

## DIPLOMARBEIT

---

# Anwendung von Deep Learning auf die Rekonstruktion von Elektronen-Spuren im CMS-Experiment

---

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Physik

eingereicht von

Martin Loesener

Matrikelnummer 01229469

ausgeführt am Institut für Hochenergiephysik  
(in Zusammenarbeit mit CERN)

Betreuung

Betreuer: Univ.-Doz. DI Dr. Rudolf FRÜHWIRTH

Wien, 28.03.2019

\_\_\_\_\_  
(Unterschrift Verfasser)

\_\_\_\_\_  
(Unterschrift Betreuer)

## Declaration of Authorship

I, Martin E. M. LOESENER DA SILVA VIANA, BSc., declare that this thesis titled, “Application of Deep Learning to the reconstruction of electron tracks in the CMS Experiment” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

---

Date:

---

*“The measure of intelligence is the ability to change.”*

Albert Einstein

VIENNA UNIVERSITY OF TECHNOLOGY

*Abstract*Institute for High Energy Physics (HEPHY) /  
Conseil Européen pour la Recherche Nucléaire (CERN)

Master of Science

**Application of Deep Learning to the reconstruction of electron tracks in the CMS  
Experiment**

by Martin E. M. LOESENER DA SILVA VIANA, BSc.

This work proposes an implementation of a Deep Regression model for the purpose of predicting electron track parameters from collision events at the CMS Experiment at CERN. It is entirely written in Python, one of the most popular programming languages in the field of machine learning, and makes use of PyTorch, a cutting-edge Deep Learning framework, widely known for its dynamic graph structure. Several architectures were used, including the base architecture proposed by Bernkopf (Master's Thesis under revision) and convolutional neural networks. They were trained and tested using a variety of algorithms and hyper-parameters to assess their performance. A **training error reduction of a factor 2 to 3.5** was achieved with respect to the configuration proposed by Bernkopf, depending on the parameter. With under 5k network parameters the model offers a light-weight and precise tool to predict electron tracks. With automated predictions of streaming particle track data in mind, a possible bottleneck regarding data imports using different data formats was tested.



TECHNISCHE UNIVERSITÄT WIEN

## *Abstract*

Institut für Hochenergiephysik (HEPHY)/  
Conseil Européen pour la Recherche Nucléaire (CERN)

Diplomingenieur

### **Anwendung von Deep Learning auf die Rekonstruktion von Elektronen-Spuren im CMS-Experiment**

von Martin E.M. LOESENER DA SILVA VIANA, BSc.

Diese Arbeit schlägt eine Implementierung eines Deep Regression-Modells vor, um Elektronenspурparameter aus Kollisionsereignissen im CMS-Experiment am CERN vorherzusagen. Es ist vollständig in Python geschrieben, einer der beliebtesten Programmiersprachen im Bereich des maschinellen Lernens, und verwendet PyTorch, ein hochmodernes Deep-Learning-Framework, das für seine dynamische Graphenstruktur bekannt ist. Es wurden mehrere Architekturen verwendet, darunter die von Bernkopf vorgeschlagene Basisarchitektur (Masterarbeit in Bearbeitung) und Convolutional Neural Networks. Sie wurden mit einer Vielzahl von Algorithmen und Hyperparametern trainiert und getestet, um ihre Leistung zu beurteilen. Abhängig vom Parameter wurde eine **Reduzierung des Trainingsfehlers um einen Faktor 2 bis 3,5** in Bezug auf die von Bernkopf vorgeschlagene Konfiguration erreicht. Mit weniger als 5.000 Netzwerkparametern bietet das Modell ein leichtes und präzises Werkzeug zur Vorhersage von Elektronenspuren. Mit dem Ziel der automatisierten Vorhersage von Echtzeit-Teilchenspурdaten wurde ein mögliches Bottleneck beim Datenimport mit unterschiedlichen Datenformaten getestet.

## *Acknowledgements*

First and foremost I want to thank my supervisor Rudolf Fröhlich for his kind and expert support throughout every stage of my Thesis project. He was always committed to helping me make the most out of this work, be it through hours of detailed explanations and discussions or his flexibility in allowing me to participate in additional activities such as an Artificial Intelligence School in Rome where I could further strengthen and practice my Machine Learning skills.

I also want to thank Julian Gamboa for all the interesting discussions that often lead to new ideas. He is an amazing friend who I admire for his intelligence, scientific reasoning and deep sense of justice and morality. He is always able to stimulate my curiosity about a great variety of topics ranging from science and technology to politics and diplomacy.

Also, very special thanks to my parents and brothers who are always there when I need them. I have the best family I could wish for.

# Contents

<b>Declaration of Authorship</b>	<b>ii</b>
<b>Abstract</b>	<b>iv</b>
<b>Abstract</b>	<b>v</b>
<b>Acknowledgements</b>	<b>vi</b>
<b>1 Introduction to Artificial Intelligence</b>	<b>1</b>
<b>2 The CMS Experiment</b>	<b>3</b>
2.1 CMS Tracker . . . . .	5
2.2 Electron track reconstruction . . . . .	5
2.3 Contribution of this Thesis . . . . .	5
<b>3 Theoretical framework of Deep Neural Networks</b>	<b>6</b>
3.1 Neurons and neural networks . . . . .	7
3.1.1 Activation functions . . . . .	8
3.2 Optimization . . . . .	10
3.2.1 Stochastic Gradient Descent (SGD) . . . . .	12
3.2.2 Adam . . . . .	13
3.2.3 Resilient Backpropagation (Rprop) . . . . .	13
3.2.4 Levenberg-Marquardt . . . . .	13
3.3 Regularization . . . . .	14
3.3.1 Diagnosing overfitting . . . . .	14
3.4 Convolutional Neural Networks - Locality vs Globality . . . . .	15
3.4.1 Summary of CNN hyper-parameters . . . . .	19
3.5 Frontiers of theoretical Research on Neural Networks . . . . .	19
3.5.1 Probabilistic and statistical models . . . . .	20
3.5.2 Statistical inference . . . . .	23
3.5.3 Information Theory . . . . .	24
3.5.4 Topology and data analysis . . . . .	27
3.5.5 The curse of dimensionality and dimensionality reduction . . . . .	28
t-distributed Stochastic Neighbor Embedding (t-SNE) . . . . .	28
<b>4 Track parameter estimation by Kalman and Gaussian-sum filter</b>	<b>31</b>
4.1 Kalman filters . . . . .	31
4.2 Gaussian-sum filters . . . . .	33
4.2.1 Optimal Gaussian mixtures . . . . .	33
4.2.2 Number of components . . . . .	33

<b>5</b>	<b>CMS Data</b>	<b>34</b>
5.1	Structure of the data	34
5.2	Baseline	36
5.3	Tests	36
5.3.1	t-SNE dimensionality reduction	36
5.3.2	Test on Gaussian weights	38
5.3.3	Tests on correlation between targets and weighted averages	38
<b>6</b>	<b>Software Tools</b>	<b>41</b>
6.1	Docker	41
6.2	Jupyter Lab	44
6.3	Plotly/Dash	45
6.3.1	Plotly plot	46
<b>7</b>	<b>Step-by-step installation and execution Guide</b>	<b>47</b>
7.1	Installation	47
7.2	Executing a model	47
7.3	Dataset Generator	49
7.4	Tests	49
7.5	Parameter list	49
7.6	Tests on data formats performance	49
<b>8</b>	<b>Model architectures</b>	<b>50</b>
8.1	Base architecture	50
8.2	Convolutional Neural Network architectures	50
<b>9</b>	<b>Results</b>	<b>55</b>
9.1	Data formats performance	55
9.2	Models performance	55
9.3	Phase I - Optimization of base architecture	57
9.3.1	Adadelata	57
9.3.2	Adagrad	59
9.3.3	Adam	61
9.3.4	Adamax	63
9.3.5	ASGD	65
9.3.6	RMSprop	67
9.3.7	Rprop	69
9.3.8	SGD	71
9.3.9	Levenberg-Marquardt	73
9.4	Phase II - Addition of Convolutional Layers	75
<b>10</b>	<b>Conclusion</b>	<b>77</b>
	<b>Bibliography</b>	<b>78</b>

# List of Figures

2.1	Design of CMS Detector . . . . .	4
2.2	Silicon pixels . . . . .	4
2.3	Silicon strips . . . . .	4
3.1	Representation of a layer of neurons . . . . .	7
3.2	Example neural network . . . . .	8
3.3	Landscapes of convex and non-convex objective functions . . . . .	11
3.4	Effect of varying learning rates on convergence . . . . .	12
3.5	Effect of application of convolutional filters to images . . . . .	16
3.6	Successive application of convolutions in a CNN . . . . .	16
3.7	Depiction of a 2-dilated kernel . . . . .	18
3.8	"Graph example" by Römert, located at <a href="https://commons.wikimedia.org/wiki/File:Graph_example_(Graph_theory).png">https://commons.wikimedia.org/wiki/File:Graph_example_(Graph_theory).png</a> , Creative Commons Attribution-Share Alike 3.0 Unported License . . . . .	21
3.9	Directed graph from x to y . . . . .	21
3.10	Directed graph from y to x . . . . .	21
3.11	Undirected graph between x and y . . . . .	22
3.12	Comparison of Student's t-distribution with degrees of freedom $\nu = 1, 5$ and 10 with a Gaussian distribution . . . . .	29
5.1	Feynman diagrams of bremsstrahlung process in first order Born approximation . . . . .	34
5.2	Fractional energy loss distribution for several t-values . . . . .	35
5.3	Track inputs . . . . .	36
5.4	Track target . . . . .	36
5.5	t-SNE dimensionality reduction of particle track data (perplexity=5) . . . . .	37
5.7	t-SNE dimensionality reduction of particle track data (perplexity=50) . . . . .	37
5.6	t-SNE dimensionality reduction of particle track data (perplexity=30) . . . . .	38
5.8	Weighted average of parameter 1 (q/p) vs. its target . . . . .	39
5.9	Weighted average of parameter 2 vs. its target . . . . .	39
5.10	Weighted average of parameter 3 vs. its target . . . . .	40
6.1	Typical jupyter lab front end . . . . .	44
8.1	Shapes of parameters in base architecture . . . . .	51
8.2	Shapes of parameters in undilated CNN architecture . . . . .	52
8.3	Shapes of parameters in dilated CNN architecture with kernel size 2 . . . . .	53
8.4	Shapes of parameters in dilated CNN architecture with kernel size 3 . . . . .	54
9.1	Adadelta training loss history for lr = 0.1 and batch size = 32 . . . . .	58
9.2	Adadelta validation loss history for lr = 0.1 and batch size = 32 . . . . .	58
9.3	Adagrad training loss history for lr = 0.01 and batch size = 128 . . . . .	60
9.4	Adagrad validation loss history for lr = 0.01 and batch size = 128 . . . . .	60
9.5	Adam training loss history for lr = 0.0001 and batch size = 64 . . . . .	62

9.6	Adam validation loss history for lr = 0.0001 and batch size = 64 . . . . .	62
9.7	Adamax training loss history for lr = 0.001 and batch size = 64 . . . . .	64
9.8	Adamax validation loss history for lr = 0.001 and batch size = 64 . . . . .	64
9.9	ASGD training loss history for lr = 0.01 and batch size = 32 . . . . .	66
9.10	ASGD validation loss history for lr = 0.01 and batch size = 32 . . . . .	66
9.11	RMSprop training loss history for lr = 0.001 and batch size = 32 . . . . .	68
9.12	RMSprop validation loss history for lr = 0.001 and batch size = 32 . . . . .	68
9.13	Rprop training loss history for lr = 0.001 and batch size = 512 . . . . .	70
9.14	Rprop validation loss history for lr = 0.001 and batch size = 512 . . . . .	70
9.15	SGD training loss history for lr = 0.01 and batch size = 128 . . . . .	72
9.16	SGD validation loss history for lr = 0.01 and batch size = 128 . . . . .	72
9.17	Levenberg-Marquardt training loss history for $\lambda = 1$ and batch size = 64 (slight bias towards Gauss-Newton Method) . . . . .	74
9.18	Levenberg-Marquardt validation loss history for $\lambda = 1$ and batch size = 64 (slight bias towards Gauss-Newton Method) . . . . .	74
9.19	Adam training loss history for undilated CNN with lr = 0.0001 and batch size = 32 . . . . .	76
9.20	Adam validation loss history for undilated CNN with lr = 0.0001 and batch size = 32 . . . . .	76

# List of Tables

3.1	Activation functions	9
3.2	Summary of CNN hyper-parameters	19
5.1	Mean, maximum and minimum of the sum of Gaussian weights across all tracks	38
9.1	Time to import data and save it in a numpy array	55
9.2	Peak memory to import data and save it in a numpy array	55
9.3	Performance of Adadelta optimizer for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE	57
9.4	Performance of Adagrad optimizer for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE	59
9.5	Performance of Adam optimizer for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE	61
9.6	Performance of Adamax optimizer for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE	63
9.7	Performance of ASGD optimizer for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE	65
9.8	Performance of RMSprop optimizer for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE	67
9.9	Performance of Rprop optimizer for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE	69

9.10 Performance of SGD optimizer for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE . . . . .	71
9.11 Performance of Levenberg-Marquardt optimizer with slight bias towards a Gauss-Newton algorithm for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, under several $\lambda$ s and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE . . . . .	73
9.12 Performance for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, using the CNN architectures proposed in Chapter 8 and the optimal optimization algorithms and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE . . . . .	75



# List of Abbreviations

<b>CMS</b>	<b>Compact Muon Solenoid</b>
<b>ML</b>	<b>Machine Learning</b>
<b>DL</b>	<b>Deep Learning</b>
<b>ANN</b>	<b>Artificial Neural Network</b>
<b>CNN</b>	<b>Convolutional Neural Network</b>
<b>MLP</b>	<b>Multi-Layer Perceptron</b>
<b>ReLU</b>	<b>Rectified Linear Unit</b>
<b>MSE</b>	<b>Mean Squared Error</b>
<b>t-SNE</b>	<b>t-distributed Stochastic Neighbor Embedding</b>
<b>LM</b>	<b>Levenberg-Marquardt</b>
<b>Rprop</b>	<b>Resilient Backpropagation</b>
<b>ONNX</b>	<b>Open Neural Network Exchange</b>



## Chapter 1

# Introduction to Artificial Intelligence

Artificial Intelligence (AI), or rather Machine Learning for the time being, is the manifestation of decades of thorough research into the fundamental nature of learning. It encompasses a wide range of applications from relatively simple tasks like object recognition to complex ones like fully automated control systems in self-driving cars. Its role in the scientific endeavour and technological progress is ever increasing, given that most interesting problems in science and technology are non-linear, complex and compositional in nature, meaning that a model capturing several layers of abstraction, as is the case of neural networks, have a great potential to solve them.

Although there is currently much hype around the field, it is not a new idea. Understanding how Artificial Intelligence was born and how it evolved over time can be helpful to understand current developments in the field and the reason behind the use of today's state-of-the-art techniques. To serve this purpose, some of the most important and relevant historical developments will be now highlighted. Goodfellow et al. (2016) describe three crucial developmental phases of AI that are paraphrased and discussed as follows. For readers interested in a more comprehensive historical account, rich literature is available on the topic (for instance Nilsson, 2009).

Artificial Intelligence Research greatly fluctuated in popularity over time and changed its focus and paradigm repeatedly. As aforementioned, one can group its history into three sections of rising popularity: *cybernetics* in the 1940s to the 1960s, *connectionism* in the 1980s to 1990s and the *Deep Learning* era starting in 2006 with the discovery of a fast learning algorithm for Deep Belief Networks (Hinton, Osindero, and Teh, 2006) and lasting to the time of this writing.

Initially, Artificial Intelligence was fundamentally inspired by the biology and psychology of the learning mechanisms in the brain. McCulloch and Pitts (1943) described a biological neuron from the perspective of propositional logic, which marked the beginning of the cybernetics phase. In 1950 (Turing) the idea of a thinking machine was introduced and the Turing test was developed to assess the intelligence of a computer. Soon various machine learning programs followed, making use of the perceptron (Rosenblatt, 1958) that "sensed" a signal and produced a binary output based on a threshold value. A neuron is a similar, but more generalized concept for continuous values. Of course, such linear operators on their own have great limitations because most interesting problems are non-linear and a single perceptron/neuron can only learn one representation in a linear fashion. This concluded

the cybernetics period.

In the 1980s researchers realized that by combining various neurons in several layers, many representations of the input data could be learned in a distributed way, forming layers of abstraction that could generalize concepts. Any Borel measurable function from one finite dimensional space to another can be approximated to an arbitrary accuracy given a multilayer feedforward neural network with sufficient layers (Hornik, Stinchcombe, and White, 1989). With that idea, and the discovery of backpropagation (Rumelhart, Hinton, and Williams, 1986) that allowed for automatic gradient computation and parameter updates, connectionism was born. AI was the big promise of its time, but the lack of computational power and data impeded its progress. Soon the great dreams and ambitions of creating intelligent devices and especially creating an artificial brain, capable of solving problems like humans do, were partly abandoned because investor expectations could not be met.

However, with a rapidly developing computational infrastructure and exponentially increasing amounts of data, soon the rules of the game would change again. In 2006 (Hinton, Osindero, and Teh), with the discovery of a fast learning algorithm for Deep Belief Networks, there was a third big resurgence that lasts until the time of this writing. This time it came under the name of *Deep Learning*. Several exciting new discoveries have been made and interesting applications found. Also, with the aim of boosting the development of the field, large datasets like ImageNet (containing tens of millions of images) were made publicly available and competitions with prizes were held out and are still being held out. In 2015 Elon Musk made a staggering investment of 1 billion dollars into *Open AI*, a non-profit research institution that seeks the development of safe and human-friendly artificial general intelligence.

Today AI is transforming the way technological problems are solved in a fundamental way. It has proven its efficacy in a variety of applications, such as in object recognition for self-driving cars, speech recognition for virtual assistants like Cortana or Siri, and fundamental research like in cancer diagnostics.

Physics has also benefited from it and will do much more so in the future. To meet the challenges of an increasing amount of particle collisions and the data generated as a consequence, CERN has committed to expanding its use of machine learning algorithms to hopefully increase the speed of particle track reconstructions (Castelvecchi, 2018).

## Chapter 2

# The CMS Experiment

The following information about the Compact Muon Solenoid (CMS) Experiment largely stems from CERN's website (*About CMS*). The LHC at CERN is a 27 km particle accelerator; the largest and most powerful of its kind. It creates two opposing beams of accelerated protons and ions (*LHC collides ions at new record energy*). Protons reach about  $6.5\text{TeV}$  of energy, or equivalently 99.9999991% of the speed of light. When the beams collide at one of the four collision points (ALICE, ATLAS, CMS and LHCb), they do so with twice the energy of each beam (approximately  $13\text{TeV}$ ). In the course of those high-energy events, the energy is transformed into mass, and new particles that could potentially hint at new Physics beyond the Standard Model are created.

As aforementioned, CMS is positioned at one of the four collision points and possesses general-purpose high-tech particle detectors that were specifically developed for the extreme conditions inside the apparatus. After a collision takes place, all the tracks of all stable particles are detected, and their momenta and energies determined, with the hope of discovering something new. With its fast-response detectors and electronics the experiment is able to capture 40 million data points every second. Out of this massive amount of information, potentially interesting events are filtered out.

To obtain meaningful information about the particles, it necessary to measure their momentum and energy. Momentum is measured through forcing charged particles into a spiral movement following a helix by applying a potent magnetic field and subsequently measuring its curvature in the so-called CMS Tracker, while energy is measured in Calorimeters (ECAL and HCAL) surrounding the Tracker through scintillation and photon energy measurement. Muons pass the calorimeters and are measured at a posterior stage in special chambers (muon chambers). All of those measurements are only possible thanks to the strong magnetic field the particles are submerged in. The Tracker and Calorimeters are enclosed inside a  $3.8\text{T}$  superconducting Solenoid, carrying roughly  $18,000\text{A}$ , the most powerful magnet ever constructed, and the one that gives CMS its name. Each of the components of the experiment can be seen in figure 2.1. Given that this Thesis is a CMS Tracker project, some more details about the Tracker will be now explained.

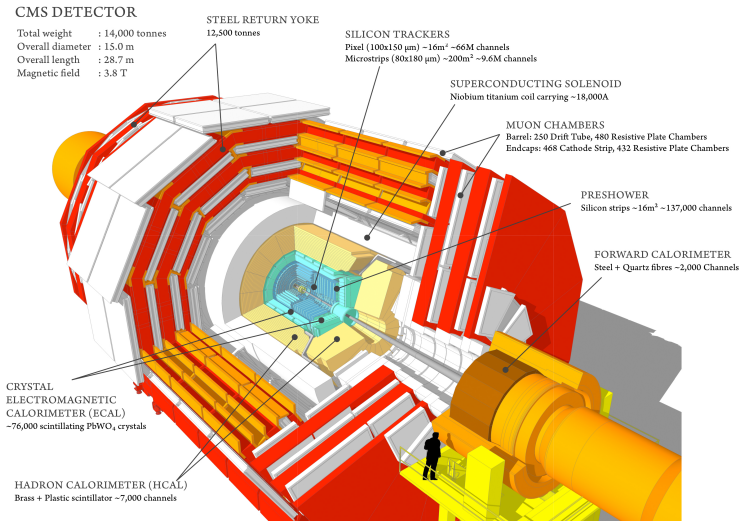


FIGURE 2.1: Design of CMS Detector

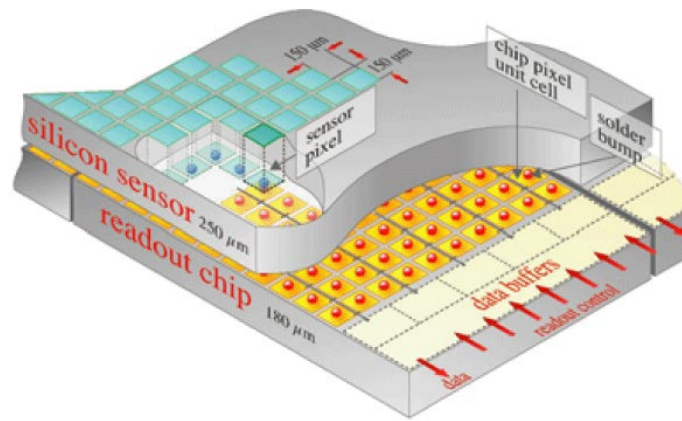


FIGURE 2.2: Silicon pixels

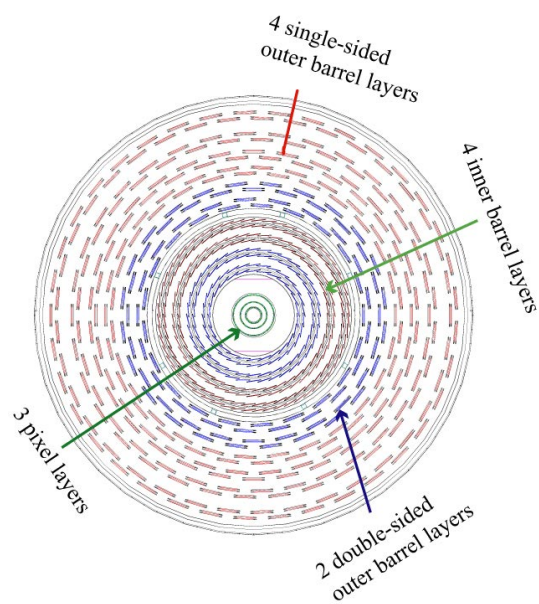


FIGURE 2.3: Silicon strips

## 2.1 CMS Tracker

The silicon Tracker is at the core of the CMS Experiment, where the track density is highest. It consists of various layers of silicon pixels (figure 2.2) and silicon strips (figure 2.3). Silicon detectors are used due to their fast response and high spatial resolution (about  $10\mu\text{m}$  for each measurement). After a particle has created an electron-hole pair, one of the tens of thousands of APV25 microchips (Raymond et al., 2000), equipped with low-noise amplifiers, amplifies the signal. Position and momentum are determined and advanced software is used for track/vertex reconstruction.

## 2.2 Electron track reconstruction

In order to accurately reconstruct tracks of electrons, one must account for their energy loss through both ionization and radiation (bremsstrahlung). For the latter, a commonly used model is the Bethe-Heitler energy distribution. In an attempt to account for this effect, and because Kalman filters used in CMS rely on Gaussian distributions, Frühwirth (2003) developed a Gaussian mixture approximation that was later implemented. A thorough analysis on the implementation of Gaussian-Sum-Filters for electron reconstruction in the CMS Tracker can be found in the work of Adam et al. (2005).

## 2.3 Contribution of this Thesis

The contribution of this Thesis to CMS is an implementation of an electron track reconstruction algorithm using Deep Neural Networks (LeCun, Bengio, and Hinton, 2015). It is completely written in Python, one of the most popular programming languages in the field of machine learning, and makes use of PyTorch, a cutting-edge Deep Learning framework. Several architectures were used, including the base architecture proposed by Bernkopf in his Master's Thesis (under revision) and Convolutional Neural Networks. They were trained and tested using a variety of algorithms and hyper-parameters to assess their performance. The baseline for performance is set to be Bernkopf's most successful model (trainlm) which is a Levenberg-Marquardt algorithm. With automated predictions of streaming particle track data in mind, a possible bottleneck regarding data imports using different data formats was tested.

### Why Convolutional Neural Networks?

Convolutional Neural Networks (CNNs) were originally developed based on findings about the visual cortex of monkeys (Hubel and Wiesel, 1968). One of its most prominent use case examples can be found in the field of object recognition. Images have spatial patterns, for which CNNs are perfectly suited. However, the network can be used in much more general scenarios. One of its significant advantages over Feedforward Neural Networks is its ability to capture locality in features, and locality can be encoded into non-visual data as well. The provided electron track data is composed of twelve Gaussian components that are interrelated, and it is this relationship that conveyed the inspiration for exploring CNNs for track reconstruction. Details on the relationships between features are presented in Chapter 5.

## Chapter 3

# Theoretical framework of Deep Neural Networks

Before diving deep into the structure and functioning of neural networks, let us first introduce the kind of problems they are designed to solve. In practice there is input data describing the state of some object of study. Each variable that describes this state is called a *feature* and features can be organized into a vector  $x$ , the *feature vector*, that is an element of the *feature space*  $F$ . There are several instances  $x_i$  of such vectors and the set of all such instances is the input data. Each  $x_i$  is an example from which the neural network can extract information and learn a concept. There are two ways of learning, namely *Supervised Learning* and *Unsupervised Learning*. The ideas and purposes behind them are

- in Unsupervised Learning to extract some meaningful information from the mentioned input data to get some insight, or as pre-training. This the case, for instance, in clustering algorithms and dimensionality reduction tools like t-SNE which will be thoroughly discussed later in this chapter.
- in Supervised Learning to use input data, along with an associated output value to learn their relationship and be able to predict a desired output value for unseen, new inputs. In other words, we seek a functional relationship (although in general highly non-linear) given by a network with optimally tuned architecture and weights that maps a given new input  $x_{new}$  to an output  $y = f(x_{new})$ .

