Alma Mater Studiorum · University of Bologna

School of Science
Department of Physics and Astronomy
Master Degree in Physics

OPTIMIZATION AND APPLICATIONS OF DEEP LEARNING ALGORITHMS FOR SUPER-RESOLUTION IN MRI

Supervisor:
Prof. Gastone Castellani

Co-supervisor:
Dr. Nico Curti

Submitted by:
Mattia Ceccarelli

Academic Year 2019/2020

Abstract

Abstract

Contents

Chapter 1

Introduction

Brief introduction of the work

1.1 Neural Network and Deep Learning

A neural network is an interconnected structure of simple procedurals units, called nodes. Their functionality is inspired by the animals' brain and from the works on learning and neural plasticity of Donald Hebb [6]. From his book :

> *Let us assume that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes that add to its stability.[...] When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased*

which is an attempt to describe the change of strenght in neural relations as a consequence of stimulations. From the so-called *Hebbian Theory* rose the first computational models such as the *Perceptron*, *Neural Networks* and the modern *Deep Learning*. The development of learning-based algorithms didn't catch up with the expected results until recently, mainly due to the exponential increase in available computational resources.

From a mathematical point of view, a neural network is a composition of non-linear multi-parametric functions. During the *training phase* the model tunes its parameters, starting from random ones, by minimizing the error function (called also loss or cost). Infact, machine learning problems are just optimization problems where the solution is not given in an analytical form, therefore trough iteratives techniques (generally some kind of gradient descent) we progressively approximate the correct result.

In general, there are 3 different kind of approach to learning:

- **supervised** It exists a labeled dataset in which the relationship between features (input) and expected output is known. During training, the model is presented

with many examples and it corrects its answers based on the correct response. Some problems tied to supervised algorithms are classification, regression, object detection, segmentation and super-resolution.

- **unsupervised** In this case, a labeled dataset does not exist, only the inputs data are available. The training procedure must be tailored around the problem under study. Some examples of unsupervised algorithms are clustering, autoencoders, anomaly detection.

- **reinforced** the model interacts with a dynamic environment and tries to reach a goal (e.g. winning in a competitive game). For each iteration of the training process we assign a reward or a punishent, relatively to the progress in reaching the objective.

This work will focus on models trained using labeled samples, therefore in a supervised environment.

Perceptron

The Perceptron (also called *artificial neuron*) is the fundamental unit of every neural network and it is a simple model for a biological neuron, based on the works of Rosenblatt [12]. The *perceptron* receives N input values $x_1, x_2, ...x_N$ and the output is just a linear combination of the inputs plus a bias :

$$y = \sigma(\sum_{k=1}^{N} w_k x_k + w_0) \tag{1.1}$$

where σ is called *activation function* and $w_0, w_1, ...w_N$ are the trainable weights.

Originally, the activation function was the *Heaviside step function* whose value is zero for negative arguments and one for non-negative arguments:

$$H(x) := \begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x \geq 0 \end{cases} \tag{1.2}$$

In this case the perceptron is a *linear discriminator* and as such, it is able to learn an hyperplane which linearly separates two set of data. The weights are tuned during the training phase following the given update rule, usually :

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n + \eta(t - y)\boldsymbol{x} \tag{1.3}$$

where η is the learning rate ($\eta \in [0, 1]$) and t is the true output. If the input instance is correctly classified, the error $(t-y)$ would be zero and no weight is changed. Otherwise, the hyperplane is moved towards the misclassified example. Repeating this process will lead to a convergence only if the two classes are linearly separable.

Fully Connected Structure

The direct generalization of a simple perceptron is the *Fully Connected Artificial Neural Network* (or *Multy Layer Perceptron*). It is composed by many Perceptron-like units called nodes, any one them performs the same computation as formula 1.3 and *feed* their output *forward* to the next layer of nodes. A typical representation of this type of network is shown in figure 1.1:

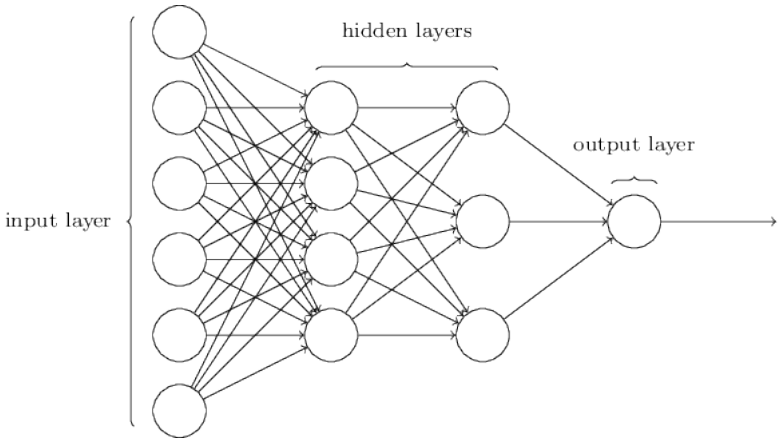

Figure 1.1: *A common representation of a neural network: a single node works as the perceptron described above.*

While the number of nodes in the input and output layers is fixed by the data under analysis, the best configuration of hidden layers is still an open problem.

The mathematical generalization from the perceptron is simple, indeed given the i-th layer its output vector $\boldsymbol{y}_i$ reads:

$$\boldsymbol{y}_i = \sigma(W_i \boldsymbol{y}_{i-1} + \boldsymbol{b}_i) \tag{1.4}$$

where W_i is the weights matrix of layer i and $\boldsymbol{b}_i$ is the i-th bias vector, equivalent to w_0 in the perceptron case. The output of the i-th layer becomes the input of the next one until the output layer yields the network's answer.

