University of Applied Sciences Aachen
Campus Jülich

Faculty: Medical Engineering and Technomathematics
Course of Study: Scientific Programming

Autonomous Fault Detection Using Artificial Intelligence Applied to CLAS12 Drift Chamber Data

A Bachelor's Thesis by Christian Peters

Jülich, August 5, 2018

Declaration of Authorship

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references.

Christian Peters
Jülich, August 5, 2018

This thesis has been supervised by:
1. Advisor: Prof. Dr. Andreas Terstegge
2. Advisor: Günter Sterzenbach

Contents

1 Introduction

In order for physical experiments at particle accelerators and detectors to succeed, highly precise ways of measuring particle interactions are required. It is therefore crucial to recognize immediately when parts of the experimental setup show signs of deficiency, because these faulty components can lead to severe distortions in measurement accuracy.

At the CLAS12 particle detector, most of the measurements are performed within the CLAS12 drift chamber, a subsystem of the detector consisting of 24,192 wires, each designed to capture the presence of particles to reconstruct a particle track. Due to the extreme conditions within the drift chamber, it is common that single wires or collections thereof stop working properly during an experimental run. In order to still be able to perform accurate computations on the basis of these measurements, faulty components have to be detected during runtime and their effects have to be filtered out. This task requires huge amounts of data to be processed in real time, which makes it infeasible to solve this problem based on real time human interaction. An *autonomous* approach of fault detection is therefore required.

Luckily though, there is one field which has gained much popularity in recent years that happens to lend it self particularly well to the task of fault detection: *Artificial Intelligence*. Many powerful algorithms have originated from this domain, dominating the benchmarks of challenges such as large scale image classification, even reaching human like performance [Rus+14].

The goal of this thesis is to apply algorithms of artificial intelligence and machine learning, namely deep convolutional neural networks, to the problem of autonomous fault detection within the context of the CLAS12 drift chamber.

After describing the environment presented by the CLAS12 detector and the CLAS12 drift chamber in more detail, the fundamentals of deep learning will be outlined and the most essential algorithms will be derived. Upon this foundation, the central method utilized in the scope of this thesis, convolutional neural networks, will be analyzed to justify its applicability in the context of fault detection. The *deeplearning4j (DL4J)* framework will be used to implement the fault detection algorithm which is evaluated afterwards on real fault data.

2 The CLAS12 Drift Chamber

2.1 Overview

The CLAS12 Drift Chamber (DC) is a subsystem of the CLAS12 particle detector, located inside one of the halls of the Thomas Jefferson National Facility, Newport News Virginia, U.S.A. The detector is used at the core of many experiments, designed to gather information on particle interactions originating from an electron beam hitting a target inside of its center.

After interacting with the target, the resulting particles pass through the drift chamber subsystem that is designed to measure their momentum, which is done to aquire further insights into the underlying physical processes. This core responsibility of the drift chamber to register the results of particle interaction is the reason why it is deemed to be the most crucial component of the CLAS12 particle detector.

2.2 Layout

The drift chamber itself is composed of a hierarchical arrangement of single wires, where each wire is responsible for detecting the presence of particles by emiting signals of activation. It consists of 18 individual wire chambers, each made up of two superlayers. A superlayer is a collection of six simple layers, each consisting of 112 wires, which means that the whole system contains a total of $112 \cdot 6 \cdot 2 \cdot 18 = 24,192$ wires. A visual representation of the hierarchical structure of a single wire chamber can be found in Fig. 2.1.

In order to gather as much information as possible on the particle interactions, the 18 wire chambers are equally distributed across three regions within the drift chamber, the difference between the regions being their distance to the location of the electron beam hitting the target. Region one is located closest to the point of interest, region three is the furthest away.

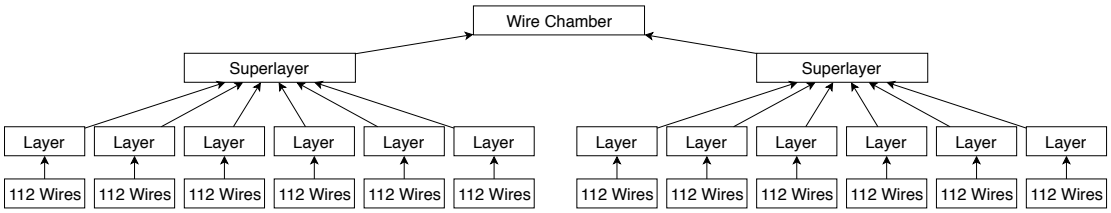

Figure 2.1: The hierarchical structure of a single wire chamber.

2.3 Drift Chamber Faults

When conducting experiments, every part of the drift chamber operates under extreme conditions, facing huge amounts of radiation. Therefore, it is very common that some components get damaged and stop working during the process. It is crucial to spot these faults in time because the experimental results strongly rely on the accuracy of the measurements.

Usually, faults occur within the scope of a single superlayer. To visualize them, one can create a heatmap out of the wires' activations that are accumulated during an experimental run, lighter areas displaying higher activation than darker ones (see Fig. 2.2 as an example). There are six different fault types that can be distinguished:

Dead Wire: This fault occurs when a single wire of a layer stops working, resulting in the wire displaying much lower activations than its surroundings. An example of a dead wire is given in Fig. 2.2.

Dead Pin: A pin is composed of several successive wires within a layer. See Fig. 2.3 for an example of a dead pin.

Dead Connector: This component connects several wires across a superlayer. If damaged, it results in a pattern of activations that is illustrated in Fig. 2.4.

Dead Fuse: A fuse is a collection of multiple connectors, therefore the resulting fault pattern appears wider than that of a dead connector as shown in Fig. 2.5.

Dead Channel: Multiple fuses can be connected within a channel. This is the biggest fault that can appear within a superlayer. A visual example of a dead channel fault is illustrated in Fig. 2.6.

Hot Wire: This anomaly is quite different from the other faults. It occurs when a single wire suddenly starts emitting huge amounts of activation, silhouetting against the wires in its immediate surroundings. An example of a hot wire can be seen in Fig. 2.7.

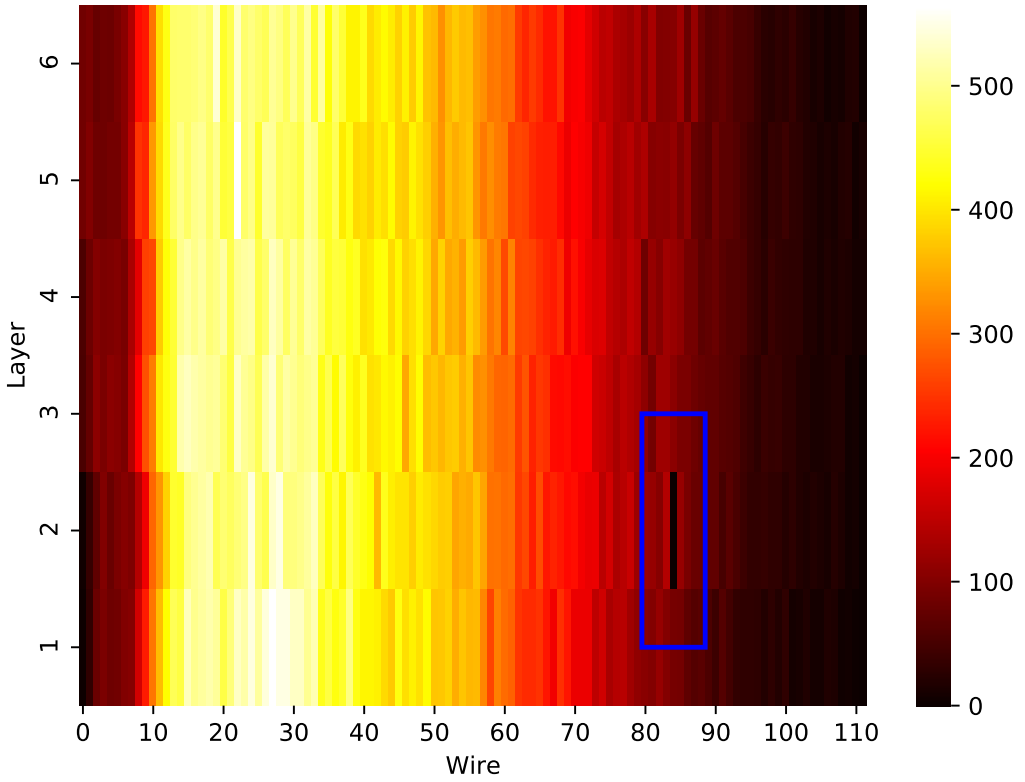

Figure 2.2: A single dead wire marked by the blue rectangle.

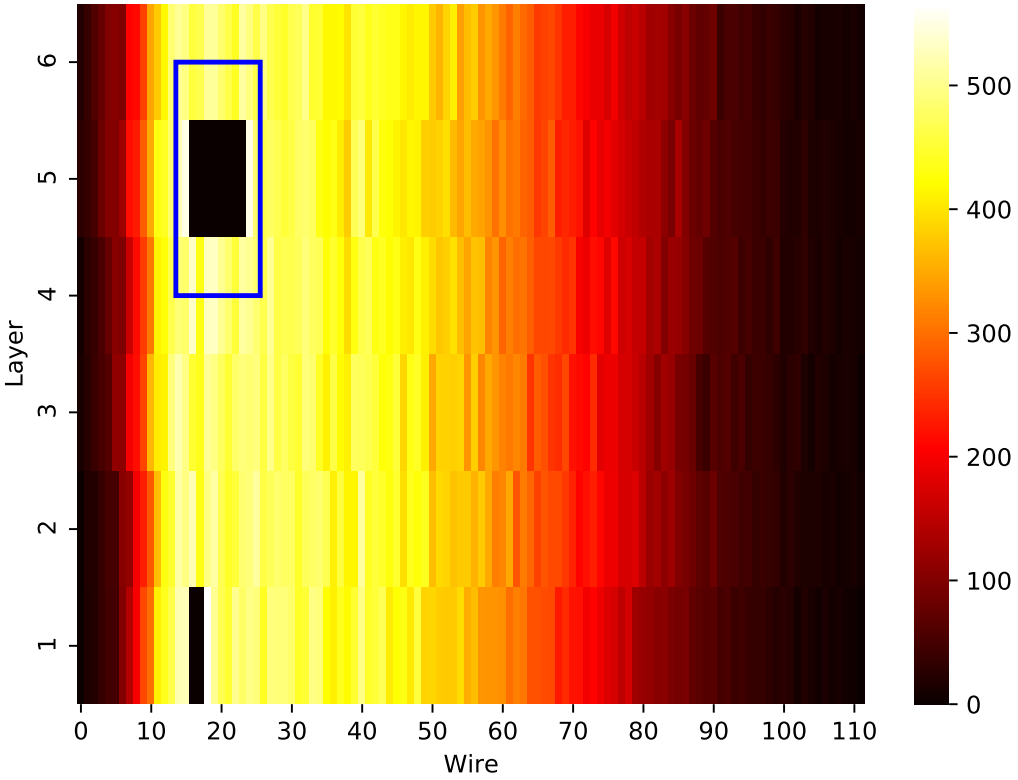

Figure 2.3: A dead pin spanning eight wires, marked by the blue rectangle. Also notice that there are two dead wires right next to each other in layer 1.

