Universität Leipzig
Fakultät für Physik und Geowissenschaften

Event Classification with Convolutional Neural Networks for Diboson Channels in the ATLAS Experiment at the Large Hadron Collider

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science in Physik

Melanie Weber

geboren am 24. Juli 1992 in Burgstädt, Deutschland.

Kommitee:
Prof. Dr. Stefan Hollands (U Leipzig)
Prof. Dr. Michael Kobel (TU Dresden)

30. September 2016

Abstract

The experimental investigation and validation of predictions of the Standard Model of Particle Physics is of fundamental importance for the theory of matter and affects deep questions on our understanding of nature. A key challenge in experimental studies is the evaluation of large amounts of data and the discrimination of the signal of interest from a dominating background. This thesis develops a machine learning tool for event classification in diboson channels in the ATLAS experiment at the Large Hadron Collider at CERN. A Convolutional Neural Network is trained to discriminate electroweak $WZjj-EW$ signal from a $WZjj-QCD$ background - demonstrating the capabilities of the new method on a use case that is of major importance for testing the Standard Model in general and the electroweak theory in particular. The developed method uses supervised deep learning to analyze event data represented in a matrix format termed *event matrix* while operating on highly efficient data flow graphs. By implementing a CNN for event classification, a novel method for the analysis of event data is introduce and a potential setup as well as an overview of challenges for utilizing deep learning in experimental studies on collision data is given.

Zusammenfassung

Die Untersuchung von Voraussagen des Standard Models der Teilchenphysik ist von fundamentalem Interesse für das Verständnis von Materie und die Beantwortung grundlegender physikalischer Fragestellungen. Eine der größten Herausforderungen experimenteller Studien ist die Analyse sehr großer Datensätze und die Unterscheidung von Signal und Untergrund. Die vorliegende Arbeit entwickelt eine Machine-Learning Methode zur Klassifikation von Ereignissen mit Produktion von $W^{\pm}Z$-Bosonenpaaren im ATLAS-Experiment am Large Hadron Collider des CERN. Dazu wird ein neuronales Netz, das auf der Faltung von Daten mit einer Hierachie von Filtern beruht, zur Identifikation von elektroschwachem Signal trainiert. Ziel ist es, die Fähigkeiten solcher neuronaler Netze für derartige experimentelle Studien anhand eines Beispiels zu demonstrieren. Die entwickelte Methode benutzt überwachtes Lernen (engl. supervised learning) zur Analyse von Ereignisdatensätzen, die in einer neu entwickelten Darstellung (Eventmatrizen) verwendet werden. Dabei greift der Algorithmus auf hoch-effiziente Netzwerkstrukturen zur internen Übertragung und Speicherung von Daten zurück. Mit der Einführung der vorliegenden Methode soll eine neue computer-gestützte Analysetechnik aufgezeigt und Grundstrukturen und -herausforderungen solcher Techniken für zukünftige Studien von Kollisionsdaten diskutiert werden.

Acknowledgements

I would like to express my gratitude to my supervisor Michael Kobel for giving me the opportunity to work in his research group and for his support and engagement through the process of this thesis.

Special thanks to Felix Socher and Carsten Bittrich for their support and advise on physical and technical matters that contributed significantly to this work. I would also like to thank Sarah Krebs for sharing data for a performance comparison with BDTs and Franziska Iltzsche for helpful comments on the draft of this thesis. Furthermore, my thanks to all members of the VBS group for helpful discussions and for giving insights into their exciting ongoing research.

Contents

Introduction

With an international effort to investigate the building blocks of matter over the last decades, the field of elementary particle physics has seen major advances. In 2012, the ATLAS and CMS collaborations at CERN in Geneva, Switzerland, announced the discovery of the Higgs boson [1] as the last missing piece of the Standard Model. More recently, in February 2016, the LIGO collaboration reported the detection of gravitational waves from colliding black holes [2] – a theoretically proposed effect that has remained open to experimental validation for more than a century. But despite important discoveries, that answered some of the biggest open questions, much is still unknown providing challenges for decades to come. One field of major interest to a fundamental understanding of the theory of matter are electroweak interactions mediated by heavy bosonic particles, the Z and W bosons.

Those interactions are by its very name *weak* in comparison to stronger chromodynamic interactions, which makes their observation an experimental challenge despite the broad occurrence across the spectrum of elementary particles. Attempts to observe weak interactions (including the here considered *vector boson scattering*) have utilized collisions of hadronic particles, for instance in experiments conducted at the Large Hadron Collider (LHC) at CERN. A major challenge of such experiments is the discrimination of weak signal 'overshadowed' by a strong background creating a need for sophisticated data analysis tools. This thesis develops a machine learning method for discerning events in diboson channels originating from purely electroweak processes from those containing strong contributions from chromodynamic interactions. The method builds on recently developed deep learning methods that have been applied very successfully in pattern recognition and image classification.

While the setup of this novel tool is broadly applicable for a wide range of event classification tasks, this thesis focuses on a specific use case: The discrimination of WZjj-EW and WZjj-QCD. A key process in understanding weak interactions is $W^{\pm}Z$ production in proton-proton (pp) collisions, since it allows for investigating the scattering of massive vector bosons. This process is closely connected to the study of the electroweak symmetry breaking – an essential element of the proposed Standard Model of particle physics. Among the known gauge boson coupling processes, $W^{\pm}Z$ diboson production is well suited as a use case for the tool: It has a larger cross section - i.e. a higher probability of occurrence - than ZZ processes

and is at the same time easier to separate from backgrounds than W^+W^- processes since it requires three charged leptons, two of them originating from a Z boson [3]. Another possible use case would be the W^+W^+ process, but it is overlayed by and hard to discriminate from $W^\pm Z$. Therefore, $W^\pm Z$ is a good compromise between a high cross section and a well measurable final state.

The interacting particles are characterized by a 'signature' consisting of a set of (independent) parameters that can be recognized as recurring patterns in the event data. The goal is to identify these patterns and utilize this information to classify the event data and therefore tag the interesting weak interactions in a stream of events. For this, a supervised deep learning method for image recognition is adapted that is operated by a Convolutional Neuronal Network (CNN). CNNs are a special form of multilayer neuronal networks that identify and encode patterns on various scales based on local correlations in a feature space. This idea is transferred to the signatures of particles in each event by learning the experimentally measured features of objects detected in pp-collisions. To enable the CNN to read the information, a data structure (termed *event matrix*) is developed that is compatible with the setup of the CNN.

Building on work by Y. LeCun et al. [4, 5], this thesis develops a CNN implementation for event classification in the Python-based deep learning language *Theano*. The method adapts commonly used statistical measures to allow for comparison between the novel approach and conventional analysis techniques used in the field. While only demonstrated for a small use case in this thesis, the method shows promising results and gives hope for better performance in future studies with larger samples of training data.

Chapter 1

Theoretical Foundation

This thesis is motivated by the challenge of observing and analyzing rare interaction events in collision experiments. A key task for collecting the relevant event data is the discrimination of signal from background. In this chapter, we will give an overview of both the physics underlying the targeted process and the theoretical foundations of the CNN that forms the basis of the introduced machine learning tool. The theoretical introduction for physical aspects is based on standard works [6, 7, 8], for references on CNNs and deep learning see Y. LeCun et al. [4, 5] as well as Lisa Lab's deep learning tutorial [9].

1.1 Physical Theory

Standard Model

The fundamental theory of particle physics, that aims to describe the 'building blocks' of matter and their interactions at the smallest scale is the *Standard Model (SM)*. It subdivides the set of 'building blocks', called elementary particles, in two groups: Fermions and bosons. Fermions are characterized by half-integer spin and the fact that they follow Fermi-Dirac statistics. Depending on the interactions they take part in, they are further subdivided into leptons and quarks. Table 1.1 summarizes the classification of fermions and lists relevant properties. Interactions between fermions are mediated by the exchange of bosons. Unlike fermions, bosons have integer spin and follow Bose-Einstein statistics.

Interactions between particles are characterized by the four physical forces, namely *electromagnetic*, *strong*, *weak* and *gravitational* interactions (see Tab. 1.2). For this work, only the electro-magnetic, strong and weak forces will be relevant. The theory of *Quantum Electrodynamics (QED)* describes electromagnetic interactions mediated by photons and is the oldest and best studied of the dynamical theories [6]. Strong interactions between quarks are mediated by gluons and subject to *Quantum Chormodynamics (QCD)*. They are the underlying forces of the formation of *hadrons* from quarks, a group of composed particles that includes protons and neutrons and that are of fundamental importance for physics on the atomic scale. The third, and for this work most important quantum field theory

is that of *weak interactions* mediated by $W^{\pm}$ and Z bosons and the corresponding *Quantum Flavourdynamics*.

More formally, interactions between particles involve the exchange of spin-1 gauge bosons characterized by local gauge principles. The gauge symmetry underlying the SM is

$$U(1)_Y \times SU(2)_L \times SU(3) \tag{1.1}$$

Class	**Generation**	**Particle**	**Mass** (in GeV*)	**Charge** (in proton charges)
Lepton	1st	e	0.0005	-1
		ν_e	$< 10^{-9}$	0
	2nd	μ	0.106	-1
		ν_μ	$< 10^{-9}$	0
	3rd	τ	1.78	-1
		ν_τ	$< 10^{-9}$	0
Quarks	1st	d	0.003	$-\frac{1}{3}$
		u	0.005	$\frac{2}{3}$
	2nd	s	0.1	$-\frac{1}{3}$
		c	1.3	$\frac{2}{3}$
	3rd	b	4.5	$-\frac{1}{3}$
		t	174	$\frac{2}{3}$

Table 1.1: Overview of leptons and quarks and their classification [6, 7]. *: The mass is given in GeV/c^2 with $c = 1$.

Force	**Theory**	**Mediator**
Strong	Chromodynamics (QCD)	Gluon
Electromagnetic	Electrodynamics (QED)	Photon
Weak	Flavourdynamics, GWS	$W^{\pm}$, Z

Table 1.2: Overview of the three physical forces relevant for this work and their corresponding mediators [6].

Electroweak Theory

The *SM* can be formulated as a quantum field theory where particles and their interactions are described with a Lagrangian that reflects the local gauge symmetries:

$$\mathcal{L}_{SM} = \mathcal{L}_{YM} + \mathcal{L}_H + \mathcal{L}_{ferm} + \mathcal{L}_{Yuk} \, . \tag{1.2}$$

$\mathcal{L}_{YM}$ characterizes gauge bosons and their interactions as given by the *Yang-Mills theory*. It requires, initially counter intuitive, massless gauge bosons – a fact that

is explained through the Higgs mechanism ($\mathcal{L}_H$) that will be reviewed in the next section. The two remaining terms, $\mathcal{L}_{ferm}$ and $\mathcal{L}_{Yuk}$ describe matter fields and the respective interactions with mediators. In particular, $\mathcal{L}_{ferm}$ characterizes the interaction of matter fields with gauge bosons and $\mathcal{L}_{Yuk}$ interactions with the Higgs boson. The Yang-Mills Lagrangian $\mathcal{L}_{YM}$ respects the previously mentioned local gauge symmetries:

$$\mathcal{L}_{YM} = -\frac{1}{4} W^i_{\mu\nu} W^{i,\mu\nu} - \frac{1}{4} B_{\mu\nu} B^{\mu\nu} - \frac{1}{4} G^a_{\mu\nu} G^{a,\mu\nu} \tag{1.3}$$

with

$$W^i_{\mu\nu} = \partial_\mu W^i_\nu - \partial_\nu W^i_\mu - g\epsilon^{ijk} W^j_\mu W^k_\nu \quad (i,j,k = 1,2,3)\ , \tag{1.4}$$

$$B_{\mu\nu} = \partial_\mu B_\nu - \partial_\nu B_\mu\ , \tag{1.5}$$

$$G^a_{\mu\nu} = \partial_\mu G^a_\nu - \partial_\nu G^a_\mu - g_s f^{abc} G^b_\mu G^c_\nu \quad (a,b,c = 1,...,8)\ ; \tag{1.6}$$

with structural constants ϵ^{ijk} and f^{abc}. In particular, this gives $U(1)$ symmetry for QED (1.5) and $SU(3)$ for QCD (1.6). Weak interactions are characterized by invariance of phase transitions under $SU(2)$ (1.4). $U(1)$ is an abelian symmetry that ensures the free propagation of the gauge field. $SU(2)$ and $SU(3)$ are non-abelian symmetries that give rise to interactions among the gauge fields resulting in constraints on the field propagation. This is of major importance for testing the SM: Investigations of multi-boson interactions (i.e. *vector boson scattering*) can be utilized to test the proposed symmetry structure and therefore the claims of the SM.