In the course of the present work, both techniques have been used for different purposes. The first to visualize in two dimensions the data that was parsed from an original file and verify that the parsing was successful, and the second one to actually predict the values of a particle's track parameters, i.e. the charge over momentum, the azimuthal angle, the polar angle and the transverse and longitudinal impact parameter, as will be explained in more detail in Chapter 5.

Just for the sake of completeness, a prediction within the framework of Supervised Learning in general can happen in two ways:

- *Classification*: The output value is discrete and called label. For example one might want to segment data about the human brain into several categories.
- *Regression*: The output value is continuous and called target. This is the case of this work. Particle's track parameters are evidently continuous.

We will be dealing with neural networks in this work. In particular with Feedforward Neural Networks and Convolutional Neural Networks (CNNs). Those networks form part of Deep Learning. They have greatly gained momentum lately for



their surprising predictive power. For clarification, Machine Learning is a subfield of Artificial Intelligence and Deep Learning a subfield of Machine Learning, but tracing clear boundaries can be tricky. There are other forms of machine learning as well, like Support Vector Machines (SVMs) and Genetic algorithms.

With that said, we will now examine the structure and workings of a neural network, starting from its fundamental building block: the neuron.

### 3.1 Neurons and neural networks

Perceptrons (Rosenblatt, 1958) take in the features  $x_i$  and produce a binary output based on a threshold. Neurons generalize this concept to real values. Each neuron produces a scalar value and a layer of them produces a linear transformation of the form

$$f(x_i) = \sum_i w_{ij} x_i + b_j. \quad (3.1)$$

Each neuron also contains an activation function that allows a value to pass if it is large enough and does not if it is too small. The idea behind such a function is that the information gets only passed on, if the signal is strong enough, just like it would happen in a human neuron. How exactly activation functions control the values passing through will be discussed at a later stage. In equation 3.1,  $w_{ij}$  is the weight matrix and  $b_j$  a bias.

A layer of neurons is depicted below:

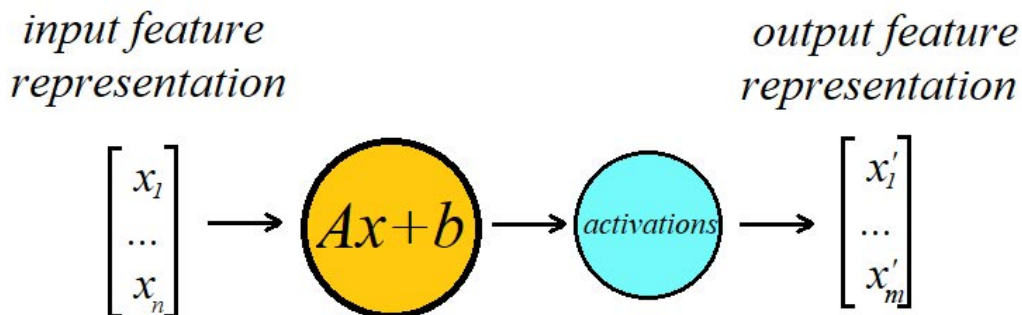


FIGURE 3.1: Representation of a layer of neurons

The output of a neuron layer is an alternative *representation* of a feature vector and is itself a feature vector that can be fed into another neuron layer. Connecting several neuron layers forms a neural network.

Now, it is probably easiest to explain the functioning of a neural network in terms of an example, that is nevertheless extremely simplified for a better understanding. The general idea however translates perfectly into real networks. Let us consider the following problem: We want to discriminate images of real persons from random objects. There are different layers of abstraction and therefore feature representations to be learned in order to tell them apart. A person has eyes, ears, a nose, a mouth, and many more facial parts. Each of those have a certain color and geometrical shapes that can themselves be broken down into more simple characteristics like lines, circles and edges. To build a human face and tell it apart from other objects, several

layers of feature representations need to be combined in just the right way. For the sake of argument, let us assume that the weights in the transformation matrix  $w_{ij}$  and the bias  $b_i$  to produce one desired feature representation are already known. Let us also assume that we know how many feature representations are needed. Then we can build a network of neurons that detects (after passing the activation function) the presence of each of those features, combines them in just the right way and finally outputs a binary answer of whether the image shows a human or not. For this example we will use the network in figure 3.2.

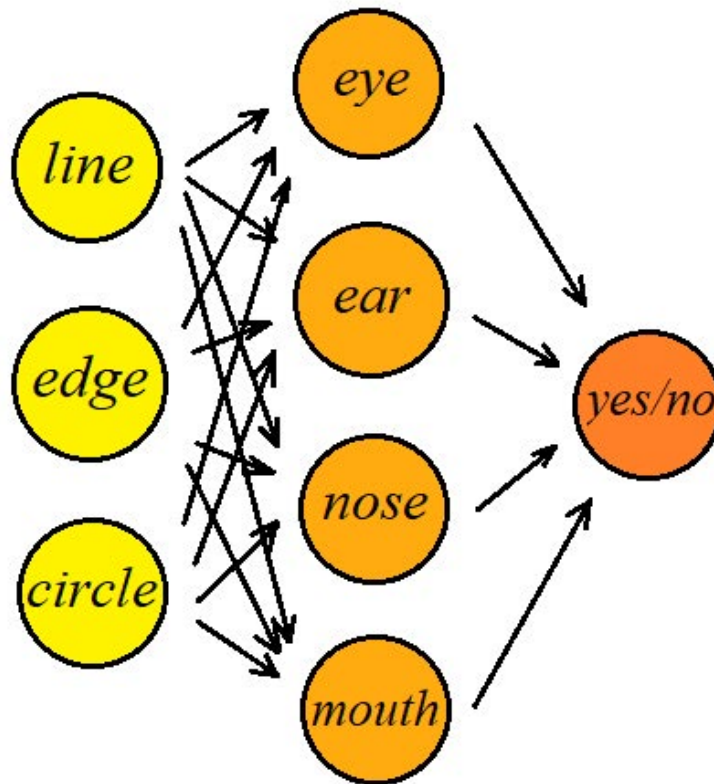


FIGURE 3.2: Example neural network

Each neuron detects the existence of a feature. If we input a picture of a female model where her ears are not visible, the following would happen. All neurons in the first layer would fire with more or less intensity. The features are combined in different ways to form structures like mouths, eyes, ears and noses that are detected in the second layer. In the second layers all neurons would fire, except the ear-feature neuron. Finally, although an ear was not detected, the other features were, and that is enough for the network to determine the existence of a human face. For regression, something similar happens, this time the representation being a numerical value and being averaged with different weights.

### 3.1.1 Activation functions

There are several kinds of activation functions. Some examples are shown in table 3.1.

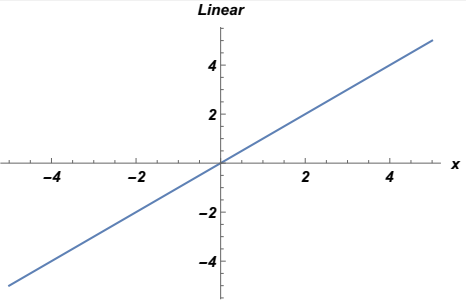
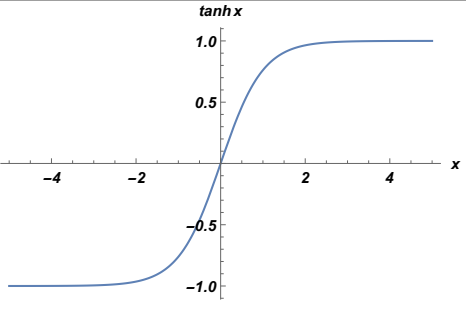
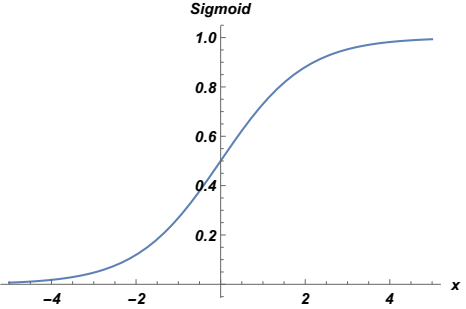
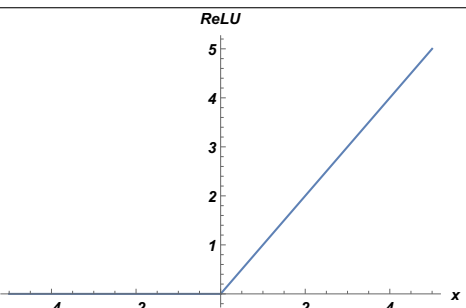
Activation function	Mathematical representation	Graphical representation
Linear	$activation(x) = x$	
Tanh	$activation(x) = \tanh(x)$	
Sigmoid	$activation(x) = \frac{1}{1+e^{-x}}$	
ReLU	$activation(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$	

TABLE 3.1: Activation functions

Many problems in science and technology are non-linear, and the reason activation functions are powerful is that they introduce non-linearities into the network. According to the *universal approximation theorem*, any Borel measurable function from one finite dimensional space to another can be approximated to arbitrary accuracy by a multilayer feedforward neural network with sufficient layers (Hornik, Stinchcombe, and White, 1989). A linear activation function can be hardly considered an activation because its whole point is to introduce non-linearities. Originally, the sigmoid function was widely used because it is biologically plausible. However, because sigmoids are not zero-centered, they tend to introduce a bias into the gradients used for weight updates. Of course a sigmoid can be shifted to be zero-centered, but this would add to the computational complexity. The *tanh* activation solves this problem. Although not biologically plausible, it is zero-centered and very similar to a sigmoid. Two problems that both of them have however, is for one part the vanishing gradient problem, and for another sparsity. The vanishing gradient problem refers to the fact that departing from the origin, the gradient starts vanishing very quickly, which can cause the algorithm to stop learning. A lack of sparsity refers to small changes in the inputs causing many neurons to change accordingly, even if the change should affect just a few. For a network's ability to generalize concepts, sparsity is a favourable property. Glorot et al. (Glorot, Bordes, and Bengio, 2011) show that Rectified Linear Units improve sparsity and thereby the performance of a network, despite its non-differentiability at the origin. They also solve the vanishing gradient problem for positive values and are computationally less expensive. For non-vanishing gradients in the negative domain, leaky ReLU activations can be considered. The only difference with respect to normal ReLUs is a slightly tilted linear region (with small but positive slope) in the negative domain. Here are some of the reasons why, according to the mentioned paper, sparsity is a desirable property:

- Information disentangling: In a sparse network, small changes in inputs only translate to changes in some neurons, affecting only some of the entries of feature representation vectors. This way information can be disentangled more easily.
- Efficient variable-size representations: Inputs vary in information content and would be better represented by representations that fit their size, while in dense networks representations would be excessively large.
- Linear separability: Representations are more easily separable, requiring less non-linearities.
- Distribution: According to Bengio (Bengio, 2009) dense, distributed representations are exponentially more efficient than local ones, but Glorot et al. argue that sparse and distributed representations are again exponentially more efficient.

## 3.2 Optimization

Mathematical optimization has the goal to either maximize or minimize a so-called *objective function*. The objective function in machine learning is usually called a loss or cost function that is to be minimized and is a measure of deviation between predictions and target values. To conserve the mathematical generality and avoid using

different names to refer to the same, from now on we will use the term objective function or just objective. One of them, which is in fact the one used in this work, is the mean squared error, which is defined as follows:

$$MSE = \frac{\sum_{i=1}^n (y_i - y_{i,target})^2}{n} \quad (3.2)$$

$y_i$  correspond to the prediction values. We can talk about two types of optimization problems: convex and non-convex ones. They are visualized in figure 3.3.

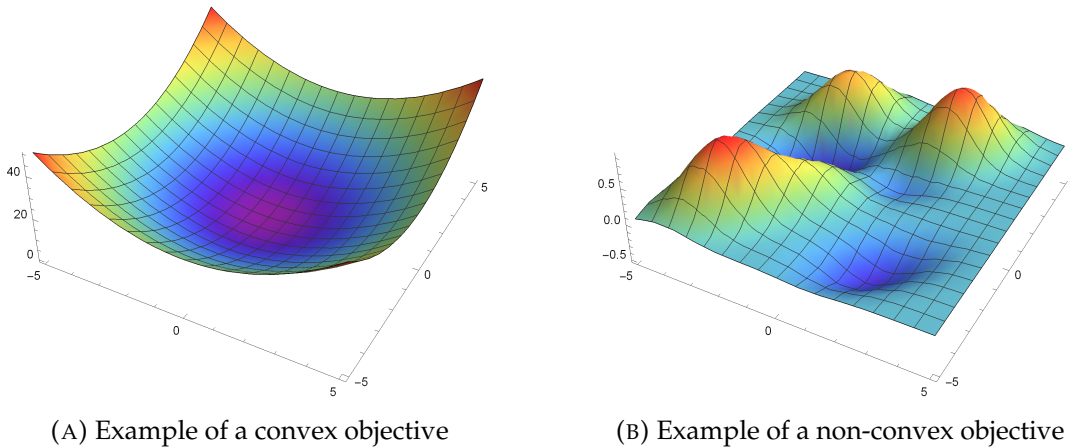


FIGURE 3.3: Landscapes of convex and non-convex objective functions

Convex problems provide a guarantee for a global minimum of the objective as in figure 3.3a. They arise for instance in linear regression problems and can be solved by Gradient Descent and related methods. The minimum is located where the gradient vanishes. In non-convex problems like the one depicted in figure 3.3b however, there is no unique minimum. A vanishing gradient therefore only indicates local optimality. Most problems are non-linear, non-convex and NP-hard to solve (meaning they cannot be solved exactly in polynomial time by a given algorithm). Any algorithm attempting to find the global minimum of an objective, can get stuck in a local minimum. Some are less likely to do so, but since there is no optimality-guarantee, we can generally not speak of absolute optimality, but only of "good enough" solutions.

In figure 3.3, for simplicity, a hypothetical bivariate objective was visualized. However, in a neural network, an objective landscape is very high-dimensional. In fact its dimension corresponds to the number of weights. The weights are the function's parameters and we talk about the function as being embedded in *parameter space*. We will henceforth denote the parameters of a neuron by  $w_{i,j}$ .

In the neural network example provided in the previous section, we have assumed that all feature representations in all neurons are fixed and known. To get to this final configuration, the network has to be trained. In order to do that, an optimization algorithm needs to iteratively update the parameters. There are different update rules with their respective up- and downsides, but most of them make use of the notion of gradient descent (Cauchy, 1847). An update rule is herein denoted by  $\Delta w_{i,j}$ .

Gradient descent is based on the idea that the gradient of an objective points in the direction of largest ascent. Therefore, taking a step in its opposite direction is likely to diminish its value. This, however, depends on the step size. The parameter update for gradient descent is given by the following equation

$$\Delta w_{i,j} = -\eta \nabla_{i,j} J \quad (3.3)$$

where  $J$  is the objective and  $\eta$  a hyper-parameter called learning rate.  $\eta$  controls the step size. If it is too large the algorithm might diverge; if it is too small it restrains convergence speed. This is illustrated by figure 3.4. Hence, finding an appropriate learning rate is crucial. The mechanism of backpropagation (Rumelhart, Hinton, and Williams, 1986) allows to assess how much the weights and the objective function have changed in an iteration. With those two pieces of information, the partial derivatives in equation 3.3 can be numerically computed. Backpropagation is an essential element of neural networks.

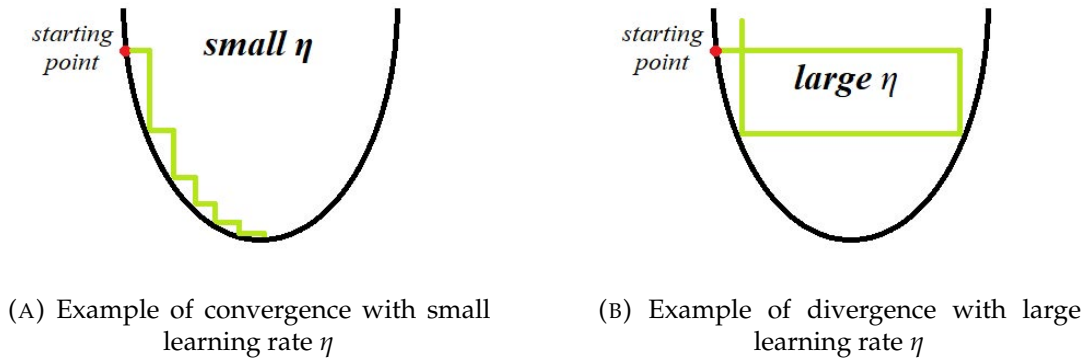


FIGURE 3.4: Effect of varying learning rates on convergence

Given that so many machine learning optimizers rely on gradient descent, it is also interesting to look at the signal-to-noise ratio (SNR) of the gradient. It can be defined as follows:

$$SNR = \frac{mean(\nabla_{i,j} J)}{std(\nabla_{i,j} J)} = \frac{\frac{1}{n} \sum_i \sum_j \nabla_{i,j} J}{\sqrt{\frac{1}{n} (\nabla_{i,j} J - \nabla_{i,j} J)^2}}. \quad (3.4)$$

A large SNR indicates a high certainty of moving in the correct direction and vice versa. In a converging problem, after many iterations, the SNR should decrease.

All of the methods we are going to present are first-order methods, meaning that they only make use of the gradient and not the Hessian or higher order derivatives. Higher order methods are not only computationally much more expensive, but they often do not work in practice. We will now introduce three common optimizers that were used in this work, namely the Stochastic Gradient Descent (SGD), the Adam optimizer and Resilient Backpropagation (Rprop). We will also introduce the Levenberg-Marquardt algorithm which yielded the optimal results in Bernkopf's Master's Thesis (under revision).

### 3.2.1 Stochastic Gradient Descent (SGD)

SGD (Robbins and Monro, 1951) only differentiates itself from common Gradient Descent in that it only uses samples of the training set for parameter updates, instead of the complete dataset. This significantly accelerates convergence speed and is also useful for generalization.

### 3.2.2 Adam

The Adam optimizer (Kingma and Ba, 2014) is a quite recent state-of-the-art optimizer that makes use of stochastic gradient descent and lower-order moments estimations to effectuate the parameter updates. It is suitable for gradients with high SNR, and is efficient in terms of computational and storage complexity.

Regarding the implementation, after estimating the first moment  $m_{i,j}$  and the second moment  $v_{i,j}$  which make use of the gradient  $\nabla_{i,j}J$  and introduce some momentum so that local minima can be surpassed, the parameter updates are given by

$$\Delta w_{i,j} = -\eta \frac{m_{i,j}}{\sqrt{v_{i,j} + \varepsilon}} \quad (3.5)$$

where  $\varepsilon$  is a small number (default initialization:  $10^{-8}$ ) and  $\eta$  the learning rate. For further details about the precise moment estimation, please refer to the mentioned paper.

### 3.2.3 Resilient Backpropagation (Rprop)

The Rprop algorithm (Riedmiller and Braun, 1992) proposes the following update rule:

$$\Delta w_{i,j}(t) = \begin{cases} \Delta w_{i,j}(t-1) \eta^+ & , \text{ if } \nabla_{i,j}J(t-1) \nabla_{i,j}J(t) > 0 \\ \Delta w_{i,j}(t-1) \eta^- & , \text{ if } \nabla_{i,j}J(t-1) \nabla_{i,j}J(t) < 0 \\ \Delta w_{i,j}(t-1) & , \text{ otherwise} \end{cases} \quad (3.6)$$

with  $0 < \eta^- < 1 < \eta^+$ . As opposed to the Manhattan rule, this method adapts to local changes in the objective function and one of its strengths is that it usually converges very quickly.

### 3.2.4 Levenberg-Marquardt

The Levenberg-Marquardt algorithm is an efficient optimization algorithm that makes use of first order quantities like the Jacobian of the objective (transpose of its gradient in case of a one-dimensional objective) to compute second order approximations. It is therefore a second order method with the complexity of a first order method. In Bernkopf's Master's Thesis (under revision) this algorithm yielded the best results, which are taken as the baseline of the present work. In the present work a Levenberg-Marquardt optimizer was implemented and will be discussed in Chapter 9.

Let us now examine the exact functioning of the algorithm. For a second order approximation, the Hessian of the objective is needed. It can be approximated as

$$H = J^T J + \lambda I, \quad (3.7)$$

where  $J$  denotes the Jacobian of the objective,  $I$  the identity matrix and  $\lambda$  a scalar that controls the contribution of the identity matrix (more on that later). From there the update rule follows to be

$$\Delta w_{i,j} = -H^{-1} J^T = -(J^T J + \lambda I)^{-1} J^T. \quad (3.8)$$

Now let us consider the cases of a vanishing and a large  $\lambda$  value. If  $\lambda$  is very large the  $J^T J$  contribution to the Hessian becomes negligible, essentially yielding a gradient descent approximation, as follows:



$$\Delta w_{i,j} \approx -(\lambda I)^{-1} J^T = -\left(\frac{1}{\lambda}\right) J^T = -\left(\frac{1}{\lambda}\right) \nabla L = -\eta \nabla L. \quad (3.9)$$

If  $\lambda$  vanishes, the  $J^T J$  contribution to the Hessian dominates and the weight update becomes

$$\Delta w_{i,j} = -(J^T J)^{-1} J^T, \quad (3.10)$$

which corresponds to a Gauss-Newton update. That means the Levenberg-Marquardt algorithm is essentially a combination of Gradient Descent and a Gauss-Newton algorithm, and  $\lambda$  intrinsically controls how much trust is given to each algorithm.

The remaining question is the one of  $\lambda$ 's choice. A Gauss-Newton approximation works best near a minimum, while Gradient Descent works best far away from it. It is a common implementation to decrease  $\lambda$  by a given factor when the objective decreases after an iteration, and increment it by the same factor if the objective increases.

In order to compare Bernkopf's results with other optimizers, the author decided to write an additional Levenberg-Marquardt optimizer with and without adaptive momentum. The version without adaptive momentum contains a slight bias towards the Gauss-Newton Method, favouring  $\lambda$ 's change in this direction (reduction). The version with momentum (Ampazis and Perantonis, 2000) could not be executed because the update matrices turn out to be the inverse of a singular matrix.

### 3.3 Regularization

A common issue in the realm of machine learning is the lack of generalization, commonly known as overfitting. Once a model becomes too complex, it becomes less likely to predict new, unseen data because it starts memorizing data instead of recognizing patterns in it. Although theoretically not entirely understood, some possible explanations for this can be found in section 3.5 where some research identifying different training phases and generalization mechanisms (Shwartz-Ziv and Tishby, 2017b) are explained.

There are ways to diagnose overfitting and counteract it by either modifying the model architecture or introducing some penalty or regularization technique.

#### 3.3.1 Diagnosing overfitting

Overfitting can be diagnosed by validation and testing. To this end, after randomizing the dataset—to avoid learning a particular sequential order—a section of it is reserved for validation and another section for testing. Validation and testing are essentially the same procedure, where predictions from the respective dataset are generated and the deviation from target values computed using the same loss function as for training. The only difference between the two is that validation happens repeatedly after a given number of training iterations, while the test happens once the training is completed. During both assessments, no optimization happens.

In the end, besides the training loss history, also a validation loss history and a test loss are available and can be examined. An excess of the validation loss over the training loss is an indicator for overfitting. Also, if the test loss is excessively large, the model was clearly not able to generalize to new data.

If that is the case, some measures can be taken. One approach is to penalize overly



large parameters by adding some measure of the parameter size to the loss function. This can be done using several norms, but the most common implementations are the L1 and L2 regularizations used in *Lasso* and *Ridge* Regression respectively.

L1 penalization in Lasso Regression:

$$L = L_0 + \lambda \|\theta\|_1^2 \quad (3.11)$$

L2 penalization in Ridge Regression:

$$L = L_0 + \lambda \|\theta\|_2^2 \quad (3.12)$$

$\theta$  in the above equations denotes the parameters.

Another very effective and commonly used state-of-the-art regularization technique is called Dropout (Hinton et al., 2012). It basically consists of random data sampling between layers and thereby forces the network to ignore irrelevant information. This method will later be used in case it is needed.

### 3.4 Convolutional Neural Networks - Locality vs Globality

Besides common neurons, there are other operations that can be performed upon input features. One of them is the convolution operation. This idea was originally inspired by findings about the visual cortex in monkeys (Hubel and Wiesel, 1968), and we will see how it applies to neural networks and what its use cases are.

Let us first contemplate the concept of a cross-correlation between two functions  $f(x), g(x) \in \mathbb{R}^d$  that is defined as follows:

$$(f \star g)(x) := \int_{\mathbb{R}^d} f(\xi) g(x + \xi) d\xi. \quad (3.13)$$

In the discrete case it is given by

$$(f \star g)(x) := \sum_{\xi} f(\xi) g(x + \xi). \quad (3.14)$$

This is a measure of how related two functions are. When both are simultaneously large and share the same sign, the summand is large and positive. On the other side, if the functions have opposite signs, the summand is negative and makes the overall integration or sum smaller, resulting in a smaller correlation. In other words, cross-correlation is a measure of similarity between functions. The convolution is a very similar concept, but with the reversal of one of the functions, i.e.:

$$(f * g)(x) := \int_{\mathbb{R}^d} f(\xi) g(x - \xi) d\xi, \quad (3.15)$$

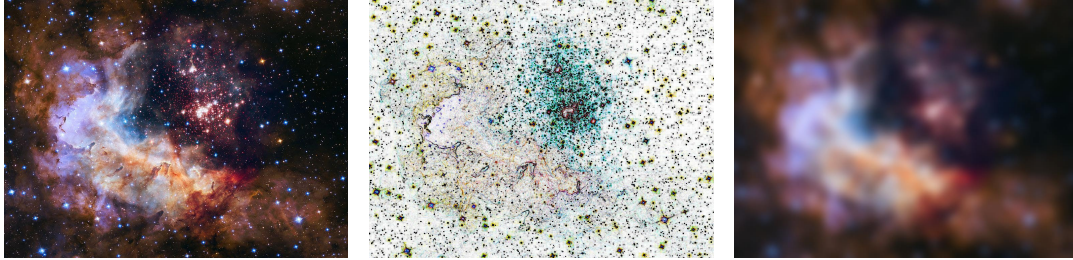
or in the discrete case

$$(f * g)(x) := \sum_{\xi} f(\xi) g(x - \xi). \quad (3.16)$$

The difference between both concepts is only formal, not practical. Both are a measure of similarity. In signal processing for example, while a cross-correlation signifies

how related two signals are, the convolution amounts to how much one signal affects another; two sides of the same coin.

Now, in image processing, convolutions are a widely used tool. Convolutional filters can transform images with various purposes ranging from technical to artistic. In figure 3.5 the effect of two convolutional filters is shown. For this, Adobe Photoshop Elements 15 was used. In figure 3.5b contours are filtered out, while figure 3.5c corresponds to a Gaussian spreading.



