Justus-Liebig University Giessen

& Technische Hochschule Mittelhessen

- University of Applied Sciences

Physics and Technology for Space Applications

II. Physics Institute

FB07

Reducing Noisy Clusters in the NA61/SHINE Project with Help of Deep Learning

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE (B.Sc)

JANIK PAUL PAWLOWSKI

Contact: Janik.P.Pawlowski@physik.uni-giessen.de

Student Number: 3116893

Semester: 7

Date of Submission: 09.03.2021

Advisor: PD Dr. Olena Linnyk

2nd reader: PD Dr. Jens Sören Lange

List of Abbreviations

CPU	central processing unit
FN	false negative
FNR	false negative rate
FP	false positive
FPR	false positive rate
GPU	graphics processing unit
MTPCL	main time projection chamber left
SGD	stochastic gradient descent
TN	true negative
TNR	true negative rate
TP	true positive
TPC	time projection chamber
TPR	true positive rate
VTPC2	vertex time projection chamber two
1Px20T	one pad at twenty time steps
11Px1T	eleven pads at one time step
11Px20T	eleven pads at twenty time steps

Table of Contents

1 Introduction

Technologies enabled by machine learning and artificial intelligence dominate modern society. From web searches and smart assistants like Cortana (Microsoft) or Siri (Apple), to spam-filters in our e-mail programmes. Artificial intelligence is increasingly present in our daily live.

We live in an era where huge amounts of data are being generated in all sectors of science and industry. We call it "Big Data". With Big data, we face the challenge of its analysis and interpretation. Therefore, the need for novel machine learning and artificial intelligence methods has drastically increased in recent years. Likewise, the range of areas in which machine learning has been successfully applied has increased significantly: including image recognition (Krizhevsky et al., 2017; Wu & Chen, 2015), speech recognition (Abdel-Hamid et al., 2014; Nassif et al., 2019), natural language processing (Ganapathiraju et al., 2004; Padmanabhan & Johnson Premkumar, 2015; Talafha & Rekabdar, 2019), computational biology (Angermueller et al., 2016) and particle physics (Baldi et al., 2014). Recent breakthroughs inspire us to apply machine learning techniques in more and more areas.

Modern tracking devices for particle collision experiments, such as the four large-volume Time Projection Chambers (TPC) in the NA61/SHINE project at the European Organization for Nuclear Research (CERN), collect huge amounts of data produced by particle collisions. The data contains not only valuable information but also a lot of noise.

The question arises: Can we reduce the noise with the help of Deep Learning?

In this thesis work, the above mentioned question is answered and multiple machine- and deep learning algorithms are not only proposed, but the underlying principles explained and visualized. This thesis tries to answer why those algorithms work and what they might learn. In the attachments a detailed explanation will be provided, how the experimental data was obtained and pre-processed.

1.1 Solving Classification with Machine Learning Techniques

One form of supervised machine learning is classification. Classification is the process of separating the data of a certain data set in different classes. The process starts with predicting the respective class of given data points. An arbitrarily complex function therefore maps the input variables x_i of the given data to discrete output variables, which represent the different classes. Classes are usually referenced to as labels or targets.

One of the best-known classification tasks are image recognition (face recognition) and voice recognition (e.g., Alexa and Siri from Amazon and Google, respectively).

2 Theory

2.1 Single Layer Perceptron

The basic single layer Perceptron only consists of the input layer and one layer of $i \in N$ so-called neurons. Every neuron n_i represents a linear function $f_i(\vec{X})$.

The connections between two layers are equipped with weights and biases, used to feed forward information. While they are initialized randomly, the goal is to learn the correct weights, so that the network is able to make correct predictions for unknown data.

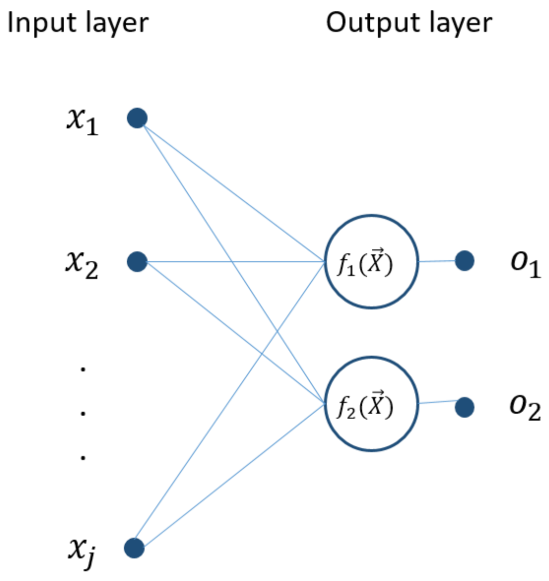


Figure 1: Concept of a single Layer Perceptron with two neurons. All input values are connected to each of the two output neurons via a weight and a bias

The output o_{ni} of a single neuron is defined by the sum of all weighted input values plus a constant bias b:

$$o_{n_i} = \sum f_{ij}(x_j) + b_i \tag{2.1}$$

With

$$f_{ij}(x_j) = w_{ij}x_j + b_i \tag{2.2}$$

Resulting in the layers output vector:

$$\vec{O} = \begin{pmatrix} o_{n1} \\ o_{n2} \\ \cdots \\ o_{ni} \end{pmatrix} \tag{2.3}$$

Note that all n nodes receive the same j input values; the individual characterization is given via the corresponding weights and biases.

The output appears in form of a two-dimensional vector:

$$\vec{O} = \begin{pmatrix} p_s \\ p_n \end{pmatrix} \tag{2.4}$$

In this thesis, 220 input values lead to two output values. These values are interpreted as probabilities for the corresponding classes signal and noise, which will be referred to as positives and negatives or good and bad hits.

By definition, the first entry corresponds to the probability for signal and the second to that for noise where:

$$p_s + p_n = 1 \tag{2.5}$$

For example an output of :

$$Output = \binom{o_{n_1}}{o_{n_2}} = \binom{\sum w_{1j}x_j + b_1}{\sum w_{2j}x_j + b_2} = \binom{0.7}{0.3} \tag{2.6}$$

Would mean that the network predicts the sample to be signal because $p_s > 0.5$ where 0.5 is the default decision threshold in binary classification (classification with two classes).

However, it is possible to manipulate the threshold in order to influence the ratio of false positives to false negatives. This yields the option to force the network to be more or less sure about predictions for a certain class.

2.2 Activation

2.2.1 Softmax

In classification algorithms, the output vector is usually normed by a so-called activation function. The activation function can be written as a function of the sum of the weighted input values:

$$activation = f(\sum w_{ij}x_j + b_i) \tag{2.7}$$

This allows an easier interpretation since it normalizes the input values in a certain way. Furthermore, logistic activation functions like Softmax can enable a probabilistic interpretation.

Softmax takes the exponent of each calculated value of the last layer (in case of Single Layer Perceptron there is only one layer) and then divides by the sum of these exponents. That way the resulting probability values sum up to one.

The mathematical definition writes as follows (Sharma, 2017):

$$O_{n_i} = Softmax(o_{n_i}) = \frac{e^{o_{n_i}}}{\sum e^{o_{n_j}}} \tag{2.8}$$

To put it in context, Figure 2 exemplarily shows the steps between input and output of one neuron. The output is calculated by the sum of all input values $x_1 - x_n$, weighted by their respective weight $w_1 - w_n$ plus a bias b_0 , usually normed by an activation function f.

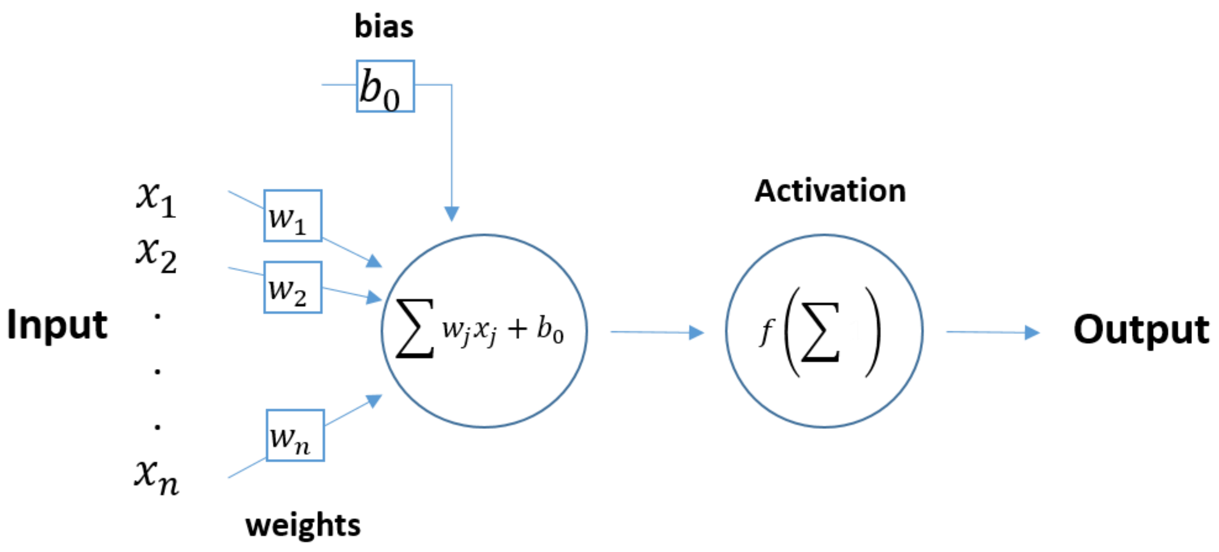


Figure 2: Sketch of a one-neuron Perceptron. The input vector is weighted and a bias is added. Output is their sum, normed by an activation function f

2.2.2 Rectified Linear Unit

A rectified linear unit (ReLU) function is commonly used in deep learning as an activation function. For every input below zero, it returns zero and for every positive value, it returns the same positive value (Figure 3). ReLU helps to speed up the learning convergence (Hara et al., 2015).

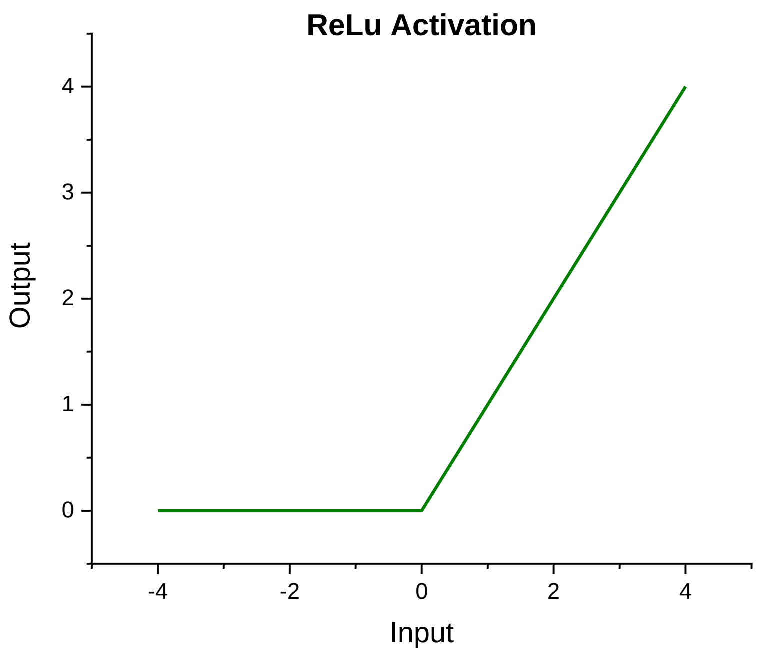


Figure 3: Rectified linear unit. Returns zero for negative input values and for positive inputs returns the same positive value

2.3 Loss Function

For the first step, the classification error has to be put in numbers. Loss functions like root mean squared error, or, in this thesis binary cross entropy, calculate the error margin (loss) for each node of the final layer. The sum of those losses is called cost. The loss is calculated via a specific equation, called loss function. In general, this is a function of the margin between the calculated output values and the corresponding true values. The most used loss-function in deep neural networks for classification tasks is the binary cross-entropy (Figure 4).

The loss is calculated for each class by the following equation (Ismail Fawaz et al., 2019):

$$binary\ cross - entropy(X)_i = -\sum Y_i \cdot \log(O_{n_i}) \tag{2.9}$$

Where X is the training example, O_{n_i} is the i-th scalar value in the model output and Y_i is the corresponding target value.

The total loss for one example calculates by the following average:

$$Loss(X) = -\frac{1}{n_{out}} \cdot \sum Y_i \cdot \log(O_{n_i}) + 1(-Y_i) \cdot \log(1 - O_{n_i}) \tag{2.10}$$

Where n_{out} is the number of values in the model output (the number of classes), X is the training example, O_{n_i} is the i-th scalar value in the model output and Y_i is the corresponding target value.

For understanding the learning process, the formula does not need to be discussed in detail but it is important to note that this loss function is fully derivable, so gradients can be calculated. It can therefore be used as a metric that represents the level of error for each example in a data set. A smaller total loss corresponds to a more precise prediction model. Thus, the weights should be learned in such a way that the cost function is minimized.

2.4 Optimizer

At first, the input is passed to the network, second, the network computes an output: its prediction. Third, the total loss is calculated using a loss function. Fourth: The network pushes the loss function through the network in a backward manor. Fifth, to minimize the loss, the weights are updated accordingly (Figure 4). The algorithm that calculates how the weights have to be adjusted in order to minimize the cost function is called optimizer.

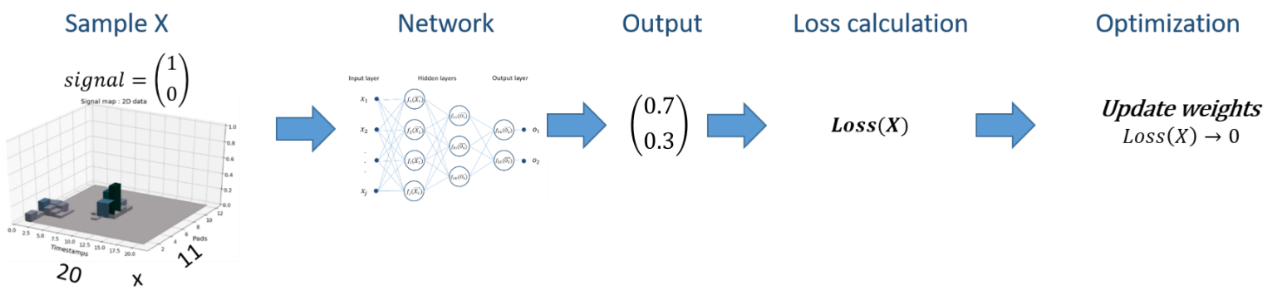


Figure 4: Training process of an artificial neural network. After the prediction process is complete, the weights are updated in order to minimize the cost function

2.4.1 Gradient Descent

Gradient descent is a method to optimize the modelling function by minimizing an objective function $J(w)$. $J(w)$ is parameterized by the model's parameters w and optimized by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_w J(w)$. The learning rate lr determines the step size that is taken each update to reach a (local) minimum. Figuratively speaking, the optimizer follows the direction of the steepest descent of the valley potential calculated by the objective function downhill. In the best case, until a valley is reached, but depending on the step size, possibly also beyond the valley, which can lead to oscillation around the valley (Ruder, 2016).

Three main variants of gradient descent have to be distinguished. They differ in the data used to compute the gradient of the objective function. It is a trade-off between accuracy of the parameter update and the computing time, depending on the amount of data used (Ruder, 2016).

1) Batch gradient descent

Vanilla gradient descent, also called batch gradient descent, computes the gradient of the cost function with respect to the parameters w for the entire data set.

$$w_{updated} = w - lr \cdot \nabla_w J(w) \tag{2.11}$$

Since gradients for the entire data set must be calculated for each update, batch gradient descent can be computationally slow. The training consists of a pre-defined number of epochs. For each epoch, the gradient vector of the loss function for the whole data set with respect to the parameter vector is computed. The gradient is usually computed via backpropagation (Rumelhart et al., 1986). Then, the parameters are updated in the gradients direction with a step size determined by the learning rate.

“Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces” (Ruder, 2016).

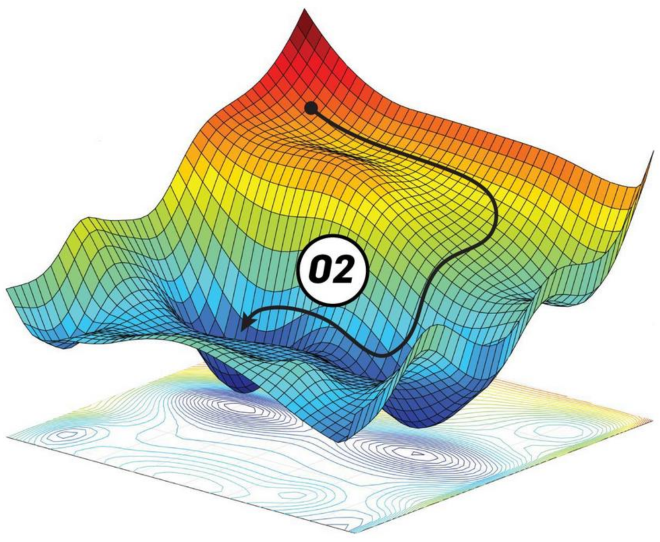


Figure 5: exemplary error surface and possible batch gradient descent optimizers learning path visualized. Credit: Ahmed Fawzy Gad

2) Stochastic gradient descent

Stochastic gradient descent (SGD) in contrast updates the parameters for each training example $x^{(i)}$ and label $y^{(i)}$.

$$w_{updated} = w - lr \cdot \nabla_w J(w; x^{(i)}; y^{(i)}) \tag{2.12}$$

This way it takes less time to compute the parameters update.

