technische universität
dortmund

Master Thesis

Deep Learning on Raw Telescope Data

Stefan Rötner

November 2017

Gutachter:
Prof. Dr. Katharina Morik
Dr. Christian Bockermann

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Künstliche Intelligenz (LS-8)
http://www-ai.cs.tu-dortmund.de

Contents

1 | Introduction

1.1 | Motivation

Experiments in high-energy astroparticle physics like MAGIC or FACT [26, 2] produce a continuous stream of high-volume data. The processing steps from raw data recorded by the telescope to high-level prediction tasks can be realized using the `fact-tools` library. Machine Learning algorithms are employed for solving these high level prediction tasks based on preprocessing steps and features which have been hand-designed by human experts, while the original raw data is no longer considered for classification.

Due to the efficient computation on Graphical Processing Units (GPU) it is possible to learn Deep Neural Networks with millions of weights on massive datasets of raw data resulting in breakthrough performance on a couple of benchmark datasets, e.g. [19]. Therefore the time has come to evaluate the potential benefits that Deep Learning could bring to processing the data recorded by the FACT telescope.

Deep Convolutional Neural Networks (CNNs) excel at classifying image data which closely resembles the video-like data produced by the telescope. Representation learning may provide an efficient alternative to the features manually designed by human experts and a compressed representation may even help with regard to memory requirements and network transfer costs which are the main constraints on processing the data. While training of neural networks is computationally expensive, prediction is fast and can be performed online favoring real-time processing.

Applying CNNs to FACT data has to meet three main challenges: The high dimensionality of observed events, organization of the camera as hexagonal grid opposed to existing convolution operations for square grids and the additional time dimension.
CNNs answer the challenge of high dimensional inputs by parameter sharing. A num-

ber of simpler learning tasks can be identified by treating the spatial, temporal and spatio-temporal aspect of the data separately.

2 | The First G-APD Cherenkov Telescope (FACT)

Modern astronomy studies celestial objects by observing high-energy beams emitted by these sources, e.g the energy plotted over time results in a *light curve* that can be used to classify the source. High-energy beams can not be observed directly, but particles interact with elements in the atmosphere inducing cascading air showers which emit *Cherenkov light* that can be measured by the FACT telescope. To classify a source the energy of a gamma particle needs to be reconstructed from these camera measurements.

The camera consists of 1440 hexagonal pixels which are sampled at a rate of 2GHz. If a hardware trigger detects an *event*, a series of 300 camera samples (the *region of interest*) is written out. Raw data has dimension 1440×300 and the 1440 pixels can be arranged in two dimensions according to their spatial position in the camera.

Obviously, the vast amount of events detected by the telescope is unlabeled. To solve the different tasks using supervised machine learning methods labeled data is required for training. The interaction of particles with elements in the atmosphere is well understood and monte carlo simulation of the probabilistic collisions together with simulation of the FACT camera can be used to simulate labeled raw data. The simulation is computationally expensive and therefore the amount of labeled examples available for training is limited.

2.1 | Processing Pipeline of fact-tools

The following processing steps are applied to the raw data in order to obtain the final energy estimate (see Figure 2.1):

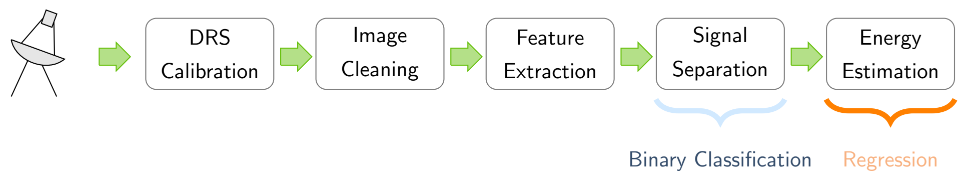

Figure 2.1: Data processing steps from raw data acquisition to energy estimation in fact-tools adapted from [6]

1. *DRS-Calibration*: The pixel values are adapted in order to correct for environment parameters like temperature or small differences in the physical properties of the hardware used to trigger each pixel.

2. *Image Cleaning*: Not all the photons reaching the camera stem from the observed shower. To remove the noisy influence of background light the image cleaning step explicitly determines the pixels that belong to the shower. The implemented cleaning methods estimate the number of photons per pixel and the arrival time (indicated by a rising edge in the 300 value time series of the event) from raw data and use thresholds on the number of photons for each pixel and its neighboring pixels to determine the shower pixels[34].

3. *Feature Extraction*: A number of hand-designed features is extracted based on the shower pixels. The most important parameters are the geometric Hillas-parameters[14] that are based on fitting an ellipse to the shower pixels. Figure 2.2 visualizes the detected shower pixels and the fitted ellipse.

4. *Signal Separation*: The telescope triggers not only for the desired gamma events but also for background noise produced from hadronic rays. These hadron events outnumber the gamma events by a factor of 1000 to 10.000 and as a conclusion a lot of unnecessary data will be collected to measure a single gamma event. Therefor it is an important step to separate the signal from noise. The underlying binary classification task is typically addressed by random forests operating on Hillas-parameters.

5. *Energy Estimation*: The final step is to estimate the energy of the gamma particle from a number of energy-related features, which can be interpreted as a regression task from a machine-learning perspective.

2.2 | Preview on learning tasks

A visualization of one time slice of a single event can be seen in Figure 2.2. The color of a camera pixel gives the voltages measured by the telescope which should correspond to the amount of photons that hit that pixel. The brighter pixels form a photon shower.

The final prediction task we are interested in is to perform a binary classification of the event class from a sequence of these images (i.e three dimensional data).

In addition to the real three dimensional data we have access to the information whether a pixel is actually part of the shower induced by the observed event. From this information we can derive a supervised image cleaning tasks, that will try to answer for each pixel if it actually is a shower pixel.

Image Cleaning can be realized as a supervised learning task by extracting the ground truth information whether a pixel is a shower pixel or not from meta attributes of the Monte Carlo Simulation (see Figure 2.2).

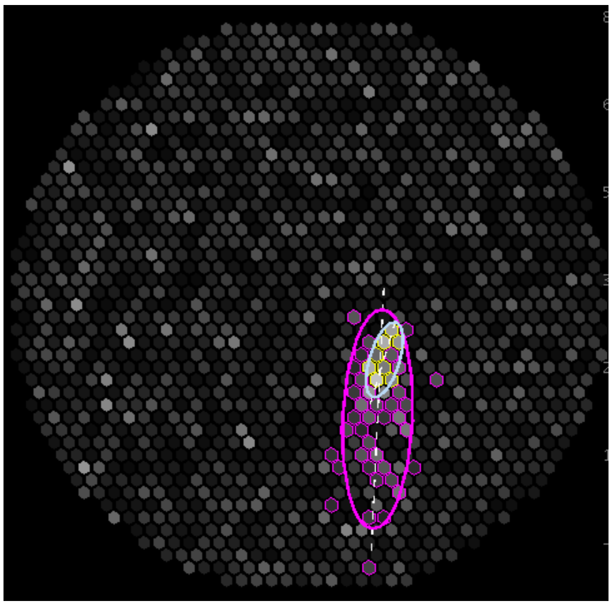

Figure 2.2: Comparison of the shower pixels derived from meta attributes of the simulation (pink) in comparison to the shower pixels derived using Two Level Time Neighbor Cleaning (gray) together with the fitted ellipses from which the Hillas-parameters are extracted. The intersecting pixels are displayed in yellow. Note that the ellipses differ not only in size but also in the angle of the main axis.

Image cleaning performance can be evaluated with the typical classification performance measures based on a contingency table for counting the true and false detection of shower pixels.

A basic supervised *pixel cleaning* task can be performed as binary classification task that decides for each pixel given its values over the complete region of interest, whether the pixel belongs to the shower or not. A more advanced setting tries to predict a cleaned image from the raw data as a whole.

Throughout this thesis problems that operate on inputs of different dimensionality and therefore tackle either temporal, spatial or spatio-temporal aspects of the data are distinguished as *tasks* which can be applied to different *data sets*, e.g. real data vs. ground truth photon charges.

3 | Deep Learning

3.1 | Overview

The buzzword *Deep Learning* is used to describe a trending topic in machine learning which has also gained a lot of public attention due to its role in solving a number of spectacular tasks, e.g. *AlphaGO* [28] beating the human GO champion in 2016. Deep Learning refers to the training of Artificial Neural Networks(ANN) with many hidden layers. Multi-layer ANNs have been developed in the 1970s and theoretical works imply that a universal function approximator can be realized with only one hidden layer. However, more layers allow to learn more complex functions and therefore often help to improve performance. The increasing amount of available data helps to prevent models with a huge number of weights from over-fitting and processing on modern Graphical Processing Units (GPU) allows to train models with many parameters efficiently.

Prior to explaining the mathematics behind Neural Networks and Convolutional Neural Networks, this section presents an overview of some of the other aspects of Deep Learning and explains how they relate to the processing steps of the *fact-tools* pipeline.

Representation Learning The idea of representation learning is to automatically find a representation that captures the internal structure of the data and is well suited for classification [4]. Usually the last layer of a neural net corresponds to very high level features that can be used for solving a specific prediction task, while the first layers capture basic properties of the underlying data distribution, e.g. edges or colors for image data. The activations of a hidden layer can therefore be interpreted as a learned representation of the data that is well suited for prediction. An *Autoencoder* is a special form of neural network for unsupervised learning of a compressed representation. The Autoencoder tries to reconstruct its input using one ore more hidden layers. If the number of neurons at the hidden layer is much

smaller than the number of input neurons this corresponds to a representation of reduced dimension.
Applied to FACT data a learned representation can provide an alternative (to the hand designed features which are extracted using fact-tools) that possibly improves prediction performance or speeds up the feature extraction. A compressed representation might help with regard to memory requirements and network transfer costs. Sometimes the Autoencoder is set up to predict not its exact input but a version of the input that is cleaned from noise (*Denoised Autoencoder*). Such a Denoised Autoencoder could be used to address the image cleaning task in the FACT pipeline.

Transfer Learning Transfer learning aims at incorporating an existing model in order to improve a prediction task that operates on the same input domain but with a different target variable. Applied to neural networks one could reuse the lower layers of the existing model (that are supposed to capture the underlying structure of the available data) and replace the top layer in order lo learn high level features that are more suited for predicting the new target [36]. Often it is useful to fine-tune the existing weights to better match the new task.
As we have seen in the previous section, a number of different learning tasks has to be solved based on the raw telescope data. Obviously it would be nice to reuse some of the extracted *features* for the remaining learning tasks.

Unsupervised Pretraining In many cases only a small set of labeled data but a huge amount of unlabeled data is available. In this case it can be beneficial to use representation learning on the unlabeled data and apply transfer learning to solve the supervised learning task. During pretraining the weights can be efficiently computed in a greedy layer-wise fashion [5]. Unsupervised pretraining perfectly matches the situation of vast amounts of real but unlabeled data measured by the telescope in comparison to computation intensive simulation of labeled data.

Recurrent Neural Networks Recurrent Neural Networks maintain an internal state by feeding some activations as input to neurons of lower layers at the next time step. This form of internal memory allows RNNs to operate efficiently on sequences or time series. The region of interest for a single camera pixel typically consists of 300 time slices and can therefore be interpreted as a time series.

3.2 | Artificial Neural Networks

Machine Learning can be divided into the three subfields *Supervised Learning*, *Unsupervised Learning* and *Reinforcement Learning*. Supervised Learning deals with problems in which a target variable $y \in \mathcal{Y}$ is assigned to an observation $\vec{x} \in \mathcal{X}$ using a function $f^* : \mathcal{X} \rightarrow \mathcal{Y}$. The central idea is to use a model $f_\theta(\vec{x})$ and determine the trainable parameter vector $\theta \in \Theta$ [1] in a way that $f_\theta \approx f^*$.

Gamma-hadron separation and image cleaning are instances of binary classification in which a binary class value is assigned to a vector of real-valued photon charges, i.e $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \{0, 1\}$.

One possible model choice is a linear function $f_\theta(\vec{x}) = f_{(\vec{w},b)}(\vec{x}) = \vec{w}^T \cdot \vec{x} + b$. For example the Perceptron[21, 27, 23] is a linear model for binary classification:

$$\hat{f}_\theta(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w}^T \cdot \vec{x} \geq b \\ 0 & \text{else} \end{cases}$$

The perceptron will only provide good predictions if the data is indeed linear separable (see figure 3.1).

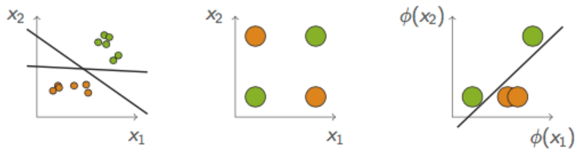

Figure 3.1: Perceptron on data that is linear separable (left), not linear separable (middle) and linear separable after transformation (right).