(A) Unfiltered image of star cluster Westerlund 2 in the Milky Way (public image from NASA/ESA) (B) Image (a) filtered with contour filter (C) Image (a) filtered with Gaussian spread filter

FIGURE 3.5: Effect of application of convolutional filters to images

Now, let us examine how this happens on a mathematical level. Based on equation 3.16, let us consider a two-dimensional convolution between a discrete function  $F : \mathbb{Z}^2 \rightarrow \mathbb{R}$  and a kernel/filter function  $k : [-r, r]^2 \cap \mathbb{Z}^2 \rightarrow \mathbb{R}$  (Yu and Koltun, 2015):

$$(F * k)(\mathbf{x}) := \sum_{\xi} F(\xi)k(\mathbf{x} - \xi) = \sum_{\xi + \tilde{x} = \mathbf{x}} F(\xi)k(\tilde{x}). \quad (3.17)$$

For consistency with equation 3.16 the notation slightly differs from the one in the paper. The convolution operation  $F * k$  above assigns a correlation-like score to a square of pixels. The kernel  $k$  is essentially a probe image that will be superposed on various locations of the actual image to compute the convolution.

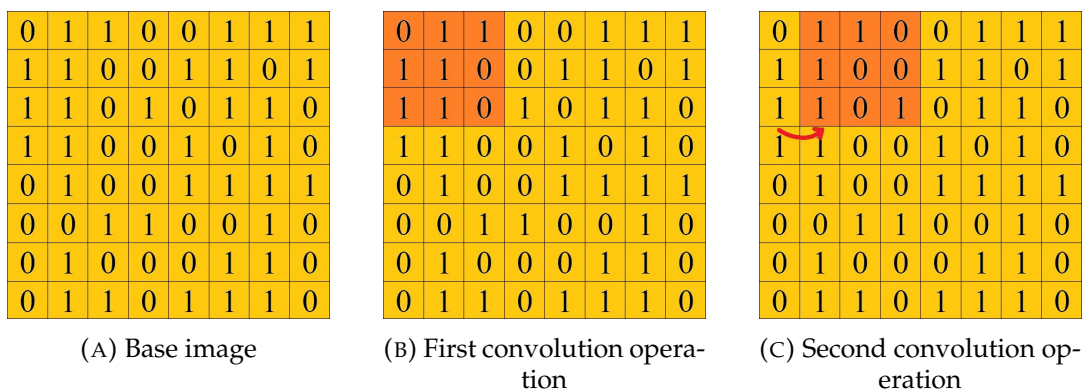


FIGURE 3.6: Successive application of convolutions in a CNN

Figure 3.6 depicts a simplified black-and-white image (binary entries). As we can see, the convolution operation is applied step-wise in segments of the base image. This is done by sliding the kernel to the right, until reaching the end of the image, then taking one step down and starting to slide from left to right again, repeating the

process until the whole image is covered. The result of each convolution is entered into a new two-dimensional array (matrix), called a *feature map*. It shows which segments of the base image coincided the most with the feature encoded in the kernel. Figures 3.5b and 3.5c are feature maps of 3.5a. In the former the kernel encodes the feature of a contour and in the latter the kernel consists of a Gaussian feature.

This operation can be used inside a neural network. The idea goes back to the work of LeCun et al. (1990) on the recognition of handwritten digits. His work was very influential in computer vision and machine learning. In fact, one of the first data sets machine learning practitioners are exposed to is MNIST, which contains 60,000 training and 10,000 testing images of handwritten digits. Instead of using neurons in the first layers of figure 3.2 for instance, convolutions can be used. The learned feature representations in the hidden layers are now given by feature maps instead of the result of a linear transformation. The feature representations in each layer can even be visualized (Zeiler and Fergus, 2014).

This type of network usually performs much better in image recognition tasks, as compared to a Feedforward Neural Network architecture. But the inputs need not be images. In fact, we can see that—as opposed to neuron layers—convolutional filters scan through all segments of an image, and therefore are extremely well suited to find local patterns. In other words, any kind of data that displays some locality information is well suited for CNNs. Of course, in the limit case, as the kernel size approaches the size of the base image, the CNN degenerates into a regular Feedforward Neural Network with fully connected layers, where all information is global. Since the kernel is a square, the kernel size is characterized by the amount of pixels/-data points on each side. A kernel size of 1 would mean the features are so localized, that not even their next neighbors have an influence on the prediction. The kernel size in figure 3.6 is 3.

So the kernel size must be fixed at a reasonable value that depends on the task. Often experimenting with different sizes is the only way forward. There are other CNN-specific hyper-parameters to be fine-tuned. One of them is the stride, which corresponds to the number of steps the kernel needs to slide after each convolution. In figure 3.6 the stride equals 1.

Another hyper-parameter is the amount of padding. Padding refers to the addition of cells around the input data, so that feature-learning in the corners is improved. In figure 3.6 the padding size is zero because no additional cell layer is added. Each cell layer would increase padding by 1.

There is a third property distinctive to CNNs: pooling. Feature maps produced by convolutions tend to become very large, which is why some downsampling mechanism is needed. Pooling is precisely that. Similarly to a convolution it consists of a sliding window producing a new and smaller representation, but the underlying operation can have several types, most famously max pooling and average pooling. In max pooling, each square gets mapped to the maximal entry, while in average pooling an average of all entries is taken. The pooling kernel size can also be fixed. The larger the kernel size, the larger the downsampling effect, which inevitably results in lower resolution. As maintained by Springenberg et al. (2014), as an alternative to pooling, the stride of the convolutional layer can often be increased without loss of accuracy, which makes the network simpler.

Nevertheless, independently of which one of those downsampling mechanisms are used, the fundamental problem of resolution loss remains. This might be no concern for classification tasks, but especially for deep regression it becomes problematic because less resolution translates into less accuracy. A solution to this is offered by (Yu and Koltun, 2015) by means of a dilation of the receptive field. This does not

impair resolution at all, but it does have a downsampling effect. The idea consist of generalizing equation 3.17 to

$$(F * k)_l(\mathbf{x}) := \sum_{\xi + l\bar{x} = \mathbf{x}} F(\xi)k(\bar{x}). \quad (3.18)$$

$(F * k)_l$  is said to be  $l$ -dilated. In figure 3.7 a 2-dilated kernel is depicted.

0	1	1	0	0	1	1	1
1	1	0	0	1	1	0	1
1	1	0	1	0	1	1	0
1	1	0	0	1	0	1	0
0	1	0	0	1	1	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	1	0	1	1	1	0

FIGURE 3.7: Depiction of a 2-dilated kernel

In the present discussion, only two-dimensional convolutional networks were considered. The same notion, however, can be extended to  $N$ -dimensional data with localized information (Mrazova, Pihera, and Velemínska, 2013).

Considering all of the above, the feature map produced by a convolution has the following size (Dumoulin and Visin, 2016):

$$O_d = \lfloor \frac{I_d + 2p - k - (k - 1)(l - 1)}{s} \rfloor + 1, \quad (3.19)$$

where  $I_d$  is the input size in dimension  $d$ ,  $p$  denotes padding,  $k$  is the kernel size,  $l$  the dilation ( $l = 1$  is undilated),  $s$  the stride, and  $O_d$  the output size of the convolution (size of the feature map) in dimension  $d$ .

The output size of a pooling layer would correspond to

$$O_d = \frac{I_d}{k}. \quad (3.20)$$

As previously discussed, a Fully Connected Layer can be regarded as a degeneracy of a Convolutional Layer where  $k = I_d$ ,  $s = 1$  and  $p = 0$ . Using equation 3.19 the output size is, as expected, equal to 1.

A convolutional neural network is usually composed of

- an alternation of Convolution Layers with pooling layers (the output sizes being consistent with the above equations),
- and few Fully Connected Layers.

This architecture balances local features (Convolutions) with global features (Fully Connected Layers). So,  $p$ ,  $k$ ,  $l$  and  $s$ , as well as the pooling size, are some of the hyper-parameters to be fine-tuned. More general hyper-parameters include the learning rate, the batch size and the general architecture in terms of the amount of layers and neurons. The learning rate was already discussed; the batch size indicates how many training examples are sampled per iteration; the architecture on the other hand must be carefully chosen and tested on a case-by-case basis. A larger network usually learns more, but is more computationally expensive. The concepts to be learned from the data include a given number of levels of abstraction. If the number of layers exceeds the levels of abstraction, subsequent layers are redundant and just add to the computational complexity. Also, too many neurons in a layer means redundant representations.

### 3.4.1 Summary of CNN hyper-parameters

In table 3.2, the tunable hyper-parameters, as well the effect of an increment in their size is summarized.

hyper-parameter	effect of incrementing size
convolutional kernel size	focus on higher-level features
padding	improved learning of edge features
dilation	downsampling effect (no resolution loss)
stride	downsampling effect ( $\rightarrow$ resolution loss)
pooling kernel size	downsampling effect ( $\rightarrow$ resolution loss)
batch size	more stable but slower convergence
neurons	more feature representations
learning rate	increment in convergence speed until optimal value, from then on less stable convergence or divergence

TABLE 3.2: Summary of CNN hyper-parameters

The hyper-parameter choice for the task of this Thesis will be explained in Chapter 8. Assessing the optimality of a network can be difficult and a matter of trial and error, since there is still no solid theory behind neural networks. This, however, is changing. There are serious attempts at theoretically clarifying the underlying mechanisms of statistical learning in neural networks, which could in turn provide practical tools to build optimal architectures. Some recent and exciting theoretical research will be thoroughly examined in section 3.5.

## 3.5 Frontiers of theoretical Research on Neural Networks

Neural networks, and in particular Deep Neural Networks, are said to be black boxes. Even though at a neuron level their behaviour is very clear, much like with many-body simulations in physics, it is especially hard to predict their behaviour once the system becomes increasingly complex. There have been attempts at exposing the internal workings of such a network and also to examine an already trained network.

We will first discuss some methods to gain insight into and visualize learned patterns in a Convolutional Neural Network that has already been trained. One approach is to use a Deconvolutional Neural Network that reconstructs and visualizes the representations in previous layers (Zeiler and Fergus, 2014, as presented



by Arxiv-Insights, 2017). While this approach is definitely useful in understanding the learned features and provides a tool to assess the performance contribution of a given layer, it also needs specific input images to produce a visualization, making it difficult to recognize more general patterns that go beyond that particular image.

As proposed by Arxiv-Insights (2018), a solution to this could be to feed the network with randomly initialized images and update its pixels using gradient descent to maximally excite a given neuron. This approach generates images that describe the learned patterns in a more general manner. In fact, so-called Generative Adversarial Networks make use of this method to modify images (adversarial images) to trick a CNN into misclassification (Goodfellow et al., 2014), which poses an important security concern.

The discussed methods already provide some insight, but designing a neural architecture is still a matter of trial and error, since there is no solid theoretical foundation on the superiority of one model over another. Schwartz-Ziv and Tishby (2017) have tried to shed some light onto the issue by making use of information theory. The authors describe the learning process as being limited by an information bottleneck, providing a theoretical limit onto the performance of a network, a ground-breaking achievement in the context of a field where uncertainty in the network optimality has been prevalent. This description offers a novel and much more complete picture of the learning mechanisms on a larger scale and will help to understand the model design, which is why the author of the present work decided to dive deeper into the topic, starting from probabilistic and statistical models, clarifying the mathematics behind information theory and step by step building the groundwork for this theory.

### 3.5.1 Probabilistic and statistical models

The modern and most mathematically rigorous way to treat probabilistic and statistical models would be in the context of measure theory, which is especially useful when dealing with continuous data, but independently of the mathematical paradigm under which it is seen, it is essentially an extension of formal logic that describes the amount of uncertainty in a system. The Algebra describing a probability space is given by a triple  $(\Omega, E, P)$ ,  $\Omega$  being the sample space corresponding to the set of possible outcomes,  $E$  being a set of subsets of  $\Omega$  and thus forming the set of events, and  $P$  (probability measure function) being a map from an event  $e \in E$  to  $p(e) \in [0, 1]$ . Of course  $p(e) = 0$  signifies the impossibility of  $e$ , whereas  $p(e) = 1$  signifies absolute certainty.

If the goal is to describe a complex system, several variables and parameters have to be taken into account. If we are talking about Deep Neural Networks with several layers of depth, it is helpful to examine the probabilistic treatment of a general graph that is meant to describe the system.

Let  $G = G(V, E)$  be a graph. It is composed of  $V$  nodes or vertices specifying a current state and  $E$  edges specifying the transition between nodes. This can be seen in figure 3.8.

The graph can be:

- directed: a direction of flow between two nodes is specified. This is the case in neural networks. Here, information can only flow forward. Backward propagation is only to update the network and could be seen as a separate graph.
- undirected: direction of flow is irrelevant.

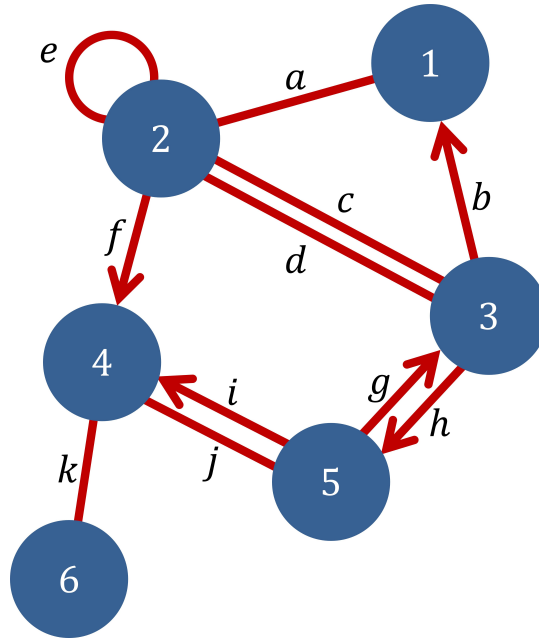


FIGURE 3.8: "Graph example" by Römert, located at [https://commons.wikimedia.org/wiki/File:Graph\\_example\\_\(Graph\\_theory\).png](https://commons.wikimedia.org/wiki/File:Graph_example_(Graph_theory).png), Creative Commons Attribution-Share Alike 3.0 Unported License

Nodes could technically also point to themselves, although in the case of neural networks this will not be important. Now, to describe our system probabilistically we could assign each node a probability. Here, the concept of conditional probabilities comes in, which is in fact at the core of statistical inference. We will discuss this in much more detail.

In figure 3.9 we can observe that  $y$  depends on  $x$ . The corresponding probabilities would be  $p(x)$  for the first node and  $p(y|x)$  for the second node. Note that  $p(y|x) \neq p(y)$  since the outcome of  $y$  depends on the outcome of  $x$ .

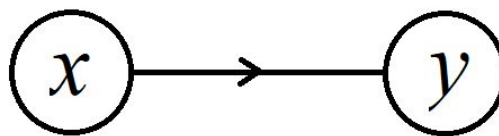


FIGURE 3.9: Directed graph from  $x$  to  $y$

In figure 3.10 the probabilities are inverted. The first node has probability  $p(x|y)$  and the second one  $p(y)$ . Again,  $p(x|y) \neq p(x)$  because  $x$  depends on  $y$ .

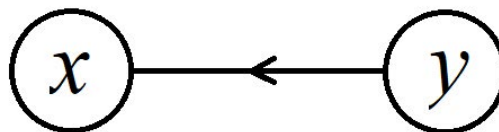


FIGURE 3.10: Directed graph from  $y$  to  $x$

In figure 3.11 there is no dependence relationship between  $x$  and  $y$ . The corresponding probabilities are  $p(x)$  and  $p(y)$ , where  $p(x) = p(x|y)$  and  $p(y) = p(y|x)$ .

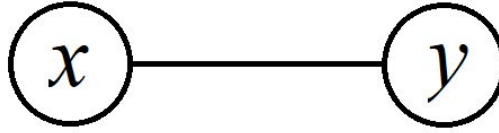


FIGURE 3.11: Undirected graph between x and y

JOINT PROBABILITY:

A joint probability is a probability of two or more events happening simultaneously. It can be computed by multiplying the individual node probabilities of a graph. The above figures result in the following joint probabilities:

$$\text{Figure 3.9: } p(x, y) = p(x)p(y|x) \quad (3.21)$$

$$\text{Figure 3.10: } p(x, y) = p(x|y)p(y) \quad (3.22)$$

$$\text{Figure 3.11: } p(x, y) = p(x)p(y) \quad (3.23)$$

In general a joint probability of variables  $x_1, x_2, \dots, x_n$  is given by

$$p(x_1, x_2, \dots, x_n) = p\left(\bigcap_i x_i\right) = p(x_1) \prod_{i=2}^n p(x_i | x_1, \dots, x_{i-1}) \quad (3.24)$$

This is equivalent to a logical *AND* operation. An *OR* operation between probabilities is given by

$$p\left(\bigcup_i x_i\right) = \sum_{i=1}^n p(x_i) - \sum_{i=1}^n \sum_{j \geq i} p(x_i, x_j) \quad (3.25)$$

For the sake of completeness, we will also define the marginal probability. It is computed based on the joint probability. In essence it sums over the probabilities of all possible outcomes of the nodes it depends on, resulting in an unconditional probability. Mathematically this means:

$$p(x_i) = \sum_{x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n} p(x_1, x_2, \dots, x_n) \quad (3.26)$$

BAYES' THEOREM

The concept of conditional probabilities is tightly linked with the concept of statistical inference which is based on Bayes' Theorem. The derivation follows now:

$$p(x|y) = \frac{p(x, y)}{p(y)} \iff p(x, y) = p(x|y)p(y) \quad (3.27)$$

$$p(y|x) = \frac{p(y, x)}{p(x)} \iff p(y, x) = p(y|x)p(x) \quad (3.28)$$

$$p(x, y) = p(y, x) \iff p(x|y)p(y) = p(y|x)p(x) \quad (3.29)$$

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} \quad (3.30)$$

Equation 3.30 is Bayes' Theorem and basically states how a conditional probability can be obtained using the conditional probability in inverse order, which is often easier to determine. For example, it is easier to determine how likely it is that it is



cloudy, given that it is raining ( $p(\text{cloudy}) = 1$ ) than to determine how likely it is that it is raining, given that it is cloudy ( $p(\text{rain}) \leq 1$ ).

### 3.5.2 Statistical inference

Now, while probability theory provides a strong theoretical background to reason about uncertainty, statistics is the direct connection between measurement and probability theory, and Bayes' Theorem gives us a tool to statistically infer new information. Machine Learning can be regarded as a sophisticated statistical inference tool that is able to learn highly non-linear information. In practical applications statistical estimators, which can be more or less robust, are of crucial importance, which is why we are going to examine them now. Robustness is the insensitivity with respect to gross errors in the data. Outliers should not affect the predictions too much. This is not only a concern in the choice of estimators, but also in the design of a neural network itself, as will be discussed later. If a network is too sensitive to changes in the input data, it is not able to generalize; it essentially just memorizes data, which is of no use for pattern recognition.

As for predictors, let  $A$  be an estimator of the parameter  $\alpha$ . There are a few properties that make for a good estimator, the most important ones being:

- **Unbiasedness:** To the extent possible the expectation should coincide with the true value, i.e.  $E[A] = \alpha$ .
- **Consistency:** extension of sample size  $n$  should decrease dispersion, i.e.  $\lim_{n \rightarrow \infty} \text{dispersion}[A] = 0$ . Measures of dispersion are discussed below.
- **Efficiency:** unbiased estimator  $A$  with minimum possible dispersion compared to alternatives, i.e.  $\text{dispersion}[A] < \text{dispersion}[A_{\text{alternative}}]$

Those are just base criteria for ensuring truthfulness and minimum dispersion. Outliers in the data affect predictions; however, as previously mentioned, robust estimators minimize this inconvenience. Robust statistics is a whole field in mathematics that studies their quality, and without going into too much depth in this topic, some of the classical examples of measures of central tendency and dispersion and more robust versions are presented below.

For Euclidean spaces, some examples for measures of central tendency are:

- mean:  $\frac{1}{n} \sum_{i=1}^n x_i$
- median: middle value of sequence of  $x_i$  values
- mode: the most frequent number in the sequence of  $x_i$  values
- Hodges-Lehmann estimator: median of means of subsets of sample space (Hodges Jr and Lehmann, 1963)

As for measures of dispersion or deviation, the most common examples are:

- variance:  $\sigma^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x}_i)^2$
- standard deviation:  $\sqrt{\sigma^2} = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x}_i)^2}$

- MAD (median absolute deviation):  $\text{median} |x_i - \bar{x}_i|$
- $S_n = 1.1926 \text{med}, \text{med}_j |x_i - x_j|$  and .25 quantile of the distances  $Q_n = |x_i - x_j|; i < j$  (Rousseeuw and Croux, 1993)

The question of what measures of central tendency and deviation are going to be most useful totally depends on the data and the skewedness of the underlying probability distribution. In general the mean and variance/standard deviation are not considered to be robust, while the most robust measures previously presented are the Hodges-Lehman estimator and  $S_n/Q_n$ . The latter even significantly improves on the Gaussian efficiency of  $MAD$  and has a computational time complexity of  $O(n \log n)$  and storage complexity of  $O(n)$ .

Although in some contexts it is absolutely necessary and even advisable to take into account robustness, for the sake of this work it is not. The reason is due to the nature of neural networks. It is a highly distributed graph solving linear problems at each node with non-linearities in between. At each node a Gaussian distribution is assumed. One of the biggest strengths of machine learning lies precisely in the fact of being able to solve complex, high-dimensional and non-convex problems by composing many very simple linear problems with subsequent non-linear activations in a highly distributed manner. For this reason, the mean and variance will suffice as statistical measures of central tendency and dispersion for the purposes of this work. The variance of the predicted value is called MSE loss (mean-squared error loss) in the context of neural networks.

The fact that each node in a neural network assumes Gaussian distributions is due to the continuity of data. A Gaussian distribution contains minimal information about the system, or in other words, its entropy in an information-theoretical sense is maximal. To prove this and to proceed to examining the neural networks in more detail, let us introduce some information theory.

### 3.5.3 Information Theory

The first quantity we will introduce is the entropy. Entropy has originally been defined in a discrete statistical physics context by Boltzmann (Boltzmann, 1877). This concept was later generalized to the continuous domain by Shannon (Shannon, 1948) in the frame of reference of telecommunications. It was important at the time to examine the information content in electrical signals. The entropy, as in Physics, is a measure of "disorder" or uncertainty. It can also be regarded as the expectation value of information.

Let  $p(x)$  be a probability distribution of  $x$ . Information can be defined as

$$I(x) = \ln \frac{1}{p(x)} \quad (3.31)$$

and as a result the entropy can be written as

$$h(p) = E[I(x)] = \int p(x) \ln \frac{1}{p(x)} dx \quad (3.32)$$

As a side note, in telecommunications  $\ln$  is replaced by a base-2 logarithm.

*Gaussian as a maximum entropy distribution*

The reason Gaussians are often chosen is because in nature and also many artificial systems entropy is maximized and Gaussians are maximum entropy distributions for continuous real-valued data for a given mean  $\mu$  and variance  $\sigma^2$ . Maximizing entropy means maximizing the function

$$h(p) = \int p(x) \ln \frac{1}{p(x)} dx. \quad (3.33)$$

This can be done using Lagrange multipliers with the constraints

$$\mu = E[x] \iff \mu = \int_{-\infty}^{\infty} xp(x)dx \quad (3.34)$$

$$\sigma^2 = E[(x - E[x])^2] = E[x^2] - E[x]^2 \iff \sigma^2 = \int_{-\infty}^{\infty} x^2 p(x)dx - \mu^2 \quad (3.35)$$

In equation 3.35 Steiner's Theorem was used. This yields exactly the Gaussian distribution, meaning it contains the least amount of prior knowledge.

The entropy is a measure that allows to assess and compare the amount of uncertainty between distributions. A higher entropy signifies a higher uncertainty. There are metrics specifically designed for such comparison. One of the most commonly used ones is the Kullback-Leibler divergence, or also relative entropy (Kullback and Leibler, 1951), as shown in equation 3.36.

$$D(p(x), q(x)) = \int p(x) \ln \frac{p(x)}{q(x)} dx \quad (3.36)$$

This is important because often times it is necessary to know how different distributions are. For instance, it was used by Frühwirth (2003) to find an optimal Gaussian mixture that is as close as possible to a probability distribution that describes the electron energy loss more precisely (Bethe-Heitler model). There are other measures of distribution divergence, like the ones proposed by Lin (1991), but the Kullback-Leibler divergence is one of the most commonly used ones. This concept can also be used to compare joint distributions with their marginals, as is the case with mutual information:

$$D(p(x,y), p(x)p(y)) = \int p(x,y) \ln \frac{p(x,y)}{p(x)p(y)} dx \quad (3.37)$$

As previously discussed, in the case of the nodes or variables being independent of one another,  $p(x,y)$  should coincide with  $p(x)p(y)$ . However, this is not always the case. As we could imagine, the higher the value of the relative entropy in 3.37, the more the entropy increased by passing from one node to another. In other words, a high relative entropy causes a more rapid decrease in information.

If we go back to the idea of neural networks as probabilistic graph models, we can easily see how information can get lost by passing from one node to another. In fact, just like Schwartz-Ziv and Tishby (Tishby and Zaslavsky, 2015) realized in their work, regarding a neural network as a Markov chain  $X \rightarrow Y \rightarrow Z$ , information can only decrease:

$$I(X;Y) \geq I(X;Z) \quad (3.38)$$

The goal however is to minimize this information loss. In fact, by training a neural network, we are able to increase the information in subsequent layers. There is a theoretical maximum to that process, indicated by the information bottleneck (Tishby, Pereira, and Bialek, 1999), but information still "travels" from one layer to the next. That gives us a much more complete picture of how neural networks learn information. But the most interesting part is the mechanism by which this happens.

This can be examined by looking at the gradient flow when training a network using stochastic gradient descent. Shwartz-Ziv and Tishby (2017) observed two distinct gradient signal-to-noise ratios. They compared the mean and standard deviation of the gradients as the number of epochs increase. As can be expected, at first the SNR is high. Of course, it decreases over time as the weights converge to some configuration. However, even when the noise surpasses the signal, information transfer continues. This is a novel paradigm and exciting discovery by the mentioned authors, since it gives a theoretical explanation of the dynamics of deep learning. The way this was interpreted by them is that learning can be separated into two phases:

- **drift phase:** Here empirical risk minimization (ERM), or just usual fitting, happens. Lots of information about features of previous layers is somehow "memorized".
- **diffusion phase:** This is a stochastic relaxation (in particular Wiener) process where the network "forgets" irrelevant information. Here irrelevant dimensions of high-dimensional data are compressed away. Much lower-dimensional representations remain.

Although it is already known that compression can contribute to learning as illustrated by Floyd and Warmuth (Floyd and Warmuth, 1995), the existence of these two phases is new.