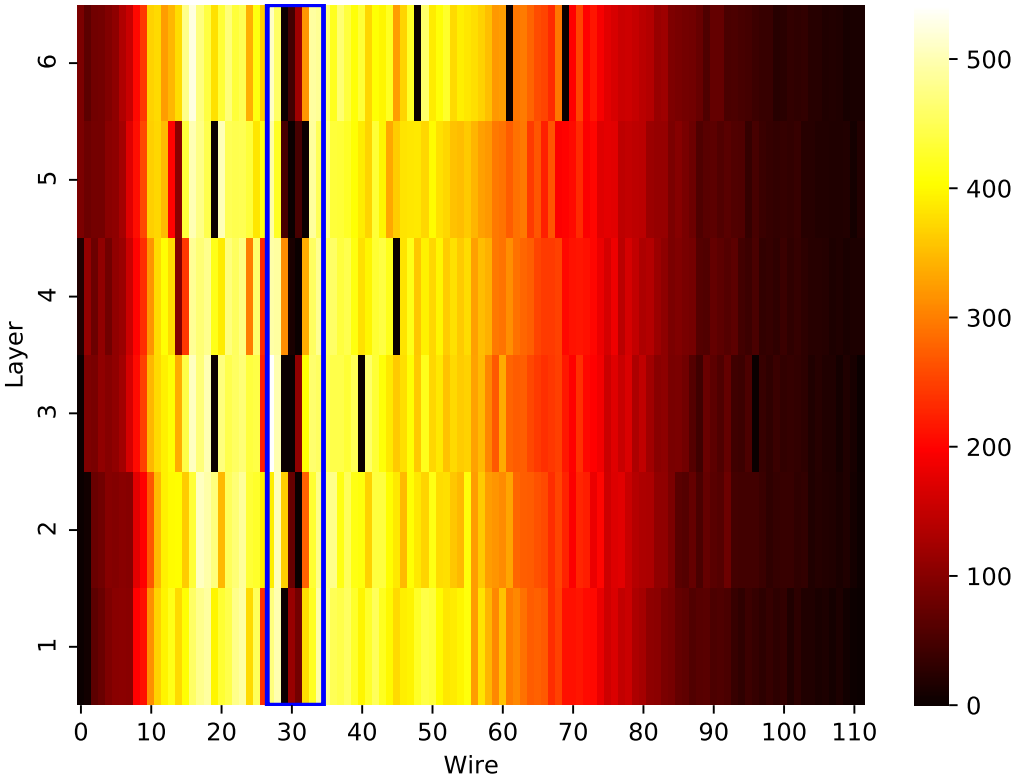

Figure 2.4: A dead connector surrounded by some dead wires.

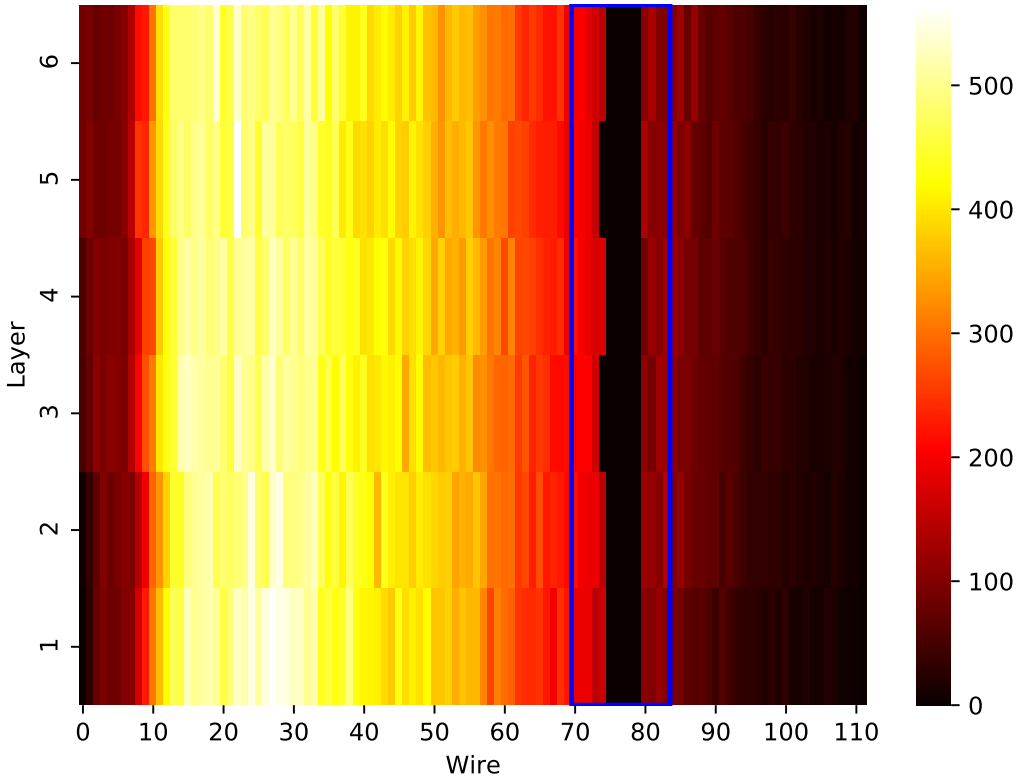

Figure 2.5: An example of a dead fuse.

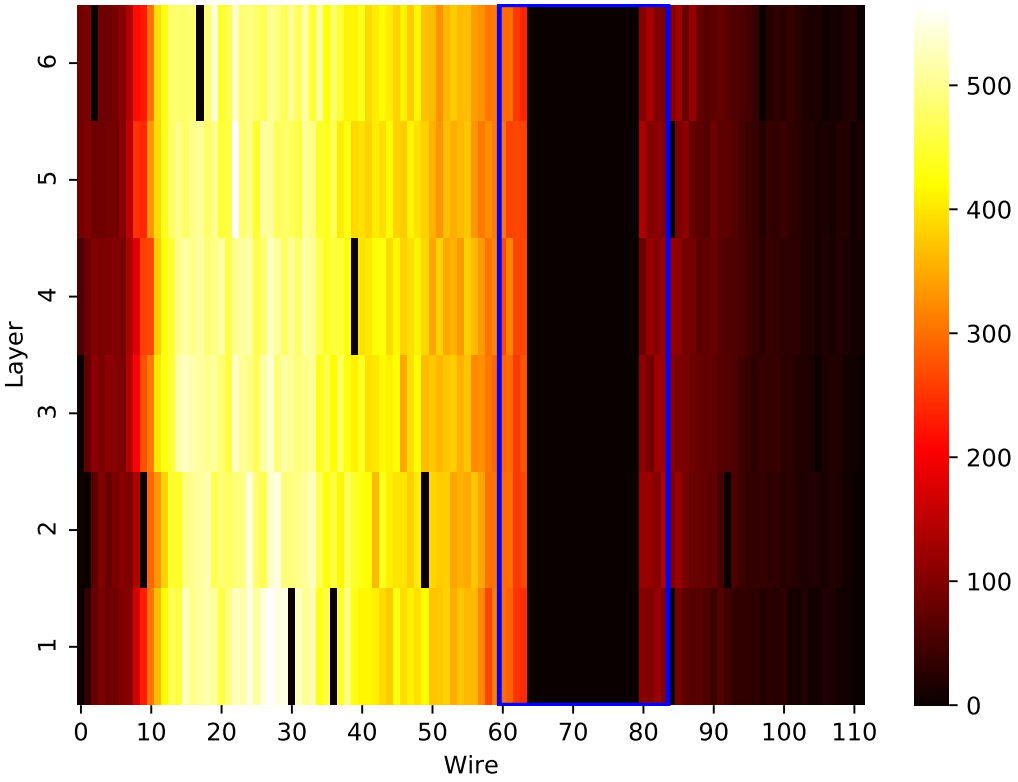

Figure 2.6: A dead channel with some dead wires.

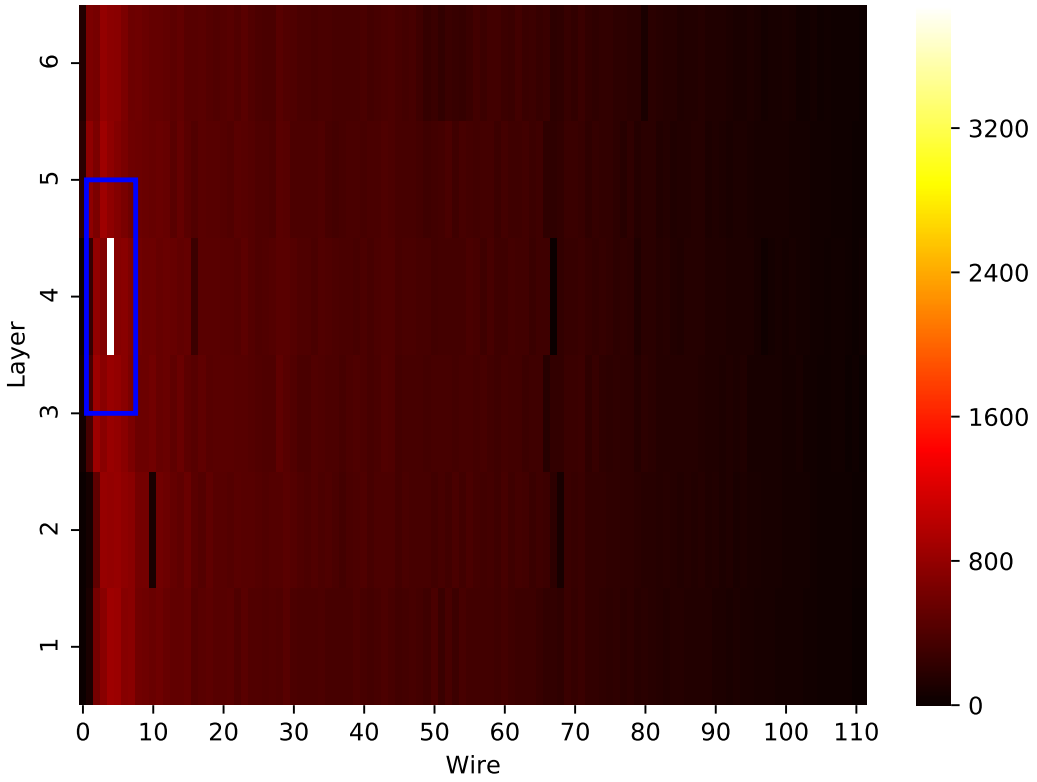

Figure 2.7: A hot wire displaying much more activation than its surrounding wires.

3 Deep Learning Fundamentals

To lay out a foundation upon which to build in the process of creating the fault detection system, the essential principles of deep learning will be illustrated within the context of this chapter. The central concept that most deep learning algorithms have originated from is the *artificial neural network*, which is crucial to understand in order to to grasp the core principles of deep learning.

3.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are a class of machine learning algorithms that are loosely inspired by the structure of biological nervous systems [Hay08]. To be precise, each ANN consists of a collection of artificial neurons that are connected with each other, enabling them to exchange signals. The structure of an ANN can be described by a directed graph, i.e. a collection of nodes as well as directed edges. The nodes of the graph represent the neurons, the edges denote their connections. A common way to arrange artificial neurons within a network is to organize them in layers as depicted in Fig. 3.1.

When an artificial neuron receives a signal from its incoming connections, it may elect to become active based on the input it collects, see section 3.1.1. In this state, it also influences all neurons it has an outgoing connection to by passing a signal along their channel. These other neurons in turn may also elect to become active, this way a signal can propagate through the network along the connecting edges as seen in Fig. 3.1.

Usually, each ANN consists of at least one layer of neurons that is responsible for receiving signals from the environment, this special type of layer is called an *input layer*. When the neurons in this layer receive a signal, they propagate it to their connected neighbors in the next layer. This process repeats until the *output layer* is reached. The neurons in the output layer represent the result of the whole network. Each layer in between the input and the output layer is called a *hidden layer* because there is no direct communication between the neurons in such a layer and the environment. Networks that satisfy this basic architectural model, where each layer is fully connected with its

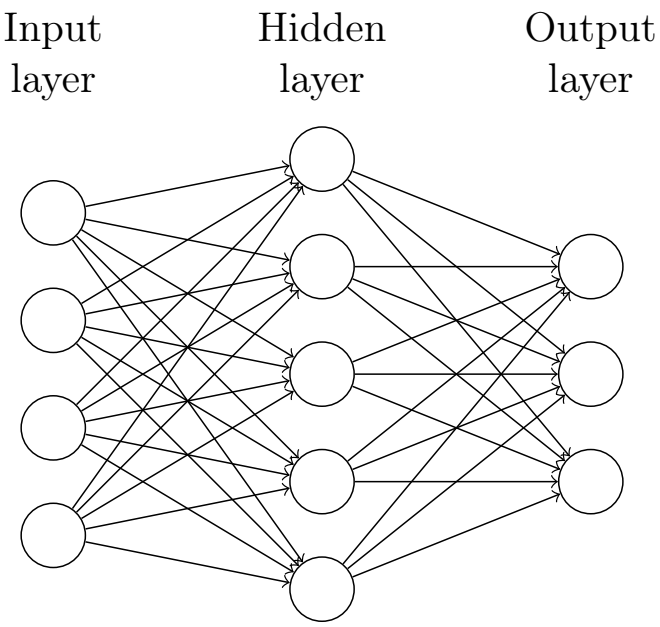

Figure 3.1: A common ANN-structure, consisting of three layers of neurons represented by a directed graph.

following layer and signals only flow in one direction without cycles, are called *fully connected feedforward networks*.