Quantum electrodynamics arise from interactions of fermions with the photon field A_μ. Besides the photon field, there is a second neutral gauge field, Z_μ, associated with the Z boson that mediates electroweak interaction. Both can be expressed as linear combination of W^3_μ and B_μ:

$$\begin{pmatrix} Z_\mu \\ A_\mu \end{pmatrix} = \begin{pmatrix} \cos\theta_W & -\sin\theta_W \\ \sin\theta_W & \cos\theta_W \end{pmatrix} = \begin{pmatrix} W^3_\mu \\ B_\mu \end{pmatrix} \tag{1.7}$$

The coefficients can be written in terms of a rotation with the *Weinberg angle* θ_W and coupling constants g and g':

$$\cos\theta_W = \sqrt{1 - \sin\theta^2_W} = \frac{g}{\sqrt{g^2 + g'^2}} \tag{1.8}$$

$$e = \frac{gg'}{\sqrt{g^2 + g'^2}} \tag{1.9}$$

The field associated with charged $W^\pm_\mu$ can be written as linear combination

$$W^\pm_\mu = \frac{1}{\sqrt{2}} \left(W^1_\mu \mp i W^2_\mu \right) \tag{1.10}$$

Electroweak Symmetry Breaking

The second component $\mathcal{L}_{\text{Higgs}}$ describes the *Higgs mechanism*, a spontaneous electroweak symmetry breaking due to non-vanishing vector bosons and fermion masses. Theoretically predicted by P. Higgs [10], F. Englert and R. Brout [11], and G. S. Guralnik, C. R. Hagen and T. W. B. Kibble [12] the effect was experimentally confirmed in 2012. The mechanism proposes a symmetric Brout-Englert-Higgs field that requires non-vanishing masses for bosons and a gain of mass for fermions when interacting with the field. It was indirectly observed through the, then hypothetical, Higgs boson that characterizes excitations of the Higgs field.

Vector Boson Scattering

The interaction of gauge bosons followed by their decay is called *Vector Boson Scattering* (or short *VBS*). Such processes occur between all vector bosons, however, this thesis focuses on diboson channels – scattering of W and Z bosons associated with two jets (symbolic: $WZjj$), see Fig 1.

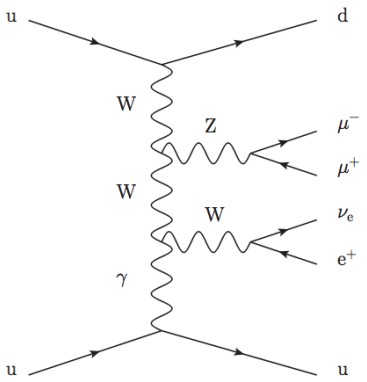

Figure 1: Exemplary Feynman diagram of vector boson scattering. From [13].

There are multiple processes leading to a $WZjj$ final state making an efficient discrimination between the event of interest and the remaining background a key task in experimental studies as conducted at ATLAS. This thesis focuses on the separation of $WZjj$ states that result from electroweak interactions ($WZjj-EW$) against QCD-background ($WZjj-QCD$). The key issue is that QCD has a larger total cross section causing a dominance of $WZjj-QCD$ in a given event sample.

1.2 The ATLAS Experiment

The experimental data for which the analysis tools in this thesis are designed, consist of measurements of particle collisions at the ATLAS experiment at the Large Hadron Collider (LHC) at CERN in Geneva, Switzerland. The present

section summarizes technical details of the ATLAS detector specifically, and the LHC in general, following the ATLAS collaboration [14].

Large Hadron Collider

Completed in 2008, the *Large Hadron Collider* (short *LHC*) at the European research center for particle physics CERN (Conseil Européen pour la Recherche Nucléaire) is currently the most advanced particle collider worldwide. Consisting of superconducting electromagnets arranged in a ring 27 kilometers in diameter, it accelerates particles to near speed of light to study the features and interactions of elementary particles in collision experiments.

Two high energy beams are directed to collide at four locations in the ring staffed with detectors, including ATLAS (**A** **T**oriodial **L**HC **A**pparatu**S**) and CMS (**C**ompact **M**uon **S**olenoid). The beams consist of up to 10^{11} protons each and their collision results in about 40 million individual collision events per second with a design luminosity of $10^{34}\text{cm}^{-2}\text{s}^{-1}$. Both experiments have led to revolutionary insights, most notably the detection of a Higgs boson that validated the theoretically predicted Higgs mechanism in 2012 [1].

ATLAS experiment

The ATLAS detector consists of a set of nested cylindrical detectors that are arranged with azimutal symmetry and are centered at the interaction point as shown in figure Fig.2. At its core is the *inner detector*, a high resolution semiconductor pixel detector that is immersed by the 2 Tesla field of the surrounding superconducting solenoid. It records the traces of collision events. Jets, as well as electrons and photons are detected in the outer *calorimeter* consisting of an electromagnetic and a hadronic calorimeter. For a precise measurement of muons, a *muon spectrometer* surrounds the calorimeter. The spectrometer consists of trigger chambers and precision chambers. The trigger chambers with timing resolution of 1.5-4 ns achieve high time resolution, whereas the precision chambers have a high local resolution.

Unfiltered, each run of the system (with a collision rate of 1 GHz) would produce enormous amounts of data far beyond the capacity of current computational data analysis. A triggering system controls the processing of event data, filtering out large parts of the recorded data and reducing the rate of reported events to 200 Hz.

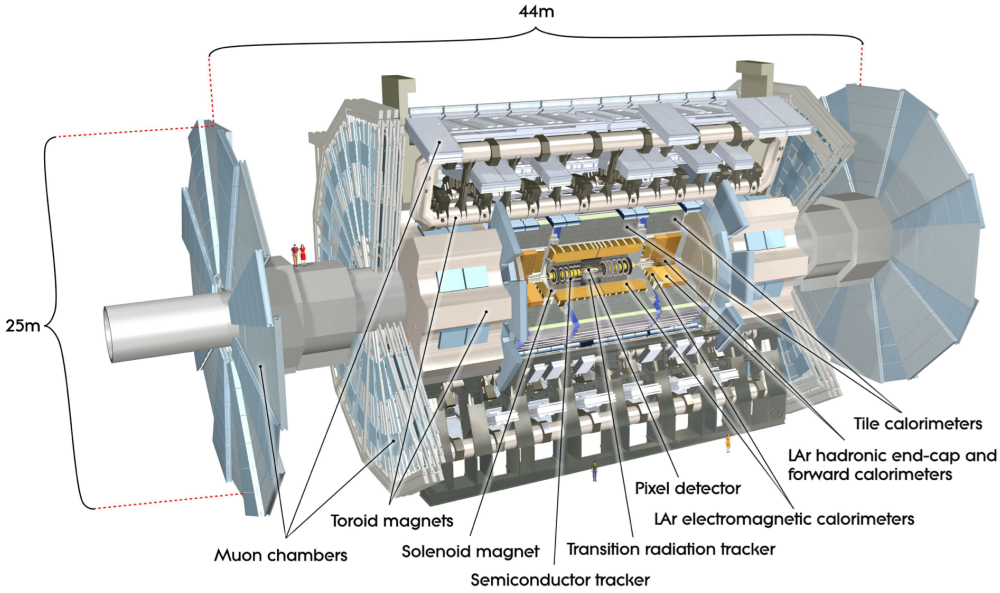

Figure 2: Schematic representation of the structure of the ATLAS detector, from [14]. The figure shows the nested specialized detectors that form ATLAS as described in the text.

The cylindrical structure of the detector is also reflected in the processed event data and forms the basis of the event representations introduced in the next chapter. For each event a location in cylindrical coordinates is recorded. The origin of the corresponding coordinate system is the interaction point. The direction of the beam defines the z-axis and the $x-y$-plane lies transverse to it. In particular, the x-axis forms a line from the interaction point to the center of the LHC ring with the y-axis perpendicular to it, pointing upwards. In practice, mostly cylindrical coordinates are used as opposed to the cartesian ones to account for the structure of the detector. For this, the polar angle θ (angle between the direction of flight of the particle and the beam pipe) and the azimutal angle ϕ (angle around beam axis) are recorded. Instead of θ one typically uses the *pseudorapidity*

$$\eta = -\ln\left(\tan\frac{\theta}{2}\right) , \tag{1.11}$$

or for massive objects like jets

$$y = \frac{1}{2}\ln\left(\frac{E+p_z}{E-p_z}\right) \tag{1.12}$$

with momentum p_z (for high energies the pseudo-rapidity and the rapidity are approximately equal). This gives a representation of (p_z, η, ϕ) for each object. In the cylindrical system, the inner detector measures a range of $|\eta| < 2.5$, the calorimeter $|\eta| < 5$ and the high-precision chambers of the muon spectrometer $|\eta| < 2.7$.

1.3 Deep Learning with Convolutional Neural Networks (CNNs)

Deep Learning is a hierarchical approach to machine learning that is usually based on a neural network. In a first construction step (*training*), representations of a system are *learned* on increasing levels of abstraction using a given set of data [4]. The trained network is able to classify and analyze previously unseen data through retrieval of learnt information and recognition of patterns, achieved using a backpropagation algorithm. Deep Learning methods are superior to conventional machine learning methods in their ability to capture information on different scales of detail.

One of the most successful Deep Learning approach of the last years are *Convolutional Neural Networks* (short: *CNNs*). The inspiration for CNNs lies in the structure of the visual cortex of primates as described in, e.g., Hubel and Wiesel [15]. They report a hierarchical structure of cells in the striate cortex of macaques, arranged in horizontal *cortical layers*. The low layers contain *simple* cells with a small receptive field but high sensitivity to stimuli. Moving vertically upwards, the cells are more coarse-grain consisting of more complex cells with a larger receptive field. The upper layers are built of *hypercomplex* cells with a large receptive field but low sensitivity. This structure allows for a sequential processing of visual information in vertical direction, where hyper complex cells capture coarse features and simple cells the finer details.

Following the natural model of the visual cortex, Y. LeCun et al. constructed a hierarchical multilayer neural network that processes complex information by taking into account varying levels of detail [5]. CNNs form a special class of *Multilayer Perceptrons* (short *MLP*) that will be introduced in what follows.

Multilayer Perceptrons

The basic concept of MLPs is the transformation of input data in a higher dimensional feature space where it is linearly separable. This is typically achieved through multiple transformation steps (represented by multiple neural layers). For the sake of notation, the example of a one-layer network, i.e. *input* $\rightarrow$ *hidden layer* $\rightarrow$ *output* is considered.

The core of the MLP is a non-linear transformation function

$$f : R^D \rightarrow R^L \tag{1.13}$$

where D is the dimension of the input data and L the dimension of the output. f is chosen as

$$f(x) = G\left[b^{(2)} + W^{(2)}\left(s\left(b^{(1)} + W^{(1)}x\right)\right)\right] \tag{1.14}$$

with

$$h(x) = s\left(b^{(1)} + W^{(1)}x\right) \tag{1.15}$$

representing the hidden layer. W denotes the connectivity matrix

$$W = (w_{ij}) \tag{1.16}$$

that specifies the strength (called *weight*) of the connection between neurons or cells i and j. b is a bias typically containing previously known information. G and s are sigmodial activation functions that filter insignificant information by only activating the network above a certain threshold, typically chosen as

$$\text{sig}_1(a) = \frac{1}{1 + e^{-a}} \tag{1.17}$$

or

$$\text{sig}_2(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \tag{1.18}$$

where a is the value of the respective signal.