What is especially exciting about this work is that it provides tangible tools to assess the optimality of the network, something that is often not even considered because of the general point of view of neural networks as black boxes. Schwartz-Ziv and Tishby argue that the accuracy and sample complexity of a neural network is predictable using just two mutual informations. In such a network inputs become encoded into some representation and finally decoded. The mutual informations of encoder and decoder are apparently good indicators of accuracy and sample complexity, as long as enough data is available and the network is large enough. This is the first time, to the author's knowledge that such a prediction is possible. Also, Schwartz-Ziv and Tishby provided the code they used on github, so that their findings can be reproduced and adapted to future applications. Their source code is available at <https://github.com/ravidziv/IDNNs> (Shwartz-Ziv and Tishby, 2017a). In the present Thesis, an attempt was made to adapt the code to the task at hand, and while it was certainly possible, the computation time was too high, making it impossible to compute information planes in the available time. However, part of the reason might be the use of docker containers running a Linux OS on top of a host Windows OS. Docker does not count with a GPU throughput for Windows at the time of writing of the present document. Training Deep Learning models with CPUs significantly decreases computational speed. The enabling of GPU resources would considerably accelerate the learning process. In the future the provided code could be tested directly on a Linux system, or in a Docker container once the company introduces a GPU throughput for Windows systems. Another reason could be the size of the network. This could be solved by using more sophisticated visual

representations using for example Gaussian mixture models on the layers. What matters is that information transfer can be properly visualized. In that way an optimal network with minimal resources could be designed. For time constraints, and also given the remarkable performance of the model presented in the present Thesis, such undertaking was not necessary. It could however be considered for future improvements or use in other machine learning tasks at CERN and HEPHY. As a side note, there is a follow-up paper based on the information bottleneck theory (Saxe et al., 2018) where *ReLU* activations (Rectified Linear Units) were used instead of *tanh*, basically fusing the two phases into one. The use of *ReLUs* is usually encouraged to improve sparsity.

### 3.5.4 Topology and data analysis

At first sight one might pose the question of how such an abstract discipline like topology can give insight into real-world data. The answer is that data, unless it is randomly distributed, in which case it would be useless anyway, has an inherent structure. It is composed of various features that characterize it and that display relationships between one another. The usefulness of topology lies precisely in capturing the spatial relationship between its elements.

For further analysis let us formally define a topology and topological space. It is in terms of these notions that the data analysis tools presented thereafter will be explained.

Let  $X$  be a set and  $\tau \subseteq P(X)$ , where  $P(X)$  is the power set of  $X$ .  $\tau$  is called a topology on  $X$  if the following conditions hold true:

- $\emptyset, X \in \tau$
- $\forall t_i \in \tau : \bigcup_i t_i \in \tau$
- $\forall t_i \in \tau : \bigcap_{i=1}^n t_i \in \tau$

The motivation for this definition is the fact that the elements of a topology are (by definition) open sets and thanks to this notion of open sets concepts like continuity and closeness between points can be rigorously defined in the most abstract way without introducing unnecessary structure to the mathematical object. It is in that way that point similarity is well-defined in a topological space despite the lack of a distance metric.

Topological data analysis is a discipline aimed at capturing topological features and visualizing them in a geometric fashion in two or three dimensions. The cardinality of the feature space corresponds to the dimensionality of the problem. The feature-richness of data is completely dependent on its type. Experimental data from Physics or Chemistry, for example, is usually much more comprehensive than, for instance, data from psychological studies. That is because in the former disciplines it is much easier to design experiments in a controlled environment and to reduce the amount of variables, i.e. the dimensionality, significantly. The author clarifies that those are just examples and not meant to judge the relevance of the disciplines, but to clarify the variation in complexity in the process of feature selection. In fact one of the most important problems to be solved when applying machine learning algorithms is not only designing a suitable network architecture, but selecting the minimal set of independent and information-rich features that describe the problem

in a way that is as complete as possible. Having some sort of structural understanding of the data before applying a neural network can be of great help. We will discuss tools that aim at doing exactly that.

### 3.5.5 The curse of dimensionality and dimensionality reduction

It is self-evident that data with higher dimensionality will result in a need for more training examples. As the dimensionality increases, the volume of the space it spans grows exponentially, decreasing the density of data points. Less density means less meaningful data to learn from. If the method of analysis depends on some geometric information, for instance if it makes use of Euclidean distances, the volume of the feature space becomes relevant, and the amount of examples needed becomes exponential to dimensionality. In the case of neural networks we have already determined the sample complexity (equation ??). It is not exponential, but linear or even less, but dimensionality is still a problem. Reducing it would help, in case the problem becomes large enough. But moreover, it also helps in getting insight into the data. For that, not even target data for training is needed. Dimensionality reduction techniques and topological data analysis tools allow to embed high-dimensional input data into two or three dimensions to gain insight into its structure. They represent similar groups of data points with point nodes and connect the nodes with edges, according to the relationship they have to one another. This could also be very helpful in the feature selection process, although for the present task features were already provided. In order to apply dimensionality reduction, different techniques like stochastic neighbor embeddings are used. In fact, the author applied a two-dimensional t-distributed Stochastic Neighbor Embedding (t-SNE) (Maaten and Hinton, 2008) to the particle track data provided by CMS, which is one of the most powerful and recently developed state-of-the-art techniques for dimensionality reduction. The result will be shown in Chapter 5. There are other topological data analysis tools like Principal Component Analysis (PCA), autoencoders or multidimensional scaling (MDS), among others, but t-SNE was very effective for visualizing the particle track data.

#### t-distributed Stochastic Neighbor Embedding (t-SNE)

Let us briefly introduce Student's t-distribution as t-SNE makes use of it. As previously discussed, the Gaussian normal distribution is the maximum entropy distribution for random variables  $x \in \mathbb{R}$  with finite mean  $\mu$  and variance  $\sigma$ . However,  $\sigma$  is the population standard deviation, while the sample standard deviation is often denoted by  $s$ . Since  $\sigma \neq s$  in general, the resulting distribution—called t-distribution—is going to be slightly different from a Gaussian, having heavier tails, although it converges to it for increasing degrees of freedom/sample size. Degrees of freedom  $\nu$  and sample size  $n$  have the following relationship

$$\nu = n - 1 \tag{3.39}$$

and the t-distribution is of the following form:



$$p(x) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\Gamma(\frac{\nu}{2})} \left(1 + \frac{t^2}{\nu}\right)^{-\frac{\nu+1}{2}}. \quad (3.40)$$

Such distributions with varying degrees of freedom are depicted in 3.12.

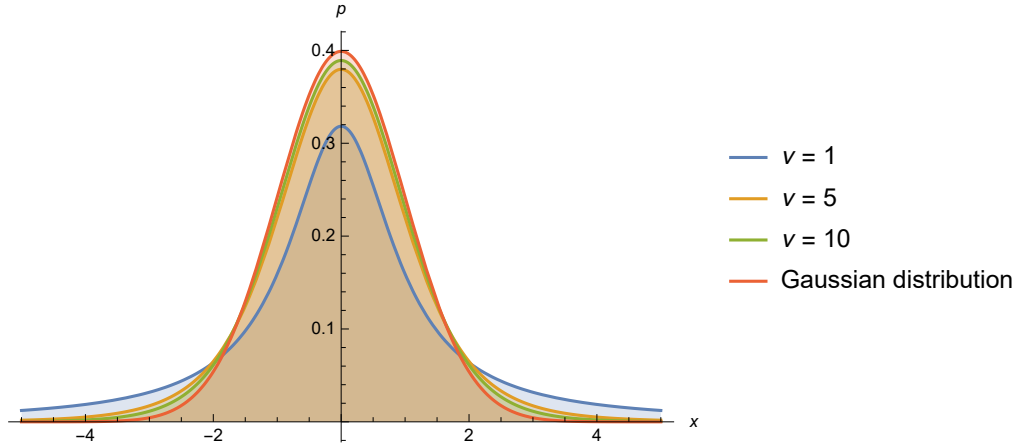


FIGURE 3.12: Comparison of Student's t-distribution with degrees of freedom  $\nu = 1, 5$  and  $10$  with a Gaussian distribution

The reason t-distributions are important in practice is that they very well describe the behaviour of random variables that are normally distributed but whose population variance is unknown, which is often the case. We will later see how this will affect the way t-SNE works.

t-SNE (Maaten and Hinton, 2008) is an improved version of stochastic neighbor embeddings or SNE for short (Hinton and Roweis, 2003). Let  $x_i$  be the points in the high-dimensional space (dimension  $d$ ) and  $y_i$  be the points in the low-dimensional embedding (dimension 2 or 3). For the embedding, two probability distributions are needed, one for each space.

$$d - \text{dimensional space distribution} : p_{j|i} = \frac{\exp(-\frac{\|x_i - x_j\|^2}{2\sigma_i^2})}{\sum_{k \neq i} \exp(-\frac{\|x_k - x_i\|^2}{2\sigma_i^2})} \quad (3.41)$$

$$\text{distribution in embedding} : q_{j|i} = \frac{\exp(-\|y_i - y_j\|^2)}{\sum_{k \neq i} \exp(-\|y_k - y_i\|^2)} \quad (3.42)$$

They represent the probability of a point of index  $j$  to be a neighbour of  $i$ . Small distances are favoured and as distance increases the probabilities rapidly decrease. For the embedding to be a truthful representation  $p_{j|i}$  must be equal to  $q_{j|i}$ . In the section about information theory we already introduced a quantity that captures the difference between distributions, i.e. the relative entropy or Kullback-Leibler divergence (equation 3.36). In the discrete case it is given by

$$D(p_{j|i} || q_{j|i}) = \sum_j p_{j|i} \ln \frac{p_{j|i}}{q_{j|i}}. \quad (3.43)$$

Bringing the distributions closer to each other is equivalent to minimizing the Kullback-Leibler divergence. The SNE algorithm uses gradient descent to minimize the following cost or loss function that is composed of the sum of all KL-divergences:

$$C = \sum_i \sum_j p_{j|i} \ln \frac{p_{j|i}}{q_{j|i}} \quad (3.44)$$

One problem with SNE is its difficulty in minimizing  $C$ . Another one is called crowding problem and consists in the fact that, as the Kullback-Leibler divergence is asymmetric, a mapping into nearby points is less penalized than a mapping into distant points, effectively squeezing points together. The solution that t-SNE offers is to symmetrize the probability distributions and use t-distributions instead of Gaussians for the low-dimensional embedding. t-distributions are not only computationally less expensive, but they have heavier tails, which helps in counteracting excessive attractive forces leading to crowding.

The symmetrized and modified probability distributions used for t-SNE are specified below:

$$p_{ij} = \frac{\exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2})}{\sum_{k \neq l} \exp(-\frac{\|x_k - x_l\|^2}{2\sigma^2})} \quad (3.45)$$

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|y_k - y_l\|^2)^{-1}} \quad (3.46)$$

A hyperparameter of t-SNE is the perplexity. According to the mentioned paper it is defined as

$$\text{perplexity}(P_i) = 2^{H(P_i)} \quad (3.47)$$

where  $H(P_i)$  is the entropy of  $P_i$ . It essentially controls how many neighbors are chosen and therefore how much local properties are being preserved. Low perplexity preserves local effects more strongly, while high perplexity focuses on global shape. Although t-SNE is quite robust to changes in perplexity, Wattenberg, Viégas, and Johnson (2016) highlight examples where checking various perplexities makes a difference. They also stress some issues with preservation of distances and cases of random data being embedded into incorrect clusters, while still emphasizing its power and flexibility. The method will be applied in Chapter 5 to get insight into the particle track data and check whether its behaviour actually corresponds to the expectations. Some other qualitative and quantitative tests will also be performed in this chapter.



## Chapter 4

# Track parameter estimation by Kalman and Gaussian-sum filter

Parameter estimation lies at the heart of statistical inference and many fields in science and engineering that rely on it. Two very powerful estimation tools that are widely used and combine a priori information with measurement data will be now presented.

### 4.1 Kalman filters

Kalman filters are used for parameter estimations in dynamical systems. They have been extensively researched and applied to many fields in science and engineering, since their discovery (Kalman, 1960). Prominent applications can be found in control theory and navigation, where position estimation is crucial. They were successfully used for this purpose in NASA's Apollo Mission (McGee and Schmidt, 1985).

The application that is relevant to the present work is track parameter estimation, so control terms are not considered. Most of the mathematics subsequently presented originate from Frühwirth's paper about the application of Kalman filtering to track and vertex fitting (1987). There are two quantities that are meant to describe the state of a dynamical system and are updated as it evolves in time. Those are

- the state vector  $\mathbf{x}_k$  at time  $k$ ,
- and the measurement vector  $\mathbf{m}_k$  at time  $k$ .

The Kalman filter demands the following assumptions about  $\mathbf{x}_k$  and  $\mathbf{m}_k$ . They must be

- independent, i.e.  $p(\mathbf{x}_k | \mathbf{m}_k) = p(\mathbf{x}_k)$  and  $p(\mathbf{m}_k | \mathbf{x}_k) = p(\mathbf{m}_k)$ , and
- normally distributed, i.e.  $\mathbf{x}_k \sim \mathcal{N}(\mu_x, \sigma_x)$  and  $\mathbf{m}_k \sim \mathcal{N}(\mu_m, \sigma_m)$ .

The equations describing the evolution of the state with each time step are given by

$$\mathbf{x}_k = \mathbf{F}_{k-1}\mathbf{x}_{k-1} + \mathbf{w}_{k-1} \quad (4.1)$$

$$\mathbf{m}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{e}_k \quad (4.2)$$

where  $\mathbf{F}_k$  is the process transition matrix,  $\mathbf{w}_k$  the process noise,  $\mathbf{H}_k$  the measurement transition matrix and  $\mathbf{e}_k$  the measurement noise and

$$\mathbf{w}_k \sim \mathcal{N}(0, \mathbf{Q}_k) \quad (4.3)$$

$$\mathbf{e}_k \sim \mathcal{N}(0, \mathbf{V}_k) = \mathcal{N}(0, \mathbf{G}_k^{-1}) \quad (4.4)$$

The process noise  $\mathbf{w}_k$  in the track reconstruction process is predominantly due to multiple Coulomb scattering at the nuclei. As explained in the mentioned paper, there are three possible operations in the Kalman filtering process:

- Filter: uses past states to estimate the present state
- Prediction: estimation of a future state vector
- Smoothing: estimation of a past state vector using measurements until the present time

The same notation from the paper is adopted as follows:

- $x_k$  ... state vector at time  $k$
- $\tilde{x}_k^i$  ... estimated state vector at time  $k$  using measurements up to time  $i$
- $r_k$  ... residual at time  $k$
- $\tilde{r}_k^i = \mathbf{m}_k - \mathbf{H}_k \tilde{x}_k^i$  ... estimated residual at time  $k$
- $\mathbf{C}_k^i = \text{Cov}(\tilde{x}_k^i - x_k)$  ... state covariance
- $\tilde{\mathbf{R}}_k^i = \text{Cov}(\tilde{r}_k^i)$  ... residual covariance

Since the covariance matrices were not used for the present work, only the transformations of the state vector and the residuals are shown. For this we need the Kalman gain matrix which is given by

$$\mathbf{K}_k = \mathbf{C}_k \mathbf{H}_k^T \mathbf{G}_k \quad (4.5)$$

and the smoother gain matrix which is given by

$$\mathbf{A}_k = \mathbf{C}_k \mathbf{F}_k^T (\mathbf{C}_{k+1}^k)^{-1} \quad (4.6)$$

### Updates

#### Prediction:

$$\begin{aligned} \tilde{x}_k^{k-1} &= \mathbf{F}_{k-1} x_{k-1} \\ \tilde{r}_k^{k-1} &= \mathbf{m}_k - \mathbf{H}_k \tilde{x}_k^{k-1} \end{aligned}$$

#### Filtering:

$$\begin{aligned} x_k &= \tilde{x}_k^{k-1} + \mathbf{K}_k (\mathbf{m}_k - \mathbf{H}_k \tilde{x}_k^{k-1}) \\ r_k &= \mathbf{m}_k - \mathbf{H}_k x_k = (\mathbf{I} - \mathbf{H}_k \mathbf{K}_k) \tilde{r}_k^{k-1} \end{aligned}$$

#### Smoothing:

$$\begin{aligned} \tilde{x}_k^n &= x_k + \mathbf{A}_k (x_{k+1}^n - x_{k+1}^k) \\ \tilde{r}_k^n &= r_k - \mathbf{H}_k (x_k^n - x_k) = \mathbf{m}_k - \mathbf{H}_k \tilde{x}_k^n \end{aligned}$$

## 4.2 Gaussian-sum filters

Not always can a distribution be approximated by a single Gaussian. The Bethe-Heitler distribution for instance, which describes electron energy loss through bremsstrahlung, is much better approximated by a mixture of Gaussians. That is the purpose of a Gaussian-sum filter. It is a non-linear generalization of a Kalman filter, running several of them in parallel (Adam et al., 2005b). This is useful for electron track reconstructions because electron energy loss is highly non-Gaussian (Frühwirth and Frühwirth-Schnatter, 1998).

### 4.2.1 Optimal Gaussian mixtures

In Chapter 3 we already introduced a measure of distance between probability distributions: the Kullback-Leibler divergence. There is another such measure based on the distance between the cumulative distribution functions (CDF) of both distributions. Those distance measures are shown below and were used by Adam et al. (2005) to minimize the difference between the Bethe-Heitler distribution and a Gaussian mixture.

$$D_{CDF} = \int_{-\infty}^{\infty} |F(z) - G(z)| dz \quad (4.7)$$

$$D_{KL} = \int_{-\infty}^{\infty} \ln[f(z)/g(z)] f(z) dz \quad (4.8)$$

### 4.2.2 Number of components

As described by Adam et al. (2005) the energy loss approximation using Gaussian mixtures corresponds to a convolution between the mixture and a current state which is also composed of some Gaussians. This happens in every layer of material, leading to an excessive growth in the number of components. Therefore this number has to be limited to a maximum  $N$ .

In the mentioned paper, this was done in two ways:

- by choosing the  $N$  components with the largest weights
- by merging components using a distance metric and attracting clusters of Gaussians until the number of components equals  $N$

Although the second method was computationally more expensive, the results exceeded the ones of the first method in Adam et al.'s work. The authors found that Gaussian-sum filters improved precision in the electron reconstruction as compared to individual Kalman filters.

## Chapter 5

# CMS Data

### 5.1 Structure of the data

To understand the structure of the data that is consequently analysed, some background will be given on the work of Frühwirth (2003) which is taken as a basis for this one. A commonly used model for describing electron energy loss through bremsstrahlung is the Bethe-Heitler model (Bethe and Heitler, 1934). It is based on a first order Born approximation of the interaction between an electron and a nucleus. This can be understood as either an electron's absorption of a virtual photon originating from the nucleus and a subsequent emission of a photon, or the same process in time-reversed order. Both cases are depicted in figure 5.1.



FIGURE 5.1: Feynman diagrams of bremsstrahlung process in first order Born approximation

Using Quantum Field Theory the cross-section and other relevant quantities can be determined. What is relevant for this work is the energy distribution. Let  $z$  be the quotient of the post-bremsstrahlung electron energy over original energy. The probability density function of  $z$  is given by (Frühwirth, 2003)

$$f(z) = \frac{[-\ln z]^{c-1}}{\Gamma(c)} \text{ with } c = \frac{t}{\ln 2}, z \in (0, 1). \quad (5.1)$$

This relationship is depicted in figure 5.2. Equation 5.1 shows a strongly non-Gaussian behaviour. Since Kalman filters are based on the assumption of normally distributed inputs, they cannot account for bremsstrahlung on their own. Nevertheless, Gaussian-sum filters that allow for Gaussian mixture approximations, can in fact approximate the effect. The Gaussian mixture is obtained by minimizing the Kullback-Leibler-divergence between  $f(z)$  and the mixture  $g(z)$ . Such a mixture is called a KL-mixture. Frühwirth's model was then implemented in a Gaussian-sum filter for

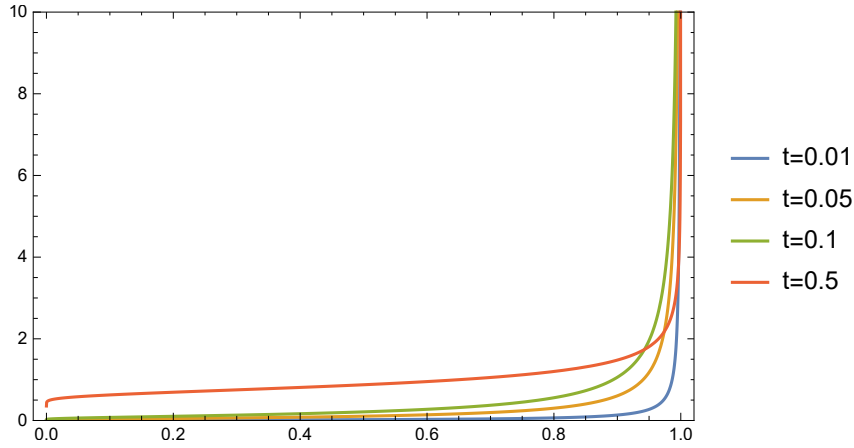


FIGURE 5.2: Fractional energy loss distribution for several  $t$ -values

electron reconstruction in the CMS tracker (Adam et al., 2005a).

The CMS Tracker is essentially a cylinder around a beam line axis. Particles move in it in a helix trajectory and they have distinct track parameters forming the five-dimensional *state vector*  $x$ . As specified in Bernkopf's Master's Thesis (under revision), the parameters are:

- Charge over momentum  $\frac{q}{p}$  [GeV]: momentum is proportional to the helix projection curvature radius,
- Azimuthal angle  $\phi$  [rad]: angle enclosing x-axis and tangent of projected helix,
- Polar angle  $\theta$  [rad]: angle enclosing the beam line and the tangent to the track,
- Transverse impact parameter  $d_0$ ,
- Longitudinal impact parameter  $d_z$ .

The original data set used for this work is contained in a CSV file that is split into various parts, namely *ex1\_split\_data\_\*.csv* and *ex2\_split\_data\_\*.csv*. They contain 2,270,694 simulated tracks, plus some truncated ones that could not be used. Out of them, some had to be cut out because they caused difficulties to reduce the objective function. In the end 2,062,223 tracks with their respective Gaussian mixture data (weights, means and covariances) and target values remained. Covariances however were not used. Because the tracks are simulated, the target data are the true values that are to be predicted.

The unparsed information is listed in a single column. Since the original file contained truncated tracks that are of no use for this work, those were cut off. In a second step, all tracks were read into a multidimensional array in MATLAB using *read\_data.m* and internally merged. Finally, the input and target data for the deep learning model was generated using *make\_input.m* and the multidimensional array saved as *mydata.mat* with float64 precision. The parsing scripts are the same as the ones used by Bernkopf.

The resulting data has the following structure:

Covariance data was not used for this work. The above figures depict the complete input and target data used to train the network. The 72 input data entries in figure 5.3 correspond to the 6 target data entries in figure 5.4.

Gaussian weight 1	parameter 1	parameter 2	parameter 3	parameter 4	parameter 5
Gaussian weight 2	parameter 1	parameter 2	parameter 3	parameter 4	parameter 5
Gaussian weight 3	parameter 1	parameter 2	parameter 3	parameter 4	parameter 5
Gaussian weight 4	parameter 1	parameter 2	parameter 3	parameter 4	parameter 5
Gaussian weight 5	parameter 1	parameter 2	parameter 3	parameter 4	parameter 5
Gaussian weight 6	parameter 1	parameter 2	parameter 3	parameter 4	parameter 5
Gaussian weight 7	parameter 1	parameter 2	parameter 3	parameter 4	parameter 5
Gaussian weight 8	parameter 1	parameter 2	parameter 3	parameter 4	parameter 5
Gaussian weight 9	parameter 1	parameter 2	parameter 3	parameter 4	parameter 5
Gaussian weight 10	parameter 1	parameter 2	parameter 3	parameter 4	parameter 5
Gaussian weight 11	parameter 1	parameter 2	parameter 3	parameter 4	parameter 5
Gaussian weight 12	parameter 1	parameter 2	parameter 3	parameter 4	parameter 5

FIGURE 5.3: Track inputs (inputs.csv)

Target 1	parameter 1	parameter 2	parameter 3	parameter 4	parameter 5
----------	-------------	-------------	-------------	-------------	-------------

FIGURE 5.4: Track targets (targets.csv)

CSV files have significant drawbacks when using them for data science and particularly machine learning applications, most notably their exponential reading and writing time order. Other commonly used data formats such as HDF5, ASDF and ROOT instead have a linear time order, which significantly improves efficiency. For this reason, those alternative data formats were examined in terms of time and memory complexity.

## 5.2 Baseline

Bernkopf implemented a feedforward neural network for electron track parameter reconstruction in his Master's Thesis (under revision). After trying out some optimization algorithms, he obtained the best results with MATLAB's *trainlm* optimizer which is a Levenberg-Marquardt algorithm and already improves on a simple residual mean calculation.

## 5.3 Tests

### 5.3.1 t-SNE dimensionality reduction

Let us now observe a visualization of the particle track data. Given that t-SNE uses Euclidean distances, it is affected by the "curse of dimensionality" and there is a limitation to the amount of data that can be used as input. Even though the available data includes 2,062,223 tracks, only 10,000 of them were randomly chosen. The dimensionality reduction was conducted with the perplexities 5, 30 and 50 and the results are shown below. Since the dimensions of the embedding have no precise meaning, the axes are not labeled.

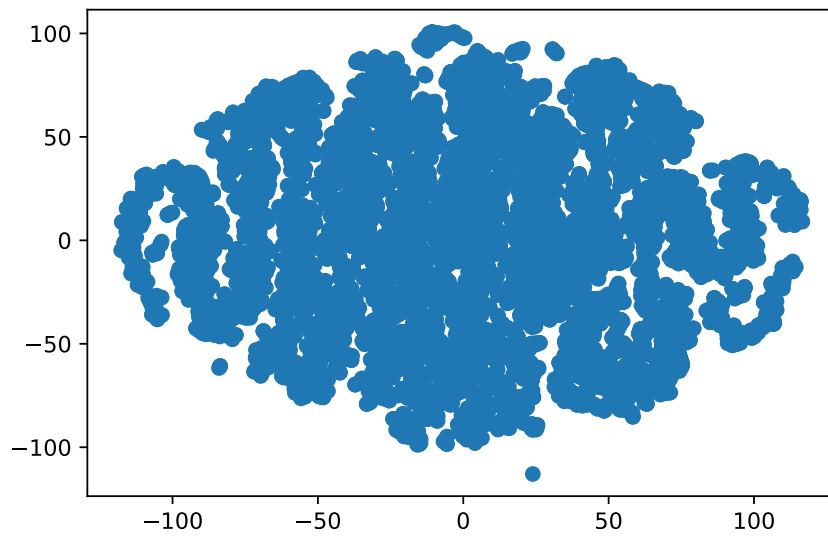


FIGURE 5.5: t-SNE dimensionality reduction of particle track data (perplexity=5)

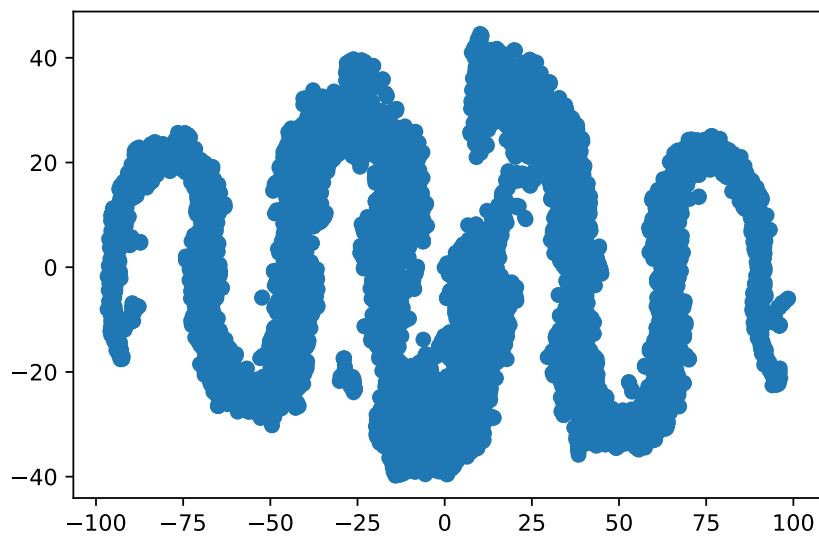


FIGURE 5.7: t-SNE dimensionality reduction of particle track data (perplexity=50)

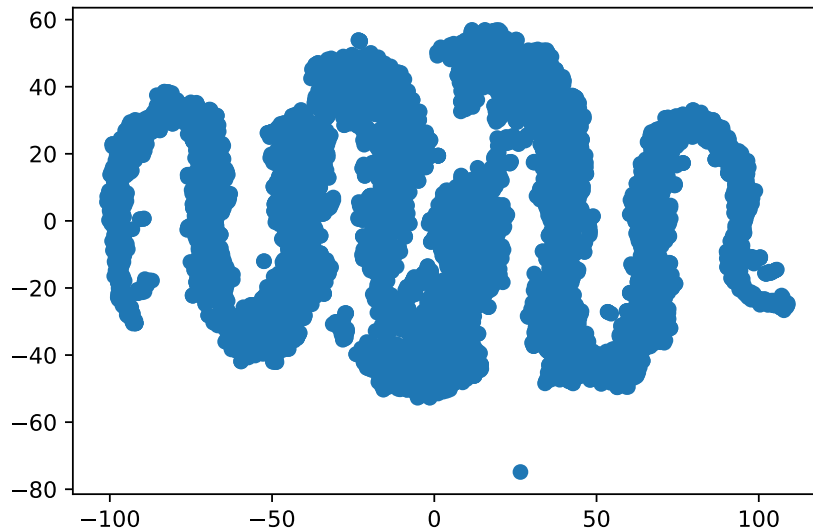


FIGURE 5.6: t-SNE dimensionality reduction of particle track data (perplexity=30)

As we can see, a low perplexity focuses too much on local structure, which is why a pattern is less visible. Its default value lies at 30, as shown in figure 5.6. Even though just 10,000 tracks were used, it can be clearly seen in the mentioned figure, that the data is strongly related. The configuration into a fully connected snake-like structure suggests a strong relationship between data points. All of the above observations are a strong indicator that the used data has been parsed correctly and actually represents the behaviour of a particle trajectory.