The goal behind artificial neural networks is to convert an input signal into a meaningful output by feeding it through the network. If the network is able to detect relevant features or patterns in the input signal, it can be used to perform tasks such as classification or regression, i.e. approximate discrete or continuous functions. In order for this to be possible, some kind of learning has to take place which enables the network to capture the the patterns present in the data it is confronted with. We will take a further look at these aspects as well as the mathematical model of a neural network in the following sections.

3.1.1 Modeling Artificial Neurons

To fully understand how each neuron processes the signals it receives, it is necessary to develop a mathematical model that describes all the operations taking place. The following descriptions are partially based on the explanations that are provided in [Hay08].[1] As shown in Fig. 3.2, the neural model basically consists of three components:

[1] See chapter I.3: *Models of a Neuron* for more details.

1. **A set of weighted inputs:** Each connection that is leading into a neuron has a weight w_{kj} associated with it where k denotes the neuron in question and j denotes the index of the neuron that delivers its input to the current neuron k. The signal that passes the connection is multiplied by the related weight of that connection before arriving at the next component. There might arise the question why the indexing of a weight from neuron j to neuron k is w_{kj} and *not* w_{jk}. This is the case because in practical implementations of neural networks, the weights are usually stored in matrices where each row corresponds to a neuron k and each column corresponds to an input j which allows for much faster computations by heavily utilizing matrix-multiplication.

2. **A summation unit:** This component adds up all the weighted signals that arrive at the neuron as well as a constant bias value b_k that is independent of the inputs. The reason for adding the bias term is explained in section 3.1.3.

3. **An activation function:** The activation function $\phi(\cdot)$ applies a transformation to the output of the summation unit that is usually non-linear. The value of the activation function is the output of the neuron which will travel further through the network alongside the corresponding connections. In section 3.1.2, a more detailed explanation of activation functions as well as some commonly used examples will be provided.

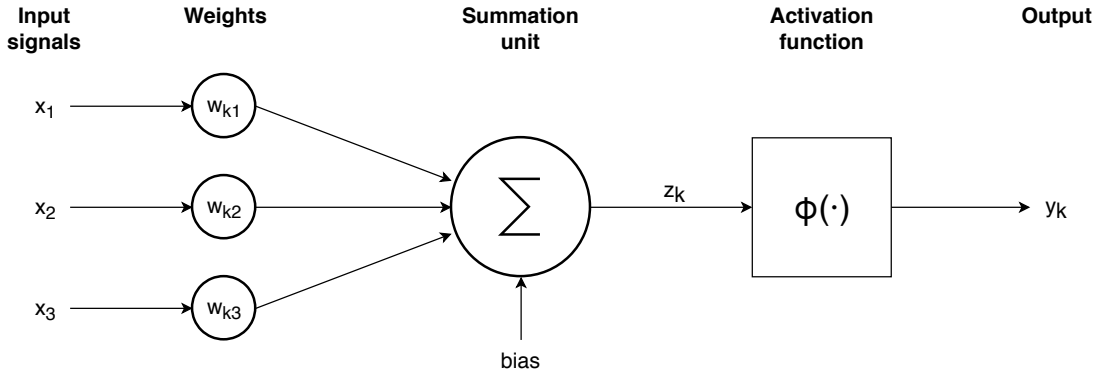

Figure 3.2: The components of the neural model. This neuron k receives three input signals that are first multiplied by the associated weights, summed up including a bias and then fed into an activation function that will determine the ouput signal.

Transforming this model into mathematical equations, the output of the summation unit of a particular neuron k with n input signals x_j can be described by the following formula:

$$z_k = \sum_{j=1}^{n} x_j \cdot w_{kj} + b_k \tag{3.1}$$

where b_k denotes the bias term of neuron k and z_k describes the result of the summation unit.

As a consequence, the output signal y_k of neuron k can be computed by applying the activation function $\phi(\cdot)$ to the output of the summation unit which can be described by the following expression:

$$y_k = \phi(z_k) \tag{3.2}$$

As we shall see in section 3.2.3, the entire knowledge of the network is stored within the weights as well as the biases. Adjusting this configuration in order to better match the data that the network receives is the main goal of training, see section 3.2.3.

3.1.2 Activation Functions

The basic task of an activation function is to determine the level of activity that a neuron emits based on the input it receives. Because the incoming signals are first weighted and summed up by the summation unit, they arrive at the activation function as a single value z_k. Since the output y_k of neuron k is also a scalar, each activation function can be described as $\phi : \mathbb{R} \to \mathbb{R}$. A very important property of an activation function is *nonlinearity*. This is due to the fact that chaining together multiple linear functions, as would be the case if each artificial neuron had a linear activation function, collapses into just a single linear transformation, which would make it impossible for the network to learn any concepts beyond simple linear relationships. In the following paragraphs, an overview of the most popular activation functions will be presented that is based on the descriptions found in [PG17].[2]

The Sigmoid Function This activation function transforms an input z into a range between 0 and 1 based on the following equation:

$$\phi(z) = \frac{1}{1 + e^{-\theta \cdot z}} \tag{3.3}$$

[2]See section *Activation Functions* in chapter two.

The θ parameter is used to adjust the sensitivity of the sigmoid function with respect to its input signal. High values of θ lead to steep slopes around $z = 0$ while smaller values will lead to smoother slopes. An illustration of this relationship is presented in Fig. 3.3. One important reason why the sigmoid function was often used at the time

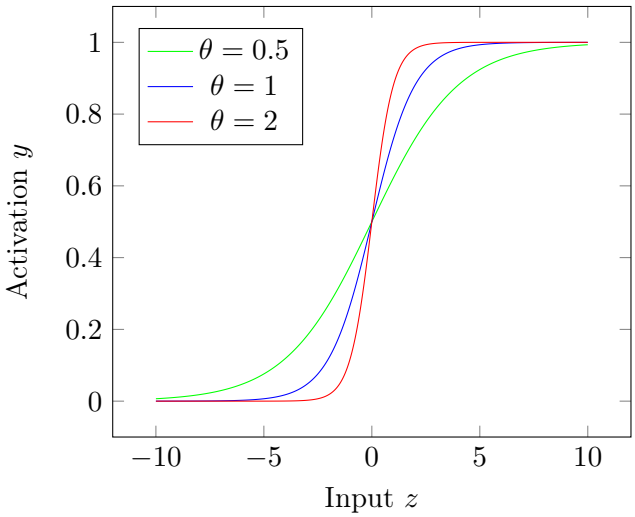

Figure 3.3: The sigmoid activation function plotted for different values of θ.

neural networks were first developed, is that it reduces the impact of outliers in the data without removing them. When the input of a neuron is large, it is reduced to a number near one, when it is very negative, the activation evaluates to a number near zero. This behaviour adds to the overall robustness of the network.

In the early days of neural networks, people were also seeking for biological inspirations when constructing ANNs. The graph of the sigmoid function can also be interpreted as the firing rate of a biological neuron that saturates for big inputs, which contributed to its popularity in the past. However, caution is advised when putting too much weight onto these interpretations, since real neurons are much more complex in practice than simple mathematical equations.

The Rectified Linear Unit (ReLU) Because it is not always desirable to reduce large signals to a smaller scale, the ReLU function will only replace negative values with zero and leave positive values untouched. This behaviour can be modeled by the following expression:

$$\phi(z) = \max(0, z) \tag{3.4}$$

When building deep neural networks, one of the problems that sometimes arise is that a signal will fade out when propagating through many hidden layers. This issue is remedied to some degree by using the ReLU function because big signals are not cut down. The fact that all negative values are set to zero when the ReLU function is used leads to sparsity among the neuron activations which promotes simpler and possibly richer representations. This analogy can also be found in real biological neurons, which makes it an interesting thing to note that ReLUs are actually more biologically inspired than the sigmoid function [GBB11]. Another benefit of the ReLU function is that its derivative is either 1 or 0, which makes it very simple to compute. This will turn out to be important when looking into the training of neural networks. Because of all these advantages, ReLUs are one of the state-of-the-art activation functions in deep neural networks. A plot of the ReLU function is presented in Fig. 3.4.

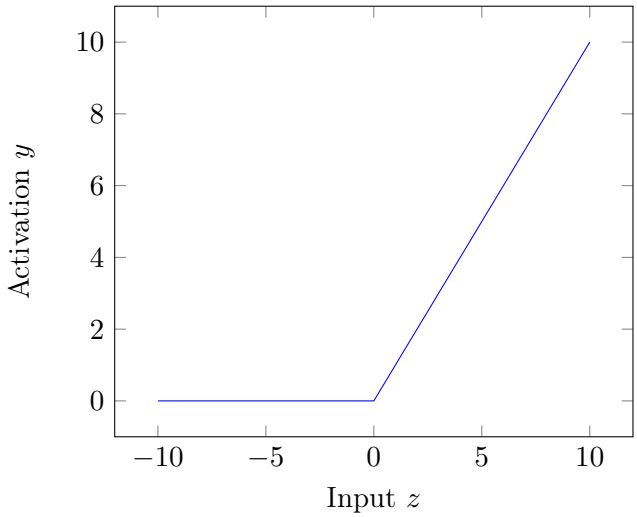

Figure 3.4: The ReLU activation function.

The Softmax Activation Function This activation function is usually applied to the output neurons of a network. When a neural network is used to perform classification tasks, each output neuron is commonly associated with a specific class. In classification tasks, it is highly desirable to assign a probability to each class that represents how likely it is that the input data belongs to that class. The softmax activation function is used to achieve this by setting up the output neurons to represent a probability distribution over all possible classes. In an output layer consisting of n output neurons, the softmax function for each neuron i of that layer can be described by the following equation, where

z_i denotes the summation units' output of the i^{th} neuron:

$$\phi(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \tag{3.5}$$

The softmax activation function represents, loosely speaking, the percentage of the current neurons activation with respect to the compound activation of all neurons in the layer. The inputs z_i of the ouput layer can thus be interpreted as the *unnormalized log-probabilities* for each class.

There might arise the question why each input z_i is first fed into the exponential function e^x before translating the activations into probabilities. This is done to further amplify the strongest signals and attenuate the weaker ones which results in more clear-cut values which makes learning for the network easier, as shown in the following example:

Imagine the z_i inputs of the output layer are given by the following vector: $(2, 4, 2, 1)^T$. If we just normalize these values to obtain a probability for each neuron, we get $(0.22, 0.44, 0.22, 0.11)^T$. Using the exponential function first, we roughly get $(0.1, 0.76, 0.1, 0.04)^T$ which amplifies the most likely outcomes and attenuates the less likely ones.

3.1.3 The Role of the Bias Value

There still remains the question why in each artificial neuron there is a bias value b_k added to the weighted sum of the inputs. The reason for this is related to the activation function: The bias term acts like a parameter that determines how to shift the activation function along the x-axis. We already know from Eq. 3.1 that for a neuron k with n inputs the toal input signal z_k adds up to:

$$z_k = \sum_{j=1}^{n} x_j \cdot w_{kj} + b_k$$

Let us denote the weighted sum of the input signals as a separate value $a_k = \sum_{j=1}^{n} x_j \cdot w_{kj}$ that describes the raw input of the neuron. This means that $z_k = a_k + b_k$ and using the sigmoid function (see section 3.1.2) as an example to demonstrate the effects of the bias value, we can slightly rewrite it as

$$\phi(a_k) = \frac{1}{1 + e^{-(a_k + b_k)}}$$

also setting $\theta = 1$ for demonstration purposes.

Plotting the activation function for different values of b_k immediately reveals the effect

of the bias value as a shift-parameter, which can be seen in Fig. 3.5.

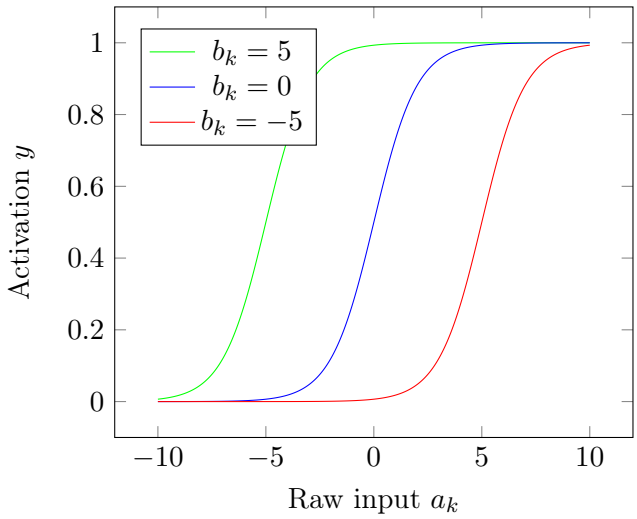

Figure 3.5: The sigmoid activation function plotted for different bias values.