Otherwise a non linear transformation $\Phi : \mathbb{R}^d \rightarrow \mathbb{R}^{d'}$ of the inputs into a linear separable representation is required to formulate a model that is capable of predicting the data $\hat{f}'_\theta(\vec{x}) = \vec{w}^T \cdot \Phi(\vec{x})$.

The transformation can be explicitly designed by human experts, e.g the processing steps in the `fact-tools` pipeline prior to the actual classification using Random Forests corresponds to hand designed feature transformations. The alternative approach is to

[1]The vector arrow is discarded for better readability

learn a suitable representation from the raw data over the course of training. Assuming the feature transformation in turn to be a linear model

$$\Phi'_\theta(\vec{x}) = \begin{bmatrix} \phi_1(\vec{x}) \\ \phi_2(\vec{x}) \\ \vdots \\ \phi_n(\vec{x}) \end{bmatrix} = \begin{bmatrix} \phi_1(\vec{w_1}^T \cdot \vec{x} + b_1) \\ \phi_2(\vec{w_2}^T \cdot \vec{x} + b_2) \\ \vdots \\ \phi_n(\vec{w_n}^T \cdot \vec{x} + b_n) \end{bmatrix} \tag{3.1}$$

defines a hierarchy of linear models, e.g.

$$\hat{f}''_\theta(\vec{x}) = \vec{w}^T \cdot \Phi^1_\theta(\Phi^2_\theta(\vec{x})) + b$$

whose combined parameters θ can be learned during training.

Obiously the hierarchical model can be realized as combination of multiple perceptron models, where the inputs $\vec{x}$ are fed into different perceptrons whose outputs in turn are fed into another perceptron. However a linear combination of linear models is necessarily a linear model and the expressive power of the resulting model still limited. The thresholding after each neuron is not differentiable which can be a problem with regard to gradient based optimization during parameter learning. Both problems can be addressed by exchanging the thresholding for a differentiable non-linear activation function $h : \mathbb{R} \to \mathbb{R}$, e.g the sigmoid function $h(z) = \sigma(z) = \frac{1}{1+exp(-z)}$.

A computation unit that computes

$$\tilde{f}_{(w,b)}(\vec{x}) = h(\vec{w}^T \cdot \vec{x} + b)) \tag{1}$$

is called a *neuron* and is the fundamental building block of a Multi-Layer-Perceptron (MLP). A MLP is build from multiple neurons that are organized in fully-connected layers and can be visualized as an acyclic graph (see figure 3.2). The first layer is called *input layer* and represents the raw input values $\vec{x}$ where each neuron of the layer corresponds to one entry of $\vec{x}$. Neurons in the input layer perform no mathematical operation but simply provide the input value as output to be processed in the next layer. The last layer is named *output layer* and typically has one neuron per class, that will output the final score for this class. However the perceptron example shows that one output neuron can be enough for binary classification. In between any number of layers containing any number of neurons is possible. Fully-connected means that the output of each neuron of a layer is fed as an input to each neuron of the next layer. We can enumerate the layers from left to right and use M^l for the number of neurons in layer l and assign the weight

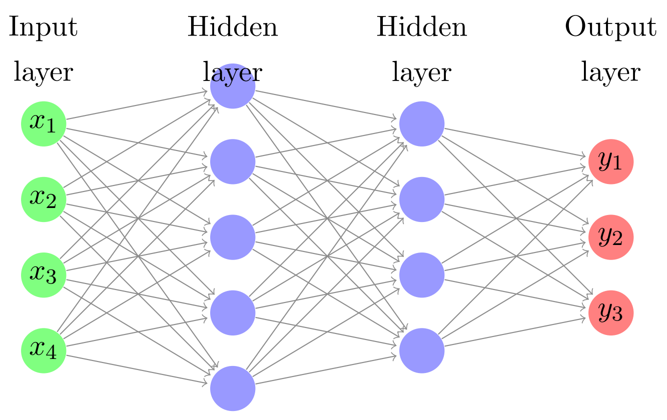

Figure 3.2: Multi-Layer Perceptron.

$w^l_{i,j}$ to the edge connecting neuron i of layer $l-1$ to neuron j of layer l. Formally neuron j in layer l computes its output value using equation (1) to

$$f^l_j = h(\sum_{i=1}^{M^{(l-1)}} w^l_{i,j} \cdot f^{l-1}_i + b^l_j) \tag{2}$$

The computation can be realized in two steps:

$$y^l_j = \sum_{i=1}^{M^{(l-1)}} w^l_{i,j} \cdot f^{l-1}_i + b^l_j \tag{3}$$

$$f^l_j = h(y^l_j) \tag{4}$$

Theoretical work proves that one hidden layer is enough to realize an universal function approximator [15]. However practice shows that increasing the number of layers can drastically improve the approximation quality [33]. Networks with more than three hidden layers are referred to as *Deep Neural Networks* [3].

For learning the parameters of the model a loss function is used to define how much a prediction for an example differs from its actual class:

$$l : \Theta \to \mathbb{R}, \text{ e.g. } E^p_\theta(\vec{x}, y, \theta) = \frac{1}{2}(f_\theta(\vec{x}) - f^*(\vec{x}))^2$$

The averaged loss over the complete training data $\mathcal{D} = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_N, y_N)\}$

$$E = \frac{1}{N}\sum_{i=1}^{N} E^p_\theta(\vec{x}_i, y_i, \theta)$$

is minimized to determine the optimal model parameters:

$$\theta = \arg\min_{\theta\in\Theta} \frac{1}{N}\sum_{i=1}^{N} E^p(\vec{x}_i, y_i, \theta)$$

The actual learning is realized using gradient descent, an iterative method in which the weights are adapted in negative gradient direction (i.e. the direction of steepest descent) by a small step size $\alpha \in \mathbb{R}$:

$$\theta_{i+1} = \theta_i - \alpha \cdot \frac{\partial E}{\partial \theta}$$

Computing the loss and update step over the entire training set is referred to as *batch training*. In praxis Stochastic Gradient Descent [7] yields better learning speed by computing an update step for each example, where the example is randomly drawn from the training dataset. Today most Deep Learning models are trained using mini-batches of the data. The gradient for each update step is averaged over small batch of samples, which can result in more robust gradient estimates. With traditional stochastic gradient descent the learning rate is chosen to be large in the beginning and decreased during training. However there exist a number of optimization techniques for automatic learning rate adaptation (see section 5.3.2).

Using the two step processing for each neuron described in equations (3) and (4) we can compute the required gradients in a straightforward way - if the gradient $\frac{\partial E^p}{\partial f_j^l}$ is known - using the chain rule:

$$\frac{\partial E^p}{\partial y_j^l} = \frac{\partial E^p}{\partial f_j^l} h'(y_j^l) \tag{5}$$

$$\frac{\partial E^p}{\partial w_{i,j}^l} = \frac{\partial E^p}{\partial y_j^l} \frac{\partial y_j^l}{\partial w_{i,j}^l} = \frac{\partial E^p}{\partial y_j^l} \cdot f_i^{(l-1)} \tag{6}$$

$$\frac{\partial E^p}{\partial f_i^{(l-1)}} = \frac{\partial E^p}{\partial y_j^l} \frac{\partial y_j^l}{\partial f_i^{(l-1)}} = \frac{\partial E^p}{\partial y_j^l} \cdot \sum_{j=1}^{M^l} w_{i,j}^l \tag{7}$$

Note that the gradient computation in equation (7) is not computed for a parameter update but rather to ensure that the required gradient will be available during gradient computation of the previous layer if computation starts from the highest layer downwards. This leads immediately to the dynamic programming based recursion of the famous backpropagation algorithm. The principal that each unit computes the gradients and passes the gradient with regard to its inputs to previous neurons is used in modern deep learning frameworks like `Tensorflow` by creating a computational graph from building blocks that are designed this way to realize automatic differentiation.

3.3 | Convolutional Neural Networks

CNNs make use of symmetries in the data by using local connectivity and weight sharing to reduce the number of parameters and connections in comparison to fully-connected MLPs described in the previous section. Therefore two new layer types - the convolution and the pooling layer - are introduced.

The first successful application of convolutional neural networks [20] stems from the 1990s, while the first application of *deep* convolutional neural networks in 2012 [19] achieved breakthrough performance in the ImageNet challenge. Many adaptations to the net architecture have led to further improvements in recent years [38, 32, 31, 29, 13]. Detailed descriptions of the convolution operation and efficient training of CNNs can be found in [12, 17].

The name is inspired by the mathematical convolution operation which describes the convolution of two continuous functions $s(t) = \int x(a)w(t-a)da$. However representations that can be stored in memory consist of measurements from discrete points in time and are of limited size. In addition the indices are oftentimes flipped for easier computation on arrays. Therefore the operation actually realized by convolutional neural networks is rather a *cross-correlation* than a *convolution* [12]. The formulation for the default use case of image processing, i.e. two dimensional data, the operation is:

$$S(i,j) = (I * K)(i,j) = \sum_{m=1}^{M} \sum_{n=1}^{N} I(i+m, j+n)K(m,n)$$

A filter $K_{M \times N}$ is convoluted with input I. Intuitively the filter of little spatial extension compared to the original input is moved across the input and for each position a dot product of values covered by the filter and the filter weights is computed (see Figure 3.3).

Compared to fully-connected MLPs the neurons of each layer are ordered in multiple dimensions. In case of 2D-Convolution the neurons are organized in 3D-shape (width, height, channels). The channel axis corresponds to multiple features that exist for each spatial position, e.g. image processing on RGB images has channel size 3 on the input layer (see Figure 3.4).

The convolution operations always involves the full depth at each spatial location (see Figure 3.5). For filter size 3×3 and 4 input channels at each filter position the dot product of 36 input values and 36 weights is calculated in order to obtain a single output

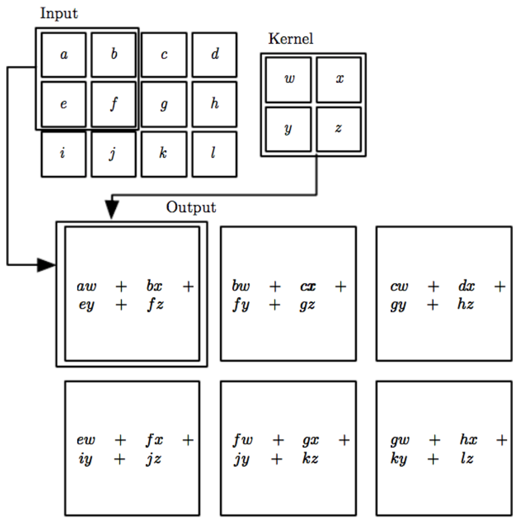

Figure 3.3: Example of 2D-Convolution-Operation for a single 3×3-Kernel with only one channel of input data. The filter is moved across the window, producing an output of reduced spatial extend. [12]

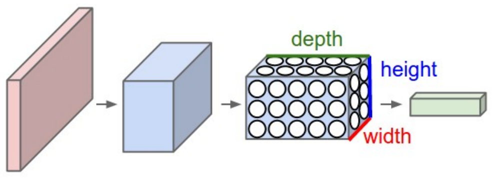

Figure 3.4: A 2D-Convolutional Neural Network processes three dimensional volumes of decreasing spatial extend. [17]

value. The spatial extension of the outputs therefore corresponds to the number of input positions at which a filter is applied. Each convolution layer applies not only a single filter but multiple filters whose two dimensional outputs are stacked resulting in one output channel per applied filter.

The hyper parameters of a convolution layer are the number of filters K, kernel size F, stride S and padding P. The stride describes how far the kernel is moved after each application. The number of positions for filter applications depends on stride and kernel size. Often the positions defined by stride and kernel size do not fit the input shape perfectly and adding zeros around the borders (*padding with zeros*) can help to cover the border of the input with filter positions. Restricting the output to positions for which the filter is entirely inside the input dimension and discarding possible border elements is often referred to as *valid padding*. Choosing the hyper parameters in a meaningful way

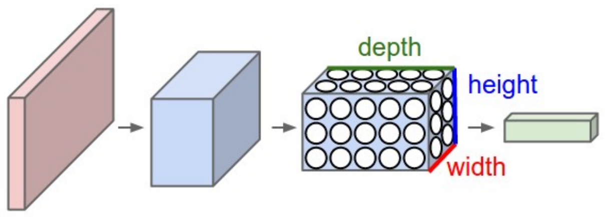

Figure 3.5: Example of 2D-Convolution-Operation for 3×3 convolution on true 3D tensors, i.e. multiple input channel and multiple filters. [17]

can be challenging. For input shape $W_1 \times H_1 \times D_1$ the output shape $W_2 \times H_2 \times D_2$ can be derived from the hyper parameters using the following relationships [17]:

$$W_2 = (W_1 - F + 2P)/S + 1$$

$$H_2 = (H_1 - F + 2P)/S + 1$$

$$D_2 = K$$

Each filter is defined using one weight per entry plus one bias weight resulting in a total of $(F \cdot F \cdot D_1) \cdot K + K$ weights for the entire layer.