### 5.3.2 Test on Gaussian weights

Since every track is a Gaussian mixture, the Gaussian weights must add up to 1 for it to be normalized. If this were not the case, it would indicate some parsing error. The sum of Gaussian weights was examined and its mean, maximum and minimum across all tracks taken. The results are shown in table 5.1. Since all three statistics are sufficiently close to 1, there is no indication of any parsing errors in this respect.

Mean of sum of Gaussian weights	0.9999999940425501
Maximum of sum of Gaussian weights	1.0000002
Minimum of sum of Gaussian weights	0.9999998

TABLE 5.1: Mean, maximum and minimum of the sum of Gaussian weights across all tracks

### 5.3.3 Tests on correlation between targets and weighted averages

The simplest approximation of the target values is a weighted average of the twelve Gaussian components. Its correlation with the actual target data should of course be close to 1, which would mean that it already provides a reasonable estimate that



is then sought to be improved by a deep learning model which should be better at learning fine details and non-linearities of the problem. A deviation from this behaviour would indicate the insertion of some fault in the parsing process.

The MATLAB file *testdata.m* compares the weighted averages to the targets and visualizes them. The results for the first three parameters are shown in figures 5.8, 5.9 and 5.10.

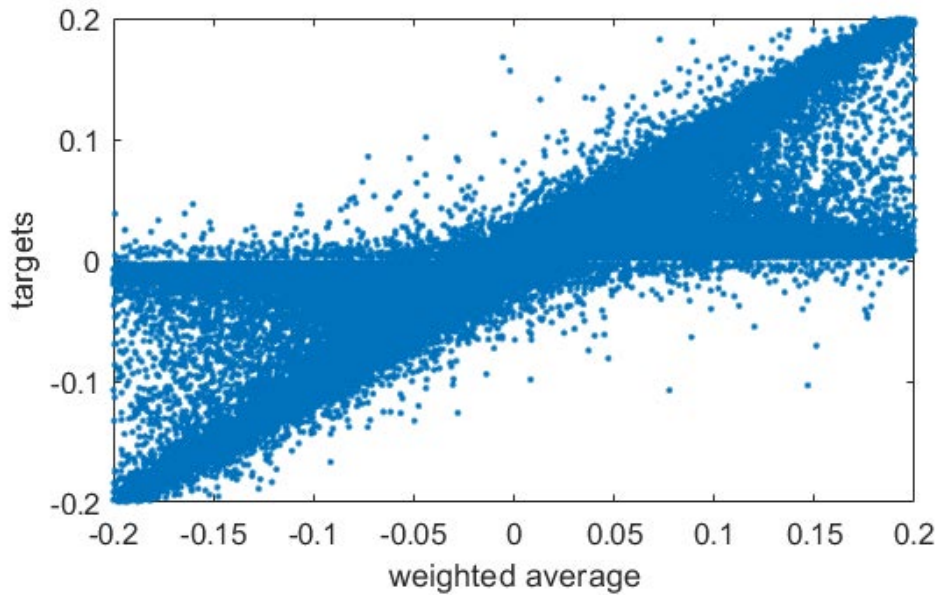


FIGURE 5.8: Weighted average of parameter 1 (q/p) vs. its target

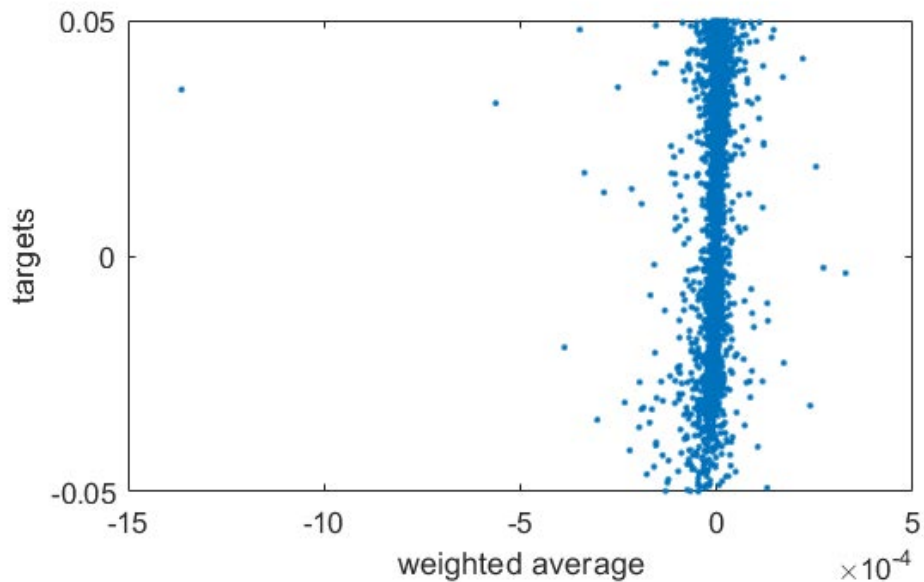


FIGURE 5.9: Weighted average of parameter 2 vs. its target

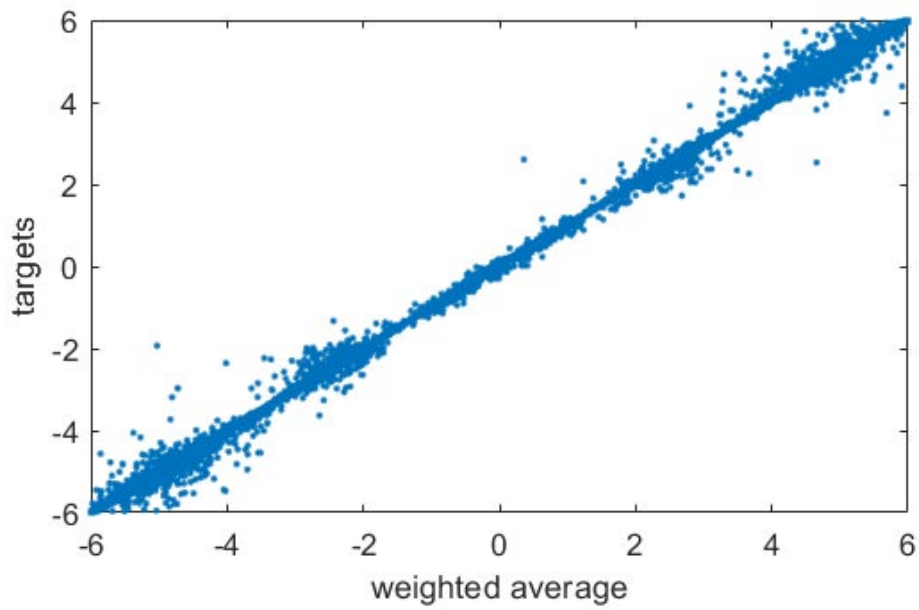


FIGURE 5.10: Weighted average of parameter 3 vs. its target

In all three parameters, a linear correlation can be seen. In figures 5.8 and 5.10 the correlation is, as expected, approximately one. Additionally, in figure 5.8 it can also be observed that as the target approaches zero, the weighted average predictions become increasingly worse due to the singularity of the first parameter ( $q/p$ ) at  $p = 0$ . In the case of figure 5.9 the weighted averages seem not to predict the targets very well. They predict values very close to zero. However, in the absence of any other abnormalities, this was assumed to just be a weakness of the prediction.

## Chapter 6

# Software Tools

There are several stages in the software development process, including task assessment, solution design, development, testing, documentation, and maintenance. This is commonly known as Systems Development Life Cycle. Having a solid development infrastructure is important for building useful, understandable, easy-to-use and easy-to-maintain software. Having maximal efficiency in all development and deployment stages in mind, the frameworks and packages described below were chosen.

### 6.1 Docker

Virtualization refers to the abstraction of applications, or also data storage, servers, etc., from the available resources. This can happen at different levels, namely on a hardware, operating system (OS), library or application basis. It can be used for a variety of tasks that require one thing in common: an isolation of a process by using resource abstraction. An example would be hard disk partitioning, to make distinct guest operating systems work on top of a host OS.

For creating applications, several libraries and packages are needed. However, there are usually numerous versions of them, and not all of them are compatible with each other. A virtual environment encapsulates a process, so that the packages installed therein, do not have any interaction with the outside. The deletion of such an environment would automatically eliminate all installed packages. Apart from allowing a more structured development, it also avoids contaminating the host OS with unnecessary packages.

Using virtualization is a matter of efficiency and good practice. With that said, containerization includes all of its benefits and goes beyond. It not only creates a totally isolated entity, the so-called container, but it does so by abstracting on the operating-system-level using a resource management system that efficiently shares common resources. In Docker, which is currently one of the most popular containerization softwares, this system is called *Docker Engine*. It requires significantly less space on disk and containers can be easily and quickly built and deployed and do not require the installation of an entire guest OS. Another advantage is the ability to build images which are a snapshot of a container. Entire trees of images can be built, with children nodes inheriting parent nodes' properties and building additional applications on top of them. With such a structure, debugging is reduced to evaluating the impact of a bug on children nodes, which makes the development process more structured. Moreover, inside the container, a different operating system to the host OS can be used without additional installations. In this work, the latest Ubuntu version was used. Finally, because containers are so easy to build, it is great for collaborative work and cross-platform application deployment. Virtual environments

still have their own strengths and applications, but for the purposes of this work, Docker is very well suited.

### How to install docker

The installation guides can be found on the following web sites:

- Windows: <https://docs.docker.com/docker-for-windows/install/>
- Ubuntu: <https://docs.docker.com/install/linux/docker-ce/ubuntu/>
- Mac OS: <https://docs.docker.com/docker-for-mac/install/>

### How to navigate through docker

Docker is organized into images and containers. Images are essentially snapshots of containers, or put another way, containers are instances of images. Information about all available images can be accessed by the command `docker images`, and the information about all containers by `docker ps -a`. A container ID is specified by a string of twelve alphanumeric characters.

To open and work with a previously created container, the following commands must be used:

- `docker start ***container ID***`
- `docker attach ***container ID***`

To exit the container and return to the terminal, the command `exit` must be executed.

### How to run a docker container?

A docker container can be easily created by just entering the following command in a command line:

```
docker run -ti -h container_name -v %cd%:/root/temp/ ...  
... -p 9999:9999 ubuntu:latest /bin/bash
```

This opens a new Ubuntu container with the name `container_name`, maps the current working directory, referenced to with `%cd%`, to the internal directory `/root/temp/` inside the container, maps the internal port 9999 with the external port 9999 and opens a bash shell. Mapping the communication ports is important for the proper use of the browser-based Python IDE called Jupyter that will later be discussed. Just with this command, there are still no packages available, apart from the default ones delivered with Ubuntu. The desired packages can either be installed manually or the installation commands incorporated into an installation script that should be located in the working directory where the container was created. The latter was chosen for this work, and the script named `installer.sh`. It must be executed using the following command:

```
source installer.sh
```

The script is depicted in the code below. Some parts of it were adopted from the `install.sh` script in <https://gist.github.com/yhilpisch/bda2479093216b299e59cf8c41bfa3e7> (Hilpisch, 2018) and adjusted. Each command is carefully commented, and after

each installed library, there is a short description of its function.

**Note:** It is important to execute it using *source* instead of *bash*, given that otherwise environment variables are not globally modified.

```
#!/bin/bash

# The advanced packaging tool or "apt" is a package management tool
# already built in into Ubuntu and many other Linux systems

# the flag -y automatically confirms the execution of a command

apt-get update # updates the package version lists
apt-get upgrade -y # upgrades packages according to those lists

# some Linux library installations
apt-get install -y vim # light-weight text editor
apt-get install -y wget # retrieves content from web servers
apt-get install -y htop # shows running processes and available
                        resources
apt-get install -y git # version-control software development tool

# packages for CERN ROOT
apt-get install -y libtool automake gettext \
gfortran libssl-dev libpcre3-dev \
xlibmesa-glu-dev libglew1.5-dev libftgl-dev \
libmysqlclient-dev libfftw3-dev libcfitsio-dev \
graphviz-dev libavahi-compat-libdnssd-dev \
libldap2-dev python-dev libxml2-dev libkrb5-dev \
libgs10-dev libqt4-dev
source /root/temp/root/bin/thisroot.sh # sets environment variables
                                     to activate root

apt-get upgrade -y bash # upgrades bash shell
apt-get clean # cleans apts local repository

# miniconda installation
wget https://repo.continuum.io/miniconda/Miniconda3-4.5.1-Linux-
      x86_64.sh -O Miniconda.sh #
      retrieves installation script
      for miniconda (version
      compatible with Python 3.6.7)
bash Miniconda.sh -b # runs miniconda installation script
rm -rf Miniconda.sh # removes script
export PATH="/root/miniconda3/bin:\$PATH" # sets path variable for
      miniconda in current session
export DISPLAY=localhost:0.0 # sets display variable for ROOT in
      current session
alias root="root -l" # suppresses welcome information when calling
      root

# set environment variables and aliases in bash configuration file
cat >> ~/.bashrc <<EOF
# for miniconda
export PATH="/root/miniconda3/bin:\$PATH"
```

```

# for CERN ROOT
export DISPLAY=localhost:0.0
alias root="root -l"
EOF

# conda installations
conda install -y jupyterlab # browser-based interactive IDE for
                             data science
conda install -y matplotlib # library for plots
conda install -y pytorch -c pytorch # deep learning framework used
                                     in this work

# pip installations
pip install --upgrade pip # upgrades pip package management tool
pip install plotly # library for interactive plots
pip install torchvision # package containing common datasets,
                        architectures and computer
                        vision tools

pip install asdf # allows to read and write asdf files
pip install --user root_numpy # allows to read and write root files
                              from numpy arrays
pip install uproot # allows to read and write root files from numpy
                  arrays

pip install memory-profiler # allows to keep track of memory usage
pip install scikit-learn # machine learning library

# move jupyter configuration file to its folder
mkdir /root/.jupyter/
cp /root/temp/jupyter_config/jupyter_notebook_config.py /root/.
    jupyter/

jupyter lab --allow-root

```

## 6.2 Jupyter Lab

Jupyter Lab is one of the most commonly used interactive integrated development environments for machine learning in Python. Its name derives from the programming languages JULia, PYthon and R, which are some of the many supported languages. It is a browser-based front end and offers abundant and rich functionalities.

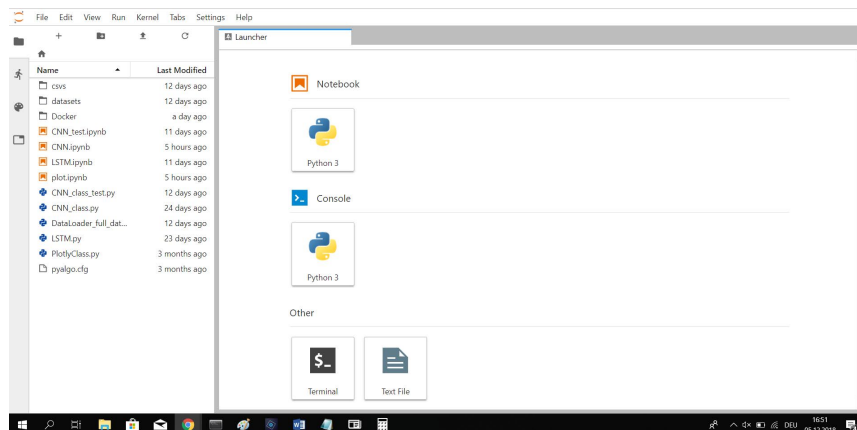


FIGURE 6.1: Typical jupyter lab front end

In figure 6.1, a typical jupyter lab front end is shown. As we can see, the environment is divided into a section on the left where the working directory with all of its files and folders is shown, and a section on the right where files can be opened and interacted with. The environment allows the generation of Python files, text files and a completely functional terminal. If ROOT is installed in the container, there is also an option for creating ROOT files.

One of the big advantages of Jupyter is that inside a notebook the code is divided into snippets. This makes debugging very straightforward. The individual parts of the code are executed one by one and problematic snippets can be quickly isolated. The order of execution is irrelevant. A snippet can be run by pressing `Shift + Enter`. In fact it is useful to know the most common key shortcuts. They can be found in the Command Palette.

### Accessing jupyter lab

In the configuration file, the port with which jupyter exchanges information should be specified. Additionally, to allow communication from inside a docker container to the outside, the flag `-allow-root` has to be placed when calling `jupyter lab`, i.e. `jupyter lab -allow-root`.

### Configuration file

Many Jupyter settings can be specified using a configuration file. For example the port through which the program communicates, whether or not a browser window should be automatically opened and SSL encryption certificates. In the following code snippet the Jupyter configuration file is shown.

```
c.NotebookApp.ip = '0.0.0.0' # binds to 0.0.0.0 IP address
c.NotebookApp.port = 9999 # binds to port 9999 which is the
                           communication port between the
                           container and the host system
c.NotebookApp.open_browser = False # avoids automatic browser
                                   opening
```

The command `c.NotebookApp.ip='*'` tells jupyter to connect to all possible ip addresses, while `c.NotebookApp.port=9999` tells it to connect to port 9999, which is the same port that communicates with the host system. That way data can be transmitted from within the container to a browser front end in the host OS through jupyter.

Whenever information is exchanged between local resources and a browser, especially on a server, the question of security arises. Using jupyter locally, with or without docker, is not more or less vulnerable than any other local operation, since the browser only accesses a local host. When accessing a server, it would be advisable to use an ssl encryption certificate. A password can also be configured, but would only secure direct access from the front end, ignoring commonly used cyber attacks.

## 6.3 Plotly/Dash

Plotly is an open source framework for creating interactive plots using only Python. Dash additionally allows to create entire Dashboards where it is possible to embed plotly plots and many additional features that would otherwise only be implementable with additional knowledge of some *html*, *css* and *javascript*. The online

version, running on plotly's server, also allows to create a live stream of data for up to 25 charts. If additional charts are needed, one must switch to a paid version. Plotly's offline chart generator does not support live streaming, but it is possible to update the data on a continuous basis. For this work, only plotly plots (as html files) were created without a Dashboard. Their structure and an example code are provided as follows.

### 6.3.1 Plotly plot

To build a plotly plot, the packages `plotly.offline` and `plotly.graph_objs` must be imported. Plotly supports plotting various variables in the same graph. The values for those variables must be stored in *traces* which are `graph_objs` objects. Also, to define the chart's layout, a `Layout` object, which also belongs to `graph_objs`, is necessary. All of those objects are then passed to a `Figure` object, which is also an instance of `plotly.graph_objs` and summarizes all the needed information for generating the plot. Then the plot generator `plot()`, which is a method of `plotly.offline` uses this object, together with a specified destination directory and filename, to generate an html file containing the plot. This plot is interactive, meaning that the data points can be hovered over, traces can be hidden or displayed, segments of the plot can be zoomed in or out, amongst other features that can be implemented additionally. Having knowledge of html, css or javascript can be useful for generating very customized features that are not part of the standard routines, but it is by no means necessary. `plotly.offline.plot()` is passed to a plot generator that is a method of `plotly.offline`. An example code would be:

```
import numpy as np
import plotly as py
import plotly.graph_objs as go

np.random.seed(42)

x = np.arange(100)
y = np.random.random(100)
z = np.random.random(100)+1

trace_1=go.Scatter(x=x, y=y, mode='lines', name='name 1 in legend')
trace_2=go.Scatter(x=x, y=z, mode='lines', name='name 2 in legend')
data = [trace_1, trace_2]
layout = go.Layout(title='Title')

figure = go.Figure(data=data, layout=layout)

py.offline.plot(figure, filename='test_plot.html')
```



## Chapter 7

# Step-by-step installation and execution Guide

### 7.1 Installation

All the software was designed to be fully automated, easy to install and user-friendly. With just a few commands it should be ready to execute on any Host Operating System, with the only requirement of having Docker installed. If it is not installed, follow the instructions in the Docker section of Chapter 6.

To create a docker container with all needed installations, open a terminal and follow the instructions below:

- `cd` to the desired directory containing the installation script `installer.sh`
- run `docker run -ti -h container_name -v %cd%:/root/temp/ ...`  
`... -p 9999:9999 ubuntu:latest /bin/bash`
- run `cd root/temp`
- run `source installer.sh`

When the installation finishes, a Jupyter Lab session is automatically initiated. To access Jupyter's front end, copy the link that appears in the terminal to a browser (some expressions in parentheses might have to be removed).

### 7.2 Executing a model

The file `model_configurations.py` can be found on the sidebar on the left. It contains all the possible configurations of the model, including

- architecture,
- epochs,
- learning rates,
- batch sizes,
- optimizers,
- and a saving directory for output files.

Each of those can be easily modified by commenting out undesirable options or by changing numerical values, simply following the instructions in the comments. After saving the file with the desired configuration, the next step is to execute the code in a terminal. To open a terminal, click "+" on the sidebar, after which a window with several options will appear. Then click "terminal". Once the terminal appears, run

- `bash`
- `python Model.py`

Some parts of the structure of the model were originally inspired by the Deep Learning Wizard PyTorch tutorials (Ng, 2018). From here on the training starts and several output files are generated. All directories where those files ought to be saved are created if they do not exist previously. Also, if a directory is not empty, the given configuration will be skipped to avoid unnecessary computation. Each step is reported in the terminal and the trained model will be saved. Already pretrained models can be used for prediction using `Model_pretrained.ipynb` or `Model_pretrained.py`.

#### Alternative:

The jupyter notebook file `Model.ipynb` contains the same model and one can execute the code snippet by snippet if desired. Each snippet is executed by pressing `Shift + Enter`. This is particularly useful for debugging because snippets can be skipped and run in any order. It is also possible to run all cells by choosing "Run" and "Run All Cells" in the menu bar. For running more complex configurations however, it is generally easier to just pick a configuration in `model_configurations.py` and run `python Model.py` in a terminal as discussed before.

#### Output files:

Several html files, two csv files and a `state_dict` are created per configuration. The html files show the complete loss history, an average loss history and a standard deviation history of training, validation and testing for the three parameters. The csv files contain some relevant information. One of them is a more complete description of the results, while the other is a summary that contains exactly the same values as the ones presented in Chapter 9. The `state_dict` file is saved in the `pretrained_models` directory and contains the network parameters of the trained model.

Once everything has been executed, the jupyter browser window can be closed, the jupyter session interrupted with `Ctrl + C` and the container exited by running `exit`.

#### **Future sessions**

For future sessions, accessing the container is straightforward. The command `docker ps -a` lists all available containers. Copying the container name and running `docker start container_name` and `docker attach container_name` starts the container. Then, by changing directory with `cd root/temp/` one can access all files. If ROOT is needed, one must first activate it by moving to `root/bin` and executing `source thisroot.sh`. To run a jupyter lab session, one must execute `jupyter lab -allow-root`.

## 7.3 Dataset Generator

The file `DataLoader.py` takes as input the data from the csv files "inputs.csv" and "targets.csv" and generates a dataset of a specified format. To generate it one must just execute `python DataLoader.py` and follow the instructions on the terminal. A dataset will be saved in the directory "datasets".

## 7.4 Tests

The jupyter notebook file `tests.ipynb` contains the following tests:

- Gaussian weight test: This is to make sure that the Gaussian weights in the tracks sum up to 1. The minimum, maximum and mean of those sums over all tracks are computed.
- t-SNE dimensionality reduction of a subset of the inputs tracks (10,000 inputs)

The test comparing targets with the weighted averages of the Gaussian components is not included here. For this, the matlab file `testdata.m` in the directory "matlab\_data\_generation\_and\_tests" was used. The results of all tests were already presented in Chapter 5.

## 7.5 Parameter list

The jupyter notebook file `parameter_list.ipynb` generates a list of the parameter shapes in each layer of a given network. The architecture is defined in `model_configurations.py` as always. The results are presented in Chapter 8.

## 7.6 Tests on data formats performance

The jupyter notebook file `test_data_formats.ipynb` tests the time and memory complexity of the reading process of three data formats: ROOT, HDF5 and ASDF. The results are presented in Chapter 9.

## Chapter 8

# Model architectures

### 8.1 Base architecture

In his Master's Thesis (under revision) Bernkopf analyzed several Feedforward Neural Network architectures and determined that the optimal configuration was one with 2 Fully Connected Layers of 48 and 24 neurons, and a readout layer of 3 neurons. This was used as the base architecture. Details on the exact shapes of all parameters in every layer of the network are shown in figure 8.1. As we can also see in the figure, the amount of parameters totals 4755.

### 8.2 Convolutional Neural Network architectures

The Convolutional Neural Network (CNN) architectures used in this work build on top of the base architecture, just stacking three Convolutional Layers with 12, 6 and 3 output channels in front of it.

As to the hyper-parameters, given the small size of the input features (12,6), the kernel size could not be chosen to be too large and there was little flexibility with the hyper-parameter choice. The minimal possible kernel size where next-neighbor information is still captured is  $k=2$ , so  $k=2$  and  $k=3$  were tested. To ensure fine granularity, a stride of 1 was chosen, meaning that the kernel slides over the input data in steps of only 1 data point.