For training, a *Stochastic Gradient Decent* (short *SGD*) is used. SGD denotes an iterative procedure to find an optimal choice of the weight matrix W that achieves minimal test error over a randomly chosen subsample of the full training set called *minibatch*.

The actual classification task is performed using *logistic regression*. For this, the input is projected onto a higher dimensional feature space (using 1.14) where each class is represented as a hyperplane. The distance between the projected input and the hyperplanes determines the probability that a data point belongs to a certain class:

$$P\left(Y = i|x, W, b\right) = \text{sig}\left(Wx + b\right) = \frac{e^{W_i x + b_i}}{\sum_j e^{W_j x + b_j}} \tag{1.19}$$

where *sig* is a sigmodial function as specified above. Based on the calculated probabilities a class is assigned to each data point using a classification function

$$\text{pred} = \text{argmax}_i P\left(Y = i|x, W, b\right) \tag{1.20}$$

The SGD is implemented to minimize the error function

$$L(W, b, \mathcal{D}) = \sum_{i=0}^{\sharp D} \log\left(P\left(Y = i|x, W, b\right)\right) \tag{1.21}$$

over a data set $\mathcal{D}$.

Convolutional Neural Networks

CNNs are a special class of MLPs with a hierarchical structure that is inspired by the visual cortex of mammals as discussed in the introduction of this chapter. Their (hidden) layers consist of filters whose size (receptive field) increases with the index of the layer. At the same time, the level of detail (sensitivity) decreases, allowing for capturing small details with the lower layers and the "big picture" with the upper layers (see Fig. 2).

Due to their very construction, the filters capture local spacial correlations while assuming translational invariance, i.e. certain patterns can occur in principle at any time and any place in the feature space. The spacial correlations imply connectivity patterns between groups of neurons in adjacent layers forming the filters in a supervised training procedure.

Let $h^k,\ k = 1, ..., n$ denote the filters in a layer. Each filter is convoluted with the data points $x \in \mathcal{D}$ and the resulting signal transformed with a sigmodial function

$$h_{ij}^k = \tanh\left[\left(W^k * x\right)_{ij} + b_k\right] . \tag{1.22}$$

The sigmodial function acts as *detection function* as it vanishes for low signal indicating a low probability for the feature represented by the filter at x. Convolution, denoted with $*$ is adapted from classical signal processing, where in one dimension

$$o(n) = f(n) * g(n) = \sum_{i=-\infty}^{\infty} f(i) \cdot g(n-i) = \sum_{i=-\infty}^{\infty} f(n-i) \cdot g(i) \tag{1.23}$$

Since each filter of the horizontal layers is internally stored as a two-dimensional matrix structure, one needs to perform a two-dimensional convolution, i.e.

$$o(n,m) = f(n,m) * g(n,m) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(i,j) \cdot g(m-i, n-j) \tag{1.24}$$

Each layer has multiple feature maps, their number and size depending on the dimension and structure of the given data. The choice of the filter parameters is crucial for the performance of the CNN and is very sensitive to the features of the underlying data - as one shall see in the next section.

Chapter 2

Event Classification with CNNs

2.1 Representation of Events: Event Matrices

The development of Deep Learning techniques has lead to major advances in data analysis due to its superior abilities in recognizing patterns and retrieving information on different scales. This is in stark contrast to conventional Machine Learning methods which are limited in their ability to view data in multiple dimensions. Starting from this finding, one could hope that these special abilities of deep learning methods are useful for the discrimination of signal from background in event data by recognizing objects on various scales and making full use of special features in the data. In the present thesis, this hope is investigated for the case of separating rare electroweak interactions from a QCD background.

A major challenge resides in the fact, that CNNs are constructed with a strong resemblance of the visual cortex and therefore designed to process *visual information*, not numerical data. Therefore, one needs to find a representation of event data that can be processed similar to a visual input. Visual input in the form of images is internally (i.e., within the network) stored as 2D grids of pixels represented by matrices. Hence, one needs to find a matrix representation for the event data that can be - similar to a visual input - stored as a 2D array.

For this, three possible formats of *event matrices* are proposed that represent event data in a form processable by CNNs:

1. **Cylindrical representation**
   Event data contains spacial information on detected objects in ϕ-η space specifying cylindrical coordinates that resemble the geometry of the detector.

   The indices of each object entry are determined through a scaling scheme: Due to the geometrical features of the inner detector, one has

   $$\phi \in [-3.2, 3.2] \tag{2.1}$$

   $$\eta \in [-5, 5] \tag{2.2}$$

   To define a two-dimensional grid that can be directly translated into a matrix, step sizes $\Delta\phi$ and $\Delta\eta$ are chosen. For the analysis reported in this

thesis, the default choice are $\Delta\phi = 0.1$ and $\Delta\eta = 0.1$. ϕ is scaled onto the grid using the transformation

$$\hat{\phi} = \lceil \frac{|\phi|}{\Delta\phi} \rceil \tag{2.3}$$

$$\phi_{scale} = \begin{cases} \lceil \frac{\max \phi}{\Delta\phi} - \hat{\phi} \rceil, & \phi > 0 \\ \lfloor \frac{\max \phi}{\Delta\phi} + \hat{\phi} \rfloor, & \text{else.} \end{cases} \tag{2.4}$$

Similarly, η is scaled by

$$\hat{\eta} = \lceil \frac{|\eta|}{\Delta\eta} \rceil \tag{2.5}$$

$$\eta_{scale} = \begin{cases} \lceil \frac{\max \eta}{\Delta\eta} - \hat{\eta} \rceil, & \eta > 0 \\ \lfloor \frac{\max \eta}{\Delta\eta} + \hat{\eta} \rfloor, & \text{else.} \end{cases} \tag{2.6}$$

The value of the entry is determined by the transverse momentum $p_T \in [15, 300]\ MeV$ scaled onto a $[0, 255]$ scale (resembling the values of pixels), where $\min(p_T) \mapsto 0$ and $\max(p_T) \mapsto 255$.

2. **Cartesian representation**
   Alternatively, one can represent the events in x-z space. For this, each object is projected onto the x-z plane and their cartesian coordinates determine the location of the object in the matrix, obtained directly from the four-vector. The value of the respective entry equals again the transverse momentum p_T scaled onto a $[0, 255]$ scale (resembling the values of pixels). A quadratic x-z plane spanned by a grid with $x, z \in [0, 500]$ (interval chosen according to parameter range in the use case) and $\Delta x = \Delta z = 10$ is assumed that can be translated into a matrix. The indices of each object entry are determined through a scaling scheme:

$$\hat{x} = \lceil \frac{|x|}{\Delta x} \rceil \tag{2.7}$$

$$x_{scale} = \begin{cases} \lceil \frac{\max x}{\Delta x} - \hat{x} \rceil, & x > 0 \\ \lfloor \frac{\max x}{\Delta x} + \hat{x} \rfloor, & \text{else.} \end{cases} \tag{2.8}$$

   and analogously

$$\hat{z} = \lceil \frac{|z|}{\Delta z} \rceil \tag{2.9}$$

$$z_{scale} = \begin{cases} \lceil \frac{\max z}{\Delta z} - \hat{z} \rceil, & z > 0 \\ \lfloor \frac{\max z}{\Delta z} + \hat{z} \rfloor, & \text{else.} \end{cases} \tag{2.10}$$

3. **Feature representation**
   A different approach to representing object information would be to construct a feature matrix: Columns would represent features and rows the set of all

detected objects in one event, the entries specifying the respective feature values using a color code. While this representation looked promising at first, its performance is poor due to a lack of translational invariance as further discussed in Appendix A. This affirms the importance of translational invariance for the performance of CNNs.

In the proposed formats, each event is represented by one *event matrix*. The coordinates in (η, ϕ)- or (x, z)-space directly translate into column and row numbers as given by the transformation functions above. The entry represents the transverse momentum p_T of the object. To translate the matrix into an image internally, the momentum must be transformed onto a $[0, 255]$ integer scale. Additionally, information on the type of object is gained from the layer of the detector in which it was recorded (see previous chapter). This information can be encoded in a symbolic representation:

1. ◯: a circle symbolizing a cone for jets
2. ×: a cross mark for leptons
3. •: a single dot or pixel for MET

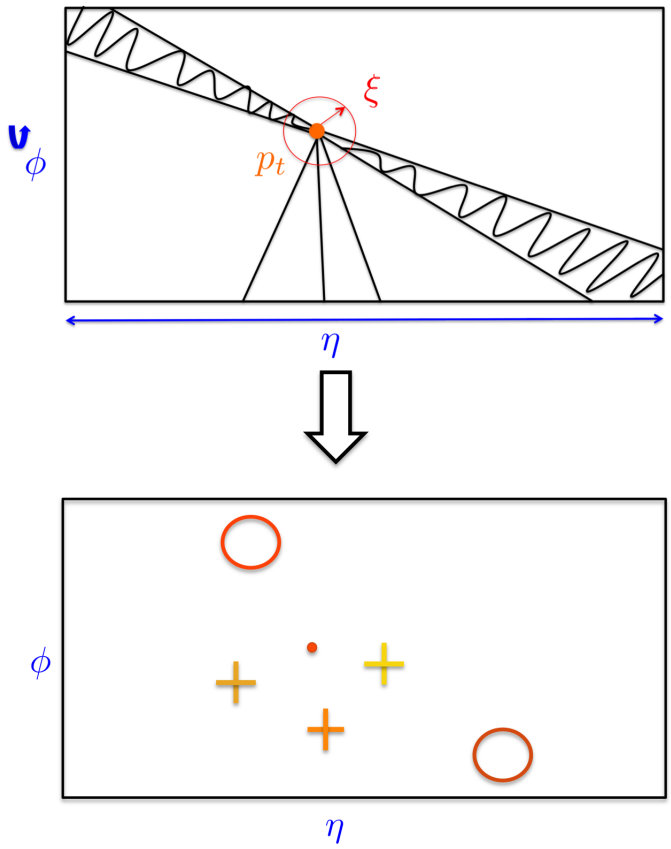

Figure 1: Event matrix with cylindrical coordinates built of triples (η, ϕ, p_T) per detected object. The figure shows schematically, how the information of the detected objects is stored.

2.2 $W^{\pm}Z$ Process and Event Data

As discussed earlier, $W^{\pm}Z$ diboson production in *pp*-collisions is chosen as a 'test case' for the classification tool since this process is of major interest for experimental testing of predictions of the Standard Model; in particular the electroweak theory. Data reported in [3] is used, which was obtained through simulation of *pp*-collisions at the ATLAS experiment at CERN's LHC with a center of mass $\sqrt{s} = 8\ TeV$ and integrated luminosity of 20.3 fb^{-1}. The kinematics of the gauge bosons are inferred from the kinematics of the leptons into which they decay. This information is used to construct event matrices as specified above: The transverse momentum p_T specifies the colorcode of the entry and the location of detection (in (x, z)- or (η, ϕ)-space, depending on the format) its indices. Among the reported data, the present study focuses on $W^{\pm}Zjj$ events with at least two jets. The samples are generated through Monte Carlo simulation and overlaid with additional *pp*-interactions as reported in [3].

The implemented CNNs are trained to perform a classification task of separating electroweak $WZjj - EW$ signal from a $WZjj - QCD$ background. Signal consists of dibosons and associated leptons as well as two jets. A missing transverse energy E_T^{miss} is included to account for neutrinos and is assumed to have no other source than neutrinos. The invariant mass of the neutrino and the charged lepton associated with the Z boson decay is assumed to equal the W boson mass [13]. The interference between $WZjj - EW$ and $WZjj - QCD$ is very small and can be neglected to a good approximation.

The raw data set includes $\sim 7,000$ signal and $\sim 12,000$ background events and is already filtered for events with at least three leptons (muons or electrons) detected. Additional filtering is performed during the construction of the event matrices as described below.

