Fachhochschule Aachen
Campus Jülich

Fachbereich: Medizintechnik und Technomathematik
Studiengang: Scientific Programming

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

Eine Bachelorarbeit von Christian Peters

Jülich, den June 25, 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.

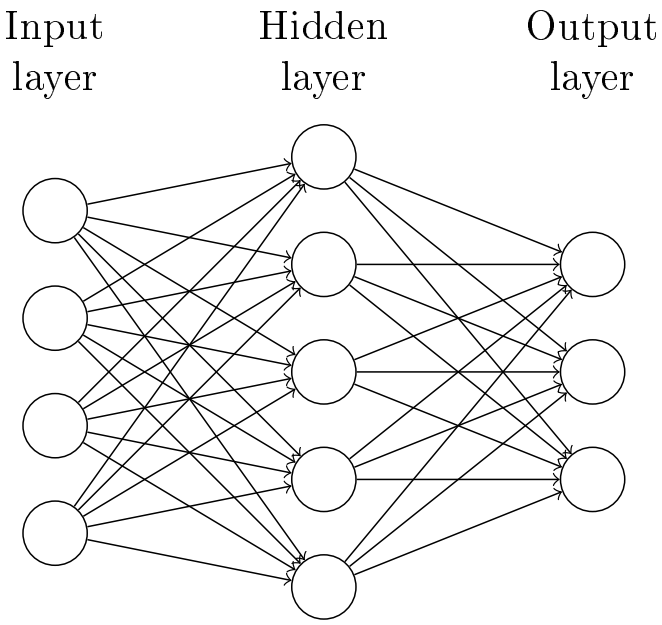

Figure 3.1: The structure of an ANN can be described by a directed graph. The nodes represent the neurons, the edges represent their connections, also indicating the flow of information.

When an artificial neuron receives a signal 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.

3. **An activation function:** The activation function $\phi(\cdot)$ takes the output of the summation unit and applies a transformation to it 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 following page, a more detailed explanation of activation functions as well as some commonly used examples will be provided.

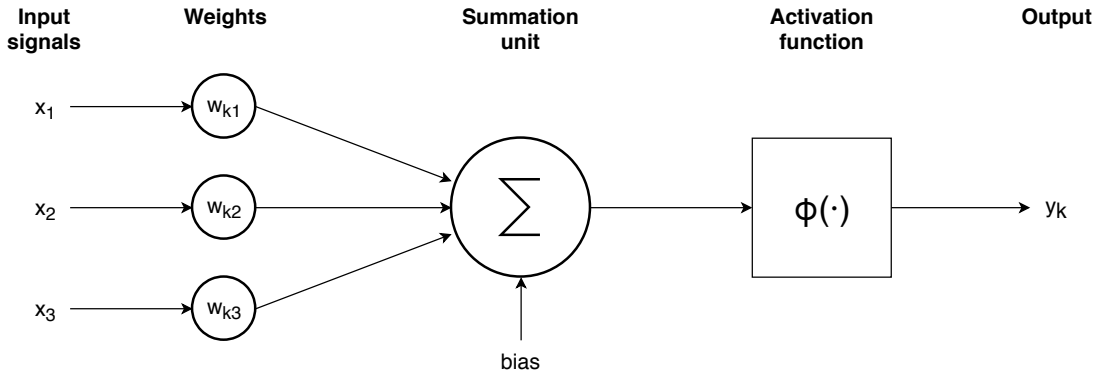

Figure 3.2: The components of a single artificial neuron. This neuron k receives three input signals that are first multiplied by the associated weights, summed up including a bias and then fed into an activation function that will determine the ouput signal.

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 $\phi : \mathbb{R} \to \mathbb{R}$. 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.

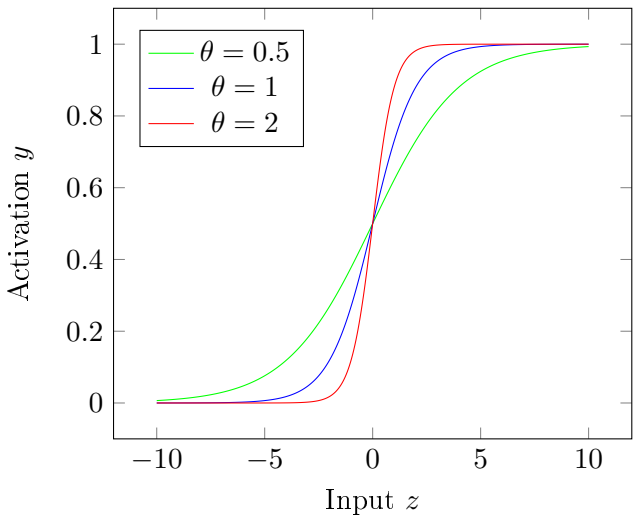

Figure 3.3: The sigmoid activation function plotted with different values of θ.

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

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 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.

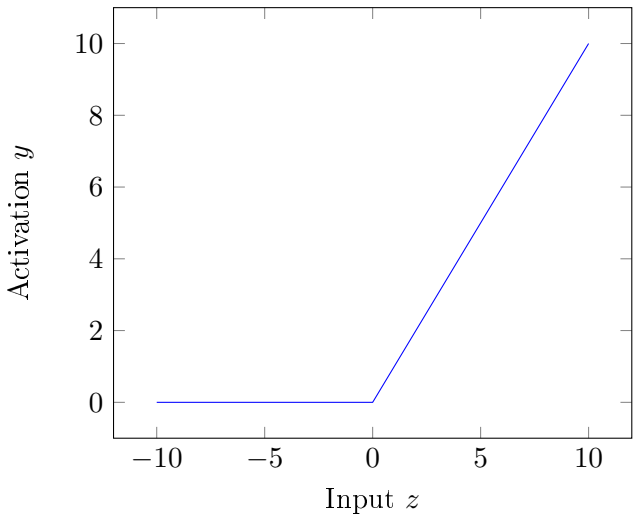

Figure 3.4: The ReLU activation function.

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.

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]

3.2 The Backpropagation Algorithm

3.3 Deep Networks

[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 outcome and attenuates the less likely values. 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.

4 Convolutional Neural Networks

5 Implementing and Testing a CNN-Model in DL4J

6 Discussion

7 Conclusion

Bibliography

[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.

List of Figures