A.A.
2018/19

Convolutional Neural Networks

WHAT'S IN THE BLACK BOX?

ANDREA GAFFURI RIVA & MARCELLO SBORDI

UNIVERISTÀ DEGLI STUDI DI MILANO BICOCCA

Index

1. Neural Nets
    1.1. Architecture
    1.2. Feed forward
    1.3. Back Propagation
    1.4. Technical issues
    1.5. An interpretation of the model
2. From Neural Nets to ConvNets
3. ConvNets architecture
    3.1. Convolutional layers
    3.2. Pooling layers
4. Neural Nets vs ConvNets
5. References

1. Neural Nets

The term neural network (NN) is used to indicate a broad class of models and machine learning methods. The first NN introduced here will be the basic type formed by the input layer, a hidden layer and the output layer; more complex NNs based on this structure are just a generalization given by the addition of hidden layers between the input and output one.

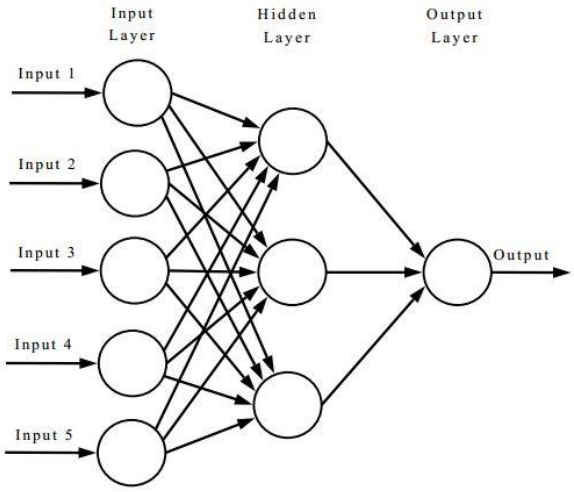

Figure 1 Neural network basic architecture

1.1. Architecture

A NN is a two (or multiple) stage regression or classification model, typically represented by a diagram as in Figure 1. Both regression and classification share the same structure for the NN. In general, the data are inputted in the first layer, the hidden layer computes a linear combination of the inputs and then the output layer returns the results (e.g. the probability of the input belonging to each class).

Mathematically, the NN is descripted by these formulae:

$$z_h{}^{(l)} = w_{h0}^{(l-1)} + \sum_{j=1}^{p_{l-1}} w_{hj}^{(l-1)} a_j^{(l-1)}$$

$$a_h^{(l)} = g^{(l)}\left(z_h{}^{(l)}\right)$$

$$a_h^{(1)} = x_h \; and \; p_1 = p$$

Where:

- l indexes the layers;

- h indexes the neurons in the l^{th} layer;
- j is the number of neurons in the $(l-1)^{th}$ layer.

A more compact notation is obtained by using vector notation:

$$z^{(l)} = W^{(l-1)}a^{(l-1)}$$

$$a^{(l)} = g^{(l)}\left(z^{(l)}\right)$$

Where:

- $z^{(l)}$ represents the linear combinations;
- $W^{(l-1)}$ represents the weights matrix (including the intercepts $w_{l0}^{(l-1)}$) that goes from layer $l-1$ to l;
- $a^{(l)}$ is the vector of neurons activations at layer l;
- $g^{(l)}$ operates elementwise on its vector argument.

$g^{(l)}\left(z^{(l)}\right)$ is called the activation function and it is used for two main purposes, which are:

- determine the values within each neuron (i.e. activating the neuron);
- compute non-linearity in the model.

Because $g^{(l)}\left(z^{(l)}\right)$ is a function that takes a linear combination of the previous layer $z^{(l)}$ as its only argument, if we choose the identity function, each neuron becomes, simply, a linear combination of the previous layer, so the model can't compute non-linearity.