Batch gradient descent always converges to a minimum, whereas SGD often fluctuates around minima. This allows SGD to jump to new and potentially better local minima, but at the same time makes convergence to an exact minimum more difficult, since SGD can overshoot again and again.

However, it has been shown that if the learning rate is slowly being decreased during the training process, SGD shows the same convergence behaviour as vanilla gradient descent (Ruder, 2016).

3) Mini batch gradient descent

The third variant combines the best out of both methods and performs an update for every mini-batch of n samples.

$$w_{updated} = w - lr \cdot \nabla_w J(w; x^{(i:i+n)}; y^{(i:i+n)}) \tag{2.13}$$

This has two main advantages (Ruder, 2016):

a) The variance of the parameter updates can be reduced which results in a better converging behaviour.
b) Computing the gradient is a lot faster than in vanilla gradient descent, as state-of-the-art deep learning libraries contain optimized matrix operations, which make computing the gradient with respect to a mini-batch very efficient.

Mini-batch gradient descent, or an optimizer based on it, is usually the option of choice in deep learning applications.

2.4.2 Gradient Descent Optimizations

2.4.2.1 Momentum

Momentum tackles the challenge of finding the (local) minima in the error surface faster and more reliably. It adds another term to the gradient descent and combines to following method (RUMELHART et al., 1988):

$$w_{updated}(n+1) = w(n+1) - lr \cdot \nabla_w J(w) + \alpha \cdot \Delta \mathrm{w}(\mathrm{n}) \tag{2.14}$$

With $\Delta w(n)$ defined as follows:

$$\Delta w(n) = w_{updated}(n) - w(n) = -lr \cdot \nabla_w (J(w(n))) \tag{2.15}$$

This way the update of the parameters is always influenced by the previous updates, such that the optimizer can gain momentum if it is heading in the same direction of the error surface over multiple periods. Imagine it moving slowly downhill but swinging strongly sideways due to a fault surface that is slightly downhill but very steep to the left and right. The lateral components will cancel each other out, while the forward component keeps adding up (Figure 6).

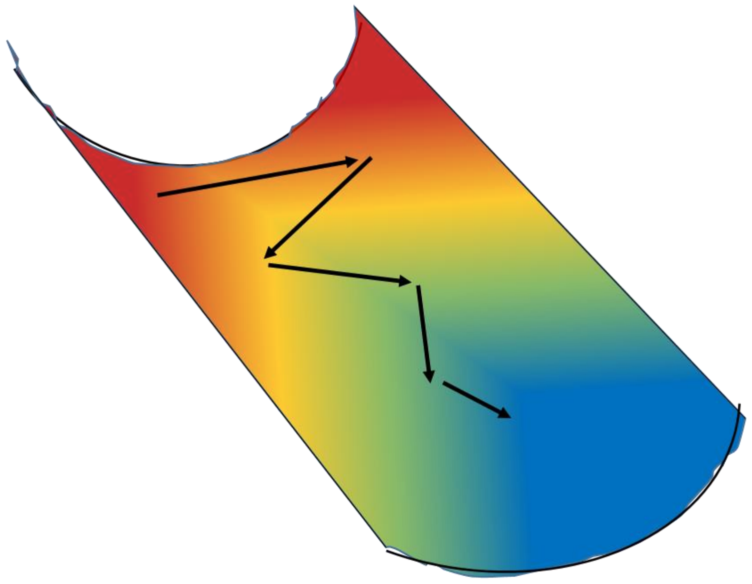


Figure 6: An exemplary error surface where the momentum term helps lowering the oscillation. The two opposite gradients nullify each other but the downhill term is accumulative increased

2.4.2.2 Adaptive Learning Rate Optimization Algorithm

For this thesis project, an adaptive learning rate optimization algorithm called Adam (Kingma & Ba, 2014) was used. Adaptive Moment Estimation (Adam), like most state-of-the-art optimizers, is an optimized version of gradient descent.

The Adam optimizer calculates an adaptive learning rate for each parameter. As for momentum, Adam stores an exponentially decaying average of past gradients m_n and in addition the average of past squared gradients v_n.

$$m_n = \beta_1 m_{n-1} + (1 - \beta_1) g_n \tag{2.16}$$

$$v_n = \beta_2 v_{n-1} + (1 - \beta_2) g_n^2 \tag{2.17}$$

m_n and v_n are estimates of the first moment (the mean) and the second moment (uncentered variance) of the respective gradients.

Diederik P. Kingma and Jimmy Ba experimentally showed, that deep artificial neural networks tend to converge faster when using Adam than with other state-of-the-art methods (Kingma & Ba, 2014).

In 2019 S. Bock and M. Weis have proven that Adam at least converges to a local minimum (Bock & Weis, 2019). Since its introduction, Adam has been used in many cases and proven to be one of the best choices when training a deep convolutional neural network (Ruder, 2016).

2.5 Multi-layer Perceptron

Machine learning transitions into deep learning if the algorithm consists of more than one layer of neurons. The layers, which lie in between the input and the output layer, are called hidden layers. Multi-layer Perceptrons belong to the class of feedforward networks and form the basis for most deep learning models. A feedforward neural network is an artificial neural network in which the information only moves in one direction - from the input nodes through the hidden nodes (if any) to the output nodes. Convolutional neural networks, which will be discussed in more detail later, are also a subclass of feedforward networks.

These models are mostly used for supervised learning, where the network is supposed to match a result that is already known. They are very important for practicing machine learning and build the foundation for many commercial applications nowadays. Their presence, especially convolutional neural networks greatly affected areas such as Image recognition and natural language processing.

Multi-layer Perceptrons consist of multiple layers, where each layers input is the previous layers output.

The input values $x_1 - x_j$ of the first layer can be written as a vector $\vec{X}_1$:

$$\overrightarrow{X_1} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_j \end{pmatrix} \tag{2.18}$$

The output o_{1i} of the first layer is given by a function $f_i(\overrightarrow{X_1})$:

$$f_i(\overrightarrow{X_1}) = \sum w_{ij} \cdot x_j + b_i \tag{2.19}$$

Resulting in:

$$O_1 = \begin{pmatrix} o_{11} \\ o_{12} \\ \cdots \\ o_{1i} \end{pmatrix} = \begin{pmatrix} f_1(\overrightarrow{X_1}) \\ f_2(\overrightarrow{X_1}) \\ \cdots \\ f_i(\overrightarrow{X_1}) \end{pmatrix} \begin{pmatrix} \sum w_{1j} \cdot x_j + b_1 \\ \sum w_{2j} \cdot x_j + b_2 \\ \cdots \\ \sum w_{ij} \cdot x_j + b_i \end{pmatrix} \tag{2.20}$$

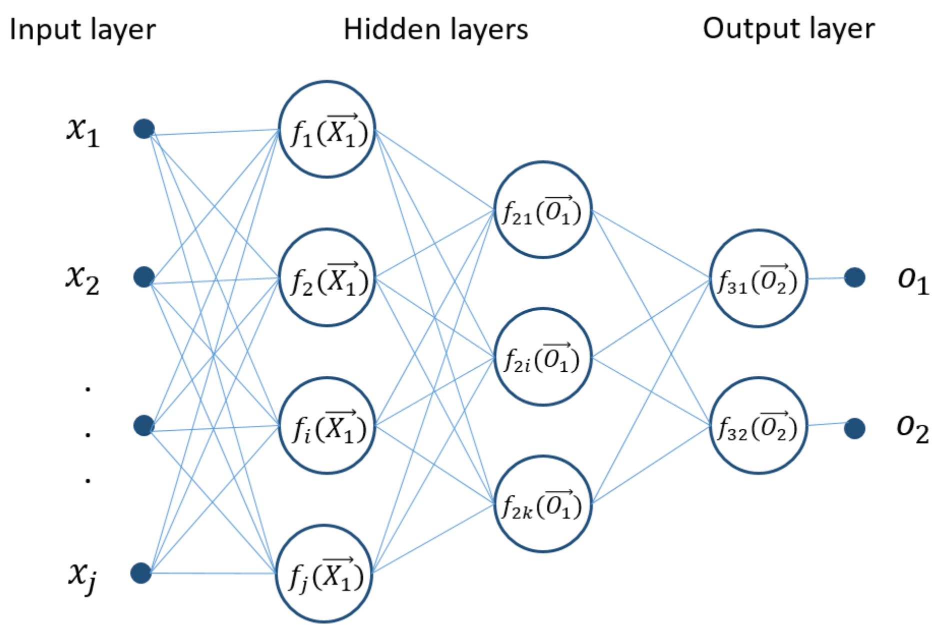
The output of a second layer neuron o_{2k} is then a function of the output from the previous layer $f_k(O_1)$ with:

$$f_k(o_{1i}) = \sum a_{ki} \cdot o_{1i} + b_k \tag{2.21}$$

Where i is the respective number of nodes in layer 2 and k in layer 3, as they equal the number of generated outputs, resulting in:

$$\overrightarrow{O_2} = \begin{pmatrix} f_1(\overrightarrow{O_1}) \\ f_2(\overrightarrow{O_1}) \\ \cdots \\ f_k(\overrightarrow{O_1}) \end{pmatrix} = \begin{pmatrix} \sum w_{1i} \cdot o_{1i} + b_k \\ \sum w_{2i} \cdot o_{1i} + b_k \\ \cdots \\ \sum w_{ki} \cdot o_{1i} + b_k \end{pmatrix} \tag{2.22}$$

The third layers output is generated in the same fashion as in the previous layer but with the previous layers output as input. This holds true for all following layers (Figure 7).



Input layer
Hidden layers
Output layer
x_1
x_2
x_j
$f_1(\overrightarrow{X_1})$
$f_2(\overrightarrow{X_1})$
$f_i(\overrightarrow{X_1})$
$f_j(\overrightarrow{X_1})$
$f_{21}(\overrightarrow{O_1})$
$f_{2i}(\overrightarrow{O_1})$
$f_{2k}(\overrightarrow{O_1})$
$f_{31}(\overrightarrow{O_2})$
$f_{32}(\overrightarrow{O_2})$
o_1
o_2

Figure 7: Multi-layer Perceptron with two hidden layers

2.6 Convolutional Neural Networks

Convolutional neural networks work via kernels/filters. Instead of linking every input value on its own to a node of the following layer, filters allow parameter sharing, which drastically reduces the number of parameters.

Imagine a $4x4$ (two-dimensional) input. In a classic MLP, the 16 weighted and biased input values add up as one neurons output (neuron of the first layer). The layers output is always of the form of an $nx1$ vector, where n is the number of neurons in the respective layer.

For convolutional layers the filter size, stride size and number of filters determine the output dimension. For each filter, one output map the so called feature map is generated instead of one scalar value. Its size depends on the filter size, pathing and padding. In this example, an $n \in N$ $2x2$ filter is used, pathing with a stride of one and no padding. Starting in the left upper corner, a stride of one indicates that after each calculation the filter is moved by one column. It follows that the output map is smaller than the input map. To hinder this from happening, padding can be used. This method adds a temporary line of zeros to the input map such that the output is of the same size as the input. However, for simplicity reasons, there is no padding in this example.

The output is calculated as illustrated in Figure 8. Before training, an mxn-filter is initialized with m times n arbitrary scalar values. The goal during training is to adjust these values in such a way that the filter can extract a valuable feature. Imagine an exemplary filter like $\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$. Such a filter might find slanted edges if the input were to the individual grey tone values of the image pixels. Each entry in the output array then represents to which degree there is such a slanted edge at the respective location in the input array. In other words, for every filter, the output is a map that represents to which degree the filter specific feature is present at the corresponding location (Figure 9), hence the output is named feature map.

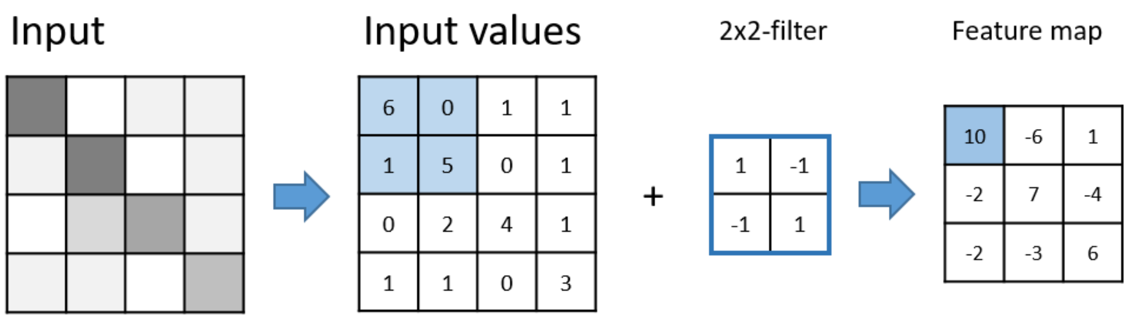


Figure 8: Convolutional layer with one filter of size (2,2). Each step the filter is applied and the four weighted values added together.

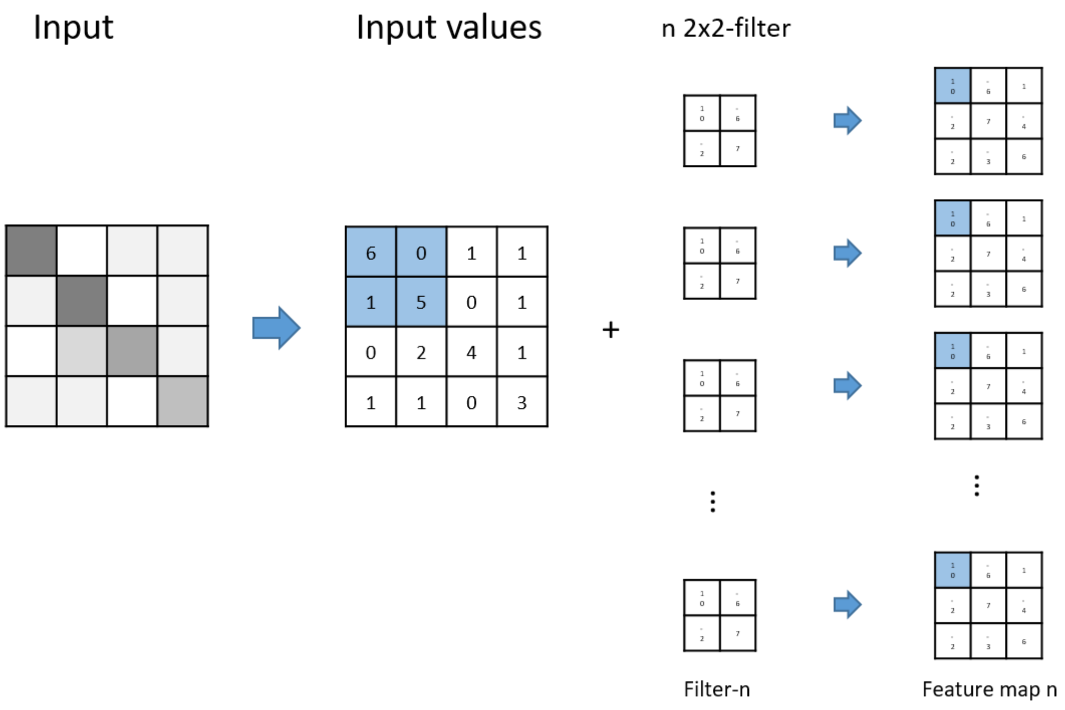


Figure 9: Schematic of a convolutional layer in an artificial neural network. Multiple 2x2-filter generate a feature map each

2.7 ResNet

In 2015, Kaiming et. Al. introduced "Deep Residual Learning for Image Recognition". A Network based on their approach is called ResNet. Deep residual learning tackles a common problem called downgrading and helps deeper artificial neural networks to converge better (He et al., 2015). Convolutional layers often build the fundament of ResNets.

The architecture is based on convolutional blocks, each consisting of a few convolutional layers, followed by a dense layer as output layer. But in between two convolutional blocks, shortcuts are added such that the output of a block consists of the output of the convolutional layers as well as their input added on top (Figure 10). With these shortcuts, the network is able to cut out deeper layers if they are not needed to approximate the solution. If a classifier is imagined whose best solution is a straight line (linear function), it would be easier to find the correct solution for a network with one linear layer than for a network consisting of several stacked linear layers. The addition of shortcuts helps deeper networks with this, since the networks can learn to use only the shortcut by setting the weights of a certain layer to zero, thus turning off the layer.

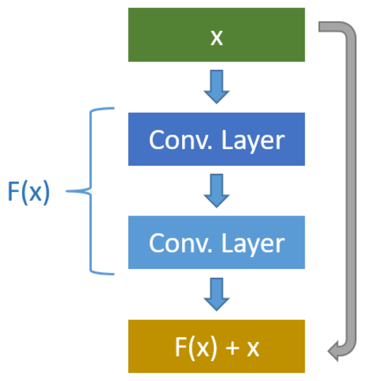


Figure 10: Principle of a ResNet block. One copy of the input is fed into convolutional layers, another copy is passed unchanged. At the end of the block both copies are added together

Due to the modification that results from the addition of these shortcuts, the function to be approximated is now a residual function $F_{new}(x) = F(x) - x$. Hence the name "Residual neural network", since it works via approximating residual functions.

2.8 Layers

Many different layer types for artificial neural networks have been invented and are freely available for implementation. The layer functions used in this thesis are from Tensorflow (https://www.tensorflow.org/), an open source library by the Google Brain Team and Python implementation for all sorts of deep learning applications.

A brief explanation of the layers important for this work will be given in the following.

2.8.1 Dense Layer

Dense layers are standard fully connected linear layers. Each neuron of the previous layer is connected to each neuron of the following neuron via a weight and a bias.

2.8.2 Conv2D Layer

Conv2D layers are convolutional layers (chapter 2.6) with two-dimensional filters (e.g., $2x2$-filters).

