Constructing a Convolutional Neural Network from ground up

Grzegorz Kajda
Bachelor Student, Robotics and Intelligent Systems
Department of Informatics The faculty of Mathematics and Natural Sciences
Email: grzegork@ifi.uio.no

1 Abstract

To gain a more comprehensive understanding of the inner mechanisms of deep learning networks, we developed a Convolutional Neural Network (CNN) from scratch for image classification. The network is structured into two main parts: a feed-forward network and a filtering network consisting of convolutional and pooling layers. Fortunately, we had previously built a feed-forward neural network as part of the subject ”FYS-STK3155” at the University of Oslo. Hence, we utilized this code as a foundation for our new network.

To enhance the modularity and resemblance to deep learning libraries like TensorFlow and PyTorch, we modularized the architecture of our network. Instead of having the entire network in one module, we created separate modules representing different types of layers. This approach enables us to construct flexible architectures in a simple and transparent manner while emphasizing the blackbox concept.

Finally, we evaluated the performance of our initial implementation of a CNN using the 28x28 MNIST dataset, which served as a test benchmark.

2 Introduction

Machine Learning, particularly the sub-branch known as deep learning, has emerged as a powerful tool for data analysis and classification in the contemporary world. It enables us to extract meaningful insights from data with minimal effort and automate tasks like data clustering or generating new data from existing sources. With the advent of deep learning libraries such as TensorFlow and Keras, this process has become even simpler, allowing data scientists and developers to focus solely on their specific tasks without concerning themselves with the implementation details of these algorithms. While this ”blackbox” approach saves valuable time, it can limit our understanding of how the algorithms truly work. Consider the example of backpropagation: deep learning libraries utilize automatic differentiation, eliminating the need to manually compute or propagate gradients backwards from the output to input. However, without referring to external resources, it is unlikely that one would be able to implement the algorithm from scratch.

To gain a deeper understanding of how data is processed and transformed layer by layer, we will develop a Convolutional Neural Network (CNN) from scratch. This endeavor aims to draw insightful conclusions about the inner workings of the network. While prioritizing the blackbox ideology mentioned earlier, we will also ensure that the code is accessible and more readable than that of existing Machine Learning APIs.

Before delving into the theoretical aspects of this project, it is important to note that we assume the reader is already familiar with concepts such as backpropagation in feed-forward networks and the use of schedulers for optimizing gradient descent. If these concepts are unfamiliar, we highly recommend reading our previous paper, which provides a detailed explanation of implementing a feed-forward network, including backpropagation and how it is affected by the use of various activation and cost functions. The paper can be found at: Neural networks for solving regression and classification problems.

3 Theory

3.1 Description of structure

When building a standalone neural network from scratch, a common approach is to encapsulate the code within a single method or class. Typically, the architecture is defined by providing a list of integer values, where the length of the list indicates the number of hidden layers, and the values within the list specify the dimensions of each layer. While this approach is acceptable, it can sometimes be counterintuitive and prone to errors. In our implementation, we opt for a more organized approach by dividing the code into manageable modules called ”layers.” These layers serve as explicit representations of the individual layers within the neural network.

Similar to TensorFlow, we instantiate these layers, which automatically add them to a list in the order they are created. This allows for easy comparison with the functionality provided by existing machine learning libraries. The list containing our layers is stored within a container class called **CNN**. Additionally, the **CNN** class includes additional functionality for controlling all layers simultaneously.

Having discussed the general structure of our code, we will now describe each type of layer that we will be using and explain their functionalities.

3.2 Fully connected layer

3.2.1 Forward pass

The fully connected layer, as the name suggests, represents a standard layer type commonly found in feed-forward neural networks. In the feed-forward mode, this layer takes a batch of data as input and performs a batch-weight matrix multiplication. The resulting product is then passed through an activation function, which elevates the weighted input into a higher-dimensional encoding. This process enables the network to learn arbitrary relationships between the input data's features. The key difference is that instead of using a for loop to iterate over each layer, we explicitly pass data from one layer to another.

Mathematically, we can define the action of this layer using the following equation:

$$a_i^l = f^l(z_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l\right). \tag{1}$$

Where the variables are defined as follows: a_i^l represents the activation of the i-th neuron in layer l, $f^l(\cdot)$ denotes the activation function specific to layer l, z_i^l is the weighted sum of inputs for neuron i in layer l, w_{ij}^l represents the weight connecting neuron j in layer $l-1$ to neuron i in layer l, a_j^{l-1} is the activation of neuron j in layer $l-1$, and b_i^l is the bias term for neuron i in layer l. This equation captures the essential computation performed by the fully connected layer, where input data is transformed into a higher-level representation using a combination of weighted sums and activation functions, and then propagated forward to the next layer.

3.2.2 backward pass

When it comes to backpropagation, the principles remain the same, and only our method of encapsulation has changed. By applying the chain rule repeatedly, we can iteratively calculate the gradient for each layer, starting from the output layer and working our way backward. This process, known as reverse mode of automatic differentiation, is computationally preferable over forward mode, where we begin at the input layer [**?**, 416]. The equations used for gradient calculation in a feed-forward neural network (FFNN), which we will not derive here, are as follows [**?**]:

First, we calculate the delta terms for all layers, beginning with the output layer:

$$y_i^l = f^l(u_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right). \tag{2}$$

We then utilize this term in an iterative process, moving backward through the network with the following equation:

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l). \tag{3}$$

Using these delta terms, we can compute the gradients for the weights and biases in each layer:

$$w^l_{jk} \leftarrow= w^l_{jk} - \eta\delta^l_j a^{l-1}_k,\ b^l_j \leftarrow b^l_j - \eta\frac{\partial C}{\partial b^l_j} = b^l_j - \eta\delta^l_j. \tag{4}$$

These equations demonstrate how the gradients are updated for the weights w^l_{jk} and biases b^l_j in each layer, where η represents the learning rate, δ^l_j is the delta term for neuron j in layer l, and a^{l-1}_k denotes the activation of neuron k in the previous layer. By iteratively updating these gradients, we optimize the network's parameters and enhance its ability to learn and make accurate predictions.

3.3 Convolution

In order to construct a convolutional neural network (CNN), it is crucial to comprehend the fundamental principles of convolution and how it aids in extracting information from images. Convolution, at its core, is merely a mathematical operation between two functions that yields another function. It is represented by an integral between two functions, which is typically expressed as:

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{5}$$

Here, f and g are the two functions on which we want to perform an operation. The outcome of the convolution operation is represented by $(f * g)$, and it is derived by sliding the function g over f and computing the integral of their product at each position. If both functions are continuous, convolution takes the form shown above. However, if we discretize both f and g, the convolution operation will take the form of a sum between the elements of f and g:

$$(f * g)[n] = \sum_{m=0}^{n-1} f[m]g[n-m] \tag{6}$$

The key idea we utilize to extract the information contained in an image is to slide an $m \times n$ matrix *g* over an $m \times n$ matrix *f*. In our case, *f* represents the image, while *g* represents the kernel, oftentimes called a filter. However, since our convolution will be a two-dimensional variant, we need to extend our mathematical formula with an additional summation:

$$(f * g)[i, j] = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f[m, n]g[i-m, j-n] \tag{7}$$