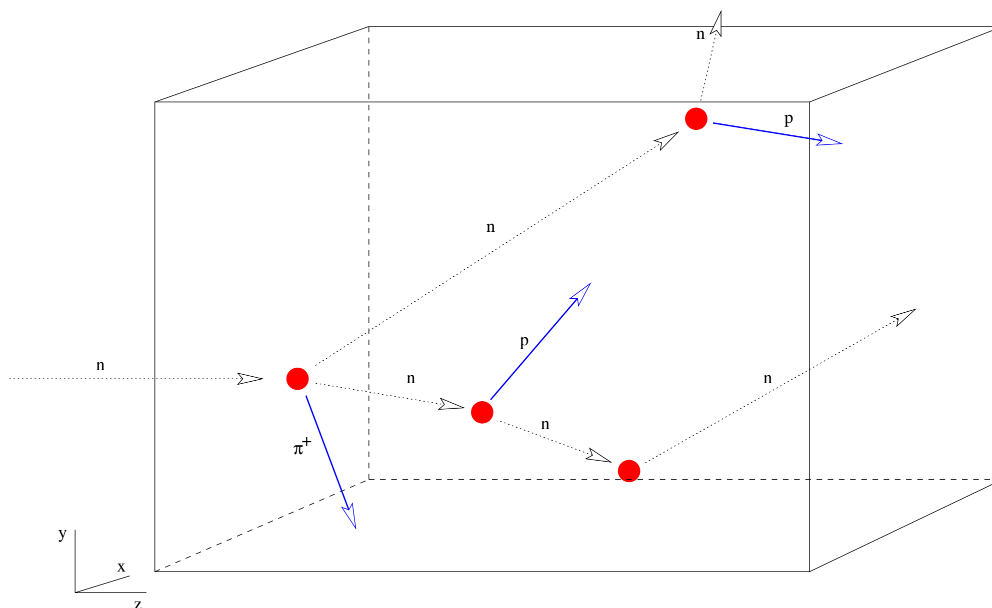
CHALMERS
UNIVERSITY OF TECHNOLOGY



Convoluted Events

Neutron Reconstruction using Neural Networks

Master's thesis in Subatomic Physics

MARKUS POLLERYD

Department of Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

MASTER'S THESIS IN SUBATOMIC PHYSICS

Convoluted Events

Neutron Reconstruction using Neural Networks

Markus Polleryd

Department of Physics
Division of Subatomic and Plasma Physics
Chalmers University of Technology
Gothenburg, Sweden 2017

Convoluted Events
Neutron Reconstruction using Neural Networks
Markus Polleryd

© Markus Polleryd, 2017.

Examiner: Andreas Heinz, Department of Physics

Department of Physics
Division of Subatomic and Plasma Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Schematic of neutron paths in NeuLAND. Only the charged particles (blue lines) are directly detectable, hinting at the complexity of reconstructing neutron events.

Chalmers Reproservice
Gothenburg, Sweden 2017

Convoluted Events
Neutron Reconstruction using Neural Networks
Markus Polleryd
Department of Physics
Chalmers University of Technology

Abstract

The R^{3}B experiment at FAIR will study properties of unstable nuclei through detection of reaction products of projectile-target interactions. It is essential that these reaction products can be measured with sufficient accuracy. Uncharged neutrons do not excite the scintillator material in the detector, and therefore can only be detected indirectly via charged products from neutron-nucleus interactions. These interactions can create multiple new particles including neutrons, in turn, interacting with other nuclei. Reconstructing the multiplicity and momenta of the neutrons entering the detector from these shower patterns is not trivial and requires sophisticated algorithms.
This thesis explores the possibility of reconstructing neutron events with 3-dimensional image recognition using Convolutional Neural Networks, focusing mainly on neutron multiplicity. When a passing charged particle excites the scintillator material in the detector, it outputs the spatial coordinates of the excitation point along with the time and the energy the particle has deposited. The output can be converted into a sparse 3-dimensional image with time and deposited energy as pixel values. The 300 000 pixel values in each image make the required amount of parameters, even in the smaller networks, very large. It is shown that training these large but simple networks using a Central Processing Unit is not practically feasible, requiring months to train a single network. The use of a Graphics Processing Unit introduced a speed up in training with a factor of up to 185.
By accounting only for the total deposited energy and number of hits in the detector, 72 % correct predictions were achieved on a large test set. Accounting also for the image of each event, an accuracy of 78 % correct predictions was achieved, showing that the networks are able to extract important features from the images.

Keywords: Neutron detection, convolutional neural networks, machine learning, R^{3}B collaboration, GSI/FAIR

Acknowledgements

Before jumping to the actual report, I would like to thank everyone who helped making this thesis successful. First and foremost I would like to thank *Andreas Heinz* for his constant support and daily checkups, always keeping me on the right path, and *Håkan T. Johansson*, the computer guru, for many long and short (mostly long) talks, advice, ideas and support during the whole thesis project.
I also want to thank *Hans Salomonsson* for leading us to use convolutional neural networks, the Swedish National Infrastructure for Computing (SNIC) at High Performance Computing Center North (HPC2N) for providing compute time on Tesla K80 GPUs, *Giovanni Bruni* for support and small talks and *Andreas Johansson* for providing an errata making the final report flawless, I hope.
Finally, a huge thanks to *Thomas Nilsson* and everyone else at the Subatomic Physics Group at Chalmers for giving me the opportunity to do a Master's thesis in Subatomic Physics.

Markus Polleryd, Gothenburg, November 2017

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Machine Learning . . . 4

2 Preliminaries **5**
2.1 NeuLAND . . . 5
2.2 Machine Learning . . . 5
2.2.1 Artificial Neural Networks . . . 7
2.2.2 Convolutional Neural Networks . . . 8
2.2.3 Optimization Algorithms . . . 8

3 Setting up a working machine learning model **11**
3.1 Simulations using ggland/GEANT4 . . . 11
3.1.1 Light propagation in scintillating paddles . . . 11
3.1.2 Neutrons in NeuLAND . . . 11
3.1.3 What do neutrons in plastic do? . . . 12
3.2 Network implementation in TensorFlow . . . 12
3.2.1 Building a TensorFlow graph . . . 13
3.2.2 Running a TensorFlow graph . . . 15
3.2.3 Storing the TensorFlow graph . . . 16
3.2.4 Challenge of using TensorFlow . . . 16
3.3 Data reading . . . 16

4 Developing and testing models **17**
4.1 Simple multilayer perceptron as accuracy reference . . . 18
4.2 Baseline CNN model . . . 19
4.3 Attempting to speed up the learning process . . . 19
4.4 Uncertainty in training . . . 21
4.5 Improving the model . . . 22
4.5.1 Removing events from the training set . . . 22
4.5.2 Adding total deposited energy and number of hits . . . 23
4.5.3 Data normalization . . . 24
4.5.4 Reducing the training dataset . . . 26
4.5.5 Variations of the extended baseline model . . . 26
4.5.6 Increasing the number of layers . . . 28
4.6 Evaluating the individual neutron multiplicities . . . 28
4.7 Accounting for light loss due to Birk's law . . . 28

4.8 Predicting neutron momentum 29

5 Discussion and conclusion **33**
5.1 Training on Graphic Processing Units 33
5.2 CNN for reconstructing neutron events 33
5.3 Accuracy cost due to Birk's law 35

6 Outlook **36**

Chapter 1

Introduction

1.1 Background

A charged particle propagating through scintillator material in a detector will excite molecules via Coulomb interaction. The light emitted from a molecule as it returns to its ground state can then be detected with photomultiplier tubes. Uncharged neutrons, on the other hand, solely interact with matter via strong interaction and cannot excite the scintillator material via Coulomb interaction. The neutrons of interest in this thesis have kinetic energies between 0.1 and 1 GeV, for which scintillating detectors are optimal [1]. Therefore, neutrons can only be detected indirectly via charged products from neutron-nucleus interactions, i.e. collisions with nuclei. These collisions can knock out or create multiple new particles including neutrons, that in turn interact with other nuclei. In this way, each neutron entering the detector typically creates a shower of particles, that is, each neutron-nucleus collision branches into multiple new particles and neutron-nucleus interactions (figure 1.1). Reconstructing the multiplicity and momenta of the original neutrons entering the detector from these shower patterns is not trivial, specially for high neutron multiplicities. Figure 1.2 illustrates the complexity of an increased neutron multiplicity.
Currently NeuLAND (new Large Area Neutron Detector) is under construction at GSI Helmholtzzentrum für Schwerionenforschung. NeuLAND is a scintillator-based detector with an active face size of 2.5 x 2.5 m^2 and a total depth of 3 m. It is designed to be a key part of the experimental setup for studies of Reactions with Relativistic Radioactive Beams (R^3B) as part of FAIR (Facility for Antiproton and Ion Research). FAIR, currently under construction at the site of the GSI facility in Darmstadt, will be one of the largest and most complex accelerator laboratories in the world.
The R^3B experiment will study properties of unstable nuclei near the dripline through detection of reaction products of projectile-target interactions. The neutron (proton) dripline marks the limit where a nucleus has no bound states for additional neutrons (protons). Due to short lifetimes, it is not feasible to create targets of these unstable nuclei, instead, they constitute an incoming beam bombarding a target of light nuclei. This process, performed inversely to traditional physics experiments, is called inverse kinematics. When studying nuclear states with energies above the particle emission threshold, the relative energy, i.e. the difference in energy of the

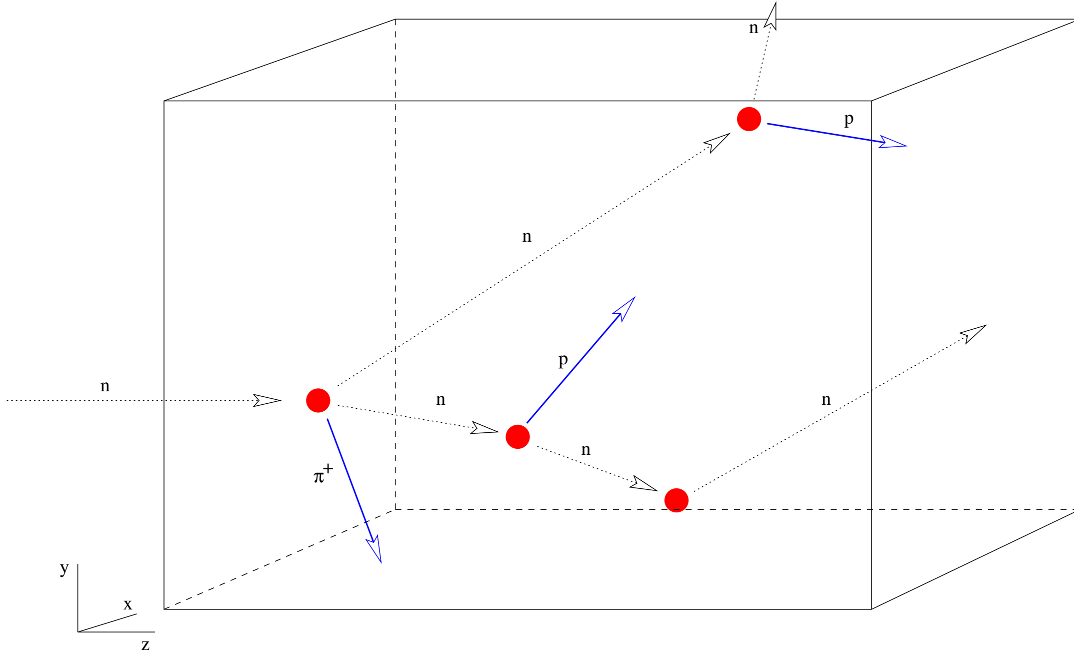
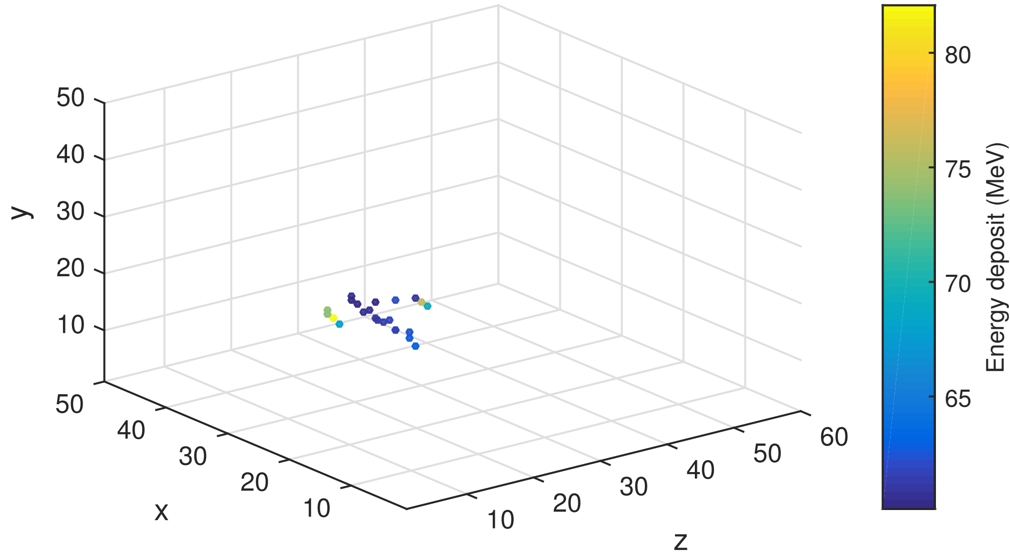