2.8.3 Dropout Layer

Dropout layers randomly cut out a certain ratio of connections between two layers during training process. E.g., a dropout layer with a rate of 0.5 after a dense layer with 100 neurons would leave out 50 random neurons each prediction cycle. This can help larger and/or deeper artificial neural networks to partition their memory such that certain groups of neurons learn specific things.

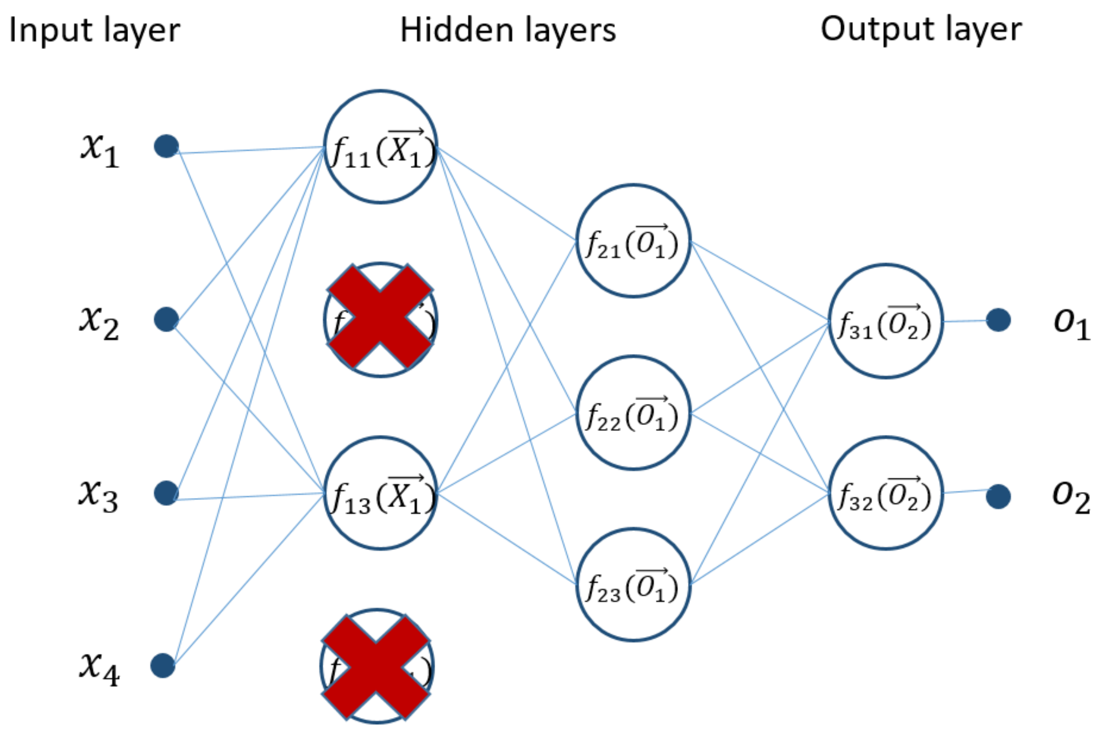

Input layer
Hidden layers
Output layer
x_1
x_2
x_3
x_4
$f_{11}(\overrightarrow{X_1})$
$f_{13}(\overrightarrow{X_1})$
$f_{21}(\overrightarrow{O_1})$
$f_{22}(\overrightarrow{O_1})$
$f_{23}(\overrightarrow{O_1})$
$f_{31}(\overrightarrow{O_2})$
$f_{32}(\overrightarrow{O_2})$
o_1
o_2

Figure 11: Multi-layer Perceptron with a dropout layer after the first hidden layer. Dropout rate = 0.5 → two neurons are randomly left out before each samples prediction

2.8.4 Global Average Pooling Layer

Global average pooling layers calculate and output the scalar mean for each two-dimensional input map.

Example:

$$\begin{pmatrix} a1 & b1 \\ c1 & d1 \end{pmatrix} \to \frac{a1 + b1 + c1 + d1}{4}$$

Global average pooling layers are sometimes used at the end of the convolutional part of complex artificial neural networks to simplify the feature maps before they are fed into a dense network.

2.8.5 Flatten Layer

A Flatten-2D layer converts a two-dimensional array to a one-dimensional array consisting of the two original input dimensions.

Example:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \to (a\ b\ c\ d)$$

2.8.6 Add Layer

An add layer adds its different inputs elementwise. Therefore, all inputs must be of same dimension.

Example:

$$\begin{pmatrix} a1 & b1 \\ c1 & d1 \end{pmatrix} + \begin{pmatrix} a2 & b2 \\ c2 & d2 \end{pmatrix} \to \begin{pmatrix} a1 + a2 & b1 + b2 \\ c1 + c2 & d1 + d2 \end{pmatrix}$$

2.8.7 Concatenate Layer

A concatenate layer appends all inputs on an axis that has to be defined. Therefore, the output size usually differs from the input size.

Example:

$$\begin{pmatrix} a1 & b1 \\ c1 & d1 \end{pmatrix} + \begin{pmatrix} a2 & b2 \\ c2 & d2 \end{pmatrix} \to \begin{pmatrix} a1 & b1 \\ c1 & d1 \\ a2 & b2 \\ c2 & d2 \end{pmatrix}$$

2.9 Evaluation Metrics

2.9.1 Confusion Matrix

In dependency of the prediction result, each sample can be assigned to one of the following four groups:

1) True positive (TP): a positive example, which is correctly predicted to be positive.
2) True negative (TN): a negative example, which is correctly predicted to be negative.
3) False positive (FP): a negative example, which is wrongly predicted to be positive.
4) False negative (FN): a positive example, which is wrongly predicted to be negative.

It can often be useful to look at the corresponding rates instead of absolute numbers:

$$\text{True positive rate (TPR)} = \frac{\text{Number of true positives}}{\text{Total number of positives}} = \frac{TP}{P} = \frac{TP}{TP+FN} \tag{2.23}$$

$$\text{True negative rate (TNR)} = \frac{\text{Number of true negatives}}{\text{Total number of negatives}} = \frac{TN}{N} = \frac{TN}{TN+FP} \tag{2.24}$$

$$\text{False positive rate (FPR)} = \frac{\text{Number of false positives}}{\text{Total number of negatives}} = \frac{FP}{N} = \frac{FP}{TN+FP} \tag{2.25}$$

$$\text{False negative rate (FNR)} = \frac{\text{Number of false negatives}}{\text{Total number of positives}} = \frac{FN}{P} = \frac{FN}{TP+FN} \tag{2.26}$$

The confusion matrix combines these four groups and consists of either TPR, TNR, FPR and FNR or the respective absolute numbers (TP, TN, FP and FN) (Figure 12).

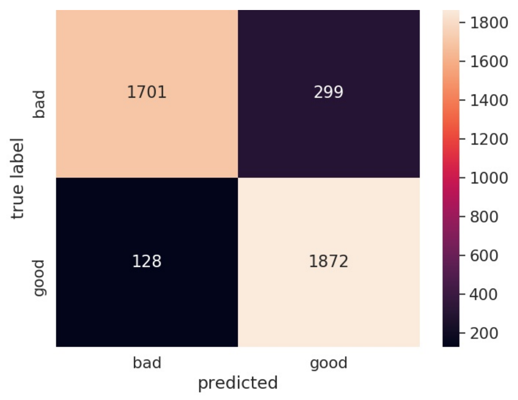


Figure 12: exemplary confusion matrix with the absolute number of TP (upper left corner), TN (lower right corner), FP (upper right corner) and FN (lower left corner)

2.9.2 Sensitivity

The TPR is often referred to as sensitivity as it measures how sensitive the network is in regard of positive examples.

$$Sensitivity = TPR \tag{2.27}$$

2.9.3 Specificity

Specificity on the other hand measures how good the network is in detecting negative examples. It can be written as FPR subtracted from one:

$$Specificity = TNR = 1 - FPR \tag{2.28}$$

This is true because TNR plus FPR always equal one:

$$TNR + FPR = 1 \tag{2.29}$$

2.9.4 ROC Curve

A common way to measure the performance of a neural network is the receiver operating characteristic (ROC) curve, where the $Sensitivity$ (TPR) to one minus $Specifciy$ (FPR) is plotted for various threshold values between zero and one. A perfect performance would be a TPR of one, versus a constant FPR of zero resulting in a curve that goes straight up from zero to one at the y-axis and then remains there for all thresholds. In reality this is never achieved. However, the presence in the upper left corner is directly linked to the level of performance. A high TPR/FPR means that most of the positive examples are detected as positives, while none or only a few negative examples are wrongly classified as positives. Figure 13 illustrates a network with high (green) and one without discriminative power (blue).

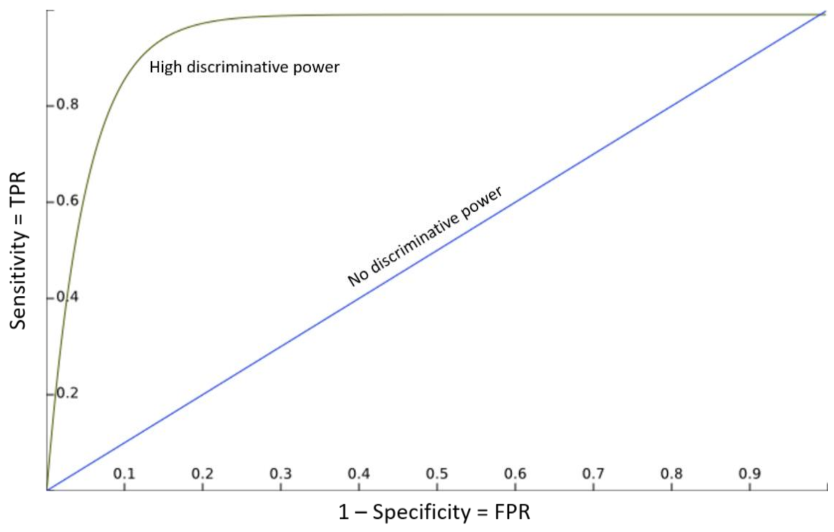


Figure 13: ROC curves for two different networks. One with high (green) and one with no (blue) discriminative power

The ROC curve of an exemplarily trained Perceptron is shown in Figure 14 (blue curve). The curve itself can give a hint whether the network is performing well or not. However, its full potential comes to the forefront, if used for comparison of different networks. In this thesis, the ROC curve will be used to compare performances of different networks on various data sets.

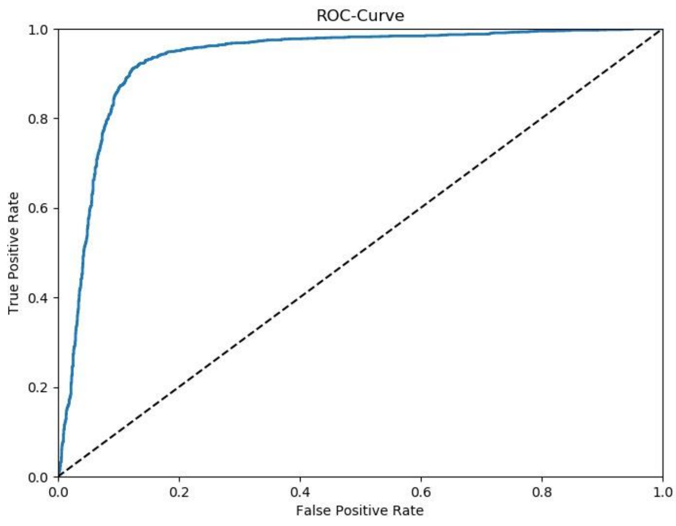


Figure 14: ROC curve - Perceptron trained and validated on MTPCL data

3 Analysis

3.1 Data

For this project, two data sets were available, consisting of data from a particle accelerator at CERN. More precise: data from one of the small Time Projection Chambers (TPC's) at the end of the experiments setup (MTPCL) and from one of the front TPC's (VTPC2) (Figure 15). The chambers pads measure and log the electric charge they receive. Electric charge is generated when particles produced by atomic collisions in the particle accelerator pass through the chamber.

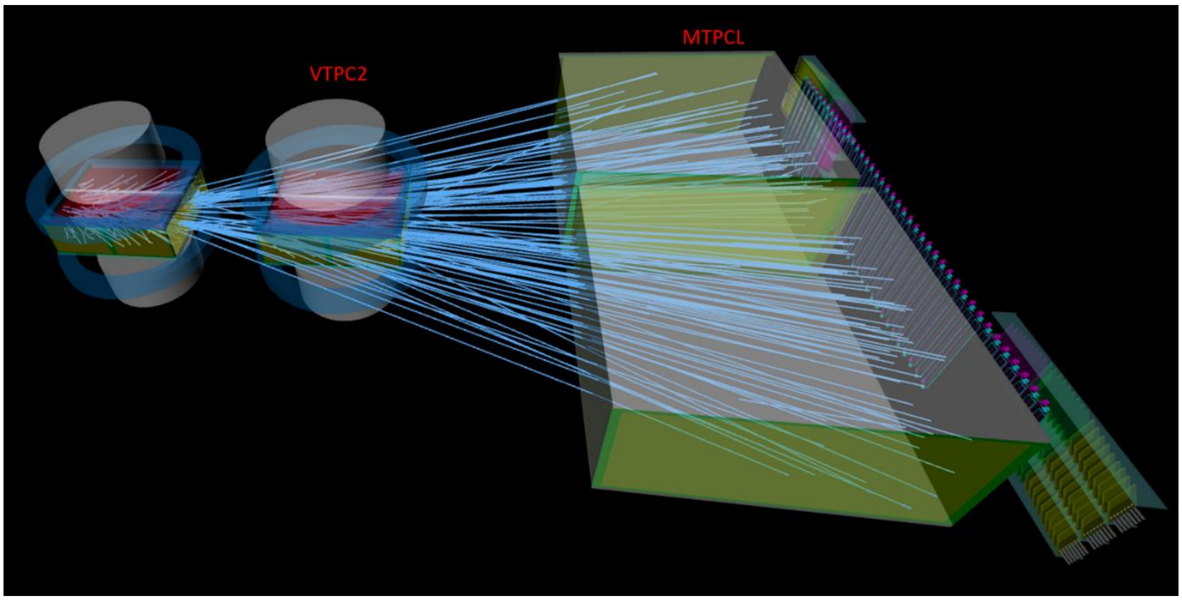


Figure 15: 3D Visualization of TPC chamber of NA61/Shine experiment. Credit: (Michalski & Palayda)

Both data sets consist of samples that contain charge deposition data at twenty time steps from eleven pads referred to as clusters. These clusters are labelled as signal or as noise depending on whether the cluster belongs to a track found by a reconstruction algorithm. Classified as signal if it belongs to a track or as noise if it does not. The eleven chosen pads for each cluster are, the one pad where the average charge deposition was maximal during an event (a particle passing through the chamber), plus the ten nearest neighbours (five to the left and five to the right). Typical signal and noisy clusters are shown in Figure 16 and Figure 17 respectively.

How the data was obtained and pre-processed is explained in more detail in the attachment (Internship at FIAS – A Report from Janik Pawlowski). If one visualizes examples, most samples seem to be distinguishable by eye but there are some that do not fit in the scheme. Note that some signal samples look like typical noise samples and some noise samples like typical signal (Figure 18).

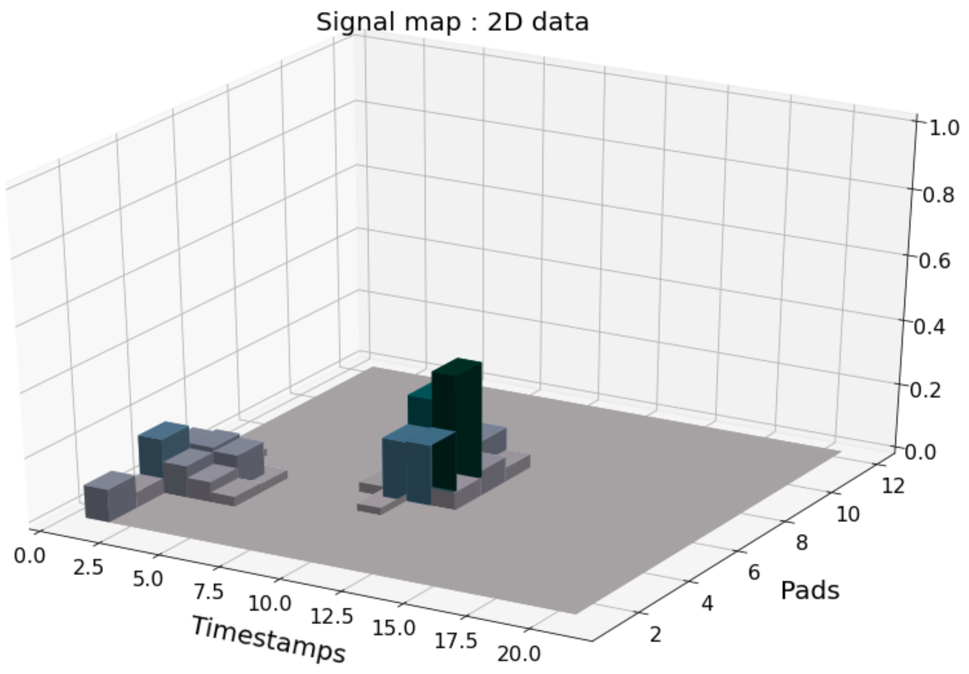


Figure 16: Exemplary signal cluster. Three-dimensional plot of the measured charge amplitude (Z-axis) from the pad where the maximum average charge was measured during an event and its nearest neighbours (Y-axis) at various time steps (X-axis)