The output function $g^{(L)}$ allows a final transformation of the vector $z^{(L-1)}$ to make it useful for prediction. In case of regression is commonly used the identity function because there is no need to further transform the results; instead, for K-class classification problems, it is useful to have probabilities that the inputs belong to each class. So, for this purpose, the logistic regression transformation (SoftMax) is the most used:

$$g^{(L)}\left(z_k^{(L)}; z^{(L)}\right) = \frac{e^{z_k{}^{(L)}}}{\sum_{j=1}^{K} e^{z_j{}^{(L)}}}$$

The units in the middle of the network, computing the derived features $a^{(l)}$, are called hidden units because the values $a^{(l)}$ are not directly observed. The $a^{(l)}$ could be interpreted as the basis expansion of the original inputs X, the NN is then a standard linear model, or linear multilogit model, using these transformations as inputs. There is, however, an important

difference over the traditional basis expansion techniques; here the parameters of the basis functions are learned from the data.

From another stand point, an interpretation of what a NN is doing, is given by looking at the activation function. If the $g^{(l)}$ is the identity function, the entire model collapses to a linear model in the inputs. Hence a NN can be thought of as a nonlinear generalization of the linear model, both for regression and classification.

Finally, a consideration on the name "neural network". The term originated from the fact that they were first developed as models for the human brain. Each unit represents a neuron and the connections the synapses. In early models, the neurons fired when the total signal passed to that unit exceeded a threshold. This corresponds to the use of a step function for activation and output functions. Later these models were recognized as a possible useful tool for statistical modelling and so smoother functions were chosen to simplify and improve optimization.

When fitting the NN there are two stages: feedforward, the process of going from the input to the output, and the backpropagation, the process of calculating gradients from the output to the input via chain rule, to upgrade the weights of the linear combinations.

1.2. Feedforward

To do the feedforward, all we need to do is to take the inputs and, via the formulae previously shown, calculate all the activations for all the neurons and compute the output of the network. After this passage we must evaluate the performance of the model, so we use a loss function.

When fitting a NN we are solving this optimization problem:

$$\min_{\mathcal{W}} \left\{ \frac{1}{n} \sum_{i=1}^{n} \Lambda[y_i, f(x_i; \mathcal{W})] \right\}$$

The issue is that we cannot just compute $\frac{\partial \Lambda[y_i, f(x_i; \mathcal{W})]}{\partial W^{(l)}} = 0$, since we only know that loss functions are usually convex in f but not in the elements of $\mathcal{W}$; this brings us to compute a minimum that is most probably a local one and not a global one.

This problem has the solution in the process of backpropagation.

1.3. Backpropagation

Backpropagation is the process of going backward up the NN and compute all the derivatives of the loss function with respect to the weights via chain rule. We would like to compute an error term that measures the responsibility of each node for the error in predicting the output. If we take a generic training pair (x, y) the formulae to compute the gradients are the following:

- for each output unit in the L[th] layer

$$\delta_h^{(L)} = \frac{\partial \Lambda[y, f(x; \mathcal{W})]}{\partial z_h{}^{(L)}} = \frac{\partial \Lambda[y, f(x; \mathcal{W})]}{\partial a_h{}^{(L)}} \dot{g}^{(L)}\left(z_h{}^{(L)}\right), where\ \dot{g} = {}^{\partial g}/_{\partial z}$$

- for layers $l = L - 1, \dots, 2$ and for each neuron in them

$$\delta_h^{(l)} = \left(\sum_{j=1}^{p_{l+1}} w_{jh}^{(l)} \delta_j^{(l+1)}\right) \dot{g}^{(l)}\left(z_h{}^{(l)}\right)$$

- finally, the partial derivatives are given by

$$\frac{\partial \Lambda[y, f(x; \mathcal{W})]}{\partial w_{hj}{}^{(l)}} = a_j^{(l)} \delta_h^{(l+1)}$$