In other words each output neuron corresponds to one combination of filter and filter position and depends only on the input values covered by the filter at this position. In terms of the network representation as acyclic graph, each neuron is only connected to a few input neurons. Compared to the fully-connected MLP, CNNs are *locally connected.* Further the weights of each filter are *shared* across all filter positions. Fully connected layers with m inputs and n outputs require $m \times n$ parameters, while convolution layers with a total of k entries per filter rely on only $k \times n$ parameters. Accordingly calculating the dot products for fully-connected layers requires $O(m \times n)$ computations while the calculation of dot products for a convolution layer requires only $O(n \times k)$ computations.

Weight sharing and local connectivity therefore help to reduce the runtime for high dimensional inputs. More important parameter sharing reduces the number of parameters, while each parameter is presented with more data (filter positions $\times$ number of examples) resulting in robuster statistical estimates of the true gradient and allowing to fit networks with a large number of parameters on comparably few training examples.

Sharing parameters across applications to different input regions requires a certain symmetry in the data in a way that specific local features can be observed at any position. The important property of CNNs realized by translating the filter across the input is *equivariance to translation*, i.e. the translating the input prior to convolution yields the same result as applying the same translation on the output. Formally function f is equivariant to function g if $f(g(x)) = g(f(x))$. It is important to check whether this property holds for a specific data set prior to using CNNs [12].

The second operation introduced by CNNs is *pooling*. Pooling layers are added for down-sampling the data and have no trainable parameters. Pooling windows similar to the filters of convolution layers are translated across the input. The output for each application is a simple aggregation of input values covered by the pooling window. Commonly used aggregation functions are average and max.

A CNN architectures typically consists of alternating convolution and pooling layers followed by two or three fully connected layers. Pooling layers require no activation function either and oftentimes the activation function is represented as an actual layer in between th convolution and pooling layers.

For an intuitive understanding of CNNs filters can be visualized and interpreted. For image recognition problems the filters of lower layer represent simple concepts like lines with a certain orientation or colors. Filters in higher layers represent complex features that are obtained as a combination of different simple feature from lower layers.

The raw data produced by the telescope closely resembles standard image data and the detected showers differ in angle and size but the final prediction is invariant to the actual position of the shower on the camera. The handcrafted image cleaning explicitly inspects the neighbor pixels. This perfectly fits the properties of CNNs described above.

Due to the hexagonal structure of the camera a transformation is necessary in order to realize the convolution layer as an efficient matrix multiplication and to keep the symmetries required for weight sharing.

4 | Convolution on Hexagonal Grids

CNNs have originally been developed in the context of image processing and generally operate on arrays, that are indexed by a square grid. A number of popular deep learning frameworks and libraries provide efficient implementations of the convolution operation by utilizing the GPU. However these implementations operate on data that is sampled on a square grid.

Due to the hexagonal shape of the individual camera pixels the camera surface of FACT is sampled on a hexagonal grid. Therefore it is important to verify if and how the convolution operation can be implemented or existing frameworks can be utilized. The first possibility is to find a representation of the hexagonal grid as two dimensional array in order to apply standard convolution operations designed for square grids. The second possibility is to define a custom convolution operation and implement it from scratch using efficient GPU computations.

4.1 | Using Square Grid Implementations

A formal introduction of indexing schemes and computations on hexagonal grids can be found in [22], while [25] provides an implementation focused view on hexagonal grids in combination with animated visualizations.

For a square grid on the one hand the obvious way of indexing is to use the two axes of translational symmetry. Hexagonal grids on the other hand have three axes of symmetry that could be used for indexing.

Cube coordinates take into account all three axis and index each pixel using a triple (x, y, z). In order to prevent redundancy the constraint $x + y + z = 0$ must hold. Cube coordinates are a popular choice for the implementation of algorithms on hexagonal grids, but provide little help in mapping values to a square grid.

Axial coordinates are obtained by using only two of the translational symmetry axes for indexing. The axes are skewed (i.e. their inner angle is 60° or 120° depending on the choice of axes) resulting in a diagonal indexing structure and consequently usually sparse arrays.

Offset coordinates are obtained by offsetting every second column (or row) (see figure 4.1). Offset coordinates are the straightforward way to represent hexagonal grids using square arrays and will be used in this thesis.

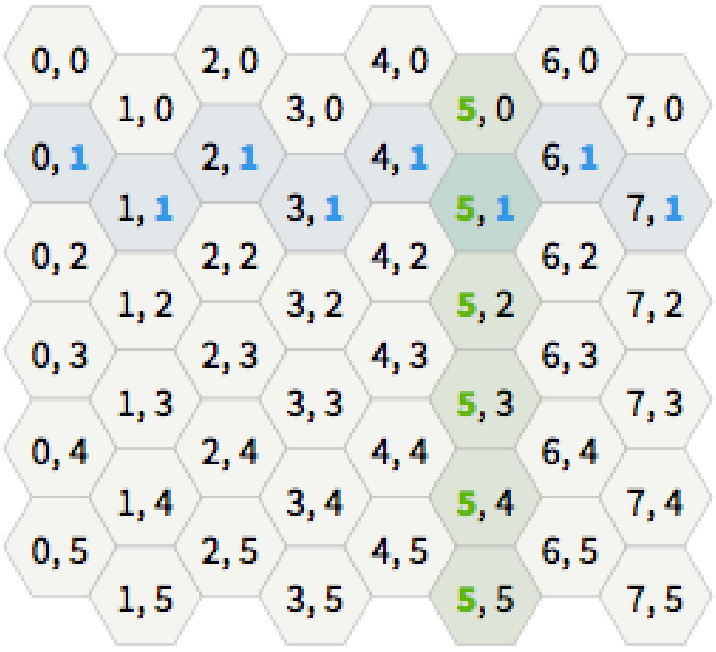

Figure 4.1: Offset coordinates for indexing a hexagonal grid [25]

While offset coordinates provide efficient storage of almost rectangular panes, the camera surface of FACT is of approximately circular or hexagonal shape. As a consequence a partly sparse array of dimension 45×40 is required for representing the surface in offset coordinates (the same applies for axial coordinates). In total 1800 values compared to the 1440 real camera pixels need to be stored and finally processed by the first convolution layer.

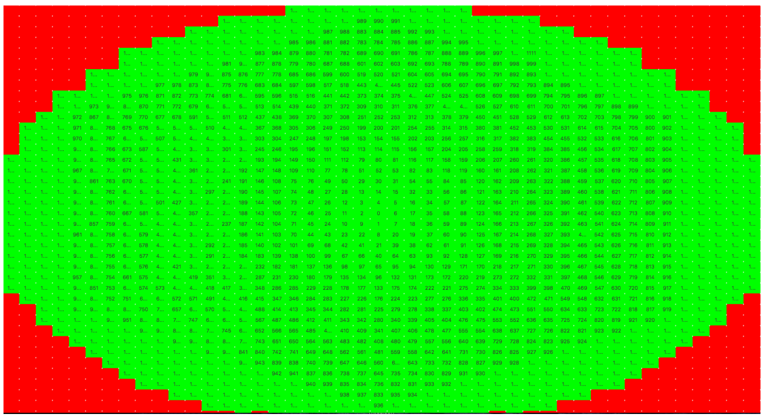

Figure 4.2: The form of the camera surface represented in offset coordinates does not perfectly fit a two dimensional array. Therefore the red pixels in the corners are missing camera measurements and are filled with zeros.

In order to capture a pixel and its six neighboring pixels in an offset coordinate array, at least filter size 3 × 3 is required and therefore two additional pixels are captured. Due to neighboring columns being offset the relative location of the original neighbors is mirrored for even and uneven columns (see figure 4.3). The property of equivariance to

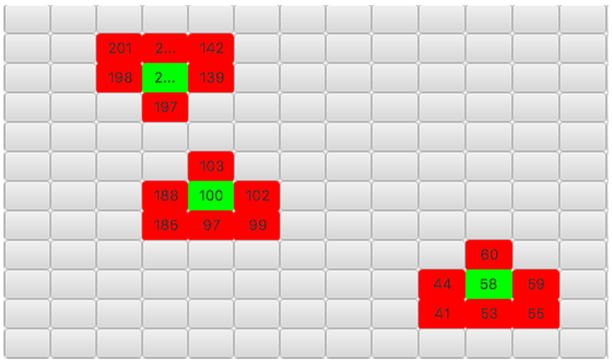

Figure 4.3: Three pixels (green) together with their neighboring pixels in the hexagonal grid (red) projected into and offset coordinate array.

translation of which CNNs make use is therefore not fulfilled. It will hold though if the image is translated by an even number of pixels. Accordingly even stride in horizontal direction has to be used to preserve the translational equivariance for the trained filters.

In order to understand how large neighborhoods are arranged in an offset coordinate array figure 4.4 shows the second order neighborhood of a pixel (i.e the neighbors and their neighbors).

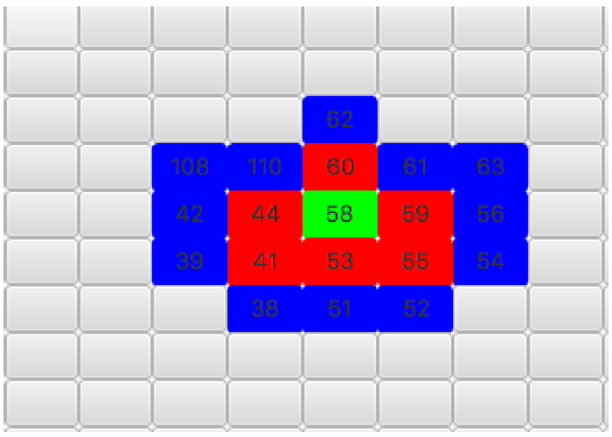

Figure 4.4: One pixel (green), its neighbors (red), and its second order neighbors (blue) in an offset coordinate array.

4.2 | True Hexagonal Convolution

Applying true hexagonal convolution requires the efficient selection of a pixel and its neighbors. The neighbors in offset coordinates can be easily computed by adding specific offsets to the pixel index. Again the offsets are different for even and uneven rows.

Efficient neighborhood computation is possible for most hexagonal indexing schemes and storage as two dimensional array is no longer required. Efficient computation can be realized by copying the the relevant patches of the input data into a matrix of shape *filter positions* × *number of neurons per filter* and the weights accordingly into a matrix of shape *number of neuron per filter* × *number filters* and computing all filter application in parallel using a single matrix product. The result matrix needs to be converted to the correct spatial arrangement.

Different possibilities for selecting the filter positions and applying *stride* to the hexagonal grid need to be defined. Figure 4.5 shows an example of overlapping convolution were a filter of 7 pixels covering exactly the neighbors is applied to every pixel of the input. The hexagonal shape of the input is similar to the actual camera surface and can be viewed as central hexagon surrounded by a number of rings of hexagons. If the filter is applied with *valid* padding, the output shape will be of the same hexagonal shape missing the outer ring. Therefore the the same convolution operation can be applied on the output shape.

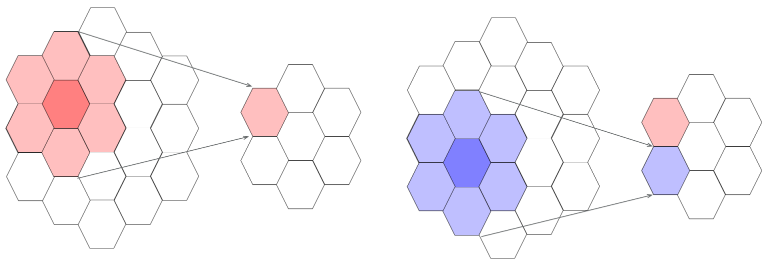

Figure 4.5: Overlapping convolution.

Figure 4.6 shows an example of non-overlapping hexagonal convolution. The input shape corresponds to an hierarchical representation of a hexagonal grid which can be defined recursively. Starting with a simple hexagon in level 0, each layer is constructed by using the area represented by the level below and adds 6 identical areas as neighbors (see Figure 4.6 from right to left). The HIP Framework [22] indexes this structure by assigning the numbers 0 to 6 the central and neighboring positions of each level in clockwise order. The complete address can be obtained by concatenating the index used to describe the relevant substructure at each layer. The result is a base 7 number. Using this indexing scheme structures can be stored as one dimensional array in a form that allows to convolute over true hexagonal neighborhoods using 1D-Convolution with filter size 7 and stride 7. Note that this works only for non-overlapping convolution.