Figure 1.1: A schematic illustration of neutron-nucleus interactions in a scintillator-based detector.

system before and after particle emission, is important. In order to reconstruct the relative energy, the invariant mass of the system before and after particle emission needs to be known. Additionally, since the invariant mass is calculated from the four-momenta of the resulting fragments and emitted particles, it is essential that the reaction products can be measured with sufficient accuracy. In this thesis, we focus on detecting and reconstructing neutrons using the neutron detector NeuLAND.
NeuLAND features a higher detection efficiency and resolution along with a better multi-neutron-hit resolving power compared to the current Large Area Neutron Detector (LAND) [2]. For LAND the reconstruction is currently done using the shower algorithm [3], an algorithm that essentially sorts the hits in time and assigns the first hit as a neutron interaction vertex. Here, a hit is defined as single detection of a charged particle traversing a scintillator block in the detector. Further hits that can be associated as results of scattering from the first vertex are removed as secondaries. If hits remain, another neutron is assigned by reiterating the routine. These assignments are sometimes wrong, leading to erroneous event reconstruction. NeuLAND on the other hand, due to its improved energy resolution, will be able to correctly determine the number of incident neutrons by examining the total number of neutron-nucleus hits and total energy deposited with a certain probability. This can be done using previously gathered distributions (from simulations) for both the deposited energy and number of hits for each number of incident neutrons. To assign four-vectors, it still needs to inspect the individual interactions. In order to fully take advantage of the advanced detection capabilities NeuLAND offers, it is important to also improve and adapt the reconstruction algorithms.



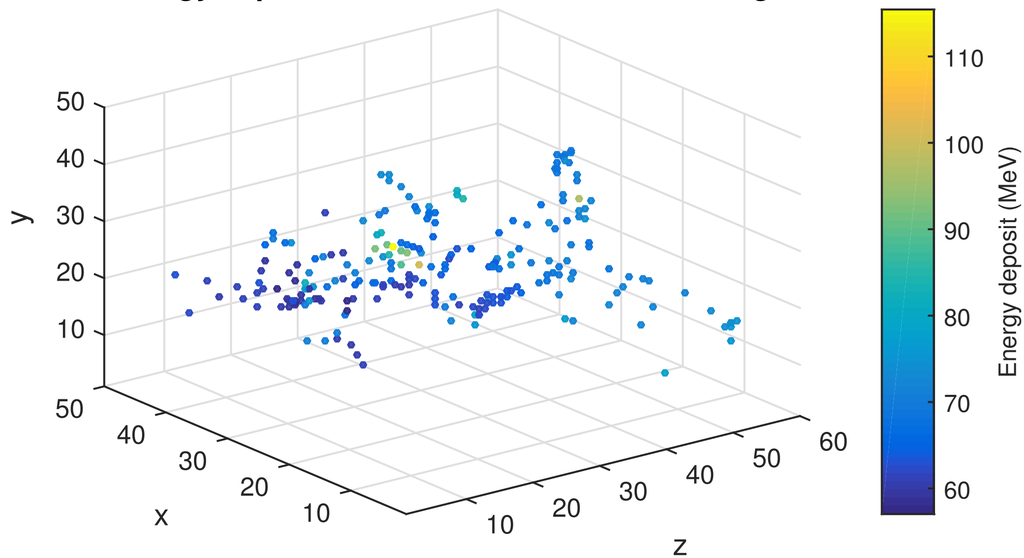


Figure 1.2: Simulated output from the scintillator-based detector NeuLAND (see text), converted into 3-D images showing the deposited energy of particles traversing the detector, for both 1 and 5 incoming neutrons travelling in the positive z-direction. The axes, given in pixel number, represent the full geometry of NeuLAND.

1.2 Machine Learning

The task of reconstructing neutron events is complex due to the fact that only charged particles originating from neutron-nucleus interactions are detected. The number of possible events creating a certain hit pattern in the detector is large making it impossible to reconstruct the incoming neutrons with 100 % certainty, and therefore only the most probable event is the one of interest. For instance, "ghost hits", where a neutron solely knocks out another neutron, are invisible to the detector. The two outgoing neutrons can generate new hits that cannot be related by scattering from one another without breaking causality. The shower algorithm currently used for LAND will therefore not be able to realize the connection between hits originating in ghost hits. An improved algorithm [4] taking a probabilistic approach has been tested, but suffers from becoming too computationally expensive when the number of interactions increase.

Machine learning can potentially improve the accuracy in reconstructing neutron events by learning the most probable events without explicitly implementing laws of physics. The concept of machine learning is to let an algorithm experience data and by itself extract connections between different features; essentially high-dimensional curve fitting. Which kind of features the algorithm learns is characterized by their recurrence in the experienced data, i.e. it learns the most probable features first, potentially reducing the computational power needed. Machine learning has proven to be an impressive success story, performing tasks such as language translation [5], image recognition [6] and even tasks in particle physics [7].

An event of neutrons entering NeuLAND can be represented as a three-dimensional (3-D) image with time and deposited energy as pixel values. The problem of reconstructing neutron events is then essentially identical to that of 3-D image classification and regression. Currently Convolutional Neural Networks (CNN) are one of the leading machine learning algorithms used in image classification. In this thesis, TensorFlow [8], an open-source software library for machine intelligence, is used to develop and test different CNN architectures for reconstructing neutron events in NeuLAND.

Chapter 2

Preliminaries

2.1 NeuLAND

The final design of NeuLAND consists of 3000 plastic scintillator paddles, each with dimensions of $5 \times 5 \times 250$ cm^3 arranged in 30 double planes. Each double plane contains 50 horizontal and 50 vertical paddles. When the plastic material in a paddle is excited by a passing charged particle, light will be emitted from the plastic and detected by PM-tubes (photomultiplier tubes) mounted at the ends of the paddle. The time and position where the charged particle traversed the paddle can be calculated using the times t_1 and t_2 that describe when the light was detected at each PM tube

$$
\begin{aligned}
t &= \frac{t_1 + t_2}{2}, \\
p &= v\frac{t_2 - t_1}{2},
\end{aligned}
\tag{2.1}
$$

where v is the effective speed of light in the scintillator material. Since the paddles are rotated by 90 degrees for each plane, p, of one paddle, becomes the horizontal coordinate for even planes and the vertical coordinate for odd planes. From now on, we define a hit as a single detection in both PM-tubes, unfortunately making the name somewhat misleading.

2.2 Machine Learning

Machine learning is a subfield of computer science that studies algorithms, which can learn from and make predictions on data. A widely quoted definition is provided by Tom M. Mitchel [9, p. 2]: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.".
Two machine learning approaches are employed in this thesis, namely *classification* and *regression*. For classification the algorithm should specify to which of k categories some input $\mathbf{x}$ belongs, for instance recognizing the number of neutrons in the 3-D images of neutrons interacting with NeuLAND. Regression is similar to classification

apart from the different output; the algorithm is intended to predict a continuous numerical value from the input data $\mathbf{x}$.
The performance measure P is needed in order to quantitatively evaluate the ability of the algorithm to perform some task. For classification, P is often defined as the proportion of the times the algorithm predicts the correct category, known as the accuracy.
Machine learning algorithms can be divided into supervised and unsupervised learning determined by the data they are fed during learning. An unsupervised learning algorithm experiences only the input data $\mathbf{x}$ from which it then learns useful features. Typically, it is desired that the algorithm finds the underlying distribution that generated the input dataset. During supervised learning the algorithm is also provided with output data $\mathbf{y}$ for each input $\mathbf{x}$ corresponding to some mapping $\mathbf{y} = \mathbf{f}(\mathbf{x})$, where the objective is to approximate the (unknown) function $\mathbf{f}$. In a classification task, $\mathbf{y}$ would be a label of the category to which $\mathbf{x}$ belongs.
In order to achieve good generalization, i.e. good performance on data that is independent of the data used during learning, the dataset is typically split into three independent subsets: a training set, a validation set and a test set. The training set is solely used by the learning algorithm to optimize the ability of the model to map $\mathbf{x}_{train}$ to $\mathbf{y}_{train}$. Generally, near-perfect performance on the training set can be obtained by sufficiently increasing the capacity of the model. The capacity is essentially the variety of functions a model can fit. Good performance on the training set does however not mean good performance on the test set. Increasing the capacity of a model eventually leads to overfitting: the model performs well on the training set while the generalization is bad. With too low capacity, the model is instead subject to underfitting which means alongside with bad generalization also poor performance on the training dataset.
The test dataset is used to obtain a final estimate of the ability of the model to generalize, called generalization or test error. The test dataset gives a good estimation of the generalization error assuming the examples in each dataset are independent and that the training and test datasets are identically distributed, both generated from the same probability distribution. As a result of this assumption, the expected test error will always be greater than or equal to the expected training error [10, p. 111].
Machine learning models most often contain hyperparameters, which are parameters not altered by the learning algorithm itself. These hyperparameters do affect the overall performance of the model, for instance increasing or decreasing the capacity. If the test set is used in any way to optimize the performance, e.g. by modifying the model, it will not provide a good estimate of the generalization error. It is therefore necessary to include the independent validation dataset to estimate the generalization error after each training period in order to find the hyperparameters resulting in best performance. In other words, after the test set has been used to evaluate a model, the model cannot be changed to get better performance on the test set without being biased. In practice, this is a problem when dealing with small limited datasets.

2.2.1 Artificial Neural Networks

An artificial neural network is a computational model that loosely mimics the network of neurons and synapses in a biological brain. The networks considered in this thesis are feedforward networks, or multilayer perceptrons (MLP). Given the input $\mathbf{x}$ and parameters $\boldsymbol{\theta}$, a network defines a function $\mathbf{f}(\mathbf{x}, \boldsymbol{\theta})$. The goal of a MLP is to approximate some function $\hat{\mathbf{f}}(\mathbf{x})$ by learning the parameters $\boldsymbol{\theta}$ that result in the best function approximation. Typically, a feed forward network can be represented by a chain of functions,

$$\mathbf{f}(\mathbf{x}, \boldsymbol{\theta}) = \mathbf{f}^{(N)}(...\mathbf{f}^{(2)}(\mathbf{f}^{(1)}(\mathbf{x}, \theta_1), \theta_2)..., \theta_N),$$

forming a network. The functions are called layers in the network and the more layers the network has, the wider is the variety of functions the network can fit, i.e. the capacity is increased. The name "deep learning" originates from this terminology, where many layers result in a deep network [10, p. 169].
Multilayer perceptrons contain multiple nodes, or neurons, connected to each other. They are arranged as an input and an output layer with multiple layers in-between, called hidden layers. The nodes in a hidden layer are called hidden nodes or hidden units where the usual type computes an affine transformation

$$z = \mathbf{W}^T\mathbf{x} + b, \tag{2.2}$$

corresponding to each connection between nodes having a weight W and bias b acting on the signal. A nonlinear activation function $g(z)$ is then applied to the output with the most typical being the rectified linear activation function defined as $g(z) = max\{0, z\}$. A node employing this function is called a rectified linear unit (ReLU).
In classification tasks, a softmax function is typically applied to the final output of a network. The softmax function, applied to all classification networks in this thesis, is defined as

$$\sigma(\mathbf{y})_i = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}}, \tag{2.3}$$

where the output represents a probability distribution over the n classes [10, p. 185].
Alongside the depth of a network, the number of nodes in hidden layers can be increased to gain capacity. In the same way that many layers make the network deep, we say that a large amount of nodes in the hidden layers makes the network wide.
Up to this point it may seem that making the network sufficiently deep and wide one can design a model able to learn any mapping $\mathbf{y} = \mathbf{f}(\mathbf{x})$. In fact, the universal approximation theorem [10, p. 198] states that a feedforward network with a linear output layer and at least one hidden layer, with some mild assumptions on the activation functions, can approximate any continuous function on a closed and bounded subset of $\mathbb{R}^n$ from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network contains enough hidden units. Unfortunately, too large networks encounter problems such as overfitting and being too computationally expensive. When constructing a neural network, the design of individual nodes and network architecture that results in best performance, must

be found. The architecture is essentially the number of nodes and layers a network contains and how they are connected to each other. There exists no theoretical framework for choosing the optimal node design and network architecture, making the process of designing a network experimental, that is, by trial-and-error while continuously monitoring the validation error.