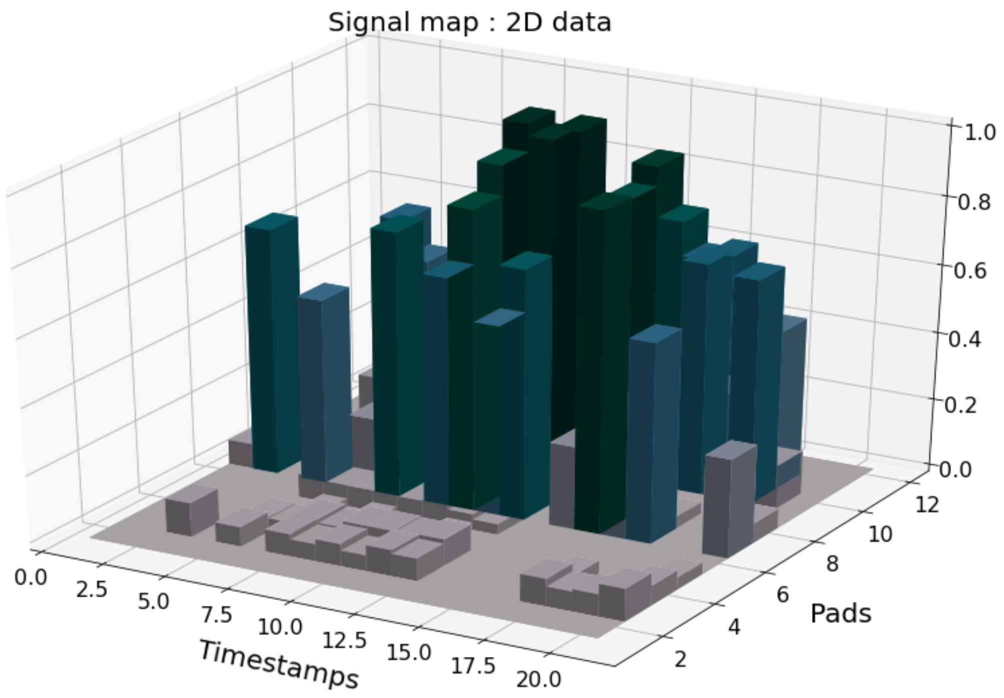


Figure 17: Exemplary noisy cluster. Three-dimensional plot of the measured charge amplitude (Z-axis) from the pad where the maximum average charge was measured during an event and its nearest neighbours (Y-axis) at various time steps (X-axis)

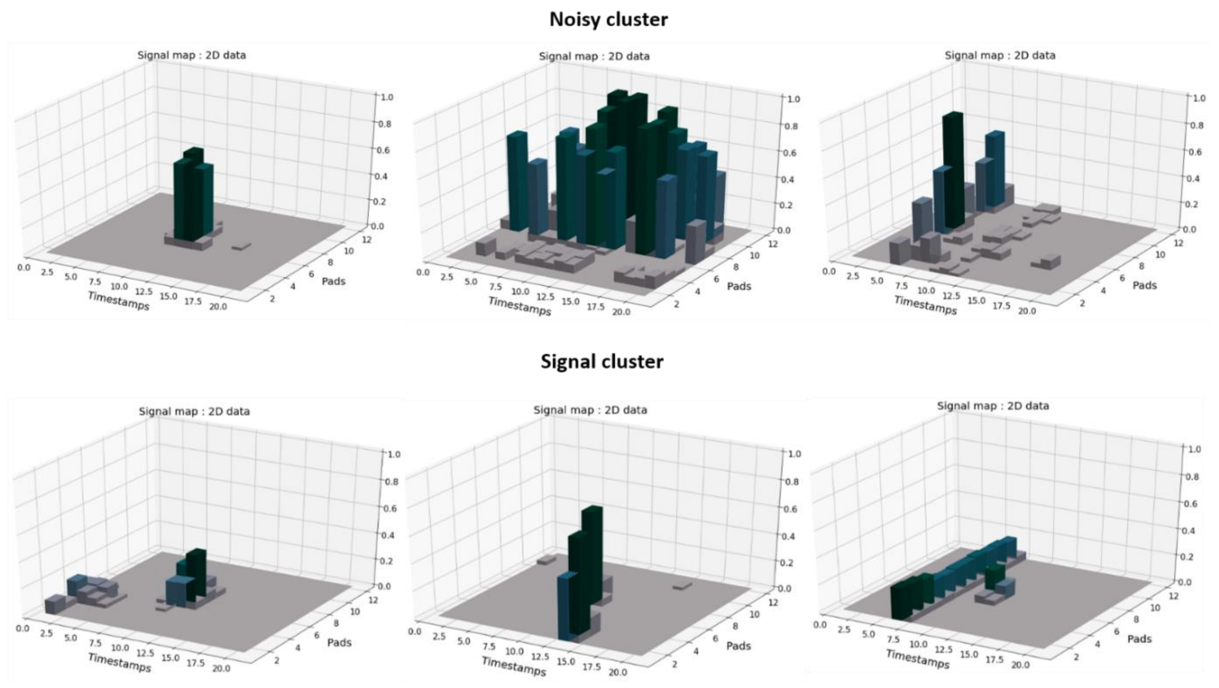
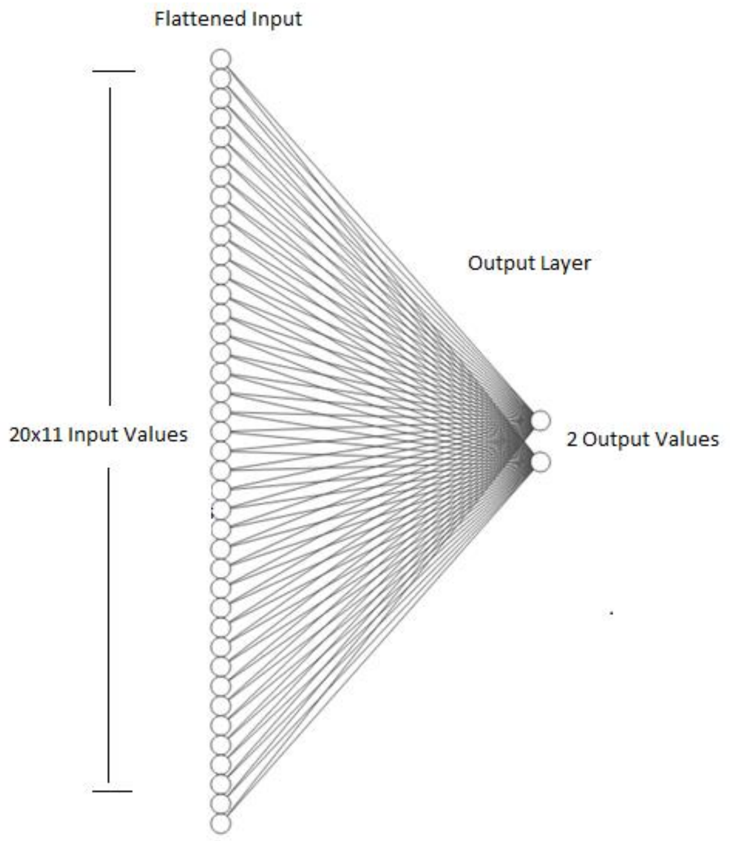


Figure 18: Typical noise and signal clusters – not always distinguishable by eye

3.2 Perceptron

The first approach was to use a simple two-neuron Perceptron where the input is first flattened and then fed directly into two output neurons generating a two-dimensional output vector. Its entries represent the degree to which the network assigns the input sample to one of the two classes, noise or signal. This can be achieved by transforming the output vector with a Softmax activation function so that the sum of the two values always equals one. This ultimately allows a probabilistic interpretation, even if the output does not represent a true probability. For example, a value of 0.99 does not mean that the network is correct in its prediction 99% of the time. However, it is definitely more likely than in an example where it predicts the class with a lower value. Therefore, it is still an indicator of how confident the network is in its prediction, even though it is not a probability in the common sense.

Having eleven times twenty values sums up to 220 values, which are all multiplied by a constant, learned factor $a1_i$ on their way to neuron_1 and by $a2_i$ to neuron_2 (220 + 220 weights), plus applying a bias to each node, sums up to 442 parameters in total (Figure 19). It is common to use the total number of trainable parameters as a first indicator of complexity of a neural network. The number of parameters usually correlates with training and prediction time. Note that modern deep neural networks nowadays easily reach a few millions of trainable parameters (E.g. AlexNet over 60 million parameters on ImageNet (Krizhevsky et al., 2017) , VGG over 100 million parameters (Simonyan & Zisserman, 2014))



Strategy

- Input flattened and fed directly into 2 output neurons (Perceptron)
- Softmax Activation Function

  Output example: (0.2|0.8)

Both outputs combined always add up to 1

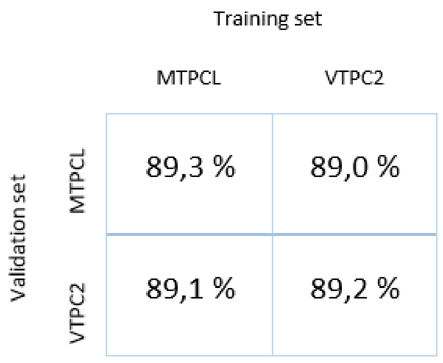
→ can be interpreted as probability for the corresponding class label

Figure 19: One-Layer Perceptron with one fully connected layer (+ input layer)

The beauty of this approach thus lies within the simple structure, which makes it faster as well as easier to get insights on learned patterns. Hence, we are able to understand what the network bases its decisions on. There is a big difference in developing a network, which can classify correctly, but functions like a black box, and one, where it is still possible to understand what is happening. Even though a lot of effort is nowadays put into developing techniques to understand more complex networks (Montavon et al., 2018; Yosinski et al., 2015), simple networks are still easier to understand due to their lower level of complexity. Thus, in this work, the focus during the analysis of the Perceptron lies on understanding what the network learns. This can provide valuable information for more complex approaches. In chapter 3.4 a solution based on these insights is proposed.

3.2.1 Accuracy

Accuracy measurements are very useful as a primary indicator of performance and metric during the training process. In addition, they can be a great comparative metric if verified by other measurements. E.g., verifying a similar prediction accuracy on a different independent data set and making sure the input data is balanced. As shown in Figure 20, the Perceptron is able to label around 89 % of the data correctly for four different setups. This holds true even for the case of training the network on a data set A and validating the networks performance of a second data set B. This shows the Perceptrons ability to generalize since data set A and B consist of data from different TPC’s.



Validation set \ Training set	MTPCL	VTPC2
MTPCL	89,3 %	89,0 %
VTPC2	89,1 %	89,2 %

Figure 20: Perceptron's classification accuracies for both MTPCL and VTPC2 data sets

3.2.2 Confusion Matrix

For the next step, it is useful to look at the number of false positives and false negatives as well as their ratio. This can be an indicator whether the network favour a particular class. Classifiers with asymmetrical distributions of false positives and false negatives may turn out to perform significantly worse in real life than on the training data. Remember the example, where the network would classify every sample as class zero because it was trained on an asymmetrical data set, where 99 % of the data belongs to class zero. Recursively, an asymmetry between FP and FN might indicate an asymmetry in the underlying data set. In this case, there is indeed some systematic disparity between the number of FP and FN. The network falsely predicts around 150 signal clusters as noise, but it falsely predicts more than 250 noisy clusters as signal (if trained and validated on MTPCL (Figure 21)). The same behaviour is observable when training and validating on VTPC2 data, even though the difference between FP and FN is not that large (Figure 22). Even though it may be favourable behaviour in this case, since it is worse to lose signal than to keep noise, the question of why must be asked.

One explanation could be that more noisy clusters exist that look like signal than the other way around. As already shown in Figure 18, clusters are often not distinguishable with the naked eye. In addition, the tracking algorithm is naturally error-prone itself and may mislabel some signal clusters as noise or vice versa. FP and FN clusters would have to be further investigated. One could check for example three-dimensional visualizations of the tracks to check whether those FP and or FN are clustered around specific locations. This could shed light on whether the reconstruction algorithm might be incorrectly classifying tracks in certain edge regions as noise.

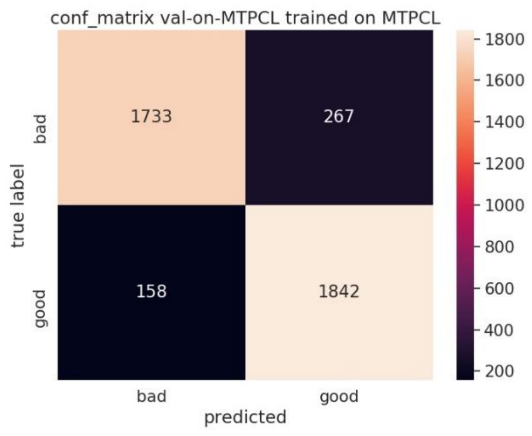


Figure 21: Confusion matrix – Perceptron trained and validated on MTPCL

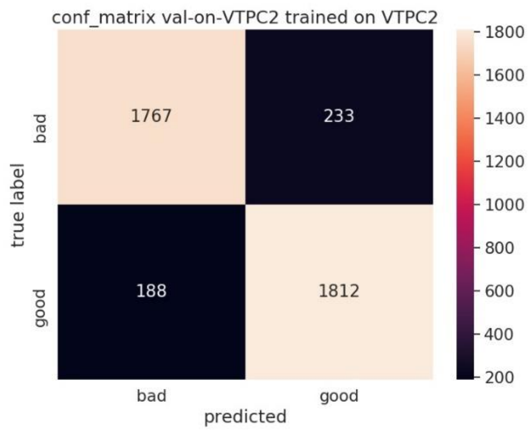


Figure 22: Confusion matrix - Perceptron trained and validated on VTPC2

3.2.3 Training on One-dimensional Data Sets

In addition to the original two two-dimensional data sets from the two TPC's "VTPC2" and "MTPCL" the Perceptron was tested on one-dimensional sub data sets in order to answer the following two questions:

1) Is the network really benefiting from two or higher dimensional data or is one dimension enough?
2) Does one of the dimensions (time or location) yield higher valued information?

The one-dimensional (sub) data sets comprise the following:

The first data set is filled with max charge deposition data from all the 20 time steps but only uses the pad in the middle (pad number five). The second one only comprises data from the 10th time step (middle) but all eleven pads (Figure 23).

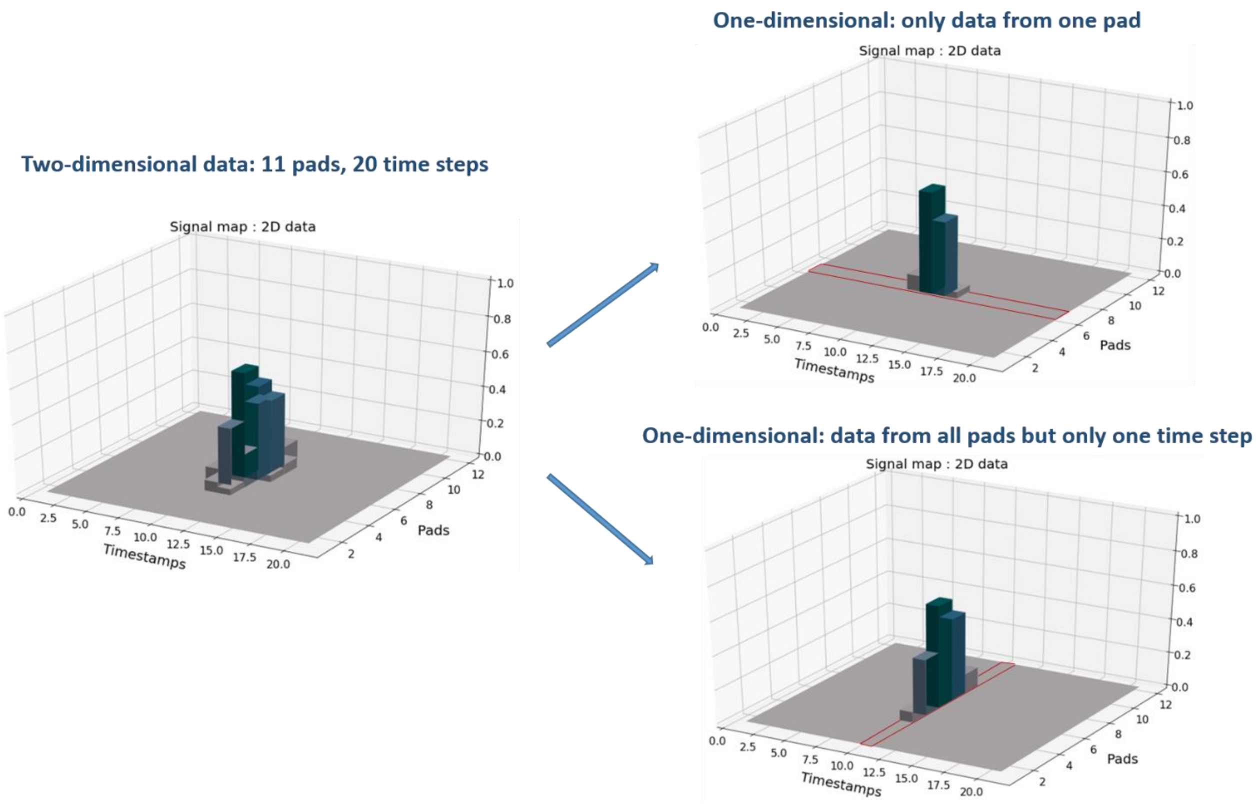


Figure 23: One row or one column was cut out at a time to prepare one-dimensional data sets consisting of slices of the original two-dimensional data

In case of accuracy the network performs best on two-dimensional data. However, while the Perceptron performs significantly worse on data from eleven pads at one time step (11Px1T), there is only a surprisingly small gap between the performance on two-dimensional data and on data from one pad at 20 time steps (1Px20T).

Data	Accuracy
2D – Data (11Px20T)	89,4 %
1 pad, 20 time steps (1Px20T)	88,1 %
11 pads, 1 time step (11Px1T)	86,1 %

Table 1: Accuracies after training on different sub data sets of a two-dimensional data set

This might indicate that information of what happened at other time steps is more important for the learning ability of the network than knowing what happened at the surrounding pads. Nevertheless, it is quite surprising that this simple network architecture is already sufficient to correctly predict at least 86 % (11Px1T) of the samples.

