MASTER THESIS

MSC IN SIGNAL THEORY AND COMMUNICATIONS

Convolutional Neural Networks.

Author
García-Torres Fernández, Jorge

Professors
Zazo Bello, Santiago

Universidad Politécnica de Madrid, ETSIT

—

Madrid, March, 2018

1. Introduction

Convolutional Neural Networks (**?**), also known as *CNNs* or *ConvNets*, are a powerfull artificial neural network technique for processing data that has a known, grid-like topology (**??**). They are popular since they are tremendously succesfull in practical applications, particularly in processing time series data, which can be thought of as a 1D grid taking samples at regular intervals, and image data, which can be thought of as a 2D grid of pixels (**?**). In general, convolutional networks are simply neural networks or multilayer perceptrons that use a mathematical operator called convolution in place of general matrix multiplication in at least one of their layers.

1.1 Perceptron

A perceptron (**?**) is a single artifical neuron model that was a precursor to larger neural networks. It is a simple computacional unit that have weighted input signals (including a bias) and produce an output signal using tradicionally a nonlinear activation function that allows the network to combine the inputs in a more complex way and in turn provide a richer capability in the function it can model (**?**).

Figure 1: Scheme of a single perceptron. Reference: http://www.techmaru.com/technology/artificial-neural-networks/neural-network-elements.

Perceptrons can be arranged into networks, forming neural networks or multilayer perceptrons (MLP). The MLP architecture contains:

- Input or Visible Layer: Bottom layer that takes input from our data.
- Hidden Layers: Layers that are not directly exposed to the input.

- Output Layer: Layer responsible for outputting a value or vector of values that correspond to the format required for the problem. In the output layer, the choice of activation function is strongly constrained by the type of problem that we are modeling.

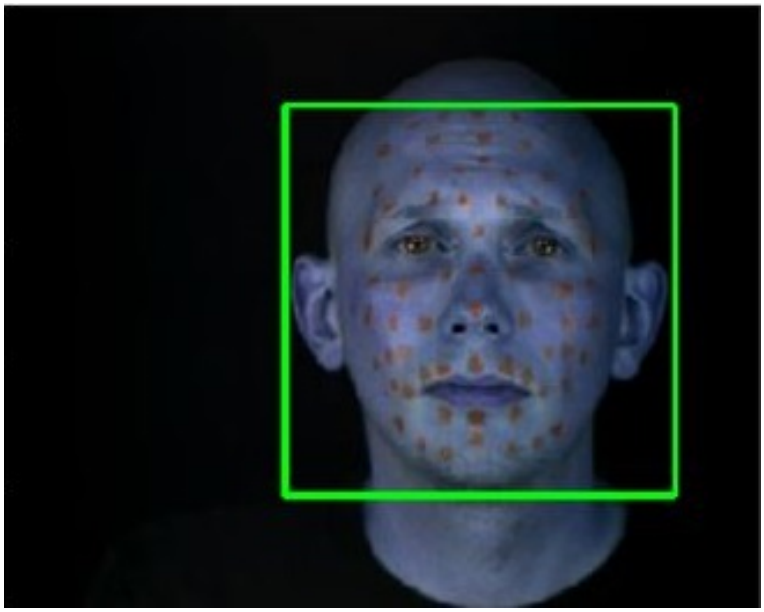

Figure 2: Scheme of a Multilayer perceptron. Reference: https://www.researchgate.net/figure/A-hypothetical-example-of-Multilayer-Perceptron-Network_fig2_273768094.

1.2 Convolution

In case of CNNs, convolution operator is employed instead of full matrix multiplication in at least one layer. Convolution (**?**) is a function that expresses the overlap of one function g as it is shifted over another function f (**?**):

$$s(t) = (f * g)(t) = \int f(a)g(t-a)da \tag{1}$$

and the discrete convolution:

$$s(t) = (f * g)(t) = \sum_{a=-\infty}^{\infty} f(a)g(t-a) \tag{2}$$

In neural networks terminology we refer to the first argument (the f function) as *input*, the second argument (the g funtion) as *kernel* or *filter* and the s output as the *feature map*. In machine learning applications, each element of the input and the kernel must be explicitle stored separately, hence we can assume that these functions are zero everywhere but the finite set of points we store the values. This means that in practice we can impement the infinite summation over a finite number of array elements (**?**).

Then, if we take a two-dimensional input (an image) we probably also want to use a two-dimensional kernel:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{3}$$

Due to the conmutative property of convolution, we can equivalently write:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{4}$$

The commutative property of convolution arises because we have filpped the kernel in the sense that as m increases, the index into the input increases, but the index into the kernel decreases (**?**). This implementation is not usually necessary in neural networks, so many machine learning libraries implement a related function called cross-validation (**?**) without flipping the kernel but even so they call it convolution:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{5}$$

At the end, discrete convolution can be viewed as multiplication in which a small weight matrix is sliding around the original input.

