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, July 3, 2018

Contents

1 Introduction

2 The CLAS12 Particle Detector

3 Deep Learning Fundamentals

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. To be precise, each ANN consists of a collection of artificial neurons that are connected with each other. The neurons are able to exchange information along their connections. A common way to arrange artificial neurons within a network is to organize them in layers as depicted in figure 3.1.

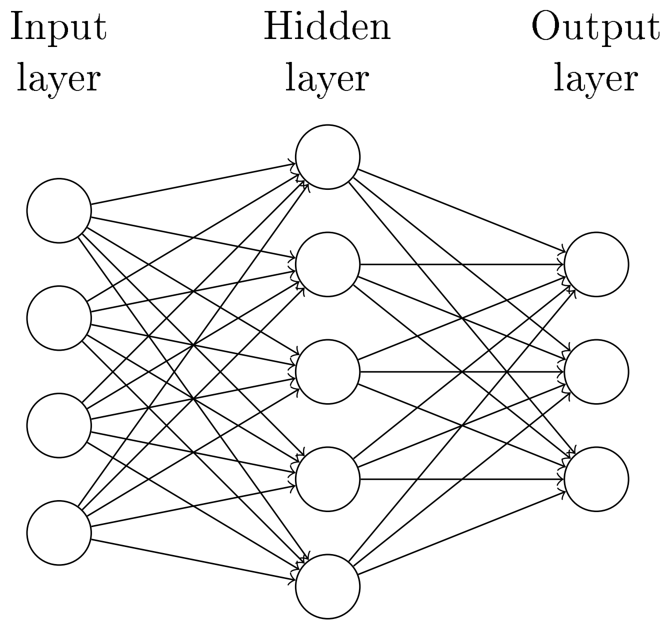

Figure 3.1: The structure of an ANN can be described by a directed graph. The nodes represent the neurons, the edges represent their connections, also indicating the flow of information.

When an artificial neuron receives signals on some of its incoming connections, it may elect to become active based on the input it collects.[1] In this state it also influences all neurons it has an outgoing connection to by passing a signal along their channel. Those other neurons in turn may also elect to become active - this way a signal can propagate through the network along the connecting edges.

Usually, each ANN consists of at least one layer of neurons that is responsible for receiving signals from the environment - we call this an *input layer* (see figure 3.1 on the preceding page). When these neurons receive a signal from the environment, they propagate it to their connected neighbors in the next layer. This process repeats until the *output layer* is reached. The neurons in this layer represent the output of the whole network. Each layer in between is called a *hidden layer* because there is no direct communication between the neurons in this layer and the environment. Networks that satisfy this basic architectural model where each layer is fully connected with its following layer and signals only flow in one direction without cycles are called *fully connected feedforward networks.*

The goal behind this procedure usually 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 essence of 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].[2] As shown in figure 3.2 on the next page, each artificial neuron basically consists of three components:

1. **A set of weighted inputs:** Each connection that is leading into the 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.[3] The signal

[1] The details of this process are further illustrated in section 3.1.1.

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

[3] 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 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.

that passes the connection is multiplied by the related weight of that connection before arriving at the next component.

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 on page 10.

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 on the next page, a more detailed explanation of activation functions as well as some commonly used examples will be provided.