Also, in this case, the matrix-vector notation helps in making these formulae more readable:

$$\delta^{(L)} = \dot{\Lambda}\left(y, g^{(L)}\left(z^{(L)}\right)\right) \circ \dot{g}^{(L)}\left(z^{(L)}\right), where\ \dot{\Lambda} = {}^{\partial \Lambda}/_{\partial g(z)}\ and \circ\ the\ Hadamard\ product$$

$$\delta^{(l)} = \left(W^{(l)^T} \delta^{(l+1)}\right) \circ \dot{g}^{(l)}\left(z^{(l)}\right)$$

$$\frac{\partial \Lambda[y, f(x; \mathcal{W})]}{\partial W^{(l)}} = \delta^{(l+1)} a^{(l)^T} = \Delta W^{(l)}$$

At last, we update the parameters with a function of the old weights, the derivatives we have just computed and the learning parameter (a tuning parameter)

$$W^{(l)} \leftarrow u\left(W^{(l)}, \lambda, \Delta W^{(l)}\right)$$

The processes of feedforward and backpropagation are iterated many times, typically until we reach the minimum of the loss function.

1.4. Technical Issues

In this chapter we would like to present the main mathematical issues related to the activation function and to the backpropagation process.

First, we will take care of the activation function $g^{(l)}$.

A first definition of an activation function could be: “something that receives an input and computes an output, which could be 0 or 1 with respect to some kind of threshold”.

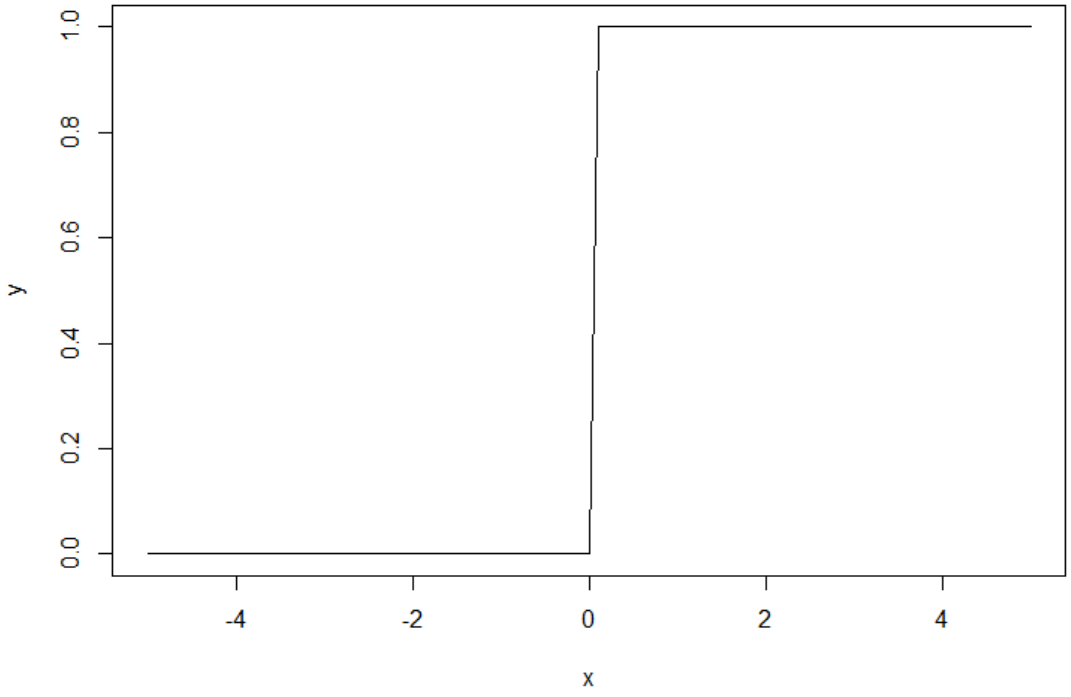

Figure 2 Plot of a generic activation function

This function is defined as:

$$f(x) = \begin{cases} 0, & x < c \\ 1, & x \geq c \end{cases}$$

With c used as a threshold (in Figure 2 we choose $c = 0$).