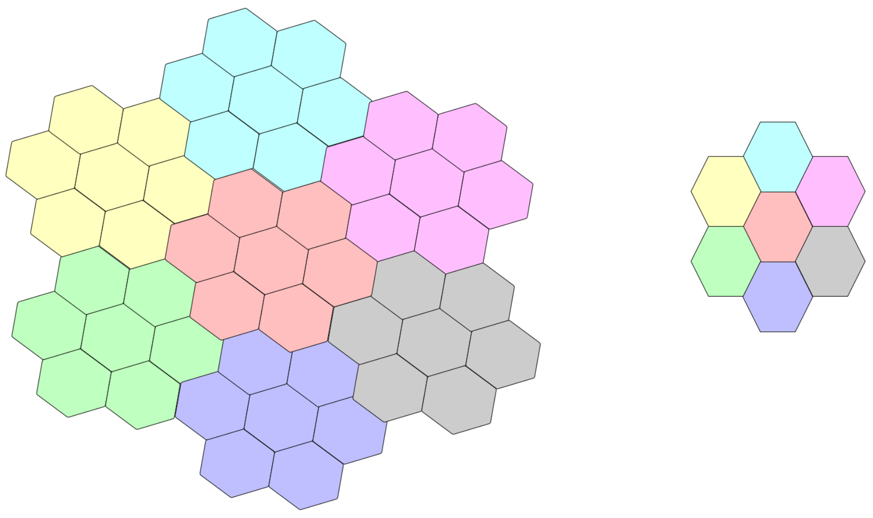

Figure 4.6: Non-overlapping convolution.

5 | Experiment Design

Prior to diving into the actual gamma-hadron separation and image cleaning tasks on telescope data in chapter 6, this chapter aims at providing some insight on the technical realization of the experiments. Further it guides through the careful preparation of a hierarchical data file that is suitable for deep learning and allows to realize different learning tasks on different datasets efficiently from a singe file.

5.1 | Choice of Framework

In order to efficiently evaluate different learning tasks and different approaches of meeting these tasks implementing every operation from scratch is a tedious exercise. Luckily several frameworks providing efficient implementations are available which make use of GPGPU computations. In context of the number of possible learning tasks it is important that the different building blocks can be combined in a flexible way.

The existing `fact-tools` pipeline is realized in `java` and there exist a few java libraries for deep learning e.g. H_2O[1] and `DeepLearning4J`[2]. These frameworks can be viable for integrating deep learning in the existing processing pipeline in production mode but provide little support for the rapid prototyping required for investigation of the best suitable approach for each of the learning tasks at hand.

`Caffee`[3] provides excellent support for efficient training of Convolutional Neural Networks but lacks support of recurrent neural networks, which might be investigated in the future with regard to the time dimension of the data.

[1] `https://www.h2o.ai`
[2] `https://deeplearning4j.org`
[3] `http://caffe.berkeleyvision.org`

Suitable choices are therefore `Theano`[4], `Tensorflow` [1] and `Torch`[5] of which Theano and Tensorflow come with the common frontend `Keras` [8], i.e. experiments defined in Keras can be executed in both frameworks. The experiments in this thesis are conducted with Keras using the Tensorflow backend. Keras is written in python and Tensorflow provides a python layer of abstraction to handle core functionalities implemented in C or CUDA.

5.2 | Data Preparation

The data formats currently used to store real and simulated data for processing with the java based fact-tools pipeline are far from ideal for training deep neural networks using python based frameworks like Keras or Tensorflow. When transforming data into a more suitable format it is important to consider aspects like sampling, shuffling, train-test-split, normalization and efficient feeding capabilities (into the training routines of the applied framework).

This chapter describes the data format used for the experiments in this thesis alongside the data transformations to convert from the existing data format into one favoring deep learning.

5.2.1 | Data Formats

The original FACT data is stored according to the Flexible Image Transport System (FITS) [35] and calibrated against external calibration files in the first processing step within fact-tools. The data is stored in a one dimensional array and ordered in a way that neighborhoods are not preserved, and therefore can not be used for spatial convolution using out of the box convolution operators. As a solution the data will be rearranged into an offset coordinate array with an additional time dimension.

For simulated data the event class (label) can be retrieved from meta parameters of the simulation. From the meta parameter *photon weights* we can infer the information whether a pixel was actually hit by a photon that originated from the observed shower during simulation (e.g. to use as label in the pixel cleaning process).

[4]`http://deeplearning.net/software/theano/`
[5]`http://torch.ch`

The necessary fields are serialized into a binary format using `Protobuf`[6]. Protobuf is developed by Google and requires the definition of a data format from which wrappers for reading and writing the binary format from within different programming languages can be compiled. The generated python wrappers allows to read the data into python but require a very intensive parsing step that proved impractical for feeding the data during experimentation.

The final solution is to write the data into a hierarchically organized HDF-file. The file is split into groups for training, validation and testing which each contain the same datasets *real data*, *photon weights*, *labels*, *binary photon weights*, a pixel based time series dataset and subsampled versions of the the pixel and real data. The order of examples within these datasets is the same and datasets for different learning task can be created in a flexible manner by feeding different (data,label) pairs to the training methods of Keras. E.g. (real data, labels) realizes the gamma-hadron separation, while (real data, photon weights) corresponds to the learning task of image cleaning.

Each dataset is organized in chunks of 100 examples. The chunks are attempted to be stored on disk as continuous block, which promises little I/O overhead during feeding if the data is read chunk by chunk (due to reduced reading head movement).

5.2.2 | Sampling and Data Splitting

Training of neural networks using stochastic gradient descent relies on random sampling from the training data. Keras provides methods for shuffling the data (or at least batches of data for HDF files). However due to the long training times of large networks cross validation is not feasible and therefore the data needs to be split into fixed datasets for training, validation and testing.

The data is stored in the binary Protobuf format in the order determined from sequentially reading the different FITS files. It is not clear if the evaluation on holdout data is more meaningful if the data stems from completely unseen FITS files or is drawn from the entirety of available examples. For the experiments in this thesis the latter approach is taken and a memory external shuffling is conducted by distributing the binary data randomly on different files (without parsing) and concatenating the files again.

[6]`https://developers.google.com/protocol-buffers/`

Right from the beginning the gamma and proton events are stored in two different files and during creation of the HDF file the data is combined from these files in a way that ensures equal label proportions for each chunk of data.

5.2.3 | Final Data

The resulting HDF file is suitable for feeding to experiments addressing different learning tasks in a flexible manner by combining different datasets. It is already shuffled and organized according to a train-test-validation split. The values of the real data have already been normalized to have zero mean and unit variance favoring deep learning.

The data contains 214.000 examples in the training sets, and 25.000 examples for validation and another 25.000 examples for testing. The ratio of hadron to gamma events is $1 : 1.2885$, the ratio of shower-pixels to non-shower pixels in the pixel based datasets is $1 : 24.43$.

5.2.4 | Model Builder

In order to systematically evaluate hyper parameters and design choices a model builder has been implemented which allows to run the same model multiple times while setting all the relevant hyper parameters to different values in (possibly nested) loops. With regard to interpreting the experiments in subsequent sections it is important to understand how the network architecture is parameterized within the model builder. A convolution layer is defined by the number of filters, kernel size, stride and padding. A pooling layer is characterized by the pooling operation (e.g max, average or none) and pool size. Fully connected layers are characterized by the number of neurons in each layer. A complete network architecture can be characterized using lists of values for each of these hyper parameters where the i^{th} entry in the list is used for the i^{th} layer in the network. If only one value is provided instead of a list, the value will be used for all layers. For symmetrical kernels, strides or pool sizes a single number can be given, e.g kernel size $= 2$ corresponds to a 2×2 convolution for two dimensional models.

If default values for each parameter are implied a 2D-CNN architecture M can be defined as:

$$
\begin{aligned}
M = \{&\text{kernel size} = [2, 3, (1, 2)] \\
&\text{strides} = 3 \\
&\text{neuron numbers} = [64, 32] \\
&\text{pooling} = \text{none} \\
\}&
\end{aligned}
$$

M has three convolution layers with stride 3×3 each and no pooling layers. The convolution layers are followed by two fully connected layers (with 64 and 32 neurons). The first convolution layer has kernel size 2×2, the third 1×2. For an example of model definition and matching architecture visualization see model M_6 in section 6.2.

5.3 | Training Convolutional Neural Networks

Training deep neural networks in general and CNNs in specific involves a lot of design choices and hyper parameter optimization methods. This chapter explains the general process of model training but also introduces techniques for improved learning and design choices that have not been addressed in the theory sections. If possible the specific hyperparameter is presented together with empirical evidence based on example plots from the actual hyperparameter optimization process for model M_6 from 6.2.

5.3.1 | Performance Measures

The standard approach for performance measurement in classification tasks is reporting error rates or accuracy scores. However in the presence of imbalanced classes accuracy is oftentimes a weak measurement because always predicting the majority class can provide better accuracy than really trying to separate the data. Therefore a careful investigation of precision and recall for each class can help to choose a meaningful model.

One approach to correct for the class imbalance is to use class weighting or equivalently instance weighting where each instance is assigned a weight based on its class frequency compared to the majority class. By assigning a bigger weight to rare classes during backpropagation the model is likely to predict all classes equally well. The same weighting mechanism of examples during accuracy prediction can approximate the accuracy that would be achieved on a balanced test set.

By using the binary cross entropy loss function the model is designed to output class probabilities instead of simple classification decisions. A simple solution is to use thresholding at 0.5 to obtain the final prediction. However especially in the presence of class imbalance choosing another threshold can be a better choice with regard to recall or precision. Two methods for choosing a meaningful threshold are *Receiver Operation Characteristic (ROC)* and the *Prediction-Recall Curve*. Every predicted class score on a validation set can be evaluated as threshold for obtaining the final prediction, resulting in specific (precision, recall) and (false positive rate, true positive rate) pairs which can be plotted against each other.

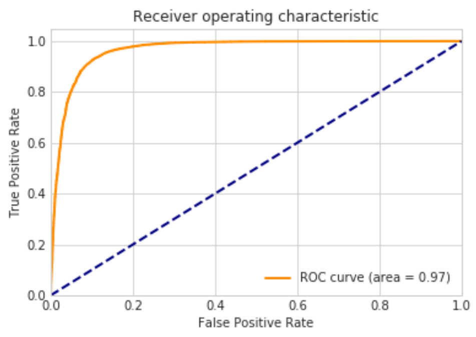

Figure 5.1: The ROC Curve plots the true positive rate against the false positive rate in order to obtain an optimal threshold for the decision function. The best possible performance is represented by the point in the top left corner. The blue line indicates the minimum performance which can be achieved by guessing.

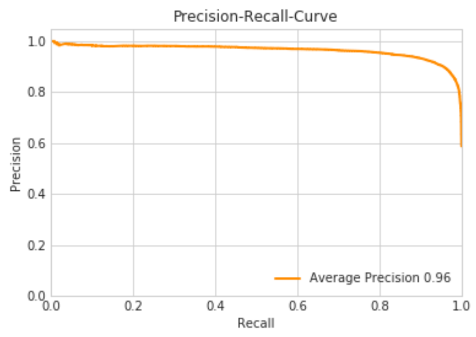

Figure 5.2: The Prediction-Recall Curve plots the precision against the recall in order to obtain an optimal threshold for the decision function. The optimal model performance is represented by the point in the top right corner.

The distribution of the observed particles in the real world is not exactly known and the simulation process adapted to real world observations from time to time. In praxis it is not important to label every observed event but more important to have a high confidence for the selected events. Therefore it is common to apply confidence cuts to discard examples for which the classifier is unsure. From this perspective it is preferable to determine the final threshold as late as possible during the training process. A single value for estimating how well a model can be fine-tuned to the desired specification is the *Area Under The Curve* (AUC) for the ROC curve. The Precision-Recall Curve is not linearly interpolatable, but the *Average Precision* can be computed as average over the precision scores weighted by the gain in recall compared to the last threshold [11].

Average Precision can provide a single value indicating the tradeoff of precision and recall under different thresholds.

The prediction quality of models can be evaluated and compared in terms of Average Precision, area under the ROC curve and weighted accuracy.

5.3.2 | Hyper Parameter Optimization

Hyper parameters of neural networks can be divided into design choices of the network architecture and parameters of the general training process. The different hyper parameters depend on each other. However an extensive grid search over the parameter space in not feasible due to the large number of parameters and long training times of the networks. In general it is advisable to use random search over the hyper parameter space and evaluate different choices on a small number of epochs only and then gradually increase the number of epochs during training for promising parameter combinations. This section describes hyperparameter related to learning, while different architectures are evaluated in section 6.2

Regularization

During Optimization we try to minimize the loss function over the training set, i.e. minimize the empirical risk which is only an estimate of the prediction quality regarding the real unknown distribution of the data. Neural networks are very vulnerable to overfitting the training set. In order to prevent the model from overfitting a number of techniques exist.

By constantly monitoring the classification performance on a validation set we can detect if the model starts overfitting and either reduce the learning rate or stop training completely (*early stopping*). In addition by saving the model in regular intervals or every time the validation performance improves it is possible to return to a state of training prior to overfitting.

Techniques like weight regularization, i.e. adding a penalizing term for large or dense weight vectors to the loss function can in some cases prevent the network from overfitting. However this effect has been unnoticable in the correspondig experiments on FACT data.

