, torch.tensor([accu.item()], device=device)))

            if opt.__class__.__name__ == 'Adahessian':
                loss.backward(create_graph=True) # verlust zurück propagieren
            else:
                loss.backward() # verlust zurück propagieren
            opt.step()
            bar.next()

        model.eval()
        with torch.no_grad():
            for inputs, labels in valid:
                if device == "cuda":
                    startGPU = time.time()
                    inputs = inputs.to(device)
                    labels = labels.to(device)
                    endGPU = time.time()
                    numbers["gpuTime"] += endGPU-startGPU
                guesses = model(inputs)
                valloss = lossFunc(guesses, labels)
                vallossepoch = torch.cat((vallossepoch, torch.tensor([valloss.item()],
device=device)))
                bar.next()

        # metriken ...
        losses = torch.cat((losses, torch.tensor([lossepoch.mean().item()], device=device)))
        validation = torch.cat((validation, torch.tensor([vallossepoch.mean().item()],
device=device)))
        confidence = torch.cat((confidence, torch.tensor([confepoch.mean().item()],
device=device)))
        accuracy = torch.cat((accuracy, torch.tensor([accuepoch.mean().item()],
device=device)))
        timeEpoch = torch.cat((timeEpoch, torch.tensor([time.time()-startEpoch])))
        bar.finish()
        Time = printTime(time.time()-startEpoch)
        learnRRate = opt.state_dict()['param_groups'][0]['lr']
        learnRates.append(learnRRate)
        print("learnRate: {:.6f}, time: {}".format(learnRRate, Time))
        print("loss: {:.3f}, confidence: {:.3f}, accuracy: {:.3f}, validation:
{:.3f}".format(lossepoch.mean().item(), confepoch.mean().item(), accuepoch.mean().item(),
vallossepoch.mean().item()))
        if scheduler.__class__.__name__ == 'ReduceLROnPlateau':
            scheduler.step(valloss)
        else:
            scheduler.step()

    numbers["trainingTime"] = time.time()-startTrain
    Time = printTime(numbers["trainingTime"])
    print("\nfinished training, it took {}".format(Time))
    print("it took {} on average per epoch".format(printTime(timeEpoch.mean())))
    numbers["epochTime"] = timeEpoch.mean()
    numbers["losses"] = losses
    numbers["validation"] = validation
    numbers["confidence"] = confidence
    numbers["accuracy"] = accuracy
    numbers["learnRates"] = learnRates

"""
888    d8P  888
888   d8P   888
888  d8P    888
888d88K     888  8888b.  .d8888b  .d8888b   .d88b.  88888b.
8888888b    888     "88b 88K      88K      d8P  Y8b 888 "88b
888  Y88b   888 .d888888 "Y8888b. "Y8888b. 88888888 888  888
888   Y88b  888 888  888      X88      X88 Y8b.     888  888
888    Y88b 888 "Y888888  88888P'  88888P'  "Y8888  888  888
"""

# diese klasse enthält sämtliche einstellungen bzgl. des trainings
class settingsClass():
    batchSize = 64
    learnRate = 0.1
    momentum = 0.9
    epochs = 50
    kFold = 4
    weightDecay = 0.

    scheduler = 'step'
    stepSize = 1
    gamma = 0.5
    milestones = [int(epochs/4), int(2*epochs/4), int(3*epochs/4)]
    learnMin = learnRate/10
    learnMax = learnRate*5
    learnPeak = int(epochs/4)
    cycles = 5

    nesterov = False
    rho = 0.9
    eps = 1e-6
    alpha = 0.99
    learnRateDecay = 0
    beta = [0.9, 0.999]
    dampening = 0

    datapath = 'data'
    nosinglepixels = False

    device = "cpu"
    threads = torch.get_num_threads()
    categories = "bb+dd+pi+pp+sp"
    categoryNames = []
    balanced = True
    setFactor = 0.65
    retrain = False
    save = False
    optim = "SGD"
    outfile = "default name"
    modelName = ""
    @staticmethod
    def list():
        return [s for s in dir(settingsClass) if not
            (s.startswith('__') or callable(getattr(settingsClass, s)))]

    def getValue(self, key):
        return getattr(self, key)

    def setValue(self, key, value):
        setattr(self, key, value)

    def saveName(self):
        return "{}_{}_{}_bs{}_kf{}_e{}_lr{}".format(self.modelName, self.optim,
self.categories, self.batchSize, self.kFold, self.epochs, self.learnRate)