But if we use a function like this as our $g^{(l)}$ when we try to use the backpropagation and compute $\dot{g} = {}^{\partial g}/_{\partial z}$ we can see that $\dot{g}$ = 0 so the $\Delta W^{(l)}$ wrt to the l-th layer becomes zero for all possible values of l.

So, we need to use a different kind of function for the activation of the neurons, which should be like the previous function and it should fix the problem in the backpropagation process.

Here we can see some example of the most common use function for $g^{(l)}$:

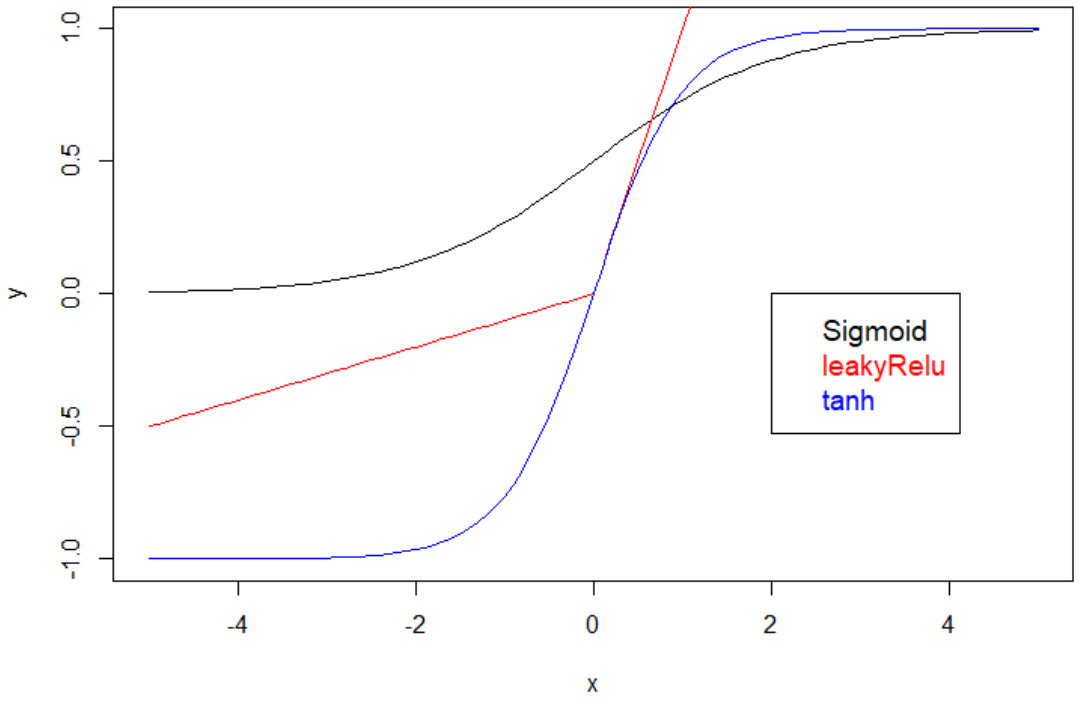

Figure 3 Plot of most used activation function

The functions in Figure 3 are:

- sigmoid: $\sigma = {}^{1}/_{1+e^{-x}}$;
- leakyReLU: $lReLU = \max(\gamma x, x)$, in our case $\gamma = 0.1$;
- hyperbolic tangent: $tanh = {}^{e^{x}-e^{-x}}/_{e^{x}+e^{-x}}$.

As we can see from the plot, these functions, by being continuous, fix the problem of not being easily optimizable and, therefore, we can apply backpropagation.