As before, σ is the activation function which can be different for every node, but it usually differs only from layer to layer. The choice of the best function for a given problem is still an open issue.

In a supervised environment, the model output is compared to the desired output (*truth*) by means of a cost function. An example of cost function is the sum of squared error :

$$C(W) = \frac{1}{N} \sum_{j=1}^{N} (y_j - t_j)^2 \tag{1.5}$$

where N is the dimensionality of the output space. C is considered as a function of the model's weights only since input data and true label t are fixed.

overcome perceptron problems

Those architectures are *universal approximators*, that means given an arbitrarly complex function, there is a fully connected neural network that can approximate it.

This type of network is called *feed forward* because the information flows linearly from the input to the output layer: however, it exists a class of models called *Recurrent* where this is not the case anymore and feedback loop are possible, but they are outside the scope of this work.

Gradient Descent

To minimize the loss function an update rule for the weights is needed. Given a cost funtion $C(w)$, the most simple one is the gradient descent:

$$w \leftarrow w - \eta \nabla_w C \tag{1.6}$$

The core idea is to modify the parameters by a small step in direction that minimize the error function. The lenght of the step is given by the *learning rate* η, which is a hyperparameter chosen by the user, while the direction of the step is given by $-\nabla_w C$, which point towards the steepest descent of the function landscape.

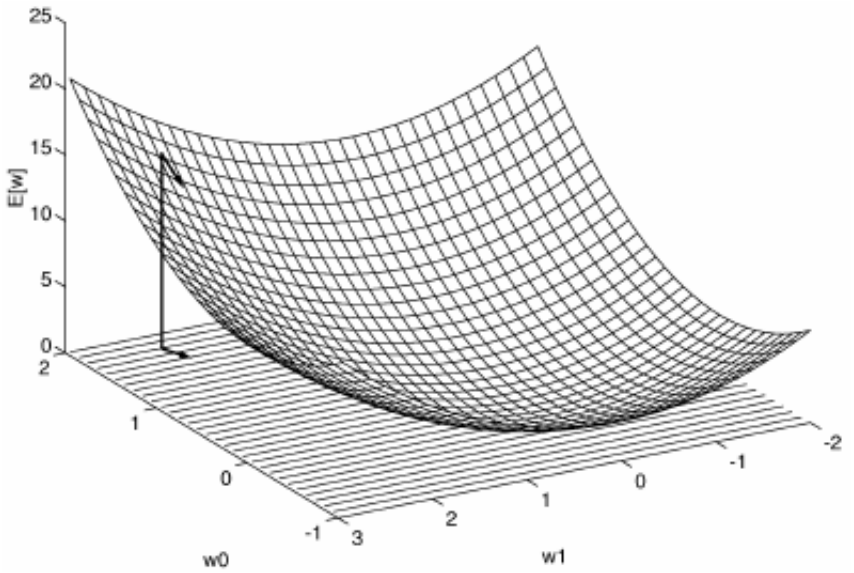

Figure 1.2: *Visual example of gradient descent for a model with 2 weights. The idea is to modify the weights to follow the direction of steepest descent for the landscape of the error function*

The speed at which the algorithm converge to a solution and the precision of said

solution are greatly influenced by the update rule. More complex and efficient update rules do exist, but they follow the same idea as the gradient descent.

Error Back Propagation

The most common algorithm used to compute the updates to weights in the learning phase is the *Error Back Propagation*. Given a differentiable cost function $C(W)$, let's define :

$$\boldsymbol{z}_l = W_l \boldsymbol{y}_{l-1} + \boldsymbol{b}_l \tag{1.7}$$

$$\boldsymbol{a}_l = \sigma(\boldsymbol{z}_l) \tag{1.8}$$

respectively the de-activated and activated output vectors of layer l for a model with L total layer, and:

$$\boldsymbol{\delta}_l = (\frac{\partial C}{\partial z_l^1}, \ldots, \frac{\partial C}{\partial z_l^N}) \tag{1.9}$$

as the vector of errors of layer l. Then we can write the 4 equations of back propagation for the fully connected neural network [10]:

$$\boldsymbol{\delta}_L = \nabla_a C \odot \sigma'(\boldsymbol{z}_L) \tag{1.10}$$

$$\boldsymbol{\delta}_l = (W_{l+1}^T \boldsymbol{\delta}_{l+1}) \odot \sigma'(\boldsymbol{z}_l) \tag{1.11}$$

$$\frac{\partial C}{\partial b_l^j} = \delta_l^j \tag{1.12}$$

$$\frac{\partial C}{\partial w_l^{jk}} = a_{l-1}^k \delta_l^j \tag{1.13}$$

where $\odot$ is the element-wise product. Those equations can be generalized for others kind of layer, as I will show in the next chapters.

The full training algorithm is :

- define the model with random parameters
- compute the output for one of the inputs
- compute the loss function $C(W)$ and the gradients $\frac{\partial C}{\partial w_l^{jk}}$ and $\frac{\partial C}{\partial b_l^j}$ for each l.
- updates the parameters following the update rule,
- iterate from step 2 until the loss is sufficiently small

1.2 Super Resolution

The term Super-Resolution (SR) referes to a class of techniques which aim is to enhance the spatial resolution of an image, thus converting a given low resolution (LR) image to a corresponding high resolution (HR) one, with better visual quality and refined details. Image super-resolution is also called by other names like image scaling, interpolation, upsampling and zooming [1]. **FINISH**

Image Quality

[7]

PSNR

SSIM

1.3 Magnetic Resonance

Chapter 2

Algorithms

Brief Intro to the works on algorithm (NumPyNet, Byron, Layers,)

2.1 Byron

Byron (Build YouR Own Neural network) is a novel Deep Learning framework written in C++.