    def catNames(self):
        categorySplit = self.categories.split('+')
        if "test" in categorySplit:
            self.categoryNames = ['test1', 'test2', 'test3', 'test4']
            self.categories = "test"
        else:
            if "bg" in categorySplit:
                self.categoryNames += ["background"]
            if "dd" in categorySplit:
                self.categoryNames += ["antideuterons"]
            if "pi" in categorySplit:
                self.categoryNames += ["pions"]
            if "pp" in categorySplit:
                self.categoryNames += ["protons"]
            if "sp" in categorySplit:
                self.categoryNames += ["slowpions"]
            if "bp" in categorySplit:
                self.categoryNames += ["boxedpions"]
            if "bb" in categorySplit:
                self.categoryNames += ["beambackground"]
            if "ev" in categorySplit:
                self.categoryNames += ["everythingelse"]
            if "kk" in categorySplit:
                self.categoryNames += ["kaons"]
            if "gg" in categorySplit:
                self.categoryNames += ["gammas"]
            if "el" in categorySplit:
                self.categoryNames += ["electrons"]
            if "sl" in categorySplit:
                self.categoryNames += ["slowelectrons"]
            if "mm" in categorySplit:
                self.categoryNames += ["muons"]
            if "lb" in categorySplit:
                self.categoryNames += ["lightBG"]
            if "mb" in categorySplit:
                self.categoryNames += ["mesonBG"]
            if "hb" in categorySplit:
                self.categoryNames += ["heavyBG"]
            if "ab" in categorySplit:
                self.categoryNames += ["allBG"]
            # prefix für output files wird hier generiert
            # ich mache das so, weil ich im prefix immer die selbe reihenfolge will
            categorySplit.sort()
            catNames = ''
            for i, name in enumerate(categorySplit):
                if i == 0:
                    catNames += name
                else:
                    catNames += "+"
                    catNames += name
            self.categories = catNames

# diese klasse enthält sämtliche einstellungen bzgl. des netzes
class networkClass():
    linLayer = [81,49,21,4]
    dropout = [0,0,0]
    channels = []
    kernelSize = []
    padding = []
    actilin = ["relu", "relu", "softmax"]
    acticonv = []
    @staticmethod
    def list():
        return [s for s in dir(networkClass) if not
            (s.startswith('__') or callable(getattr(networkClass, s)))]

    def getValue(self, key):
        return getattr(self, key)

    def setValue(self, key, value):
        setattr(self, key, value)

    def saveName(self):
        name = ""
        if len(self.kernelSize) > 0:
            name += addStrings("conv-", self.kernelSize)
            name += addStrings("-", self.padding)
            name += addStrings("-", self.channels)
            name += '--'
        name += addStrings("lin-", self.linLayer)
        name += addStrings("-", self.dropout)
        return name

    # testet ob die netzwerk parameter sinn ergeben
    def test(self, outputSize):
        print("testing network parameters, if they will fit...")
        # output layer hinzufügen
        if self.linLayer[-1] != outputSize:
            if self.linLayer[-1] > outputSize:
                self.linLayer.append(outputSize) # output layer hinzufügen
            else:
                self.linLayer[-1]=(outputSize) # breite des outputs anpassen
            print("added the output layer")
        # dropout an länge der linLayer anpassen
        # falls dropout zu kurz ist, wird es mit dem letzten wert aufgefüllt
        self.dropout = layerSize(self.linLayer, self.dropout, 0)

        # überprüfen der channels, der erste kanal muss immer 1 sein
        # die länge von channels muss mindestens 2 sein
        if len(self.channels) == 1:
            self.channels.insert(0,1)
            print("channels was too short and added the first channel")
        if len(self.channels) > 0:
            if self.channels[0] != 1:
                self.channels.insert(0,1)
                print("added a first channel")
        # nun passn wir kenelSize an channels an, wie wir es mit linLayer und dropout machten
        self.kernelSize = layerSize(self.channels, self.kernelSize, 3)
        # falls kernels angegeben sind, aber keine channels, dann werden 1er channel
aufgefüllt
        if len(self.channels) == 0 and len(self.kernelSize) != 0:
            self.channels = [1] * (len(self.kernelSize)+1)
        # anpassen von padding, sodass es mit kernelSize zusammenpasst
        # padding wird so angepasst, dass die größe des bilds sich NICHT ändert
        self.padding = kernPadSize(self.kernelSize, self.padding)

        # auffüllen der aktivierungs listen
        self.actilin = layerSize(self.linLayer, self.actilin, 'relu')
        self.acticonv = layerSize(self.channels, self.acticonv, 'relu')
        self.actilin[-1] = 'softmax' # der ausgabe layer soll keine aktivierung haben

        # anpassen des ersten linLayers
        firstLinLayer = firstLinSize(self.channels, self.kernelSize, self.padding)
        if len(self.channels) >= 1 and self.linLayer[0] != firstLinLayer:
            if all(element == self.linLayer[0] for element in self.linLayer[:-1]):
                for i in range(len(self.linLayer)-1):
                    self.linLayer[i] = firstLinLayer
                print("adjusted the all linLayers to fit to convolutional layers")
            else:
                self.linLayer[0] = firstLinLayer
                print("adjusted the first linLayer to fit to convolutional layers")
        elif self.linLayer[0] != 81 and len(self.channels) == 0:
            self.linLayer.insert(0,81)
            print("added an input layer")
```