When we choose one of these functions we should consider that:

- if $\dot{\sigma} = {}^{\partial\sigma}/_{\partial x}$ then for $x \to \pm\infty$ we have $\dot{\sigma} \to 0$ and $0 < \dot{\sigma} < 1$;
- if $\dot{tanh} = {}^{\partial tanh}/_{\partial x}$ then for $x \to \pm\infty$ we have $\dot{tanh} \to 0$ and $0 < \dot{tanh} < 1$.

Because we are using the chain rule to obtain the gradient wrt $W^{(l)}$ and as we've seen before $0 < \dot{\sigma} < 1$ and $0 < \dot{tanh} < 1$ and both these functions saturate the gradient for $x \to \pm\infty$. To compute the gradient of the first layers, we multiply these small numbers about l times, meaning that the gradient decreases exponentially and these layers train very slowly; this effect is known as the 'vanishing gradient' problem.

The leaky ReLU function can partially solve this problem; if we compute $\dot{lReLU}$ we can see that $\dot{lReLU} \in (\gamma; 1)$, so, if we choose for example $\gamma = 0.5$, we can partially fix this problem.

Now we can take care of the update function in the backpropagation process.

As we've seen before, in general the update is represented as following:

$$W^{(l)} \leftarrow u\left(W^{(l)}, \lambda, \Delta W^{(l)}\right)$$

A first approach to compute the update of the parameters in the l-th layer hidden layer is called stochastic gradient descent:

$u\left(W^{(l)}, \lambda, \Delta W^{(l)}\right) = W^{(l)} - \lambda \Delta W^{(l)}$

This approach has the advantage of being very simple. However, the convergence rate is quite slow and so it cannot solve, even partially, the vanishing gradient problem.

Another approach, more used nowadays, is the accelerated gradient descent:

$$V_{t+1} = \mu V_{t+1} - \alpha(\Delta \mathcal{W}_t + \lambda \mathcal{W}_t)$$

$$\mathcal{W}_{t+1} = V_{t+1} + \mathcal{W}_t$$

Using $\mathcal{W}_t$ to represent the entire collection of weights at iteration t, V_{t+1} is a velocity vector that accumulates gradient information from previous iterations and it is controlled by an additional momentum parameter μ.

The accelerated gradient descent can achieve much faster convergence rates than the stochastic gradient descent and can partially fix the vanishing gradient problem.

1.5. An interpretation of the model

The NN can generalize most of the model that are used for prediction. Here we present some examples:

- suppose we have a regression task to solve, if we use a squared error loss with no regularization as a loss function, an identity function for the activation of the neuron and for the output function, only one hidden layer with one neuron, we obtain a linear regression model;
- if we keep the same features of the NN above except for the loss to which we add an L_2 regularization we obtain a ridge regression model.

In these terms the NN is a model that generalizes most of the models used for regression and classification tasks.

Now suppose that we have set the loss function and the activation function to solve a specific task (i.e. we have fixed our model to predict our observations), we can interpret the neural nets like an algorithm that computes feature engineering on the input layer (on the variables that describe the model) which minimizes the loss function. So, in these terms, the NN is a feature engineering algorithm.

2. From Neural Nets to ConvNets

The idea of the convolutional neural networks (CNN) comes from a study of Wiesel and Hubel in the 50's and 60's regarding the cats' and monkeys' visual cortex. They noticed that cortical neurons respond to stimuli in a restricted area of the visual field and the receptors partially overlap to cover the entire field.

These discoveries led, in more recent days, to rethink the architecture of the NN in a way that could preserve the 3D aspect of the input, crucial to image recognition (a common task for neural networks in general). The major advantage of this approach is that the data needs close to zero pre-processing compared to other image classification algorithms.

Here we will not proceed to discuss the fitting of the CNN because feedforward and backpropagation are the same as for the classic NN and we would like to focus more on the architecture which is the defining feature of this model compared with other neural nets in general.

3. ConvNets Architecture

The CNN is usually formed by a series of convolutional/activation layers alternating pooling (or subsampling) layers; at the end of the actual ConvNet there is a fully connected NN that performs the classification (as shown in Figure 4).

