Introduction

NN (perceptron) depuis (60-70?) aujourd'hui, bc d'applications, CNN Notre but est alors de comprendre et savoir comment mettre en oeuvre, NN qui s'execute dans un temps raisonnable

Table of contents

1 Classical Neural Networks (Multilayer Perceptron)

NN can be used for solving a wide array of problems in computer science, arguably the most common of which lies in computer vision. As such, for the purpose of demonstration, we will explain the structural motivation of a typical NN, the Multi-Layered Perceptron (MLP), in the context of computer vision.

Say for instance you have an image containing an animal and you want to identify what animal it is, or to classify the image using a NN. The network itself can then be considered a function with a number of inputs equal to the number of pixels in the image times three in the case of an RGB. The number of outputs is in this case the number of animal types. This is obviously an extremely complicated function and finding the exact function is near impossible. A more intelligent approach is to approximate f with composition of several easily computed functions, such as linear functions (affine) (Figure 1). The structured is loosely based on neuroscience, hence neurons, weights, biases. A simple network might consist of two such layers with Y number of neurons each. This means (Y) number of adjustable parameters! With the structure defined, we have yet to determine the parameters. This is where the training phase enters.

1.1 Training phase

The idea is simple. We start out with a large dataset with classified examples, called a training set and we adjust our network to fit these data. The hope is then that the network will be able to classify new images with these same parameters. In order to quantify the performance of the NN during the training phase, we introduce a cost function. Knowing the desired output with the corresponding input, we look to minimize this cost function by adjusting the parameters of the NN i.e. its weights and biases. This can be seen as an optimization problem with thousands of inputs (in our case, Y weights and biases). For demonstration purposes, we will explain the concept of gradient descent, an intuitive and common optimization algorithm.

1.2 Optimization using gradient descent

We have a function f that we want to minimize while avoiding calculating all its possible values. The gradient of f is an operation that tells us which inputs to adjust and by what weight to adjust them in order to increase f by as much as possible in a given point. Visually, it can be considered the direction in the input space that increases the output the most. In our case, we want to decrease f by as much as possible so, assuming f is continuous, we take the negative value of the gradient. As such, the gradient descent algorithm consists of, at each iteration, calculating f and gradf, adjusting the input by "stepping in the direction of gradf" (Figure 2) and repeating until we have found a local minimum or until the value of f is sufficiently low.

In the context of training a neural network, we have yet to introduce a way to calculate this crucial gradient of the cost function. To do so, we first need to determine a mathematical expression of the neural network itself. In fact, the network typically consists of more than just linear functions.

1.3 Activation function

There are a few different motivations behind an activation function. Firstly, it introduces non-linearity to a problem that is manifestly not linear. For optimization purposes, we also want this function to be continuous. Better still if we know the expression of its derivate. Both the sigmoid function defined as _ and the arctan function have the desired properties, though the simple ReLU (Figure 3) is shown to be more efficient for optimization (source) and thus more common in practice.

1.4 Softmax

Before evaluating a loss function, we transform the output layer in a way that their values are positive and their sum is equal to 1. In this way, each value in the output layer can be interpreted as the probability that a given input is classified as such. While in the training phase, we obviously know the desired output to each input, thus we want that specific "neuron" to have a value close to 1, and all the others to be close to 0. The loss function, which quantifies "how close" the network's guess is to a given output, can have different definitions depending on the problem. A common loss function is the so called ilogit defined as _ . In order to quantify how well the network performs on the entire training set, we want to evaluate the loss function on each data-input. The cost function can then be defined as _. Or the average of losses.

1.5 Forward propagation

We now have an expression for each component in the neural network. Let's see how they are assembled in the forward propagation (Figure 4).

1.6 Backward propogation

Backward propagation, finding the gradient. As previously mentioned, the NN can be seen as a composition of functions of several functions of which we can easily find the derivate. Intuitively, in order to find the gradient of the cost function with respect to the weights and biases, we can just apply the chain rule (Figure 5). As we can see, the output layer is dependent on its preceding layer, weights and biases, which, in turn, depends on the previous parameters. This creates a cascade of gradients calculated from the output- to the input layer, hence the name backward propagation.

After iterating over the training set, we now have the gradients needed for the optimization algorithm, which in turn allows us to adjust the weights and biases in order to, iteration by iteration, reach a local minimum. In practice, this approach performs relatively well, though there are improvements to be made. In the following chapters, we will examine how convolutional neural networks differs from the classic MLP.

2 *Convolutional* Neural Networks

A convolutional neural network is an evolution of a classical multilayer percepetron network. Recall the basic principle underpinning how a normal neuron in a neural network is supposed to work. The neuron is supposed to 'look' for features in its input data. If the neuron 'thinks' that those features are present in the input data it 'fires'. Otherwise the neuron does not fire.