2.3 Implementation

Python Library *Theano*

Theano is a Python library developed and maintained by the *Theano Development Team* R. Al-Rfou et al. since 2008 [16]. The library is specialized for the evaluation of mathematical expressions on multi-dimensional arrays and allows for resourceful computations on large data sets. Its high efficiency is based on clever storage of partial results and optimal usage of available working memory. Additionally, it utilizes the typical sparsity of large real-world data and processes large data sets in batches (in *theano* achieved with so-called shared variables).

Theano adapts and extends the Python library *NumPy* by introducing additional complex data types on top of those already implemented by *NumPy*. They include theano vectors and matrices which substitute numpy arrays and theano tensors that operate on the matrix objects. A special functional structure, *theano.function*, is used to define routines on theano objects. Internally, compu-

tations are processed as data flow graphs: Functions and computational routines are modeled in directed bi-partite graphs, where nodes represent mathematical operators that can receive information, i.e. a data input.

Theano is the underlying framework of many recently developed advanced machine learning tools, especially in deep learning. This includes the CNN implementation scheme *LeNet* that is the basis of the scheme developed in this thesis.

CNN implementation scheme *LeNet*

The CNN constructed and used here is modeled after an early CNN scheme, *LeNet*; that went on to become one of the most successful CNN implementations. It was developed in 1998 by Y. LeCun et al. at AT&T labs as a tool for the recognition of handwritten letters with the goal of automatic processing of banking checks [5].

LeNet implements the theoretical model reviewed in chapter two, schematically shown in figure Fig. 2.

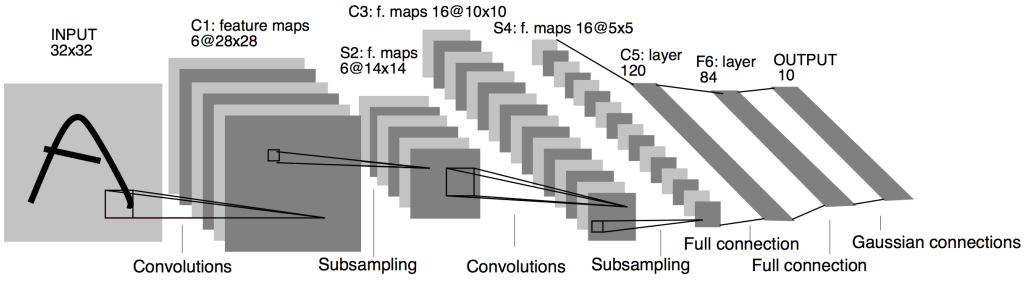

Figure 2: Recognition of handwritten letters and digits with *LeNet* (from [5]). The schematic overview shows the convolutional pooling in the hierarchy of layers in vertical direction.

The original image data set *NIST* (later updated to *MNIST*) [17] consists of 70,000 images of size 32×32 pixels that show handwritten letters or digits. The CNN recognizes elementary visual features and encodes them during training in the filters of its lower layers by adjusting the respective weights. The hierarchical system of layers combines those elementary features into increasingly complex patterns that form digits or letters. When splitting MNIST into 60,000 images for training and 10,000 for testing, a performance of over 90% correctly classified images was achieved on the available machines (24 GB RAM).

In this thesis, the basic scheme of *LeNet* was adapted for developing a CNN event classification method. The implementation of this system and essential steps of the work flow are presented in the next section. MNIST was used as test data for setting up the basic structure of the tool and its structure served as example for the event matrix format.

Implementation of Event Classification

The development of a CNN tool for event classification directly implements the theoretical model described in Chapter 2 and is inspired by the *LeNet* implementation scheme presented above (following [5, 9]). The work flow is summarized in Fig. 3.

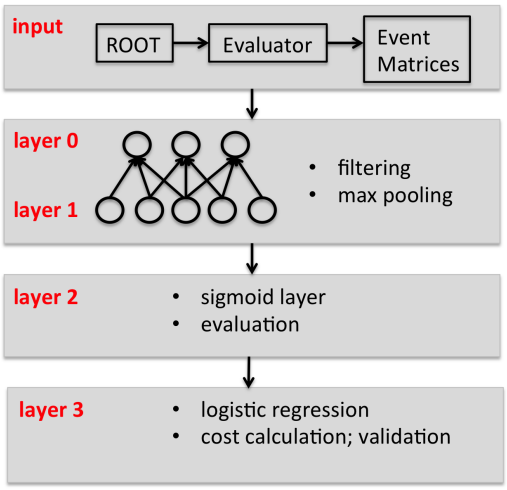

Figure 3: Work flow for CNN Event Classification that underlies the implementation in this thesis.

In more detail, the event classification requires the following steps:

1. **Processing event data**
   The event information is given as ROOT data, i.e. in the data format used by CERN's data analysis framework ROOT. For this, the class *Evaluator* includes an interface to extract for each event the features of every detected object. Those include the transverse momentum p_T, as well as the isolation ξ and - depending on the chosen format of the event matrix - either (ϕ, η) or (x, z).

2. **Constructing event matrices**
   Evaluator also constructs event matrices for each event from the given data by filling the information in an empty matrix using the earlier introduced scaling functions, symbols and color codes (see section 3.1). Event matrices are constructed for each event and stacked in a three-dimensional *theano* object that can be read by the CNN. To ensure comparability with other methods, three commonly used cuts are applied to the data:

   - The trigger match has to be 1.
   - There should be at least two jets.
   - The invariant mass of the two jets with the highest transverse momentum has to be at least 150 GeV

3. **Constructing the CNN**
   For the basic setup of the CNN the following classes from the *LeNet* scheme are utilized:

   (a) *LogisticRegression*: Implements logistic regression for performing classification tasks (see 1.19).

   (b) *load_data*: Enables efficient processing of large input by splitting the data into small batches using *theano's* shared variables.

   (c) *LeNetConvPoolLayer*: Provides the setup for the convolutional filters. During construction, all filters are initialized with random weights that are adjusted during training.

   (d) *HiddenLayer*: Implements layers that hold filters and set up a routine for vertical processing and convolution of input information.

4. **Event data**
   The event data, containing both signal and background events, is split into three subsets:

   - *training* (85 %)
   - *validation* (5 %)
   - *testing* (10 %)

   The subsets are randomly sampled from both signal and background sets and shuffled to eliminate influences of any ordering structure. The subsamples consist of *theano* matrix objects. During sampling, labels are stored in a separate *theano* vector object to track the true classification.

5. **Training**
   The CNN consists of four layers in total, all objects of the basic class *LeNetConvPoolLayer*:

   (a) *input*: Initial layer that reads in sets of event matrices.

   (b) *hidden layer 1*: Lower convolutional layer that performs a first convolution with simple filters and dimensional reduction using *maxpooling*. The layer attempts to recognize simple features.

   (c) *hidden layer 2*: Upper convolutional layer that performs a second convolution with complex filters that summarize the simple filters in a hierarchical merge. The goal is to recognize more complex patterns that are composed of simple features.

   (d) *output*: The output layer performs the classification task based on the recognized patterns in the proceeding layer. It also returns the results.

   The CNN is based on two principles, *convolutional pooling* and *maxpooling*. Convolutional pooling reduces the image size by "zooming out", i.e. attempting to focus on features on a more coarse scale by summarizing the

filters of the previous layer into larger units. Maxpooling performs a direct dimensional reduction of the image size utilizing the (usually present) sparsity of the input data. For this, $n \times n$-submatrices are replaced by a single entry chosen as the maximum norm of the submatrix. In this study, $n = 2$ is chosen constantly.

6. **Validation and Testing**
   In each training step, a validation routine is performed that adjusts filter weights using a stochastic gradient decent on the validation set. Subsequently, the performance of the CNN is evaluated on new, unseen data (the test set). During the training of the CNN, one loops over training step, validation and testing until the learning rate (i.e. the improvement in test performance after one training step) drops below a predefined minimal learning rate and the number of, then, additionally performed steps (determined by the variable *patience*) is done.

With the transverse momentum p_T and the location, the CNN takes as input 'simple' parameters. Given that the event matrices can only accommodate a rather small number of parameters, one would intuitively think to choose more complex variables that summarize a number of parameters in order to store as much information as possible. However, it is crucial to choose a pairwise independent input for the first layer that can be captured by the filters on low levels. As information is convoluted and moves upwards in the CNN hierarchy, those independent information are summarized into more complex features. The goal of the training step is to adjust the filter weights in a way that the filters in the upper hidden layer recognize features that depend on multiple input parameters – acting as those complex variables used in standard statistical analysis.

2.4 Results of test analysis

CNN analysis with Cylindrical Event Matrices

As test case, the CNN method is applied to $WZjj - EW$/ $WZjj - QCD$ event data as described in section 3.2 . In what follows, two cases are considered:

1. All cuts described above are used to filter the data. This reduces the size of the data set considerably from $\sim 7,000/12,000$ to $\sim 5,200/6,600$, but ensures comparability with other analysis tools for which benchmark results are available.

2. In a second analysis, it is only filtered for events with at least two jets, i.e. using the full set of $\sim 7,000/12,000$. This leads to about one third larger data sets. To ensure structural comparability, a $3:4$ ratio is maintained for signal : background in the unfiltered analysis.

To evaluate the performance of the CNN, both the pre-implemented *CNN score* and conventionally used significance scores are computed. The later ones are

determined in terms of S and B where S is the number of correctly classified signal events and B the number of false positives. Results for both test cases can be found in the tables below.

Parameter		**Value**
signal events	in raw data	6991
	detected by CNN	3440
	scaled	11.46
background events	in raw data	8900
	false positive signal detected by CNN	560
	scaled	14.87
performance	CNN score	0.86 (86 %)
	significance $S/\sqrt{B}$	2.97
	significance $S/\sqrt{S+B}$	2.23

Table 2.1: Results for event data with the full set of event data, i.e. original data only filtered for the presence of two jets.

Parameter		**Value**
signal events	in raw data	5208
	detected by CNN	2610
	scaled	8.69
background events	in raw data	6636
	false positive signal detected by CNN	1390
	scaled	36.97
performance	CNN score	0.63 (63 %)
	significance $S/\sqrt{B}$	1.43
	significance $S/\sqrt{S+B}$	1.29

Table 2.2: Results for event data with additional cuts to ensure comparability as described in the previous section.

The *CNN score* is based on the validation score computed by the CNN. It measures the success in classification for the optimal assignment of filter weight, i.e. the assignment with the best testing performance. The significance scores $S/\sqrt{B}$ and $S/\sqrt{S+B}$ are also computed for the optimal weights. For this, the number of detected events in the test set (10% of all events) is projected onto the total number of events. The scaling of detected events follows [3] where weights are assigned in terms of the number of expected events at $20.3fb^{-1}$: 23.75 for $WZjj - EW$ and 327 for $WZjj - QCD$ [18].

CNN analysis with Cartesian Event Matrices

The same two test cases were analyzed using event matrices in Cartesian format, the results are summarized in the tables below.

Parameter		**Value**
signal events	in raw data	6991
	detected by CNN	3490
	scaled	11.62
background events	in raw data	8900
	false positive signal detected by CNN	1510
	scaled	40.17
performance	CNN score	0.70 (70 %)
	significance $S/\sqrt{B}$	1.83
	significance $S/\sqrt{S+B}$	1.61

Table 2.3: Results for event data with no additional preselection using event matrices of Cartesian format.

Parameter		**Value**
signal events	in raw data	5208
	detected by CNN	2450
	scaled	8.16
background events	in raw data	6636
	false positive signal detected by CNN	1550
	scaled	41.23
performance	CNN score	0.61 (61 %)
	significance $S/\sqrt{B}$	1.27
	significance $S/\sqrt{S+B}$	1.16

Table 2.4: Results for event data with cuts using event matrices in Cartesian format.

As in the study with event matrices in cylindrical format, one observes an increase in performance when the larger training set is used. Overall, the Cartesian format performs significantly weaker than the cylindrical one.

Comparison with other methods

Finally, the results are compared with conventionally used methods, namely cut-based analysis (rectangular filters) and Boosted Decision Trees (BDT). The comparison is based on the filtered data set with cuts as specified above.