2.2.2 Convolutional Neural Networks

A convolutional neural network (CNN) is a type of network that uses discrete convolution to process data with grid-like topology such as time-series data or images [10, p. 330]. The convolution operation, which allows a network to extract features of a local group of points in the input data, has proven to be essential in the field of image recognition. A discrete convolution of some data $x(i)$ with a kernel $k(a)$ taking only integer values of i and a, for instance the pixel values of an image, is defined as [10, p. 332]

$$s(i) = (x * k)(i) = \sum_{a=-\infty}^{\infty} x(a)k(i-a). \tag{2.4}$$

In a CNN, the input data $x(i)$ will only be defined for values of i in some finite interval $[i_1, i_2]$ corresponding to, for instance, the width or height, in number of pixels, of an image or number of samples in a time series. The kernel $k(j)$ is set to have non-zero values only for $j \in [-r, r]$ where r typically is small, hence the output $s(i)$ is only affected by input points $x(i')$ with $|i' - i| \leq |r|$. In 2-D image recognition the input and the kernel are 2-D arrays; the input representing the pixel values of the image and the kernel consisting of parameters to be learned by the training algorithm. Employing one convolution operation with one type of kernel can, loosely speaking, only extract one type of property. It is therefore common that multiple convolutions with different kernels, which are computed in parallel to extract more kinds of features. These parallel convolutions result in the output being wider than the input. The discrete convolution operation can be realized by matrix multiplication as illustrated in figure 2.1.
A convolution operation in a CNN is usually accompanied by a pooling operation that computes a summary of nearby outputs. The max pooling operation for example, computes the maximum value of a rectangular region in the output. Pooling introduces a small degree of translational invariance, meaning that a small shift in the input data does not affect the output [10, p. 342]. This property is very important in object recognition since it reduces the relevance of the exact position of an object. A typical layer in a CNN consists of three parts, a number of parallel convolutions, non-linear activation functions and finally pooling.

2.2.3 Optimization Algorithms

The goal of a MLP is to approximate some function $\hat{\mathbf{f}}(\mathbf{x})$ by minimizing the cost function $g(\boldsymbol{\theta})$, an error measure of the approximation, with respect to the parameters $\boldsymbol{\theta}$. A common cost function used for classification tasks is the *cross-entropy* defined

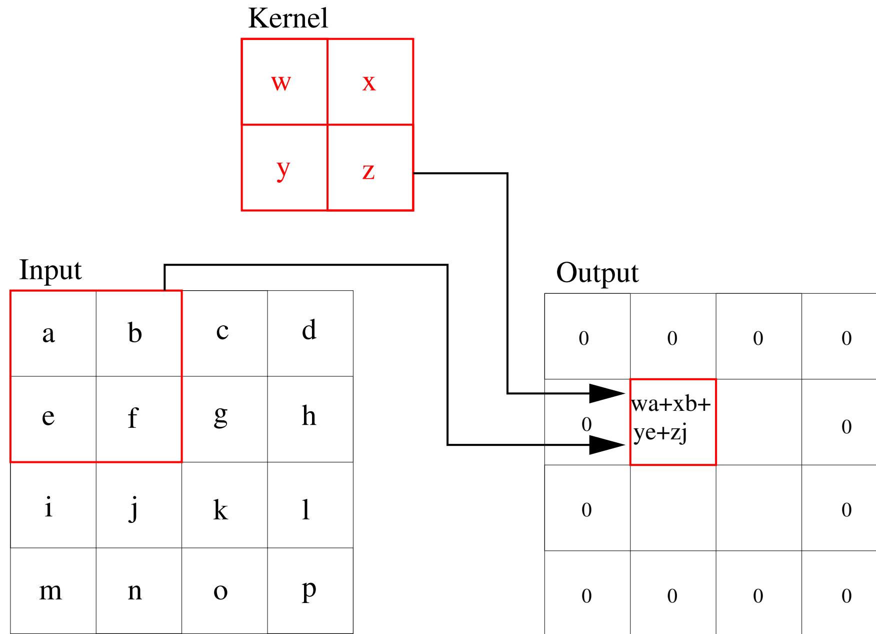


Figure 2.1: An example of a two-dimensional pooling operation with a stride between the pooling operations of two. In the current case, the stride size of two means that only four pooling operations are needed and will result in an output dimension of [2x2]. In order to the obtain output of the same size as the input, additional zeros can be added around the border, called zero padding.

as

$$H(p,q) = -\sum_i p_i \log q_i \tag{2.5}$$

where q is the predicted probability distribution of the model and p the true distribution [8]. The cross-entropy will be used as cost function for all classification models in this thesis.

In gradient-based learning the cost function is minimized iteratively by computing the gradient with respect to the parameters $\boldsymbol{\theta}$, which are then updated by taking a small step in the gradient direction. In gradient descent [10, p. 82], or steepest descent, the next point is found by

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n + \epsilon \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}_n), \tag{2.6}$$

where ϵ is the learning rate and the gradient is calculated on the whole training set. Computing a gradient on the whole training set is not practically feasible when the dataset is large. Additionally, gradient descent is prone to get stuck in local minima since the cost function is generally not convex. A computationally less expensive optimization algorithm is the stochastic gradient descent (SGD) [10, p. 294]. In SGD, the gradient is estimated by calculating an averaged gradient of a loss function $\hat{g}_m$ acting on a small batch, called a minibatch, of m examples randomly sampled from the dataset. Since the gradient estimate is calculated from a fixed size minibatch of examples, the time per step does not grow when the dataset is increased. The estimated gradient will introduce noise due to the random minibatch

sampling, hence the algorithm is less prone to end up in local minima. In practice, the learning rate ϵ needs to be gradually decreased to compensate for noise when the global minimum is approached.
Adopting the stochastic gradient descent optimization for learning can in some cases be slow, for instance due to noisy gradients. In order to decrease the learning time, many optimization algorithms implement the method of momentum [10, p. 296]. The algorithm acquires a momentum, an average of previous gradients, that reduces the noise from the individual minibatch gradients, smoothing the path towards the global minimum. The variable $\mathbf{v}$ plays the role of momentum and the values are updated in each step according to

$$\begin{aligned} \mathbf{v}_{n+1} &= \alpha \mathbf{v_n} - \epsilon \nabla_{\boldsymbol{\theta}} \hat{g}_m(\boldsymbol{\theta}_n), \\ \boldsymbol{\theta}_{n+1} &= \boldsymbol{\theta}_n + \mathbf{v_n}, \end{aligned} \tag{2.7}$$

where $\alpha \in [0, 1)$. The algorithm accelerates in one direction, if a similar gradient $\nabla_{\boldsymbol{\theta}} \hat{g}_m$ is obtained at each step, until it reaches the maximum step size

$$\frac{\epsilon ||\nabla_{\boldsymbol{\theta}} \hat{g}_m||}{1 - \alpha}. \tag{2.8}$$

Along with the learning rate ϵ, α has an impact on the time consumption and convergence of the learning algorithm. Tuning the initial values of these hyperparameters and how they adapt during training is important although can be time consuming in itself. Algorithms with adaptive learning rates are designed to find the best learning rates for each individual parameter of $\boldsymbol{\theta}$ and adapt them during the optimization. Although algorithms with adaptive learning rates have shown robust performance, no single overall best algorithm exists.

Chapter 3

Setting up a working machine learning model

3.1 Simulations using ggland/GEANT4

In order to train neural networks, a large amount of data is required. In this work, this data is gathered by simulations using ggland [11], a wrapper program intended to simplify simulations with GEANT4 [12] in different experimental setups. GEANT4, provided by an international collaboration including CERN, is a software toolkit for simulating the passage of particles through matter.

3.1.1 Light propagation in scintillating paddles

When simulating a charged particle traveling through a scintillating paddle in NeuLAND, GEANT4 provides ggland with the energy loss of that particle due to Coulomb interaction i.e. the energy transferred to excite the molecules in the scintillating material. This means that the energy value provided by the detector in the simulations correspond to that of an ideal detector, not accounting for subsequent losses of scintillation photons propagating towards the PM tubes.

3.1.2 Neutrons in NeuLAND

The geometry and functionality of NeuLAND is easily replicated with ggland and all that remains to obtain a working simulation is to define a particle source, called gun.

In the simulations, NeuLAND was placed 15 meters from a neutron source along the z-axis (figure 1.1). Neutrons with a momentum directed randomly within an angle of 2.4° relative to the z-axis, corresponding to a disc with diameter 1.25 m on the face of the detector, were generated simultaneously with energies ranging from 550 to 650 MeV. All PM-tube signals with time values larger than 200 ns after the neutron generation in each event were removed to avoid hits corresponding to, for instance, particle decay. The hit coordinates were converted into discrete values on a grid with an x, y and z axis consisting of 50, 50 and 60 points respectively. Each

event is now represented as a sparse 3-D image with two pixel values corresponding to time and energy.
For each neutron multiplicity ranging from 1 to 7, roughly 200 000 events were simulated. The data was separated into a training, a validation and a test dataset of 95 000, 1 600 and 100 000 images, respectively, for each neutron multiplicity. The images were split into different text files, one each for a given number of generated neutrons. Events with zero neutrons are needed in order to properly train the models and were created by simply writing blank events to a text file, consequently avoiding special treatment of zero neutron events.

3.1.3 What do neutrons in plastic do?

A neutron-nucleus collision in the plastic scintillators of NeuLAND can create or liberate a variety of particles from the nucleus. In order to get an intuition of products resulting from a neutron colliding with a nucleus in the plastic, neutrons incident on a 0.01 cm thick plastic slab are simulated. If the neutron deposits less than 5 MeV with no other particles emitted, it is considered a small-angle elastic scattering and ignored. For each neutron energy, a total of 10^8 neutrons illuminate the plastic slab with approximately 14 000 resulting in a reaction. This corresponds to a mean free path of 71 cm, and furthermore, that 1.5 % of the neutrons entering NeuLAND pass straight through without interacting.
It is worth noting that only one of the reactions, presented in table 3.1, lacks an outgoing neutron, additionally, 12.9 % of the 500 MeV neutron collisions result in ghost hits, where a neutron solely knocks out another neutron. Most of the reactions also produce gamma rays, although, which travel far in the detector without interacting, and are thus hard to associate with a certain neutron-nucleus collision.

3.2 Network implementation in TensorFlow

A significant part of this work was spent on understanding and creating a working TensorFlow program. Even though TensorFlow is well documented, there is a significant learning curve, especially for a user inexperienced in both, machine learning and TensorFlow. This chapter provides a short introduction to the TensorFlow components used when building the neural networks presented in this thesis and how they are implemented in Python code.
TensorFlow is an open source software library written in Python and C++ with multiple Application Programming Interfaces (APIs) at different levels. The different levels of the APIs makes implementing a simple machine learning algorithm straight-forward, while still giving the user a lot of freedom and control. A TensorFlow program is created by defining a computational graph representing the machine learning algorithm and a suitable optimization algorithm for minimizing the loss function. Many of the usual loss functions and optimization algorithms are defined in TensorFlow making it simple to use them in a model.
Installing TensorFlow with CPU support is straightforward and only requires an existing Python installation. However, installing TensorFlow with GPU support has

Table 3.1: The 10 most probable products resulting from simulated neutron-nucleus collisions in the scintillator plastic for neutrons with kinetic energy of 100, 500 and 1000 MeV, respectively. Neutrons which have deposited less than 5 MeV with no other particles emitted, are ignored. Most reactions result in additional gamma rays which are not presented in the tables. n: neutron, p: proton, d: deuteron, α: alpha particle, π: pion.

100 MeV, 14,109 reactions

Products	%
n, p	33.3
n	16.4
2n	9.6
n, α	7.5
2n, p	7.1
3n, 2p	3.1
n, 2α	2.8
2n, p, d	2.6
3n, p	1.5
n, 2p	1.4

500 MeV, 12,578 reactions

Products	%
n, p	28.0
2n	12.9
2n, p	6.2
3n, 2p	5.8
n	5.2
3n, 2p α	3.2
2n, p, d	2.3
n, α	2.1
3n, p	1.9
n, p, π^-	1.8

1000 MeV, 14,616 reactions

Products	%
n, p	24.1
2n	7.4
n, p, π^-	6.4
3n, 2p	4.0
2n, p	3.8
2n, π^+	3.1
n	2.9
3n, 2p, α	2.7
2n, 3p, π^-	1.8
p, π^-	1.6

some additional requirements making it somewhat tedious. For instance, TensorFlow is only compatible with Nvidia GPUs that have CUDA Compute Capability 3.0 or higher [13]. The TensorFlow webpage [8] provides detailed instructions of all additional libraries and drivers needed and how to install them.

3.2.1 Building a TensorFlow graph

The basic building blocks of a TensorFlow program are tensors, represented by n-dimensional arrays of a certain data type. All parameters needed in a model will be represented by these tensors, for instance, the input data and all nodes and layers in a neural network. A TensorFlow program consists of multiple tensors that are computed according to a computational graph.

In the case of building a graph for a multilayer perceptron, the first step is to create a *placeholder* tensor representing the input data, i.e. setting the same data type and dimension as the desired input data. The placeholder creates an empty tensor that requires values to be provided during execution. There are two additional tensors, namely *constants* and *variables*. As the names imply, the values of a constant can only be set once and never be changed, while the values of a variable can be updated any time during execution. The variables will be the tensors representing the trainable parameters in the network, for instance the bias b and weights W in equation 2.2.