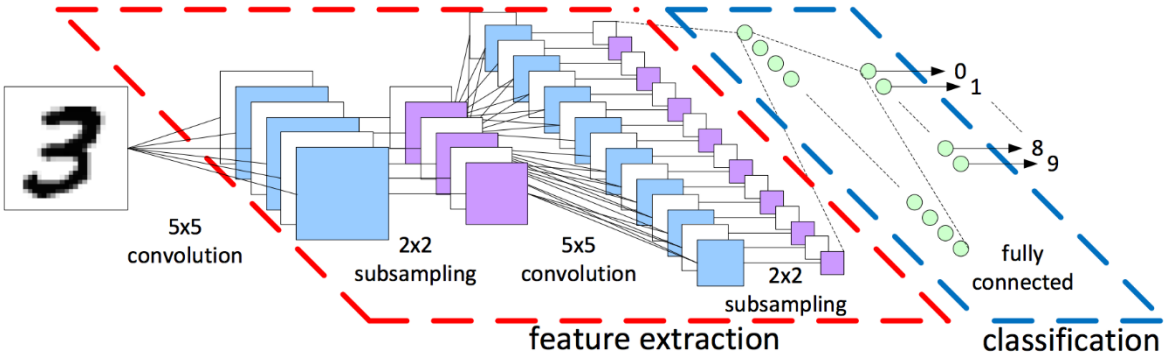

Figure 4 ConvNet basic architecture

3.1 Convolutional layers

In the convolutional layers we apply a filter (an array of three matrices of weights) to the full depth of the input and slide it over the image for its entire dimensions; at each location we compute the dot product and every result of the operation is put into a matrix which is called a feature map. We can apply different filters to each image to generate different maps and to read different aspects of every input; to each neuron of each map it is then applied an activation function (just like the NN neurons). It must be noted that each neuron, differently from standard NN, it is not liked to other neurons in its map or in other maps but only to the region of the input that generated it.

The Figure 5 shows a step of the process of convolution:

1. the filter is placed on the first zone for the full 3-layered depth and the dot product is computed (including a bias or intercept term);
2. the filter is slid by a certain quantity called stride (usually lower than the filter so that each zone partially overlaps, imitating the animal visual cortex);
3. this process is repeated for each filter until all the input image has been covered.

A frame of zeroes (or more than one) is added in certain cases either to obtain a map of the same dimensions as the input image or to fit the filter to the entire image (e.g. we have a 6x6 image and 3x3 filter with stride 2; the filter will not fit to the image completely at the third slide).