Essential to training deep neural networks is Dropout regularization [30]. By setting each neuron's activation to zero with a certain probability during training the network is prevented from relying completely on the output of specific neurons. In general redundant representations can make the model less sensitive to noise. A common choice for the dropout rate is a value of 0.5. For the FACT data a dropout rate of 0.25 provided the largest gain in performance. The values in figure 5.3 correspond to different dropout rates for the convolutional and fully connected layers.

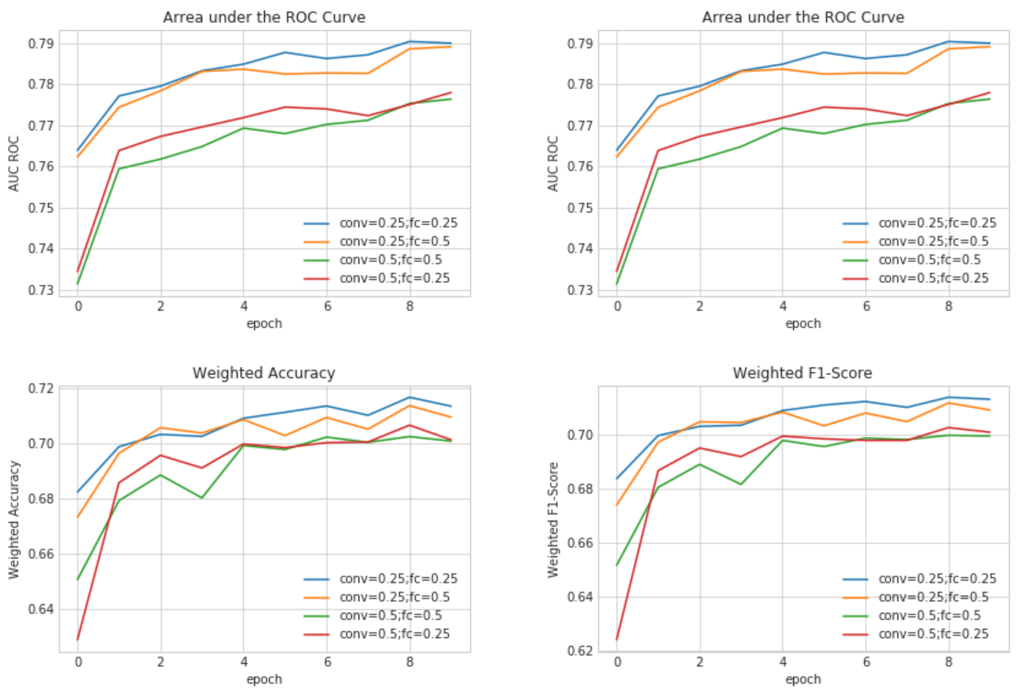

Figure 5.3: Comparison of dropout rates.

Activation Function

In contrast to fully connected MLPs which typically utilize sigmoid activations CNNs are usually trained using the Rectified Linear Unit (ReLU):

$$h_{\text{ReLU}}(y) = max(0, y)$$

The sigmoid can easily suffer from saturation and vanishing gradients due to the saturating regime at its border. In comparison to sigmoid the ReLU is not differentiable (in point 0). However in praxis choosing any subgradient works quite well. Apart from zero the derivatives are 0 and 1 and as a result no computation is required during backpropagation. Figure 5.4 compares the ReLU, sigmoid and other common activation functions on FACT data.

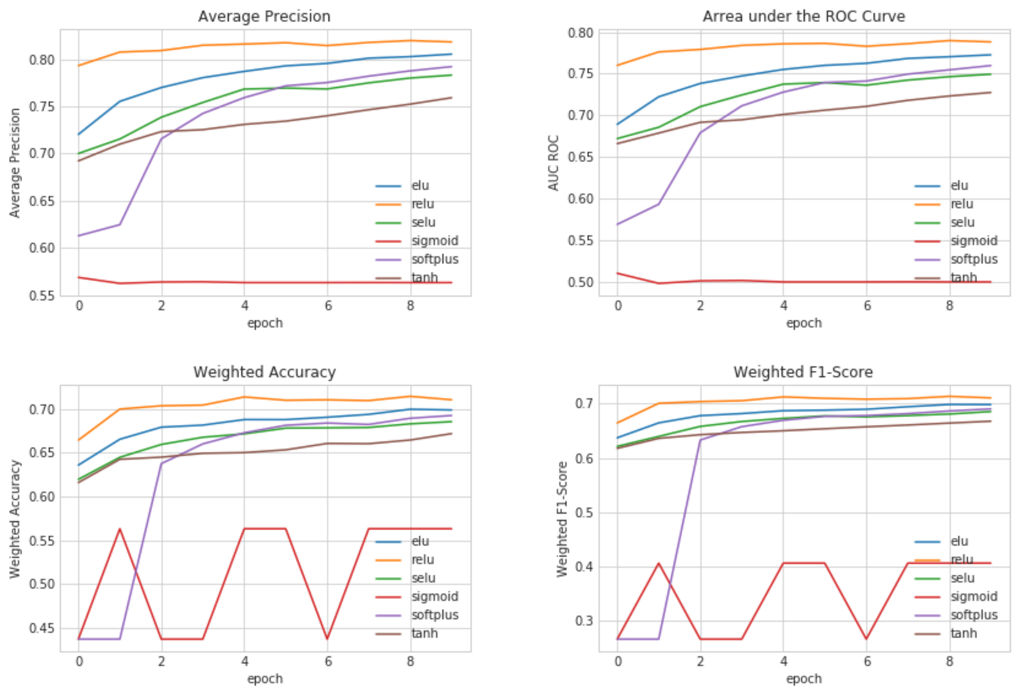

Figure 5.4: Comparison of activation functions.

Optimization

In praxis normalizing the inputs to have zero mean and equal variance results in improved learning [24]. For the FACT data described in thesis the normalization of inputs has already been performed during data set creation.

With regard to deep neural network batch normalization [16] is applied in order to center and scale the inputs of hidden layers during training and therefore correct for internal covariate shift. To this purpose batch normalization layers are added in between hidden layers which can compute the required shift and scaling based on parameters that are learned together with the actual model parameters during backpropagation. However in experiments on FACT the prediction quality decreased when batch normalization was used.

Chapter 3 already described the common practice of using mini-batches instead of stochastic gradient descent for the computation of update steps during learning because of more robust gradient estimates. In addition the time required per epoch was usually lower for larger batch sizes. On the other hand the high dimensionality of the data in combination with limited GPU memory prevents large batch sizes for 3D data.

Figure 5.5 compares the choice of different batch sizes for 2D data. The findings for 3D data are similar but due to memory requirements the maximum batch size is reached at 32.

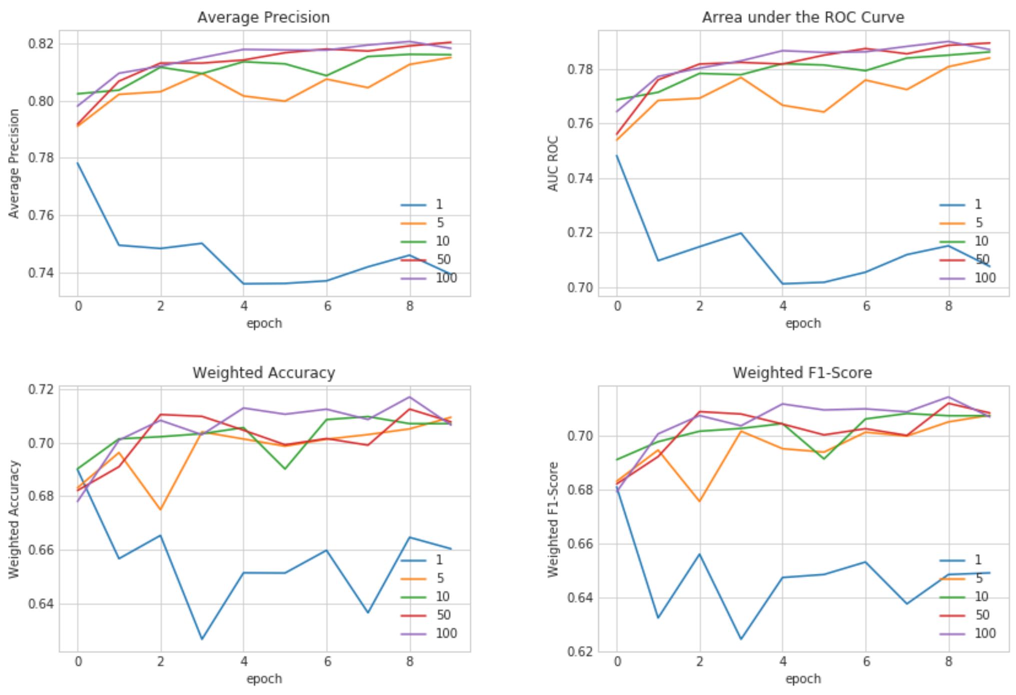

Figure 5.5: Comparison of mini-batch sizes. Mini-batch size 1 is actually stochastic gradient descent.

A number of variations for the formulation of the update step (e.g. adding momentum to gradient computations) and schemes for automatic adaption of the learning rate exist. Popular choices are AdaGrad [10], RMSprop, Adadelta [37], Adam [18] and Nadam[9].

Some of the optimizers have their own tunable hyper parameters like e.g. the initial learning rate and momentum initialization. Figure 5.6 compares different optimizers after a brief individual hyper parameter optimization on FACT data. The best performance are achieved by Adam and Adadelta. For the final experiments in this thesis Adam is used with initial learning rates between 0.0001 and 0.00005

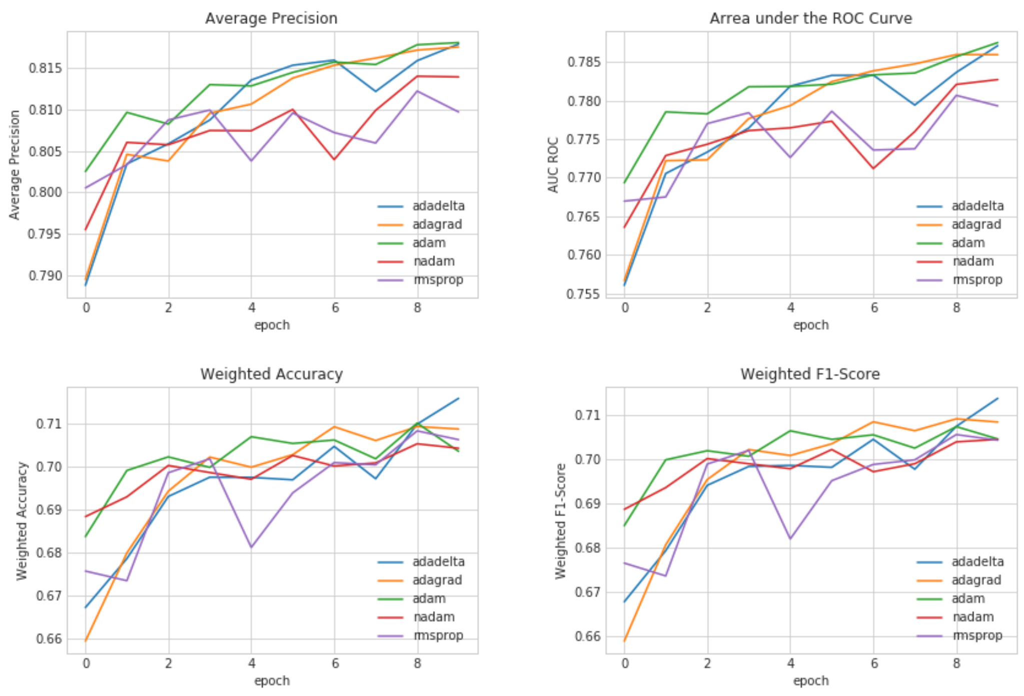

Figure 5.6: Comparison of popular optimization schemes which differ in how the learning rate is adapted and the gradient value for the update step is computed.

6 | Experiments and Evaluation

6.1 | Learning Tasks

The final gamma-hadron classification task is a binary classification on video like inputs of dimension $432,000$ (1440 pixels $\times$ 300 time slices). Depending on the transformation from a hexagonal camera grid to a square grid using offset coordinates the input dimension increases further to $540,000$ (45×40 pixels $\times$ 300 time slices). The number of available training examples on the other side is only $214,000$. In terms of memory a factor 300 from the two dimensional image data to actual three dimensional data corresponds to an increase in data size from approximately 3GB to 900GB.

The incredibly huge dimension in combination with comparably little data is a gigantic task and optimization in an end to end manner is difficult. As an end to this a number of simpler classification tasks aiming for the temporal, spatial or spatio-temporal aspect of the data can be defined to pave the way towards the final prediction model.

As a simple approach towards reducing the dimensionality of the data is to use only a subset of the original time slices. When a hardware trigger detects an event a number of observations prior to the trigger and a specified number of observation after the trigger are recorded into one event. As a result the beginning and end of the time series usually represent only random noise, while the actual event can be observed in the middle of the time series. Sampling every second frame in the range from time slice 30 to 150 results in a *sliced dataset* of size $45 \times 40 \times 75$.