Once the bias and weights for the first layer in the network have been defined, multiplication of the tensors can be implemented with a TensorFlow function called 'tf.matmul'. To add an activation function acting element-wise on the output from the tensor multiplication, the output should be provided as an argument to the

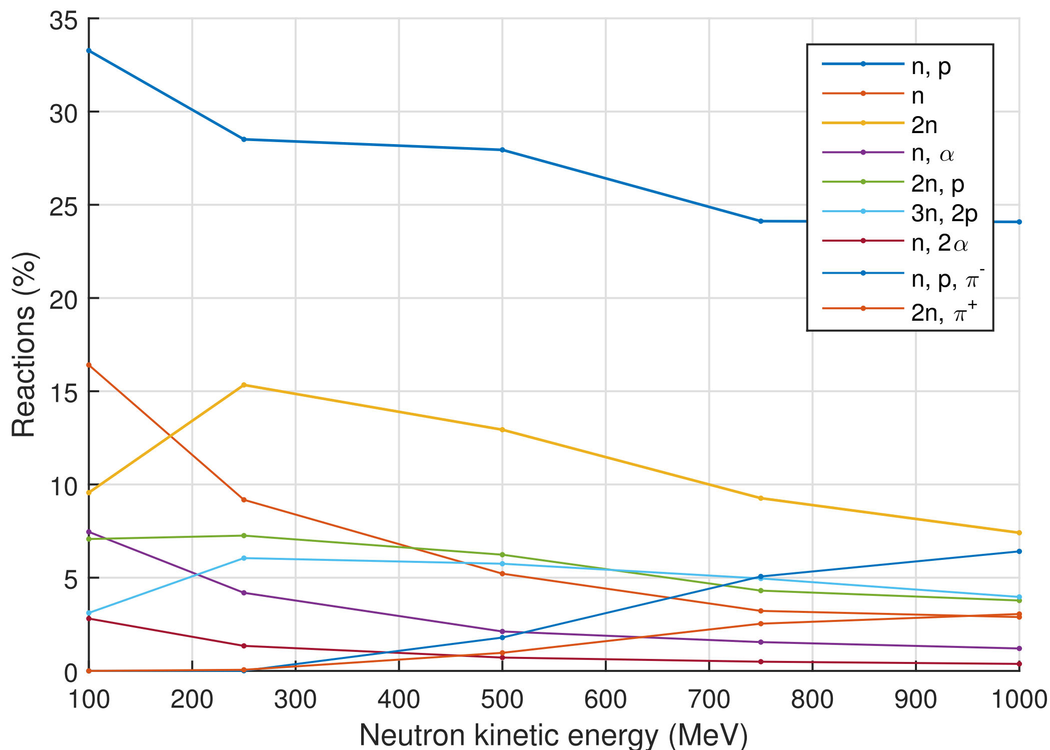


Figure 3.1: Simulated reaction probabilities of neutrons incident on scintillator material. Neutrons which have deposited less than 5 MeV with no other particles emitted, are ignored. Most reactions result in additional gamma rays which are not presented in the figure. n: neutron, p: proton, d: deuteron, α: alpha particle, π: pion.

desired activation function defined in TensorFlow.

Listing 3.1: The code creates two fully-connected layers with 2 and 128 nodes, respectively, the latter having rectified linear activation functions. The input tensor is set to have the dimension `[None, 2]`, where None means that the dimension can be of any length. This allows to process a batch of any number of events during execution.

```
1  """ Input """
2  x = tf.placeholder(tf.float32, [None, 2])
3  """ Weights """
4  W = tf.Variable(tf.truncated_normal([2, 128], stddev=0.1))
5  """ Bias """
6  b = tf.Variable(tf.constant(0.1, [128]))
7  """ Matrix multiplication """
8  y = tf.matmul(x, W) + b
9  """ Rectified linear activation function """
10 relu = tf.nn.relu(y)
```

It is important to initialize the weights with noise, to avoid zero-gradients and break eventual symmetry that can aggravate the optimization [8]. In the code above and

the models presented in this thesis, all weights are initialized by sampling from a truncated normal distribution with zero mean and standard deviation of 0.1, where values whose magnitude is more than 2 standard deviations from the mean are dropped and re-picked. The bias values are all set to 0.1.
Altering the code above, it is straight-forward to expand the model with more layers and nodes along with different activation functions or even increasing the input dimension and performing convolution and pooling operations. The next step is to apply a desired loss function and optimization algorithm.

Listing 3.2: The cross-entropy is calculated on top of a softmax function applied to the final layer. The mean value of the batch is then computed and passed to the ADAM optimizer (see section 4.1). The tensor `y_final` represents the output from the final layer.

```
1  """ The correct output from the training datset """
2  y_ = tf.placeholder(tf.float32, [None, 8])
3  """ Cross entropy of the softmax output of the final layer """
4  cross = tf.nn.softmax_cross_entropy_with_logits(labels=y_,
5                                                  logits=y_final)
6  """ Compute the mean over the batch """
7  cross_mean = tf.reduce_mean(cross)
8  """ Set the ADAM algorithm as optimizer """
9  train_step = tf.train.AdamOptimizer().minimize(cross_mean)
```

3.2.2 Running a TensorFlow graph

Thus far, we have all we need to create a graph representing a simple multilayer perceptron. In order to train the perceptron, a TensorFlow session is required. When run, the session performs a single step of the chosen optimization algoritm, altering the variables in the model to reduce the loss function. During the training, the session requires data being passed to it. Reading from a file can be done by adding file reading instructions to the TensorFlow graph, or by passing the data directly from python code. In this work, the whole file is read at the start of the program into a python object `dataset`. Python then feeds the data to the session for each run. The function `next_batch()` of the object `dataset`, generates a batch of data points from the dataset, which is then fed to session via `feed_dict`.

Listing 3.3: A TensorFlow session used to initialize all variables and run 1 000 training iterations.

```
1
2  """ Create the session """
3  sess = tf.InteractiveSession()
4  """ Initializing the variables """
5  init_op = tf.group(tf.global_variables_initializer(),
6                     tf.local_variables_initializer())
7  sess.run(init_op)
8
9  """ Run 1000 training iterations with batch size 50 """
10 for i in range(1000)
```

```
11      x_train, y_train = dataset.train.next_batch(50)
12      sess.run(train_step, feed_dict={x: x_train, y_: y_train})
```

3.2.3 Storing the TensorFlow graph

When dealing with large models, requiring up to days or weeks to train, it is important to monitor and save learned parameters periodically. The `MonitoredTrainingSession` object automates this process, the user only needs to specify a path to the desired directory where all model summaries will be stored. The method `tf.summary()` lets the user choose tensors in the model that will be stored in the summary. `MonitoredTrainingSession` also periodically generates checkpoint files, from which the training can be resumed. Upon starting a Monitored Training Session, the training is automatically started from the state of the model in the last saved checkpoint file.

3.2.4 Challenge of using TensorFlow

The main challenge encountered during this work was setting up a working TensorFlow program. TensorFlow provides detailed tutorials of the basic components, allowing to quickly set up a simple machine learning model, however, straying from the path of these tutorials required significantly more work. For an inexperienced user, understanding how to design the code demands some effort due to the way the user is required to build the TensorFlow graph. This type of programming has a significant learning curve. For instance, TensorFlow provides routines for sequentially reading files during training, but due to the time needed to understand them, an easier approach was chosen in this work by reading the whole file at the start of the program.

3.3 Data reading

Routines for sequentially reading data from files during training are often needed when processing large data sets of images. For instance, storing the training data set of 800 000 event images in floating-point format would require a storage size of about $800\,000 \cdot 300\,000 \cdot 4\text{B} = 960$ GB, which is too much to fit into the memory of a computer with a reasonable price. Although TensorFlow has good file reading routines, the sparse nature of the neutron event images make it possible to load all events into memory. The average amount of detected hits in each event is of the order of 100, which allows to load the coordinates along with time and deposited energy for each hit into memory. This reduces the memory needed to store 800 000 events to approximately 320 MB. The images are then created when used by inserting the time and energy values of each hit into a 3-D image with all pixel values initially set to zero. This is repeated for each batch of events during training.

Chapter 4

Developing and testing models

Developing a deep learning model often requires an extensive process of trial and error. Following the recommended procedure proposed by Ian Goodfellow, Yoshua Bengio and Aaron Courville [10, p. 421], this process is systematized and can be divided into three main parts. First, a working example of a simple CNN is set up as a baseline model. The performance of this model is analyzed in order to find potential improvements and the model is adjusted accordingly. This procedure is repeated until satisfactory results are achieved.
It is important to be able to determine to what extent the CNNs can extract geometrical properties from the 3-D images. Therefore, we start by evaluating models that only take the total deposited energy and the number of hits as input, serving as a performance reference. The architecture of the baseline CNN is chosen to be relatively simple such that the performance of different parts in the network can be assessed. By individually changing different parts in the network, for instance the type of activation functions or number of nodes in a layer, their effect on performance can be determined. In this way, the procedure becomes systematized and we obtain an indication of how further changes could improve performance.
A significant part of the work was spent while attempting to set up the baseline model. The main reason for this was the time-wise poor performance of using Central Processing Units (CPUs). The training could run for weeks without any indication of learning, only to find errors in the code resulting in, for instance, a network architecture different from the one intended. An initial benchmark, using data provided by the TensorFlow web page, indicated that the speed increase of using a Graphics Processing Unit (GPU) did not justify the work needed to acquire access to a suitable GPU (see table 4.5). Fortunately, this could later be thoroughly tested, showing that the speed increase was highly dependent on the size and architecture of the network, and performing about 145 times faster on the baseline model when trained on Nvidia's Tesla K80 GPU [14].

4.1 Simple multilayer perceptron as accuracy reference

In order to have a accuracy reference of how well the neutron multiplicity can be predicted by only accounting for the total deposited energy and the number of hits, two simple feedforward networks were created with the ambition of reproducing results similar to the ones presented in the NeuLAND technical design report [1, p. 57]. The method used in the technical design report additionally uses geometrical information of the events by merging nearby hits into "clusters" according to a certain routine.
The networks have two input nodes (total energy and number of hits), two fully-connected layers with 128 ReLUs each in the first network and 8000 ReLUs each in the second network and a final fully-connected layer with 8 ReLUs. In comparison, the CNN AlexNet [6] with input images of dimension [256x256x3], has two final 4096-node fully-connected layers and one 1000-node output layer.
The output nodes create a vector of dimension 8 where the index of the largest output value represents the predicted neutron number. The loss function is the cross-entropy of a softmax function applied to the final layer and the minimization was achieved with the ADAM optimization algorithm [15] and a batch size of 200 images. The ADAM optimization, which uses adaptive learning rates, is used throughout this thesis, with the parameters set to the same values as proposed by Diederik P. Kingma and Jimmy Lei Ba [15]. Every 2 000 steps the models were evaluated on the validation dataset containing 1 600 images for each neutron multiplicity (figure 4.1). At the final step of training the models were tested on the large independent test dataset containing 100 000 images for each neutron multiplicity (table 4.1).

		generated					
	%	0n	1n	2n	3n	4n	5n
detected	0n	**100**	3	0	0	0	0
	1n	0	**93**	17	2	0	0
	2n	0	4	**75**	26	4	0
	3n	0	0	9	**59**	28	6
	4n	0	0	0	13	**47**	23
	5n	0	0	0	0	20	**56**
	6n	0	0	0	0	0	12

		generated					
	%	0n	1n	2n	3n	4n	5n
detected	0n	**100**	2	0	0	0	0
	1n	0	**89**	14	1	0	0
	2n	0	9	**82**	31	6	1
	3n	0	0	4	**38**	14	2
	4n	0	0	0	29	**69**	39
	5n	0	0	0	0	10	**46**
	6n	0	0	0	0	0	8

Table 4.1: Neutron identification matrices given by the two-layer fully-connected models. Both models are tested on the test set containing 100 000 images for each neutron multiplicity. **Left**: Model with 128 nodes in each layer. Overall correct predictions: 71.8 %. **Right**: Model with 8000 nodes in each layer. Overall correct predictions: 70.8 %.