2.2 NumPyNet

2.3 Convolutional Neural Network

A Convolutional Neural Network (CNN) is a specialized kind of neural network for processing data that has known grid-like topology [4], like images, that can be considered as a grid of pixel. The name indicates that at least one of the functions employed by the network is a convolution. In a continuos domain the convolution between two functions f and g is defined as:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau \tag{2.1}$$

The first function f is usually referred to as the input and the second function g as kernel. For Image Processing applications we can define a 2-dimensional discrete version of the convolution in a finite domain using an image I as input and a 2 dimensional kernel k:

$$C[i,j] = \sum_{u=-N}^{N} \sum_{v=-M}^{M} k[u,v] \cdot I[i-u, j-v] \tag{2.2}$$

where $C[i,j]$ is the pixel value of the output image and N, M are the kernel dimensions. Practically speaking, a convolution is performed by sliding a kernel of dimension $N \times M$ over the image, each kernel position corresponds to a single output pixel, the value of

which is calculated by multiplying together the kernel value and the underlaying pixel value for each cell of the kernel and summing all the results, as shown in figure 2.1:

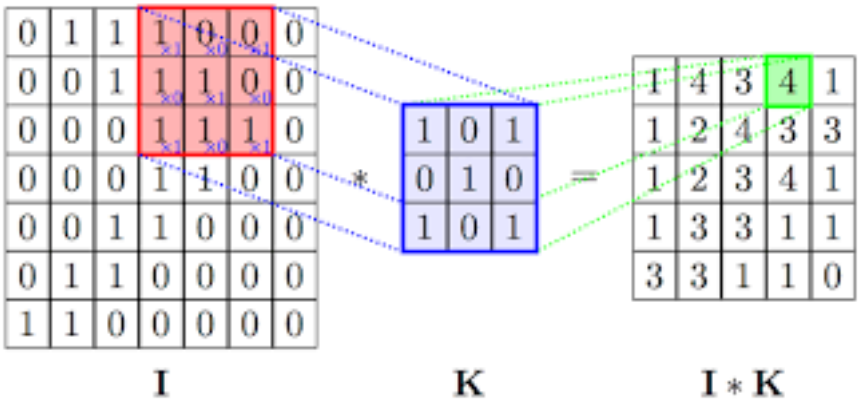

Figure 2.1: *Visual example of convolution of an image I 7x7 with a 3x3 kernel K.*

The convolution operation is also called *filtering*. By chosing the right kernel (filter) it is possible to highlight different features. For this reason the convolution operation is commonly used in image analysis: some of the most common applications are denoising, edge detection and edge enhancement.

2.4 Layers

As described above, a neural network can be considered as a composition of function: for this reasons every Deep Learning framework (e.g. Keras/Tensorflow, Pytorch, Darknet) implement each function as an independent object called *Layer*. In Byron and NumPyNet, each layer contains at least 3 methods:

- **forward** the forward method compute the output of the layer, given as input the previous output.
- **backward** the backward method is essential for the training phase of the model: indeed, it computes all the updates for the layer weights and backpropagates the error to the previous layers in the chain.
- **update** the update method applies the given update rules to the layer's weights.

By stacking different kinds of layer one after another, it is possible to build complex models with tens of millions of parameters. For the purposes of this work, I'm going to describe layers used in super resolution, however, Byron is developed also for different applications (object detection, classification, segmentation, style transfer, Recurrent NN etc...) and as such, many more layers are available.

Convolutional Layer

The convolutional layer (CL) object is the most used layer in DL image analysis, therefore its implementation must be as efficient as possible. Its purpose is to perform multiple (sometimes thousands) convolution over the input to extract different high-level features, which are compositions of many low-level attributes of the image (e.g edges, simple shapes). In the brain/neuron analogy, every entry in the output volume can also be interpreted as an output of a neuron that looks at only a small region, the neuron's *receptive field* in the input and shares parameters with all the neuron spatially close. As more CLs are stacked, the receptive field of a single neuron grows and with that, the complexity of the features it is able to extract. The local nature of the receptive field allows the models to recognize features regardless of the position in the images. In other words, it is independent from translations [4].

The difference from a traditional convolutional approach is that instead of using predetermined filters, the network is supposed to learn its own. A CL si defined by the following parameters:

- **kernel size** : it is the size of the sliding filters. The depth of the filters is decided by the depth of the input images (which is the number of channels.). The remanining 2 dimensions (widht and height) can be indipendent from one another, but most implementation require square kernels.
- **strides** : defines the movement of the filters. With a low stride (e.g. unitary) the windows tends to overlap. With highr stride values we have less overlap (or none) and the dimension of the output decrease.
- **number of filters** : is the number of different filters to apply to the input. It also indicates the depth of the output.
- **padding** : is the dimensions of an artificial enlargement of the input to allow the application of filters on borders. Usually, it can be interpreted as the number of rows/columns of pixel to add to the input, however some libraries (e.g Keras) consider it only as binary: in case is true, only the minimum number of rows/columns are appended to keep the same spatial dimension.

Given the parameters, it is straightforward to compute the number of weights and bias needed for the initialization of the CL: indeed, suppose an image of dimensions (H, W, C) slided by n different 3-D filters of size (k_x, k_y) with strides (s_x, s_y) and padding p, then:

$$\#weights = n \times k_x \times k_y \times C \tag{2.3}$$

$$\#bias = n \tag{2.4}$$

Note that the number of weights does not depend on the input spatial size but only on its depth. It is important because a fully convolutional network can receives images of any size as long as they have the correct depth. Moreover, using larger inputs do not requires more weights, as is the case for fully connected structure.