```
# eine klasse die sämtliche outputs und post-processing enthält
class outputsClass():
    def __init__(self, numOutputs, device):
        self.total = 0
        self.classTotal = list(0. for i in range(numOutputs))
        self.correct = 0
        self.classCorrect = list(0. for i in range(numOutputs))
        self.classAccuracy = list(0. for i in range(numOutputs))
        self.confMatrix = torch.zeros((numOutputs, numOutputs), device=device)
        self.TP, self.TN, self.FP, self.FN = {}, {}, {}, {}
        self.sensitivity, self.specificity, self.precision, self.f1score, self.matthew = {},
{}, {}, {}, {}
        self.scores = ["sensitivity", "specificity", "precision", "f1score", "matthew"]
        self.reports = {}
        self.allGuesses = torch.zeros((1,numOutputs), device=device)

    def accuracy(self):
        return 100 * self.correct/self.total

    # berechnet TP, TN, FP und FN aus der confusion matrix
    def tptnfpfn(self, categoryNames):
        numCategories = len(categoryNames)
        for k, name in enumerate(categoryNames):
            for i in range(numCategories):
                for j in range(numCategories):
                    if i == j:
                        self.TP[name] = self.confMatrix[k,k]
                    if i == k:
                        self.FP[name] = self.confMatrix[:,k].sum()-self.confMatrix[k,k]
                    if j == k:
                        self.FN[name] = self.confMatrix[k].sum()-self.confMatrix[k,k]
                    self.TN[name] = self.confMatrix[k].sum()+self.confMatrix[:,k].sum()-
2*self.confMatrix[k,k]

    # berechnet sensitivity etc aus TP, TN, FP, FN für jede klasse
    # reports in ein geschachteltes dictionary
    # -> erst die scores
    # --> dann categoryNames
    def classReport(self, categoryNames):
        for name in categoryNames:
            try:
                self.sensitivity[name] =
self.TP[name].item()/(self.TP[name].item()+self.FN[name].item())
            except ZeroDivisionError:
                self.sensitivity[name] = 0
            try:
                self.specificity[name] =
self.TN[name].item()/(self.TN[name].item()+self.FP[name].item())
            except ZeroDivisionError:
                self.specificity[name] = 0
            try:
                self.precision[name] =
self.TP[name].item()/(self.TP[name].item()+self.FP[name].item())
            except ZeroDivisionError:
                self.precision[name] = 0
            try:
                self.f1score[name] =
2*(self.sensitivity[name]*self.specificity[name])/(self.sensitivity[name]+self.specificity[nam
e])
            except ZeroDivisionError:
                self.f1score[name] = 0
            try:
                self.matthew[name] = (self.TP[name].item()*self.TN[name].item()-
self.FP[name].item()*self.FN[name].item())/math.sqrt((self.TP[name].item()+self.FP[name].item(
))*(self.TP[name].item()+self.FN[name].item())*(self.TN[name].item()+self.FP[name].item())*(se
lf.TN[name].item()+self.FN[name].item()))
            except ZeroDivisionError:
                self.matthew[name] = 0
        self.report = {"sensitivity": self.sensitivity, "specificity": self.specificity,
"precision": self.precision, "f1score": self.f1score, "matthew": self.matthew}

    def adjustGuesses(self, numOutputs):
        self.allGuesses = self.allGuesses[1:]
        self.allGuesses = torch.round(20*self.allGuesses)/20
```

```
"""
888b    888          888                                             888
8888b   888          888                                             888
88888b  888          888                                             888
888Y88b 888  .d88b.  888888 88888888 888  888  888  .d88b.  888d888 888  888
888 Y88b888 d8P  Y8b 888       d88P  888  888  888 d8P  Y8b 888P"   888 .88P
888  Y88888 88888888 888      d88P   888  888  888 88888888 888     888888K
888   Y8888 Y8b.     Y88b.   d88P    Y88b 888 d88P Y8b.     888     888 "88b
888    Y888  "Y8888   "Y888 88888888  "Y8888888P"   "Y8888  888     888  888
"""

# eine funktion um flexibel die aktivierung zu wählen
def activator(activate):
    activation = nn.ModuleDict([
                ['lrelu', nn.LeakyReLU()],
                ['elu', nn.ELU()],
                ['relu', nn.ReLU()],
                ['rrelu', nn.RReLU()],
                ['sigi', nn.Sigmoid()],
                ['tanh', nn.Tanh()],
                ['id', nn.Identity()],
                ['softplus', nn.Softplus()],
                ['softmax', nn.Softmax(dim = 1)]
        ])
    return activation[activate]


# hier definiere ich convolutional layer
# eine Funktion zum erstellen von conv layern
def createConv(inChannels, outChannels, pad, activate, *args, **kwargs):
    print("created convolutional layer with {} inChannels, {} outChannels and activation
{}".format(inChannels, outChannels, activate))
    return nn.Sequential(
        nn.Conv2d(inChannels, outChannels, *args, **kwargs),
        nn.BatchNorm2d(outChannels),
        nn.ReplicationPad2d(pad),
        activator(activate)
    )