2. Motivation

Convolution leverages three important ideas (**?**):

- *Sparse Interactions* or *Sparse Connectivity*: They use fewer parameters or weights to learn than a fully connected network. Traditional NN layers use matrix multiplication by a matrix of parameters describing the iteration between each input unit and each output unit. If there are m inputs and n outputs, then matrix multiplication requieres $m \times n$ parameters and the algorithms used in practice have $O(m \times n)$ runtime. CNNs connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the *receptive field* of the neuron (equivalently this is the filter size). Since we have limited the number of connections each output may have to k, the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime. Note that the connections are local in space (along width and height), but always full along the entire depth of the input volume (**?**). Furthemore, three hyperparameters control the size of the output volume (**?**): *depth* (the number of filters we would like to use), *stride* (we must specify the stride with which we slide the filter) and *zero-padding* (sometimes it will be convenient to pad the input volume with zeros around the border to control the spatial size of the output volume).

- *Parameter Sharing*: CNNs automatically learn and generalize features from the input domain. They control the number of parameters by using the same parameter for more than one function in a model. In convolutional neural net, each member of the kernel is used at every position of the input. This means that rather than learning a separate set of parameters for every location, we can learn only one set. Although this does not affect the runtime (it is still $O(k \times n)$), it does further reduce the storage requirements of the model to k parameters.

Figure 3: *Sparse connectivity*: We highlight one output unit, s_3, and also highlight the input units in x that affect this unit. These units are known as the receptive field of s_3. (Top) When s is formed by convolution with a kernel of width 3, only three inputs affect s_3. (Bottom) When s is formed by matrix multiplication, connectivity is no longer sparse, so all of the inputs affect s_3. Reference: **?**.

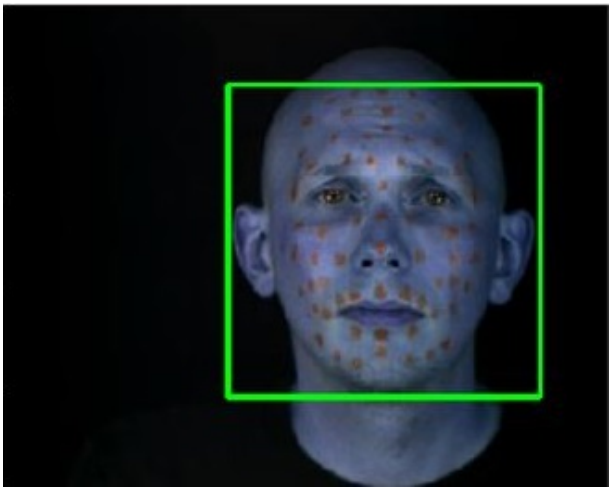

Figure 4: *Parameter sharing*: Black arrows indicate the connections that use a particular parameter in two different models. (Top) The black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Due to parameter sharing, this single parameter is used at all input locations. (Bottom) The single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing so the parameter is used only once. Reference: **?**.

- *Equivariant Representations*: CNNs are designed to be invariant to object position and distortion in the scene. A function is equivariant if when the input changes, the output changes in the same way. Specifically, a function f is equivariant to a function g if $f(g(x)) = g(f(x))$. In the case of convolution, if we let g be any function that translates the input, then the convolution function is equivariant to g.

3. CNN Architecture

CNN architecture differs from the traditional multilayer perceptrons to ensure the architectural ideas we previously mentioned. In the literature we can find many CNN architectures but their components are very similar. If we consider the typical convolutional network architecture we may recognize (**?**):

- *Convolutional Layer*: Core building block of a CNN that does most of the computational heavy lifting. The layer's paramateres consist of learnable kernels or filters which extend though the full depth of the input. Each unit of this layer receives inputs from a set of units located in the receptive fields of the previous layer. During the forward pass each filter is convolved with the input which produces a map. When multiple such feature maps that are generated from multiple filters are stacked they form the output of the convolution layer. The weight vector that generates the feature map is shared which reduces the model complexity. In order to see how convolution relates to CNN we can consider a 1D convolutional layer with inputs x_n and outputs y_n (**?**):

$$y_n = A(x_n, x_{n+1}, ...) \tag{6}$$

Generally, A would be multiple neurons, but assuming a single one we can describe it as follows:

$$y = \sigma(Wx + b) \tag{7}$$

where W is a weight matrix, b is a bias and σ is an activation function. In MLP, the weight matrix connects every input to every neuron with different weights:

$$W = \begin{bmatrix} W_{0,0} & W_{0,1} & W_{0,2} & W_{0,3} & \cdots \\ W_{1,0} & W_{1,1} & W_{1,2} & W_{1,3} & \cdots \\ W_{2,0} & W_{2,1} & W_{2,2} & W_{2,3} & \cdots \\ W_{3,0} & W_{3,1} & W_{3,2} & W_{3,3} & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix} \tag{8}$$