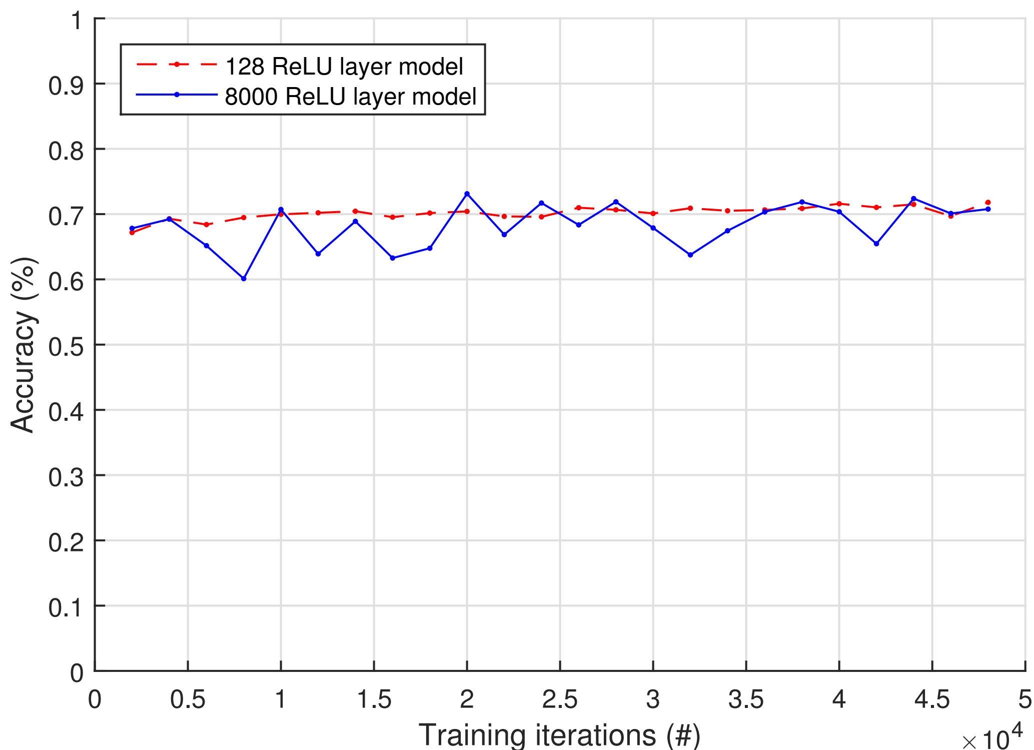


Figure 4.1: The accuracy of the two-layer fully-connected models tested on the validation set containing 1 600 images for each neutron multiplicity.

4.2 Baseline CNN model

As a starting point for image recognition, a convolutional neural network inspired by AlexNet [6], consisting of layers with convolution and average pooling together with a few last fully-connected layers, was employed. Specifically, the network has three convolutional layers, with two convolution operations, one pooling step each, and three fully-connected layers. The convolution operations are 3-D, acting on patches of size [3x3x3], with stride sizes of one. All nodes in the network are ReLUs. Table 4.2 shows the output dimension, number of stored signals and parameters needed for each operation, providing an idea of the memory consumption of the network.
The final test of the baseline CNN model (table 4.3) shows a large improvement in accuracy compared to the MLP used for accuracy reference (table 4.1): the percentage of overall correct predictions increase from 71.8 % to 78.3 %.

4.3 Attempting to speed up the learning process

The time needed to train the baseline model proved to be long, with weeks passing before any indication of learning was seen. As an attempt to speed up the learning process, the training was executed on multiple CPUs (table 4.4), i.e. distributed training, and also on GPUs.
Between-graph replication and synchronous training [16] were implemented in the

Table 4.2: Baseline CNN architecture

Action	Output size	Signals (#)	Parameters (#)
Input:	[2x50x50x60]	300,000	0
Conv3-16:	[16x50x50x60]	2,400,000	16*(3*3*3*2+1) = 880
Conv3-16:	[16x50x50x60]	2,400,000	16*(3*3*3*16+1) = 6,928
PoolAvg2:	[16x25x25x30]	300,000	0
Conv3-32:	[32x25x25x30]	600,000	32*(3*3*3*16+1) = 13,856
Conv3-32:	[32x25x25x30]	600,000	32*(3*3*3*32+1) = 27,680
PoolAvg2:	[32x13x13x15]	81,120	0
Conv3-64:	[64x13x13x15]	162,240	64*(3*3*3*32+1) = 55,360
Conv3-64:	[64x13x13x15]	162,240	64*(3*3*3*64+1) = 110,656
PoolAvg2:	[64x7x7x8]	25,088	0
FC:	[12544x1x1x1]	12,544	(64*7*7*8+1)*12,544 = 314,716,416
FC:	[12544x1x1x1]	12,544	(12,544+1)*12,544 = 157,364,480
FC:	[8]	8	(12,544+1)*8 = 100,360
Total:		∼7.1M	∼472M

Total memory of signals: ∼7.1M → 7.1M*4 B = 28.4 MB (per image)
Total memory of parameters: ∼472M → 472M*4 B = 1.9 GB

		generated					
	%	0n	1n	2n	3n	4n	5n
detected	0n	**100**	0	0	0	0	0
	1n	0	**96**	8	0	0	0
	2n	0	4	**84**	17	1	0
	3n	0	0	8	**73**	23	2
	4n	0	0	0	10	**61**	26
	5n	0	0	0	0	14	**56**
	6n	0	0	0	0	1	15

Table 4.3: Neutron identification matrix given by the baseline CNN model. The model is tested on a test set with 100 000 images for each neutron multiplicity. Overall correct predictions: 78.3 %.

distributed training program such that each CPU had its own replica of the model/graph/network. Each CPU loads the whole training data set into memory and randomly samples a batch of images from which it computes a gradient of the loss function. One of the CPUs is selected to monitor and control the rest of the CPUs, called *chief*, which collects an average of a predetermined number of gradients calculated by the other CPUs. The chief then updates the parameters of the model according to the learning algorithm and finally passes them back to the other CPUs. Although running the training distributed even on multiple CPUs did speed up the training process, running on a GPU turned out to be less time consuming in both setting up and running the training. Training and using deep neural networks

Table 4.4: Distributed CPU test. CPU 2 and CPU 3 have similar computing power while CPU 1 is slightly more powerful. We do not compare the specification of the CPUs, instead we benchmark their performance separately (single). When the training is distributed on multiple CPUs, each CPU computes an average gradient over the indicated batch size, i.e when using two (three) CPUs, 2 (3) batches are processed. Initial to a CPU processing a new batch of images, all CPUs have to have finished their current batch and we see that a larger number of CPUs participating results in a deacreased training speed per CPU. (seconds per image: s/i).

Batch size	CPU 1 (single)	CPU 2 (single)	CPU 3 (single)	CPU 1, CPU 3	CPU 1, CPU 3, CPU 2
1	9 s/i			18 s/i	
2	16 s/i			20 s/i	
10	64 s/i	85 s/i	83 s/i	45 s/i	38 s/i

involves a lot of vector and matrix operations; operations which GPUs excel at. In order to demonstrate the need of GPUs to train the CNNs considered in this thesis, three CNNs with different sizes and architecture are trained on a CPU and two different GPUs. The CPU used is an Intel Xeon E5-1650v2 [17] and the two GPUs, an Nvidia Tesla K80 [14] and an Nvidia Geforce GT 750M [18] (for laptop). The baseline CNN (table 4.2) and the CIFAR-10 model used in Tensorflow's deep learning tutorial [19] were compared to each other. The third model (which we for now call the *reduced baseline* model), has the same architecture as the baseline CNN model, but with 4 096 nodes instead of 12 544, making it small enough to fit in the lower memory (4 GB) of the laptop GPU. The CIFAR-10 model consists of two layers with one convolution and pooling each and finally two fully-connected layers. The CIFAR-10 model takes 24x24 pixel RGB images as input, corresponding to 1728 values compared to 300 000 for the NeuLAND baseline CNN model. The results in table 4.5 show that Nvidia's Tesla K80 GPU performs about 7 times faster compared to the Intel Xeon CPU for the CIFAR-10 model and about 145 times faster for the baseline model.
Taking advantage of the large speed increase accompanied with the use of GPUs, further training was performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at High Performance Computing Center North (HPC2N). Our project was approved 40 000 core-hours per month on the supercomputer Kebnekaise [20], corresponding to roughly 1.5 Tesla K80 GPUs running continuously. Kebnekaise, installed during the summer 2016, consists of 468 Intel Xeon E5-2690v4 CPUs, 20 Intel Xeon E7-8860v4 CPUs, 36 Intel Xeon Phi 7250 CPUs and 80 Nvidia Tesla K80 GPUs.

4.4 Uncertainty in training

During training of the baseline CNN model, the accuracy achieved with the validation dataset was subject to significant fluctuations, making the final performance highly dependent on when the training was stopped. Stopping at a maximum in or-

Table 4.5: Intel Xeon E5-1650v2 CPU vs. Nvidia Tesla K80 GPU vs. Nvidia GeForce GT 750M

Model	Intel Xeon E5-1650v2 CPU	Tesla K80 GPU	GeForce GT 750M GPU	Batch size	GPU speedup
CIFAR-10	0.22 s/batch	0.032 s/batch	-	128 images	∼7
Baseline	113 s/batch	0.77 s/batch	-	15 images	∼145
Reduced base.	113 s/batch	0.60 s/batch	2.65 s/batch	15 images	∼188/∼43
Approx. cost	4.5 kSEK	46 kSEK	2 kSEK		

der to achieve high performance (early stopping) is perfectly fine, although not well suited when comparing different models. For instance, the two fully-connected models used as accuracy reference (figure 4.1) show different behaviour during training. The accuracy of the 8000-node model reaches a higher maximum, but has a lower average than the 128-node model, due to large fluctuations. Although the 8000-node model reaches a higher maximum, it is difficult to argue for reproducibility when trained on a new independent dataset.
We perform five independent training runs of the baseline CNN model and compute the mean value and standard deviation of the last 21 validation steps, i.e. between training step 140 000 and 162 000, for each run (figure 4.2). The fact that all five mean values are well within one standard deviation of each other, indicate that the fluctuations during a single run have a larger impact than the variations between different runs. In figure 4.3, the variation from different training runs is compared to the fluctuations during a single run. For the single run, the graph is smoothed by a moving average filter along with a shaded area showing the standard deviation of nearby datapoints. The second graph shows the mean value and standard deviation of each validation step for the five independent runs.
Henceforth, when comparing the performance of different models, the mean value of the last 21 validation steps is used. For simplicity, we refer to this as the *end mean* value.

4.5 Improving the model

4.5.1 Removing events from the training set

Neutrons entering the detector have a significant probability[1] of passing straight through without any interactions and are then impossible to detect. In the case where neutrons have passed through undetected we want the model to predict the number of neutrons that have actually been detected. The events with undetected neutrons will have a negative effect on training because they will always have a higher neutron multiplicity than what the event image is indicating. The learning algorithm will see images that are characteristic to, for instance, four neutrons but

[1] According to the simulated values in section 3.1.3, approximately 1.5 % of the neutrons do not react in the detector at all.

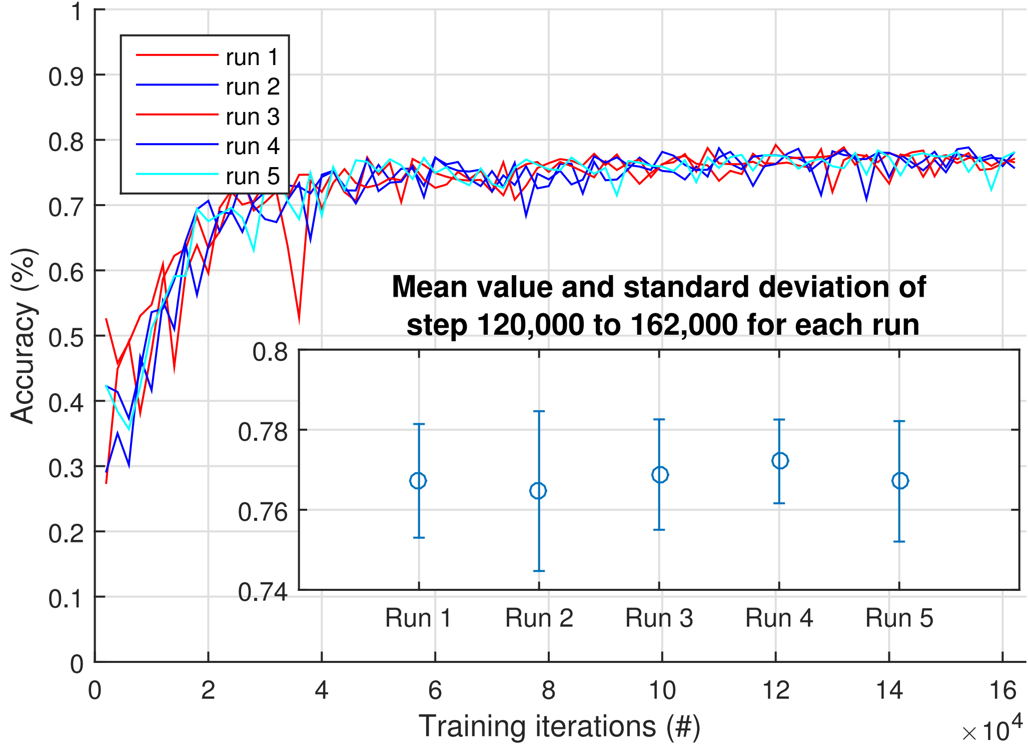


Figure 4.2: The accuracy of the baseline model of five independent training runs, for every 2 000 steps, tested on the validation set containing 1 600 images for each neutron multiplicity. The inset graph shows the mean value, with error bars of the standard deviation, over the steps from 120 000 to 162 000.

will be told that it shows both four and five neutrons or even higher in worse scenarios. In order to remove this effect a lower energy limit is introduced. Each individual neutron has to deposit a minimum of 50 MeV in the detector or the event will be removed from the training set, ensuring that all neutrons have been detected. Retraining the baseline CNN model on the modified dataset resulted in an end mean of (77.3 $\pm$1.6) % in comparison to (76.7 $\pm$1.5) % for the original dataset. The error gives the corresponding standard deviation.