The output dimensions are (out_H, out_W, n) where :

$$out\_H = \lfloor \frac{H - k_x + p}{s_x} \rfloor + 1 \tag{2.5}$$

$$out\_W = \lfloor \frac{W - k_y + p}{s_y} \rfloor + 1 \tag{2.6}$$

Even if the operation can be implemented as described above in equation 2.2, this is never the case: it is certainly easier but also order of magnitude slower than more common algorithms. A huge speed up in performances is given by realising that a discrete convolution can be viewed as a single matrix multiplication. By performing a clever transformation of the input into a flat matrix, in which every columns yields the values that have to be multiplied by the filters for each output, as shown in figure 2.2

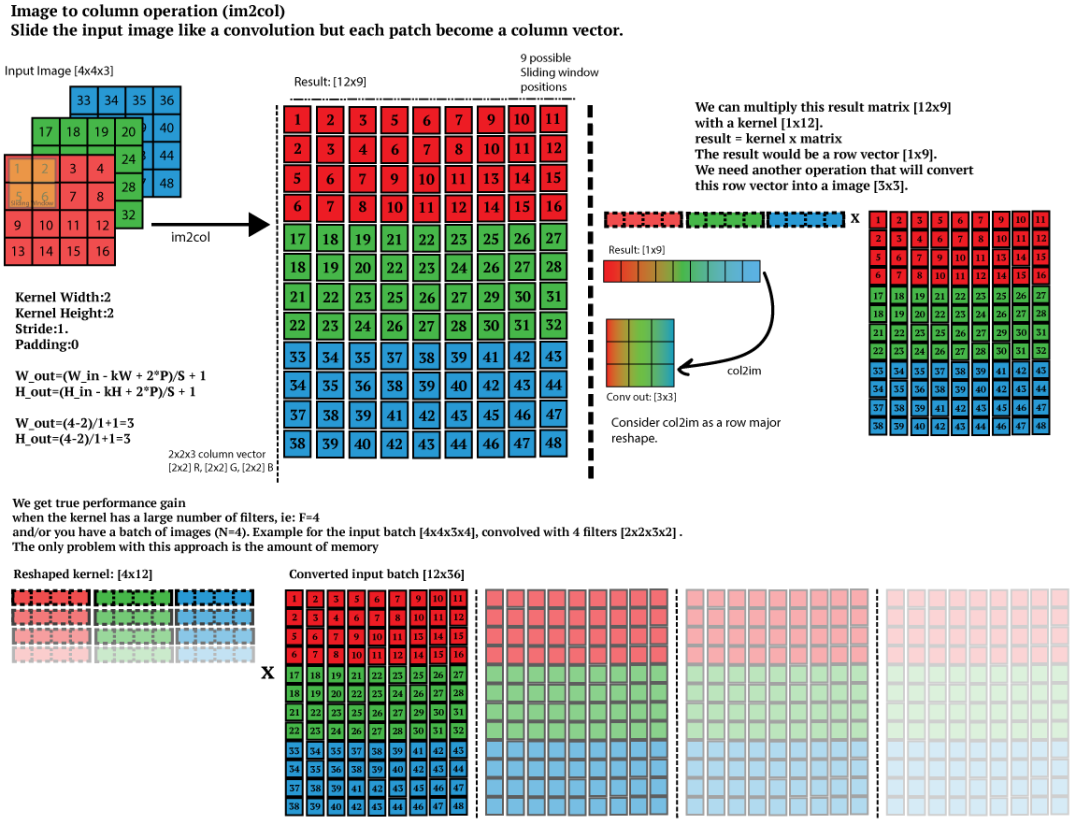

Figure 2.2: *Scheme of the im2col algorithm using a* $2 \times 2 \times 3$ *filter with stride 1 on a* $4 \times 4 \times 3$ *image. The matrix multiplication is between a* $n \times 12$ *and a* 12×9 *matrixes.*

This re-arrengement is commonly called im2col. The main downside is that a lot more memory is needed to store the newly arranged matrix. The larger the number of kernels, the higher is the time gain of this implementation over a naive one.

Another important optimization comes from linear algebra considerations and is called *Coppersmith-Winograd algorithm*, which was designed to optimize the matrix product. Suppose we have an input image of just 4 elements and a 1-D filter mask with size 3:

$$\text{img} = \begin{bmatrix} d0 & d1 & d2 & d3 \end{bmatrix} \qquad \text{weights} = \begin{bmatrix} g0 & g1 & g2 \end{bmatrix} \tag{2.7}$$

we can now use the im2col algorithm previously described and reshape our input image and weights into

$$\text{img} = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix}, \qquad \text{weights} = \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} \tag{2.8}$$

given this data, we can simply compute the output as the matrix product of this two matrices:

$$\text{output} = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} d0 \cdot g0 + d1 \cdot g1 + d2 \cdot g2 \\ d1 \cdot g0 + d2 \cdot g1 + d3 \cdot g2 \end{bmatrix} \tag{2.9}$$

The Winograd algorithm rewrites this computation as follow:

$$\text{output} = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} m1 + m2 + m3 \\ m2 - m3 - m4 \end{bmatrix} \tag{2.10}$$

where

$$\begin{aligned} m1 &= (d0 - d2)g0 \qquad & m2 &= (d1 + d2)\frac{g0 + g1 + g2}{2} \\ m4 &= (d1 - d3)g2 \qquad & m3 &= (d2 - d1)\frac{g0 - g1 + g2}{2} \end{aligned} \tag{2.11}$$

The two fractions in $m2$ and $m3$ involve only weight's values, so they can be computed once per filter. Moreover, the normal matrix multiplication is composed of 6 multiplications and 4 addition, while the winograd algorithm reduce the number of multiplication to 4, that is very significant, considering that a single multiplication takes 7 clock-cycles and an addition only 3. In Byron we provide the winograd algorithm for square kernels of size 3 and stride 1, since it is one of the most common combinations in Deep Learning and the generalization is not straightforward.