It follows that the bias term acts like a threshold that has to be overcome in order for the neuron to become active. Positive bias values lead to activity even when the raw input a_k is still negative and negative bias values require bigger input signals in order for the neuron to fire.

3.2 Neural Networks as Classifiers

After establishing a mathematical model that helps us to describe a neural network, there is still one problem to be solved: How to train the network to be able to successfully perform tasks such as classification? In order to figure this out, we will first take a look at classification tasks in general and then explore how to set up and train a neural network to perform classification.

3.2.1 Classification

The basis of a classification task is usually formed by a dataset that consists of features as well as labels. The goal of the classification algorithm is to predict the label of an instance of the dataset by only looking at its features. In order to achieve this, the classifier first has to build a model based on a training dataset. This procedure is called *training* (see section 3.2.3). In the next step, called *testing*, the classifier is presented with some new examples that it did not see during training. The classifier is tested on

these new examples to estimate its performance and to see if it was able to learn any concepts from the data, i.e. to *generalize*. Because the classifier infers a function from labeled data, classification is an example of a broader domain called *supervised learning*.

3.2.1.1 Evaluating a Classifier

In order to find out how well a classifier generalizes after training, the results of the testing phase can be entered into a *confusion matrix* that is structured as shown in Table 3.1.

	Class Positive (Predicted)	**Class Negative (Predicted)**
Class Positive (Actual)	True Positives (TP)	False Negatives (FN)
Class Negative (Actual)	False Positives (FP)	True Negatives (TN)

Table 3.1: The structure of a confusion matrix for a classification task with two classes "Positive" and "Negative".

Each entry in this matrix describes how often the classifier was presented with an example of the row-class during testing and predicted that the example belongs to the column-class. For instance, the value FP (False Positives) expresses how often the classifier saw an example of class "Negative" and predicted that it belongs to class "Positive". The same principle also applies to the other cells in the confusion matrix. It should be noted that this concept can be extended to classification tasks with more than two classes as well by simply adding new rows and columns for each new class. The resulting measurements of true positives, true negatives, false positives and false negatives can be used to compute the following evaluation metrics:[3]

Accuracy: The accuracy metric determines the percentage of examples in the testing set that the classifier predicted correctly. It can be denoted by the following equation:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.6}$$

This measurement works well if there is roughly an equal amount of examples for each class. However, if one of the classes makes up most of the examples, the classifier can

[3] A collection of these metrics can also be found in [PG17], see chapter *Evaluating Models*.

reach a high degree of accuracy by just predicting the label of the dominant class every single time. This impairs the significance of this metric when imbalances among the classes are present.

Precision: The precision score shows the percentage of examples that were correctly classified as positive among all examples that the classifier labeled positive:

$$Precision = \frac{TP}{TP + FP} \tag{3.7}$$

This metric can also be interpreted as an estimate of the conditional probability that the classifier is right given that it predicted a positive class:

$$Precision = P(\text{Classifier is right}|\text{Classifier predicted POSITIVE})$$

Recall: This measurement remedies the imbalance issues of the accuracy metric by determining the percentage of correctly classified examples for each separate class. It can be denoted by the following expression:

$$Recall = \frac{TP}{TP + FN} \tag{3.8}$$

The recall score can also be interpreted as an estimate of the conditional probability that the classifier is right given a specific class:

$$Recall = P(\text{Classifier is right}|\text{Class is POSITIVE})$$

F1 Score: This metric combines precision and recall to calculate their so called *harmonic mean*. It is often used when evaluating classification models, thus its equation is also displayed here:

$$\text{F1 Score} = \frac{2 * Precision * Recall}{Precision + Recall} \tag{3.9}$$

It should be noted that all these measurements can also be extended to classification tasks with more than two classes. This is done by first computing the metrics for each class separately and then taking the average of these values to estimate a global score.

3.2.2 Network Architecture for Classification

The architecture of the neural network that will be used to perform the classification task is highly dependent on the structure of the dataset. Remembering that a neural network consists of an *input layer* as well as *hidden layers* and an *output layer*, the question is how to assemble these layers to fit the task well.

The first consideration is that each feature in the dataset will correspond to an input signal that is fed into the network, thus the amount of neurons in the input layer must be equal to the amount of features in the dataset. Because the only responsibility of the input units is to receive a signal from the environment and pass it on to the next layer, these neurons don't have a special activation function that transforms the input. The activation of these neurons is simply the identity of the incoming signal. In order to avoid features on larger numerical scales dominate features with smaller values, the data is usually normalized and scaled to equal ranges first before being fed into the network.

The number of hidden layers that are inserted between the input and the output layer highly depends on the complexity of the task. As the number of hidden neurons grows, there are more parameters (weights and biases) left to be adjusted during training which means more capacity for the network to learn. However, the danger lays in the fact that if there are too many hidden neurons and hidden layers, the network will just use this capacity to memorize the training examples and not extract general concepts from them which will lead to low accuracy on unseen examples. This problem can also be described by the more general term *overfitting*. On the contrary, if there are not enough hidden neurons, the network won't be able to capture all concepts that are present in the data which will lead to an opposite effect: *underfitting*. Both overfitting and underfitting harm the ability of the network to generalize well beyond the training data. In practice however, it is recommended to prefer many hidden neurons and hidden layers over an insufficient amount, because there are other techniques such as *regularization* that punish increasing model complexity to prevent overfitting [Ben12]. The first choice of activation function that is used in the hidden layers is usually the ReLU (see section 3.1.2) because of its various beneficial properties.

As already mentioned in section 3.1.2 about the softmax activation function, it is highly useful if the network is able to not only predict the correct label but also to indicate how certain it is about it. This is why the output layer will consist of as many neurons as there are classes in the dataset which will enable us to use the softmax function on this layer to retrieve a set of probabilities for each example that is presented to the network. The neuron that shows the highest degree of activity, i.e. assigns the highest probability, determines the label the network will assign to the example.

3.2.3 Training the Network

In order to be able to improve the quality of the networks' predictions, i.e. training the network, we first have to introduce a way of measuring the performance of the network with respect to the training examples it is presented with.

Let x be an example input from the training dataset and $y'(x)$ be the desired output of the network that corresponds to the example. Both x and $y'(x)$ are vectors. The element x_i represents the input signal of the i^{th} input neuron and the element $y'_j(x)$ represents the desired activation of the j^{th} output neuron. To measure how close the actual output $y(x)$ of the network is to the desired output $y'(x)$, we can use the *sum of the squared errors*:

$$L_x = \sum_{i=1}^{n} (y_i(x) - y'_i(x))^2 = \left\| y(x) - y'(x) \right\|^2 \tag{3.10}$$

where n is the number of output neurons and L_x resembles the *loss of the network* for a single example x.

The *average total loss* of the network over all examples in the dataset (the total number of examples will be denoted by N) can be computed by averaging the losses of every single example:

$$L = \frac{1}{N} \cdot \sum_{x} L_x \tag{3.11}$$

We can also express this value in terms of the current configuration of the neural network that is represented by the set of weights w and the set of biases b that is currently used as the *loss function* $L(w, b)$. Now being able to measure the training performance of the network with respect to its configuration by calculating the average loss $L(w, b)$, we can define the training problem as follows:

$$\textit{Find a set of weights } w \textit{ and biases } b \textit{ such that } L(w, b) \rightarrow \textit{min}$$

This implies that training the network is an optimization problem where the weights and biases of the network are adjusted to find the minimum of the loss function $L(w, b)$.

The most common approach to solve the optimization problem is a technique called *gradient descent*. In each step of this procedure, the gradient $\nabla L(w, b)$ of the loss function L with respect to the weights w as well as the biases b is computed. This is done because the gradient always points in the direction of the steepest ascent of a function. In order to minimize L, one can simply take tiny successive steps in the direction of the *negative* gradient to arrive at a local minimum of L resulting in the following algorithm describing

how to adjust the weights w and biases b in each step t:

$$(w,b)_{t+1}^{T} = (w,b)_{t}^{T} - \alpha \cdot \nabla L(w,b) \tag{3.12}$$

The α parameter in this equation describes the size of the steps that are taken in the direction of the negative gradient and is also called the *learning rate* of the network. Choosing a reasonable value for α is essential for a successful training phase. If the learning rate is too big, the steps taken will also be too big resulting in skipping and not finding the minimum. Too small values of α will lead to slow convergence.

If the surface of L is convex, i.e. there is only one global minimum, the algorithm is guaranteed to converge for a sufficiently small learning rate. In practical application however, this property is usually not present due to the complexity of L. Despite of this circumstance, gradient descent usually still works well and converges to a local minimum of L that is usually sufficient for the network to solve the classification task.

There still remains the question how to compute the gradient $\nabla L(w,b)$ in each step of gradient descent. The answer to this is a procedure called *backpropagation* [RHW86].

3.2.3.1 The Backpropagation Algorithm

In order to derive the backpropagation algorithm that enables us to compute the gradient of the loss function, a little expansion of the current notation is necessary. The output of the k^{th} neuron of layer l in the network will now be denoted by $y_k^{(l)}$. This is done to indicate in which layer the described neuron resides. Likewise, the input z, bias b and weights w of neuron k in layer l will also receive a superscript denoting the current layer. Using n_l to describe how many neurons there are in layer l, we can slightly rewrite the equations 3.1 and 3.2 that describe the activation function input z_k, which is the output of the summation unit, and the output y_k of a neuron k like this:

$$z_k^{(l)} = \sum_{j=1}^{n_{l-1}} w_{kj}^{(l)} \cdot y_j^{(l-1)} + b_k^{(l)} \tag{3.13}$$

$$y_k^{(l)} = \phi(z_k^{(l)}) \tag{3.14}$$

Because the total loss of the network is just the average of the losses for each single example (see Eq. 3.11), the gradient of L can be computed like this:

$$\nabla L(w,b) = \nabla \left(\frac{1}{N} \cdot \sum_x L_x(w,b) \right) = \frac{1}{N} \cdot \sum_x \nabla L_x(w,b) \tag{3.15}$$

This means that computing the gradient of the total loss L is the same as computing the gradients of the losses for every single training example x and then taking the average.

The next step is to find a way to compute the partial derivatives that make up the components of the gradient. What this means is to find out how sensitive the loss function reacts to changes in a single weight $w_{kj}^{(l)}$ or a single bias $b_k^{(l)}$. Because all the weights as well as the bias of a neuron are combined with the inputs in its summation unit, it is helpful to take an intermediate step: Rather than computing the partial derivatives directly, it makes sense to think about how changes in the summed input $z_k^{(l)}$ of a particular neuron in the network impact the loss L_x. This sensitivity of the loss function with respect to the summed input of a particular neuron will be dentoted by the following equation:

$$\delta_k^{(l)} = \frac{\partial L_x}{\partial z_k^{(l)}} \tag{3.16}$$

where $\delta_k^{(l)}$ describes the sensitivity of the loss function with respect to changes in the summed input of neuron k in layer l.