The simulated data provides in addition to the real voltage values and label information for each pixel the ground truth photon charge, i.e. how many photons hit that pixel which actually stem from the source. This information is used in `fact-tools` to evaluate the hand designed processing step of image cleaning.

The existing image cleaning methods implemented in `fact-tools` as well as simple aggregation methods over the time dimension can be used to reduce the three dimensional input data to two dimensional data. Alternative reductions can be *learned* by regarding the image cleaning as supervised learning problem. Then 2D-CNNs can be applied to address the task of recognizing event classes from the spatial distribution of their shower pixels.

In principal 1D-CNNs can be used for time series classification. However the actual pixel cleaning dataset for FACT data is unbalanced by a factor of approximately 1:24 resulting in poor classification performance and will not be discussed in this thesis but may be a viable option if more training data is simulated in the future.

This chapter investigates two approaches to gamma hadron separation. Section 6.2 investigates the design of 2D-CNNs that can be used to predict the event class from image like measurements arranged according to the camera surface. On the one hand models are trained on ground truth data which provides a benchmark for how good gamma hadron separations could be, if the CNN was plugged into the processing pipeline of `fact-tools` after the image cleaning step. On the other hand the models are applied to aggregated versions of the real data in order to obtain estimates for end to end gamma hadron separation using 2D-CNNs.

Section 6.3 addresses the approach of convoluting three dimensional filters with three dimensional real data using CNNs while section 6.4 compares this approach to the approaches from section 6.2.

Finally section 6.5 investigates the influence of the amount of available training data on the final prediction quality.

6.2 | Spatial Aspect of Data (2D)

Two dimensional CNNs can be applied to the results of the image cleaning task in `fact-tools` and could be integrated in the tool chain at this position. To determine how well 2D Convolutional Neural Networks can address the spatial aspect of the data given perfectly cleaned inputs we train and evaluate the network on the ground truth photon charges and also on binary ground truth information as it might be provided by the image cleaning operation.

In addition we reduce the three dimensional data to two dimensions using mean aggregation over the time dimension. A fourth dataset is obtained by using four channels for each input pixel containing the mean, max, minimum and standard deviation of the the times series for each pixel.

Based on the representation of the hexagonal camera surface in a square array, we are required to use stride 2 (at least in one dimension) because of the broken symmetries. However stride two corresponds to downsampling by a factor of two at each layer. Therefore even if no pooling layers are added to the architecture the spatial dimension is reduced from $40 = 2 \cdot 2 \cdot 2 \cdot 5$ to 5 after only three layers. Therefore the deepest possible architecture that strictly respects the symmetries contains a maximum of 3 or 4 convolution layers. Obviously net architectures will be even shallower if non overlapping convolution is applied.

The best architecture with a maximum of three convolution layers has been empirically determined to be model M_6 (see figure 6.1):

$$
\begin{aligned}
M_6 = \{&\text{kernel size} = [3, 3, 7]\\
&\text{strides} = 2\\
&\text{neuron numbers} = [128, 64]\\
&\text{pooling} = \text{none}\\
\}&
\end{aligned}
$$

For M_6 all convolution layers have filter size 3 and stride 2, which should be the optimal choice according to chapter 4. As a consequence the convolution layers already realize the downsampling step and the model therefore has no pooling layers.

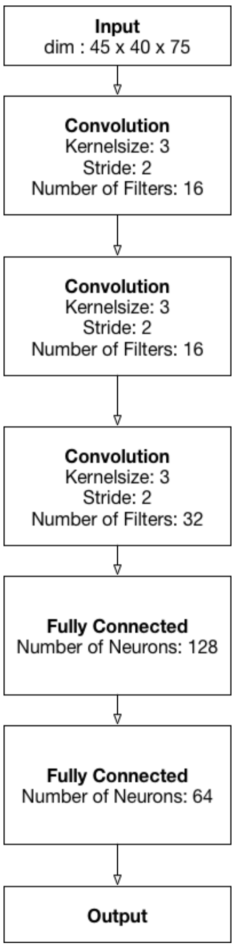

Figure 6.1: Architecture of model M_6.

Figure 6.5 compares the choice of alternative strides for M_6. As expected even strides outperform uneven strides, however only rows are offset against each other and it is therefore possible to use even stride in horizontal direction only.

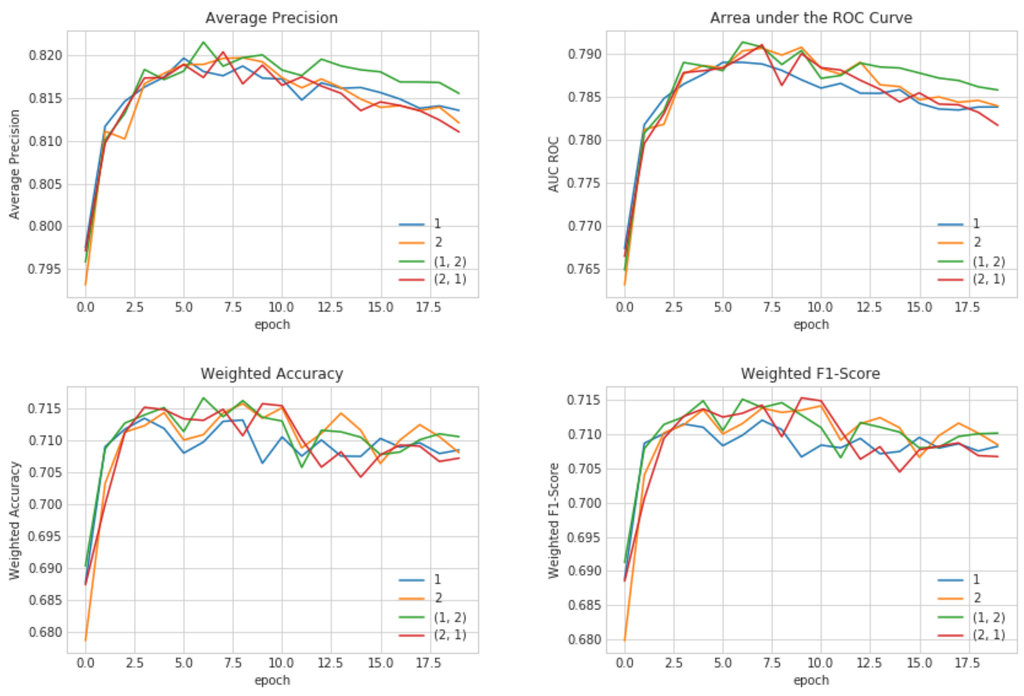

Figure 6.2: Comparison of different stride values

Adding pooling layer to an architecture with convolution stride 2, will further decrease the number of possible layers. Therefore figure 6.3 compares a pooling free model in which subsampling is achieved by stride 2 to a version of this model with stride 1 and additional pooling layers for subsamling. Although translational equivariance does not hold for stride 1. The results for average pooling are similar to those of the pooling free version.

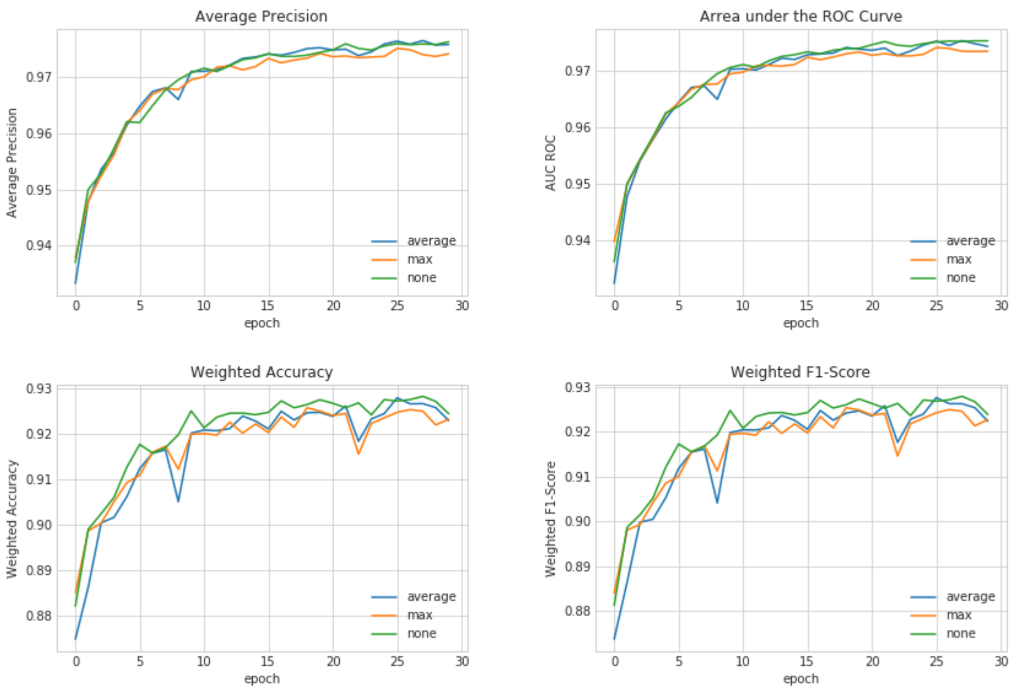

Figure 6.3: Comparison of different pooling operations based on a model with kernel sizes $[3, 3, 5]$

Experiments with varying kernel sizes yielded the best results if small filters of size 3 were used for the first layers, while performance increased with the use of larger filters in higher layers (see figure 6.4).

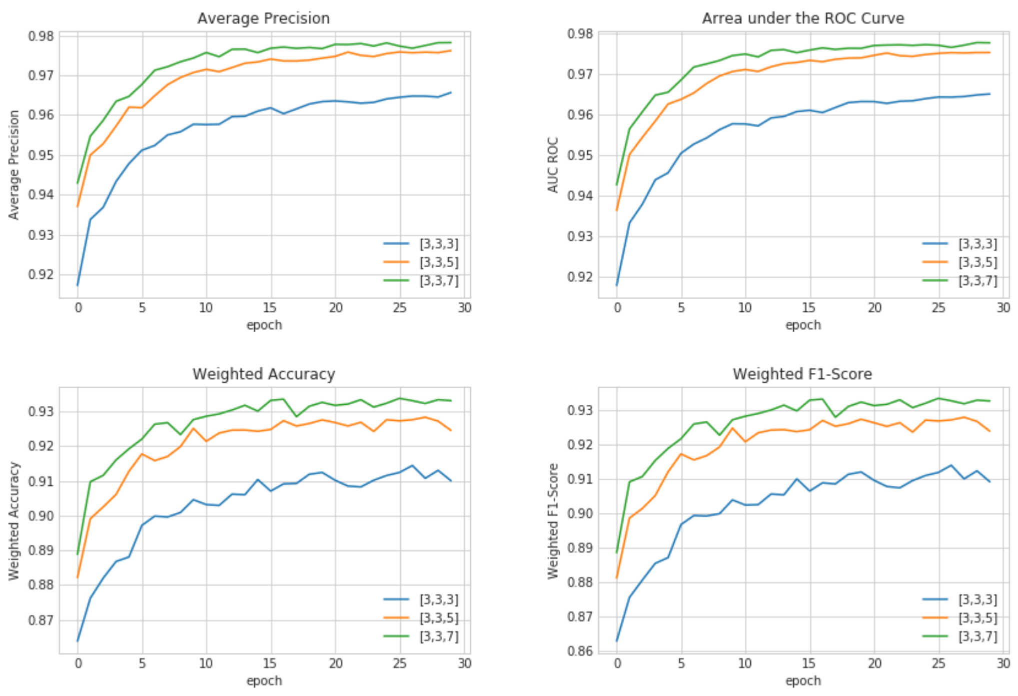

Figure 6.4: Comparison of different kernel sizes for base model M_6.

Varying the number of filters per layer had little impact on the final classification performance. Interestingly the best results were achieved if the same number of filters had been used in each layer. Filter numbers of 32 or 64 are viable for the 2D task.