If we consider a single output for the CL, the operation reads:

- col2im backward

Pooling

Pooling operations are down-sampling operations, so that the spatial dimensions of the input are reduced. Similarly to what happens in a CL, in pooling layers a 3-D kernel of size $k_x \times k_y \times C$ slides across an image of size $H \times W \times C$, however the operation performed by this kind of layers is fixed and does not change during the course of training. The two main pooling functions are max-pooling and average-pooling: as suggested by the names, the former returns the maximum value of every window of the images super-posed by the kernel, as shown in figure 2.3:

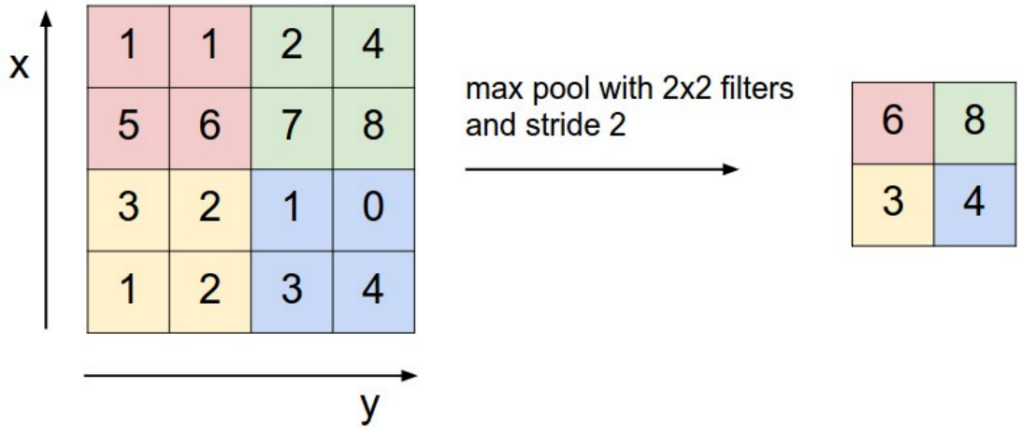

Figure 2.3: *Scheme of maxpool operations with a kernel of size* 2×2 *and stride* 2 *over an image of size* 4×4*. Picture from CS231n*

The latter instead, returns the average value of the window and can be seen as a convolution where every weight in the kernel is $\frac{1}{k_x \cdot k_y}$. The results expected from an Average pooling operations are shown in figure 2.4:

Figure 2.4: *Average pooling applied to a test image: (left) the original image, (center) average pooling with a* 3×3 *kernel, (right) average pooling with a* 30×30 *kernel. The images have been obtained using NumPyNet*

Other popular pooling functions include the L^2 norm of a rectangular neighborhood or a weighted average based on the distance from the central pixel.

A typical block of a convolutional network consists of three stages: In the first stage a CL performs several convolutions in parallel, in the second stage each convolution result is run through a non-linear activation function (sometimes called *detector*) and in the third stage a pooling function is used to further modify the output. The modification brought by pooling is helpful in different ways: first of all, it is a straightforward computational performance improvement, since less features also means less operations. Moreover, in all cases, pooling helps to make representation approximately invariant to small translation of the input and invariance to local translation can be a useful property if the objective is to decide wether a feature is present rather than where it is located [4]. The reductions of features can also prevent over-fitting problems during training, improving the general performances of the model.

A pooling layer is defined by the same parameters as a CL, minus the number of filters:

- **kernel size** : it is the size of the sliding filters. The depth of the filters is decided by the depth of the input images (which is the number of channels.). The remanining 2 dimensions (widht and height) can be indipendent from one another, but most implementations require square kernels.
- **strides** : defines the movement of the filters. With a low stride (e.g. unitary) the windows tends to overlap. With higher stride values we have less overlap (or none) and the dimension of the output decrease. Usually pooling operations have a stride of 2.
- **padding** : is the dimensions of an artificial enlargement of the input to allow the application of filters on borders. Usually, it can be interpreted as the number of rows/columns of pixel to add to the input, however some libraries (e.g Keras) consider it only as binary: in case is true, only the minimum number of rows/columns are appended to keep the same spatial dimension. Most often than not, no padding is applied during pooling operations.

The output dimensions for Pooling layers are the same as for CLs, however, since the operations does not change during the training phase, they have no weights.

Due to the similarities with the CL it is possible to implement a pooling layers through the im2col algorithm, as an example, the NumPyNet implementation shown in the snippet below make use of the function `asStride` to create a `view` of the input array:

Listing 2.1: NumPyNet version of AvgPool function

```
import numpy as np

class Avgpool_layer(object):

  def __init__(self, size=(3, 3), stride=(2, 2)):

    self.size = size
    self.stride = stride
```

```
9       self.batch, self.w, self.h, self.c = (0, 0, 0, 0)
10      self.output, self.delta = (None, None)
11
12  def _asStride(self, input, size, stride):
13
14      batch_stride, s0, s1 = input.strides[:3]
15      batch,         w,  h  = input.shape[:3]
16      kx, ky      = size
17      st1, st2    = stride
18
19      # Shape of the final view
20      view_shape = (batch, 1 + (w - kx)//st1, 1 + (h - ky)//st2) + input.
    shape[3:] + (kx, ky)
21
22      # strides of the final view
23      strides = (batch_stride, st1 * s0, st2 * s1) + input.strides[3:] +
    (s0, s1)
24
25      subs = np.lib.stride_tricks.as_strided(input, view_shape, strides=
    strides)
26      # returns a view with shape = (batch, out_w, out_h, out_c, kx, ky)
27      return subs
28
29  def forward(self, input):
30
31      self.batch, self.w, self.h, self.c = input.shape
32      kx, ky = self.size
33      sx, sy = self.stride
34
35      input = input[:, : (self.w - kx) // sx*sx + kx, : (self.h - ky) //
    sy*sy + ky, ...]
36      # 'view' is the strided input image, shape = (batch, out_w, out_h,
    out_c, kx, ky)
37      view = self._asStride(input, self.size, self.stride)
38
39      # Mean of every sub matrix, computed without considering the pad(np
    .nan)
40      self.output = np.nanmean(view, axis=(4, 5))
```

A `view` is a special `numpy` object which retains the same information of the original array arranged in a different way, but without occupying more memory. In this case, the re-arrengement is very similar to an im2col, with the only differece that we are not bound to any number of dimensions. The resulting tensor has indeed 6 dimensions. Since no copy is produces in this operation we can obtain a faster execution.