Moreover an augmentation of the receptive field was tested using dilation. Padding describes the addition of data points around the inputs, so that edges can be better processed. While a padding of 1 was generally used (meaning the data is augmented by one data point on the edges), for dilated CNNs with kernel size 3 a padding size of 2 was needed, given that otherwise the feature maps become increasingly smaller until no convolutions are possible anymore.

With all those considerations in mind, three configurations which are shown below were implemented. The list in the curly brackets corresponds to the values for the three Convolutional Layers.

- **undilated:** kernel size = {2,2,2}, stride = {1,1,1}, padding = {1,1,1}, dilation = {1,1,1}
- **dilated (k=2):** kernel size = {2,2,2}, stride = {1,1,1}, padding = {1,1,1}, dilation = {1,2,3}
- **dilated (k=3):** kernel size = {3,3,3}, stride = {1,1,1}, padding = {2,2,2}, dilation = {1,2,3}

The shapes of the layers, displaying all parameters are shown in figure 8.2 for the undilated CNN, figure 8.3 for the dilated CNN with  $k = 2$ , and figure 8.4 for the dilated CNN with  $k = 3$ . The total amount of parameters are 21168, 12096 and 12606 respectively which makes the model significantly larger than the base architecture.

```
SHAPES OF THE FOLLOWING PARAMETER GROUPS:
```

```
FullyConnectedLayer1.weight: [48, 72]
```

```
FullyConnectedLayer1.bias: [48]
```

```
FullyConnectedLayer2.weight: [24, 48]
```

```
FullyConnectedLayer2.bias: [24]
```

```
FullyConnectedLayer3.weight: [3, 24]
```

```
FullyConnectedLayer3.bias: [3]
```

```
TOTAL AMOUNT OF PARAMETERS: 4755
```

FIGURE 8.1: Shapes of parameters in base architecture

SHAPES OF THE FOLLOWING PARAMETER GROUPS:

Convolution1.weight: [12, 1, 2, 2]

Convolution1.bias: [12]

Convolution2.weight: [6, 12, 2, 2]

Convolution2.bias: [6]

Convolution3.weight: [3, 6, 2, 2]

Convolution3.bias: [3]

FullyConnectedLayer1.weight: [48, 405]

FullyConnectedLayer1.bias: [48]

FullyConnectedLayer2.weight: [24, 48]

FullyConnectedLayer2.bias: [24]

FullyConnectedLayer3.weight: [3, 24]

FullyConnectedLayer3.bias: [3]

TOTAL AMOUNT OF PARAMETERS: 21168

FIGURE 8.2: Shapes of parameters in undilated CNN architecture

SHAPES OF THE FOLLOWING PARAMETER GROUPS:

Convolution1.weight: [12, 1, 2, 2]

Convolution1.bias: [12]

Convolution2.weight: [6, 12, 2, 2]

Convolution2.bias: [6]

Convolution3.weight: [3, 6, 2, 2]

Convolution3.bias: [3]

FullyConnectedLayer1.weight: [48, 216]

FullyConnectedLayer1.bias: [48]

FullyConnectedLayer2.weight: [24, 48]

FullyConnectedLayer2.bias: [24]

FullyConnectedLayer3.weight: [3, 24]

FullyConnectedLayer3.bias: [3]

TOTAL AMOUNT OF PARAMETERS: 12096

FIGURE 8.3: Shapes of parameters in dilated CNN architecture with kernel size 2

SHAPES OF THE FOLLOWING PARAMETER GROUPS:

Convolution1.weight: [12, 1, 3, 3]

Convolution1.bias: [12]

Convolution2.weight: [6, 12, 3, 3]

Convolution2.bias: [6]

Convolution3.weight: [3, 6, 3, 3]

Convolution3.bias: [3]

FullyConnectedLayer1.weight: [48, 216]

FullyConnectedLayer1.bias: [48]

FullyConnectedLayer2.weight: [24, 48]

FullyConnectedLayer2.bias: [24]

FullyConnectedLayer3.weight: [3, 24]

FullyConnectedLayer3.bias: [3]

TOTAL AMOUNT OF PARAMETERS: 12606

FIGURE 8.4: Shapes of parameters in dilated CNN architecture with kernel size 3



## Chapter 9

# Results

### 9.1 Data formats performance

In terms of processing speed, the chosen data format to import data into a model could at some point present a bottleneck, which is why three data formats were tested on their time and memory complexity. The results are presented below:

root	18s $\pm$ 106ms per loop, 7 runs, 1 loop each
hdf5	8.01s $\pm$ 145ms per loop, 7 runs, 1 loop each
asdf	424ms $\pm$ 34.9ms per loop, 7 runs, 1 loop each

TABLE 9.1: Time to import data and save it in a numpy array

root	10,852.14 MiB per loop, 7 runs, 1 loop each
hdf5	12,154.21 MiB per loop, 7 runs, 1 loop each
asdf	12,777.88 MiB per loop, 7 runs, 1 loop each

TABLE 9.2: Peak memory to import data and save it in a numpy array

`root_numpy`, `h5py` and `asdf` were used to import root, hdf5 and asdf files respectively. The results in principle suggest asdf as the preferred data format. However, root is CERN's standard format, making an implementation somewhat impractical.

### 9.2 Models performance

The optimization process for the architectures mentioned in Chapter 8 is arranged into two phases:

- **Phase I - Optimization of base architecture:**

All PyTorch optimizers (can be found at <https://pytorch.org/docs/stable/optim.html>), except *SparseAdam*, are tested for various learning rates and batch sizes. *SparseAdam* is not included because it only works with sparse gradients. Sparse gradients make use of only part of the gradient and require some modifications.

Batch sizes: 32, 64, 128, 256, 512.

Learning rates: default and one order of magnitude above and below.

- **Phase II - Addition of Convolutional Layers:**

After establishing the most effective hyper-parameters and optimizers for the base architecture, those configurations will be tested on the CNN architectures

presented in the previous Chapter, to see whether any of them can improve performance.

## 9.3 Phase I - Optimization of base architecture

### 9.3.1 Adadelata

	lr = 0.1	lr = 1	lr = 10
batch size = 32	(i) <b>0.592, 0.487, 0.388</b> (ii) <b>0.456, 0.586, 0.532</b> (iii) <b>0.595, 0.458, 0.335</b> (iv) <b>0.488, 0.555, 0.424</b>	(i) 0.630, 0.529, 1.317 (ii) 0.373, 0.458, 1.110 (iii) 0.546, 0.487, 1.416 (iv) 0.411, 0.636, 2.146	(i) 4.419, 8.173, 13.194 (ii) 1.236, 5.037, 9.361 (iii) 4.119, 2.687, 3.346 (iv) 0.614, 1.832, 5.426
batch size = 64	(i) 0.621, 0.628, 0.493 (ii) 0.366, 0.563, 0.459 (iii) 0.648, 0.701, 0.530 (iv) 0.381, 0.661, 0.632	(i) 0.715, 0.733, 2.098 (ii) 0.328, 0.671, 1.338 (iii) 0.701, 0.764, 1.454 (iv) 0.281, 0.341, 0.655	(i) 4.834, 12.342, 19.624 (ii) 1.321, 4.421, 10.438 (iii) 4.630, 16.144, 25.679 (iv) 0.460, 1.164, 3.139
batch size = 128	(i) 0.839, 0.816, 0.605 (ii) 0.364, 0.503, 0.476 (iii) 0.863, 0.813, 0.618 (iv) 0.463, 0.544, 0.960	(i) 0.815, 0.926, 2.983 (ii) 0.259, 0.418, 1.495 (iii) 0.692, 1.232, 2.610 (iv) 0.251, 0.283, 0.357	(i) 6.385, 17.449, 27.673 (ii) 1.308, 3.904, 9.813 (iii) 4.800, 16.585, 26.845 (iv) 0.304, 0.750, 1.373
batch size = 256	(i) 1.118, 0.935, 1.003 (ii) 0.314, 0.432, 0.449 (iii) 0.229, 1.302, 0.933 (iv) 3.106, 2.352, 2.355	(i) 0.933, 1.038, 3.562 (ii) 0.310, 0.423, 1.688 (iii) 0.678, 0.692, 3.744 (iv) 2.424, 1.778, 5.865	(i) 7.979, 20.662, 36.153 (ii) 1.374, 4.020, 9.170 (iii) 2.432, 5.962, 7.357 (iv) 7.963, 19.712, 29.488
batch size = 512	(i) 1.706, 1.202, 1.128 (ii) 0.183, 0.534, 0.492 (iii) 0.279, 1.860, 1.534 (iv) 0.229, 1.302, 0.933	(i) 1.150, 1.514, 6.540 (ii) 0.216, 0.427, 1.370 (iii) 0.612, 0.939, 3.183 (iv) 0.678, 0.692, 3.744	(i) 9.168, 23.514, 47.272 (ii) 1.250, 3.548, 7.943 (iii) 1.823, 7.087, 7.285 (iv) 2.432, 5.962, 7.357

TABLE 9.3: Performance of Adadelata optimizer for parameters 1, 2 and 3, relative to Bernkopf’s trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE

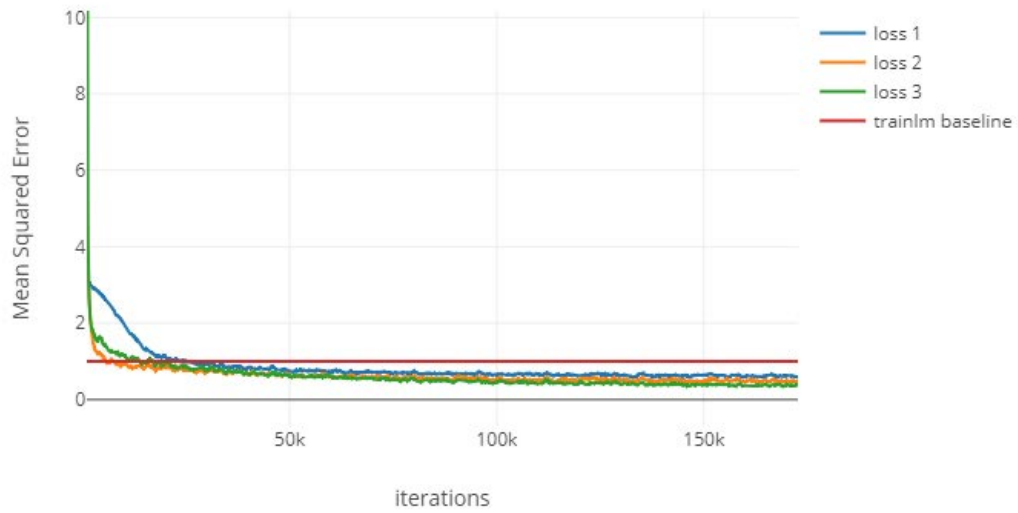


FIGURE 9.1: Adadelta training loss history for  $lr = 0.1$  and batch size = 32

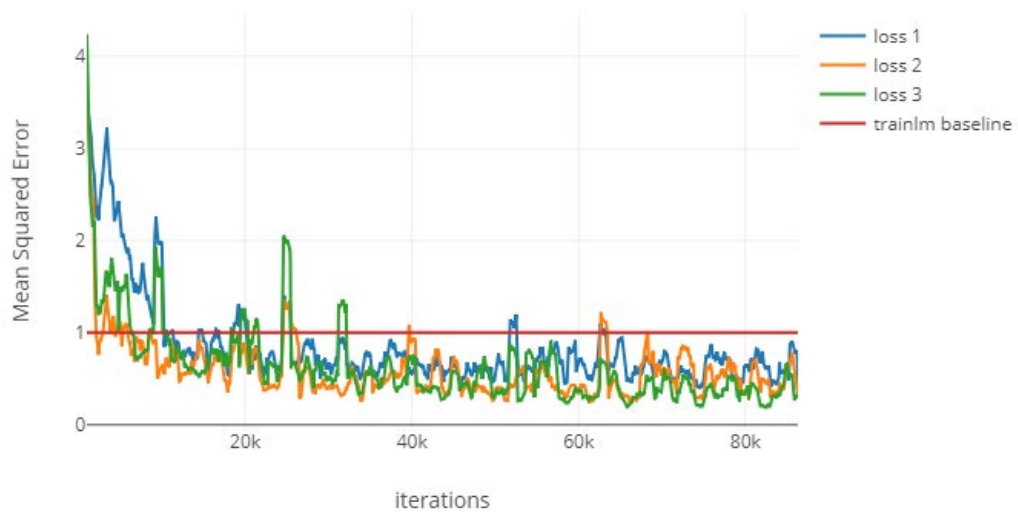


FIGURE 9.2: Adadelta validation loss history for  $lr = 0.1$  and batch size = 32

## 9.3.2 Adagrad

	lr = 0.001	lr = 0.01	lr = 0.1
batch size = 32	(i) 0.894, 0.812, 0.743 (ii) 0.629, 1.008, 1.028 (iii) 0.840, 0.797, 0.674 (iv) 0.594, 0.965, 0.817	(i) 0.534, 0.449, 0.325 (ii) 0.412, 0.654, 0.510 (iii) 0.535, 0.418, 0.347 (iv) 0.689, 0.553, 1.119	(i) 0.599, 0.62, 0.531 (ii) 0.439, 0.843, 0.675 (iii) 0.607, 0.602, 0.577 (iv) 0.462, 1.005, 1.321
batch size = 64	(i) 1.208, 1.122, 1.175 (ii) 0.578, 0.875, 1.182 (iii) 1.172, 1.095, 1.212 (iv) 0.546, 0.914, 1.276	(i) 0.631, 0.554, 0.488 (ii) 0.401, 0.734, 1.049 (iii) 0.618, 0.566, 0.460 (iv) 0.413, 0.680, 0.757	(i) 0.760, 1.557, 0.963 (ii) 0.399, 1.528, 1.371 (iii) 0.744, 1.228, 1.037 (iv) 0.408, 1.367, 2.197
batch size = 128	(i) 0.979, 1.212, 1.178 (ii) 0.430, 0.749, 0.811 (iii) 0.997, 1.253, 1.229 (iv) 0.438, 0.770, 1.460	<b>(i) 0.728, 0.717, 0.642</b> <b>(ii) 0.395, 0.525, 0.615</b> <b>(iii) 0.713, 0.687, 0.687</b> <b>(iv) 0.309, 0.462, 0.498</b>	(i) 0.874, 1.743, 0.797 (ii) 0.332, 1.123, 0.675 (iii) 0.867, 1.717, 0.819 (iv) 0.351, 1.175, 0.787
batch size = 256	(i) 1.312, 1.218, 1.439 (ii) 0.311, 0.632, 0.800 (iii) 2.740, 1.615, 18.796 (iv) 5.736, 2.134, 18.009	(i) 0.776, 0.695, 0.638 (ii) 0.250, 0.419, 0.534 (iii) 0.297, 0.934, 0.882 (iv) 1.020, 1.543, 1.514	(i) 0.941, 1.422, 2.008 (ii) 0.251, 1.025, 3.130 (iii) 0.510, 1.539, 2.681 (iv) 1.539, 3.003, 3.465
batch size = 512	(i) 1.161, 1.363, 1.758 (ii) 0.222, 0.584, 0.631 (iii) 0.291, 2.149, 18.278 (iv) 2.740, 1.615, 18.796	(i) 0.772, 0.997, 0.834 (ii) 0.181, 0.513, 0.880 (iii) 0.262, 0.293, 0.429 (iv) 0.297, 0.934, 0.882	(i) 1.330, 1.938, 1.883 (ii) 0.526, 0.807, 1.496 (iii) 0.417, 0.476, 2.130 (iv) 0.510, 1.539, 2.681

TABLE 9.4: Performance of Adagrad optimizer for parameters 1, 2 and 3, relative to Bernkopf’s trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE

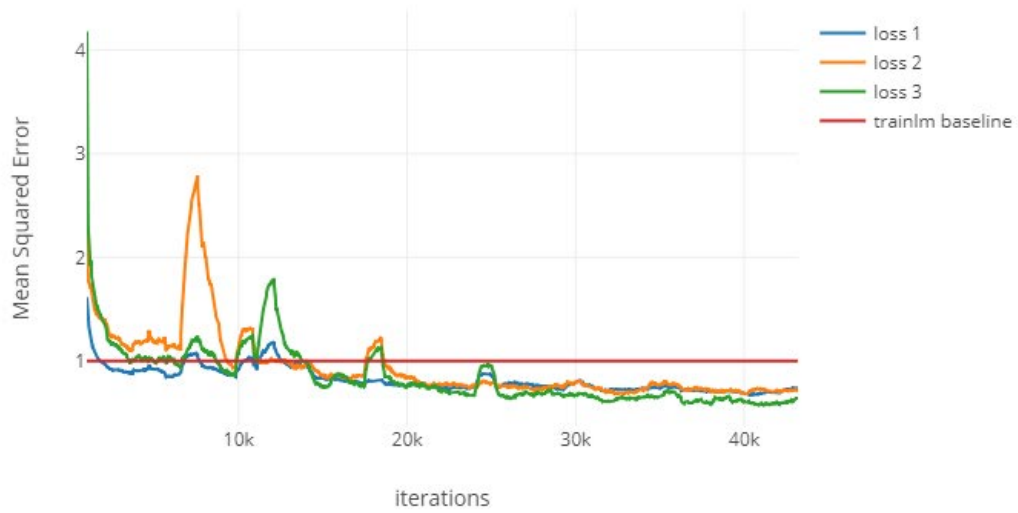


FIGURE 9.3: Adagrad training loss history for  $lr = 0.01$  and batch size = 128

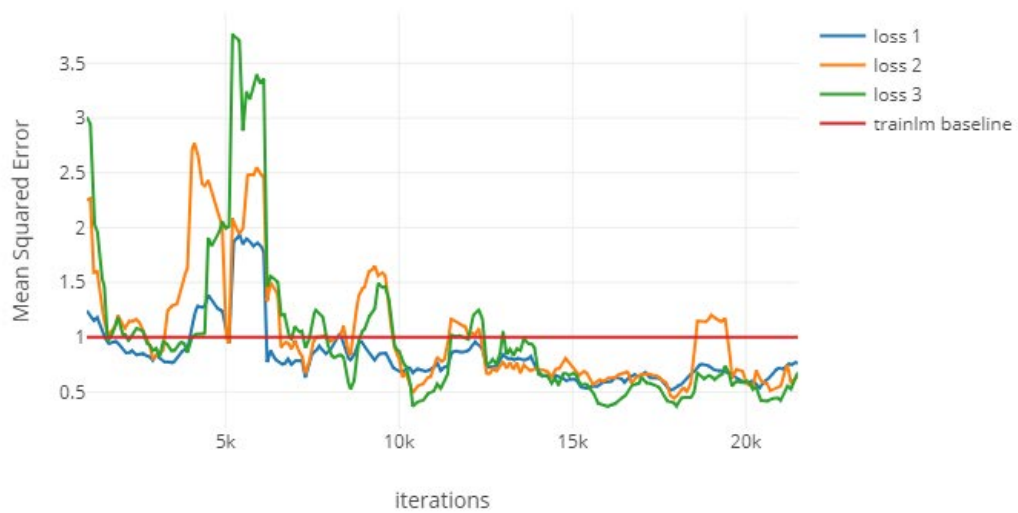


FIGURE 9.4: Adagrad validation loss history for  $lr = 0.01$  and batch size = 128

## 9.3.3 Adam

	lr = 0.0001	lr = 0.001	lr = 0.01
batch size = 32	(i) <b>0.506, 0.433, 0.292</b> (ii) <b>0.366, 0.539, 0.438</b> (iii) <b>0.480, 0.377, 0.223</b> (iv) <b>0.377, 0.461, 0.409</b>	(i) 0.574, 0.837, 0.674 (ii) 0.365, 1.162, 1.487 (iii) 0.543, 0.770, 0.465 (iv) 0.375, 0.672, 1.254	(i) 3.349, 2.485, 2.222 (ii) 0.782, 2.13, 6.051 (iii) 3.12, 1.906, 0.747 (iv) 0.677, 1.906, 2.825
batch size = 64	(i) 0.533, 0.495, 0.335 (ii) 0.306, 0.762, 0.587 (iii) 0.522, 0.683, 0.371 (iv) 0.338, 0.522, 0.739	(i) 0.588, 0.701, 0.540 (ii) 0.290, 0.532, 0.997 (iii) 0.581, 0.739, 0.715 (iv) 0.280, 0.766, 0.791	(i) 3.380, 2.676, 6.394 (ii) 0.590, 2.178, 15.150 (iii) 3.631, 2.386, 3.987 (iv) 0.493, 1.527, 5.266
batch size = 128	(i) 0.656, 0.590, 0.574 (ii) 0.345, 0.526, 0.940 (iii) 0.669, 0.590, 0.500 (iv) 0.467, 0.882, 0.501	(i) 0.690, 1.045, 1.265 (ii) 0.343, 1.105, 1.860 (iii) 0.608, 0.828, 0.726 (iv) 0.268, 0.616, 0.595	(i) 3.270, 2.541, 6.675 (ii) 0.380, 1.845, 11.694 (iii) 3.156, 2.379, 5.310 (iv) 0.342, 1.336, 7.610
batch size = 256	(i) 0.791, 0.951, 0.996 (ii) 0.942, 1.929, 3.173 (iii) 1.204, 2.089, 29.731 (iv) 2.618, 2.886, 17.335	(i) 0.658, 0.773, 1.008 (ii) 0.265, 0.555, 2.350 (iii) 0.430, 0.476, 1.057 (iv) 1.205, 1.559, 1.730	(i) 1.138, 2.947, 2.711 (ii) 0.569, 1.290, 3.953 (iii) 3.261, 1.983, 11.476 (iv) 2.740, 3.293, 8.884
batch size = 512	(i) 0.817, 0.870, 0.722 (ii) 0.202, 0.520, 0.456 (iii) 0.814, 2.080, 26.538 (iv) 1.204, 2.089, 29.731	(i) 0.710, 0.805, 0.880 (ii) 0.168, 0.321, 0.670 (iii) 0.182, 0.346, 1.164 (iv) 0.430, 0.476, 1.057	(i) 1.417, 3.454, 5.456 (ii) 0.442, 2.036, 5.366 (iii) 0.444, 2.261, 7.076 (iv) 3.261, 1.983, 11.476

TABLE 9.5: Performance of Adam optimizer for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE

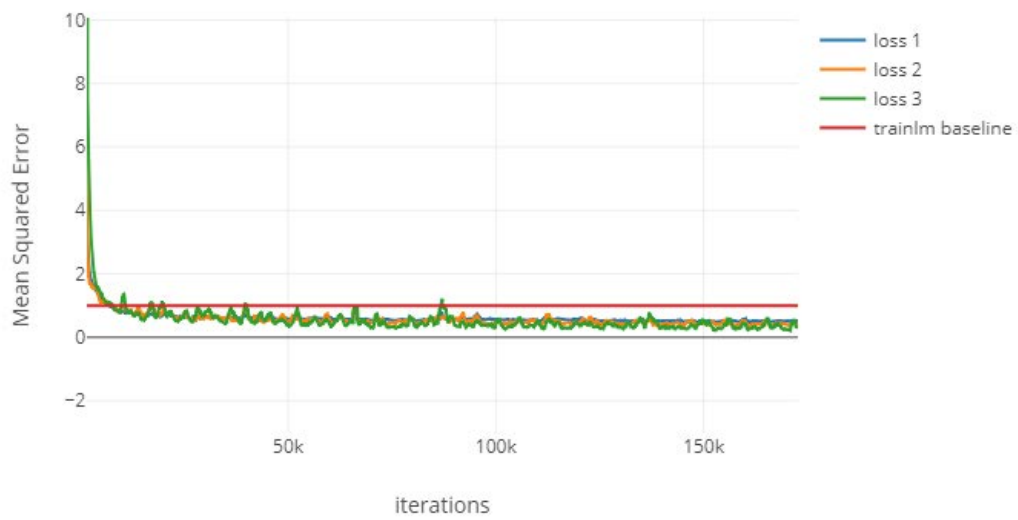


FIGURE 9.5: Adam training loss history for  $lr = 0.0001$  and batch size = 64

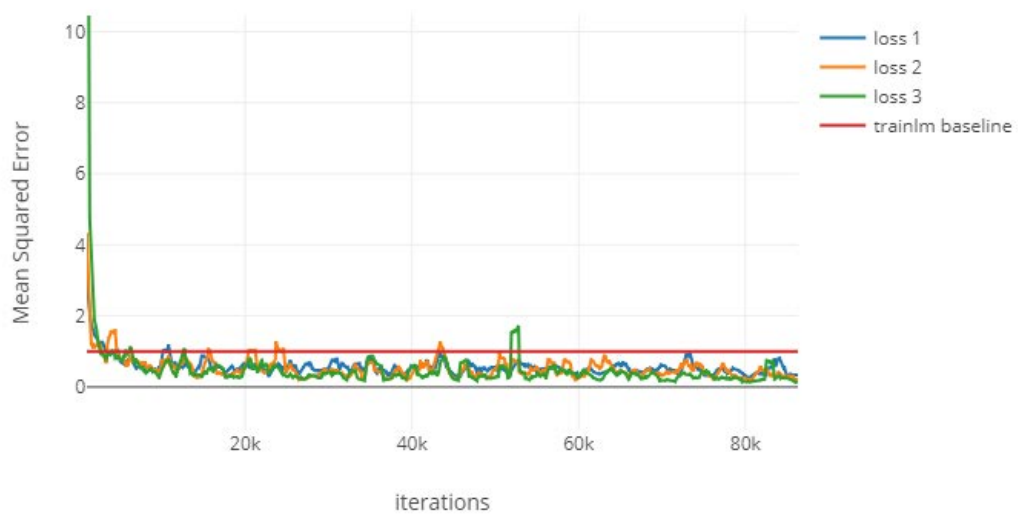


FIGURE 9.6: Adam validation loss history for  $lr = 0.0001$  and batch size = 64



## 9.3.4 Adamax

	lr = 0.0001	lr = 0.001	lr = 0.01
batch size = 32	(i) 0.631, 0.563, 0.397 (ii) 0.530, 1.446, 1.150 (iii) 0.582, 0.485, 0.343 (iv) 0.443, 0.845, 0.521	(i) 0.453, 0.356, 0.248 (ii) 0.329, 0.446, 0.375 (iii) 0.456, 0.352, 0.234 (iv) 0.336, 0.522, 0.868	(i) 0.575, 0.566, 0.432 (ii) 0.322, 0.431, 0.824 (iii) 0.589, 0.644, 0.220 (iv) 0.303, 0.360, 0.347
batch size = 64	(i) 0.716, 0.666, 0.586 (ii) 0.432, 0.737, 0.824 (iii) 0.689, 0.633, 0.614 (iv) 0.368, 0.618, 2.531	<b>(i) 0.527, 0.405, 0.284</b> <b>(ii) 0.308, 0.365, 0.355</b> <b>(iii) 0.483, 0.391, 0.252</b> <b>(iv) 0.289, 0.411, 0.407</b>	(i) 0.615, 0.583, 0.599 (ii) 0.301, 0.505, 1.372 (iii) 0.529, 0.742, 0.496 (iv) 0.284, 0.254, 0.796
batch size = 128	(i) 0.796, 0.862, 0.820 (ii) 0.332, 0.687, 0.817 (iii) 0.793, 0.913, 0.816 (iv) 0.357, 0.940, 0.825	(i) 0.585, 0.525, 0.473 (ii) 0.250, 0.404, 0.691 (iii) 0.572, 0.515, 0.560 (iv) 0.249, 0.392, 0.585	(i) 0.563, 0.582, 0.633 (ii) 0.227, 0.358, 0.801 (iii) 0.575, 0.576, 0.514 (iv) 0.225, 0.331, 1.169
batch size = 256	(i) 0.854, 0.936, 1.022 (ii) 0.289, 0.604, 0.855 (iii) 1.098, 7.555, 40.523 (iv) 4.139, 13.669, 34.051	(i) 0.687, 0.659, 0.565 (ii) 0.375, 0.680, 0.790 (iii) 0.511, 1.203, 1.821 (iv) 1.536, 2.531, 3.060	(i) 0.667, 0.813, 0.973 (ii) 0.299, 0.744, 2.005 (iii) 0.291, 1.045, 1.881 (iv) 0.951, 1.658, 1.813
batch size = 512	(i) 1.190, 1.350, 1.841 (ii) 0.315, 1.101, 3.976 (iii) 2.061, 8.551, 33.488 (iv) 1.098, 7.555, 40.523	(i) 0.641, 0.716, 0.666 (ii) 0.144, 0.498, 0.471 (iii) 0.396, 0.367, 1.691 (iv) 0.511, 1.203, 1.821	(i) 0.642, 0.723, 0.800 (ii) 0.196, 0.524, 1.130 (iii) 0.273, 0.755, 2.129 (iv) 0.291, 1.045, 1.881