In case of CNN, we connect each neuron to only a local region of the input data (sparse connectivity), reducing the number of parameters and the runtime. If we take a receptive field of 2×1:

$$W = \begin{bmatrix} W_{0,0} & W_{0,1} & 0 & 0 & \cdots \\ 0 & W_{1,1} & W_{1,2} & 0 & \cdots \\ 0 & 0 & W_{2,2} & W_{2,3} & \cdots \\ 0 & 0 & 0 & W_{3,3} & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix} \tag{9}$$

According to the parameter sharing property, CNNs reduce the storage requirements of the model to k parameters by sharing them for more than one function, in this example, two parameters:

$$W = \begin{bmatrix} w_0 & w_1 & 0 & 0 & \cdots \\ 0 & w_0 & w_1 & 0 & \cdots \\ 0 & 0 & w_0 & w_1 & \cdots \\ 0 & 0 & 0 & w_0 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix} \tag{10}$$

Multiplying the input by the above matrix is equivalent to convolving with the sequence $[...0, w_1, w_0, 0...]$. The function sliding to different positions corresponds to having neurons at those positions. This idea can be easily implemented for 2D inputs and 2D kernels as well:

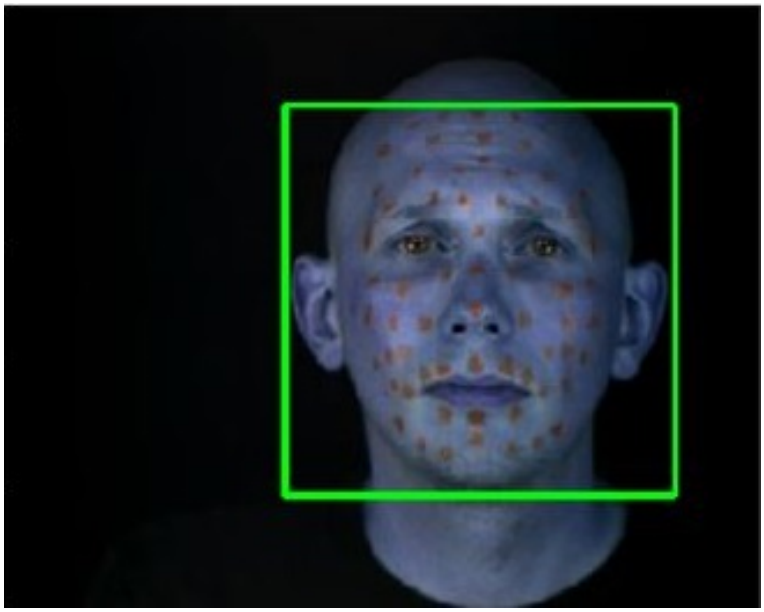

Figure 5: Example of a 2D convolution using a 3×3 kernel. Since we are not including *zero-padding*, the dimension of the feature map is smaller than the input. Reference: https://github.com/PetarV-/TikZ/tree/master/2D%20Convolution.

- *Non-linearity Layer*: Also known as activation function or detector stage, this layer introduces nonlinearities which are desirable for multilayer networks. The most common activation functions are sigmoid, tanh and ReLU. Compared with other functions, Rectified Linear Units (ReLU) (**???**) is preferable because neural netowrks train several times faster.

Figure 6: Non-linearity layer performance by using ReLU as activation function. Those numbers that are negative become zero in the feature map. Reference: https://medium.com/data-science-group-iitr/building-a-convolutional-neural-network-in-python-with-tensorflow-d251c3ca8117.

- *Pooling Layer*: Layer that down-samples the feature map obtained in the previous layer. It reduces the spatial size of the representation, thus reducing the parameters to be computed and controlling overfitting. The most popular pooling function is *max pooling* (**?**) that reports the maximum output within a rectangular neighborhood. Other pooling functions include the average or L^2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel. In all cases, pooling helps to make the representation become approximately invariant to small translations of the input.

Figure 7: Pooling layer implementation using max pooling operator (*Top*) and average pooling (*Bottom*) in a square neighborhood. Reference: https://medium.com/data-science-group-iitr/building-a-convolutional-neural-network-in-python-with-tensorflow-d251c3ca8117.

- Fully Connected Layer: Normal flat feedforward neural network layer. Fully connected layers are used at the end of the network after feature extraction and consolidation have been performed by the convolutional and pooling layers. They are used to create final nonlinear combinations of features and for making predictions by the network. These layers may have nonlinear activation function or a softmax (**?**) activation in order to output probabilities of class prediction.

Figure 8: Architecture of a fully-connected neural network. Reference: https://hackernoon.com/deep-learning-cnns-in-tensorflow-with-gpus-cba6efe0acc2/.

Figure 9: Architecture of a typical convolutional neural network. non-linearity layer are usually included between the convolutional layer and the pooling layer. Reference: https://www.pyimagesearch.com/2014/06/09/get-deep-learning-bandwagon-get-perspective/.

4. References