In pooling layer the backward function is similar to what we saw for convolutional layers, this time we don't have to compute the weights updates though, only the error to backprogate along the network. For maxpool layers, only the maximum input pixel for every window is involved in the backward pass. Indeed, if we consider the simple case in which the forward function is :

$$m = max(a, b) \tag{2.12}$$

and, as described in the dedicated chapter, we know that $\frac{\partial C}{\partial m}$ is the error passed back from the next layer: the objective is to compute $\frac{\partial C}{\partial a}$ and $\frac{\partial C}{\partial b}$. If $a > b$ we have :

$$m = a \quad \Rightarrow \quad \frac{\partial C}{\partial m} = \frac{\partial C}{\partial a} \tag{2.13}$$

m does not depends on b so $\frac{\partial C}{\partial b} = 0$.

So the error is passed only to those pixel which value is maximum in the considered window, the other are zeros. In figure 2.5 an example of forward and backward pass for a maxpool kernel of size 30 and stride 20.

Figure 2.5: *Max pooling applied to a test image: (left) the original image, (center) max pooling with a* 30×30 *kernel and stride* 20*, (right) max pooling errors image. Only few of the pixels are responsible for the error backpropagation. The images have been obtained using NumPyNet*

The backward pass for the average pool layer is the same as for the CL, considering that in this case the "weights" are fixed.

Shortcut Connections

An important advancement in network architecture has been brought by the introduction of Shortcut (or Residual) Connections [5]. Famously, deep models suffer from *degradation problems* after reaching a maximum depth. Adding more layers, thus increasing the depth of the model, saturates the accuracy which eventually starts to rapidly decrease. The main cause of this dergradation is not overfitting, but numerical instability tied to gradient backpropagation: indeed, as the gradient is back-propagated through the network, repeated multiplications can make those gradients very small or, alternatevely, very big, This problem is well known in Deep Learning and takes the name of *vanishing/exploding gradients* and it makes almost impossible to train very large models, since early layers may not learn anything even after hundreds of epochs. A residual connection

is a special shortcut which connects 2 different part of the network with a simple lnear combination. Instead of learning a function $F(x)$ we try to learn $H(x) = F(x) + x$, as shown in figure 2.6:

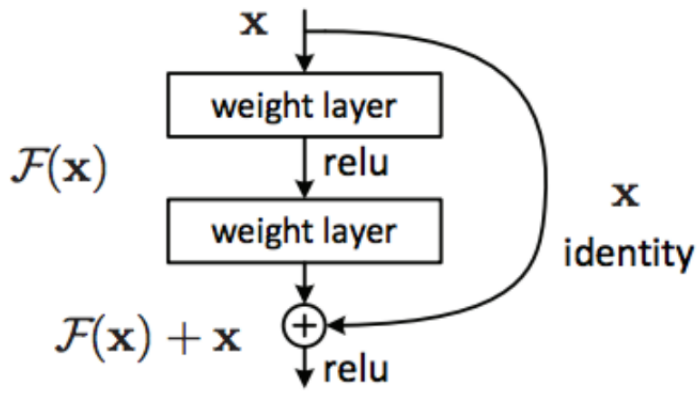

Figure 2.6: *Scheme of the shortcut layer as designed by the authors [5]. The output of the second layer become a linear combination of the input x and its own output.*

During the back propagation the gradient of higher layers can easily pass to the lower layers, without being mediated, which may cause vanishing or exploding gradient.
Even the shortcut connection can be implemented as a stand-alone layer, defined by the following parameters:

- **index** is the index of the second input of this layer x_2 (the first one x_1 is the output of the previous layer).
- **alpha** the first coefficient of the linear combination, multiplied by x_1.
- **beta** the second coefficient of the linear combination, multiplied by x_2.

Both in NumPyNet and Byron, we chose to generalize the formula as:

$$H(x_1, x_2) = \alpha x_1 + \beta x_2 \tag{2.14}$$

Where x_1 is the output of the previous layer and x_2 is the output of the layer selected by `index`. The backward function is simply :

$$\frac{\partial C}{\partial x_1} = \frac{\partial C}{\partial H}\frac{\partial H}{\partial x_1} = \delta \cdot \alpha \tag{2.15}$$

for the first layer and :

$$\frac{\partial C}{\partial x_2} = \frac{\partial C}{\partial H}\frac{\partial H}{\partial x_2} = \delta \cdot \beta \tag{2.16}$$

for the second layer. Again, δ is the error backpropagated from the next layer. Residuals connections were first introduced for image classification problems, but they rapidly become part of numerous models for every kind of application tied to Image Analysis.