The original analysis [3] achieved a score of $S/\sqrt{B} = 1.39$ with a cut-based analysis (rectangular filters) [18] and the full preselection, which is of the same order as the CNN's significance score when using the cylindrical format. The Cartesian format performs significantly weaker.

In addition, a comparison with BDTs was performed [19]. In contrast to the CNN, the BDT can take complex variables as input that are computed from simpler, independent variables. This allows for providing larger amounts of initial information than in the case of CNNs. In a first step, the four variables with the

best performance are determined from a wide range of statistical parameters. This is achieved with the algorithm *Neurobayes* that computes a ranking of variables based on performance using a neuronal network. In the present study, the four highest ranked variables were:

- η-difference of tagging jets
- ϕ-difference of tagging jets
- transverse momentum of first two tagging jets

Subsequently, the BDT was trained with these four variables and its performance was evaluated on a training set. In contrast to the CNN method, the BDT evaluation gives not a binary classification score $\in \{-1, 1\}$, but a probabilistic score $\in [-1, 1]$ that can be interpret as the likelihood of an event being signal (close to 1) or background (close to -1). Instead of counts, the BDT evaluation gives a distribution that can be cut with a significance threshold. Table 2.5 summarizes the significance scores for different thresholds.

threshold	**0**	**0.2**	**0.3**	**0.4**
$S/\sqrt{B}$	1.26	1.75	1.71	1.62
$S/\sqrt{S+B}$	1.32	2.21	2.30	2.36

Table 2.5: Results for event classification with BDT using event data with full preselection. (Courtesy: [19])

The results show overall a stronger performance of the BDT compared with the CNN. Only for the smallest threshold (0, first column) the CNN outperformed the BDT when using the cylindrical format. However, the CNN results for the larger, uncut data set suggest that higher statistics might yield better results. A training set of $\sim 4,500$ is extremely small compared to the $\sim 60,000$ used in the test set of the original *LeNet* analysis. In practice, training sets commonly contain millions of samples. This gives hope, that additional data and therefore higher statistics would significantly improve the performance of the CNN compared to other methods.

Conclusions

This thesis develops a machine learning tool that utilizes deep learning to classify events in Di-Boson channels. A Convolutional Neural Network is trained to discriminate $WZjj - EW$ signal from a $WZjj - QCD$ background using a matrix representation termed *event matrix*. The event matrices store information on the location and the transverse momentum of the detected objects in cylindrical or carthesian coordinates – providing a format that is readable by the CNN.

Implementations with both formats were able to perform the event classification task successfully to some extend, however, the cylindrical format showed a stronger performance in both test cases. Being additionally the more 'natural' format since it resembles the geometry of the detector more closely, this makes the cylindrical event matrices the format of choice for possible further studies.

Tested on a small use case [3], the CNN achieved a performance comparable to simple cut-based analysis. A comparison with Boosted Decision Trees [19] showed a weaker performance of the CNN. One reason might lie in the ability of BDTs to operate on a set of non-independent variables allowing for the use of more complex variables that summarize a set of simpler parameters. This enables the BDT to process more information than the CNN that requires an input of independent variables.

On the other hand, this thesis was not able to demonstrate the full potential of the CNN due to a number of technical constraints as discussed below. This is indicated by the following observation: When removing typical cuts and increasing the size of the training set, the performance increased significantly. While this is only a small use case, the results hint on a good performance of the CNN on larger data sets. This comes as no surprise: As with every supervised method, the performance greatly depends on the size and quality of the training set. The here implemented CNN is modeled after the image recognition tool *LeNet* which is typically trained with $\sim 60,000$ images as opposed to roughly $\sim 4,500$ in the here presented study. While there was no larger data set available at this time, the tool might show much better performance once new data becomes available.

Another limiting factor that potentially decreased the performance was the computational constraints of working memory: To accurately represent the given parameters, one would need to increase the dimension of the event matrices, i.e. refine the grid to accommodate more details. For this, the CNN would need to be trained on a larger machine than available in this study (24 GB RAM).

Larger working memory would allow for the processing of larger event matrices and therefore the inclusion of additional parameters; for instance the isolation as a scaling factor for the symbolic representation of the detected particles.

Furthermore, one could refine the tool by optimizing the number of hidden layers in the CNN. In the demonstrated use case, a static CNN architecture with the minimum number of layers is implemented. To raise the pattern recognition abilities one could include additional layers that improve the identification features using different scales. As a possible future direction, one could move from a static to a dynamic architecture that optimizes the number of layers on basis of the given data during training. This was beyond the scope of this bachelor's thesis.

While this thesis could only develop a basic CNN scheme for event classification, it still demonstrated the potential of CNNs for the specific task of discriminating a desired signal from a background. It also demonstrated their use beyond the conventional task of classifying visual features. So far, CNNs have been successfully applied in computer vision, such as face recognition or object classification in visual inputs. This thesis gave an example for the broad applicability and power of CNNs for recognizing and learning patterns in data if provided in a format that they are capable of processing. In this context, this study showed three major obstacles that challenge the applicability of CNNs:

1. **Data representation**
   The data needs to be presented in a format that is processable by the CNN, i.e. that mimics a visual representation.

2. **Large training set**
   In analogy to the visual cortex, the success of learning and the accuracy of information retrieval depends strongly on the amount of training. Large training sets can achieve a detailed adjustment of the filters that is expected to lead to significantly better classification performance. Overfitting as a common problem in conventional neural networks is expected to occur only on much larger training sets in the present method, due to the increased complexity of the CNN architecture. While there are to this day no theoretical results for determining optimal or minimal sizes of training sets, empirical values are generally large – in most applications ranging in orders of millions. Whereas a generation of training sets of this size through event simulation might not be possible under present day's technical constraints, the rapid development of computer technology gives hope that the CNN method becomes more applicable in the years to come.

3. **Computing resources**
   The downside of CNNs and deep learning in general is their computational expense that requires extensive working memory for the training step. The available computational resources limited the resolution and therefore the accuracy of this study - but this could be overcome with larger machines.

In summary, this thesis introduced a new deep learning method for event classification in diboson channels that extends the current set of analysis tools with a new approach. The application on a small use case gives hope – though constraint by the availability of data and computational power – for future applications. The rapid development of the field of machine learning together with ongoing technological advances in the collection of experimental data anticipate that existing obstacles will be overcome in the nearer future and allow for the broad applicability of deep learning in data analysis across all scientific fields. This thesis demonstrated advantages as well as key challenges for the application in event classification of deep learning methods in general and CNNs in particular, hoping to lay a basis for future methods and studies.

Appendix A

Additional approach for Event Matrices

In Chapter 2, *event matrices* are developed as representations of event data in a form that is capable for CNNs. The main text only describes the two successful approaches that were used for the case study in this thesis. This section briefly describes another initially proposed approach that turned out unsuitable for CNNs, and discusses why the approach failes. The shortcomings of this approach are generic and highlight essential features of CNNs that need to be considered when using this machine learning method for data analysis.

The general idea of the feature format is to represent features from detected objects with a color code in a matrix structure: Each row represents a feature and each column a detected object (see Fig. 1). The entries are the values associated with the respective features for each particle, scaled onto $[0, 255]$ where the minimum and maximum values are represented by 0 and 255, i.e. $\text{val}_{min} \mapsto 0$ and $\text{val}_{max} \mapsto 255$.

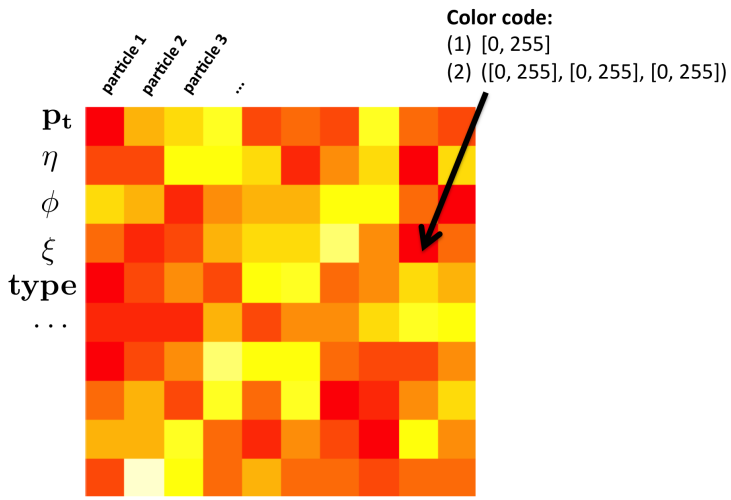

Figure 1: Schematic representation of the feature format for event matrices.

However, this approach proved unsuccessful in experiments for two reasons:

1. For a typical event there are six detected objects and four independent parameters (location, transverse momentum and isolation). This results in event matrices of very small size that does not allow for efficient application of *pooling* which is the most essential step in the CNN algorithm.

2. The feature format is not translation invariant. The convolutional pooling that is intrinsic to each layer of the CNN assumes translational invariance, since the convolutional filters that represent specific local correlation patterns are applied on the whole matrix.

The poor performance of the algorithm using this format underlines the key importance of both the translational invariance and the careful choice of parameters such that the network architecture allows for efficient pooling.

Appendix B

Implementation

This section includes the *main*-file and two classes that setup the event matrices. Basic classes for the CNN are taken from *LeNet*[9, 16].