While accuracy measurements suggest the Perceptron performs best when trained and validated on the two-dimensional data set, computing the networks performance on different decision thresholds revealed, that this might not be the case. Over a variety of thresholds, the Perceptron performed best on the 1Px20T data instead of the two-dimensional data. This indicates that it might not be able to take advantage of more dimensional data because of a too simplistic architecture. It will be shown that more complex neural networks, in contrast, benefit from having a two-dimensional data set as training set.

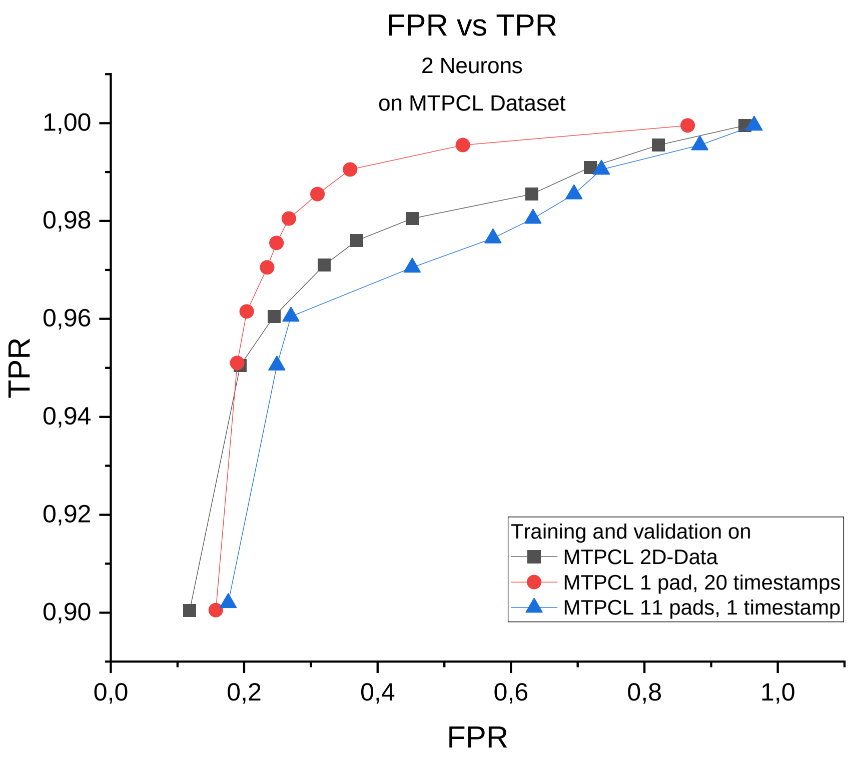


Figure 24: Performance comparison of the Perceptron after training on different sub data sets of MTPCL data

3.2.4 Learned Weights – Decision Boundaries on 1Px20T Data

The learned weights helped to get an insight into what exactly the Perceptron learns on the different data sets. For this purpose, they were plotted in a two-dimensional plot if trained on a one-dimensional data set and in a three-dimensional plot if trained on a two-dimensional data set. Figure 25 illustrates the weights of the Perceptron's neuron, whose output represents to which degree the network predicts the sample to be signal.

A quick reminder: passing through the network, the twenty input values are all scaled by scalar factors, the weights. The sum of these weighted values plus a bias b is then normalized by a Softmax activation function to values between zero and one, representing the degree to which the network predicts the sample to belong to one class or another.

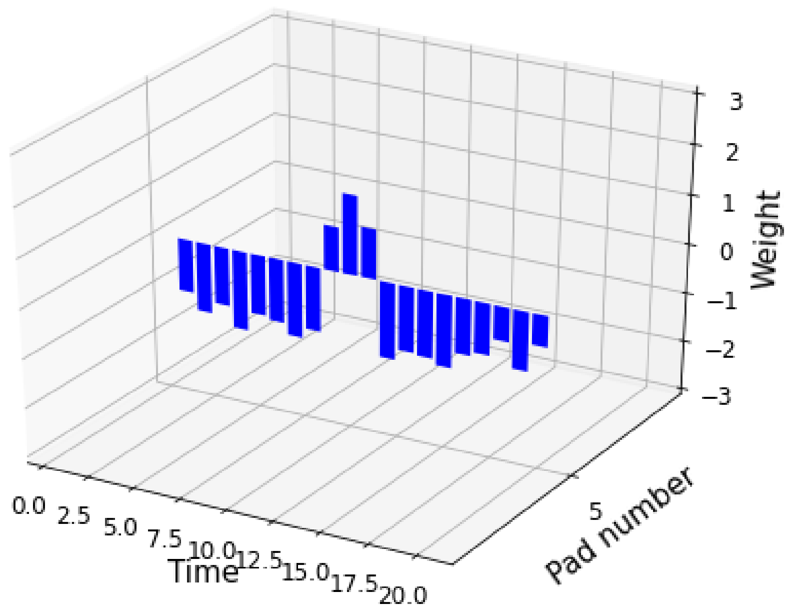


*Figure 25: 2D Plot of the network weights for the second neuron (if output > 0.5 signal cluster) *trained on 1Px20T data*

Plotting the weights revealed, when trained on 1Px20T data, the neurons output is simplified the outer values subtracted from the values in the centre. If this calculation leads to a value greater than 0.5, the network classifies the sample as signal. So if there is a peak in the centre and no peaks outside the centre, it predicts the input to be a signal cluster. If there are peaks outside the centre, it predicts the input to be a noisy cluster.

The other neurons weights plot, the one whose output indicates whether the network predicts the sample to be noise, turned out to represent the same decision criteria, as it is almost an exact mirrored copy of Figure 25 (Figure 26).

The plot of weights, from the neuron whose output indicates to which degree the network predicts the sample to be noise, shows the same decision criteria, being almost an exact mirror of Figure 25.

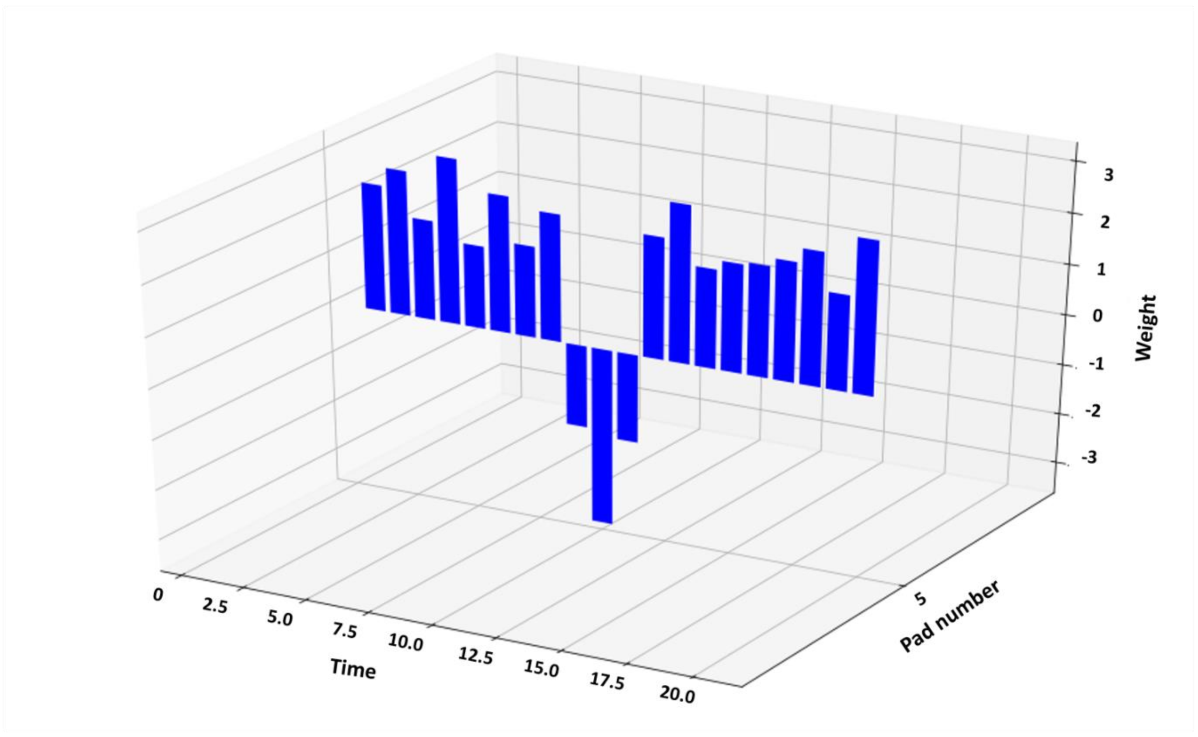


*Figure 26: Plot of the network weights for the second neuron (if output > 0.5 --> noisy cluster) *trained on 1Px20T data*

3.2.5 Learned Weights – Decision Boundaries on 11Px1T Data

To understand the weights the Perceptron adapts when trained on 11Px1T data (Figure 27), it is helpful to recall how the base training data set was created. By definition, all samples, whether they are signal or noise, peak on the middle pad at the middle time step. Therefore, the signal on the pad in the middle alone cannot be sufficient to make a decision, since this is true for both signal and noise. Instead, the network expects that three or at least two of the three pads in the middle must be triggered if signal were involved (Figure 27). In addition, pads farther from the centre are unlikely to be triggered at the same time, so they are weighted negatively, indicating a noisy cluster.

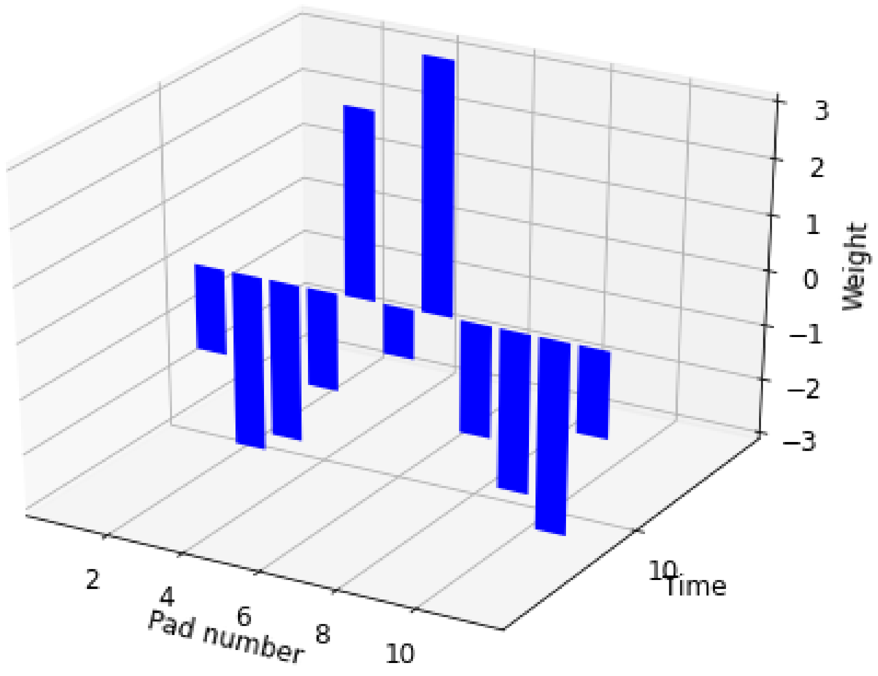


*Figure 27: Learned weights of the first neuron (if output > 0.5 --> signal) * trained on 11Px1T data*

3.2.6 Learned Weights – Decision Boundaries on Two-Dimensional Data

Combining both dimensions helped to understand what the network learns if trained on one of the original two-dimensional data sets (Figure 28).

The output is again calculated by subtracting the outer values from the middle values of the two-dimensional input map (11 pads x 20 time steps). According to this, a sample with a peak in the centre is predicted to be signal, while multiple peaks around the centre should indicate a noisy cluster. This goes hand in hand with the previous observations in chapter 0 and 0, where signal was detected in the shape of multiple peaks observed by multiple pads located in the centre, and multiple peaks outside the centre indicated noise.

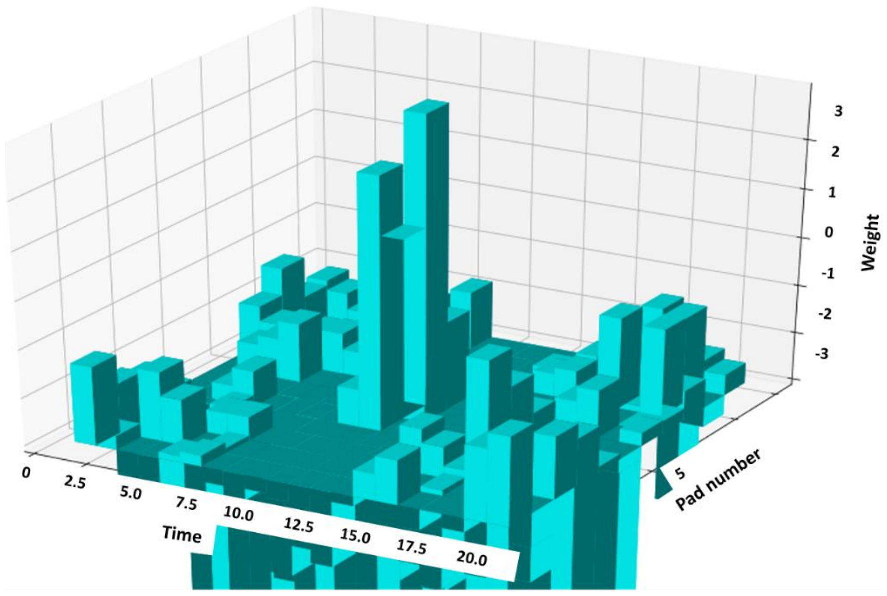


Figure 28: A three-dimensional plot of the learned weights of the neuron that is activated if input is signal

However, there is more to learn from Figure 28. It is unlikely that in all corners (outer pads at early or late time steps) the weights are positive purely by chance. They may not be as high as those in the centre but they are still noticeable. The reason for this could be that the incoming signal, which peaks at the centre also trigger other pads on its way through the chamber. From physics point of view, one could argue that it should not trigger two pads at the same time unless they are close neighbours, as well as a single pad no more than once. This could be the reason why one can observe a cross in the weights map. It becomes more obvious when the weight map is rotated (Figure 29). Values on this cross are weighted positively and the rest is subtracted.

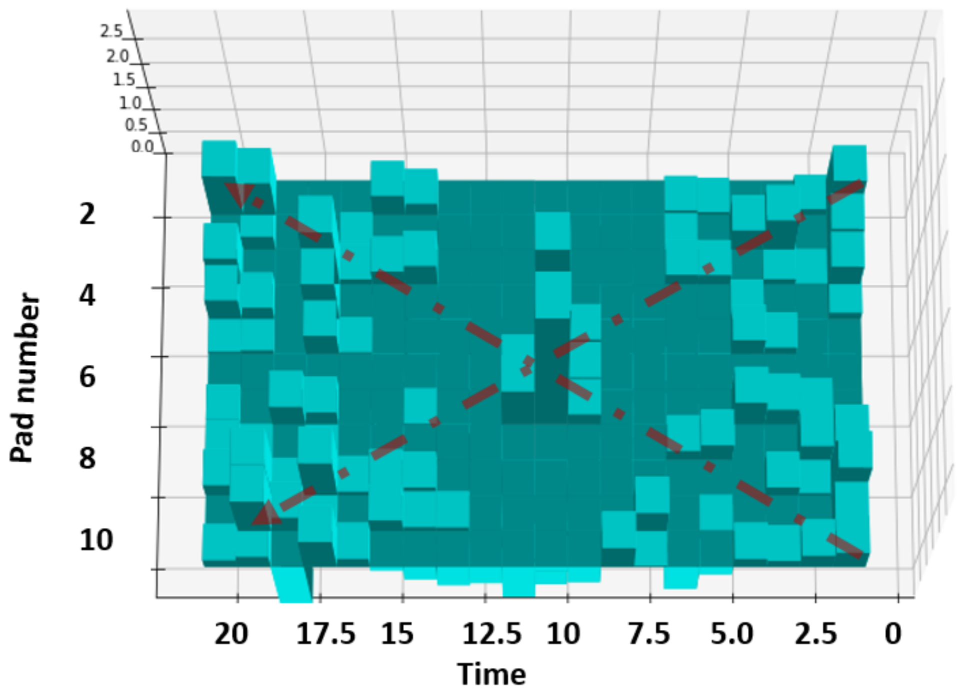


Figure 29: Figure 28 but different angle

3.3 ResNet

The second approach is a very popular one, the so-called ResNet, which contains shortcut connections in between hidden layers. While the network itself is based on convolutional blocks, there are parallel connections between these blocks, which feed the input straight through the network.

Over the last couple of years, residual neural networks (short ResNet) and networks based on this architecture, have proven to be very powerful (Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A., 2017). Especially in the field of image classification in a variety of tasks (e.g., hyperspectral image interpretation (Zhong et al., 2017)). While they can be computationally intensive, they are often able to show better overall performance and solve problems with better accuracy than their non-residual counterpart (Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A., 2017).

As illustrated in Figure 30, the two-dimensional input is first fed into an input layer (blue). One copy is then send through a block of convolutional layers (brown), which consists of one layer with 32 3x3-filters and a second layer with 64 2x2-filters (red-dashed), the second copy is parallel forwarded via a shortcut. The output of the first ResNet block is calculated by the output of the convolutional block plus the original (purple). Both copies are normed by a ReLU activation function prior to summation (for clarity reasons not shown in the graph). The convolutional part of the network consists of three such blocks. A final layer of convolutional filters that feeds into a global-average-pooling layer (green) follows these three blocks. Its output is fed into a fully connected dense neural network (cyan) consisting of first a 128-nodes layer followed by a dropout layer (grey), second of a 64-nodes layer followed by a dropout layer and third of a two-nodes output layer (corresponding to the two classes noise and signal).