# erstelle conv-transposed layer
def createConvTranspose(inChannels, outChannels, pad, activate, *args, **kwargs):
    print("created convolutional transposed layer with {} inChannels, {} outChannels and
activation {}".format(inChannels, outChannels, activate))
    return nn.Sequential(
        nn.ConvTranspose2d(inChannels, outChannels, *args, **kwargs),
        nn.BatchNorm2d(outChannels),
        nn.ReplicationPad2d(pad),
        activator(activate)
    )


# Schrittweise Erstellung der conv layer
class myConv(nn.Module):
    def __init__(self, channels, kernels, pads, activate):
        super().__init__()
        layerBlock = []
        for kern, pad, inSize, outSize, acti in zip(kernels, pads, channels, channels[1:],
activate):
            if kern > 0:
                layerBlock.append(createConv(inSize, outSize, activate=acti, kernel_size=kern,
pad=pad))
            else:
                kern = -kern
                layerBlock.append(createConvTranspose(inSize, outSize, activate=acti,
kernel_size=kern, pad=pad))
        #self.layers = nn.ModuleList([createConv(inSize, outSize, activate=activate,
kernel_size=kern, pad=pad) for kern, pad, inSize, outSize in zip(kernels, pads, channels,
channels[1:])])
        self.layers = nn.Sequential(*layerBlock)

    def forward(self, x):
        x = x.view(-1, 1, 9, 9)
        for layer in self.layers:
            x = layer(x)
        return x
```

```
# hier definiere ich meine linearen layer
# eine Funktion zum erstellen von linearen layern
def createLinear(numInputs, numOutputs, drop, activate):
    print("created linear layer with {} inputs, {} outputs and activation
{}".format(numInputs, numOutputs, activate))
    return nn.Sequential(
        nn.Linear(numInputs, numOutputs),
        nn.BatchNorm1d(numOutputs),
        nn.Dropout(drop),
        activator(activate)
    )

# Schrittweise Erstellung der linaren layer
class myLinear(nn.Module):
    def __init__(self, linLayers, dropout, activate):
        super().__init__()
        layerBlock = []
        self.layers = nn.ModuleList([createLinear(inSize,outSize, drop, acti) for inSize,
outSize, drop, acti in zip(linLayers, linLayers[1:], dropout, activate)])

        for layer in self.layers:
            nn.init.xavier_uniform_(layer[0].weight)
            nn.init.zeros_(layer[0].bias)

    def forward(self, x):
        x = x.flatten(1)
        for layer in self.layers:
            x = layer(x)
        return x


# das ist das eigentliche netz
# hier wird alles zusammen gestellt, die conv und linearen layer
class pixelNet(nn.Module):
    def __init__(self, linLayers, dropout, actilin, channels, kernels, pads, acticonv):
        super().__init__()
        layerBlock = []
        # hier werden die conv layer erstellt
        if len(kernels) > 0:
            layerBlock.append(myConv(channels=channels, kernels=kernels, pads=pads,
activate=acticonv))

        # hier werden die linearen layer erstellt
        layerBlock.append(myLinear(linLayers=linLayers, dropout=dropout, activate=actilin))
        self.layers = nn.Sequential(*layerBlock)

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return x


"""
8888888b.  888          888
888   Y88b 888          888
888    888 888          888
888   d88P 888  .d88b.  888888 .d8888b
8888888P"  888 d88""88b 888    88K
888        888 888  888 888    "Y8888b.
888        888 Y88..88P Y88b.       X88
888        888  "Y88P"   "Y888  88888P'
"""

# plotten wie viele daten in einem datensatz enthalten sind, wie viel für training und
validierung verwendet werden
# save ist ein bool der festlegt ob die plots gespeichert werden sollen
# fileSaveName ist der name unter dem gespeichert wird
def plotbars(categoryNames, numbers, fileSaveName):
    fig = plt.figure()
    plt.title("Number of data points per category")
    width = 0.3
    x = np.arange(len(categoryNames))
    plt.ylabel("Number of data points")
    plt.grid(which='both', axis='y', ls=':')
    plt.xticks(np.arange(len(categoryNames)), (categoryNames))
    plt.bar(x-width,[numbers["{}total".format(name)] for name in categoryNames], width=width,
label="total")
    plt.bar(x,[numbers["{}train".format(name)] for name in categoryNames], width=width,
label="train")
    plt.bar(x+width,[numbers["{}valid".format(name)] for name in categoryNames], width=width,
label="valid")
    plt.legend()
    fig.savefig("plots/dataSet-{}.png".format(fileSaveName))

# wie der name sagt, plottet in einem gitter ein paar 9x9 daten
def previewPlot(categoryNames, train, batchSize):
    images, labels = next(iter(train))
    xsize = int(np.floor(math.sqrt(batchSize)))
    ysize = int(batchSize/xsize)
    fig, axes = plt.subplots(ysize,xsize, figsize=(10,10))
    k = 0
    for i in range(ysize):
        for j in range(xsize):
            axes[i,j].set_title(categoryNames[labels[k]])
            axes[i,j].imshow(images[k].reshape(9,9))
            k += 1
    fig.tight_layout()
    plt.show()