Utilizing the *chain rule* of calculus, one can now write the partial derivatives of the loss function with respect to the weights as well as the biases like this:

$$\frac{\partial L_x}{\partial w_{kj}^{(l)}} = \frac{\partial L_x}{\partial z_k^{(l)}} \cdot \frac{\partial z_k^{(l)}}{\partial w_{kj}^{(l)}} = \delta_k^{(l)} \cdot \frac{\partial z_k^{(l)}}{\partial w_{kj}^{(l)}} \tag{3.17}$$

$$\frac{\partial L_x}{\partial b_k^{(l)}} = \frac{\partial L_x}{\partial z_k^{(l)}} \cdot \frac{\partial z_k^{(l)}}{\partial b_k^{(l)}} = \delta_k^{(l)} \cdot \frac{\partial z_k^{(l)}}{\partial b_k^{(l)}} \tag{3.18}$$

Computing the terms $\frac{\partial z_k^{(l)}}{\partial w_{kj}^{(l)}}$ and $\frac{\partial z_k^{(l)}}{\partial b_k^{(l)}}$ is fairly straightforward:

$$\frac{\partial z_k^{(l)}}{\partial w_{kj}^{(l)}} = \frac{\partial}{\partial w_{kj}^{(l)}} \sum_{i=1}^{n_{l-1}} w_{ki}^{(l)} \cdot y_i^{(l-1)} + b_k^{(l)} = y_j^{(l-1)} \tag{3.19}$$

$$\frac{\partial z_k^{(l)}}{\partial b_k^{(l)}} = \frac{\partial}{\partial b_k^{(l)}} \sum_{i=1}^{n_{l-1}} w_{ki}^{(l)} \cdot y_i^{(l-1)} + b_k^{(l)} = 1 \tag{3.20}$$

Now the only component that is left to be calculated is the sensitivity of the loss with respect to the summed input of each neuron, $\delta_k^{(l)}$. In order to compute this value, two cases have to be distinguished: First, if l is the output layer, $\delta_k^{(l)}$ will only influence the loss through one single neuron k. Keeping this in mind, calculating $\delta_k^{(l)}$ for the output layer goes as follows:

$$
\begin{aligned}
\delta_k^{(l)} &= \frac{\partial L_x}{\partial z_k^{(l)}} \overset{\textit{chain rule}}{=} \frac{\partial L_x}{\partial y_k^{(l)}} \cdot \frac{\partial y_k^{(l)}}{\partial z_k^{(l)}} \\
&= \left(\frac{\partial}{\partial y_k^{(l)}} \sum_{i=1}^{n} (y_i^{(l)}(x) - y_i'(x))^2 \right) \cdot \frac{\partial}{\partial z_k^{(l)}} \phi(z_k^{(l)}) \\
&= 2 \cdot (y_k^{(l)} - y_k'(x)) \cdot \phi'(z_k^{(l)})
\end{aligned}
\tag{3.21}
$$

The second case is l being a hidden layer. In this scenario, $\delta_k^{(l)}$ will influence the output of the loss function through all the neurons in layer $l+1$. Taking this into consideration, $\delta_k^{(l)}$ for each hidden neuron can be computed like this:

$$
\begin{aligned}
\delta_k^{(l)} &= \frac{\partial L_x}{\partial z_k^{(l)}} \overset{\textit{chain rule}}{=} \frac{\partial L_x}{\partial y_k^{(l)}} \cdot \frac{\partial y_k^{(l)}}{\partial z_k^{(l)}} \\
&= \frac{\partial L_x}{\partial y_k^{(l)}} \cdot \phi'(z_k^{(l)}) \overset{\textit{infl. on next layer}}{=} \left(\sum_{i=1}^{n_{l+1}} \frac{\partial L_x}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial y_k^{(l)}} \right) \cdot \phi'(z_k^{(l)}) \\
&= \left(\sum_{i=1}^{n_{l+1}} \delta_i^{(l+1)} \cdot \frac{\partial z_i^{(l+1)}}{\partial y_k^{(l)}} \right) \cdot \phi'(z_k^{(l)}) \\
&= \left(\sum_{i=1}^{n_{l+1}} \delta_i^{(l+1)} \cdot \frac{\partial}{\partial y_k^{(l)}} \left(\sum_{j=1}^{n_l} w_{ij}^{(l+1)} \cdot y_j^{(l)} + b_i^{(l+1)} \right) \right) \cdot \phi'(z_k^{(l)}) \\
&= \left(\sum_{i=1}^{n_{l+1}} \delta_i^{(l+1)} \cdot w_{ik}^{(l+1)} \right) \cdot \phi'(z_k^{(l)})
\end{aligned}
\tag{3.22}
$$

Putting it all together, we can formulate the backpropagation algorithm for a single training example x as shown in Algorithm 1. This procedure is repeated for every example x and the average of the computed gradients determines the direction of each step during gradient descent.

3.2.3.2 Stochastic Gradient Descent

It should be noted that there are several extensions to the algorithm of gradient descent. One very popular variation called *stochastic gradient descent* [Bot12] does not compute the gradient with respect to every single example, but divides the whole dataset into separate randomly sampled batches instead, resulting in an approximation of the gradient of the total loss with respect to every example in the dataset. Each batch is then used to take a step of gradient descent by computing the average gradient of all the examples

Algorithm 1 Backpropagation

$x \leftarrow$ current example
for each layer $l = 2, \ldots, n$ **do** ▷ Feed the input through the network
  for each neuron k in l **do**
    $z_k^{(l)} \leftarrow \sum_{j=1}^{n_{l-1}} w_{kj}^{(l)} \cdot y_j^{(l-1)} + b_k^{(l)}$ ▷ Compute the summed input
    $y_k^{(l)} \leftarrow \phi(z_k^{(l)})$ ▷ Compute the activation
  end for
end for
for each layer $l = n, \ldots, 2$ **do** ▷ Backward pass
  for each neuron k in l **do**
    if l is output layer **then** ▷ Compute delta
      $\delta_k^{(l)} \leftarrow 2 \cdot (y_k^{(l)} - y'_k(x)) \cdot \phi'(z_k^{(l)})$ ▷ See 3.21
    else
      $\delta_k^{(l)} \leftarrow \left(\sum_{i=1}^{n_{l+1}} \delta_i^{(l+1)} \cdot w_{ik}^{(l+1)} \right) \cdot \phi'(z_k^{(l)})$ ▷ See 3.22
    end if
    for each neuron j in $l-1$ **do**
      $\frac{\partial L_x}{\partial w_{kj}^{(l)}} \leftarrow \delta_k^{(l)} \cdot y_j^{(l-1)}$ ▷ Calculate gradient w.r.t. weight, see 3.17
    end for
    $\frac{\partial L_x}{\partial b_k^{(l)}} \leftarrow \delta_k^{(l)}$ ▷ Calculate gradient w.r.t. bias, see 3.18
  end for
end for
return $\nabla L_x(w, b)$ ▷ Return the gradient of L_x

in the batch, which is done to speed up the process of learning by not having to iterate over the whole dataset to take one step in the parameter space. The size of of a batch in stochastic gradient descent controls the quality of the approximation. Small batch-sizes lead to noisier gradient updates as they are more sensitive to single examples while bigger batch sizes lead to smoother optimization. It is not always advantageous however, to use a big batch size, as this sometimes leads to slow training and stagnation of the updating procedure.

3.2.4 When to Stop Training

Being able to compute the weight and bias updates during each step of optimization, there still remains the question when to stop the training procedure. In practice it is very common to only train for a predefined number of passes through the whole dataset. Such a pass through the training data, during which every single training example is presented to the network once, is called an epoch.

It is also common to have a third dataset next to the training and testing set which is used to validate that the network still improves its generalization ability during training. This dataset is called the *validation set*. Usually, during each epoch, the error rate on the validation set is computed. If this error does not decrease further during training for a predefined number of epochs, the training is stopped. This technique is also called *early stopping*.

3.2.5 Initializing the Network

The last step that has to be taken before training a neural network to solve a classification problem is initializing the weights and biases. At first glance it may be tempting to initialize all the parameters with the same number, for instance zero. Investigating further on this idea yields that this way of initializing the weights and biases makes it impossible for the network to learn. If all the weights in a layer have the same value, then every neuron in that layer will receive the same compounded input and thus will emit the same signal. Looking back at the backpropagation algorithm, it becomes evident that this leads to equal weight updates as well. This causes the neurons in a layer to continuously show equal amounts of activity which prevents the network from learning any meaningful concepts.

To overcome this symmetry between neuron activations that is induced by initializing the weights equally, one has to employ initialization techniques that lead to *symmetry breaking* among the neurons. This is best achieved by initializing each weight with a

random number. A very common approach is to use either a gaussian or a uniform distribution with a mean of zero and a variance of $\frac{2}{n_{in}+n_{out}}$, where n_{in} is the number of neurons in the preceding layer and n_{out} is the number of neurons in the following layer of the weight. This procedure is also called "Xavier Initialization" and leads to improved learning beyond just breaking the symmetry between neuron activations by achieving a balance of the weight values that works well in practice [GB10].

4 Convolutional Neural Networks

While fully connected feed forward networks work well on data of moderate dimensionality, training them becomes increasingly more difficult once the number of inputs grows. In the case of image classification for example, even an image with just a resulution of 256x256 pixels produces $256 \cdot 256 = 65536$ inputs. Considering that colored images usually have at least three color channels (take the popular **R**ed **G**reen **B**lue color model as an example), this multiplies the amount of input dimensions by a factor of 3, yielding $65536 \cdot 3 = 196608$ dimensions total. If we wanted to use a fully connected hidden layer with only half as many hidden neurons, we would have to optimize $196608 \cdot 98304 = 19,327,352,832$ weights only for the first layer. Let us assume that storing a weight in single precision floating point format costs 4 bytes. This would lead to spacial requirements of $19,327,352,832 \cdot 4 = 77,309,411,328$ bytes or 77.31 gigabytes! One should quickly notice that using this kind of neural networks to classify images is beyond infeasible. But how is it possible that state-of-the-art classifiers achieve human like performance in image classification, also relying on artificial neural networks [Rus+14]? In the following sections, we will explain how to modify our current feed forward architecture in order to cope with these challenges and how these astonishing results are possible.

4.1 Overview

One characteristic of fully connected feed forward networks is that every neuron in the first hidden layer is connected to every input node. While this allows every neuron to make use of every piece of information accross the whole input, it also increases the complexity of the task that each neuron tries to solve. If the data has a specific spatial structure, as would be the case for images, this knowledge is not incorporated into the fully connected architecture. In Convolutional Neural Networks (CNNs), the goal is to reduce the model complexity by making use of the spacial structure of the input. This is done by connecting each neuron of the first hidden layer only to a locally constrained area of the image. We call this area the *local receptive field* of the neuron, see Fig. 4.1.

Each neuron in the first hidden layer will receive its own local receptive field of inputs that it manages. This way, every hidden neuron is no longer fully connected to every input, instead it is only connected to a specific part of it. Looking at it from a different perspective, the local receptive field acts like a sliding window that is moved across the image, where each indermediate position is represented by a hidden neuron. To reduce the complexity of the model even further, every hidden neuron of a layer will share its weights and biases with all the other neurons of that same layer, drastically decreasing the amount of parameters. Intuitively, this means that each neuron will look for the same input feature in the data, the only difference being the local receptive field that each individual neuron observes. This concept, also called a *convolution layer*, is the main foundation of the CNN architecture [Nie15].

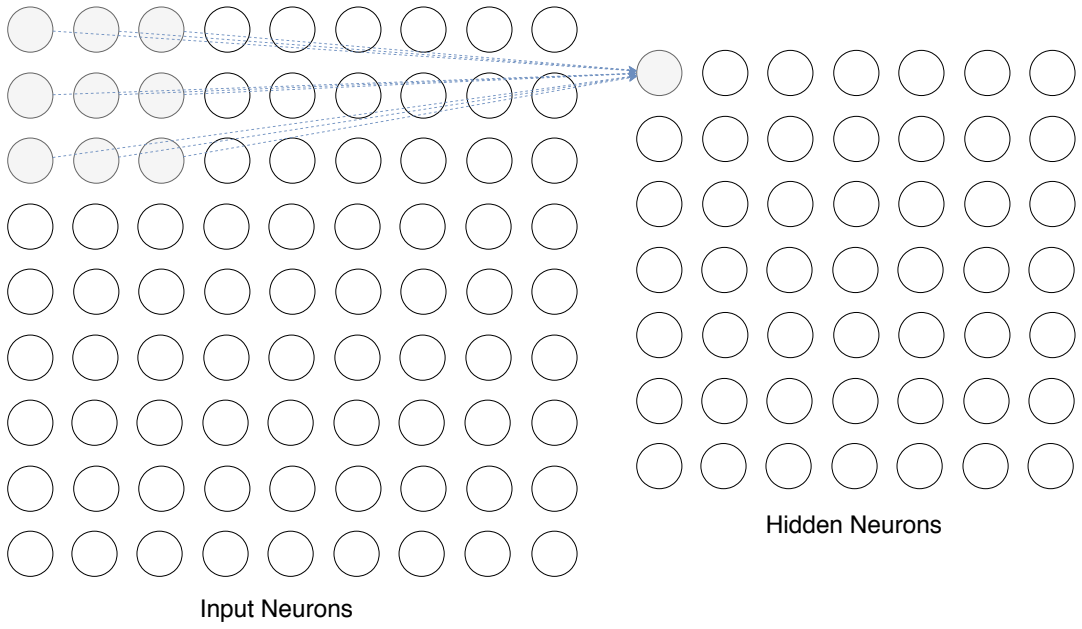

Figure 4.1: Illustration of the local receptive field of the first hidden neuron represented by the grey input units. The weights (depicted by the blue arrows) are shared among the hidden neurons.

