Conv Net

Francesco Morri

February 7, 2020

1 Introduction

The aim is to build a convolutional neural network that will be able to classify images. This task is *data driven*, i.e. we need a large amount of data to work with and to let the network learn. We will start with basic classifiers and we will build up to a proper CNN (convolutional neural network).

1.1 k-Neighbour

This classifier is fairly simple, nonetheless is a good start. Given a set of labelled images (like the CIFAR-10, 50000 images labelled and 10000 for testing) we compute the difference between an image from the test set and all the images from the train set, the we select the one that return the smallest value. That will be the prediction for the test image.
There are two main functions that can be used as *distance*:

- **L1**: $d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$
- **L2**: $d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$

where I_i^p is an element of the array that represents the image.
The description above refers to the Nearest Neighbour classifier, that finds the single closest element from the training set. This classifier works pretty bad, since it is likely that there is an image of the wrong class that is close (color-wise) to the one we are classifying.
It is possible to improve the algorithm selecting not one, but k nearest neighbours. The value of k is a hyperparameter, which means that it is not possible to know the best value a priori, some testing is then needed.

1.2 Cross-Validation

It is important to test these hyperparameters on a different set form the training set, since we would end creating an algorithm that performs well only on our set (this is called overfitting). It is always good practice then to divide the training set in two parts, not of equal size: usually it is $\frac{1}{5}$ for the validation set and the rest for training. This is really important not only for this classifier, but in general for all kind of application of machine learning.

1.3 Linear Classifier

1.4 Gradient Descent

We need now a way to select the best possible set of weights W, which will minimize the loss function. This process of searching the best parameters is called **optimization**. Since we need to change W, the first (and worst) idea is a ***random search*** for the best set. This way we end up with a better

set than the starting one, but it is time-expensive procedure and it does not produce a good result. We could improve the search using *little steps* instead of completely random change, meaning that we change the set W by a little amount and check if it is better than before, if it is we continue, if not we get back at the starting point.
The best way to optimize the set is following the gradient, that can give us the direction of max increase (if we take that negative is going to be the direction of max decrease). This method is called **gradient descent** and it is the standard way for optimize a neural network.
To compute the gradient we can either do it numerically (more inefficient, but less prone to errors) or using calculus (more efficient, but more prone to errors!). It is common practice to verify the formula obtained from calculus with the value returned from the numerical implementation (***gradient check***).
Once we have computed the value of the gradient it is pretty easy to update the weights. Since the training set may be quite big (order of millions of elements), it is best to compute the gradient of batches of the set. Doing so, we will not lose any valuable information and we will reduce considerably the computing time.

1.5 Chain Rule and Backpropagation

In a neural network we work with functions with many parameters linked together. We have then to derive the derivatives of this multi-paramaters function with respect of every parameter.
It is important to recall the chain rule, that will help us *chain* together all the gradients through multiplication. A quick example:

$$f(x,y,z) = (x+y)\cdot z \qquad q(x,y) = (x+y)$$

$$\frac{\partial f}{\partial q} = z \qquad \frac{\partial f}{\partial z} = q \qquad \frac{\partial q}{\partial x} = 1 \quad \frac{\partial q}{\partial y} = 1$$

and using the chain rule we can easily see that $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\cdot\frac{\partial q}{\partial x}$.
When we compute the output of a net we are going ***forward***, while when we need the gradient we work the other way around, going ***backward***. This is the core principle of backpropagation: during the forward phase we compute the output and the value of the gradient in every *neuron*, we then have the final output and during the backward phase we just need to proceed in the opposite direction and use the gradient value instead of the output. A graph will make everything clearer (Fig. 1).

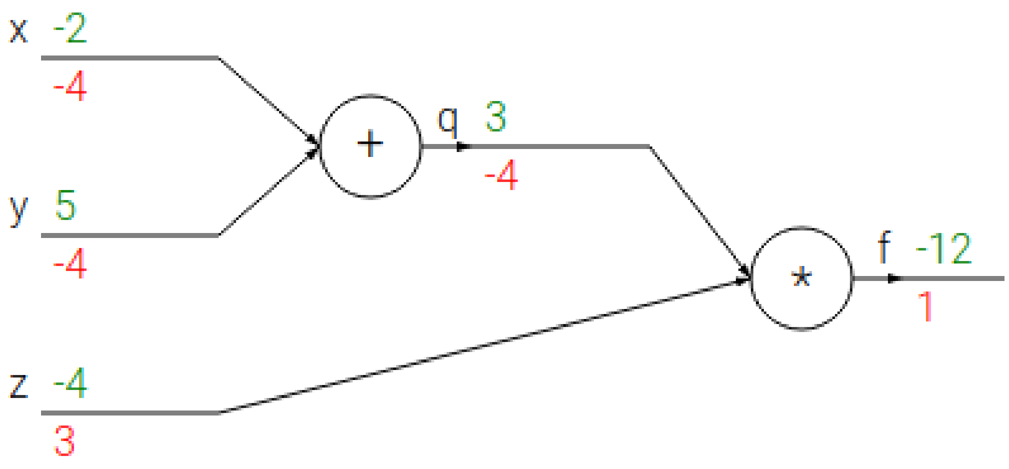

Figure 1: Graphic representation of the chain rule

2 Neural Network Architecture