# plottet die verlust etc funktionen nachdem trainingsprozess
# save ist ein bool der festlegt ob die plots gespeichert werden sollen
# fileSaveName ist der name unter dem gespeichert wird
# ich weiß, dass diese funktion nicht besonders elegant gelöst ist
# aber sie funktioniert... immerhin etwas
def plotLoss(titles, epochs, numbers, fileSaveName):
    x = np.linspace(0,epochs,epochs)
    ySize = int(np.floor(math.sqrt(len(titles))))
    xSize = int(len(titles)/ySize)
    while ySize*xSize < len(titles):
        ySize += 1
    fig, axes = plt.subplots(ySize,xSize, figsize=(10,10))
    fig.suptitle("Losses during training")
    if ySize == 1:
        for i in range(len(titles)):
            axes[i].plot(x,numbers[titles[i]].cpu())
            axes[i].grid(ls=':')
            axes[i].set_title(titles[i])
            axes[i].set_xlabel("epoch")
            axes[i].set_ylabel("value")
    else:
        k = 0
        for i in range(xSize):
            for j in range(ySize):
                axes[i,j].plot(x,numbers[titles[k]].cpu())
                axes[i,j].grid(ls=':')
                axes[i,j].set_title(titles[k])
                axes[i,j].set_xlabel("epoch")
                axes[i,j].set_ylabel("value")
                k += 1
    fig.tight_layout()
    fig.savefig("plots/losses-{}.png".format(fileSaveName))

def plotLRLoss(learnRate, trainLoss, validLoss, fileSaveName):
    fig, ax = plt.subplots()
    x = np.arange(len(learnRate))
    ierror = torch.full((len(x),), validLoss.min())

    ax2 = ax.twinx()
    ax.set_ylabel("Learn Rate")
    ax.set_xlabel('Epoch')
    ax2.set_ylabel("Losses")
    ax.grid(ls=':')
    ax.plot(x,learnRate, color="tab:red", label='Learn Rate')
    ax2.plot(x,trainLoss, color="tab:blue", label='Traing Loss')
    ax2.plot(x,validLoss, color="tab:green", label='Validation Loss')
    ax2.plot(x,ierror, color="tab:grey", ls='--', label='Instrictic Error')
    ax.legend()
    ax2.legend()
    #plt.show()
    fig.savefig("plots/lrloss-{}.png".format(fileSaveName))
```

```
# ich bin mir recht sicher, dass ich die nächsten zwei funktionen zu einer kombinieren kann
# diese und die nächste machen grob das selbe

# die confusion matrix plottet die tatsächlichen klassen gegen die erratenen klassen
# daraus kann man TP, FP, TN, FP und alles weitere ableiten
# save ist ein bool der festlegt ob die plots gespeichert werden sollen
# fileSaveName ist der name unter dem gespeichert wird
def plotConfMatrix(dataSet, categoryNames, fileSaveName):
    fig = plt.figure()
    plt.title('Confusion Matrix')
    plt.xlabel("Predicted")
    plt.ylabel("Classes")
    plt.yticks(np.arange(len(categoryNames)), (categoryNames))
    plt.xticks(np.arange(len(categoryNames)), (categoryNames), rotation=45)
    plt.imshow(dataSet.cpu(), cmap='Blues')
    plt.colorbar()
    threshold = dataSet.max()/2.
    for i in range(len(categoryNames)):
        for j in range(len(categoryNames)):
            plt.text(j, i, format(dataSet[i, j]), horizontalalignment="center", color="white"
if dataSet[i, j] > threshold else "black")
    plt.tight_layout()
    fig.savefig("plots/confMatrix-{}.png".format(fileSaveName))

# plottet sensitivity etc. für jede klasse in einem übersichtichem diagramm
# save ist ein bool der festlegt ob die plots gespeichert werden sollen
# fileSaveName ist der name unter dem gespeichert wird
def plotClasses(report, categoryNames, scores, fileSaveName):
    fig = plt.figure()
    plt.title('Class Report')
    plt.xlabel("Score")
    plt.ylabel("Class")
    plt.yticks(np.arange(len(categoryNames)), (categoryNames))
    plt.xticks(np.arange(len(scores)), (scores), rotation=45)
    imfile = torch.zeros(len(categoryNames), len(scores))
    for i, name in enumerate(categoryNames):
        for j, score in enumerate(scores):
            imfile[i,j] = report[score][name]
    plt.imshow(imfile.cpu(), cmap='Reds')
    plt.colorbar()
    for i, name in enumerate(categoryNames):
        for j, score in enumerate(scores):
            plt.text(j,i, "{0:.2f}".format(report[score][name]), horizontalalignment="center",
color='white' if report[score][name]  >= 0.7 else "black")
    plt.tight_layout()
    fig.savefig("plots/reportMatrix-{}.png".format(fileSaveName))