Pixel Shuffle

Using pooling and convolutional layers with non unitarian strides is a simple way to downsample the input dimension. For some applications though, we may be interested in upsampling the input, for example :

- in image to image processing (input and output are images of the same size) it is common to perform a compression to an internal encoding (e.g Deblurring, U-Net Segmentation).
- project feature maps to a higher dimensional space, i.d. to obtain a image of higher resolution (e.g Super-Resolution)

for this purposes the *transposed convolution* (also called *deconvolution*) was introduced. The transposed convolution can be treated as a normal convolution with a sub-unitarian stride, by upsampling the input with empty rows and columns and then apply a single strided convolution, as shown in figure 2.7:

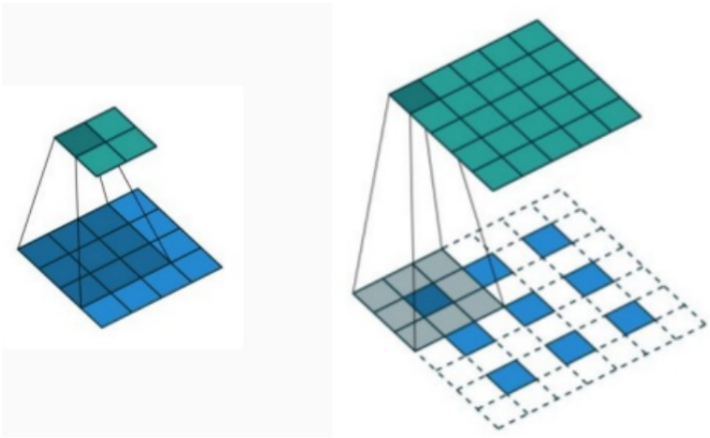

Figure 2.7: *example of deconvolution: (left) a normal convolution with size 3 and stride 1, (right) after applying a "zeros upsampling" the convolution of size 3 and stride 1 become a deconvolution*

Although working, the transposed convolution is not efficient in terms of computational and memory cost, therefore not suited for modern convolutional neural network. An alternative is the recently introduced *sub-pixel convolution* [13] (also called Pixel Shuffle). The main advantages over the deconvolution operation is the absece of weights to train: indeed the operation perfermod by the Pixel Shuffle (PS) Layer is deterministic and it is very efficient if compared to the deconvolution, since it only performs a re-arrengement of the pixels.

Given a scale factor r, the PS organizes an input $H \times W \times C \cdot r^2$ into an output tensor $r \cdot H \times r \cdot W \times C$, which generally is the dimension of the high resolution space. So, strictly speaking, the PS does not perform any upsample, since the number of pixels stays the same. In figure 2.8 is shown an example with $C = 1$:

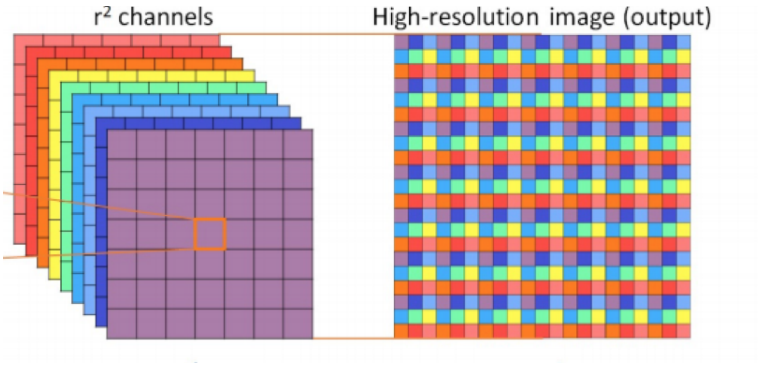

Figure 2.8: *Example of pixel shuffling proposed by the authors [13]. In this example,* r^2 *features maps are re-arranged into a single-channeled high resolution output.*

As suggested by the authors, the best practice to improve performances is to upscale from low resolution to High resolution only at the very end of the model. In this way the CL can efficienlty produce an high number of low resolution feature maps that the PS can organize into the final output.

In both NumPyNet and Byron, the pixel shuffle layer is defined only by the `scale` parameter, which lead the entire transformation. in the first case, it is possible to implement forward and backward using the functions `split`, `reshape`, `concatenate` and `transpose` of the `numpy` library [11]. This implementation has been tested against `tensorflow`'s `depth_to_space` and `space_to_depth`. Despite beign available in most deep learning library, a low level `C++` implementation for the PS algorithm is hard to find. In Byron we propose a dynamic algorithm able to work for both `channel last` and `channel first` input. The algorithm is essentially a re-indexing of the input array in six nested for-loops. The first soulution taken into account during the development was the contraction of the loops into a single one using divisions to obtain the correct indexes: however the amount of required divisions weights on the computational performances, given that divisions are the most expensive in terms of CPU clock-cycles.

The backward function of this layer does not involve any gradient computation: instead, it is the inverse of the re-arrengement performed in the forward function.

Batch Normalization

When training a neural network, the standard approach is to separate the dataset in groups, called *batch*. In this way**FINISH**

Batch normalization is the operation that normalizes the features of the input along the batch axis. Let's M be the number of examples in the group and ϵ a small variable added for numerical stability, the batch normalization function is defined as:

$$\mu = \frac{1}{M}\sum_{i=1}^{M} x_i \tag{2.17}$$

$$\sigma^2 = \frac{1}{M}\sum_{i=1}^{M}(x_i - \mu)^2 \tag{2.18}$$

$$\hat{x}_i = \frac{(x_i - \mu)^2}{\sqrt{\sigma^2 + \epsilon}} \tag{2.19}$$

$$y_i = \gamma\bar{x}_i + \beta \tag{2.20}$$

where γ and β are the trainable weights of this layer. In the case of a tensor of images of size $M \times H \times W \times C$ all the quantities are multidimensional tensor as well and all the operations are performed element-wise.