4.5.2 Adding total deposited energy and number of hits

As a first step of improving the baseline model, we extend the architecture by adding the total deposited energy and the number of hits as input. There is no reason to assume that the network will, on its own, calculate these values. Two ReLUs, taking the total deposited energy and number of hits as input, are concatenated with the output of the last pooling operation, altering the output size from 25 088 to 25 090. Adding the two nodes increases the number of parameters by 25 088 which is negligible compared to the 472 million already existing parameters, thus not noticeably affecting training time. We refer to this model as the *extended baseline model*.

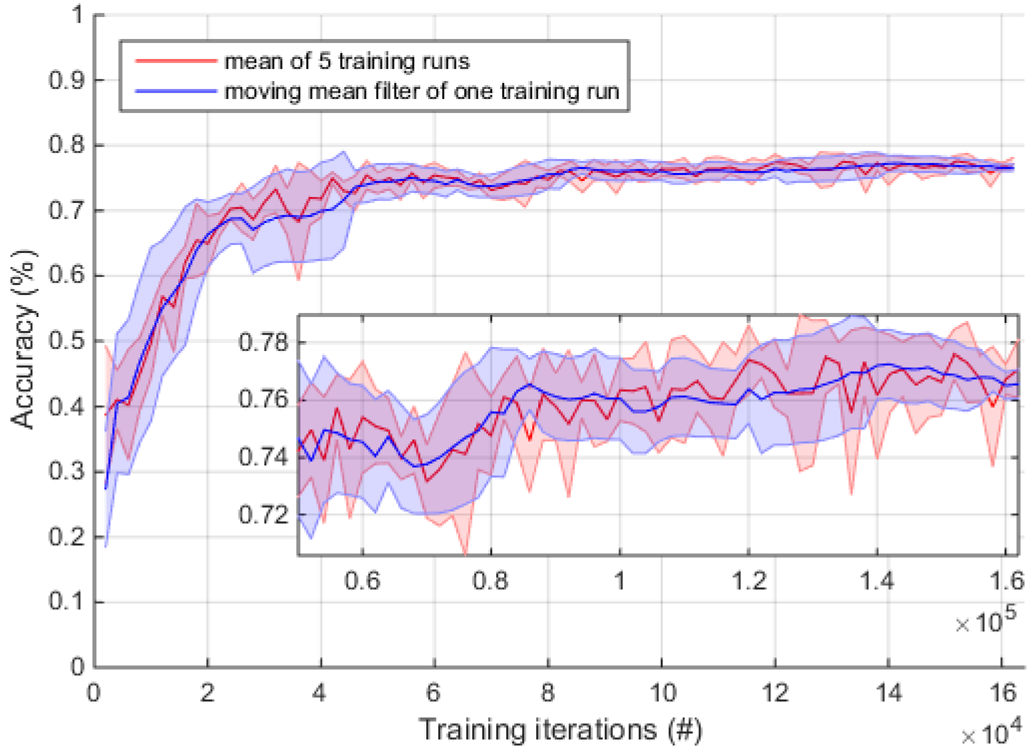


Figure 4.3: The accuracy of the baseline model, for every 2 000 steps, tested on the validation set containing 1 600 images for each neutron multiplicity. **Red**: The mean value and standard deviation (shaded area) of 5 individual training runs for each step in training. **Blue**: The accuracy of a single training run, smoothed with a moving average filter along with a shaded area representing the standard deviation of nearby data points.

The accuracy on the validation set shown in figure 4.4, indicates that no learning was achieved without properly normalizing the input data (discussed in next section). When the input data was properly normalized, the training resulted in an end mean of (77.1 ±1.4) % as compared to (76.7 ±1.5) % for the baseline CNN model.

4.5.3 Data normalization

The absolute values of the total deposited energy and the energy deposited in each hit will on average be of different scales; the total energy will be larger when measured in MeV. When the optimization algorithm alters a parameter in the network, it will have a larger relative effect on the the larger input values. In practice, this imposes difficulties for the learning algorithm, which is also the current case. In order to avoid this, the input data is normalized per channel to be in the interval [0, 1]. Here, the pixel energy values, pixel time values, total energy and total hits are different input channels. Each channel is divided by the channel's maximum value of the entire dataset.
Normalizing the data in this way seems to solve the problem of having input channels

of varying magnitude (figure 4.4), although, with an end mean of (72.6 ±1.3) %, showing a lower performance than the baseline CNN model.
Depending on the amount of energy deposited per particle, the images will vary in brightness, possibly affecting how well the network can extract important features. This is examined by normalizing each image separately such that they will have approximately the same brightness. Two normalization procedures are evaluated, the first was achieved by, per channel, subtracting from each pixel value the mean value and dividing by the standard deviation such that the images have pixel values with zero-mean and a standard deviation of one. The second approach was to divide each pixel value by the image maximum, thus limiting all values to the region [0, 1]. The two per-image normalization techniques both proved to give an increased accuracy (figure 4.5), performing similar to the baseline CNN model. The zero-mean normalization resulted in an end mean of (77.1 ±1.4) % and the [0, 1] normalization in (76.7 ±0.7) %. Henceforth, the zero-mean normalized data will be used during training.

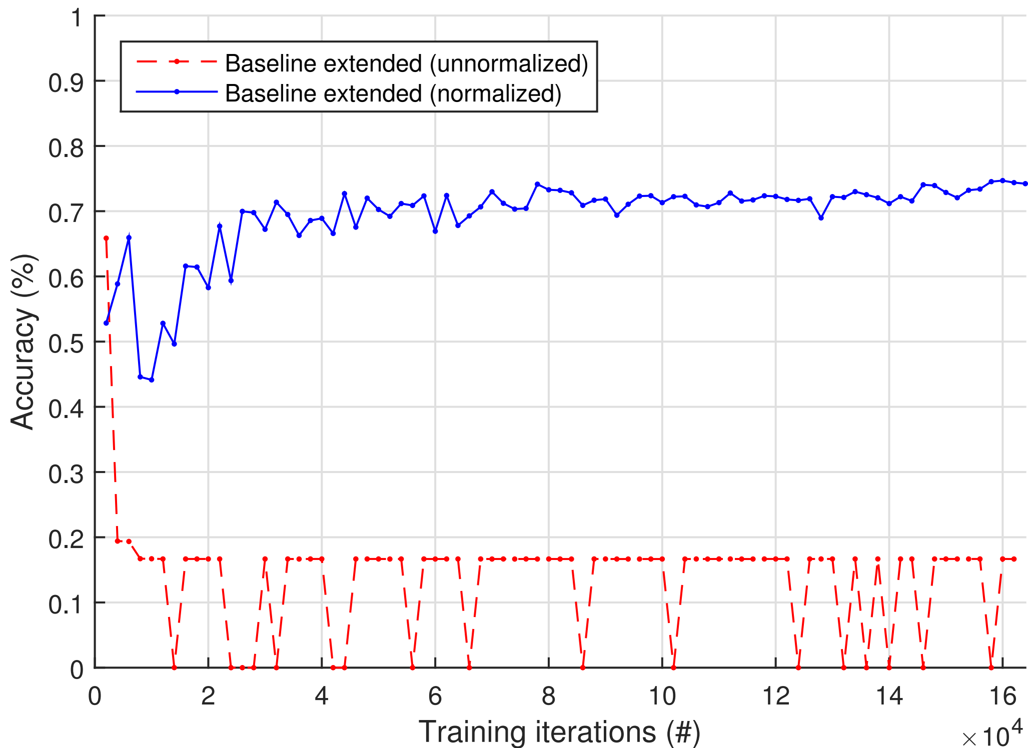


Figure 4.4: The accuracy of the extended baseline model accounting for total energy and number of hits, tested on the validation set containing 1 600 images for each neutron multiplicity. The model is trained on both normalized and unnormalized data. The normalization is obtained by dividing each channel by the channel's maximum value of the entire dataset.

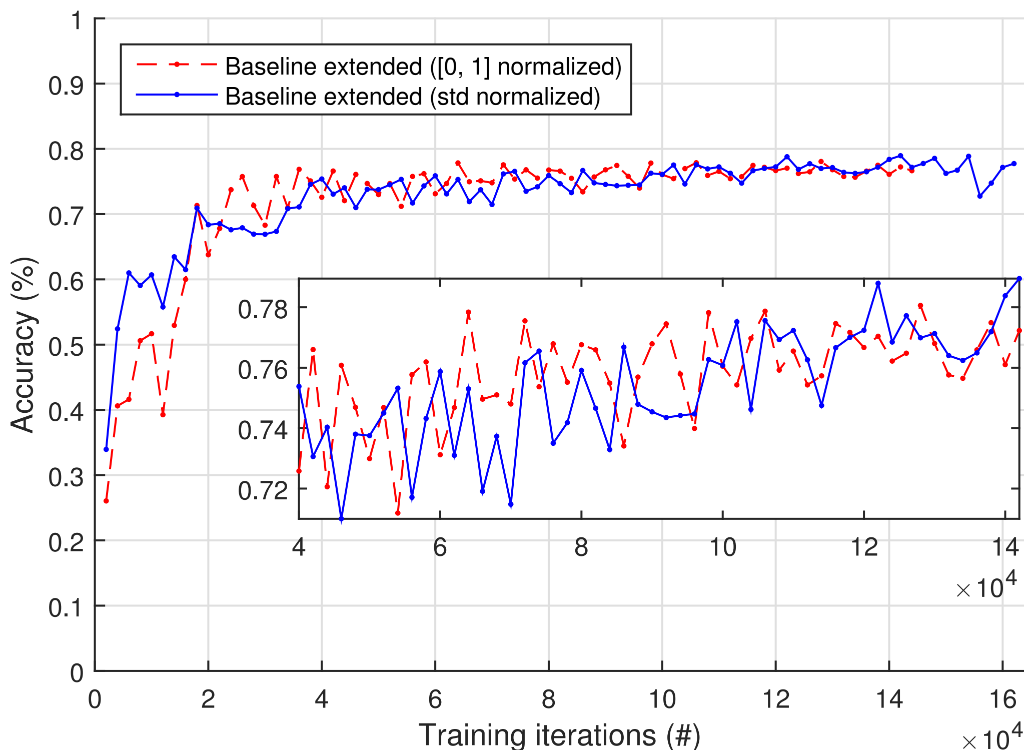


Figure 4.5: The accuracy of the extended baseline model accounting for total energy and number of hits, tested on the validation set containing 1 600 images for each neutron multiplicity. The model is trained on both the zero-mean and [0, 1] normalized data.

4.5.4 Reducing the training dataset

Up to this point, the accuracy on the validation dataset has shown no significant discrepancy from the accuracy on the training dataset, and therefore no indication of overfitting. Whether this is due to the model having a low capacity or that the training period is too short in comparison to the size of the training dataset is not obvious. By reducing the size of the training dataset from 100 000 to 10 000 images per neutron multiplicity, the extended baseline model starts overfitting the training data after approximately 40 000 training iterations, also at which the validation error starts increasing (figure 4.6). Although we cannot state whether the capacity of the model is adequate for the large dataset, the low discrepancy of the training and validation accuracy suggests a potential performance increase by extending the training period.