TABLE 9.6: Performance of Adamax optimizer for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE

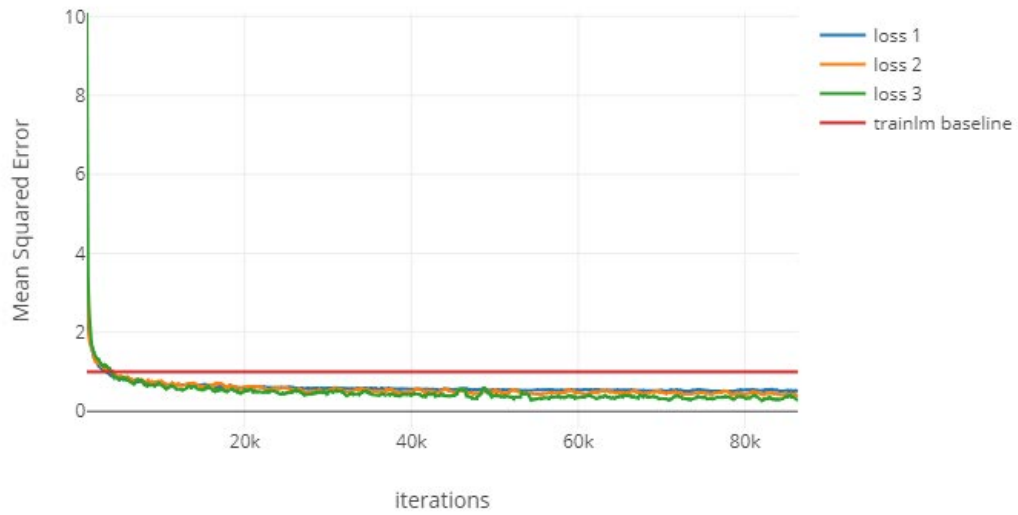


FIGURE 9.7: Adamax training loss history for  $lr = 0.001$  and batch size = 64

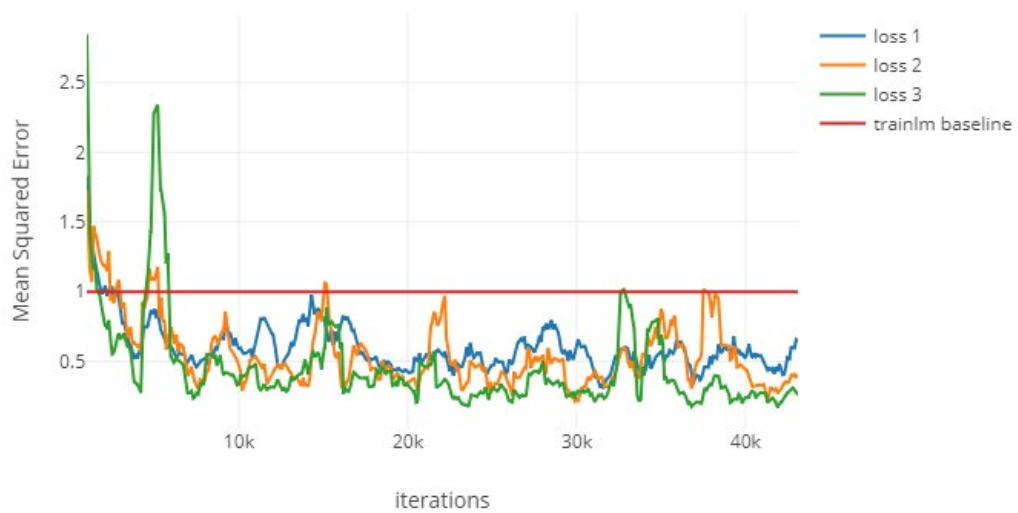


FIGURE 9.8: Adamax validation loss history for  $lr = 0.001$  and batch size = 64

## 9.3.5 ASGD

	lr = 0.001	lr = 0.01	lr = 0.1
batch size = 32	(i) 3.069, 1.275, 1.656 (ii) 0.686, 1.171, 1.411 (iii) 3.040, 1.251, 1.686 (iv) 0.668, 1.219, 1.512	<b>(i) 1.106, 0.895, 1.305</b> <b>(ii) 0.487, 0.816, 2.700</b> <b>(iii) 1.107, 0.894, 1.088</b> <b>(iv) 0.517, 0.865, 1.242</b>	(i) nan, nan, nan (ii) nan, nan, nan (iii) nan, nan, nan (iv) nan, nan, nan
batch size = 64	(i) 3.085, 1.787, 2.240 (ii) 0.559, 1.107, 1.994 (iii) 3.062, 1.809, 2.345 (iv) 0.570, 1.213, 4.314	(i) 2.206, 1.159, 0.976 (ii) 0.389, 0.752, 0.872 (iii) 2.179, 1.157, 0.959 (iv) 0.364, 0.771, 0.889	(i) nan, nan, nan (ii) nan, nan, nan (iii) nan, nan, nan (iv) nan, nan, nan
batch size = 128	(i) 2.989, 2.574, 2.718 (ii) 0.399, 1.659, 1.803 (iii) 2.990, 2.654, 2.801 (iv) 0.358, 1.793, 1.920	(i) 3.016, 1.243, 1.329 (ii) 0.323, 1.105, 0.996 (iii) 3.008, 1.195, 1.357 (iv) 0.328, 0.790, 0.988	(i) 3.142, 1.952, 26.386 (ii) 0.352, 1.596, 18.596 (iii) 3.208, 1.926, 24.240 (iv) 0.332, 1.513, 17.718
batch size = 256	(i) 3.364, 3.622, 3.980 (ii) 0.381, 1.604, 1.346 (iii) 1.388, 5.550, 40.273 (iv) 16.047, 21.541, 95.066	(i) 3.008, 1.355, 1.905 (ii) 0.246, 0.665, 0.891 (iii) 0.498, 2.411, 11.339 (iv) 3.366, 4.253, 10.294	(i) 3.251, 2.499, 29.009 (ii) 0.349, 1.265, 12.857 (iii) 0.254, 0.617, 0.897 (iv) 3.096, 1.768, 2.864
batch size = 512	(i) 3.315, 4.484, 4.707 (ii) 0.250, 1.723, 1.246 (iii) 1.044, 2.848, 29.827 (iv) 1.388, 5.550, 40.273	(i) 3.275, 2.313, 2.290 (ii) 0.197, 0.773, 0.885 (iii) 1.634, 1.077, 9.038 (iv) 0.498, 2.411, 11.339	(i) 2.663, 1.376, 1.375 (ii) 0.148, 0.428, 0.851 (iii) 0.237, 1.896, 4.871 (iv) 0.254, 0.617, 0.897

TABLE 9.7: Performance of ASGD optimizer for parameters 1, 2 and 3, relative to Bernkopf’s trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE

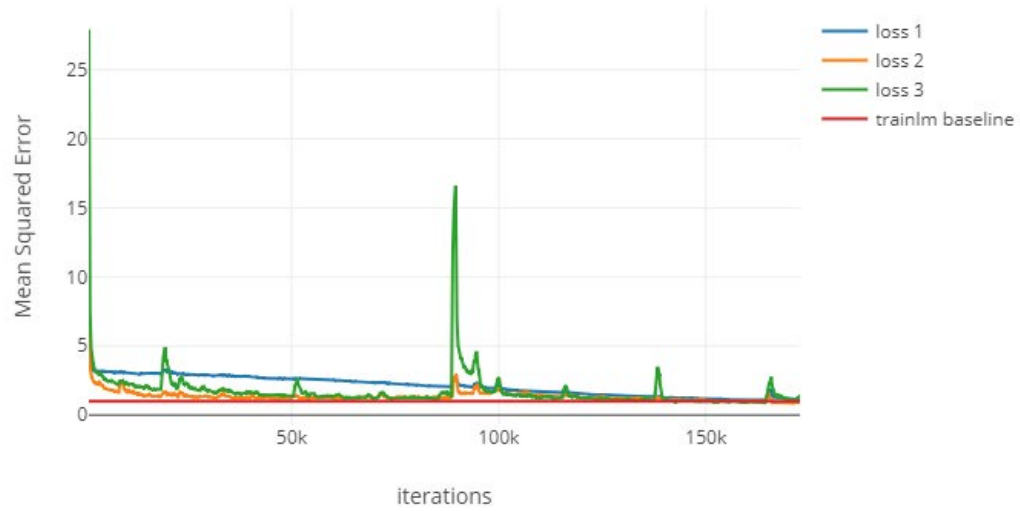


FIGURE 9.9: ASGD training loss history for  $\text{lr} = 0.01$  and batch size = 32

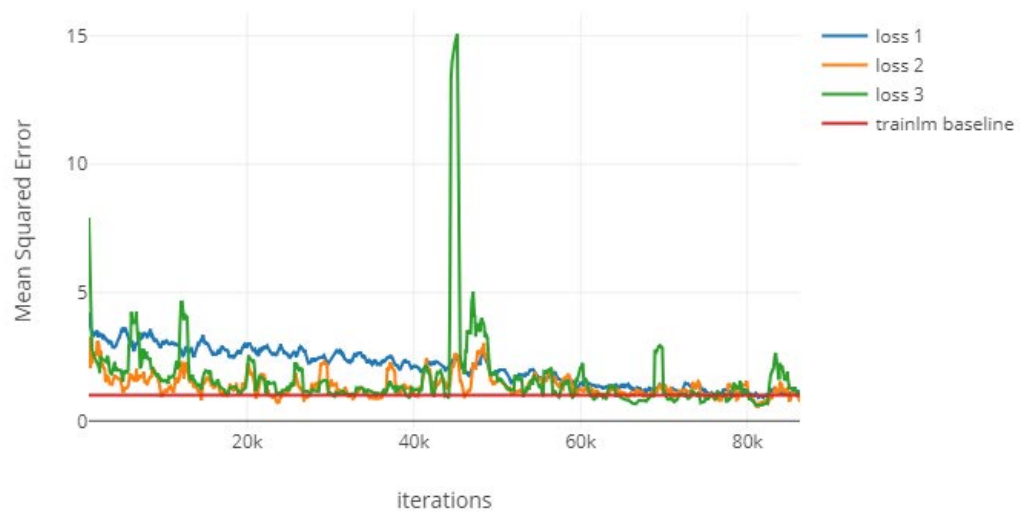


FIGURE 9.10: ASGD validation loss history for  $\text{lr} = 0.01$  and batch size = 32

## 9.3.6 RMSprop

	lr = 0.001	lr = 0.01	lr = 0.1
batch size = 32	(i) <b>0.518, 0.551, 0.771</b> (ii) <b>0.344, 0.524, 0.918</b> (iii) <b>0.425, 0.340, 0.332</b> (iv) <b>0.342, 0.460, 0.819</b>	(i) 3.410, 3.596, 5.366 (ii) 0.806, 3.091, 6.995 (iii) 3.255, 1.778, 1.681 (iv) 0.683, 1.838, 1.727	(i) 4.825, 9.985, 161.172 (ii) 1.896, 4.546, 20.070 (iii) 3.905, 10.257, 161.615 (iv) 0.651, 0.608, 19.456
batch size = 64	(i) 0.608, 0.765, 1.042 (ii) 0.290, 0.598, 1.203 (iii) 0.515, 1.220, 1.328 (iv) 0.281, 0.403, 0.627	(i) 2.285, 3.696, 5.638 (ii) 0.912, 2.754, 7.079 (iii) 1.778, 4.844, 6.511 (iv) 0.424, 1.210, 2.316	(i) 4.818, 9.987, 162.280 (ii) 1.856, 4.722, 14.126 (iii) 4.986, 12.442, 162.138 (iv) 0.461, 0.469, 14.376
batch size = 128	(i) 0.616, 0.782, 1.224 (ii) 0.241, 0.438, 1.002 (iii) 0.954, 1.220, 2.031 (iv) 0.160, 0.369, 0.523	(i) 2.321, 4.114, 8.068 (ii) 1.192, 2.911, 21.622 (iii) 2.988, 5.202, 7.966 (iv) 0.443, 0.967, 3.317	(i) 4.935, 10.638, 161.742 (ii) 1.390, 2.785, 10.214 (iii) 3.353, 12.337, 171.993 (iv) 0.348, 0.251, 10.697
batch size = 256	(i) 0.665, 0.985, 1.850 (ii) 0.216, 0.540, 1.376 (iii) 0.409, 0.794, 2.009 (iv) 1.084, 2.084, 2.747	(i) 2.325, 4.399, 7.043 (ii) 0.778, 2.479, 5.856 (iii) 1.639, 4.921, 7.832 (iv) 3.116, 6.942, 13.086	(i) 4.820, 10.917, 162.461 (ii) 1.707, 1.965, 7.203 (iii) 18.712, 47.492, 55.043 (iv) 13.498, 29.450, 96.348
batch size = 512	(i) 0.780, 1.136, 2.301 (ii) 0.218, 0.741, 2.117 (iii) 0.536, 1.198, 1.102 (iv) 0.409, 0.794, 2.009	(i) 2.061, 4.488, 7.460 (ii) 0.835, 2.785, 5.556 (iii) 2.070, 3.910, 6.042 (iv) 1.639, 4.921, 7.832	(i) 4.855, 10.977, 162.417 (ii) 1.616, 1.443, 5.172 (iii) 7.412, 12.196, 34.938 (iv) 18.712, 47.492, 55.043

TABLE 9.8: Performance of RMSprop optimizer for parameters 1, 2 and 3, relative to Bernkopf’s trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE

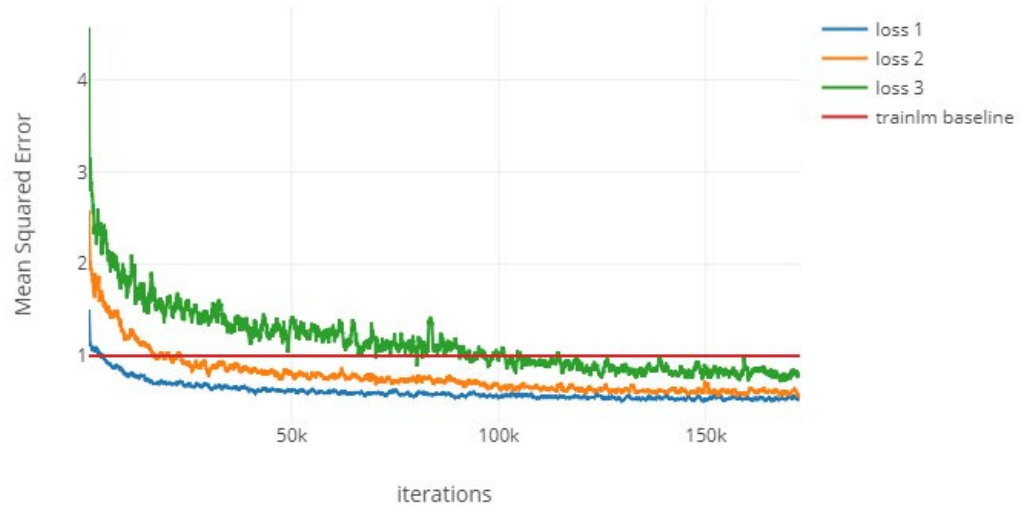


FIGURE 9.11: RMSprop training loss history for  $lr = 0.001$  and batch size = 32

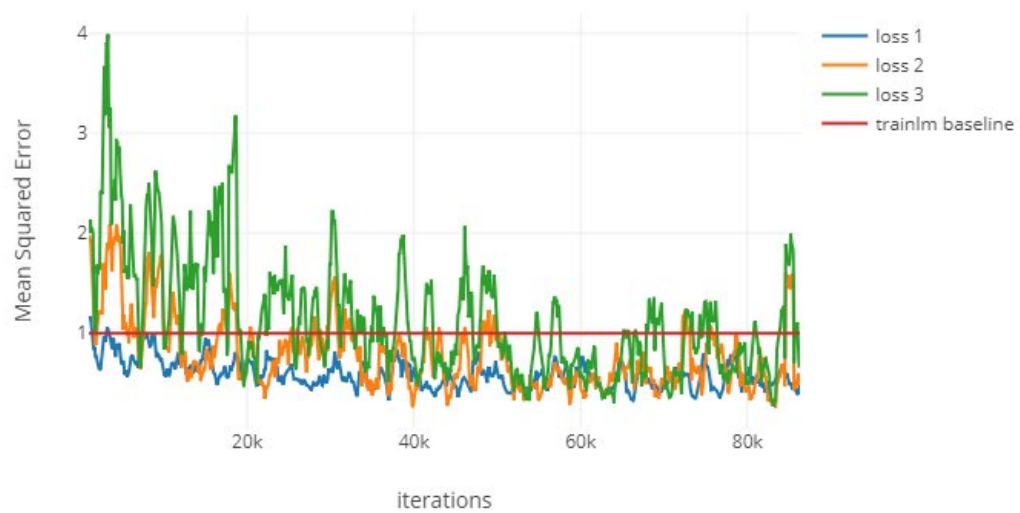


FIGURE 9.12: RMSprop validation loss history for  $lr = 0.001$  and batch size = 32

## 9.3.7 Rprop

	lr = 0.001	lr = 0.01	lr = 0.1
batch size = 32	(i) 1.189, 1.046, 1.146 (ii) 2.536, 2.537, 2.865 (iii) 1.083, 0.968, 1.009 (iv) 1.708, 2.254, 1.942	(i) 1.053, 1.462, 1.265 (ii) 1.305, 4.611, 2.486 (iii) 1.124, 1.563, 1.369 (iv) 1.398, 4.588, 2.626	(i) 2.881, 6.729, 5.855 (ii) 6.772, 15.235, 15.201 (iii) 3.588, 7.291, 5.821 (iv) 11.949, 17.748, 14.149
batch size = 64	(i) 1.051, 1.406, 1.182 (ii) 0.965, 2.657, 2.388 (iii) 1.112, 1.553, 1.514 (iv) 1.051, 2.630, 3.488	(i) 1.069, 1.324, 2.004 (ii) 0.938, 2.062, 4.055 (iii) 1.143, 1.241, 1.874 (iv) 1.945, 1.853, 3.459	(i) 4.011, 8.065, 10.334 (ii) 8.150, 12.323, 18.490 (iii) 3.973, 9.072, 11.920 (iv) 7.523, 17.887, 27.267
batch size = 128	(i) 1.064, 1.375, 1.222 (ii) 0.689, 1.264, 2.807 (iii) 1.049, 1.322, 1.160 (iv) 0.880, 1.146, 1.162	(i) 1.153, 1.061, 1.520 (ii) 1.617, 0.855, 1.861 (iii) 1.257, 1.164, 1.749 (iv) 1.447, 1.193, 2.566	(i) 6.040, 12.576, 11.053 (ii) 8.112, 15.276, 16.828 (iii) 6.078, 12.719, 10.927 (iv) 11.342, 16.180, 23.020
batch size = 256	(i) 1.047, 1.371, 1.229 (ii) 0.396, 1.136, 0.909 (iii) 0.464, 0.658, 0.827 (iv) 1.431, 2.197, 2.324	(i) 1.068, 1.501, 1.498 (ii) 0.456, 1.431, 1.247 (iii) 0.353, 1.042, 0.687 (iv) 1.807, 3.034, 2.220	(i) 3.324, 6.130, 7.537 (ii) 7.134, 7.612, 4.979 (iii) 2.250, 5.169, 3.389 (iv) 3.734, 8.616, 8.116
batch size = 512	<b>(i) 1.108, 1.368, 1.177</b> <b>(ii) 0.370, 0.690, 0.719</b> <b>(iii) 0.267, 0.618, 0.817</b> <b>(iv) 0.464, 0.658, 0.827</b>	(i) 0.982, 2.100, 1.517 (ii) 0.379, 3.692, 1.658 (iii) 0.324, 0.606, 0.840 (iv) 0.353, 1.042, 0.687	(i) 1.977, 4.040, 11.190 (ii) 0.814, 2.227, 8.059 (iii) 0.998, 1.677, 7.482 (iv) 2.250, 5.169, 3.389

TABLE 9.9: Performance of Rprop optimizer for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE

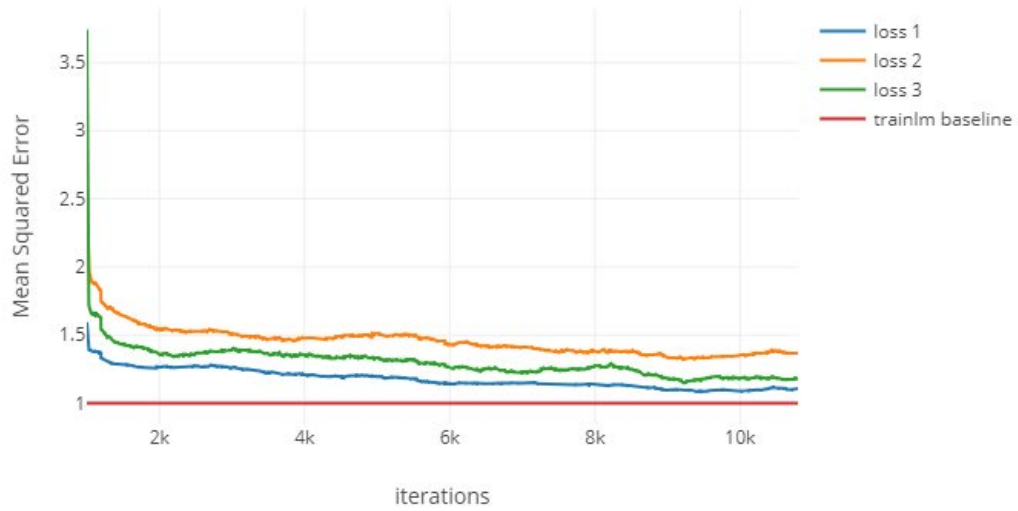


FIGURE 9.13: Rprop training loss history for  $lr = 0.001$  and batch size = 512

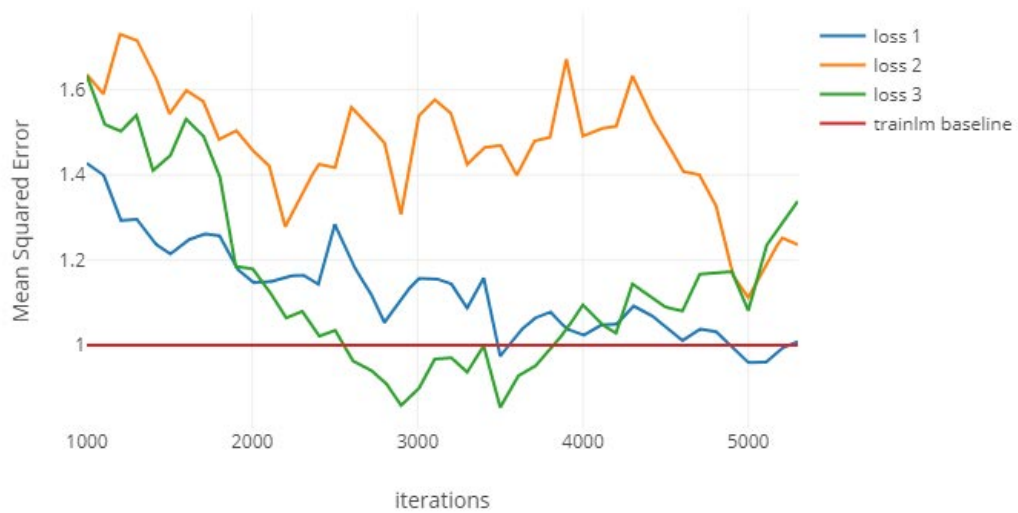


FIGURE 9.14: Rprop validation loss history for  $lr = 0.001$  and batch size = 512



## 9.3.8 SGD

	lr = 0.001	lr = 0.01	lr = 0.1
batch size = 32	(i) 3.015, 1.147, 1.962 (ii) 0.655, 1.059, 1.570 (iii) 3.014, 1.155, 1.927 (iv) 0.674, 1.191, 1.526	(i) 2.211, 1.274, 1.832 (ii) 0.466, 1.412, 3.488 (iii) 2.194, 1.279, 1.958 (iv) 0.482, 1.322, 3.953	(i) nan, nan, nan (ii) nan, nan, nan (iii) nan, nan, nan (iv) nan, nan, nan
batch size = 64	(i) 3.111, 1.758, 2.484 (ii) 0.478, 1.155, 1.540 (iii) 3.104, 1.698, 2.414 (iv) 0.505, 1.103, 1.422	(i) 2.588, 1.430, 1.812 (ii) 0.427, 2.033, 2.675 (iii) 2.591, 1.453, 3.319 (iv) 0.504, 1.366, 8.803	(i) 3.110, 1.681, 161.030 (ii) 0.493, 1.809, 14.209 (iii) 3.139, 1.725, 161.471 (iv) 0.475, 2.013, 14.249
batch size = 128	(i) 3.038, 2.478, 2.534 (ii) 0.360, 0.844, 1.753 (iii) 3.031, 2.465, 2.529 (iv) 0.362, 0.924, 1.427	<b>(i) 2.699, 1.192, 1.540</b> <b>(ii) 0.297, 0.672, 0.928</b> <b>(iii) 2.676, 1.190, 1.670</b> <b>(iv) 0.298, 0.737, 2.592</b>	(i) 3.164, 2.096, 25.201 (ii) 0.353, 1.619, 7.607 (iii) 3.286, 2.008, 21.618 (iv) 0.332, 1.521, 3.085
batch size = 256	(i) 3.493, 3.009, 4.328 (ii) 0.693, 1.382, 1.532 (iii) 11.136, 2.191, 40.877 (iv) 9.037, 14.214, 100.996	(i) 2.948, 1.773, 1.926 (ii) 0.237, 1.013, 1.110 (iii) 2.276, 2.303, 22.436 (iv) 4.277, 5.966, 18.091	(i) 1.308, 1.787, 1.820 (ii) 0.205, 0.883, 1.870 (iii) 0.264, 0.813, 6.977 (iv) 2.865, 2.159, 4.582
batch size = 512	(i) 3.735, 2.965, 9.591 (ii) 0.583, 1.357, 1.096 (iii) 0.861, 3.398, 35.620 (iv) 11.136, 2.191, 40.877	(i) 3.039, 2.232, 2.446 (ii) 0.193, 0.894, 1.315 (iii) 0.722, 4.408, 5.898 (iv) 2.276, 2.303, 22.436	(i) 2.574, 1.238, 1.197 (ii) 0.157, 0.425, 0.554 (iii) 0.216, 0.877, 3.567 (iv) 0.264, 0.813, 6.977