B.1 main file (work flow)

```
import sys
import os
import timeit

import argparse
import Evaluator
import numpy as np
import random
import math

import pprint # nice output
import yaml

import pdb # debug
import theano
import theano.tensor as T
from theano.tensor.signal import downsample
from theano.tensor.nnet import conv2d

from logistic_sgd import LogisticRegression, load_data
from mlp import HiddenLayer

""" Workflow of CNN event classification, based on
    LeNet (http://deeplearning.net/tutorial/lenet.html#lenet)

    """

class LeNetConvPoolLayer(object):
    """Pool Layer of a convolutional network """

    def __init__(self, rng, input, filter_shape, image_shape,
                 poolsize=(2, 2)):
        """
            Allocate a LeNetConvPoolLayer with shared variable
            internal parameters.

            :type rng: numpy.random.RandomState
```

```
38              :param rng: a random number generator used to
39              initialize weights
40
41              :type input: theano.tensor.dtensor4
42              :param input: symbolic image tensor, of shape
43              image_shape
44
45              :type filter_shape: tuple or list of length 4
46              :param filter_shape: (number of filters, num input
47              feature maps,filter height, filter width)
48
49              :type image_shape: tuple or list of length 4
50              :param image_shape: (batch size, num input feature
51              maps,image height, image width)
52
53              :type poolsize: tuple or list of length 2
54              :param poolsize: the downsampling (pooling)
55              factor (#rows, #cols)
56              """
57
58          assert image_shape[1] == filter_shape[1]
59          self.input = input
60
61      # there are "num input feature maps * filter height
62      # * filter width" inputs to each hidden unit
63      fan_in = np.prod(filter_shape[1:])
64      # each unit in the lower layer receives a gradient from:
65      # "num output feature maps * filter height
66      #  * filter width" / pooling size
67      fan_out = (filter_shape[0]
68                 * np.prod(filter_shape[2:])//np.prod(poolsize))
69      # initialize weights with random weights
70      W_bound = np.sqrt(6. / (fan_in + fan_out))
71      self.W = theano.shared(np.asarray(
72                  rng.uniform(low=-W_bound, high=W_bound,
73                  size=filter_shape),
74                  dtype=theano.config.floatX),borrow=True)
75
76      # the bias is a 1D tensor -- one bias per output feature map
77      b_values = np.zeros((filter_shape[0],),
78                          dtype=theano.config.floatX)
79      self.b = theano.shared(value=b_values, borrow=True)
80
81      # convolve input feature maps with filters
82      conv_out = conv2d(input=input,filters=self.W,
83                  filter_shape=filter_shape,
84                  input_shape=image_shape)
85
86      # downsample each feature map individually, using maxpooling
87      pooled_out = downsample.max_pool_2d(input=conv_out,
88                  ds=poolsize,ignore_border=True)
89
90      # add the bias term. Since the bias is a vector (1D array), we
91      # first reshape it to a tensor of shape (1, n_filters, 1, 1).
92      # Each bias will thus be broadcasted across mini-batches and
93      # feature mapwidth &amp;amp; height
94      self.output = T.tanh(pooled_out +
95                           self.b.dimshuffle('x', 0, 'x', 'x'))
96
97      # store parameters of this layer
98      self.params = [self.W, self.b]
```

```
99
100     # keep track of model input
101     self.input = input
102
103 def analyze(args):
104     evaluator = Evaluator.Evaluator()
105     evaluator.inputDefinition()
106     #evaluator.setMaxEvents(10)
107     #evaluator.setMaxEvents(3000)
108
109     # evaluateSample: cylindical format
110     # evaluateSample2: carthesian format
111     signalinfo = evaluator.evaluateSample2(args.signal)
112     bkginfo    = evaluator.evaluateSample2(args.background)
113     #for e in signalinfo[:3]:
114     #    print "size: {} x {}".format(len(e), len(e[0]))
115     #    print "sum: {} ... {}".format(sum(e[0]), sum(sum(e)))
116
117     #    print e
118     # signalinfo has 3 indices
119     # first one: event id
120     # second one: phi bin
121     # third one : eta bin
122
123     ##################
124     # PREPARE DATA #
125     ##################
126
127     config = yaml.load(open("config.yaml","r"))
128
129     def shared_dataset(data_xy, borrow=True):
130         """ Function that loads the dataset into shared variables
131
132             The reason we store our dataset in shared variables is
133             to allow Theano to copy it into the GPU memory (when
134             code is run on GPU). Since copying data into the GPU is
135             slow, copying a minibatch everytime is needed (the
136             default behaviour if the data is not in a shared
137             variable) would lead to a large decrease in performance.
138             """
139         data_x, data_y = data_xy
140         shared_x = theano.shared(np.asarray(data_x,
141                 dtype=theano.config.floatX),borrow=borrow)
142         shared_y = theano.shared(np.asarray(data_y,
143                 dtype=theano.config.floatX),borrow=borrow)
144
145         # When storing data on the GPU it has to be stored as
146         # floats therefore we will store the labels as ``floatX``
147         # as well (``shared_y`` does exactly that). But during our
148         # computations we need them as ints (we use labels as index,
149         # and if they are floats it doesn't make sense) therefore
150         # instead of returning ``shared_y`` we will have to cast it
151         # to int. This little hack lets ous get around this issue
152         return shared_x, T.cast(shared_y, 'int32')
153
154     # parameters | read from config
155     batch_size=config["batch_size"]
156     nkerns=config["nkerns"]
157     learning_rate=config["learning_rate"]
158     n_epochs=config["n_epochs"]
159     n_fg=len(signalinfo)
```

```
160     print n_fg
161     n_bg=len(bkginfo)
162     print n_bg
163     #n_event_fg=config["n_event_fg"]
164     n_event_fg=5208
165     #n_event_fg=6991
166     #n_event_bg=config["n_event_bg"]
167     n_event_bg=6636
168     #n_event_bg=12118
169     n_event=n_event_fg + n_event_bg
170     p_train=config["p_train"]
171     p_test=config["p_test"]
172     p_valid=config["p_valid"]
173
174     rng = np.random.RandomState(23455)
175     r_fg=random.sample(signalinfo,n_event_fg)
176     r_bg=random.sample(bkginfo,n_event_bg)
177
178     # index limits
179     train_end_fg=np.floor(p_train*n_event_fg).astype('int')
180     train_end_bg=np.floor(p_train*n_event_bg).astype('int')
181     test_end_fg=np.floor(p_test*n_event_fg).astype('int')+1
182             +np.floor(p_train*n_event_fg).astype('int')
183     test_end_bg=np.floor(p_test*n_event_bg).astype('int')+1
184             +np.floor(p_train*n_event_bg).astype('int')
185
186     train=np.concatenate((r_fg[1:train_end_fg],r_bg[1:train_end_bg]),
187             axis=0)
188     test=np.concatenate((
189             r_fg[(train_end_fg.astype('int')
190                     +1):test_end_fg.astype('int')],
191             r_bg[(train_end_bg.astype('int')
192                     +1):test_end_bg.astype('int')]),axis=0)
193     valid=np.concatenate((r_fg[(test_end_fg.astype('int')
194                     +1):n_event_fg],r_bg[(test_end_bg.astype('int')
195                     +1):n_event_bg]),axis=0)
196
197     # labels
198     labels_train=np.concatenate((np.ones((1,train_end_fg)),
199             np.zeros((1,train_end_bg))),axis=1)[0]
200     labels_test=np.concatenate((np.ones((1,np.floor(p_test*n_event_fg))),
201             np.zeros((1,np.floor(p_test*n_event_bg)))),axis=1)[0]
202     labels_valid=np.concatenate((np.ones((1,n_event_fg-test_end_fg)),
203             np.zeros((1,n_event_bg-test_end_bg))),axis=1)[0]
204     # shuffle
205     r1=random.random()
206     random.shuffle(train,lambda:r1)
207     random.shuffle(labels_train,lambda:r1)
208
209     r2=random.random()
210     random.shuffle(test,lambda:r2)
211     random.shuffle(labels_test,lambda:r2)
212
213     r3=random.random()
214     random.shuffle(valid,lambda:r3)
215     random.shuffle(labels_valid,lambda:r3)
216
217     # read in GPU
218     train_data=[train,labels_train]
219     test_data=[test,labels_test]
220     valid_data=[valid,labels_valid]
```

```
221
222     train_set, train_labels = shared_dataset(train_data)
223     test_set, test_labels = shared_dataset(test_data)
224     valid_set, valid_labels = shared_dataset(valid_data)
225
226     pdb.set_trace()
227
228     # compute number of minibatches for training,
229     # validation and testing
230     n_train_batches = train_set.get_value(borrow=True).shape[0] //
231                 batch_size
232     n_test_batches = test_set.get_value(borrow=True).shape[0] //
233                 batch_size
234     n_valid_batches = valid_set.get_value(borrow=True).shape[0] //
235                 batch_size
236
237     print n_test_batches
238
239     # allocate symbolic variables for the data
240     index = T.lscalar()  # index to a [mini]batch
241
242     # start-snippet-1
243     x = T.matrix('x')   # the data is presented as rasterized images
244     y = T.ivector('y')  # the labels are presented as 1D vector of
245     # [int] labels
246
247     print x
248     print y
249
250     ################
251     # BUILD MODEL #
252     ################
253     print('... building the model')
254
255     # parameters
256     #input_size = [32,20] # determine directly from event matrix
257     #input_size = [64,100]
258     input_size = [50,50]
259     filter1 = [3,3]
260     filter2 = [2,2]
261     maxpool = [2,2]
262
263     # Reshape matrix of rasterized images of shape to a 4D tensor
264     layer0_input = x.reshape((batch_size, 1, input_size[0],
265                               input_size[1]))
266     print layer0_input.shape
267
268     # Construct the first convolutional pooling layer:
269     # filtering reduces matrix size to (32-3+1 , 20-3+1) = (30, 18)
270     #       -> this is N_Pixel - N_Dimensions_Filter + 1 for
271     #           each coordinate
272     # maxpooling reduces this further to (30/2, 18/2) = (15, 9)
273     #       -> this is half in each coordinate
274     # 4D output tensor is thus of shape (batch_size, nkerns[0], 15, 9)
275     #       -> in total: (batch_size, nkerns[0],
276     #       (x_Pixel - x_DimFilter + 1)/2, (y_Pixel - y_DimFilter + 1)/2)
277     layer0 = LeNetConvPoolLayer(rng,input=layer0_input,
278             image_shape=(batch_size, 1, input_size[0], input_size[1]),
279             filter_shape=(nkerns[0], 1, filter1[0], filter1[1]),
280             poolsize=(maxpool[0], maxpool[1]))
281
```

```
282     print layer0.b
283     print layer0.W.eval()
284
285     # Construct the second convolutional pooling layer
286     # filtering reduces the image size to (15-2+1, 9-2+1) = (14, 8)
287     # maxpooling reduces this further to (14/2, 8/2) = (7, 4)
288     # 4D output tensor is thus of shape (batch_size, nkerns[1], 7, 4)
289     layer1 = LeNetConvPoolLayer(rng,input=layer0.output,
290             image_shape=(batch_size, nkerns[0],
291                          (input_size[0]-filter1[0]+1)/maxpool[0],
292             (input_size[1]-filter1[1]+1)/maxpool[1]),
293             filter_shape=(nkerns[1], nkerns[0], filter2[0], filter2[1]),
294             poolsize=(maxpool[0], maxpool[0]))
295
296     # the HiddenLayer being fully-connected, it operates on 2D matrices of
297     # shape (batch_size, num_pixels) (i.e matrix of rasterized images).
298     # This will generate a matrix of shape (batch_size, nkerns[1] * 7 * 4).
299     layer2_input = layer1.output.flatten(2)
300
301     # construct a fully-connected sigmoidal layer
302     layer2 = HiddenLayer(rng,input=layer2_input,
303             n_in=nkerns[1] * (((input_size[0]-filter1[0]+1)/maxpool[0]
304                 - filter2[0]+1)/maxpool[0]) *
305             (((input_size[1]-filter1[1]+1)/maxpool[1]
306                 - filter2[1]+1)/maxpool[0]),n_out=3,activation=T.tanh)
307
308     # classify the values of the fully-connected sigmoidal layer
309     layer3 = LogisticRegression(input=layer2.output, n_in=3, n_out=3)
310
311     # the cost we minimize during training is the NLL of the model
312     cost = layer3.negative_log_likelihood(y)
313
314     # create a function to compute the mistakes that are made by the model
315     test_model = theano.function([index],
316                 layer3.errors(y),
317                 givens={x: test_set[index * batch_size:
318                                     (index + 1) * batch_size],
319                         y: test_labels[index * batch_size:
320                                     (index + 1) * batch_size]})
321
322     validate_model = theano.function([index],
323                 layer3.errors(y),
324                 givens={x: valid_set[index * batch_size:
325                                      (index + 1) * batch_size],
326                         y: valid_labels[index * batch_size:
327                                      (index + 1) * batch_size]})
328
329     sig_s_model = theano.function([index],
330                 layer3.sig_signal(y),
331                 givens={x: valid_set[index * batch_size:
332                                      (index + 1) * batch_size],
333                         y: valid_labels[index * batch_size:
334                                      (index + 1) * batch_size]})
335
336     sig_b_model = theano.function([index],
337                 layer3.sig_background(y),
338                 givens={x: valid_set[index * batch_size:
339                                      (index + 1) * batch_size],
340                         y: valid_labels[index * batch_size:
341                                      (index + 1) * batch_size]})
342
```

```
343     # create a list of all model parameters to be fit by gradient descent
344     params = layer3.params + layer2.params + layer1.params
345                 + layer0.params
346
347     # create a list of gradients for all model parameters
348     grads = T.grad(cost, params)
349
350     # train_model is a function that updates the model parameters by
351     # SGD Since this model has many parameters, it would be tedious to
352     # manually create an update rule for each model parameter. We thus
353     # create the updates list by automatically looping over all
354     # (params[i], grads[i]) pairs.
355     updates = [(param_i, param_i - learning_rate * grad_i)
356             for param_i, grad_i in zip(params, grads)]
357
358     train_model = theano.function([index],cost,updates=updates,
359                 givens={x: train_set[index * batch_size:
360                                      (index + 1) * batch_size],
361                         y: train_labels[index * batch_size:
362                                      (index + 1) * batch_size]})
363
364     ################
365     # TRAIN MODEL #
366     ################
367     print('... training')
368     # early-stopping parameters
369     patience = 10000  # 100 # look as this many examples regardless
370     patience_increase = 2  # wait this much longer when a new best is
371     # found
372     improvement_threshold = 0.995 # a relative improvement of this much
373                                   # is considered significant
374     validation_frequency = min(n_train_batches, patience // 2)
375     # validation_frequency=100
376     print validation_frequency
377     # go through this many
378     # minibatche before checking the network
379     # on the validation set; in this case we
380     # check every epoch
381
382     best_validation_loss = np.inf
383     best_iter = 0
384     test_score = 0.
385     start_time = timeit.default_timer()
386     epoch = 0
387     done_looping = False
388
389     # actual training
390     while (epoch < n_epochs) and (not done_looping):
391         epoch = epoch + 1
392         for minibatch_index in range(n_train_batches):
393             # print minibatch_index
394             iter = (epoch - 1) * n_train_batches + minibatch_index
395             # print range(n_train_batches)
396             if iter % 1 == 0:
397                 # print('training @ iter = ', iter)
398                 cost_ij = train_model(minibatch_index)
399             # print cost_ij
400             if (iter + 1) % validation_frequency == 0:
401
402                 # compute zero-one loss on validation set
403                 validation_losses = [validate_model(i) for i
```

```
404                                 in range(n_valid_batches)]
405                         sig_s = [sig_s_model(i) for i
406                                 in range(n_valid_batches)]
407                         sig_b = [sig_b_model(i) for i
408                                 in range(n_valid_batches)]
409                         this_validation_loss = np.mean(validation_losses)
410                         this_sig_s = np.sum(sig_s)
411                         this_sig_b = np.sum(sig_b)
412                         this_significance = np.divide(this_sig_s,
413                                 np.sqrt(this_sig_b+this_sig_s))
414
415                 # if we got the best validation score until now
416                 if this_validation_loss <  best_validation_loss:
417
418                 #improve patience if loss improvement is good enough
419                 # if this_validation_loss &lt; best_validation_loss *  \
420                    if this_validation_loss < best_validation_loss *  \
421                    improvement_threshold:
422                 patience = max(patience, iter * patience_increase)
423
424                 # save best validation score and iteration number
425                 best_validation_loss = this_validation_loss
426                 best_iter = iter
427                 best_significance = this_significance
428                 best_s = this_sig_s
429                 best_b = this_sig_b
430
431                 # test it on the test set
432                 test_losses = [test_model(i)
433                                for i in range(n_test_batches)]
434                 test_score = np.mean(test_losses)
435
436             #if patience &lt;= iter:
437             if patience < iter:
438                 done_looping = True
439             break
440
441 end_time = timeit.default_timer()
442 print('Optimization complete.')
443 print('Best validation score of %f %% obtained at iteration %i, '
444           'with test performance %f %%' %
445           (best_validation_loss * 100., best_iter + 1,
446            test_score * 100.))
447 print('Best significance:')
448 print best_significance
449 print('No. of correct signal:')
450 print best_s
451 print('False correct signal:')
452 print best_b
453 #print(('The code for file ' +
454 #       os.path.split(__file__)[1] +
455 #       ' ran for %.2fm' % ((end_time - start_time) / 60.)),
456 #       file=sys.stderr)
457
458 #============================================================
459 def main(argv):
460     """
461         Main function to be executed when starting the code.
462         """
463     parser = argparse.ArgumentParser(
464             description = 'find promising variables' )
```

```
465     parser.add_argument("-s", "--signal",      type=str)
466     parser.add_argument("-b", "--background", type=str)
467     parser.add_argument("-c", "--conig", type=str)
468     args = parser.parse_args()
469
470     analyze(args)
471
472
473 #==========================================================
474 if __name__ == "__main__":
475     """
476         Here the code should appear that is executed when running
477         the code directly (and not import it in another python
478         file via 'import run.py').
479         """
480
481     # start main program
482     main( sys.argv[1:] )
```