4.5.5 Variations of the extended baseline model

A few different variations of the extended baseline model where evaluated without any increase in performance. All changes were done separately with regard to the extended baseline model. The effect on the end mean values from three of the variations was too small to make any statement of the effect on performance. The

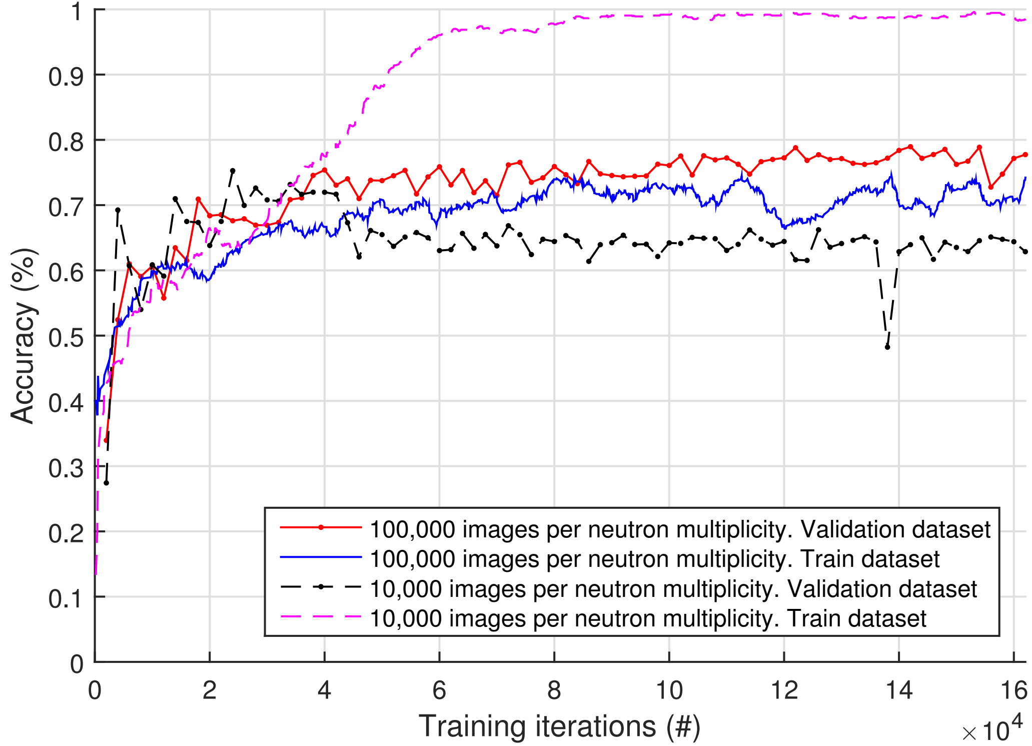


Figure 4.6: The accuracy of the extended baseline model accounting for total energy and number of hits, tested on the validation set containing 1 600 images for each neutron multiplicity. The model is trained on the entire training dataset containing 100 000 images per neutron multiplicity and a subset only containing 10 000 images per neutron multiplicity. The curve of the accuracy on the training dataset is smoothed with a moving average filter over nearby data points.

variations included the following changes to the extended baseline model:

- changing the training batch size from 15 to 30 and 50 images, respectively,
- changing the number of nodes from 12 544 to 4 096 in each of the FC-layers,
- increasing the number of parallel convolution operations by a factor of two in each layer and reducing the number of nodes in the FC-layers from 12 544 to 8 000.

The time per iteration during training increased from approximately 0.77 s to 1.23 s and 1.75 s for training with batch size of 30 and 50 images, respectively, and additionally had no effect on the accuracy as a function of iterations. In conclusion, the training period was increased without affecting the accuracy.

Reducing the number of nodes to 4 096 in the FC-layers decreased the time per iteration to around 0.6 s, i.e. reducing the training time by roughly 22 %. The fact that the model performs similar when the number of parameters in the network is reduced by almost 75 %, suggests that the baseline model has as an excessive number of nodes.

Changing all average pooling operations into max pooling operations, on the other hand, did have a negative effect on the accuracy, resulting in an end mean value of

(76.6 ±1.1) %. The decrease of 1.4 percentage points from the extended baseline model is larger than the standard deviation, and is therefore more likely caused by a lower performance of the model.

4.5.6 Increasing the number of layers

Since the variations of the baseline model did not show any clear increase in performance, a different architecture is evaluated (table 4.6). The network has five convolutional layers with two convolution operations and one pooling each, and four fully-connected layers. Since this network has two additional pooling operations, the final convolutional layer has an output of dimension of [256x2x2x2] instead of [64x7x7x8] for the extended baseline model, i.e. the number of output signals of the final convolutional layer is reduced from 25 088 to 2 048. Since the performance was unaffected when the number of nodes was reduced in the extended baseline model, we reduce the number of nodes in the fully-connected layers to 2 048. While the memory consumption of the signals remains roughly the same, the number of parameters is significantly decreased, which results in a 20 % shorter training time per image batch. The model resulted in an end mean of (78.2 ±1.3) %, compared to (77.1 ±1.4) % for the extended baseline model with per-image zero-mean normalization.

4.6 Evaluating the individual neutron multiplicities

The learning algorithm is designed to maximize the total accuracy without any limitations on the individual neutron multiplicities. This allows the learning algorithm to change the accuracy of the individual multiplicities in any way as long as the total accuracy is increased. The individual accuracies on the validation dataset during training show large fluctuations (figure 4.7).

4.7 Accounting for light loss due to Birk's law

In an attempt to approximate the light yield in the paddles, Birk's law [21] is implemented in the simulations. Birk's law is an empirical formula for the light yield per path length as a function of the energy loss per path length for a particle traversing a scintillator. Both the baseline model and the two-layer fully-connected model only accounting for the total energy and the number of hits, are trained on the new dataset. Both models suffer from a significant decrease in accuracy and the baseline model does no longer show capability of extracting important features from the images (figure 4.8), i.e. the performance is the same as when only using the now detected number of hits and total light.

Table 4.6: CNN with increased number of layers architecture

Action	Output size	Signals (#)	Parameters (#)
Input:	[2x50x50x60]	300,000	0
Conv3-16:	[16x50x50x60]	2,400,000	16*(3*3*3*2+1) = 880
Conv3-16:	[16x50x50x60]	2,400,000	16*(3*3*3*16+1) = 6,928
PoolAvg2:	[16x25x25x30]	300,000	0
Conv3-32:	[32x25x25x30]	600,000	32*(3*3*3*16+1) = 13,856
Conv3-32:	[32x25x25x30]	600,000	32*(3*3*3*32+1) = 27,680
PoolAvg2:	[32x13x13x15]	81,120	0
Conv3-64:	[64x13x13x15]	162,240	64*(3*3*3*32+1) = 55,360
Conv3-64:	[64x13x13x15]	162,240	64*(3*3*3*64+1) = 110,656
PoolAvg2:	[64x7x7x8]	25,088	0
Conv3-128:	[128x7x7x8]	50,176	128*(3*3*3*64+1) = 221,312
Conv3-128:	[128x7x7x8]	50,176	128*(3*3*3*128+1) = 442,496
PoolAvg2:	[128x4x4x4]	8,192	0
Conv3-256:	[256x4x4x4]	16,384	256*(3*3*3*128+1) = 884,992
Conv3-256:	[256x4x4x4]	16,384	256*(3*3*3*256+1) = 1,769,728
PoolAvg2:	[256x2x2x2]	2,048	0
FC:	[2048x1x1x1]	2,048	(256*2*2*2+1)*2,048 = 4,196,352
FC:	[2048x1x1x1]	2,048	(2,048+1)*2,048 = 4,196,352
FC:	[2048x1x1x1]	2,048	(2,048+1)*2,048 = 4,196,352
FC:	[8]	8	(2,048+1)*8 = 16,392
Total:		~7.2M	~16M

Total memory of signals: ~7.2M → 7.2M*4 B = 28.8 MB (per image)
Total memory of parameters: ~16M → 16M*4 B = 64 MB

4.8 Predicting neutron momentum

Predicting the momentum of a single neutron entering the detector was attempted by altering the number of output nodes of the baseline CNN from 8 to 3. The three output nodes will be the the prediction of the three spatial components x_i of the momentum. Since the cross-entropy loss function acts on probability distributions, it does not make sense in the context of regression and therefore a new loss-function is needed. The mean squared error

$$g_{mse} = \sum_{i=1}^{3}(y_i - x_i)^2, \tag{4.1}$$

is chosen as a new loss function, where y_i are the correct momentum components and the factor of $\frac{1}{3}$ has been omitted. The training is achieved with the ADAM optimization and the datasets only contain events where a single neutron has entered the detector.
The root mean squared error at the last training step is of the order of ~ 17 MeV/c (4.9), for each spatial component. For a neutron with energy within 550 and 650

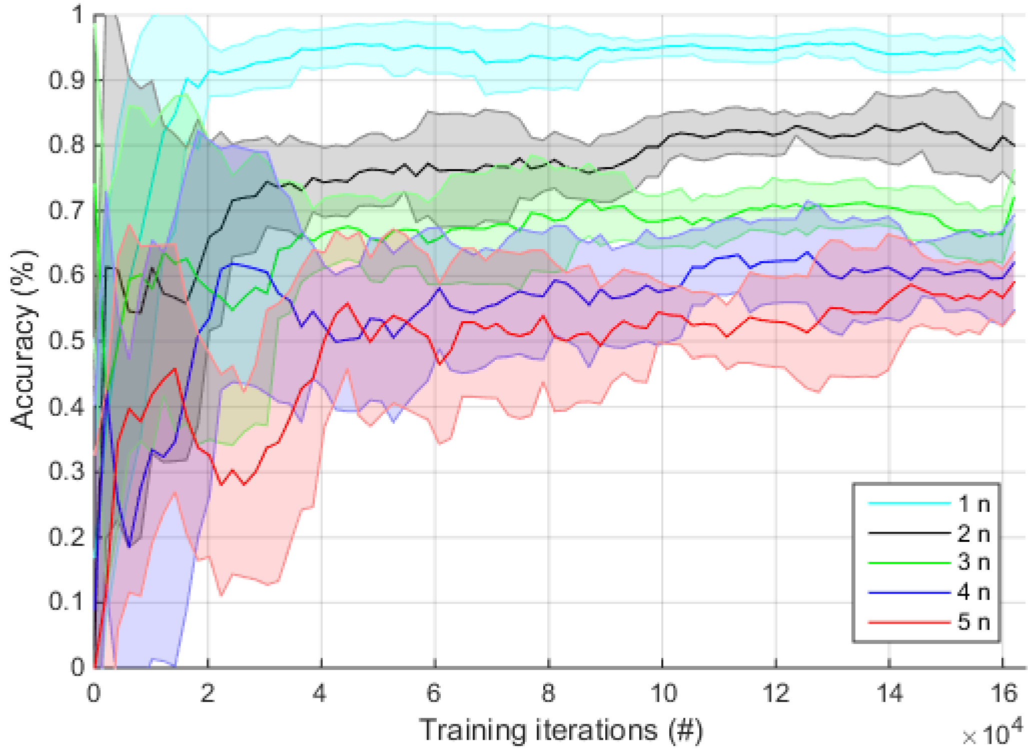


Figure 4.7: The accuracy of the extended baseline model accounting for total energy and number of hits, tested on the validation set containing 1 600 images for each neutron multiplicity. The curves are smoothed with a moving average filter along with a shaded area representing the standard deviation of nearby data points.

MeV, the z component can have a momentum between 1156 and 1282 MeV/c. The momentum difference of $\sim$ 17 MeV/c thus corresponds to an error of 1.3 to 1.5 %. This might seem promising, however, keep in mind that during training the model only sees momentum values in the z direction between 1156 and 1282 MeV/c, corresponding to a maximum error of 11 %. Furthermore, the error declines steady during training with no clear indication of convergence, suggesting that a lower error can be achieved by further training.

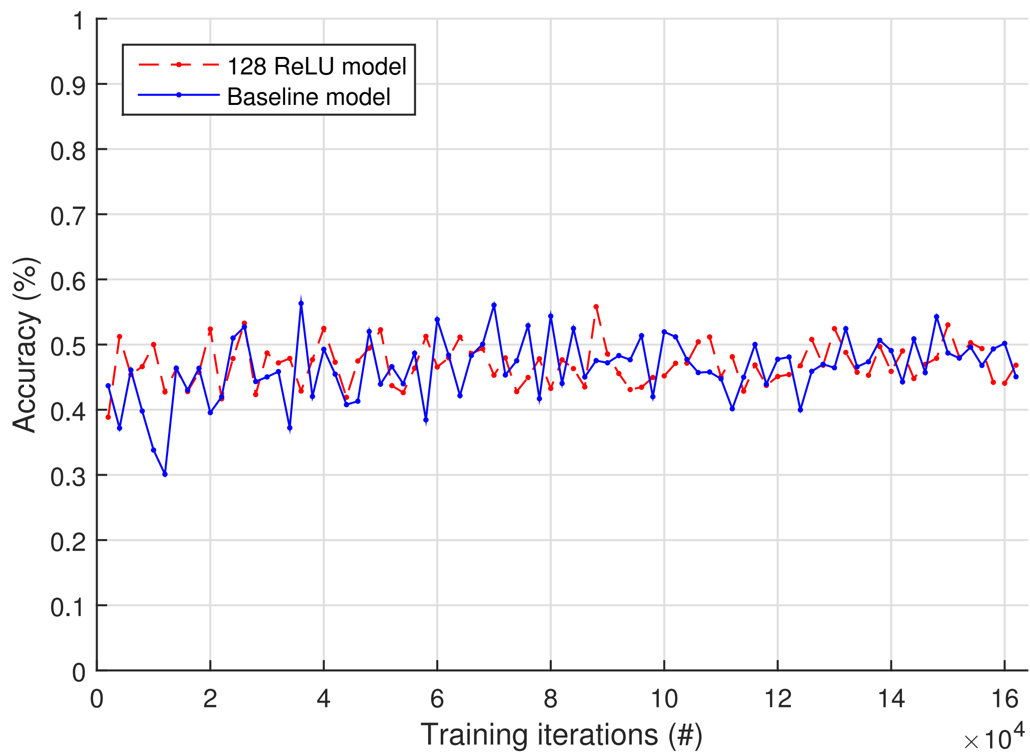


Figure 4.8: The accuracy of both the baseline model and the two-layer fully-connected model only accounting for the total energy and the number of hits, tested on the validation dataset containing 1 600 images for each neutron multiplicity. The datasets are altered in order to account for the light yield loss due to Birk's law.