In a classical multilayer percepetron network this is implemented in the following way. Each neuron contains a vector of weights, a bias and an activation function. The input to the neuron—which must be a vector of equal length to the neuron's own weight vector—is combined with the neuron's weight vector using the dot product. The neuron's bias is added onto the result which in turn is passed to the activation function which determines if the neuron 'fires' or not.

Using several layers of neurons one can achive quite remarkable results using this implementation of a neural network. However, a multilayer percepetron is inherently limited. The major problem is that neurons in these kinds of networks only accept input that is in the form of a vector. This means that for applications where it is not natural for the input to be in a vector format, say image recognition, the input first has to be translated to a vector format. Usually this results in a loss of information contained in the input. In the typical case of image recongition, the input is in the form of one or more arrays of two dimentions. For a multilayer percepetron to treat this input, the images has to be 'flattend' into a vector of one dimention before it can be passed on to the network. This procedure eliminates some of the pixel relations in the image. To deduce this, consider the process of reconstructing a flattend image. If the image's dimentions prior to being flattend is not known, it is impossible, without the aid of pattern recognition, to reconstruct the image and be sure the reconstruction is equal to the original image.

To remedy this problem, there is a simple solution. Instead of having the neuron contain a vector of weights, let it have an array of weights. Changing the neuron's vector of weights into an array of weights also necessitates a change in the operation used to combine the weights with the input (which in a multilayer percepetron is the dot product). There are two things to consider here. The purpose of the weights is to look for features or *patterns* in the input and the operation must reflect this purpose of the weights. Furthermore, the result of the operation should be a single number which, in a sense, representes the neurons 'initial' confidence that the feature is looking for is present in the input. The operation which does both of these things is the *Hadamard product*. The Hadamard product can be viewed as an extension of the dot product to two dimentional arrays. It combines two arrays—of the same dimentions—by multiplying corresponding entries together and summing the results. Which is precisly what the dot product does with two vectors.

We can even take this a step further to allow the neuron to treat inputs of not only a single two dimentional array but several two dimentional arrays. A typical example where the input would consists of several interlinked two dimentional arrays is RGB images. An RGB image consists of three arrays of pixel values (numbers) that desribe how red, green and blue an image is in each pixel. The number of two dimentional arrays present in the input is known as the number of *channels* the input has. In order for our neuron to treat inputs with more than one channel we let the neuron have as many channels as the input. That is to say, wwe equip the neuron with as many weight arrays as there are channels in the input. The weight array in each channel is combined with the inputs array in the same channel using the Hadamard product. The results of

these individual Hadamard products is then combined to form the 'final' Hadamard product—the neurons 'initial' confidence that the feature is looking for is present in the input.

Using this slightly more complicated implementation of a neuron, the neural network is able to treat inputs of one more two dimentional arrays directly (for each array present in the) A network which contains layers of these kinds of neurons, is a convolutional neural network.

A convolutional neural network implements the very same idea, a neuron which looks for features and decides to fire, but with a more complicated implementation. The benefit of this more complicated implementation is that it allows the network to preserve spatial information in its input. Consider again a normal multilayer percepetron. These networks work with inputs that are vectors of numbers. In order for these networks to accept inputs that are multi-dimentional arrays, rather than one-dimentional vectors, the input first has to be 'flattened' before it is passed on to the network. Using the MINST database of handwritten digits stored as 28x28 grayscale images as an example, for a multilayer percepetron to process such an input the image first has to be flattend to a $28 \cdot 28 = 784$ long vector.

A 'convolutional neuron' does not have this limitation, it can treat a multidimensional input directly. The idea is to replace the simple neuron of a multilayer percepetron—made up of a weight vector and a bias—with a 'kernel' or 'filter'. The kernel contains a multidimensional array of weights, instead of a simple vector, and a bias. Instead of 'looking' at a vector of inputs, the kernel can look at an array of inputs which allows it to 'look' for features that can only be properly described by a full array rather than a flattened vector. Necessarly, such a 'convolutional neuron' requires a different mathematical operation to do the 'looking for feature' job of the neuron. The operation that is used is a convolution hence the name, convolutional neural network.

2.1 The convolutional layer

A convolutional layer consists of three parts: the input (a multidimensional array), the kernel (also a multidimensional array with a bias term) and the output which is called a *feature map*. The layers forward operation is

$$\text{feature map} = \text{Convolution}\,(\text{input}, \text{kernel}) + \text{bias}$$

Figure 1 on page 6 illustrates how the feature map is calculated using the kernel and a input. The kernel 'scans' the input and produces a number based upon how confident it is that a certain feature is present at the area it is looking at. The bias is added onto this number which produces the final number placed in the feature map.