# bar charts die nichts anderes als die confusion matrix noch einmal anders darstellt
# es werden die klassen angezeigt und darüber in bars wie viele pro klasse für die klasse
erraten wurden
# außerdem wird ein schwarzer strich angezeigt, der sagt wie viele daten für die valierung
verwendet wurden
# save ist ein bool der festlegt ob die plots gespeichert werden sollen
# fileSaveName ist der name unter dem gespeichert wird
def plotClassErrors(dataSet, categoryNames, numbers, fileSaveName):
    fig, ax = plt.subplots()
    plt.grid(which='both', axis='y', ls=':')
    offset = torch.zeros(len(categoryNames))
    target = torch.zeros(len(categoryNames))
    for i,name in enumerate(categoryNames):
        ax.bar(categoryNames, dataSet[i], label=name, bottom=offset, zorder=1)
        offset = offset + dataSet[i]
        target[i] = numbers["{}valid".format(name)]
    plt.scatter(categoryNames, target, marker='_', color='k', zorder=2)
    ax.set_ylabel("Number of Evenets")
    ax.set_xlabel("Classes")
    ax.set_title("Class Prediction Error")
    ax.legend()
    fig.savefig("plots/classErrors-{}.png".format(fileSaveName))

def plotGuesses(guesses, categoryNames, fileSaveName):
    fig = plt.figure()
    plt.title('Guesses pre Data Set')
    plt.xlabel('Output values')
    plt.ylabel('Number of data points')
    plt.grid(which='both', axis='y', ls=':')
    bins = np.around(np.arange(0,1.1,0.05),1)
    uniques = np.zeros((len(categoryNames),len(bins)))
    counts = np.zeros((len(categoryNames),len(bins)))
    colors = ['tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple', 'tab:cyan']
    for i in range(guesses.shape[1]):
        for j in range(len(bins)):
            counts[i][j] = (guesses[:,i] == bins[j]).sum()
    for i in range(guesses.shape[1]):
        plt.bar(bins, counts[i], width=0.1, alpha=0.5, color=colors[i], edgecolor=colors[i],
label=categoryNames[i])
    plt.legend()
    fig.savefig("plots/numberGuesses-{}.png".format(fileSaveName))


# diese funktion plottet die trainings kurven im terminal aus
# ich mache das damit ich sehe wie das training lief, ohne das programm zu unterbrechen mit
extra fenstern
#def plotTerminal(epochs, loss, conf, accu, vali):
#    x = np.arange(epochs)
#    loss = np.array(loss)
#    conf = np.array(conf)
#    accu = np.array(accu)
#    vali = np.array(vali)
#    gp.plot((x, loss, dict(title='losses')),
#            (x, conf, dict(title='confidence')),
#            (x, accu, dict(title='accurracy')),
#            (x, vali, dict(title='validation')),
#            multiplot='title "training" layout 2,2', terminal = 'dumb 120,60', unset =
'grid')

# confusion matrix als tabelle ums im terminal zu plotten
def plotConfTerminal(dataSet, categoryNames):
    table = PrettyTable()
    table.add_column("", categoryNames)
    length = len(categoryNames)
    for i in range(length):
        table.add_column(categoryNames[i], dataSet.transpose(0,1)[i].cpu().numpy())
    return str(table)

# class report als tabelle ums im terminal zu plotten
def plotReportTerminal(report, categoryNames, scores):
    table = PrettyTable()
    table.add_column("", categoryNames)
    columns = np.zeros((len(scores), len(categoryNames)))
    for i, name in enumerate(categoryNames):
        for j, score in enumerate(scores):
            columns[j,i] = report[score][name]
            columns[j,i] = np.round(columns[j,i], 3)
    for i, score in enumerate(scores):
        table.add_column(score, columns[i])
    return(str(table))
```

The preprocessing Code

```
import torch
import numpy as np
from matplotlib import pyplot as plt
import sys

categoryNames = ["antideuterons", "pions", "protons", "slowpions", "boxedpions",
"beambackground", "electrons", "kaons", "gammas", "muons", "slowelectrons"]
imported, importedMore, indices, onepx = {}, {}, {}, {}

for name in categoryNames:
    numpyArr = np.loadtxt("data/{}.txt".format(name))
    imported[name] = torch.from_numpy(numpyArr)
    imported[name] = imported[name][:,2:-3]

for name in categoryNames:
    torch.save(imported[name], "data/{}.pt".format(name))

for name in categoryNames:
         imported[name] = torch.load("data/{}.pt".format(name))
```