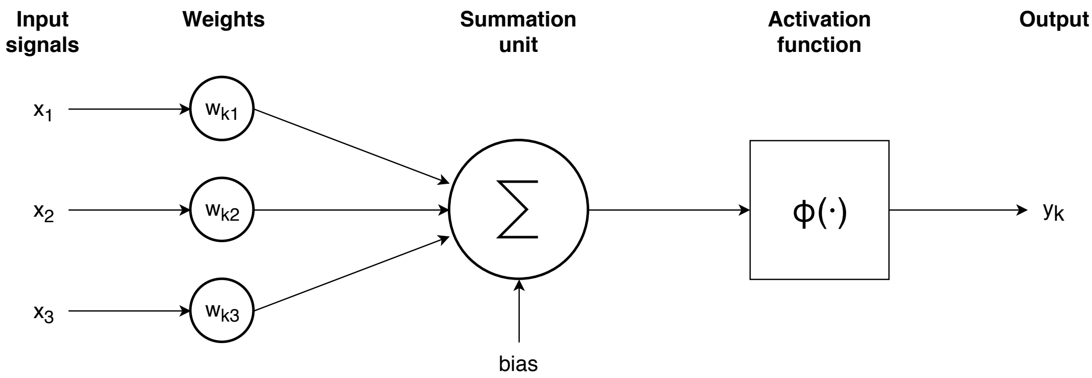

Figure 3.2: The components of a single artificial neuron. This neuron k receives three input signals that are first multiplied by the associated weights, summed up including a bias and then fed into an activation function that will determine the ouput signal.

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:

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

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. Since the output y of the neuron is also a scalar, each activation function can be described as $\phi : \mathbb{R} \rightarrow \mathbb{R}$. In the following paragraphs, an overview of the most popular activation functions will be presented that is based on the descriptions found in [PG17].[4]

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

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

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 figure 3.3 on the following page.

One important reason why the sigmoid function is often used 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.

The Rectified Linear Unit (ReLU) Because it is not always desirable to reduce large signals to a smaller scale, this 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

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

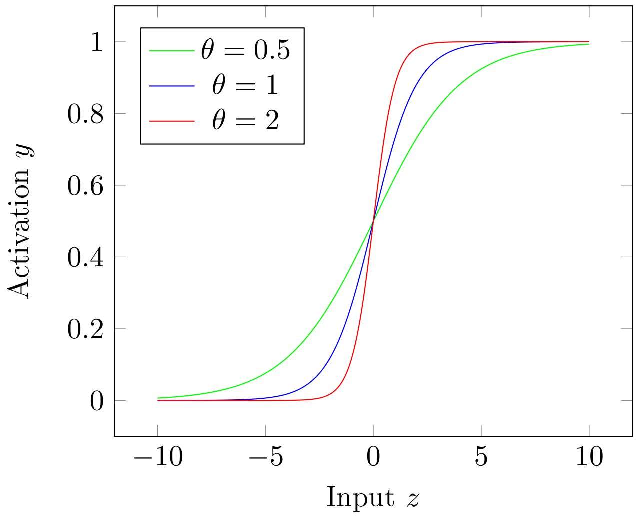

Figure 3.3: The sigmoid activation function plotted for different values of θ.

to some degree by using the ReLU function because large signals are not cut down. Due to the negative values being set to zero, the ReLU function is also non-linear when taking its whole domain into account. This is an important concept because non-linear activation functions are essential for a network to learn complex relationships. Another benefit of the ReLU function is that its derivative is either 1 or 0. This will turn out to be important when looking into the training of a neural network. Because of all these benefits, ReLUs are one of the state of the art activation functions in deep neural networks. A plot of the ReLU function is presented in figure 3.4 on the next page.

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:

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

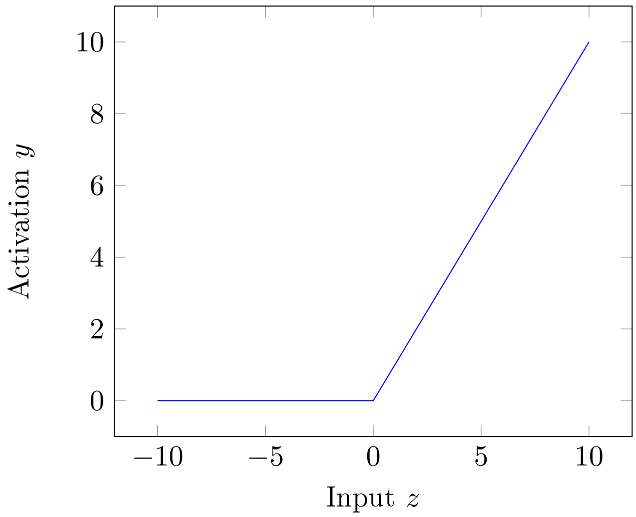

Figure 3.4: The ReLU activation function.