4.2 The Convolutional Architecture

The convolutional architecture is composed of three main parts [PG17][1]:

1. The input layer
2. Feature extraction layers
3. Classification layers

These components are stacked on top of each other to successively break down the classification task into smaller problems, by first extracting the relevant features from the data and then performing classification on the basis of these high level features. All the relevant layer types are described in more detail within the following sections.

4.2.1 The Input Layer

Because CNNs are based on spatial assumptions on the input data, it has to be arranged in a way that makes optimal use of its structure. This is why it is most common to organize the input neurons in the form of a two dimensional grid as shown in Fig. 4.1, where each cell resembles a part of the input. In the case of image classification for example, one would typically set the x and y dimensions of the grid equal to the number of pixels in the input image in the corresponding directions. This results in each input neuron in the grid resembling one pixel in the image. If multiple input channels are present, one would simply add a new layer of input neurons per channel, resulting in a three dimensional arrangement. As was the case with ordinary feed forward networks, the input layer does not compute an activation function and just passes its signal further into the network.

4.2.2 Feature Extraction Layers

4.2.2.1 Convolution Layers

The convolution layer, which was briefly described above in section 4.1, is the main component of the feature extraction block. It is used to detect features across the input by having each neuron watch a specific part of it. This can be visualized best by imagining the neurons be arranged in a two dimensional grid where each neuron in the grid watches a corresponding area in the input (see Fig. 4.1). To describe this behaviour more

[1] See *CNN Architecture Overview* in chapter 4.

precisely, we can slightly extend the neural model we have already developed for feed forward networks (see section 3.1.1).

Recalling that the first component of the basic neural model was the weighted sum of the inputs with a set of weights, we can transfer this concept to convolutional layers by interpreting the weights as a filter that each neuron applies to its local receptive field. This filter, that we will also refer to as a kernel, can be described by a stack of matrices of equal dimensions, where each matrix consists of the weights that are applied to the local region in the input within one channel. The amount of matrixes stacked corresponds to the depth of the input data. When dealing with images, this would be the same as the amount of channels (e.g. RGB has three color channels, thus the kernel would consist of three stacked matrices). To compute the weighted sum of the input with the filter, the dot product between the local area of the input and the filter is applied and a bias value is added as usual.

This procedure can also be expressed in terms of mathematical equations as follows: Let the width of the kernel K be x_K, the height be y_K and the depth be d, where d is usually equal to the amount of different channels when dealing with images. Let us further assume that the local receptive field of the current neuron k can be described by the coordinates (i, j) with respect to the input X, where (i, j) indicates the shift of the field in x and y direction. The result of applying the filter and adding the bias value b can be denoted by the following formula:

$$z_k = \sum_{n=1}^{d} \sum_{m=1}^{y_K} \sum_{l=1}^{x_K} X[i+l, j+m, n] \cdot K[l, m, n] + b \tag{4.1}$$

This operation is very similar to the concept of convolution in the area of signal processing, which is the origin of the name *convolutional* neural networks.

After calculating the summation result z_k for every neuron in the grid, an activation function is applied just like in feed forward networks. Because of its various advantages, the most common choice is the ReLU function (see section 3.1.2) which results in the following expression describing the output y_k of every neuron k in the grid:

$$y_k = max(0, z_k) \tag{4.2}$$

Carrying out this computation on every neuron in the grid yields a new pattern of activations across the layer, which is called a *feature map* and can be interpreted intuitively as follows: When computing the dot product between the kernel K and the local receptive field of the neuron, this dot product acts like a similarity measure between the local input

and K. When the dot product is large, this indicates that the kernel and the local input area very similar, when it is very negative, the input and the kernel are contrasting. If the dot product is close to zero, this means that kernel and input have almost nothing in common. Thus, areas of high activity in the feature map indicate, where the input is most similar to the kernel. Thinking further about this relationship, it follows that the kernel K itself represents a *feature* in the input that every neuron watches out for, which is why the convolutional layer is used as a *feature extractor*.

To leverage this procedure even further, each convolution layer does not only produce a single activation map for a single feature, but repeats the process multiple times with different kernels, yielding a stack of activation maps that display the presence of multiple features. In addition to the amount of different features to detect, there are a few other parameters to adjust in a convolution layer. A brief overview of all these tunable parameters will be provided in the following.

Kernel Size The kernel size can be described by the width and the height of each neurons' local receptive field, see the grey input neurons in Fig. 4.1. In Eq. 4.1, these dimensions are denoted by the parameters x_K and y_K. Note that the depth of the kernel is always predefined as the depth of the input from the previous layer.

Kernel Stride This parameter defines how much the local receptive field is shifted from one hidden neuron to the next. A stride of one in x and y direction would indicate that the field is first shifted by one unit along the x dimension and as soon as the end of the input is reached, it would be reset to $x = 0$ and shifted one unit along the y dimension, repeating the procedure. If the stride is smaller than the size of the kernel, the local receptive fields of the hidden neurons will overlap. In Fig. 4.1, there is a 7x7 grid of hidden neurons which indicates that a stride of one in both dimensions was used.

Number of Kernels This parameter describes the number of kernels to apply in a convolution layer and, as we saw earlier, corresponds to the amount of features to detect. Because every kernel results in its own feature map, the depth of the output of a convolution layer is equal to its number of kernels.

4.2.2.2 Pooling Layers

Pooling layers are usually inserted after convolution layers to capture the most essential patterns of the feature maps in order to reduce their complexity. This is done by downsampling the activations, resulting in a smaller input grid for the next layer. Quite

similar to convolution layers, a pooling layer can also be imagined as a filter that runs over the input, but instead of computing dot products, this filter only returns the maximum value of the region it is currently looking at (see Fig. 4.2). This technique is also known as *max-pooling*. It should be noted that there exist alternative techniques such as average-pooling, where the average of a region is computed and not the maximum, however max-pooling has proved itself as the most effective pooling algorithm in practical applications. Unlike convolution layers, pooling layers do not influence the depth of the input because each feature map is processed separately. There are two parameters that can be adjusted in each pooling layer:

- **Size:** This parameter controls the size of the filter that is used for downsampling.
- **Stride:** Similar to the stride parameter of a convolution layer, this value adjusts the step size in each direction that is taken during pooling.

Due to the fact that the downsampling operation purposefully discards information regarding the exact location where the feature was detected, this procedure contributes to the *spatial invariance* of convolutional neural networks. What this means it that the network is able to recognize certain features, no matter where exactly they are located in the image. This is one of the most important traits of the CNN architecture which greatly contributes to its effectiveness in practical applications.

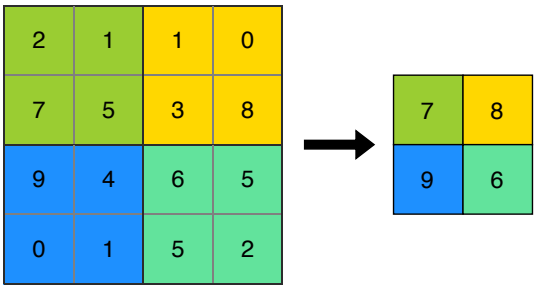

Figure 4.2: Illustration of the max-pooling procedure on a single activation map. The filter size is 2×2 and a stride of 2 is used in each direction.

4.2.3 Classification Layers

After the feature extraction has been performed, the complexity of the input data has usually been reduced significantly by stacking multiple convolution and pooling layers. On the basis of this simpler and also richer representation of the data, an ordinary fully connected feed forward network can be used to perform the classification task. This

is done by inserting a fully connected hidden layer after the last layer of the feature extraction part. This hidden layer is connected to each neuron in the grid. Depending on the complexity of the problem, it is also possible to insert multiple fully connected hidden layers right after each other to complete the classification task. As usual, the last layer consists of as many output neurons as there are classes, employing the softmax activation function to compute the classification result.

4.3 Summary

Putting all these layer types together typically results in a convolutional architecture that is similar to what is displayed in Fig. 4.3. In this example, the convolution layers as well as the pooling layers serve the purpose of feature extraction while the fully connected network in the end is used to perform the classification task based on the extracted features. Just as described in the previous sections, the convolution layers usually yield multiple feature maps while the pooling layers just downsample the input they are presented with.

Similar to feed forward networks, these architectures can also be trained using the gradient descent algorithm in combination with backpropagation. During training, the weights and biases of each layer, namely the different kernels for the convolution layers, are adjusted in order to find a locally optimal configuration that suits the classification task well.

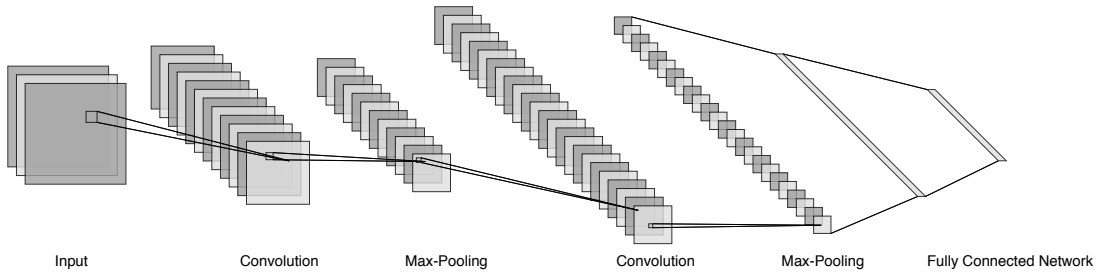

Figure 4.3: The typical architecture of a convolutional neural network.

5 Implementing the Fault Detector

After all the baseline foundations of deep learning and convolutional neural networks have been derived, it is now time to apply these methods to the problem of fault detection in the CLAS12 drift chamber. Due to the CLAS12 productive environment heavily relying on the JAVA programming language, all implementations are carried out utilizing JAVA as well. The basis of the autonomous fault detection system is formed by the deeplearning4j (DL4J) library which is briefly described in the following section.

5.1 The Deeplearning4j Library

Deeplearning4j is an open-source deep learning library written for the JAVA Virtual Machine (JVM). It is supported by the well known AI-Startup Skymind and contains implementations of many algorithms from the artificial intelligence domain, such as convolutional neural networks, that will be used when implementing the fault detector.

To speed up the computations that have to be performed during the training of deep convolutional architectures, DL4J relies on its own numerical backend engine, ND4J, that contains C++ and Cuda implementations of all required operations, most importantly matrix multiplications. This way, deep neural networks can be trained on big clusters containing multiple CPUs and GPUs, leveraging the huge amounts of parallelism introduced by these computational architectures.

Additionally, the DL4J library also provides useful mechanisms that make monitoring the training process highly accessible, such as Web-UIs which offer visual representations of the loss function as well as the weight updates during training. Evaluating the trained classifier is also not a difficult task, as DL4J provides designated classes that are designed to compute all the relevant metrics on the training dataset.[1]

[1] See `www.deeplearning4j.org` for more information.

5.2 Data Preparation

Before the data can be fed into a convolutional neural network, a few steps of preparation are necessary. Recall the heatmap plots of different faults shown in section 2.3. The data that will be presented to the network will be organized in the form of a 6×112 array representing the activation level of each wire in a superlayer, similar to the heatmaps. This way, the spatial structure that is present within the data is preserved, which is essential when working with CNNs, because as shown in section 4, these architectures heavily depend on structural assumptions on the input. Unlike in the case of images, the fault data will only consist of a single channel, resulting in an input volume of size $6 \times 112 \times 1$.

Due to the fact that activation levels can vary drastically among different regions, regions far away from the electron beam receiving less particles passing by, the data has to be normalized in order to compensate for these effects of disproportion. This is done by projecting the activation values within each input on a numerical scale between 0 and 1, the lowest activation ending up at 0, the highest at 1. This procedure will make it easier for the network to ignore the differences in absolute activation levels and focus more on the local fault patterns instead.