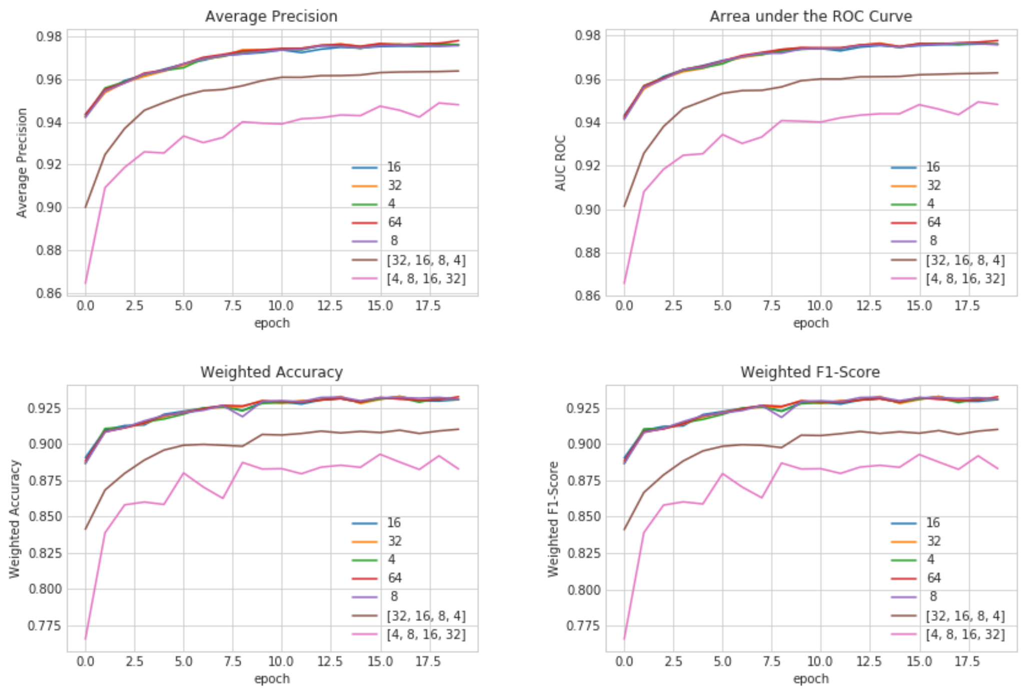

Figure 6.5: Comparison of different numbers of filters per convolutional layer.

While breakthrough performances achieved on image classification tasks rely on a large number of hidden layers, networks respecting the symmetries induced by the hexagonal grid are limited to three or four convolution layers. However with regard to a possible custom convolution operation that can be applied with stride 1 (see section 4.2), deeper architectures may be realized in the future. A study of the influence of an increasing number of convolution layers (see figure 6.6) indicates that deeper architectures are likely to significantly improve classification performance.

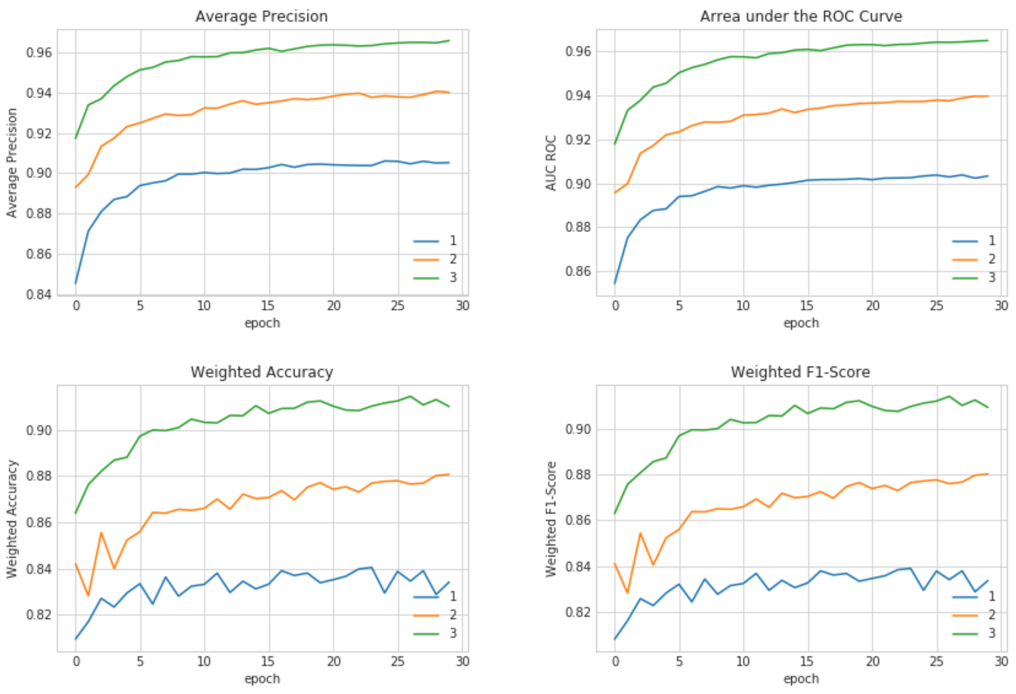

Figure 6.6: Influence of network depth on prediction quality. Comparison of models with different numbers of convolution layers (with kernel size 3, stride 2 and no pooling each).

A study of different architectures for the fully connected layers based on a fixed architecture of convolution layers shows that the influence of the fully connected architecture is much smaller than the influence of different choices for the convolution layers (see figure 6.7). In general the prediction quality depends rather on the number of neurons in the fully connected layers than the number of fully connected layers. Actually most CNN architectures in image processing use no more than two layers either.

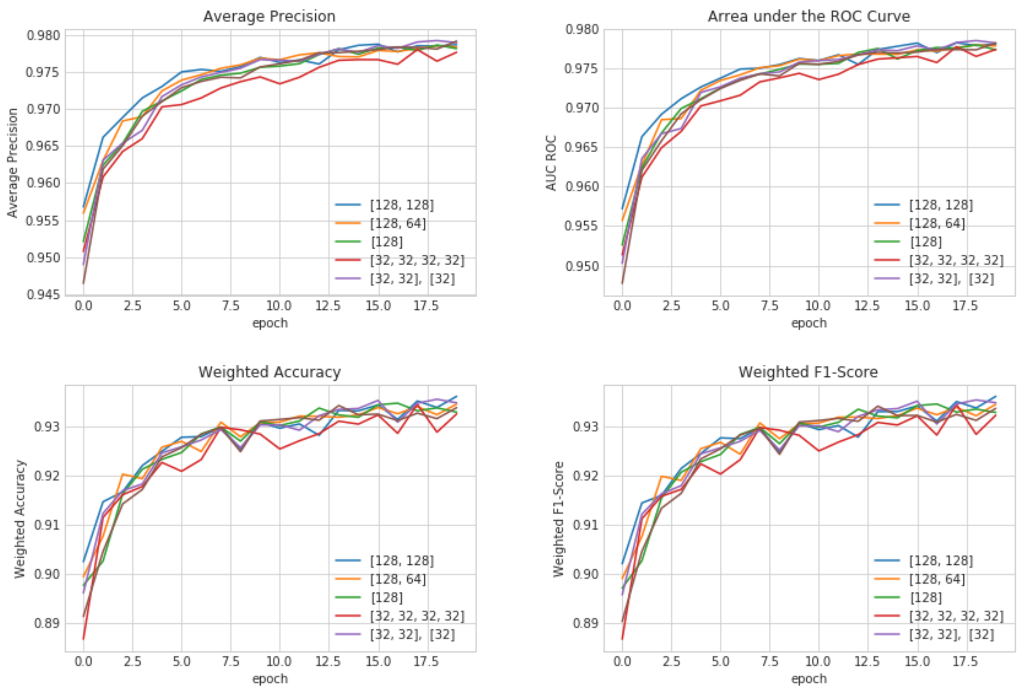

Figure 6.7: Comparison of different architectures for the fully connected layers.

6.3 | Classification of Real Data (3D)

The 3D-task on full or sliced data can be realized using a CNN architecture with three dimensional filters which capture a few frames of a spatial area at each filter position. The filters are therefore not only stridden across the camera surface but also along the time dimension.

Answering the much higher number of inputs in the time dimension the best results are no longer obtained using symmetrical filters. Due to spatial subsampling with stride 2 the constraint on the maximum depth does still hold. However additional filters of size $1 \times 1 \times d$ with stride 1 can be added without breaking the spatial symmetries, but provides no gain in performance.

The best architecture with again less than 4 layers is given by M_2:

$$\begin{aligned} M_2 = \{&\text{kernel size} = [(3,3,5),3,3] \\ &\text{strides} = [(2,2,3),2,2] \\ &\text{neuron numbers} = [32,32] \\ &\text{pooling} = \text{none} \\ \}& \end{aligned}$$

$$\begin{aligned} M_0 = \{&\text{kernel size} = [(3,3,10),3] \\ &\text{strides} = [(2,2,5),2] \\ &\text{neuron numbers} = [32,32] \\ &\text{pooling} = \text{none} \\ \}& \end{aligned}$$

$$\begin{aligned} M_1 = \{&\text{kernel size} = [(3,3,10),3,2] \\ &\text{strides} = [(2,2,5),2,2] \\ &\text{neuron numbers} = [32,32] \\ &\text{pooling} = \text{none} \\ \}& \end{aligned}$$

and compared to architectures M_0 and M_1 in figure 6.8. Again a tendency of improved performance with increasing number of layers can be noticed.

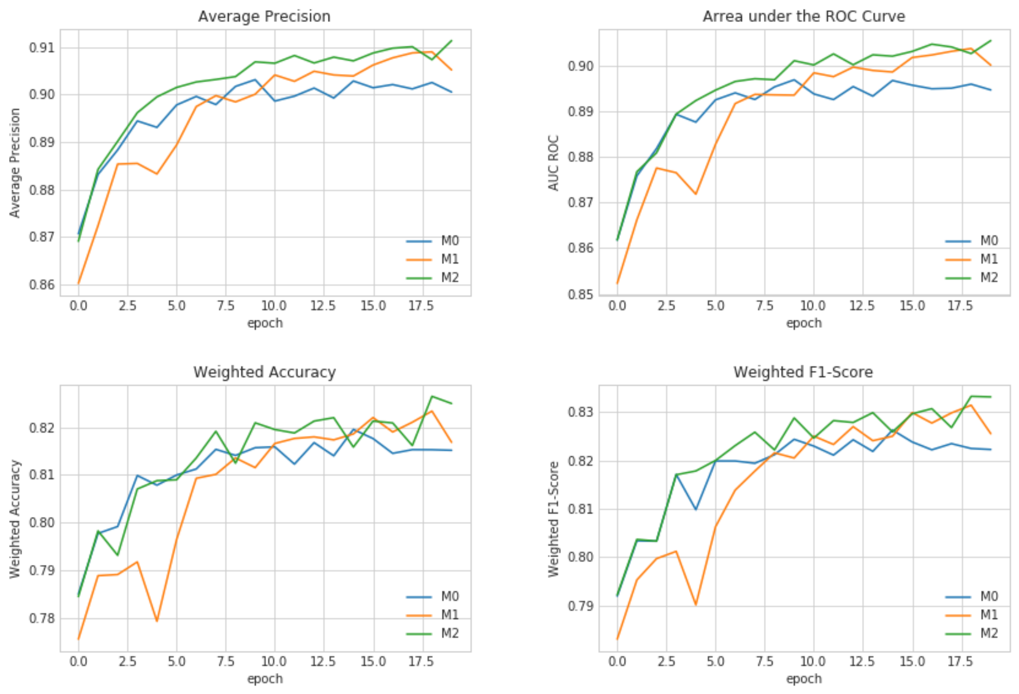

Figure 6.8: Comparison of architectures M_0, M_1 and M_2 for 3D convolution on FACT Data.

6.4 | Comparison of Learning Tasks

The previous sections have identified a best architecture for each learning task by scoring models on the validation data set during training. The best performing architecture for the 2D task achieves the best test performance on all the available datasets. The final architectures can be evaluated on the test set and compared in order to relate the utility of different learning task and datasets with regard to the final prediction quality (see table 6.1).

Predicting the final class from ground truth cleaned images works well although no time information is used. The values show how good classification based on optimal image cleaning can be. Using multiple aggregates results in better prediction than relying on the mean alone. However the results for prediction based on aggregated time information are far below the possible results achieved on ground truth data. 3D convolution on real data clearly outperforms aggregation based approaches indicating that filters can make

	Average Precision	AUC ROC	Weighted F_1-Score
2D Mean	0.8189	0.7887	0.707
2D Aggregated	0.8493	0.829	0.7521
2D Binary	0.964	0.9653	0.9139
2D Ground Truth	0.9794	0.9788	0.9361
3D Sliced	0.9135	0.9075	0.8373

Table 6.1: Comparison of learning tasks using the best model trained for each task.

use of the time dimension to some extend. However 3D-CNNs on real data cannot match the performance achieved on ground truth data.

Figure 6.9 presents the final ROC and Prediction-Recall curves for selected tasks.