```
        print(imported[name].shape)

combinedNames = ['heavyBG', 'lightBG', 'allBG', 'everything', 'mesonBG']
combinedData = {'heavyBG': torch.vstack([imported["antideuterons"],imported["pions"],
imported["protons"], imported["kaons"]]),
        'mesonBG': torch.vstack([imported["pions"], imported["kaons"]]),
        'lightBG': torch.vstack([imported["electrons"],imported["muons"],
imported["gammas"]]),
        'allBG': torch.vstack([imported["electrons"],imported["muons"], imported["gammas"],
imported["antideuterons"],imported["pions"], imported["protons"], imported["kaons"]]),
        'everything': torch.vstack([imported["electrons"],imported["muons"],
imported["gammas"], imported["antideuterons"],imported["pions"], imported["protons"],
imported["kaons"], imported["beambackground"]])
}

for name in combinedNames:
        Index = torch.randperm(len(combinedData[name]))
        combinedData[name] = combinedData[name][Index]
        torch.save(combinedData[name], "data/{}.pt".format(name))

total = 0
oneTotal = 0
for name in categoryNames:
        imported[name] = imported[name].reshape(len(imported[name]),9,9)
        nonZeros = torch.count_nonzero(imported[name], dim=(1,2))
        onepx[name] = torch.sum(nonZeros[nonZeros==1])
        indexOne = torch.where(nonZeros==1)
        indexMore = torch.where(nonZeros>1)
        importedMore[name] = imported[name][indexMore]
        total += len(imported[name])
        oneTotal += onepx[name]
        print(name, len(imported[name]), onepx[name])
        setMean = imported[name].mean(dim=0)
        setNorm = setMean - imported[name]
        index = setNorm.norm(dim=[1,2]).sort(descending=True)
        indices[name] = index

print('total', total, oneTotal)

for name in categoryNames:
        importedMore[name] = importedMore[name].reshape(len(importedMore[name]), 81)
        imported[name] = imported[name].reshape(len(imported[name]), 81)
        torch.save(importedMore[name], "data/{}-nosinglepixel.pt".format(name))

combinedDataMore = {'heavyBG':
torch.vstack([importedMore["antideuterons"],importedMore["pions"], importedMore["protons"],
importedMore["kaons"]]),
'mesonBG': torch.vstack([importedMore["pions"], importedMore["kaons"]]),
'lightBG': torch.vstack([importedMore["electrons"],importedMore["muons"],
importedMore["gammas"]]),
'allBG': torch.vstack([importedMore["electrons"],importedMore["muons"],
importedMore["gammas"], importedMore["antideuterons"],importedMore["pions"],
importedMore["protons"], importedMore["kaons"]]),
'everything': torch.vstack([importedMore["electrons"],importedMore["muons"],
importedMore["gammas"], importedMore["antideuterons"],importedMore["pions"],
importedMore["protons"], importedMore["kaons"], imported["beambackground"]])
}

for name in combinedNames:
        Index = torch.randperm(len(combinedDataMore[name]))
        combinedDataMore[name] = combinedDataMore[name][Index]
        torch.save(combinedDataMore[name], "data/{}-nosinglepixel.pt".format(name))

fig = plt.figure()
plt.title("Number of events per data set")
width = 0.5
x = np.arange(len(categoryNames))
plt.ylabel("Number of events")
plt.grid(which='both', axis='y', ls=':')
plt.xticks(np.arange(len(categoryNames)), (categoryNames), rotation=45)
plt.bar(x-width/4, [len(imported[name]) for name in categoryNames], width=width/2,
label='total')
plt.bar(x+width/4, [onepx[name] for name in categoryNames], width=width/2, label='one-pixel')
plt.legend()
plt.savefig('dataSet.png')
```

```
for name in categoryNames:
        importedMore[name] = importedMore[name].reshape(len(importedMore[name]), 9,9)
        imported[name] = imported[name].reshape(len(imported[name]), 9,9)

fig, axes = plt.subplots(len(categoryNames),7, figsize=(20,20))
fig.suptitle("preview")
for j, name in enumerate(categoryNames):
        axes[j,0].set_ylabel(name)
        for i in range(2):
                axes[j,i].tick_params(labelleft=False, labelbottom=False, left=False,
bottom=False)
                axes[j,i].imshow(imported[name][indices[name][1][i].item()])
                axes[j,i].set_title('Most deviating')
        for i in range(3):

        axes[j,i+2].imshow(imported[name][indices[name][1][int(len(imported[name])/2)].item(
)-1+i])
                axes[j,i+2].tick_params(labelleft=False, labelbottom=False, left=False,
bottom=False)
                axes[j,i+2].set_title('Mean deviation')
        for i in range(2):
                axes[j,i+5].tick_params(labelleft=False, labelbottom=False, left=False,
bottom=False)
                axes[j,i+5].imshow(imported[name][indices[name][1][-(i+1)].item()])
                axes[j,i+5].set_title('Least deviating')
plt.savefig('preview.png')
```

C. Code Explanation

This code needs the following libraries in order to run:

- PyTorch
- Numpy
- Matplotlib
- progress.bar / progress 1.5
- prettytable
- argparse
- configparser

It works rather simple, there is a default setting and one can execute the code by running it with:

```
$python nn.py
```

Then the code runs the following setup:

- batchSize = 64
- learnRate = 0.1
- momentum = 0.9
- epochs = 50
- kFold = 4
- weightDecay = 0.
- a flat learning rate
- SGD as optimizer
- 65% of each data set will be used to train
- training will only happen, if no network model will be found
- single pixel events will be included
- no output will be saved
- it will run on the CPU, with maximum amount of threads
- Beam Background, Anti-Deuterons, Pions, Protons and Slow Pions will be used to train

This configuration is declared within the settingsClass, which is located in the helper.py file. The default network settings are in the networkClass, which is also located in the helper.py, the default settings here are:

- input layer with 49 neurons and 0% dropout rate, ReLU activation
- hidden layer with 21 neurons and 0% dropout rate, ReLU activation
- output later with 4 neurons and 0% dropout rate, Softmax activation

In order to adjust these settings, one can use either CLI input flags, that is why argparse is necessary or use an input file. An input file can be read by the following command:

```
$python nn.py -i name_of_input
```

One than has to specify an input file. I will provide a full list of input flags in a table and illustrate how to use them with an example input file. Every input file consists of two categories:
[SETTING]
and
[NETWORK]

Here follow the tags for the [SETTINGS] class:

Short Flag	**Flag**	**Description**
-i	--infile	define name of settings file
-o	--outfile	define name of output files
-b	--batchSize	sets the batchsize
-l	--learnRate	sets the learning rate
-m	--momentum	sets the momentum
-e	--epochs	sets the number of epochs
-k	--kFold	sets the of k-folds
	--optim	define the optimizer used
-w	--weightDecay	sets the weight decay for optimizer

	--scheduler	define the scheduler used
-g	--gamma	factor by which learnRate is reduced
-s	--stepSize	step size with which to reduce learnRate
	--milestones	sets the milestones at which learning rate should change
	--learnMax	the maximum learnRate for lambda scheduler
	--learnMin	the minimum learnRate for lambda scheduler
	--learnPeak	the epoch of learnRate peak for lambda scheduler
	--nesterov	switches SGD to the Nesterov variant
	--rho	coefficient for running average of squared gradients
	--eps	numerical stability constant for optimizer
	--alpha	smoothing constant for RMSprop
	--learnRateDecay	determines the falloff for adagrad learnrate
	--beta	runnung average gradient coefficients for adam
	--datapath	sets where the data to be analyzed are stored
	--nosinglepixels	exclude single pixel events

-sf	--setFactor	sets a factor for the total amount of data per set
	--balanced	should all data sets be about the same size?
-d	--device	sets the processing device {cpu, cuda}
-t	--threads	sets the number of processes
-c	--categories	specify the train/valid categories {dd, pi, pp, sp, bp, bg, test}
	--retrain	force to retrain the net
	--save	save output data

And here follow the tags for [NETWORK] class:

Short Flag	**Flag**	**Description**
-ll	--linLayer	defines number of neurons per layer
-do	--dropout	defines the dropout rate per layer
-al	--actilin	defines the activation per layer
-ch	--channels	defines the number of channels per convolution
-ks	--kernelSize	defines the kernel size per convolution
-pd	--padding	defines the padding size per convolution
-ac	--acticonv	defines the activation per convolution

If some settings are not given or would generate errors, the code will try to fix it and print some information about the settings and adjustments. There is some caution advised and one should be aware of settings changes.
An example input file:

```
[SETTINGS]
retrain = True
categories = sp+bb
outfile = 01convs9C3KFixed
batchSize = 128
setFactor = 0.35
balanced = True
epochs = 100
optim = adam
learnRate = 0.00001
momentum = 0.15
gamma = 1
eps = 1e-08
scheduler = step
learnMax = 0.00005
cycles = 5
nosinglepixels = True

[NETWORK]
dropout = 0, 0.5
actilin = relu
linLayer = 81, 81, 81, 81, 81, 2
acticonv = relu
kernelSize = 3
padding = 0
channels = 1, 9
```

After a successful run and if one specified, that outputs should be saved, then the code will plot some graphs detailing the run and will save them to the directory plots. More important is the run log file, which will be saved into the directory outputs.

There will be five plots per run. The loss curves, together with a confidence and training accuracy plot. Confidence is the corresponding value of the guess. The networks output is a list of floats, one float per category, and they are normalized to be used as probability. Confidence is simply the largest number from this list. Next is the confusion matrix and based on that a plot called class error, which is just a different way representing the confusion matrix. Here each bar represents the guesses per category, each color represents the actual class. The fourth plot is a report matrix, a collection of different scores per category and it is based on the confusion matrix. The last plot shows the winning guess value per category.

The output file lists the run settings, the network settings, some information about the categories used in this run. Then comes a list of each epoch with the values:

- learnning rate
- training loss
- validation loss
- training accuracy
- confidence

And finally, how long the run took, some training statistics and the same scores as the report plot, but as a table, are written in the output log. In the output log is also the confusion matrix and a list of all scores per category.