5.3 The Fault Detection System

When looking at the examples of various faults that can occur in the drift chamber (see section 2.3), it becomes apparent that there are usually multiple faults happening within a single superlayer (see for instance Fig. 2.4, where there are a dead connector as well as multiple dead wires present in the data). To account for this circumstance, multiple CNNs are trained on the fault data, each individual CNN only specializing in recognizing a single fault type (see section 2.3 for a summary of the different fault types), thus resulting in a binary classifier for each fault. To obtain information on the various faults present in a single input, each individual CNN is presented with the corresponding data. In the next step, all the individual responses are collected, resulting in a list of faults that were recognized within the given superlayer. The network architecture used is similar for each CNN that is part of the fault detector as described in the following sections.

5.3.1 Finding the Right Architecture

One of the most challenging tasks when applying deep learning is to find a set of suitable parameters that describe the network architecture. These parameters, e.g. number of hidden layers, kernel size of convolution layers, activation functions, learning rate, ... are also refered to as *hyperparameters*, because they are set in advance and are not learned by the network during training. In practice, the most common approach of finding the right setup is to try many different architectures and configurations and stick to those that work best.

While designing the network that is used in the fault detector, the simplest models, consisting of only a single convolution layer followed by a pooling layer and fully connected layers, were tested first. While working well with faults that are easily recognizable, such as a dead channel, those models failed to capture the more complex mechanisms of fault detection like dead wires. The dead wire fault turned out to be the most complex of all because it can basically appear everywhere, even next to each other. To cope with these challenges, a deeper architecture with six layers was used which was able to successfully capture all the peculiarities associated with the various fault types. This architecture, resulting from the process of testing multiple combinations of hyperparameters, is presented in the next section.

5.3.2 Network Architecture and Configuration

After the $6 \times 112 \times 1$ input volume resulting from the normalized activations of a superlayer, a convolution layer is inserted as the first feature extractor of the network (see Fig. 5.1). The layer uses a kernel size of 2×3 which accounts for the unsymmetrical dimensions of the input. The total number of kernels within this layer, i.e. the amount of features to look for, is set to 40 and a stride of 1 in each direction is used. The ReLU is applied as an activation function to this layer. Another convolution layer is inserted afterwards, this time using 30 kernels of size 2×2, also setting stride to one. The second convolution layer is designed to search for higher level features by scanning the activation maps of the first convolution layer. Following these layers, a max-pooling layer is employed with a filter size of 2×2 and a stride of 2×2 as well. This is done in order to compress the feature data and contribute to the spatial invariance of the network (see section 4.2.2.2). Another convolution layer with 20 kernels of size 2×2 is inserted next, scanning for even higher level features in the downsampled output of the pooling layer. Afterwards, a single fully connected hidden layer with 100 hidden neurons is employed, utilizing the ReLU activation function. The output layer consists of two output neurons,

one firing if the designated fault was detected, the other firing if not. The softmax activation function is used on this layer to make the ouputs interpretable as probabilities. Overall, this architecture results in a total of 113,182 parameters (weights and biases) that have to be adjusted during training.

Network initialization is performed using the Xavier method (see section 3.2.5), ensuring a weight distribution that enhances network trainability. As for the optimization algorithm, a slight variation of the traditional stochastic gradient descent algorithm (see section 3.2.3.2) is employed, called ADAM. The advantage of this algorithm is that it is able to adjust the learning rate during training depending on the local characteristics of the loss function by scaling the gradients based on previous updates, resulting in a robust procedure for the training of convolutional neural networks [KB14]. It's default learning rate of 0.001 proved to be sufficient for the training task. The gradients in each step of the updating process are computed using the backpropagation algorithm (see section 3.2.3.1).

5.4 Training the Fault Detector

In order to train the fault detector as a whole, every single binary CNN was trained in isolation on a collection containing 100,000 positive as well as negative examples of the fault type that it is supposed to detect. The data used during training was generated by a simulation suite that works on real fault-free heatmaps and inserts random combinations of faults which are then labeled in accordance and fed into the network. In order to avoid overfitting (see section 3.2.2), the wire activations were randomly perturbed by the simulation suite during training, ensuring that every example the classifier was presented with was unique.

The learning curve that plots the model score against the first 5,000 training examples of the dead pin classifier, as illustrated by the DL4J Web-UI, can be seen in Fig. 5.2. It strikes the eye that the optimization algorithm seems to step through rough terrain in the parameter space, as there are multiple spikes occuring before convergence finally sets in. This is most likely due to the fact that a batch size (see section 3.2.3.2) of one was used during optimization, conciously inducing the spikes to add more variation to the optimization process, increasing the likelihood that a minimum will eventually be found. It should be noted that bigger batch sizes of more than 20 examples were tested as well, resulting in stagnation of the optimization procedure and not leading to convergence. For this problem, it turned out, more noise was helpful during training.

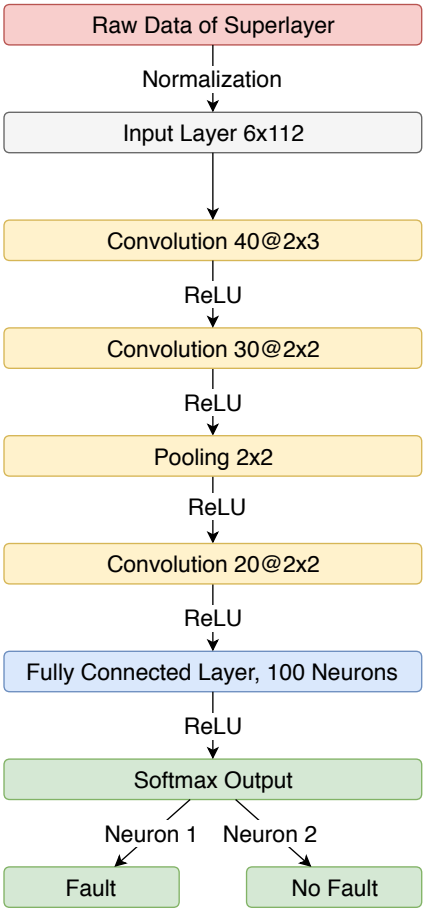

Figure 5.1: The architecture of a single CNN in the fault detector that is trained to identify a unique fault type.

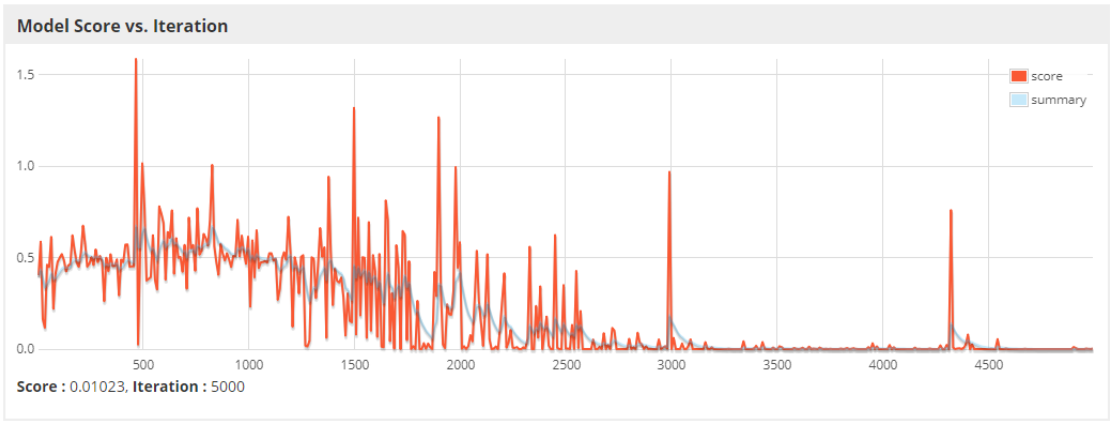

Figure 5.2: The DL4J Web-UI learning curve of the dead pin fault classifier for the first 5,000 examples shows many rough spikes but eventually leads to convergence.

5.4.1 Training Results

The training performance of every single binary classifier was evaluated on 10,000 new randomly generated examples using the common evaluation metrics as described in section 3.2.1.1. An overview of the results is presented in the following paragraphs.

Dead Wire Classifier:

- Accuracy: 97.61%
- Precision: 99.96%
- Recall: 95.65%
- F-Measure: 97.76%

The confusion matrix of the dead wire classifier can be found in Table 5.1.

	Dead Wire (Predicted)	**No Dead Wire (Predicted)**
Dead Wire (Actual)	5212	237
No Dead Wire (Actual)	2	4549

Table 5.1: Confusion matrix of the dead wire classifier.

Dead Pin Classifier:

- Accuracy: 99.95%
- Precision: 99.92%
- Recall: 99.98%
- F-Measure: 99.95%

The confusion matrix of the dead pin classifier can be found in Table 5.2.

	Dead Pin (Predicted)	**No Dead Pin (Predicted)**
Dead Pin (Actual)	4739	1
No Dead Pin (Actual)	4	5256

Table 5.2: Confusion matrix of the dead pin classifier.

Dead Connector Classifier:

- Accuracy: 98.77%
- Precision: 99.23%
- Recall: 95.69%
- F-Measure: 97.43%

The confusion matrix of the dead connector classifier can be found in Table 5.3.

	Dead Connector (Predicted)	**No Dead Connector (Predicted)**
Dead Connector (Actual)	2334	105
No Dead Connector (Actual)	18	7543

Table 5.3: Confusion matrix of the dead connector classifier.

Dead Fuse Classifier:

- Accuracy: 98.95%
- Precision: 97.32%
- Recall: 98.20%
- F-Measure: 97.76%

The confusion matrix of the dead fuse classifier can be found in Table 5.4.

	Dead Fuse (Predicted)	**No Dead Fuse (Predicted)**
Dead Fuse (Actual)	2288	42
No Dead Fuse (Actual)	63	7607

Table 5.4: Confusion matrix of the dead fuse classifier.

Dead Channel Classifier:

- Accuracy: 99.11%
- Precision: 98.84%
- Recall: 98.64%
- F-Measure: 98.74%

The confusion matrix of the dead channel classifier can be found in Table 5.5.

	Dead Channel (Predicted)	**No Dead Channel (Predicted)**
Dead Channel (Actual)	3493	48
No Dead Channel (Actual)	41	6418

Table 5.5: Confusion matrix of the dead channel classifier.

Hot Wire Classifier:

- Accuracy: 100.00%
- Precision: 100.00%
- Recall: 100.00%
- F-Measure: 100.00%

The confusion matrix of the hot wire classifier can be found in Table 5.6.

	Hot Wire (Predicted)	No Hot Wire (Predicted)
Hot Wire (Actual)	5532	0
No Hot Wire (Actual)	0	4468

Table 5.6: Confusion matrix of the hot wire classifier.

5.5 Evaluation on Real Data

When fitting a model based on simulated data, it is always important to verify that it was able to grasp the underlying physical concepts and did not just adapt to the characteristics of the simulation suite. In order to prove that the classifiers were able to generalize beyond the simulations, some real data examples showing the strenghts of the models as well as some weaknesses are discussed within the scope of this section.

Example 1 - A Pin Fault and Dead Wires: Fig. 5.3 shows a superlayer that contains a dead pin as well as some dead wires. Both the dead wire classifier as well as the dead pin classifier report with 100% certainty that they detected a fault. All the other classifiers are more than 99% certain that their designated fault type is not present within this superlayer. This example illustrates that the model had no difficulty in recognizing these clearly defined faults.

Example 2 - A Dead Connector and Dead Wires: When presented with a dead connector as shown in Fig. 5.4, the classifier succeeds spotting the dead connector with 100% certainty. The dead wires around that connector are successfully detected with 100% certainty as well. In this example, there is a little more noise present around the dead connector which turns out not to be a problem for the classifier.

Example 3 - A Blurred Dead Pin: When the fault is not as clearly defined as it was in 5.3, the model starts showing some signs of difficulty. In Fig. 5.5, we can see a dead pin that does not display such a high contrast with respect to its environment. The model struggles with this example, showing a certainty of over 99.99% that no dead pin is present in the superlayer. The dead wire that is also visible in this example was recognized with 100% certainty not showing any problems.