TABLE 9.10: Performance of SGD optimizer for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, under several learning rates and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE

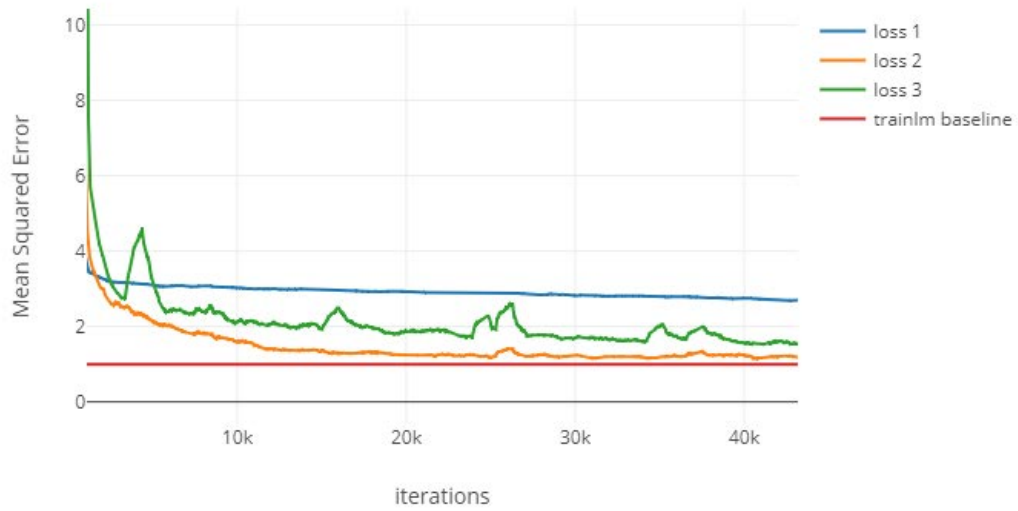


FIGURE 9.15: SGD training loss history for  $\text{lr} = 0.01$  and batch size = 128

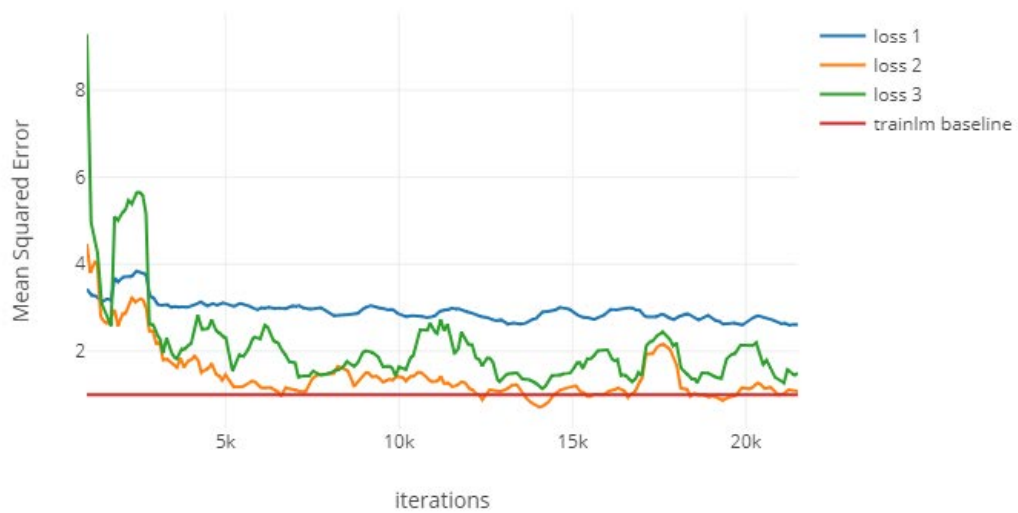


FIGURE 9.16: SGD validation loss history for  $\text{lr} = 0.01$  and batch size = 128

## 9.3.9 Levenberg-Marquardt

	$\lambda = 0.1$	$\lambda = 1$	$\lambda = 10$
batch size = 32	(i) 3.177, 1.589, 4.496 (ii) 0.687, 1.405, 6.114 (iii) 3.184, 1.675, 7.007 (iv) 0.813, 1.906, 31.66	(i) 1.690, 0.568, 0.974 (ii) 0.584, 1.072, 0.789 (iii) 1.694, 0.561, 1.010 (iv) 0.736, 0.924, 1.497	(i) 1.294, 1.262, 1.071 (ii) 0.522, 0.757, 1.018 (iii) 1.250, 1.214, 0.985 (iv) 0.489, 0.656, 0.783
batch size = 64	(i) 5.287, 4.494, 8.150 (ii) 3.286, 4.840, 8.174 (iii) 5.360, 4.492, 8.287 (iv) 3.363, 4.757, 8.673	<b>(i) 1.248, 0.874, 0.710</b> <b>(ii) 0.377, 0.600, 0.493</b> <b>(iii) 1.240, 0.883, 0.712</b> <b>(iv) 0.379, 0.656, 0.562</b>	(i) 1.287, 0.881, 0.814 (ii) 0.428, 0.663, 0.700 (iii) 1.311, 0.901, 0.805 (iv) 0.455, 0.748, 0.604
batch size = 128	(i) 6.546, 9.298, 19.549 (ii) 1.904, 3.230, 11.917 (iii) 6.614, 9.380, 20.408 (iv) 3.204, 5.012, 18.289	(i) 3.182, 1.925, 3.611 (ii) 0.353, 1.340, 3.918 (iii) 3.171, 1.926, 3.640 (iv) 0.391, 1.573, 3.638	(i) 3.263, 1.864, 2.991 (ii) 0.398, 1.271, 1.474 (iii) 3.258, 1.836, 3.060 (iv) 0.386, 1.458, 2.315
batch size = 256	(i) 3.198, 2.445, 146.922 (ii) 0.336, 1.700, 20.182 (iii) 0.551, 8.590, 34.347 (iv) 6.292, 53.393, 140.474	(i) 3.074, 2.577, 2.826 (ii) 0.275, 1.703, 0.908 (iii) 0.288, 1.392, 0.746 (iv) 3.031, 2.859, 3.654	(i) 1.152, 1.146, 1.102 (ii) 0.272, 0.539, 0.674 (iii) 0.294, 0.343, 0.807 (iv) 4.204, 1.840, 10.570
batch size = 512	(i) 3.147, 2.362, 3.260 (ii) 0.176, 0.984, 1.492 (iii) 0.256, 1.429, 11.044 (iv) 0.551, 8.590, 34.347	(i) 3.129, 2.052, 4.274 (ii) 0.183, 2.096, 2.777 (iii) 0.148, 1.631, 6.184 (iv) 0.288, 1.392, 0.746	(i) 1.144, 1.279, 1.233 (ii) 0.227, 1.910, 1.431 (iii) 0.116, 0.728, 1.299 (iv) 0.294, 0.343, 0.807

TABLE 9.11: Performance of Levenberg-Marquardt optimizer with slight bias towards a Gauss-Newton algorithm for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, under several  $\lambda$ s and batch sizes, and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE

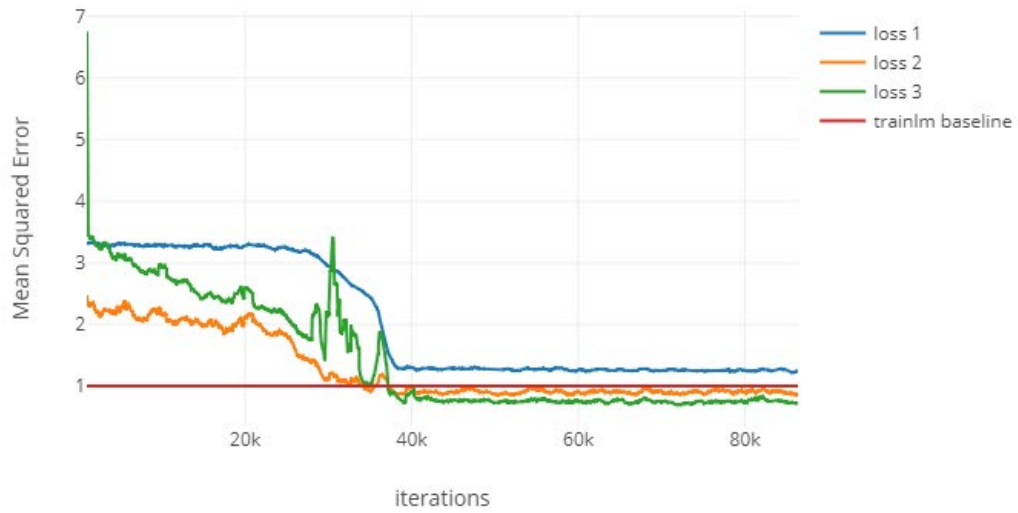


FIGURE 9.17: Levenberg-Marquardt training loss history for  $\lambda = 1$  and batch size = 64 (slight bias towards Gauss-Newton Method)

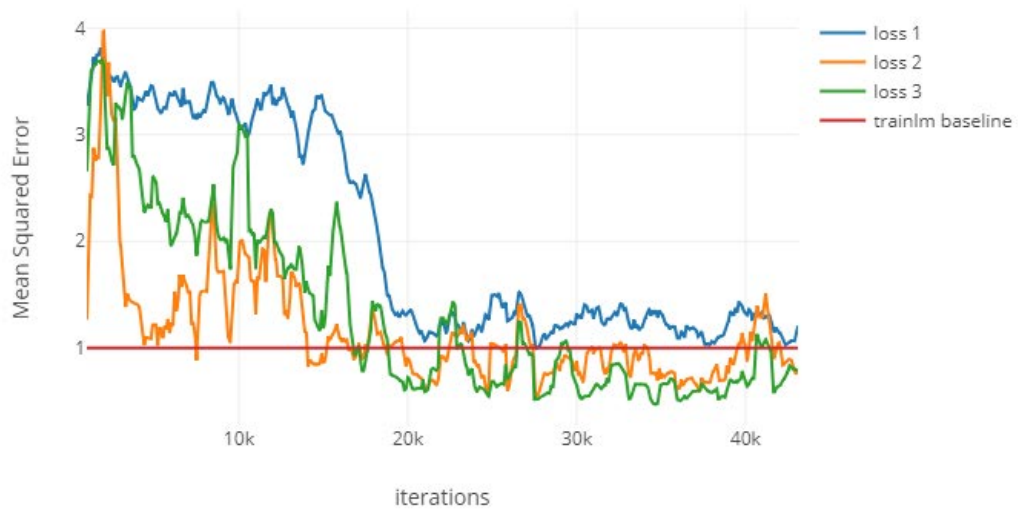


FIGURE 9.18: Levenberg-Marquardt validation loss history for  $\lambda = 1$  and batch size = 64 (slight bias towards Gauss-Newton Method)

## 9.4 Phase II - Addition of Convolutional Layers

Phase I clearly suggests that the following configurations show the best performance with no signs of overfitting:

- **Adam** optimizer with learning rate 0.0001 and batch size 32
- **Adamax** optimizer with learning rate 0.001 and batch size 64

Now these same configurations will be used with the three CNN architectures presented in Chapter 8.

	undilated	dilated (k=2)	dilated 2 (k=3)
Adam	(i) <b>0.575, 0.477, 0.641</b>	(i) 0.542, 0.514, 0.644	(i) 0.516, 0.398, 0.561
	(ii) <b>0.384, 0.545, 0.720</b>	(ii) 0.391, 1.035, 0.823	(ii) 0.388, 0.441, 0.861
	(iii) <b>0.592, 0.431, 0.553</b>	(iii) 0.498, 0.499, 0.745	(iii) 0.514, 0.438, 0.666
	(iv) <b>0.428, 0.551, 0.410</b>	(iv) 0.377, 0.563, 0.890	(iv) 0.391, 0.606, 0.864
Adamax	(i) 0.622, 0.841, 1.250	(i) 0.643, 0.590, 0.869	(i) 0.697, 0.858, 0.986
	(ii) 0.473, 2.171, 2.774	(ii) 0.355, 0.629, 1.032	(ii) 0.328, 1.224, 1.538
	(iii) 0.604, 0.556, 1.293	(iii) 0.643, 0.535, 0.737	(iii) 0.692, 0.566, 0.585
	(iv) 0.359, 0.420, 1.293	(iv) 0.418, 0.563, 1.124	(iv) 0.339, 0.593, 0.467

TABLE 9.12: Performance for parameters 1, 2 and 3, relative to Bernkopf's trainlm results, using the CNN architectures proposed in Chapter 8 and the optimal optimization algorithms and rounded to three decimal places: (i) training average MSE, (ii) training standard deviation of MSE, (iii) testing average MSE and (iv) testing standard deviation of MSE

From those results it is clear that the undilated network is the best performing of the three CNN architectures. However, it can also be seen that adding Convolutional Layers to the base architecture does not improve in any way its performance, neither with nor without dilation. In fact, even the undilated network yields a slightly worse mean squared error for the prediction of the first two parameters and a significantly worse error for the third, although still better than baseline.

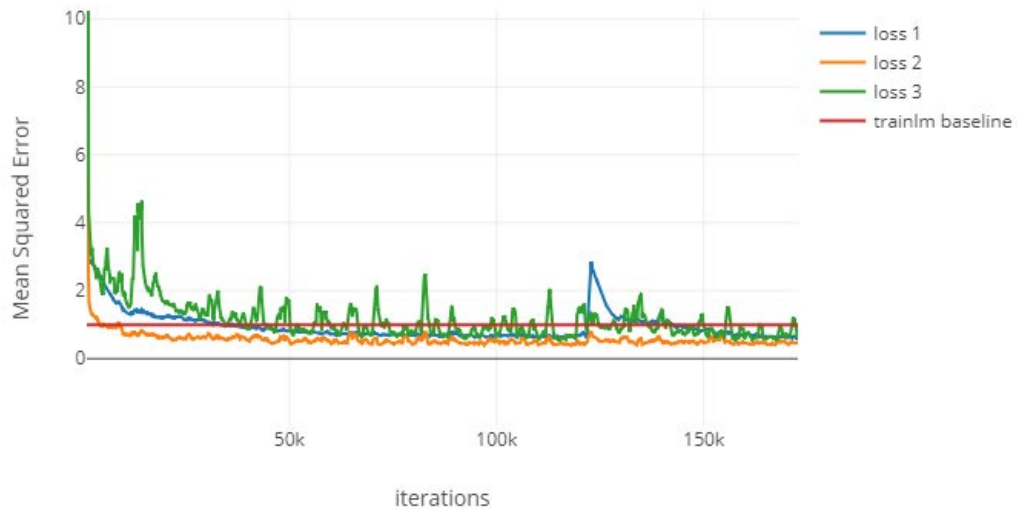


FIGURE 9.19: Adam training loss history for undilated CNN with  $l_r = 0.0001$  and batch size = 32

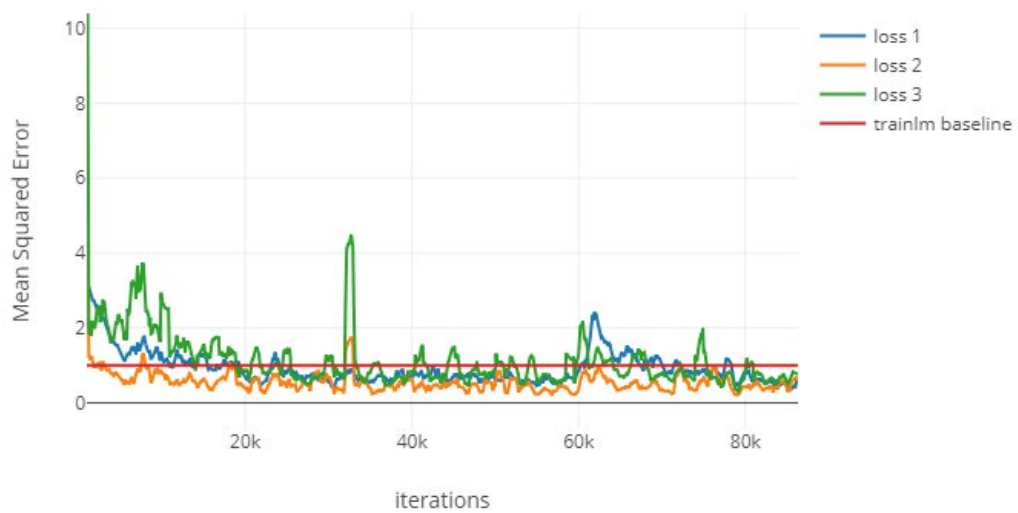


FIGURE 9.20: Adam validation loss history for undilated CNN with  $l_r = 0.0001$  and batch size = 32

## Chapter 10

# Conclusion

To ensure the reliability of the training data, several tests were performed on it and no significant abnormalities that could not be attributed to prediction issues were found. After examining it through the lens of a t-SNE dimensionality reduction, the dominance of global over local patterns could be determined, providing a first indication that Convolutional Neural Networks might not work as well as expected. This could be because local patterns are already "summarized" into the input features which basically consist of several mean values.

Consequently several architectures, including the base architecture proposed by Bernkopf (Master's Thesis under revision) and three convolutional architectures were trained using diverse optimization algorithms. The most successful configurations were

- Adam, learning rate = 0.0001, batch size = 32, and
- Adamax, learning rate = 0.001, batch size = 64.

Both yield an **improvement in performance of a factor of about 2 to 3.5** depending on the parameter, although Adamax produces slightly less noise.

The addition of convolutional layers not only did not improve performance, but had a rather detrimental effect on the learning process. Although the best performing configurations mentioned above still yielded results better than baseline for those architectures, these results were actually worse than without the additional layers.

Also, since the base architecture has under 5k parameters and the optimal convolutional architecture (undilated) surpasses the 20k parameters, the former is clearly recommendable.

Also, with an automated prediction of streaming particle track data in mind, the data formats `root`, `hdf5` and `asdf` were tested in terms of time and memory complexity of the importation process to assess an eventual bottleneck and `asdf` resulted to be the most efficient alternative, although an implementation might be impractical, given that `root` is the standard data format at CERN.

# Bibliography

- About CMS. <https://cms.cern/detector>. Accessed: 2019-01-15.
- Adam, Wolfgang et al. (2005a). "Reconstruction of electrons with the Gaussian-sum filter in the CMS Tracker at the LHC". In: *Journal of Physics G: Nuclear and Particle Physics* 31.31, pp. 9–20.
- (2005b). "Reconstruction of electrons with the Gaussian-sum filter in the CMS tracker at the LHC". In: *Journal of Physics G: Nuclear and Particle Physics* 31.9, N9.
- Ampazis, Nikolaos and Stavros J Perantonis (2000). "Levenberg-Marquardt algorithm with adaptive momentum for the efficient training of feedforward networks". In: *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks* 1, pp. 126–131.
- Arxiv-Insights (2017). 'How Neural Networks Learn' - Part I: Feature Visualization. Youtube. URL: <https://www.youtube.com/watch?v=McgxRxi2Jqo>.
- (2018). 'How neural networks learn' - Part II: Adversarial Examples. Youtube. URL: <https://www.youtube.com/watch?v=4rF0kpIOlCg&t=610s>.
- Bengio, Yoshua et al. (2009). "Learning deep architectures for AI". In: *Foundations and trends® in Machine Learning* 2.1, pp. 1–127.
- Bethe, Hans and Walter Heitler (1934). "On the stopping of fast particles and on the creation of positive electrons". In: *Proc. R. Soc. Lond. A* 146.856, pp. 83–112.
- Boltzmann, Ludwig (1877). "Über die Beziehung zwischen dem zweiten Hauptsatze des mechanischen Wärmetheorie und der Wahrscheinlichkeitsrechnung, respective den Sätzen über das Wärmegleichgewicht". In: *Kk Hof- und Staatsdruckerei*.
- Castelvecchi, Davide (2018). "Particle physicists turn to AI to cope with CERN's collision deluge". In: *Nature* 557, pp. 147–148.
- Cauchy, Augustin (1847). "Méthode générale pour la résolution des systemes d'équations simultanées". In: *Comptes Rendus Hebd. Scéances Acad. Sci. Paris* 1847, pp. 536–538.
- Dumoulin, Vincent and Francesco Visin (2016). "A guide to convolution arithmetic for deep learning". In:
- Floyd, Sally and Manfred Warmuth (1995). "Sample compression, learnability, and the Vapnik-Chervonenkis dimension". In: *Machine Learning* 21.3, 269–304.
- Frühwirth, R (2003). "A Gaussian-mixture approximation of the Bethe–Heitler model of electron energy loss by bremsstrahlung". In: *Computer Physics Communications* 154.2, pp. 131–142.
- Frühwirth, R and S Frühwirth-Schnatter (1998). "On the treatment of energy loss in track fitting". In: *Computer physics communications* 110.1-3, pp. 80–86.
- Frühwirth, Rudolf (1987). "Application of Kalman filtering to track and vertex fitting". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 262.2-3, pp. 444–450.
- Glorot, Xavier, Antoine Bordes, and Yoshua Bengio (2011). "Deep Sparse Rectifier Neural Networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323.
- Goodfellow, Ian et al. (2014). "Generative adversarial nets". In: *Neural Information Processing Systems*, pp. 2672–2680.
- Goodfellow, Ian et al. (2016). *Deep learning*. Vol. 1. MIT press Cambridge.



- Hilpisch, Yves J. (2018). *Python Tools & Skills*. GitHub. URL: <https://gist.github.com/yhilpisch/bda2479093216b299e59cf8c41bfa3e7>.
- Hinton, Geoffrey E, Simon Osindero, and Yee-Whye Teh (2006). "A fast learning algorithm for deep belief nets". In: *Neural computation* 18.7, pp. 1527–1554.
- Hinton, Geoffrey E and Sam T Roweis (2003). "Stochastic Neighbor Embedding". In: *Neural Information Processing Systems*, pp. 857–864.
- Hinton, Geoffrey E et al. (2012). "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580*, pp. 1–18.
- Hodges Jr, Joseph L and Erich L Lehmann (1963). "Estimates of Location Based on Rank Tests". In: *The Annals of Mathematical Statistics* 34.2, pp. 598–611.
- Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5, pp. 359–366.
- Hubel, David H and Torsten N Wiesel (1968). "Receptive fields and functional architecture of monkey striate cortex". In: *The Journal of physiology* 195.1, pp. 215–243.
- Kalman, Rudolph Emil (1960). "A New Approach to Linear Filtering and Prediction Problems". In: *Transactions of the ASME—Journal of Basic Engineering* 82.Series D, pp. 35–45.
- Kingma, Diederik P and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980*.
- Kullback, Solomon and Richard A Leibler (1951). "On information and sufficiency". In: *The annals of mathematical statistics* 22.1, pp. 79–86.
- LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep Learning Review". In: *Nature* 521.
- LeCun, Yann et al. (1990). "Handwritten digit recognition with a back-propagation network". In: *Advances in Neural Information Processing Systems*, pp. 396–404.
- LHC collides ions at new record energy. <https://home.cern/news/news/accelerators/lhc-collides-ions-new-record-energy>. Accessed: 2019-01-15.
- Lin, J. (1991). In: *IEEE Transactions on Information Theory* 37.1, pp. 145–151.
- Maaten, Laurens van der and Geoffrey Hinton (2008). "Visualizing data using t-SNE". In: *Journal of machine learning research* 9, pp. 2579–2605.
- McCulloch, Warren S and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133.
- McGee, Leonard A and Stanley F Schmidt (1985). "Discovery of the Kalman filter as a practical tool for aerospace and industry". In: p. 24.
- Mrazova, Iveta, Josef Pihera, and Jana Velemínska (2013). "Can N-dimensional convolutional neural networks distinguish men and women better than humans do?" In: *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pp. 1–8.
- Ng, Ritchie (2018). *Deep Learning Theory and Programming Tutorials*. Deep Learning Wizard. URL: <https://www.deeplearningwizard.com/>.
- Nilsson, Nils J (2009). *The quest for artificial intelligence*. Cambridge University Press.
- Raymond, M et al. (2000). "The CMS tracker APV25 0.25  $\mu$  m CMOS readout chip". In:
- Riedmiller, Martin and Heinrich Braun (1992). *RPROP - A Fast Adaptive Learning Algorithm*. Tech. rep. Proc. of ISICIS VII, Universitat.
- Robbins, Herbert and Sutton Monro (1951). "A stochastic approximation method". In: *The annals of mathematical statistics*, pp. 400–407.
- Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6, pp. 386–408.

- Rousseeuw, Peter J and Christophe Croux (1993). "Alternatives to the Median Absolute Deviation". In: *Journal of the American Statistical Association* 88.424, pp. 1273–1283.
- Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1986). "Learning representations by back-propagating errors". In: *nature* 323.6088, pp. 533–536.
- Saxe, Andrew Michael et al. (2018). "On the Information Bottleneck Theory of Deep Learning". In: *International Conference on Learning Representations*.
- Shannon, Claude Elwood (1948). "A mathematical theory of communication". In: *Bell system technical journal* 27.3, pp. 379–423.
- Shwartz-Ziv, Ravid and Naftali Tishby (2017a). *IDNNS*. GitHub. URL: <https://github.com/ravidziv/IDNNS>.
- (2017b). "Opening the black box of deep neural networks via information". In: Springenberg, Jost Tobias et al. (2014). "Striving for Simplicity: The All Convolutional Net". In:
- Tishby, Naftali, Fernando C Pereira, and William Bialek (1999). "The information bottleneck method". In: *Proceedings of the 37-th Annual Allerton Conference on Communication, Control and Computing*.
- Tishby, Naftali and Noga Zaslavsky (2015). "Deep learning and the information bottleneck principle". In: *IEEE*, pp. 1–5.
- Turing, Alan M. (1950). "Computing Machinery and Intelligence". In: *Mind* 59.236, pp. 433–460.
- Wattenberg, Martin, Fernanda Viégas, and Ian Johnson (2016). "How to Use t-SNE Effectively". In: *Distill*. DOI: [10.23915/distill.00002](https://doi.org/10.23915/distill.00002). URL: <http://distill.pub/2016/misread-tsne>.
- Yu, Fisher and Vladlen Koltun (2015). "Multi-scale context aggregation by dilated convolutions". In:
- Zeiler, Matthew D and Rob Fergus (2014). "Visualizing and Understanding Convolutional Networks". In: *European Conference on Computer Vision* 8689, pp. 818–833.