Listing B.1: Workflow of CNN analysis.

B.2 Class *Particle*

The code for the class *Particle* was partially written and provided by Felix Socher.

```
1  import ROOT
2  from array import array
3
4  class Particle(object):
5      """
6          Implements data structure for representing particle
7          features. Written by Felix Socher.
8
9          """
10     def __init__(self, prefix):
11         super(Particle, self).__init__()
12         self.Tree = None
13         self.prefix = prefix
14         self.tlv = ROOT.TLorentzVector(1,0,0,0)
15
16     def initialize(self, tree):
17         # Possible variables are pt, eta and phi. It is assumed
18         # that the particles are massless
19         self.Tree = tree
20         self.pt  = self.activate("f", "Pt") if not self.prefix
21                     in "EtMiss" else self.activate("f", "")
22         self.eta = self.activate("f", "Eta")
23         self.phi = self.activate("f", "Phi")
24         self.iso = self.activate("f", "Iso")
25         self.x = self.activate("f", "X")
26         self.z = self.activate("f", "Y")
27
28     def activate(self, vartype, var):
29         try:
30             branchname = "%s%s" %(self.prefix, var)
31             variable = array(vartype,[0])
32             self.Tree.SetBranchStatus(branchname,1)
33             self.Tree.SetBranchAddress(branchname, variable)
34             return variable
35         except:
```

```
36              print "No branch found for %s%s, setting it to 0"
37                      %(self.prefix, var)
38          return array(vartype,[0])
39
40      def update(self):
41          self.tlv.SetPtEtaPhiM(self.pt[0], self.eta[0],
42                                self.phi[0], 0)
43
44      def str(self):
45          return self.prefix
46
47      def pt(): return self.pt[0]
48      def eta(): return self.eta[0]
49      def phi(): return self.phi[0]
50      def iso(): return self.iso[0]
51      def x():   return self.x[0]
52      def z():   return self.z[0]
```