In the case where the input has multiple channels—as is the case in Figure 1—each input is a sequence of arrays. The kernel has to an have equal amount of channels as its input. Each channel is scanned seperately and their values are added together.

Let's us derive the proper mathematical formulation for calculating the feature map. The feature map is a two dimentional array which we will denote F, the individual entries are dentoed $f(i, j)$ where i is the row and j the column. The kernel is a multidimensional array which we will denote K, the individual entries are denoted $K(i, j, c)$ where c is the channel. The input is, same as the kernel, a multidimensional array which we will denote M, the individual entries are denoted $M(i, j, c)$. The horisontal lengths and vertical lengths of the kernel and input will be denoted using a h and v prefixes (e.g. hK is the horisontal length of the kernel). A typical MINST image of dimentions 28x28 will have $hM = vM = 28$. The number of channels in the input and kernel will be dentoed by nC. The bias term is denoted b. As we can see from Figure 1, the formula for an individual entry in the feature map is

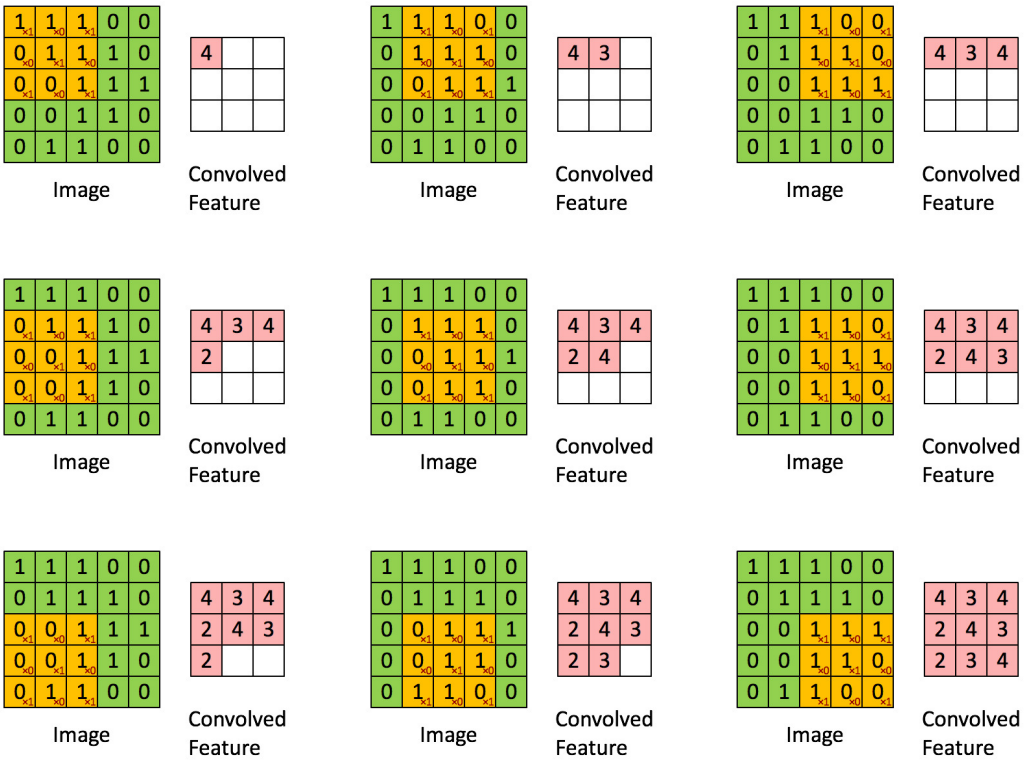

Figure 1 The basic forward operation of a convolutional layer

$$f(i', j') = \sum_{c=1}^{nC} \sum_{j=1}^{vK} \sum_{i=1}^{hK} \Big(K(i, j, c) \cdot M(i' + i, j' + j, c) \Big) + b$$

Go on ...

2.2 Multiple channels

2.2.1 Altering the stride

What the stride is and the effects of altering it.

2.2.2 Zero padding

2.3 The pooling layer

Downsampling and purpose of downsampling. Noise reduction and computational reduction.

2.4 Backward Propagation

Obtain formula for derivative of convolution and pooling.

3 Implementation

- The problem the network is going to tackle (classic MINST image recognition)
- The architecture of the network
- Choice of various functions and their associated code
- Results with time performance
- Possible Modifications

4 Conclusion

The advantages of a CNN over a NN.