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.

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.[5]

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 equation (3.1) on page 7 that for a neuron k with n inputs the toal input signal z_k adds up to:

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

[5]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 $(0.1, 0.75, 0.1, 0.05)^T$ which amplifies the most likely outcomes and attenuates the less likely ones. See `https://datascience.stackexchange.com/questions/23159/in-softmax-classifier-why-use-exp-function-to-do-normalization` for a nice explanation and the source of this example.

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 on page 8) 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 figure 3.5.

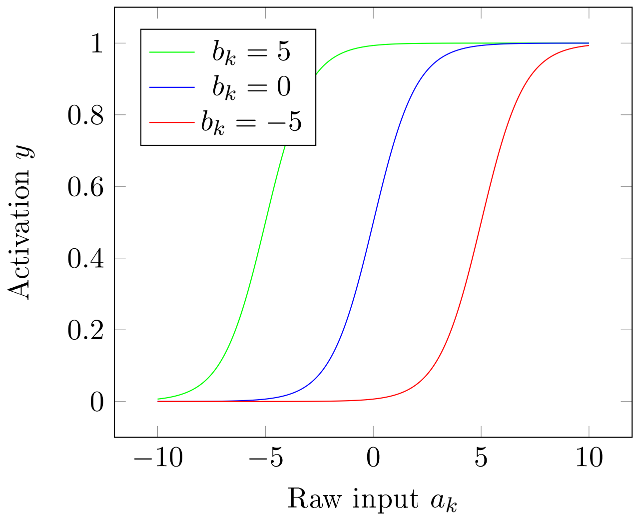

Figure 3.5: The sigmoid activation function plotted for different bias values.

What this shift means is 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*. In the next step, 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*. This phase is also called *testing*. Because classification requires pre-labeled instances and the classifier acts like a learner who learns from a teacher, 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 figure 3.6.[6]

	Class Positive (Predicted)	**Class Negative (Predicted)**
Class Positive (Actual)	True positives (TP)	False negatives (FN)
Class Negative (Actual)	False positives (FP)	True negatives (TN)

Figure 3.6: 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. The resulting measurements of true positives, true negatives, false positives and false negatives can be used to compute the following evaluation metrics:[7]

[6] 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.

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

Accuracy This measurement 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 metric 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 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 be extended to classification tasks with more than two classes as well. 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.

As already hinted at in section 3.1.2 on page 9 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.

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. The danger lays in the fact that if there are too many hidden neurons, 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.

Because there are no general rules up to this point on how to structure the hidden layers of a neural network, it is recommended to apply the basic rule of thumb of increasing the amount of hidden layers with the complexity level of the problem. It also helps to try different structures during the training phase to see which one produces the best results.

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.

Let x be an example input from the training dataset and $y'(x)$ 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 $y'_i(x)$ represents the desired activation of the i'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. Because the goal is to minimize L, one can simply take tiny successive steps in the direction of the *negative* gradient to arrive at a minimum of L resulting in the following algorithm describing how to adjust the weights w and biases b in each step t:

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

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 trainig 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 write the equations that describe the input z and the output y 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 equation (3.11) on the previous page), 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 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 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 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 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 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 figure 1 on the following page. This procedure is repeated for every example x and the average of the computed gradients determines the direction of each step during gradient descent.[8]

[8] A brilliant visual explanation of the backpropagation algorithm can be found at `https://www.youtube.com/watch?v=Ilg3gGewQ5U`. This and the following videos of the series have inspired the notation that was used during this derivation.

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 input of each neuron
    $y_k^{(l)} \leftarrow \phi(z_k^{(l)})$ ▷ 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

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 in separate randomly sampled batches instead. Each batch is then used to take a step of gradient descent by computing the average gradient of all the examples in the batch. This 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.

3.3 Deep Networks

4 Convolutional Neural Networks

5 Implementing and Testing a CNN-Model in DL4J

6 Discussion

7 Conclusion

Bibliography

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

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

[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).

List of Figures