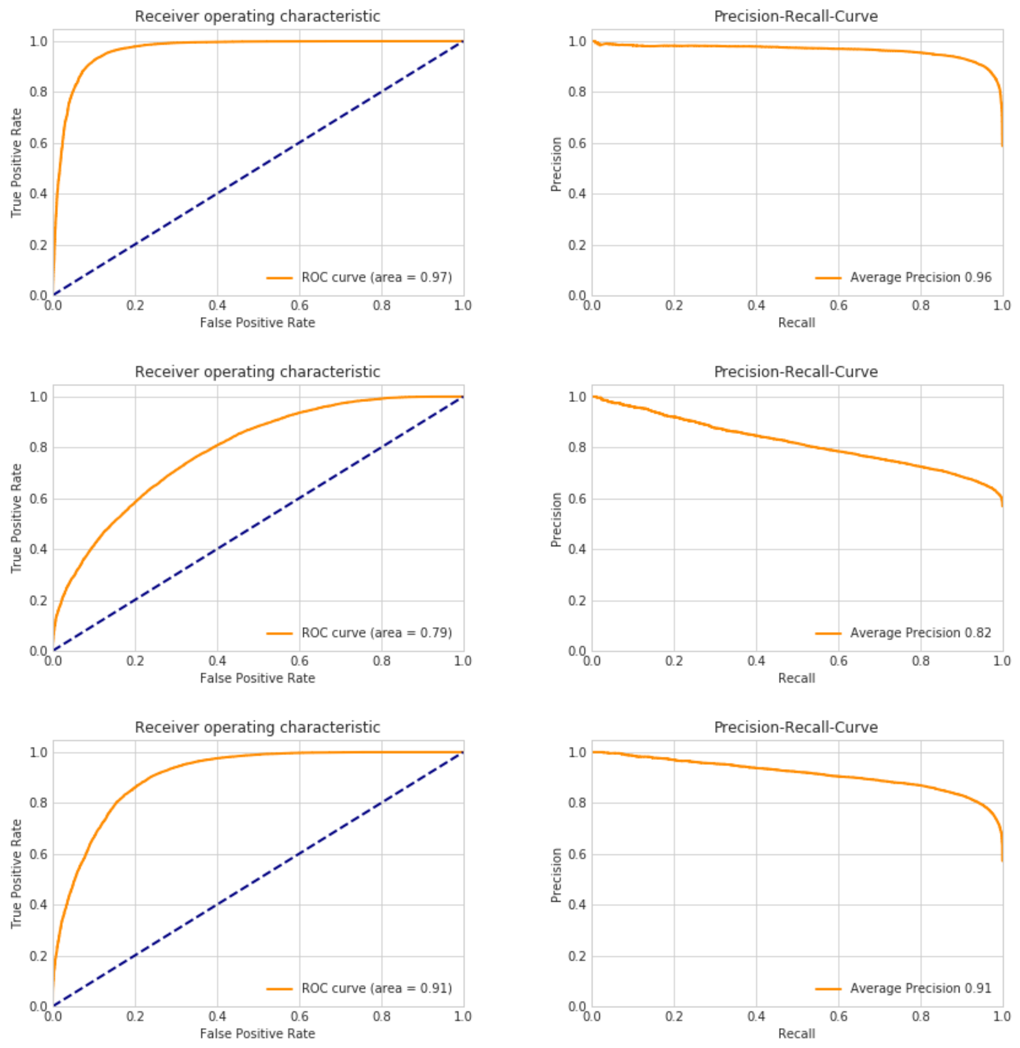

Figure 6.9: ROC (left) and Precision-Recall (right) curves for tasks 2D-Binary (top), 2D-mean (middle) and 3D-Sliced (bottom)

6.5 | Influence of Training Set Size on Performance

As described above the available training data is very limited in relation to the high dimensionality of data. However throughout the course of this thesis additional events have been generated. Therefore the same experiments can be conducted on much larger datasets in the future. In order to estimate how much gain in performance can be achieved through additional data, figure 6.10 shows the performance achieved on subsets of the currently available data.

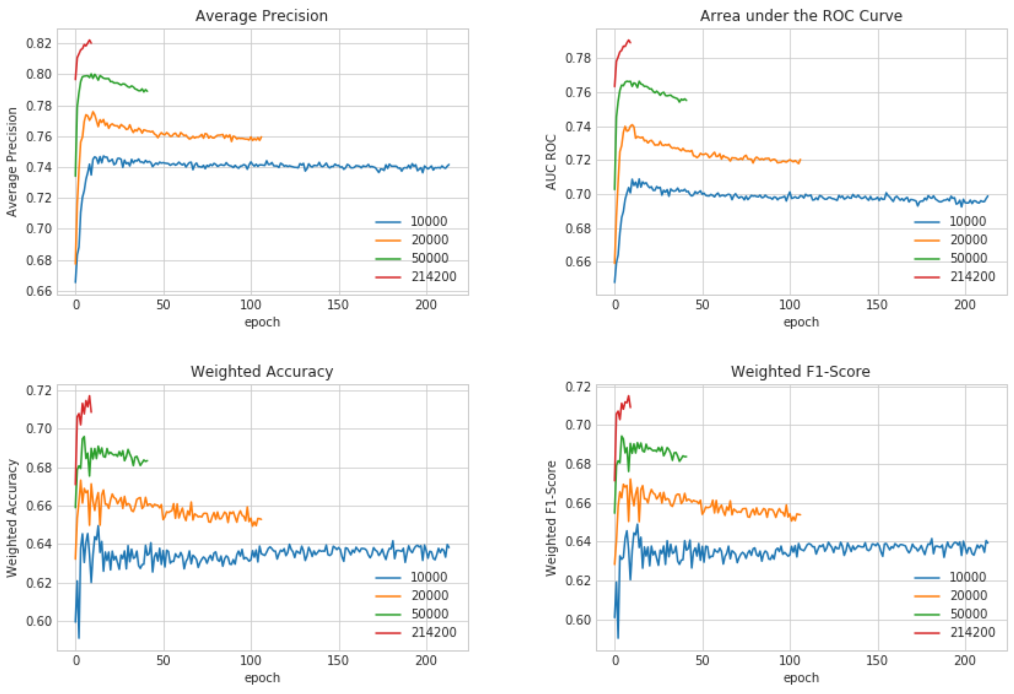

Figure 6.10: Influence of training set size on the prediction quality. For smaller training set sizes the number of epochs is proportionally higher, i.e. for the last measurement each of the plotted models has processed the same number of examples.

7 | Conclusion and Outlook

7.1 | Conclusion

This thesis motivates the realization of the processing pipeline of `fact-tools` using a fully automated end-to-end machine learning approach. Neural networks can be used to learn a representation that is suitable for classification in a supervised fashion. Convolutional Neural Networks control the high dimensionality of the data by utilizing parameter sharing and local connectivity based on the underlying symmetries of the data.

Application of existing CNNs implementations to FACT data is not straight forward. This thesis proposes a transformation of the hexagonal camera surface into offset coordinates andt to use deep learning libraries with convolution implementations for square grids. However the maximum depth is restricted to few convolution layers because even stride is required for fulfillment of the property of equivariance to translation.

Nevertheless related work and empirical research suggest that deeper architectures would improve performance. Towards this goal the implementation of a custom hexagonal convolution is sketched out. Further experiments indicate that prediction quality will heavily increase when additional training data is available.

Applying CNNs to the high dimensional raw data is a challenging task in itself. To this end a careful data preparation workflow and hyper parameter optimization framework have been implemented. The resulting data can be selected in a flexible manner to realize different learning tasks and is already preprocessed for efficient learning. Further a great effort was made to set up a `Keras` and `Tensorflow` based deep learning environment on a computation cluster with multiple GPU nodes using `docker` and `swarm`.

Prior to the comparison of models concerning their suitability for the final classification task an extensive parameter optimization over possible architectures, design choices and optimization methods has been performed. Deeper models result in better performance. Overlapping filters with stride 2 perform better than explicit pooling operations. The best performances are achieved under training with Adam optimizer and dropout regularization. The ReLU activation is key to successfully training the models.

2D-CNNs can be integrated into the `fact-tools` pipeline after the image cleaning step and replace the explicit feature extraction steps and random forest classification. Promising results are reported for the classification of optimally cleaned images. 3D-CNNs can replace the the entire pipeline from raw data to gamma-hadron separation and clearly outperform the application of 2D-CNNs after aggregation over the temporal aspect of the data.

7.2 | Outlook

During the progress of the work a huge number of new events has been simulated. Training the best architecture for each task on the new data will likely improve prediction performance a lot and may allow training of more complicated models (i.e. more neurons and layers) in the future.

If the resulting prediction quality is still not sufficient for replacing the existing processing pipeline, unsupervised pretraining should be investigated. Unsupervised pretraining on unlabeled real data (opposed to labeled simulation data) can help to bridge the gap between simulation and real world observations.

In general explicit learning of representations using deep auto-encoders and the realization of the the image cleaning step as a Denoised Auto-Encoder are promising.

1D-CNNs for handling the time aspect of the data showed poor results in first experiments but should be revisited with more available training data. Further a Recurrent Neural Networks (RNN) might be better suited for addressing the time aspect in the data. RNNs can be evaluated on the pixel cleaning task but also for aggregation of two dimensional convolutions of each individual time slice.

One great obstacle during the implementation was the efficient feeding of data into Keras which for example lacks support for protobuf based *tfrecords* which provide an efficient way of feeding data into Tensorflow. Therefore skipping the Keras abstraction and implementing models in Tensorflow from the beginning should be evaluated.

The most important area of further research is the formal definition and efficient implementation of a true hexagonal convolution operation like the one highlighted in 4.2 in order to enable meaningful filter translation with stride one and therefore paving the way toward deeper models. If the true hexagonal convolution operation is based on a sparser representation (1440 pixel array instead of $45 \times 40 = 1800$ pixels) this might help with regard to the data feeding problems as well.

Once the quality of the learned models is satisfying, the application of CNN visualization techniques to the learned model can be investigated with regard to better understanding the learned model or even the underlying structures of gamma hadron classification. Typical visualization techniques for CNNs are the visualization of filter values as images and the calculation of inputs that achieve maximum activation for each class using gradient ascent or deconvolution operations.

Another important aspect that should be addressed only after satisfying models exist is the comparison to the performance of the existing `fact-tools` pipeline and possible integration, e.g. Tensorflow provides a very basic and not officially supported java interface. For the integration pure model application is probably sufficient and a suitable hardware solution needs to be identified, e.g distributed processing, processing in computation cluster or local processing on FPGAs are possible approaches.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] H. Anderhub, M. Backes, A. Biland, A. Boller, I. Braun, T. Bretz, S. Commichau, V. Commichau, M. Domke, D. Dorner, and et al. Fact - the first cherenkov telescope using a g-apd camera for tev gamma-ray astronomy. *Nuclear Instruments and Methods in Physics Research A*, 639:58–61, May 2011.

[3] Yoshua Bengio. Learning deep architectures for ai. *Found. Trends Mach. Learn.*, 2(1):1–127, January 2009.

[4] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(1993):1–30, 2013.

[5] Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, et al. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.

[6] Christian Bockermann, Kai Brügge, Jens Buss, Alexey Egorov, Katharina Morik, Wolfgang Rhode, and Tim Ruhe. Online analysis of high-volume data streams in astroparticle physics. In *Proceedings of the European Conference on Machine*

Learning (ECML), Industrial Track. Springer Berlin Heidelberg, 2015.

[7] Léon Bottou. *Large-Scale Machine Learning with Stochastic Gradient Descent*, pages 177–186. Physica-Verlag HD, Heidelberg, 2010.

[8] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[9] Timothy Dozat. Incorporating nesterov momentum into adam. 2015.

[10] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, July 2011.

[11] Peter Flach and Meelis Kull. Precision-recall-gain curves: Pr analysis done right. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 838–846. Curran Associates, Inc., 2015.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[14] A. M. Hillas. Cerenkov light images of EAS produced by primary gamma. In F. C. Jones, editor, 19^{th} *International Cosmic Ray Conference ICRC, San Diego, USA*, volume 3 of *International Cosmic Ray Conference*, pages 445–448, August 1985.

[15] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, July 1989.

[16] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[17] Andrej Karpathy. Lecture notes cs231n: Convolutional neural networks for visual recognition., accessed 09/2017.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[20] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[21] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[22] Lee Middleton and Jayanthi Sivaswamy. *Hexagonal Image Processing: A Practical Approach (Advances in Pattern Recognition).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[23] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, MA, USA, 1969.

[24] Genevieve B. Orr and Klaus-Robert Mueller, editors. *Neural Networks : Tricks of the Trade*, volume 1524 of *Lecture Notes in Computer Science.* Springer, 1998.

[25] Amit Patel. Red blob games., accessed 09/2017.

[26] D Petry. The magic telescope-prospects for grb research. *Astronomy and Astrophysics Supplement Series*, 138(3):601–602, 1999.

[27] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[28] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[29] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[30] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[31] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261, 2016.

[32] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[33] Matus Telgarsky. benefits of depth in neural networks. In Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir, editors, *29th Annual Conference on Learning Theory*, volume 49 of *Proceedings of Machine Learning Research*, pages 1517–1539, Columbia University, New York, New York, USA, 23–26 Jun 2016. PMLR.

[34] Fabian Temme. Analysis of crab nebula data using parfact, a newly developed analysis software for the first g-apd cherenkov telescope. Diplomarbeit, TU Dortmund, 2013.

[35] D.C. Wells, E.W. Greisen, and R.H. Harten. Fits - a flexible image transport system. *Astronomy and Astrophysics Supplement*, 44:363, June 1981.

[36] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 3320–3328, Cambridge, MA, USA, 2014. MIT Press.

[37] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[38] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013.