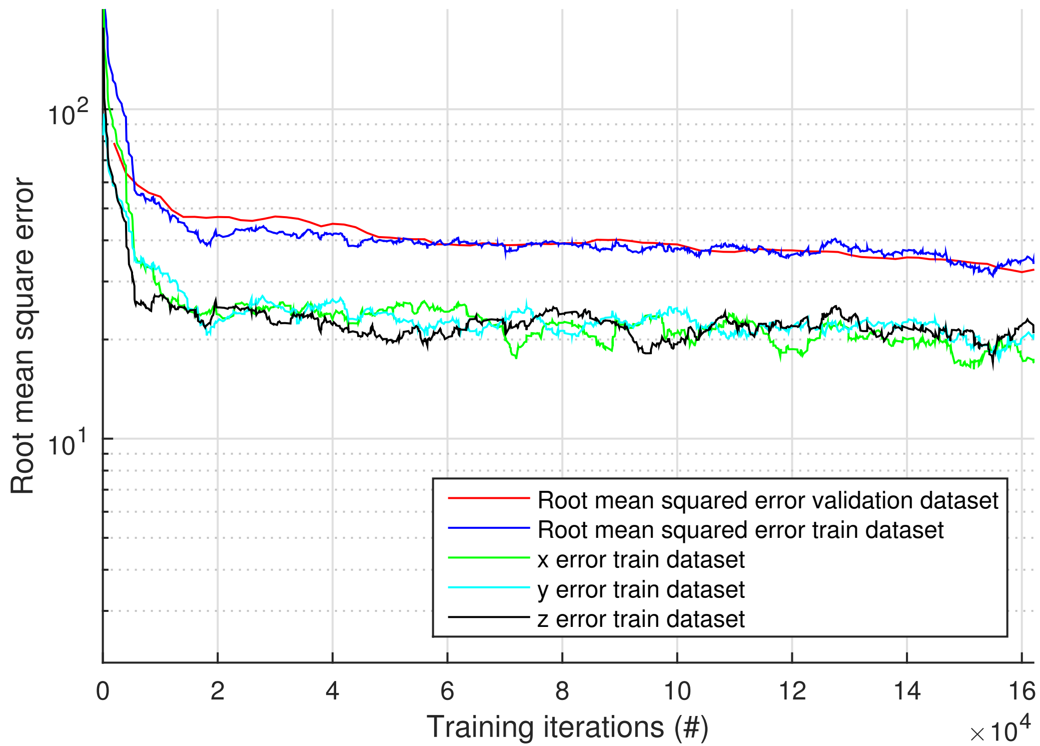


Figure 4.9: The root mean squared error of the momentum prediction of a single neutron entering the detector from the extended baseline model, tested on both the validation dataset, containing 1 600 images, and the training dataset. The plot shows the total root mean squared error along with the error of the individual x, y and z axes. The factor of $\frac{1}{3}$ has been omitted in the root mean squared error.

Chapter 5

Discussion and conclusion

5.1 Training on Graphic Processing Units

The documentation of TensorFlow recommends the use of GPUs and provides some benchmarks for different models trained on different GPUs. However, the lack lack of CPU benchmarks makes it harder to appreciate the actual gain of switching to a GPU. During this work we have learned, the hard way, the significant increase in speed obtained when using GPUs for training CNNs. The speed increase factor of 185, when using the expensive Nvidia Tesla K80, clearly advocates the use of GPUs, although, this is highly dependent on the architecture and size of the model. With a price tag approximately half of the cost of the Intel Xeon E5-1650v2 CPU, the Nvidia Geforce GT 750M reaches a speed increase factor of 40. In comparison, distributing the training on multiple machines would require at least 40 CPUs in order to obtain a similar increase in speed. An upside is that this kind of distributed training is possible also with GPUs, further increasing the potential increase in training speed.

5.2 CNN for reconstructing neutron events

The accuracy increase obtained when going from the MLPs used as accuracy reference to the extended baseline CNN model, indicates a great potential in using CNNs for neutron reconstruction. The results, particularly from the momentum prediction, suggest that the CNNs are able to extract important geometrical features from the images, although, the models do have some flaws.

Due to the significantly varying accuracy during training, it is not a trivial matter how the performance of different models should be evaluated. The different models evaluated in this thesis produce similar results and only in some cases are the differences large enough to allow to to rank them performance-wise, with certainty. Using the end mean value, we obtain an indication of the performance of the models. A summary of the different models is presented in table 5.1.

Additionally, since we set the learning algorithm to minimize the total accuracy of all neutron multiplicities, the learning algorithm is free to alter the accuracy of the individual multiplicities in any way as long as the total accuracy increases. As is evident in figure 4.7, the individual accuracies fluctuate significantly, particularly for high neutron-multiplicities. Whether this is due to the small batch size and that

Table 5.1: The end mean value of the accuracy of the different models evaluated.

Model	End mean
Baseline CNN.	(76.7 ± 1.5) %
Baseline CNN. Training dataset with a limit of 50 MeV minimum deposited energy per neutron (see section 4.5.1).	(77.3 ± 1.6) %
Extended baseline. Normalizing over the entire dataset.	(72.6 ± 1.3) %
Extended baseline. Per-image zero-mean normalization.	(77.1 ± 1.4) %
Extended baseline. Per-image [0, 1] normalization.	(76.7 ± 0.7) %
Extended baseline with max pooling.	(76.6 ± 1.1) %
Increased number of layers. (see section 4.5.6).	(78.2 ± 1.3) %

the model locally overfits the data, should be further investigated. The fact that there was no noticeable effect on the accuracy when the number of nodes in the FC-layers of the extended baseline model was significantly reduced, suggests that the extended baseline model has an excessive amount of nodes, making it capable of overfitting. The improved accuracy of the model with an increased number of layers (see section 4.5.6), further argues that a larger number of convolutional layers is more important than having many nodes in the fully-connected layers.
Training the extended baseline model without normalizing the data shows that the ADAM algorithm encounters difficulties when the values of the different input channels are largely separated in magnitude. The model reaches an accuracy on the validation dataset of approximately 65 % after only 2 000 training iterations: quicker than all the other models accounting for the images. Following the discussion about data normalization (section 4.5.3), this indicates that the learning algorithm has a larger relative effect on the larger input channels i.e. the total energy deposited and the total number of hits. The explanation of why the accuracy suddenly drops is a matter outside the scope of this thesis.
Both of the two per-image normalization techniques resulted in larger end mean values of the accuracy compared to the technique normalizing over the entire dataset. A perhaps, hasted conclusion could be that the model is better suited to extract features from images of approximately equal brightness. If this were the case, the performance of the baseline CNN model trained on unnormalized data, should be closer to the performance of the extended baseline model trained on data normalized over the entire dataset. Instead, the discrepancy in performance could originate from a too short training period, not letting the model reach its full potential.
The training period for most of the models lasted roughly 40 h, including the accuracy tests on the validation dataset. Due to time constraints and certain priorities, the training period was not increased. In future research this should be highly prioritized. On the other hand, reducing the dataset (section 4.5.4) had a significant negative effect on performance and resulted in a textbook example of overfitting.

5.3 Accuracy cost due to Birk's law

Accounting for the light yield loss due to Birk's law, resulted in a considerable decrease of accuracy. A decrease was expected, although, not to the extent that the CNNs no longer are able to extract geometrical features that result in an increased accuracy. The short time spent on studying Birk's law in detail was insufficient to make any qualative statements that explain the large decrease in performance.

Chapter 6

Outlook

Although the decrease in performance due to the implementation of Birk's law is a considerable setback, this thesis proves that CNNs are able to extract geometrical features from the neutron images (without Birk's law), motivating future research. Viewing the neutron events as 3-D images was done in an attempt to reduce the size needed of the CNNs. In further investigations the neutron events can instead be viewed as 3-D videos, where the classification and regression could be achieved with a CNN and a Recurrent Neural Network (RNN) together, thus ”absorbing” the time information. The combination of CNNs with RNNs has shown state-of-the-art performance on video recognition tasks [22].

Working with simulated data is beneficial since it provides a practically unlimited dataset: once you run out of data, you run further simulations. This removes the otherwise time consuming task of collecting data, which many other image recognition projects are subject to, allowing future research to solely focus on designing the optimal model for reconstructing neutron events.

Bibliography

[1] GSI. *Technical Report for the Design, Construction and Commissioning of NeuLAND: The High-Resolution Neutron Time-of-Flight Spectrometer for R3B*. http://www.fair-center.de/fileadmin/fair/experiments/NUSTAR/Pdf/TDRs/NeuLAND-TDR-Web.pdf. 2011. (Visited on 2017-06-10).

[2] GSI. *Neutron ToF Spectrometer NeuLAND*. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4300861/. 2016. (Visited on 2017-01-16).

[3] J.G. Keller and E.F. Moore. *Shower Recognition and Particle Identification in LAND*. GSI Annual Report 1991, 1992, p. 39.

[4] Hans Törnqvist and Linus Trulsson. "Probabilistic Neutron Tracker: Making Good Guesses on Invisible Interactions in Subatomic Physics". MA thesis. Chalmers University of Technology, 2009.

[5] DeepL. *AI Assistance for Language*. https://www.deepl.com/. 2017. (Visited on 2017-08-31).

[6] Alex Krizhevsky and Ilya Sutskever and Geoffrey E. Hinton. *ImageNet Classification with Deep Convolutional Networks*. https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf. 2012. (Visited on 2017-08-03).

[7] Adrian Cho. "AI's early proving ground: the hunt for new particles". In: *Science* 357.6346 (2017), p. 20. DOI: http://science.sciencemag.org/content/357/6346/20.full. (Visited on 2017-08-31).

[8] TensorFlow. *TensorFlow*. http://tensorflow.org. 2017. (Visited on 2017-08-28).

[9] Tom M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[10] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[11] Håkan T. Johansson. *ggland - command-line simulation wrapper*. http://fy.chalmers.se/~f96hajo/ggland/. 2017. (Visited on 2017-08-08).

[12] CERN. *GEANT4*. http://geant4.cern.ch/. 2017. (Visited on 2017-08-20).

[13] Nvidia. *CUDA GPUs*. https://developer.nvidia.com/cuda-gpus. 2017. (Visited on 2017-09-10).

[14] nvidia.com. *NVIDIA TESLA K80.* http://www.nvidia.com/object/tesla-k80.html. 2017. (Visited on 2017-08-09).

[15] Diederik P. Kingma and Jimmy Lei Ba. *ADAM: A Method for Stochastic Optimization.* https://arxiv.org/pdf/1412.6980.pdf. 2015. (Visited on 2017-08-29).

[16] TensorFlow. *Distributed TensorFlow.* https://www.tensorflow.org/deploy/distributed. 2017. (Visited on 2017-08-28).

[17] intel.com. *Intel Xeon Processor E5-1650 v2.* https://ark.intel.com/sv/products/75780/Intel-Xeon-Processor-E5-1650-v2-12M-Cache-3_50-GHz. 2017. (Visited on 2017-08-10).

[18] nvidia.com. *NVIDIA GeForce GT 750M.* https://www.geforce.com/hardware/notebook-gpus/geforce-gt-750m. 2017. (Visited on 2017-09-14).

[19] TensorFlow.org. *Convolutional Neural Networks.* https://www.tensorflow.org/tutorials/deep_cnn. 2017. (Visited on 2017-08-08).

[20] HPC2N. *Kebnekaise.* https://www.hpc2n.umu.se/resources/hardware/kebnekaise. 2017. (Visited on 2017-09-16).

[21] J.B. Birks. “Scintillations from Organic Crystals: Specific Fluorescence and Relative Response to Different Radiations”. In: *Proc. Phys. Soc.* A64.10 (1951), pp. 874–877.

[22] Li Yao and Atousa Torabi and Kyunghyun Cho and Nicolas Ballas and Christopher Pal and Hugo Larochelle and Aaron Courville. *Describing Videos by Exploiting Temporal Structure.* http://www.cv-foundation.org/openaccess/content_iccv_2015/papers/Yao_Describing_Videos_by_ICCV_2015_paper.pdf. 2015. (Visited on 2017-09-12).

Glossary

3-D - Three-dimensional
API - Application Programming Interface
CNN - Convolutional Neural Network
ConvM-N - 3d convolution of patch size [MxMxM] to same size image with N output features.
CPU - Central Processing Unit
FC - Fully-connected layer.
GPU - Graphics Processing Unit
MLP - Multilayer Perceptron
PM-tube - Photomultiplier tube.
PoolAvgN - 3d average pooling over patch size [NxNxN].
PoolMaxN - 3d max pooling over patch size [NxNxN].
ReLU - Rectified Linear Unit.