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 17, 2018

Contents

1 Introduction

2 The CLAS12 Particle Detector

3 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 single weight in 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.

3.1 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. Each neuron in the first hidden layer will receive its own local receptive field of inputs that it manages. This way, each hidden neuron is no longer fully connected to every input, instead it is only connected to a specific part of it. 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. Intuitively, this means that each neuron will look for the same input feature in the data, the only difference being the local receptive field the individual neurons watch. This concept, also called a *convolution layer*, is the main foundation of the CNN architecture.

3.2 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.

3.2.1 Feature Extraction Layers

3.2.1.1 Convolution Layers

The convolution layer, which was briefly described above, 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 . . .). To describe this behaviour more precisely, we can slightly extend the neural model we have already developed for feed forward networks.

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.

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

This filter, sometimes also refered 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 filter K be x, the height be y and the depth be z. 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 in x and y direction. The result of applying the filter and adding the bias value b can be denoted by the following formula:

$$z_k = \sum_{n=1}^{z} \sum_{m=1}^{y} \sum_{l=1}^{x} X[i+l, j+m, z] \cdot K[l, m, n] + b \tag{3.1}$$

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 which yields the following expression describing the output y_k of every neuron k in the grid:

$$y_k = max(0, z_k) \tag{3.2}$$

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 convolutiona 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. In Eq. 3.1, these dimensions are denoted by the parameters x and y. Note that the depth of the kernel is always predefined as the depth of the input from the previous layer.

Kernel Stride Up to this point, we did not yet define how to arrange the local receptive fields of the neurons. Introducing a stride parameter, we can set the first local field to the region described by a $(0,0)$ coordinate shift (see the i and j parameters in Eq. 3.1). Each consecutive neuron will receive a receptive field that is shifted by some amount. To arrange these fields, we will first only shift in the direction of the width of the input. The amount of shifting in each step is described by the x_stride of the kernel. After shifting across the width is no longer possible, the x position of the local receptive field is reset to zero and a shift happens in the y direction. This shift can be adjusted by the parameter y_stride. The same procedure is repeated until every area of the input is covered by a neuron.

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 the convolution layer is equal to the number of kernels.

3.2.1.2 Pooling Layers

3.2.2 Classification Layers

4 Implementing and Testing a CNN-Model in DL4J

5 Discussion

6 Conclusion

Bibliography

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

[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