The backward function can be computed following the chain rule for derivatives. As usual, define $\delta = \frac{\partial C}{\partial y}$ as the error coming from the next layer, the goal is to compute the updates for γ and β and the error for the previous layer. The updates are straightforward:

$$\frac{\partial C}{\partial \gamma} = \frac{-}{i} \cdot \frac{\partial y_i}{\partial \gamma} = \sum_{i=1}^{M} \delta_i \cdot \hat{x}_i \tag{2.21}$$

$$\frac{\partial C}{\partial \beta} = \frac{\partial C}{\partial y_i} \cdot \frac{\partial y_i}{\partial \beta} = \sum_{i=1}^{M} \delta_i \tag{2.22}$$

Knowing the correct operations, an example of implementation is shown in the snippet 2.2:

Listing 2.2: NumPyNet version of batchnorm function

```
1
2  def forward(self, inpt):
3      '''
4      Forward function of the BatchNormalization layer. It computes the
        output of
5      the layer, the formula is :
6                      output = scale * input_norm + bias
7      Where input_norm is:
8                      input_norm = (input - mean) / sqrt(var + epsil)
9      where mean and var are the mean and the variance of the input batch
        of
10     images computed over the first axis (batch)
11     Parameters:
12       inpt  : numpy array, batch of input images in the format (batch, w,
        h, c)
13     '''
14
15     self._check_dims(shape=self.input_shape, arr=inpt, func='Forward')
16
```

```
17    # Copy input, compute mean and inverse variance with respect the
       batch axis
18    self.x    = inpt.copy()
19    self.mean = self.x.mean(axis=0)                              # shape =
       (w, h, c)
20    self.var  = 1. / np.sqrt((self.x.var(axis=0)) + self.epsil) # shape =
       (w, h, c)
21    # epsil is used to avoid divisions by zero
22
23    # Compute the normalized input
24    self.x_norm = (self.x - self.mean) * self.var # shape (batch, w, h, c
       )
25    self.output = self.x_norm.copy() # made a copy to store x_norm, used
       in Backward
26
27    # Init scales and bias if they are not initialized (ones and zeros)
28    if self.scales is None:
29      self.scales = np.ones(shape=self.out_shape[1:])
30
31    if self.bias is None:
32      self.bias = np.zeros(shape=self.out_shape[1:])
33
34    # Output = scale * x_norm + bias
35    self.output = self.output * self.scales + self.bias
36
37    # output_shape = (batch, w, h, c)
38    self.delta = np.zeros(shape=self.out_shape, dtype=float)
39
40    return self
41
42  def backward(self, delta=None):
43    '''
44    BackPropagation function of the BatchNormalization layer. Every
       formula is a derivative
45    computed by chain rules: dbeta = derivative of output w.r.t. bias,
       dgamma = derivative of
46    output w.r.t. scales etc...
47    Parameters:
48      delta : the global error to be backpropagated, its shape should be
       the same
49        as the input of the forward function (batch, w, h ,c)
50    '''
51
52    check_is_fitted(self, 'delta')
53    self._check_dims(shape=self.input_shape, arr=delta, func='Forward')
54
55    invN = 1. / np.prod(self.mean.shape)
56
57    # Those are the explicit computation of every derivative involved in
       BackPropagation
58    # of the batchNorm layer, where dbeta = dout / dbeta, dgamma = dout /
```

```
    dgamma etc...

  self.bias_update = self.delta.sum(axis=0)                    # dbeta
  self.scales_update = (self.delta * self.x_norm).sum(axis=0) # dgamma

  self.delta *= self.scales                                     # self.
   delta = dx_norm from now on

  self.mean_delta = (self.delta * (-self.var)).mean(axis=0)    # dmu

  self.var_delta = ((self.delta * (self.x - self.mean)).sum(axis=0) *
                    (-.5 * self.var * self.var * self.var))     # dvar

  # Here, delta is the derivative of the output w.r.t. input
  self.delta = (self.delta * self.var +
                self.var_delta * 2 * (self.x - self.mean) * invN +
                self.mean_delta * invN)

  if delta is not None:
    delta[:] += self.delta

  return self
```

Activations

An important role in neural network is played by the choice of activation function.

- table
- example images
- forward backward

Loss Function

2.5 Timing

2.6

Chapter 3

Dataset and Methodology

3.1 Dataset

3.2 Models

Chapter 4

Results

Bibliography

[1] S. Anwar, S. Khan, and N. Barnes. A Deep Journey into Super-resolution: A survey. *arXiv e-prints*, page arXiv:1904.07523, Apr. 2019.

[2] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[3] N. Curti. Implementation and optimization of algorithms in biomedical big data analytics. https://nico-curti2.gitbook.io/phd-thesis/, 2019.

[4] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[5] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. *arXiv e-prints*, page arXiv:1512.03385, Dec. 2015.

[6] D. Hebb. *The Organization of Behavior*. 1949.

[7] A. Hore and D. Ziou. Image quality metrics: Psnr vs. ssim. In *2010 20th International Conference on Pattern Recognition*, pages 2366–2369, Aug 2010.

[8] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv e-prints*, page arXiv:1502.03167, Feb. 2015.

[9] B. Lim, S. Son, H. Kim, S. Nah, and K. M. Lee. Enhanced deep residual networks for single image super-resolution. *arXiv e-prints*, page arXiv:1707.02921, Jul 2017.

[10] M. A. Nielsen. *Neural Network and Deep Learning*. Determination Press, 2015.

[11] T. Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–.

[12] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[13] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang. Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network. *arXiv e-prints*, page arXiv:1609.05158, Sept. 2016.

[14] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for Simplicity: The All Convolutional Net. *arXiv e-prints*, page arXiv:1412.6806, Dec. 2014.

[15] J. Yu, Y. Fan, J. Yang, N. Xu, Z. Wang, X. Wang, and T. Huang. Wide activation for efficient and accurate image super-resolution. *arXiv e-prints*, page arXiv:1808.08718, Aug 2018.