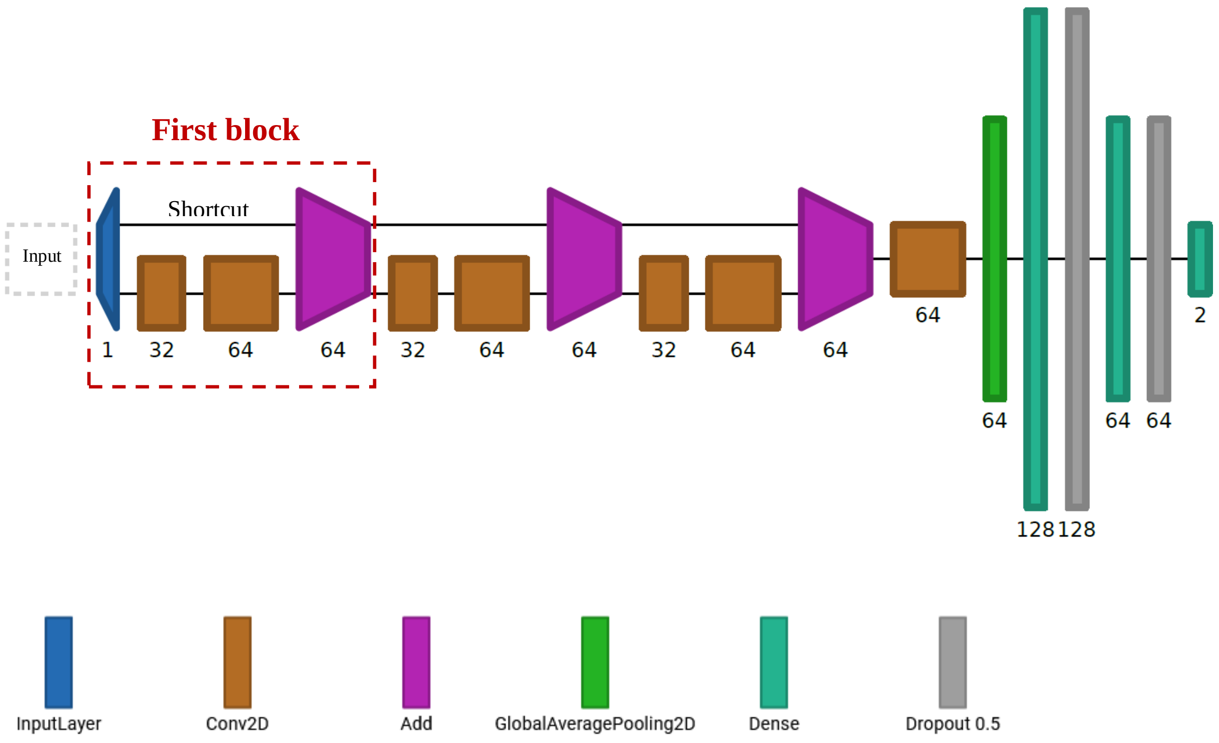


*Figure 30: Schematic structure of ResNet *Credit: created with Net2Vis (Bäuerle et al., 2019)*

Summary of the strategy:

- Input fed into three blocks of convolutional layers with shortcuts in between blocks
- Feature maps are then averaged to a single value (GlobalAveragePooling)
- Averaged scalar values fed into a dense neural network
- 2 outputs corresponding to noise and signal

3.3.1 Accuracy

ResNet seems to be performing best on two-dimensional data and in general better than the Perceptron. The comparison of performance on the different data sets mirrors the same trend as observed with the two-neuron Perceptron but with clearer distinctions. ResNet is able to classify around 92 % correctly if trained on 11Px20T data sets. On one-dimensional data, it achieves accuracies of about 89 % for the 1Px20T data and 85% for the 11Px1T data (Table 2).

Data	Accuracy
2D – Data (11Px20T)	92,0 % (89)
1 pad, 20 time steps (1Px20T)	89,3 % (88)
11 pads, 1 time step (11Px1T)	85,4 % (86)

Table 2: Accuracies of ResNet and Perceptron (brackets) on different (sub) data sets of MTPCL data

In contrary to the Perceptron, ResNet seems to be able to benefit from two-dimensional input since its threshold performance matches the same trend as the accuracy measurements (Figure 31).

For example, one can set the decision threshold so that the network achieves a TPR of 97%, i.e., that 97% of the signal samples are correctly detected as signal and about 3% are incorrectly classified as noise. ResNet then has an FPR of about 16% when trained on two-dimensional data, ~21% on 1Px20T data, and ~24% FPR on 11Px1T data. This means that at those thresholds, the network will misclassify either 16%, 21%, or 24% of noisy samples as signal (Table 3).

This highlights that in case of the ResNet, it may indeed be advantageous to use the two-dimensional data. A very high TPR is possible in exchange for keeping some noisy cluster. For example, about 74% of the noisy clusters can be filtered out while only about one percent of the signal is lost (99% TPR).

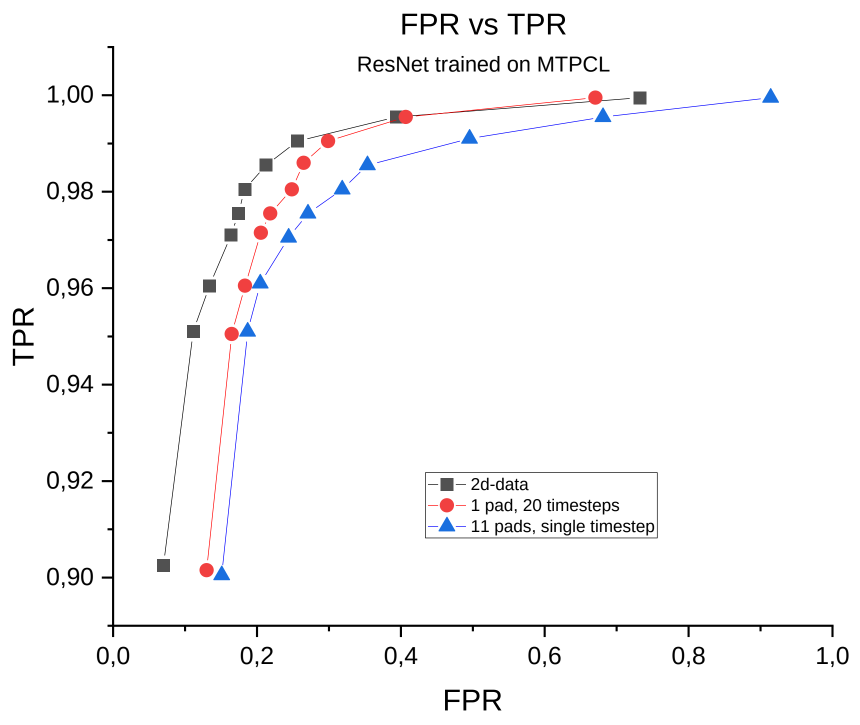
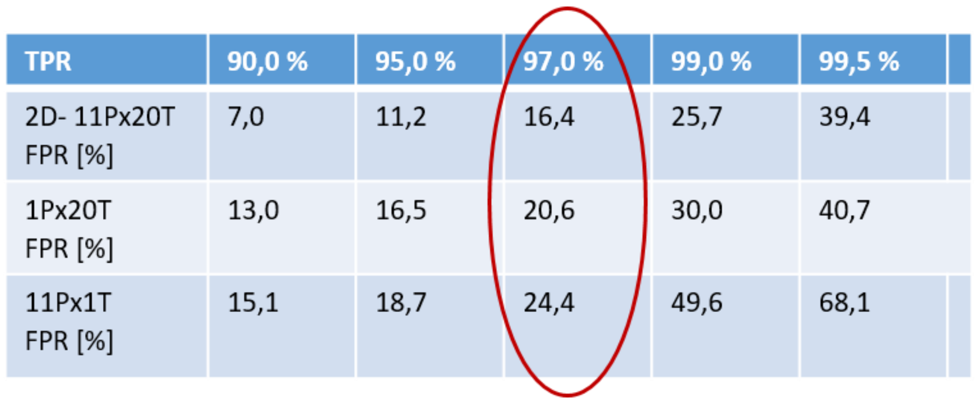


Figure 31: Performance comparison of ResNet on different sub data sets of MTPCL data



TPR	90,0 %	95,0 %	97,0 %	99,0 %	99,5 %
2D- 11Px20T FPR [%]	7,0	11,2	16,4	25,7	39,4
1Px20T FPR [%]	13,0	16,5	20,6	30,0	40,7
11Px1T FPR [%]	15,1	18,7	24,4	49,6	68,1

Table 3: ResNet's TPR and FPR for different thresholds on the one- and two-dimensional data sets

3.4 Split Convolution – Use Prior Knowledge to Invent a Custom Network

Analysing the learned weights of the Perceptron showed that a complex convolutional neural network is most likely not necessary for this classification problem. On the contrary, a complex network might be particularly prone to overfitting. As discussed in 3.2.6, the decision made by the Perceptron seems to be based on a cross, where all values on the cross are weighted positively and values outside the cross are subtracted. Furthermore, as shown in 3.2.3, the two one-dimensional data sets alone were sufficient for a Perceptron to produce good results (86-88% accuracy). On paper the Perceptron even had the highest accuracy if trained on two-dimensional data, however, it is not really able to take advantage of it. The performance on 1Px20T data was actually better (Figure 24). These circumstances lead to the idea of a split convolutional neural network, where the input is processed in two separate ways. The two-dimensional input is split in two exactly equal copies of the original input. Instead of conventional nxn-filters, $nx1$-filters on one path and $1xn$-filters on the other path are used to extract features. The feature maps are then concatenated, flattened, and finally fed into a single layer Perceptron like the one introduced in section 2.1.

This way, the network can learn filters for all the pads on their own, which find patterns in time behaviour and simultaneously, learn to recognize dependencies between different pads at the same time step. In Addition, with this type of structure, it is more difficult to learn patterns between different pads at different time steps. This is an intended consequence, as it is possible that such complex patterns may not exist and would therefore be more likely to result in overfitting. The network might "learn by rote" instead of learning general patterns, which would translate into much better performance on the training data set compared to the validation data set.

The Split Convolution architecture allows the network to learn in two separate ways, which are connected only after convolutional feature extracting has been completed. Because of the shape of the convolutional filters, it is as if the network extracts features from two separate one-dimensional data sets instead of one combined two-dimensional data set. This way the network is expected to find similar features as the Perceptron found on the one-dimensional sub data sets. But these learned features will then be combined and used as input for a single layer Perceptron, which makes the final decision.

The Split Convolution network, just like the others, is a feed forward neural network. Starting from the left, the two-dimensional input map is split into two exact same copies (blue). Followed by two sets of convolutional layers (brown) plus an activation layer (red) each. One channel consists of five 3x1-filters, the other consists of five 1x3-filters. The output feature maps of both paths are then concatenated (purple), normalized by another activation function, flattened (orange), and finally fed into the output layer (green), which consists of 2 neurons (Figure 32).

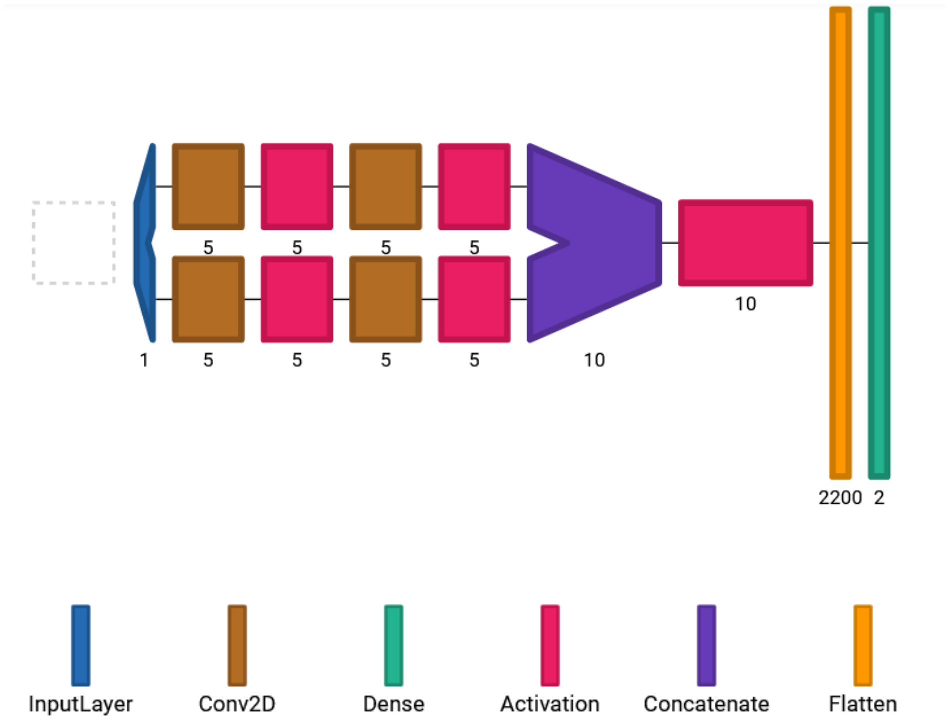


Figure 32: Structure of the Split Convolution network. Created with Net2Vis (Bäuerle et al., 2019)

Summary of the strategy:

- Feed Input into two separate blocks of one-dimensional convolutional layers
- Concatenate and flatten feature maps
- Feed the flattened values into two output nodes corresponding to the two classes signal and noise

This architecture leads to 4.600 trainable parameters, so approximately 5 % of ResNet.

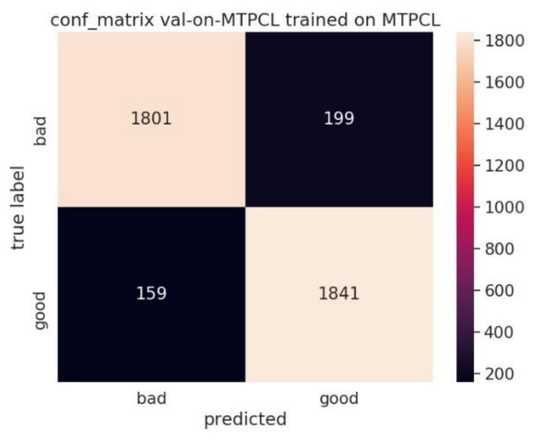
3.4.1 Accuracy

Again, as a first indicator of performance, accuracy was measured. Split Convolution achieves an accuracy of about 91 % or 92% when trained and validated on the two-dimensional MTPCL and VTPC2 data set respectively (Table 4). To check whether the network may be overfitting, the accuracy was measured on MTPCL data after training on VTPC2 data and vice versa. The results are quite promising: about 90.5% of VTPC2 data samples were correctly classified after training on MTPCL data and about 89.4% of MTPCL data samples were correctly classified after training on VTPC2 data (Table 4). This demonstrates its ability to generalize, since it is able to correctly classify about 90% of the data from another TPC. This corresponds to a loss of 1-2% compared to the prediction accuracy on data from the TPC it was trained on.

Training dataset	Accuracy on MTPCL-2D	Accuracy on VTPC2-2D
MTPCL-2D	91,1 %	90,5 %
VTPC2-2D	89,4 %	91,9 %

Table 4: Validation accuracy of Split Convolution when trained and validated on different two-dimensional data sets

Split Convolution's prediction errors are also not exactly symmetric. On MTPCL, out of 4.000 samples 159 positive examples were falsely classified as noise (FNR = 4 %) but 199 negative examples were falsely classified as signal (FPR = 5 %). On VTPC2, 141 false negatives (FNR = 3,5 %) and 184 false positives (FPR = 4,5 %) were counted. With an average gap between FPR and FNR of one percent only, the difference is smaller than for ResNet and Perceptron, but shows the same trend.



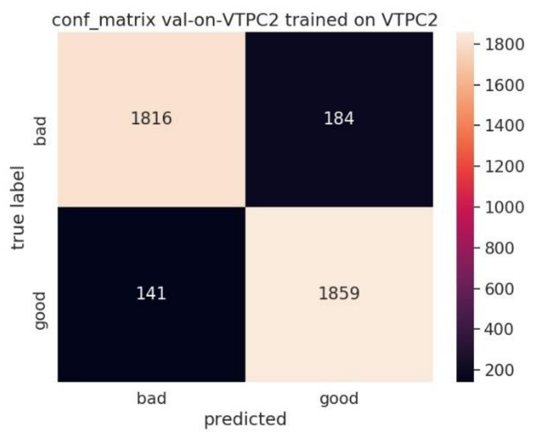


Figure 33: Confusion matrices of Split Convolution, when trained and validated on MTPCL data (left) or VTPC2 data (right)

3.4.2 Performance – Threshold Tweaking

The default decision threshold in binary classification is 0.5. If a neurons output is greater than 0.5, then it is labelled correspondingly. By adjusting the threshold, it is possible to move the prediction to a specific class. The optimal threshold adjustment depends heavily on the use case of the classifier. E.g., it might be okay to lose some signal as long as most of the noise is filtered. Then, you would have to look for an optimal threshold which combines a very low FPR with an acceptable TPR. Alternatively, the exact opposite is true and it is important not to waste any good data. Then, you would have to look for a threshold, which combines a high TPR with an acceptable FPR.

For example, for the MTPCL data, Split Convolution correctly classifies 99% of positive examples for a certain threshold, while 30% of negative examples are incorrectly classified as signal at that threshold (Figure 34). The higher the desired TPR, the higher the FPR one has to put up with. An extreme example would be 100% TPR, this would most certainly lead to 100% FPR, since the network would simply classify everything as signal to achieve this goal.

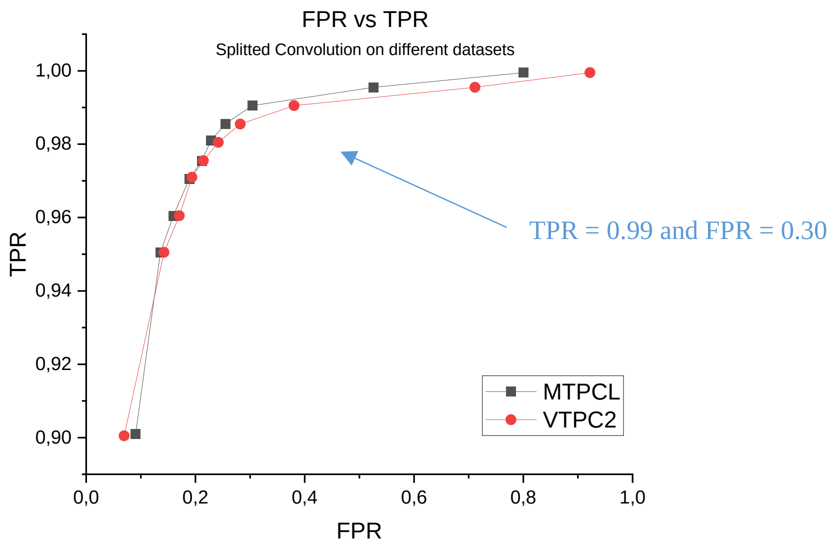


Figure 34: Threshold performance after training on MTPCL data (black) and VTPC2 data (red)

3.4.3 MTPCL vs. VTPC2

The first impression was that the three tested neural networks perform slightly better on VTPC2 data than on MTPCL data because higher accuracies were achieved. However, the comparison of threshold performance showed that this is not necessarily the case. The overall performance seems to be very similar for both data sets the Split Convolution network even seems to perform slightly better when trained and validated on MTPCL data (Figure 34).

3.5 Comparison – Perceptron vs. ResNet and Split Convolution

All three types of network architectures and their strengths and weaknesses will be discussed in the following. Therefore, one more network property has to be taken into account. It's the question how long does the network take to predict a certain set of samples.

3.5.1 Prediction Speed

Generally, the prediction speed depends very much on the setting. While simple artificial neural networks show a fast performance on central processing units (CPU), computations of more complex networks are usually significantly faster on graphics processing units (GPU).

The prediction speed was measured for all three models on a middle- to low-end notebook CPU as well as on a high-end GPU from the google colaboratory GPU cluster.

CPU:	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2712 MHz, 2 kernels, 4 logical processors
GPU:	NVIDIA® Tesla T4-GPU

In terms of speed, the Perceptron clearly stands out from Split Convolution and ResNet with an average prediction speed on CPU of 77.000 samples/s versus 14.000 samples/s for Split Convolution and 2.000 samples/s for ResNet respectively (Figure 35).

Perceptron is also the fastest on the GPU, although its prediction speed drops to 53,000 samples/s. Split Convolution and ResNet, however, increase greatly in speed to an average of 39,000 samples/s and 31,000 samples/s, an increase of 300% and 1500%, respectively.

It is important to point out again that the prediction speed depends heavily on the hardware, and thus on the available resources, the code is running on. For example, if Word or a web browser was running parallel to the Spyder Python environment were the code was running, this was noticeable in speed changes of up to 20-30%.

The above values can therefore serve as an orientation, but for a reliable statement it will be necessary to test the models on the actual hardware.

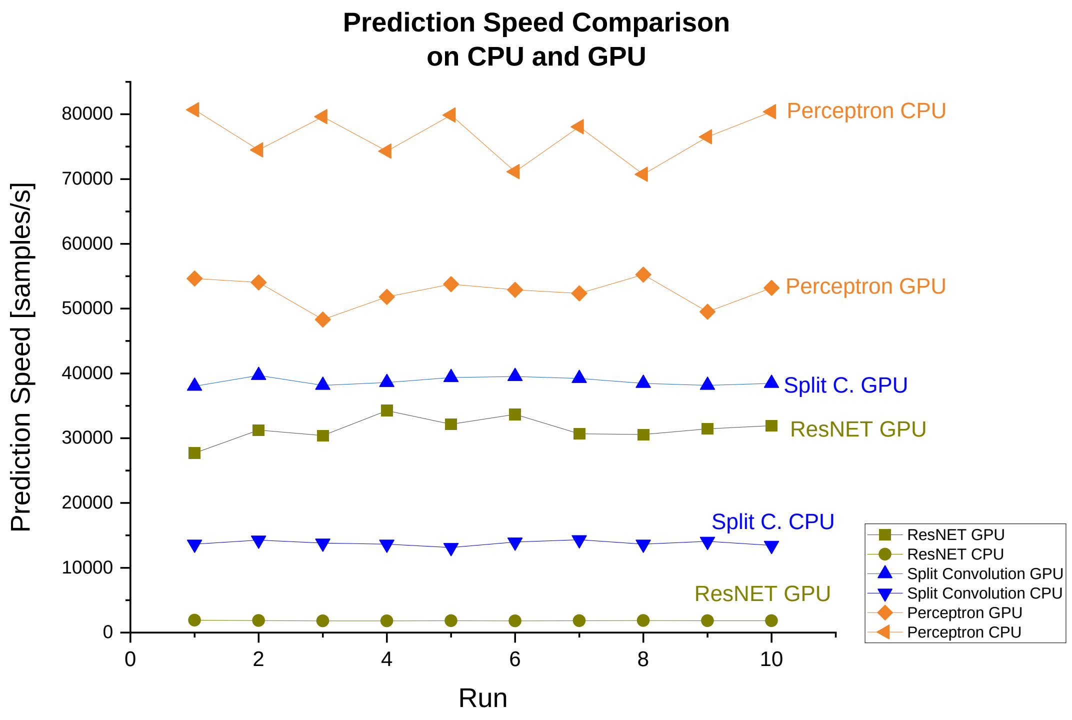


Figure 35: Prediction speed comparison of Perceptron (orange), Split Convolution (blue) and ResNet (green)

3.5.2 Threshold Performance

In a direct performance comparison for different decision thresholds, the Perceptron, although achieving prediction accuracies of up to 89%, is clearly inferior to ResNet and Split Convolution (Figure 36). ResNet and Split Convolution show a very similar performance with a small lead for ResNet

If the threshold is shifted in such a way that a higher TPR is achieved, the FPR quickly rises very sharply for the Perceptron. ResNet and split convolution, on the other hand, show a more stable threshold behaviour. A TPR of 99% can be achieved without the model simply classifying everything as positive. Of course, the FPR also increases with the TPR, but slower in comparison

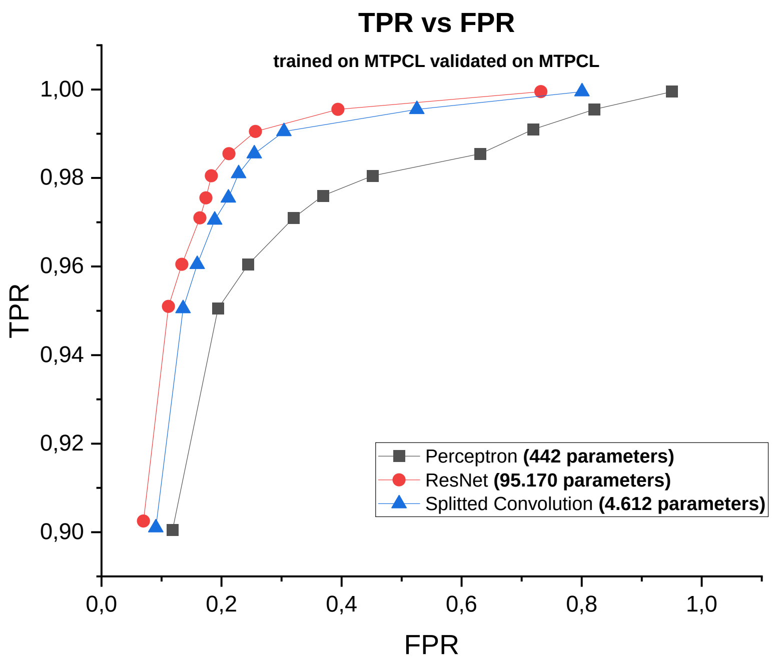


Figure 36: TPR and FPR, for Perceptron, Split Convolution and ResNet at various thresholds

4 Summary

All three of the networks showed different strengths and weaknesses. Depending on which feature is most important, different models are proposed:

1) If speed matters the most:

   The Perceptron seems the most promising, it is the simplest artificial neural network, which makes it very fast in its training and prediction time (53.000 and 77.000 samples/s on GPU and CPU respectively). It reduces probably up to 87 – 88% of the noise. The downside is that one probably loses up to 10 % of the signal. This behaviour can be influenced via adjusting the decision threshold. For example, it is possible to achieve noise reduction of about 68% while losing only three percent of the good clusters, or about 80% while losing five percent of the signal (Figure 36, Table 5).
   However, it will never be as accurate as ResNet and Split Convolution.

2) If the goal is to lose as few signal as possible:

   Then, ResNet is the most promising. It was able to correctly predict about 92 % of all clusters and achieved a TPR of up to 95%, which means that only five to seven percent of the good matches are lost. In addition, ResNet showed good threshold performance. The threshold could be adjusted in order to achieve a TPR of 99% and still reduce about 74% of the noise at that threshold (Table 5). However, it is definitely the slowest of the three networks (2000 samples/s on GPU). Even though it might be possible to reduce the number of parameters, due to its level of complexity, it will never be as fast as the other two networks.

TPR	90,0 %	95,0 %	97,0 %	99,0 %	99,5 %
Perceptron‘s FPR [%]	11,9	19,4	32,0	71,9	82,1
ResNet‘s FPR [%]	7,0	11,2	16,4	25,7	39,4
Splitted Convolution‘s FPR [%]	9,1	13,6	18,9	30,4	52,6

Table 5: TPR vs. FPR performance comparison of Perceptron, ResNet and Split Convolution

3) A compromise of both:

   On the one hand, Split Convolution has shown a very similar performance to ResNet. While ResNet performed better by a tiny margin, Split Convolution needs less than five percent of the parameters (4.600 to 95.00) and it has in general a less complex architecture. Thus, it is operating much faster and able to predict around 39.000 samples a second and it clearly outperforms the Perceptron. On the other hand, it is still slower than the Perceptron, especially when run on CPU (14.000 samples/s vs. 77.000 samples/s).

5 Outlook

Above all, this work shows that dense feed-forward and convolutional neural networks can in principle be successfully used to filter noisy clusters in the TPC data of the NA61/SHINE experiment.

Note that a little scepticism is healthy and only natural. Especially when one has to trust an algorithm whose decision path may not be fully comprehensible. The following tests are therefore proposed to increase confidence in the proposed machine and deep learning algorithms, but would have been beyond the scope of this work.

Proposal:	Learning and prediction based on cross-energy and/or cross-particle collisions.
Example 1:	Train a classifier on Argon-Scandium data and test it on data from Lead-Lead collisions. Likewise, of course, the other way around.
Example 2:	Train a classifier on x MeV Argon-Scandium data on and test it on y MeV Argon-Scandium data, with $x \neq y$.

Positive tests will increase confidence in the AI's decision-making, as they would highlight the ability to learn generalized patterns.

It is worth noting that although all three networks have shown good results, they are obviously not the only options available. For example, even though ResNet was still state of the art in 2020, many other promising approaches have also been introduced since ResNet was first introduced in 2015. However, this work highlights the great potential in further exploring the application of artificial neural networks as a method for noise filtering in the NA61/SHINE experiment. The world is changing rapidly, especially in the field of artificial intelligence, and it is exciting to see what future research might bring.

Bibliography

[1]Abdel-Hamid, O., Mohamed, A.-r., Jiang, H., Deng, L., Penn, G., & Yu, D. (2014). Convolutional Neural Networks for Speech Recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, *22*(10), 1533–1545. https://doi.org/10.1109/TASLP.2014.2339736

[2]Ahmed Fawzy Gad. *Implementing Gradient Descent in Python Part 2*. https://blog.paperspace.com/part-2-generic-python-implementation-of-gradient-descent-for-nn-optimization/

[3]Angermueller, C., Pärnamaa, T., Parts, L., & Stegle, O. (2016). Deep learning for computational biology. *Molecular Systems Biology*, *12*(7), 878. https://doi.org/10.15252/msb.20156651

[4]Baldi, P., Sadowski, P., & Whiteson, D. (2014). Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, *5*, 4308. https://doi.org/10.1038/ncomms5308

[5]Bäuerle, A., van Onzenoodt, C., & Ropinski, T. (2019, February 11). *Net2Vis -- A Visual Grammar for Automatically Generating Publication-Ready CNN Architecture Visualizations*. http://arxiv.org/pdf/1902.04394v5

[6]Bock, S., & Weis, M. (2019, July). A Proof of Local Convergence for the Adam Optimizer. In *2019 International Joint Conference on Neural Networks (IJCNN)* (pp. 1–8). IEEE. https://doi.org/10.1109/IJCNN.2019.8852239

[7]Ganapathiraju, A., Hamaker, J., & Picone, J. (2004). Applications of Support Vector Machines to Speech Recognition. *IEEE Transactions on Signal Processing*, *52*(8), 2348–2355. https://doi.org/10.1109/TSP.2004.831018

[8]Hara, K., Saito, D., & Shouno, H. (2015, July). Analysis of function of rectified linear unit used in deep learning. In *2015 International Joint Conference on Neural Networks (IJCNN)* (pp. 1–8). IEEE. https://doi.org/10.1109/IJCNN.2015.7280578

[9]He, K., Zhang, X., Ren, S., & Sun, J. (2015, December 10). *Deep Residual Learning for Image Recognition*. http://arxiv.org/pdf/1512.03385v1

[10] https://www.tensorflow.org/. *Tensorflow*. https://www.tensorflow.org/

[11] Ismail Fawaz, H., Forestier, G., Weber, J., Idoumghar, L., & Muller, P.-A. (2019). Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, *33*(4), 917–963. https://doi.org/10.1007/s10618-019-00619-1

[12] Kingma, D. P., & Ba, J. (2014, December 22). *Adam: A Method for Stochastic Optimization*. http://arxiv.org/pdf/1412.6980v9

[13] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90. https://doi.org/10.1145/3065386

[14] Michalski, F., & Palayda, T. *3D Visualization of NA61/Shine Experiment*. https://shine3d.web.cern.ch/

[15] Montavon, G., Samek, W., & Müller, K.-R. (2018). Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, *73*, 1–15. https://doi.org/10.1016/j.dsp.2017.10.011

[16] NA61/SHINE Team. *Physics*. https://shine.web.cern.ch/node/5

[17] NA61/SHINE Team. *Strong Interactions*. https://shine.web.cern.ch/node/12

[18] NA61/SHINE Team. *Time Projection Chambers*. https://shine.web.cern.ch/node/17

[19] Nassif, A. B., Shahin, I., Attili, I., Azzeh, M., & Shaalan, K. (2019). Speech Recognition Using Deep Neural Networks: A Systematic Review. *IEEE Access*, *7*, 19143–19165. https://doi.org/10.1109/ACCESS.2019.2896880

[20] Padmanabhan, J., & Johnson Premkumar, M. J. (2015). Machine Learning in Automatic Speech Recognition: A Survey. *IETE Technical Review*, *32*(4), 240–251. https://doi.org/10.1080/02564602.2015.1010611

[21] Ruder, S. (2016, September 15). *An overview of gradient descent optimization algorithms*. http://arxiv.org/pdf/1609.04747v2

[22] RUMELHART, D. E., HINTON, G. E., & WILLIAMS, R. J. (1988). Learning Internal Representations by Error Propagation. In *Readings in Cognitive Science* (pp. 399–421). Elsevier. https://doi.org/10.1016/B978-1-4832-1446-7.50035-2

[23] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0

[24] Sharma, S. (2017). Activation functions in neural networks.: Towards Data Science, 6. https://www.ijeast.com/papers/310-316,Tesma412,IJEAST.pdf

[25] Simonyan, K., & Zisserman, A. (2014, September 4). *Very Deep Convolutional Networks for Large-Scale Image Recognition*. http://arxiv.org/pdf/1409.1556v6

[26] Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. (2017). Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning.: Proceedings of the AAAI Conference on Artificial Intelligence, 31(1). https://ojs.aaai.org/index.php/AAAI/article/view/11231

[27] Talafha, S., & Rekabdar, B. (2019, January - February). Arabic Poem Generation with Hierarchical Recurrent Attentional Network. In *2019 IEEE 13th International Conference on Semantic Computing (ICSC)* (pp. 316–323). IEEE. https://doi.org/10.1109/ICOSC.2019.8665603

[28] Wu, M., & Chen, L. (2015, November). Image recognition based on deep learning. In *2015 Chinese Automation Congress (CAC)* (pp. 542–546). IEEE. https://doi.org/10.1109/CAC.2015.7382560

[29] Yosinski, J., Clune, J., Nguyen, A., Fuchs, T., & Lipson, H. (2015, June 22). *Understanding Neural Networks Through Deep Visualization*. http://arxiv.org/pdf/1506.06579v1

[30] Zhong, Z., Li, J., Ma, L., Jiang, H., & Zhao, H. (2017, July). Deep residual networks for hyperspectral image classification. In *2017 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)* (pp. 1824–1827). IEEE. https://doi.org/10.1109/IGARSS.2017.8127330

Table of Figures

Internship at FIAS – A Report from Janik Pawlowski

The NA61/SHINE Collaboration

During the internship, I have worked on a very specific project as part of the NA61/SHINE collaboration with researchers at CERN, Switzerland.

NA61/SHINE (standing for "SPS Heavy Ion and Neutrino Experiment") is a scientific research project for particle physics at the Super Proton Synchrotron (SPS) at CERN in Switzerland. About 135 physicists from 14 countries and 35 institutions are working on the NA61 project under the leadership of Marek Gazdzicki. On behalf of the collaboration, FIAS is conducting research on whether and to what extent artificial intelligence methods could be used in the filtering of measurement data, or more precisely in the filtering of noise.

The NA61/SHINE experiment uses beams of hadrons and nuclei from the SPS to measure the production of hadrons. Three different types of collisions are performed(NA61/SHINE Team):

1. Nucleus-nucleus, proton-proton and proton-nucleus:
   (part of the strong interactions programme)

   To investigate properties of the transition between the quark-gluon plasma and hadron gas by collision energy scans with various beam and target nuclei.

   > “NA61/SHINE aims to uncover properties of the onset of deconfinement by systematic and precise measurements of collision energy and nuclear mass dependence of its signals. It is also looking for evidence of a critical point on the transition line between two phases of strongly interacting matter: quark-gluon plasma and hadron gas.”(NA61/SHINE Team)

2. Proton-nucleus:
   (part of the neutrino programme)

   Recording of their interactions with the goal to determine parameters of neutrino beams produced at J-PARC, Japan and Fermilab, US.

3. Hadron-nucleus:
   (part of the cosmic ray programme)

   Interactions are measured in order to improve the modelling of cosmic ray showers. This information is needed for studies on very high-energy protons and nuclei of extragalactic origin, which are performed by the Pierre Auger Observatory and KASKADE experiments.

The main tracking devices of the NA61/SHINE experiment are four large volume Time Projection Chambers (TPC).

The Time Projection Chamber

A TPC consists of a large gas-filled volume. A uniform vertical electric field is applied by a surrounding field cage.

Particles passing through the chamber ionize the gas, leaving a trail of electrons. Due to an external electric field, the electrons drift with a constant velocity in the direction of the upper plate. There their position, arrival time, and total number is measured. The chamber top plates are divided into pads of about one square centimeter in order to achieve a high spatial resolution.(NA61/SHINE Team)

The four TPCs can be divided into two main TPCs, main TPC Left (MTPCL) and Main TPC Right (MTPCR), and two vertex TPCs (VTPC1 and VTPC2).

Each Main TPC has a readout surface at the top of around 3,9x3,9 m² and height of the field cage of around 1.1 m. They are filled with a mixture of gas composed of 95 parts of Argon and 5 parts of CO2. Twenty-five proportional chambers read out values. This way, up to 90 measurements of each particles trajectory are possible.(NA61/SHINE Team)

Each Vertex TPC has a top surface area of 2.0x2.5 m² and 0.67 m depth. Because of the electric field structure, regions of 0.12 m on each side of the beamline are excluded. The particle density in Pb + Pb reactions is so high that trajectories cannot be resolved. The gas consists of a 90/10 mixture of Argon and CO_2. Six proportional chambers perform the readout on the top, providing up to 72 measurements on the particle trajectories.(NA61/SHINE Team)

The experiments setup is shown in Figur and a 3D visualization of the measurements after a collision is shown in Figure.

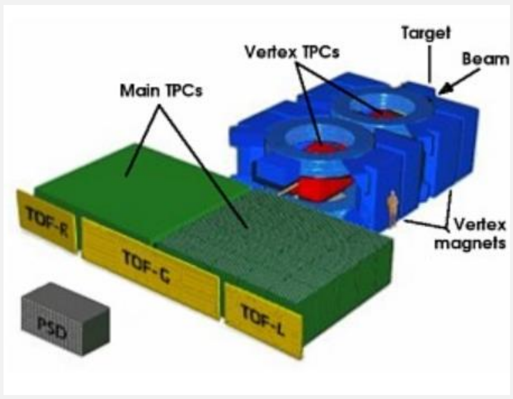


Figure 1: Tracking devices setup in NA61/SHINE experiment. The beam first passes two vertex TPCs and then two main TPCs.

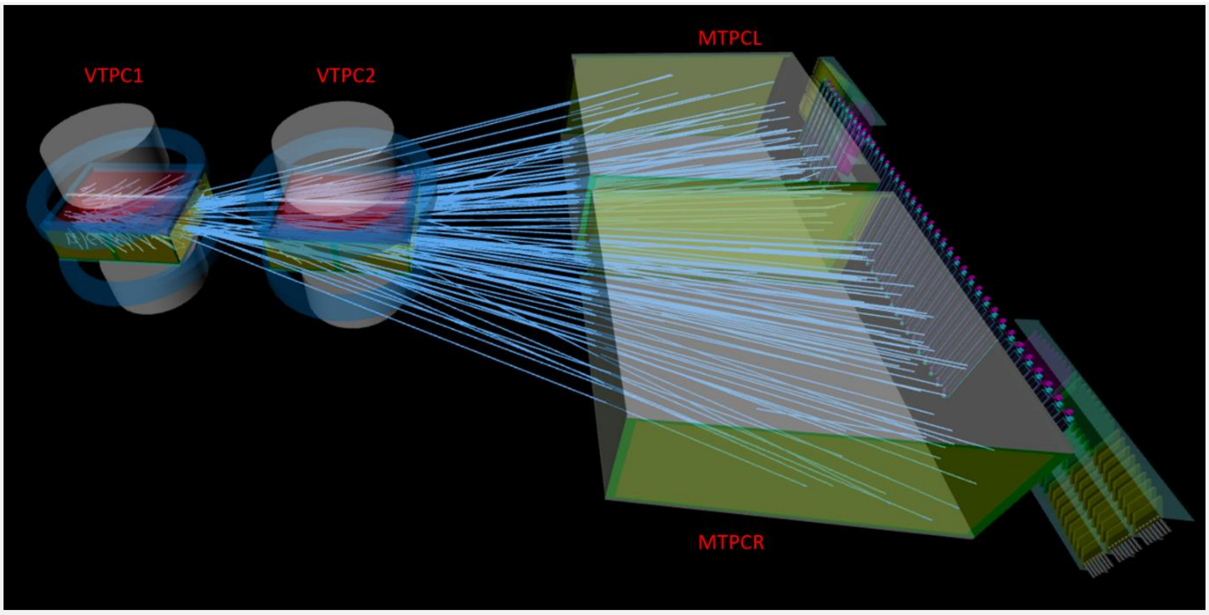


Figure 2: 3D Visualization of particle trajectories in NA61/SHINE experiment. The particles first pass two vertex TPCs and then two main TPCs. Credit: Michalski F, Palayda *(Michalski & Palayda).*

The Data Sets

Two data sets were available for my project. Data from 100 random events of one of the vertex TPCs (VTPC2) as well as from one of the main TPCs (MTPCL). One event consists of many thousands so-called clusters. A cluster consists of all charge deposition data points that belong to the same particle track. These tracks are found via a reconstruction algorithm.

For each event, the procedure was the following:

1) Take 100 signal cluster and 100 noisy clusters randomly
2) For each cluster find the pad ID and timestamp with max charge deposit and include the charge deposit information from +- 5 neighbouring pads, for + 10 and - 9 timestamps

The result is a two-dimensional picture for each cluster. For visualization, one cluster is shown in a three-dimensional Plot (Figure).

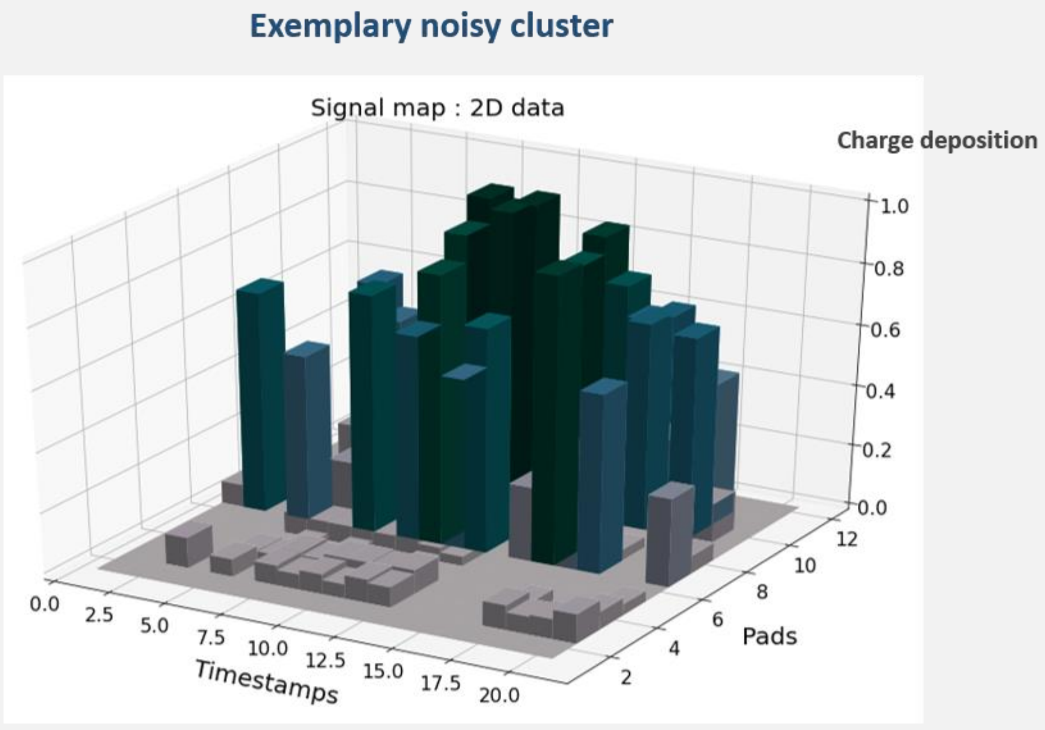


Figure 3: Exemplary cluster of charge deposition data over 20 time steps on 11 pads.

Data Classification

All clusters that belong to a track, found by the reconstruction algorithm, are labelled as signal, the rest as noise.

In sum, each data set consists of 10.000 signal clusters and 10.000 noisy clusters, each out of 100 random events. Therefore, a full data set consists of 20.000 labelled clusters. Where 10.000 are positive examples (signal) and 10.000 are negative examples (noise).

Construction of One-dimensional Data Sets

The original data sets have a shape of (20.000, 11, 20). Twenty thousand samples, each consisting of charge deposition data over 20 time steps from one pad and its ten nearest neighbours (11Px20T): five to the left and five to the right.

I constructed two sub data sets out of each. One that contains data from the one pad of a cluster, which received the maximum average charge deposition (the pad in the middle of a cluster). Consequently, this data set has a shape of (20.000, 1, 20) as it consists of 20.000 samples with data only from single pads over 20 time steps (1Px20T) similar to the one Manjunath trained his networks with but with 20 time steps included instead of 220.

Secondly, I constructed a data set, which includes data from all eleven pads but only within one time step (11Px1T), the middle time step. Consequently, these samples consist of eleven data points. The data set has a shape of (20.000, 11, 1).

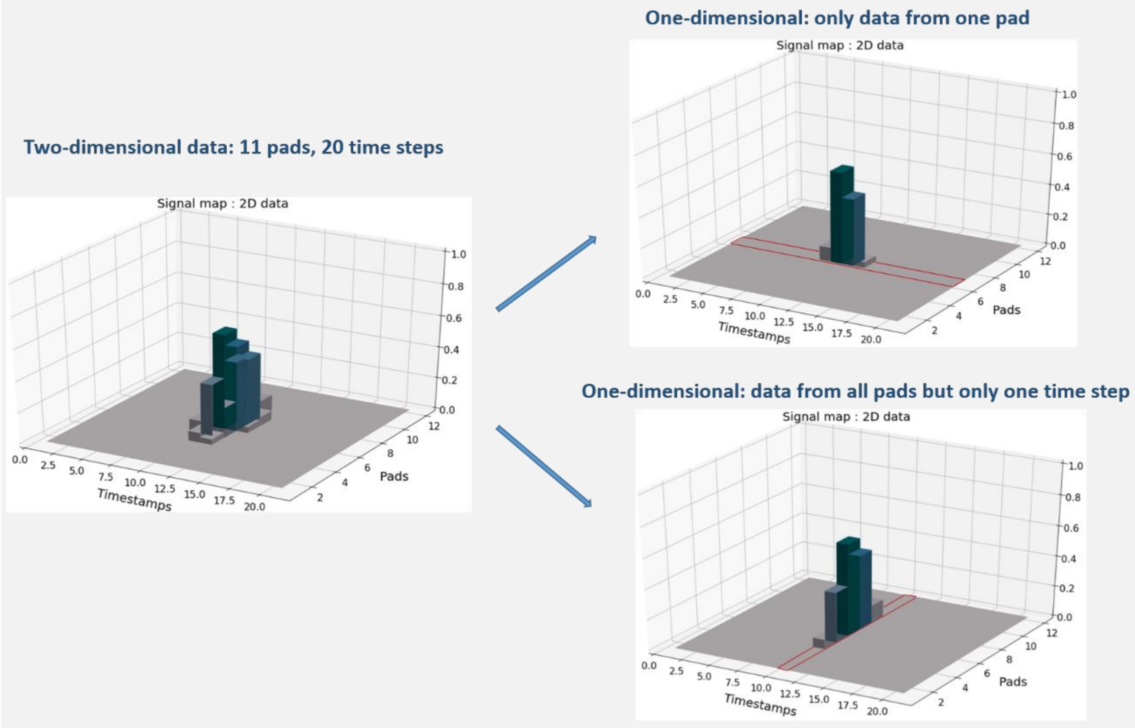


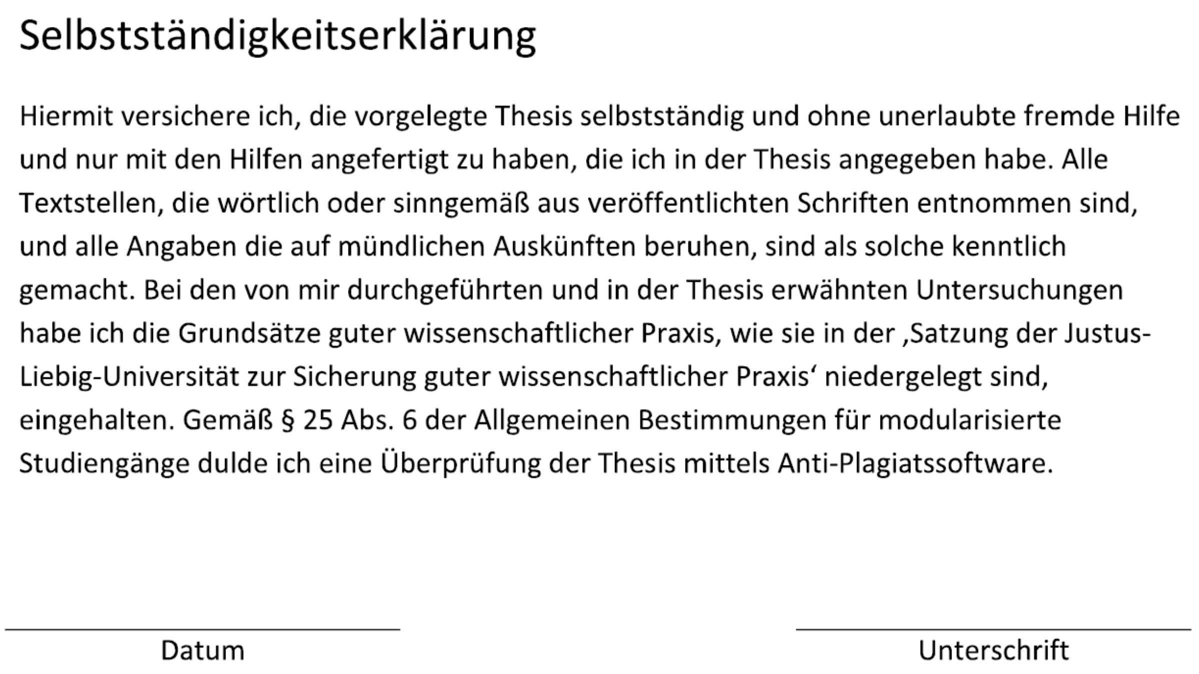
Figure 4: Cut out a row or column to prepare one-dimensional data sets consisting of slices of the original two-dimensional data

Acknowledgements

This thesis was written following my internship at the Frankfurt Institute for Advanced Studies to deepen the project I started there in spring 2020 and to further explore its theoretical background. It was particularly challenging at a time in the midst of a global pandemic, so I am especially grateful for all the people who accompanied me during this time. Therefore I would like to take this opportunity to thank all the people who have been particularly helpful and supportive to me over the last few months. I would like to thank PD Dr. Olena Linnyk and PD Dr. Sören Lange, who made this work possible in the first place. Manjunath Omana Kuttan, who besides Olena and Sören actively supported me, willingly discussed arising questions and helped me out with any sort of programming issues. Julia, Eli and William, who were always willing to proofread and help me with their English skills whenever I encountered difficulties. Not to forget, Patricia and Sophie, who whenever the situation allowed, would encourage and motivate me in the form of shared home office time.

Finally, I would like to thank both JLU and THM, friends and family, and everyone else involved who made the countless hours I spent to achieve the goal of a bachelor's degree in physics an unforgettable experience.

Declaration of Independence



Selbstständigkeitserklärung

Hiermit versichere ich, die vorgelegte Thesis selbstständig und ohne unerlaubte fremde Hilfe und nur mit den Hilfen angefertigt zu haben, die ich in der Thesis angegeben habe. Alle Textstellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen sind, und alle Angaben die auf mündlichen Auskünften beruhen, sind als solche kenntlich gemacht. Bei den von mir durchgeführten und in der Thesis erwähnten Untersuchungen habe ich die Grundsätze guter wissenschaftlicher Praxis, wie sie in der ‚Satzung der Justus-Liebig-Universität zur Sicherung guter wissenschaftlicher Praxis' niedergelegt sind, eingehalten. Gemäß § 25 Abs. 6 der Allgemeinen Bestimmungen für modularisierte Studiengänge dulde ich eine Überprüfung der Thesis mittels Anti-Plagiatssoftware.

Datum	Unterschrift