Listing B.2: class Particle

B.3 *Evaluator*: Construction of Event Matrices

```
1 import Particle
2 import ROOT
3 import numpy as np
4 import math
5 import pdb
6
7 # Variable Constructor
8 class Evaluator(object):
9     """docstring for Finder"""
10    def __init__(self):
11        super(Evaluator, self).__init__()
12        self.finalstate = dict()
13        self.maxEvents = None
14
15    def addParticle(self, prefix):
16        self.finalstate.update({prefix:Particle.Particle(prefix)})
17
18    def inputDefinition(self):
19        # This defines the particles used from the sample
20        self.addParticle("ZLepton1")
21        self.addParticle("ZLepton2")
22        self.addParticle("WLepton")
23        self.addParticle("EtMiss")
24        self.addParticle("TagJet1")
25        self.addParticle("TagJet2")
26
27    def evaluateSample(self, sampleName):
28        t = self.setupTree(sampleName)
29        nentries = t.GetEntries()
30        print nentries
31        if self.maxEvents:
32            nentries = min(nentries, self.maxEvents)
33        return self.loopOverEvents(t, nentries)
34
35    def evaluateSample2(self, sampleName):
36        t = self.setupTree(sampleName)
37        nentries = t.GetEntries()
```

```
38          print nentries
39          if self.maxEvents:
40              nentries = min(nentries, self.maxEvents)
41          return self.loopOverEvents2(t, nentries)
42
43      def setMaxEvents(self, nMax):
44          self.maxEvents = nMax
45
46      def setupTree(self, sampleName):
47          t = ROOT.TChain("MiniTree")
48          t.Add(sampleName)
49          t.SetBranchStatus("*",1)
50          [particle.initialize(t)
51              for particle in self.finalstate.values()]
52          return t
53
54      def loopOverEvents(self, t, nentries):
55          results = []
56          for n in xrange (nentries):
57              t.GetEntry(n)
58              [particle.update()
59                  for particle in self.finalstate.values()]
60              if not self.preselection(t): continue
61              results.append(self.evaluateEvent())
62                  print "done looping over {} events".format(nentries)
63                      return results
64
65  def loopOverEvents2(self, t, nentries):
66      results = []
67          for n in xrange (nentries):
68              t.GetEntry(n)
69              [particle.update()
70                  for particle in self.finalstate.values()]
71              if not self.preselection(t): continue
72              results.append(self.evaluateEventCarth())
73      print "done looping over {} events".format(nentries)
74          return results
75
76  def preselection(self, t):
77      #if t.ZLeptonsZWindow > 10: return False
78      #if t.WTransverseMass < 30: return False
79      if t.TriggerMatch != 1: return False
80          if t.NJets < 2: return False
81              if t.TagJetsInvariantMass < 150: return False
82                  return True
83
84
85  def buildEventMatrix(self,pt,phi,eta,iso,typ):
86      ## Lepton
87      # scaling: iso == cone radius
88      if typ==0:
89      # parameters
90      minPhi=-3.2
91      maxPhi=3.2
92      dPhi=0.1
93      minEta=-5
94      maxEta=5
95      dEta=0.1
96      minPt=15
97      maxPt=300
98      # create event matrix
```

```
    pmat = np.zeros(((maxPhi-minPhi)/dPhi,
                     (maxEta-minEta)/dEta))
    # scale phi
    phi2=math.ceil(abs(phi)/dPhi)
    if phi2>0: phi2=(maxPhi/dPhi)-phi2
    else:      phi2=(maxPhi/dPhi)+phi2
    # scale eta
    eta2=math.ceil(abs(eta)/dEta)
    if eta2>0: eta2=(maxEta/dEta)+eta2
    else: eta2=maxEta/dEta-eta2
    # scale Pt
    pt2=math.floor((float(pt-minPt)/(maxPt-minPt))*255)
    # symbolic
    pmat[int(phi2),int(eta2)] = pt2
    pmat[int(phi2-1),int(eta2)] = pt2
    pmat[int(phi2+1),int(eta2)] = pt2
    pmat[int(phi2),int(eta2-1)] = pt2
    pmat[int(phi2),int(eta2+1)] = pt2
    ## Jets
    if typ==1:
    # parameters
    minPhi=-3.2
    maxPhi=3.2
    dPhi=0.1
    minEta=-5
    maxEta=5
    dEta=0.1
    minPt=15
    maxPt=300
    # create event matrix
    pmat = np.zeros(((maxPhi-minPhi)/dPhi,
                     (maxEta-minEta)/dEta))
    # scale phi
    phi2=math.ceil(abs(phi)/dPhi)
    if phi2>0: phi2=(maxPhi/dPhi)-phi2
    else:      phi2=(maxPhi/dPhi)+phi2
    # scale eta
    eta2=math.ceil(abs(eta)/dEta)
    if eta2>0: eta2=(maxEta/dEta)+eta2
    else: eta2=maxEta/dEta-eta2
    # scale pt
    pt2=math.floor((float(pt-minPt)/(maxPt-minPt))*255)
    pmat[int(phi2+1),int(eta2+1)] = pt2
    pmat[int(phi2+1),int(eta2-1)] = pt2
    pmat[int(phi2-1),int(eta2)] = pt2
    pmat[int(phi2+1),int(eta2)] = pt2
    pmat[int(phi2),int(eta2-1)] = pt2
    pmat[int(phi2),int(eta2+1)] = pt2
    pmat[int(phi2-1),int(eta2+1)] = pt2
    pmat[int(phi2+1),int(eta2-1)] = pt2
    ## Others
    else:
    # parameters
    minPhi=-3.2
    maxPhi=3.2
    dPhi=0.1
    minEta=-5
    maxEta=5
    dEta=0.1
    minPt=15
    maxPt=300
```

```
160     # create event matrix
161     pmat = np.zeros(((maxPhi-minPhi)/dPhi,
162                      (maxEta-minEta)/dEta))
163     # scale phi
164     phi2=math.ceil(phi/dPhi)
165     if phi2>0: phi2=(maxPhi/dPhi)-phi2
166     else:      phi2=(maxPhi/dPhi)+phi2
167     # scale eta
168     eta2=math.ceil(eta/dEta)
169     if eta2>0: eta2=(maxEta/dEta)+eta2
170     else: eta2=maxEta/dEta-eta2
171     # scale pt
172     pt2=(float(pt-minPt)/(maxPt-minPt))*255
173     pmat[int(phi2),int(eta2)] = pt2
174     ## normalize
175     #pmat=np.divide(pmat,np.amax(pmat))
176     pmat=np.reshape(pmat, 6400, order='F')
177     return pmat
178
179 def buildEventMatrixCarth(self,pt,x,z,typ):
180     ## Lepton
181     if typ==0:
182     # parameters
183     xmin=0
184     xmax=500
185     dx=10
186     zmin=0
187     zmax=500
188     dz=10
189     minPt=15
190     maxPt=300
191     # create event matrix
192     pmat = np.zeros(((xmax-xmin)/dx,(zmax-zmin)/dz))
193     # scale x
194     x2=math.ceil(abs(x)/dx)
195     # scale z
196     z2=math.ceil(abs(z)/dz)
197     # scale pt
198     pt2=math.floor((float(pt-minPt)/(maxPt-minPt))*255)
199     pmat[int(x2),int(z2)] = pt2
200     pmat[int(x2-1),int(z2)] = pt2
201     pmat[int(x2+1),int(z2)] = pt2
202     pmat[int(x2),int(z2-1)] = pt2
203     pmat[int(x2),int(z2+1)] = pt2
204     ## Jets
205     if typ==1:
206     # parameters
207     xmin=0
208     xmax=500
209     dx=10
210     zmin=0
211     zmax=500
212     dz=10
213     minPt=15
214     maxPt=300
215     # create event matrix
216     pmat = np.zeros(((xmax-xmin)/dx,(zmax-zmin)/dz))
217     # scale x
218     x2=math.ceil(abs(x)/dx)
219     # scale z
220     z2=math.ceil(abs(z)/dz)
```

```
221     # scale pt
222     pt2=math.floor((float(pt-minPt)/(maxPt-minPt))*255)
223     pmat[int(x2+1),int(z2+1)] = pt2
224     pmat[int(x2+1),int(z2-1)] = pt2
225     pmat[int(x2-1),int(z2)] = pt2
226     pmat[int(x2+1),int(z2)] = pt2
227     pmat[int(x2),int(z2-1)] = pt2
228     pmat[int(x2),int(z2+1)] = pt2
229     pmat[int(x2-1),int(z2+1)] = pt2
230     pmat[int(x2+1),int(z2-1)] = pt2
231     ## Others
232     else:
233     #print 'Particle type unknown.'
234     # parameters
235     xmin=0
236     xmax=500
237     dx=10
238     zmin=0
239     zmax=500
240     dz=10
241     minPt=15
242     maxPt=300
243     # create event matrix
244     pmat = np.zeros(((xmax-xmin)/dx,(zmax-zmin)/dz))
245     # scale x
246     x2=math.ceil(x/dx)
247     # scale z
248     z2=math.ceil(z/dz)
249     # scale pt
250     pt2=(float(pt-minPt)/(maxPt-minPt))*255
251     pmat[int(x2),int(z2)] = pt2
252     ## normalize
253     #pmat=np.divide(pmat,np.amax(pmat))
254     pmat=np.reshape(pmat, 2500, order='F')
255     return pmat
256
257 def evaluateEventCarth(self):
258     # return value is collection of event-matrices
259     p = self.finalstate["WLepton"]
260     mat1=self.buildEventMatrixCarth(p.pt[0], p.x[0],
261                                     p.z[0], 0)
262     p = self.finalstate["ZLepton1"]
263     mat2=self.buildEventMatrixCarth(p.pt[0], p.x[0],
264                                     p.z[0], 0)
265     p = self.finalstate["ZLepton2"]
266     mat3=self.buildEventMatrixCarth(p.pt[0], p.x[0],
267                                     p.z[0], 0)
268     p = self.finalstate["EtMiss"]
269     mat4=self.buildEventMatrixCarth(p.pt[0], p.x[0],
270                                     p.z[0], 2)
271     p = self.finalstat
272     mat5=self.buildEventMatrixCarth(p.pt[0], p.x[0],
273                                     p.z[0], 1)
274     p = self.finalstate["TagJet2"]
275     mat6=self.buildEventMatrixCarth(p.pt[0], p.x[0],
276                                     p.z[0], 1)
277     mat=mat1+mat2+mat3+mat4+mat5+mat6
278     ## normalize
279     #mat=np.divide(mat,np.amax(mat))
280     return mat
281
```

```
282
283 def evaluateEvent(self):
284     # return value is collection of event-matrices
285     p = self.finalstate["WLepton"]
286     mat1=self.buildEventMatrix(p.pt[0], p.eta[0],
287                                p.phi[0], p.iso[0], 0)
288     p = self.finalstate["ZLepton1"]
289     mat2=self.buildEventMatrix(p.pt[0], p.eta[0],
290                                p.phi[0], p.iso[0], 0)
291     p = self.finalstate["ZLepton2"]
292     mat3=self.buildEventMatrix(p.pt[0], p.eta[0],
293                                p.phi[0], p.iso[0], 0)
294     p = self.finalstate["EtMiss"]
295     mat4=self.buildEventMatrix(p.pt[0], p.eta[0],
296                                p.phi[0], 0, 2)
297     p = self.finalstate["TagJet1"]
298     mat5=self.buildEventMatrix(p.pt[0], p.eta[0],
299                                p.phi[0], 0, 1)
300     p = self.finalstate["TagJet2"]
301     mat6=self.buildEventMatrix(p.pt[0], p.eta[0],
302                                p.phi[0], 0, 1)
303     mat=mat1+mat2+mat3+mat4+mat5+mat6
304     ## normalize
305     #mat=np.divide(mat,np.amax(mat))
306     return mat
307
308
309 def buildEventMatrix2(self):
310     # builds feature representation of event matrix
311     # parameter:
312     n_particles=6
313     n_features=4
314     particle=["WLepton","ZLepton1","ZLepton2",
315               "EtMiss","TagJet1","TagJet2"]
316     # create event matrix
317     pmat = np.zeros((n_particles,n_features))
318     # read features
319     p1 = self.finalstate["WLepton"]
320     p2 = self.finalstate["ZLepton1"]
321     p3 = self.finalstate["ZLepton2"]
322     p4 = self.finalstate["EtMiss"]
323     p5 = self.finalstate["TagJet1"]
324     p6 = self.finalstate["TagJet2"]
325     # fill
326     e["TagJet1"]
327     pmat[0,0]=p1.pt[0]
328     pmat[1,0]=p2.pt[0]
329     pmat[2,0]=p3.pt[0]
330     pmat[3,0]=p4.pt[0]
331     pmat[4,0]=p5.pt[0]
332     pmat[5,0]=p6.pt[0]
333     #
334     pmat[0,1]=p1.eta[0]
335     pmat[1,1]=p2.eta[0]
336     pmat[2,1]=p3.eta[0]
337     pmat[3,1]=p4.eta[0]
338     pmat[4,1]=p5.eta[0]
339     pmat[5,1]=p6.eta[0]
340     #
341     pmat[0,2]=p1.phi[0]
342     pmat[1,2]=p2.phi[0]
```

```
343         pmat[2,2]=p3.phi[0]
344         pmat[3,2]=p4.phi[0]
345         pmat[4,2]=p5.phi[0]
346         pmat[5,2]=p6.phi[0]
347         #
348         pmat[0,3]=p1.iso[0]
349         pmat[1,3]=p2.iso[0]
350         pmat[2,3]=p3.iso[0]
351         pmat[3,3]=0
352         pmat[4,3]=0
353         pmat[5,3]=0
354         pmat=np.reshape(pmat, 24, order='F')
355         pdb.set_trace()
356         return pmat
```

Listing B.3: class Evaluator

Appendix C

Parameter for reported computational results

C.1 Cylindrical format

Parameter	Value
size event matrix	[32,20]
number of batches	400
learning rate	0.15
nkerns	[1,2]
filter 1	[3,3]
filter 2	[2,2]
maxpooling	[2,2]
nepochs	1

C.2 Cartesian format

Parameter	Value
size event matrix	[50,50]
number of batches	500
learning rate	0.00001
nkerns	[1,2]
filter 1	[3,3]
filter 2	[2,2]
maxpooling	[2,2]
nepochs	1

Appendix D

Calculation of significance scores

To evaluate the performance of the trained CNN we compute *significance scores* based on its performance on a representative test set. For this, we use

$$S/\sqrt{S+B} \tag{D.1}$$

$$S/\sqrt{B} \tag{D.2}$$

where S is the number of correctly identified signal events and B the false positives. Instead of the raw numbers $N \in \{S, B\}$, we use luminescence-scaled values:

$$N_{\text{lumi}} = \frac{N}{\sharp\text{events}} \cdot A \cdot L \tag{D.3}$$

with A cross section and L luminescence of the sample. For the parameters reported in [3], we get the following scaling law

$$N_{\text{lumi}} = C \cdot N \tag{D.4}$$

$$C = \frac{1}{\sharp\text{events}} \cdot A \cdot L \tag{D.5}$$

$$\tag{D.6}$$

with scaling constants

$$C(WZ - EW) = -0.00333 \tag{D.7}$$

$$C(WZ - QCD) = -0.0266 \tag{D.8}$$

List of Figures

List of Tables

Bibliography

[1] G. Aad et al., *Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC*, Physics Letters B **716** no. 1, (2012) 1 – 29.

[2] LIGO Scientific Collaboration and Virgo Collaboration Collaboration, B. P. Abbott et al., *Observation of Gravitational Waves from a Binary Black Hole Merger*, Phys. Rev. Lett. **116** (2016) 061102.

[3] A. Collaboration, *Measurements of $W^\pm Z$ production cross sections in pp collisions at $\sqrt{s} = 8$ TeV with the ATLAS detector and limits on anomalous gauge boson self-couplings*, Physical Review D **93** no. 9, (2016). arXiv: 1603.02151.

[4] Y. LeCun, Y. Bengio, and G. Hinton, *Deep learning*, Nature **521** no. 7553, (2015) 436–444.

[5] Y. LeCun et al., *Gradient-based learning applied to document recognition*, Proceedings of the IEEE **86(11)** (1998) 2278—-2324.

[6] D. Griffiths, *Introduction to Elementary Particle Physics*. Wiley-VCH, 2008. 2nd edition.

[7] M. Thomson, *Modern Particle Physics*. Cambridge University Press, 2013.

[8] M. U. Mozer, *Electroweak physics at the LHC*. Springer, 2016.

[9] LISA lab, *Deep Learning Tutorial*, Tech. Rep. Release 0.1, University of Montreal, 2015.

[10] P. W. Higgs, *Broken Symmetries and the Masses of Gauge Bosons*, Phys. Rev. Lett. **13** (1964) 508–509.

[11] F. Englert and R. Brout, *Broken Symmetry and the Mass of Gauge Vector Mesons*, Phys. Rev. Lett. **13** (1964) 321–323.

[12] C. R. H. G. S. Guralnik and T. W. B. Kibble, *Global Conservation Laws and Massless Particles*, Physical Review Letters **13** (1964) 585–587.

[13] ATLAS, CMS Collaboration, A. Holzner, *Beyond standard model Higgs physics: prospects for the High Luminosity LHC*, arXiv:1411.0322 [hep-ex].

[14] ATLAS Collaboration, G. Aad et al., *The ATLAS Experiment at the CERN Large Hadron Collider*, J. Instrum. **3** (2008) S08003. 437 p.

[15] D. H. Hubel and T. N. Wiesel, *Receptive fields and functional architecture of monkey striate cortex*, The Journal of Physiology **195** no. 1, (1968) 215–243.

[16] Theano Development Team Collaboration, R. Al-Rfou et al., *Theano: A Python framework for fast computation of mathematical expressions*, arXiv e-prints **abs/1605.02688** (2016). http://arxiv.org/abs/1605.02688.

[17] Y. LeCun and C. Cortes, *MNIST handwritten digit database*,. http://yann.lecun.com/exdb/mnist/.

[18] F. Socher, *Study of Electroweak Gauge Boson Scattering in the WZ Channel at the ATLAS Detector at the Large Hadron Collider*. PhD thesis, TU Dresden, 9, 2016. to be published.

[19] S. Krebs. Personal communication.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Melanie Weber
Princeton (NJ, USA), den 30. September 2016