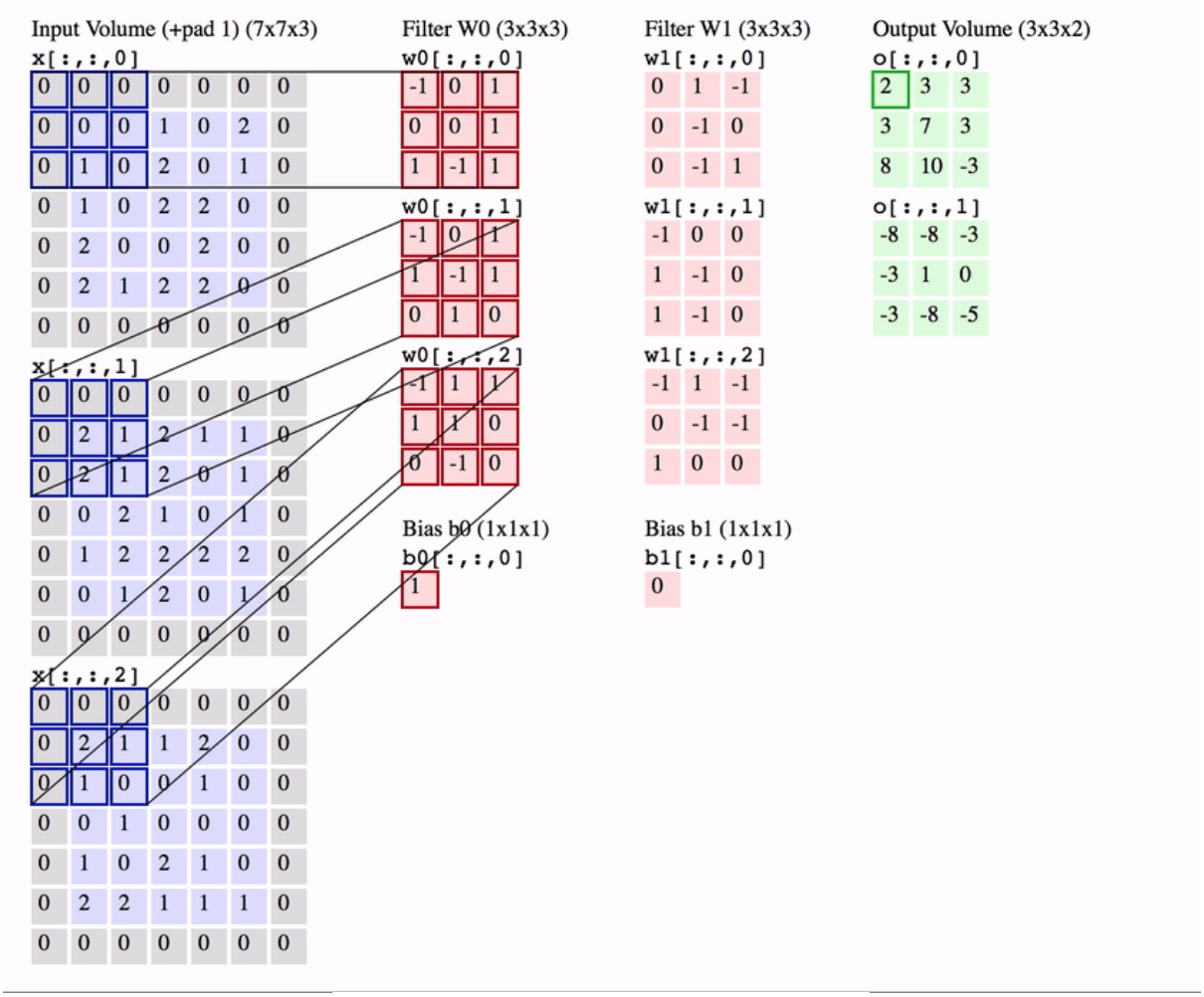

Figure 5 Convolutional layer scheme

3.2 Pooling layers

The main function of pooling layers is to reduce the size of each feature map produced at the previous convolutional layer; this is to reduce computational time of training, since the network is a very complex model and training can take up to several days of continuous computing.

To achieve this result, we consider non-overlapping regions of the input maps and apply a function that reduces the number of parameters.

The most common function is the maximum of the region; other possibilities could be the minimum, the mean, a linear combination.

So, in pooling layers, we do a very similar thing to the one in convolutional ones. There are three key differences:

1. The filters, instead of arrays of matrices, are the function that computes the pooling;
2. The maps are bidimensional (they have no depth);
3. There is no activation function applied to the results of the pooling layers.

4. Neural Nets vs ConvNets

In this final section we would like to briefly describe and show the differences between what a classic NN is seeking in the inputs and what a ConvNet looks for.

In general, when thinking about the classification of an image, we think that our brain perceives some edges and patterns in similar figures and then it puts all the information together to classify the input; since NNs are based on the brain's structure we would expect something similar in their process of learning.