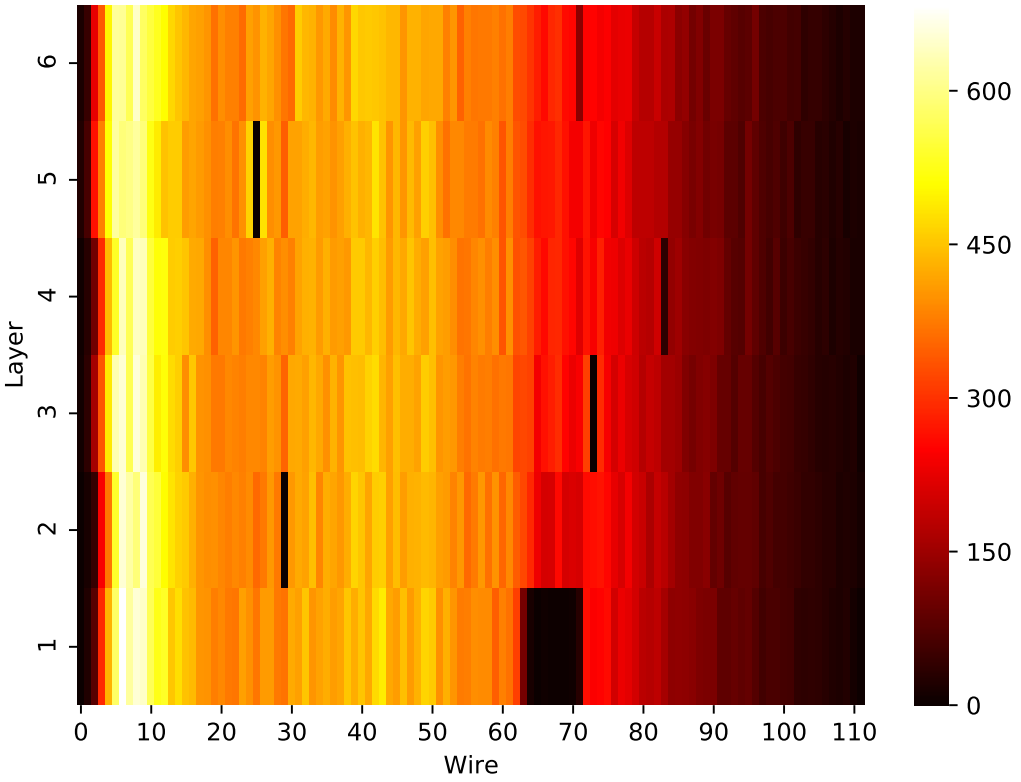

Figure 5.3: A superlayer showing a faulty pin as well as some dead wires.

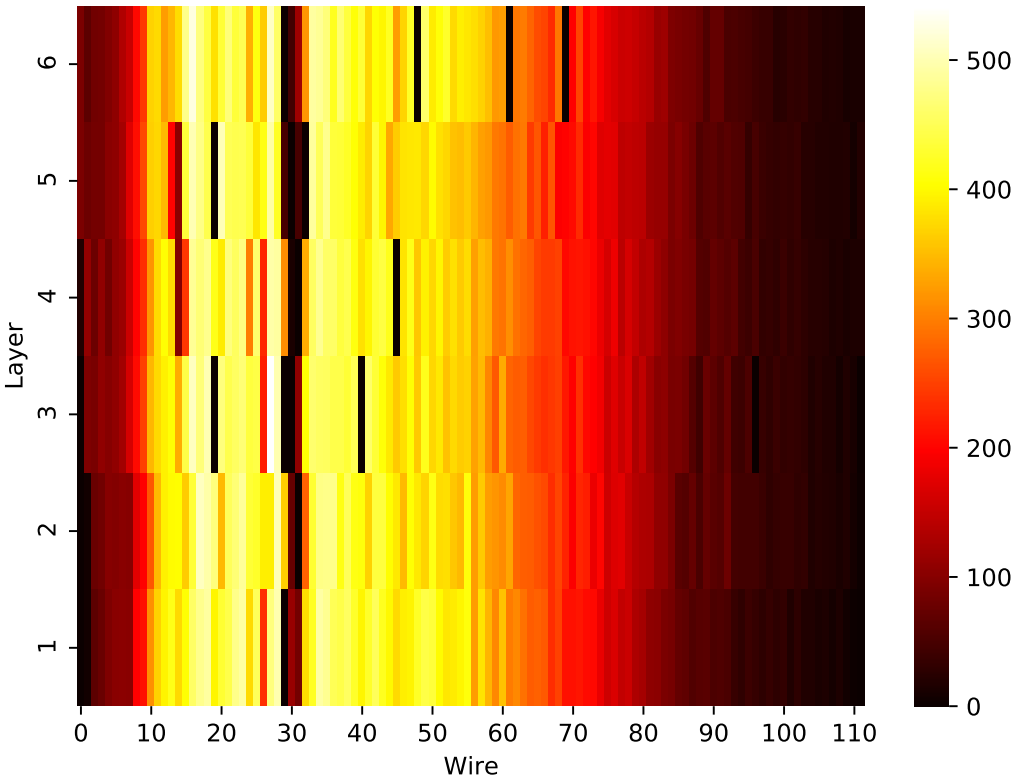

Figure 5.4: This superlayer contains a dead connector fault as well as some dead wire faults.

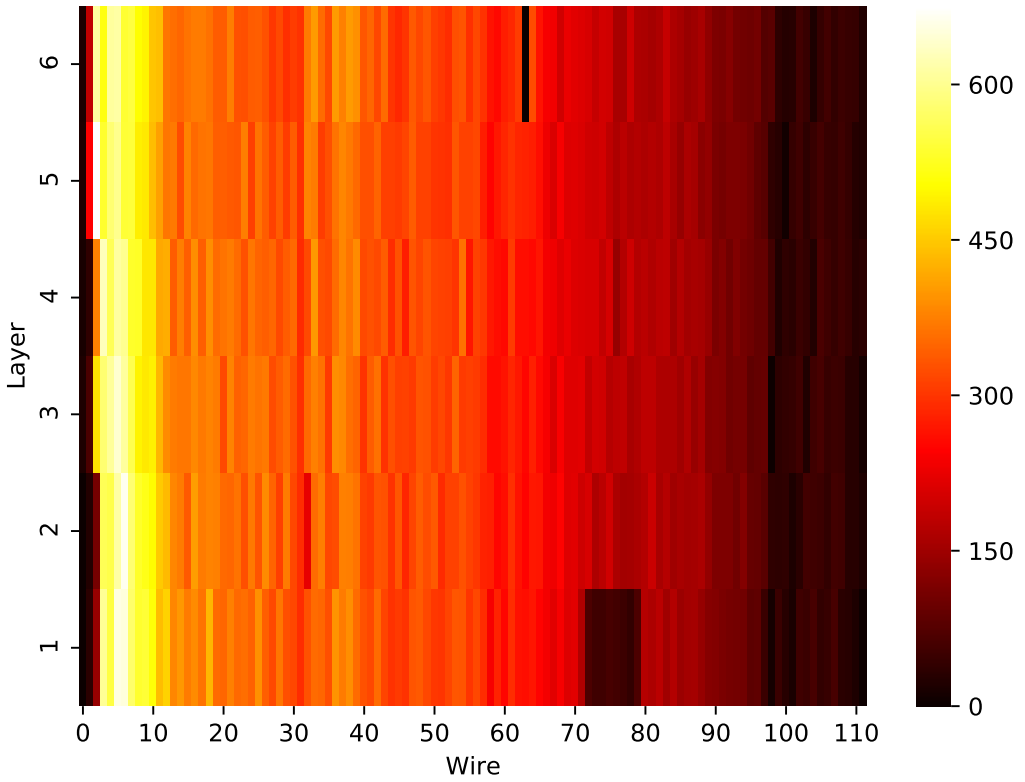

Figure 5.5: This superlayer shows a blurred example of a dead pin that caused problems in classification.

This example illustrates that the classifier shows signs of weakness when faults are blurry and not clearly defined, which is also the case with other faults such as dead channels or dead fuses. Whenever a component of a superlayer is not completely dead but clearly faulty, the classifier struggles in recognizing the fault. This problem could be remedied however, by adjusting the simulation suite to show more examples of blurry faults during training.

Example 4 - Two Dead Wires Next to Each Other: In Fig. 5.6 we can see the interesting situation that two dead wires are right next to each other. Despite this being a rare occasion, the classifier was able to report with 93.29% certainty that it spotted a dead wire. The dead pin that is also present in this example was recognized with 100% certainty.

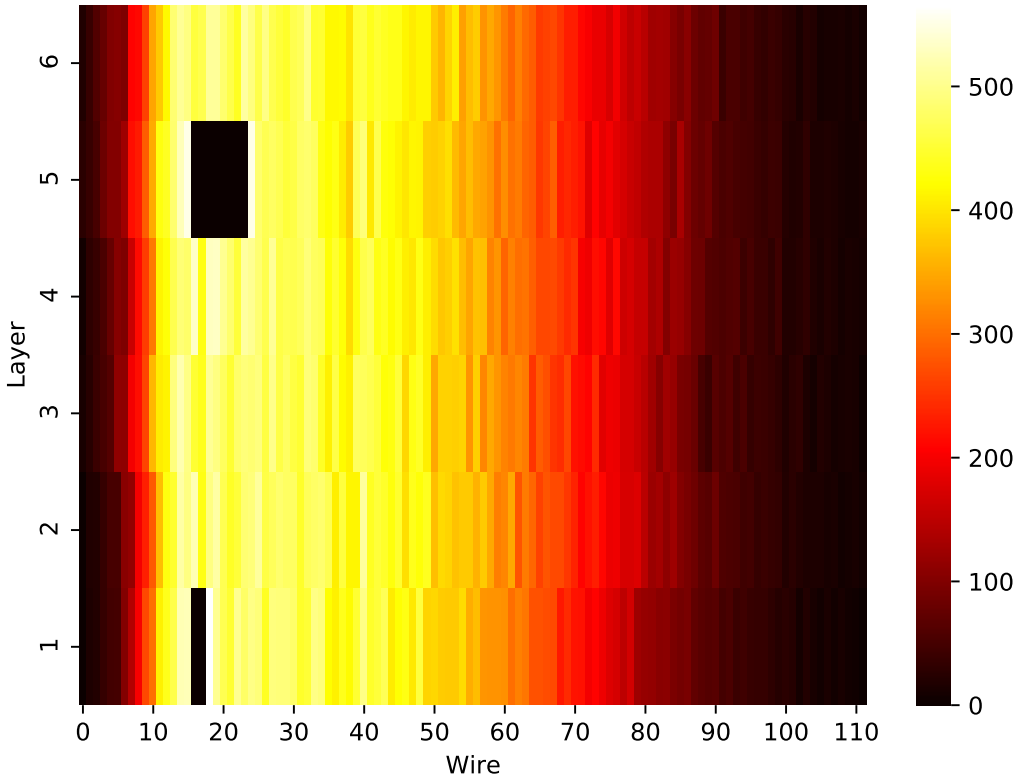

Figure 5.6: Two dead wires next to each other in layer one.

6 Conclusion

Bibliography

[Ben12] Y. Bengio. "Practical recommendations for gradient-based training of deep architectures". In: *ArXiv e-prints* (June 2012). arXiv: 1206.5533 [cs.LG].

[Bot12] Léon Bottou. "Stochastic Gradient Descent Tricks". In: *Neural Networks: Tricks of the Trade*. Springer, Berlin, Heidelberg, 2012. ISBN: 978-3-642-35288-1.

[GB10] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. PMLR, 13–15 May 2010, pp. 249–256.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Geoffrey Gordon, David Dunson, and Miroslav Dudík. Vol. 15. Proceedings of Machine Learning Research. Fort Lauderdale, FL, USA: PMLR, Nov. 2011, pp. 315–323.

[Hay08] Simon Haykin. *Neural Networks and Learning Machines*. 3rd ed. Prentice Hall International, 2008. ISBN: 978-0131471399.

[KB14] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". In: *ArXiv e-prints* (Dec. 2014). arXiv: 1412.6980 [cs.LG].

[Nie15] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.

[PG17] Josh Patterson and Adam Gibson. *Deep Learning: A Practitioner's Approach*. 1st ed. O'Reilly Media, 2017. ISBN: 978-1491914250.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *nature* 323 (1986).

[Rus+14] O. Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *ArXiv e-prints* (Sept. 2014). arXiv: 1409.0575 [cs.CV].

List of Figures