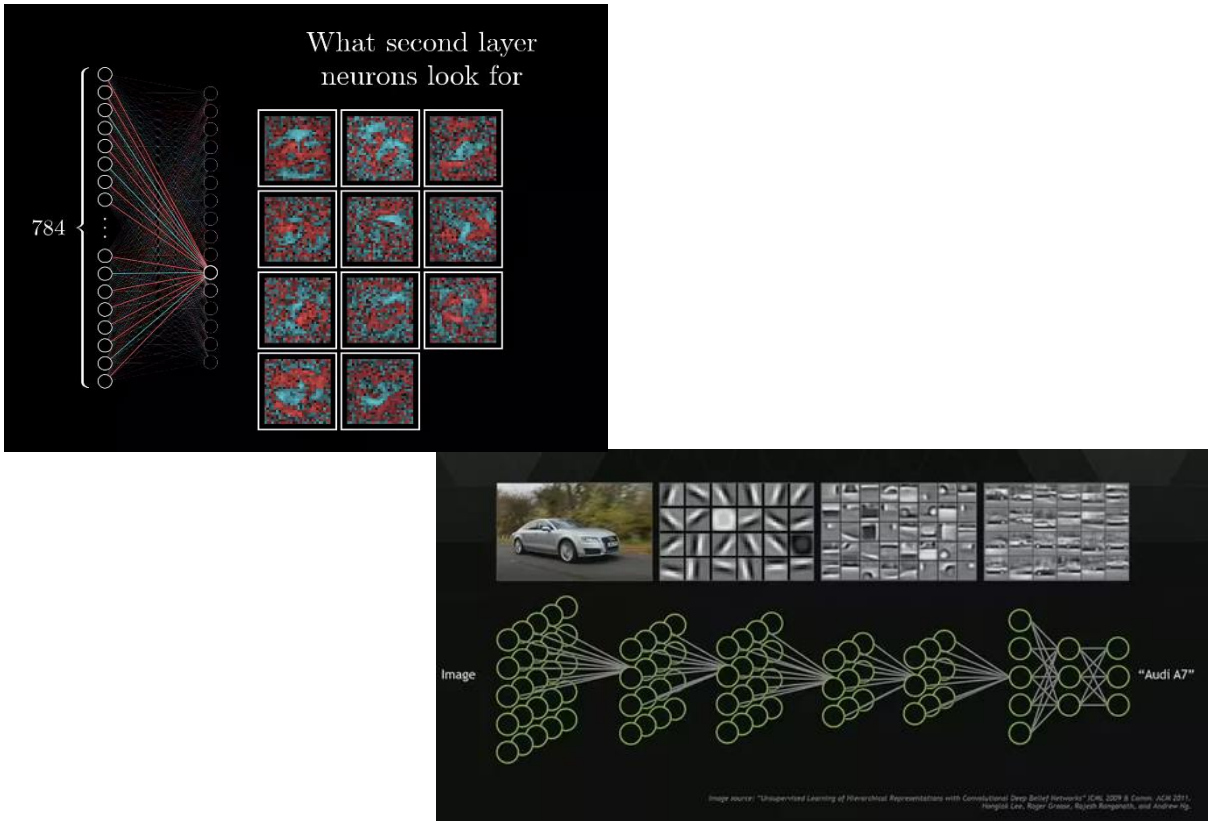

Figure 6 NN vs CNN learning process

Actually, as we can see in Figure 6, a standard NN does not capture edges or patterns but something that looks like totally random.

On the contrary, a CNN, thanks to what is done by convolutional layers, starts by reading simple information such as oriented edges and differences in shades; going down the network, convolutional layers read more complex ensembles of forms like the details of the image (wheels, side, doors) and at the end the entire car.

5. References

- Computer Age Statistical Inference, B. Efron, T. Hastie, 2016
- The Elements of Statistical Learning, T. Hastie, R. Tibhsirani, J. Friedman, 2017
- http://cs231n.github.io/, Stanford University
- You Tube Video Lessons 'Convolutional Neural Networks for Visual Recognition', Stanford University, https://www.youtube.com/playlist?list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv
- You Tube Videos 'Neural Networks', 3Blue1Brown, https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi