POLITECNICO DI TORINO

Master degree course in
MECHATRONIC ENGINEERING Classe LM-25 (DM270)

Master Degree Thesis

DEEP LEARNING FOR AUTOMATIC CRACK DETECTION INSIDE TUNNELS

Second Prototype

Thesis Advisor
Prof. Roberto Garello

Candidate
Luca Zacheo

Research Supervisors
Prof.ssa Marina Mondin
Prof. Fred Daneshgaran

Academic year 2018-2019

A mia madre

Contents

List of Figures

Chapter 1

Introduction

The rapid urbanization of cities around the globe and the corresponding exponential growth in transportation infrastructure to support ever increasing population densities, has led to the corresponding increase in road tunnels as a means of alleviating congestion, where and when possible. This necessitates development of automated techniques for detection of cracks on the surfaces inside tunnels given the high costs of manual visual inspection both monetarily and in terms of time. In addition, the growth of the sub-urban sprawl to provide affordable housing has accelerated the construction of tunnels, which became very useful for re-directing traffic flows and reducing congestions. Structural integrity testing of the road tunnels imposes the first problem of managing the traffic flow during the inspections, since it may not always be possible to block the traffic and examine the entire tunnel to perform a human visual inspection or to prepare and mount a complex robotic structure for the same purpose. Furthermore, the number of tunnels may be large, leading to potentially expensive inspection systems in terms of money and time that may not be affordable specially in the developing countries. In this scenario, a low-cost automatic system can be a smart solution and can overcome many of the previously described problems.
This thesis project is centered in the University Transportation Center, a collaboration between different American universities (California State University Los Angeles, Colorado School of Mines and Leigh University) focused on solving challenging problems in the urban infrastructures, such as galleries, buildings or bridges and financed by the Department of Transportation of United States.
In particular, this work introduces a low cost automated system for tunnel inspections, which is also translated in a very simple structure that does not require so complex preliminary work and can be used also in urban tunnels with normal traffic flow. This means that this approach is completely different from the ones present in literature, specifically in terms of cost.
The system is composed by two main blocks: the acquisition block, for the collection of images in the galleries surfaces, and the decisional block, composed by a deep

convolutional neural network which has to classify properly the images in two main categories, *Crack Detection* and *No Crack Detection*. In the acquisition system, a second generation prototype is based on a picture acquisition system completely automatized and able to collect a huge amount of pictures autonomously without any kind of post processing. In the software side, thanks to deep learning techniques, it has been possible to exploit the power of Inception-v4, a deep network built by Google which can be retrained to respond properly to the specific purpose of the crack detection. Moreover, a second deep network has been built from the scratch and trained on the same purpose, in order to make comparisons between the two approaches and in order to choose the most suitable one. Both the networks have been trained with a dataset obtained through the acquisition system and by testing different parameters tuning, so that an optimal solution can be found. All the results are then shown at the end of the dissertation.

Figure 1.1: *University Transportation Center logo*

The following thesis is organized with a first theoretical introduction on the main aspects of Deep Learning and Convolutional Neural Network, in order to have a strong background to understand properly the work behind the project. Than, the crack detector is presented, with a first focus on the acquisition system and then on the networks, by underlines also the main structural characteristics of Inception. At the end, all the test results are shown and a final conclusion tries to estimate the different approaches and tries to define the correct settings of the system, in order to have a prototype ready to be used on real time inspections in whatever urban tunnel.

Chapter 2

Deep Learning

2.1 Introduction

Human brain is one of the most complex structure known and emulating its functionalities could be very difficult or even impossible. The human brain contains around 10^{11} electrical cells called *neurons*, each one acting as a simple information processing unit. Although they are effectively slow in their progressing, the huge parallel work between all these neurons brings to computational powers greater than all the supercomputers of the current days. A neuron works in a very schematic way. It fires an electrical impulse (*potential action*) that moves from the synapse to the next neuron; the type of synapse can discriminate how much the next neuron could be excited. In fact, each synapse has an associated strength which determine the force of the input towards the successive unit. So, the next neuron receives this kind of impulse from different neurons and, if at the end the summation of these impulses overcomes a predefined treshold, it fires in turn.
Based on these notations, a lot of scientist tried to derive a mathematical model for the neuron, called *Artificial Neuron*.

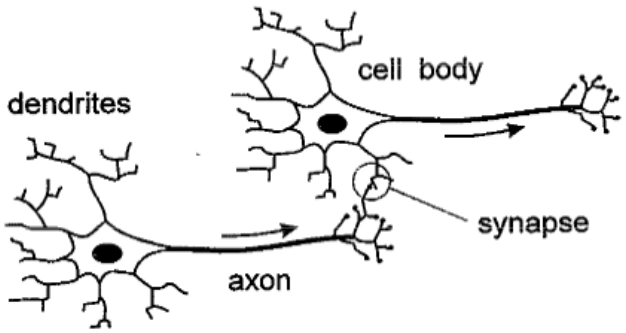


Figure 2.1: *Biological neurons*

2.1.1 Artificial Neuron

The first simple mathematical model of the neurons was introduced by McCulloch and Pits in 1943 [1], as represented in Figure 2.2.

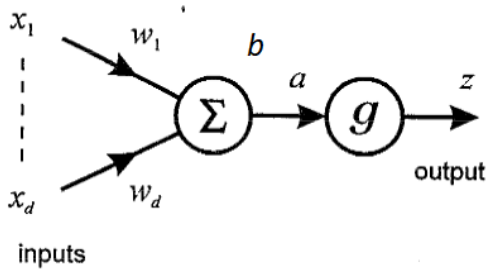


Figure 2.2: *Artificial Neuron of McCulloch and Pitts*

It shows a unit that computes the sum of the inputs $\mathbf{x}_i$ premultiplied by variables $\mathbf{w}_i$(*weights*) and at the end adds a unique term typical of the neuron **b** (*bias*).

$$z = \sum_{i=1}^{n} w_i x_i + b$$

Then, the unit ends with a logistic function **g**, which has to choose when a certain treshold has been overcame or not (*activation function*). For example it can be used a step function or a sigmoid and so on.
In summary, an artificial neuron acts as a non linear function producing a sort of binary response (*all-or-nothing*)

$$z = g(a)$$

2.2 Perceptron

The first step towards the modern deep learning technique was made by Frank Rosenblatt and its **Perceptron** in 1958 [2]. The perceptron is a neuron model not so different in its structure from the one proposed by McCulluch and Pitts and with a step as activation function, but here we have for the first time a proficient work on the learning procedure of the system. The idea of the training starts from the inspirational work of Donald Hebb, who underlined how the learning procedure in the human brain depends strictly on the formation and change of the neurons synapses and elaborated the famous **Hebb's rule**: *"When an axon of cell A is near enough to excite cell B and persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency is increasing"* [3].

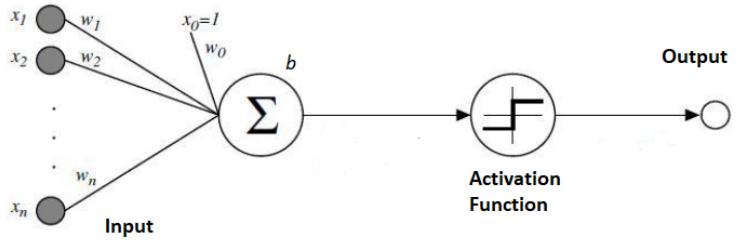


Figure 2.3: *Rosenblatt's Percepton*

The Perceptron does not follow this exact rule, but produces a very simple learning technique, explained in the following algorithm in which is supposed that the Perceptron must classify a function in two class C_1 and C_2.

Calling the variables at the time N: $\begin{cases} w_N = weigths\ vector = [w_1, w2, ..., wn]^{\mathrm{T}} \\ x_N = input\ vector = [x_1, x2, ..., xn]^{\mathrm{T}} \\ b_N = bias \\ y_N = perceptron\ output \\ y_{d_N} = desired\ output \\ \eta = learning\ rate \end{cases}$

1. **Initialization:** At the beginning, start with $w_0 = 0$ and $b = 1$
2. **Computation:** At time n, compute:

$$y_N = step([w_N^{\mathrm{T}} x_N] + b_N)$$

3. **Learning:** Update bias and weights through the following procedure:

$$w_{N+1} = w_N + \eta[y_{d_N} - y_N]x_N$$
$$b_{N+1} = b_N + \eta[y_{d_N} - y_N]x_N$$

where $y_{d_N} = \begin{cases} +1,\ if\ the\ output\ stays\ in\ class\ C_1 \\ -1,\ if\ the\ output\ stays\ in\ class\ C_2 \end{cases}$

4. **Iteration:** Update the values at the time $n+1$ and repeat from point 2 until the perceptron starts to obtain results very similar to the desired output.

The Perceptron is now able to learn simple functions with its characteristics and effectively Rosenblatt himself built an hardware able to classify into 2 separate categories input images of size $20x20$ by using the perceptron rule. But, obviously, this is not sufficient, because only one neuron can not be able to exploit good performance when the task starts to become complex. So, the scientists started thinking if it was possible to generate a perceptron with multiple outputs (figure

2.4), but this new technique did not work very well. The main problem arose from the discover of the impossibility of the Perceptron to learn functions not linearly separable, such as the easy XOR logic function. It was in this moment of difficulties

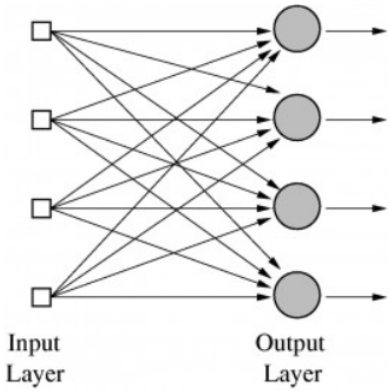


Figure 2.4: *Perceptron with multiple outputs*

that the concept of the **Neural Networks** was born. A Neural Network is nothing more than a series of perceptrons (or neurons in general) organized in layers. In fact, from this moment, we will not have anymore only input and output layer, but also the *hidden layers* (figure 2.5). The improvements obtained with the Networks were immediately huge and for example the XOR function has been implemented with a network with only one hidden layer. If the number of hidden layers is (usually) greater than 2, we can talk of **Deep Neural Networks** and obviously of *Deep Learning*, which is the collection of technique used for training properly a Deep Neural Net.

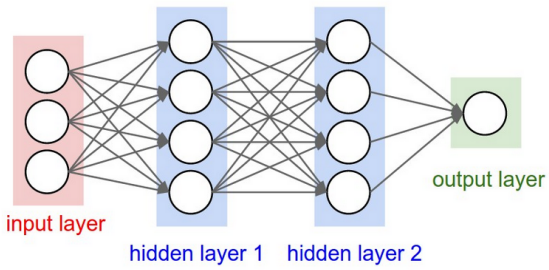


Figure 2.5: *Neural network*

We can start this argument by analyzing the Feed Forward Neural Network, which are one of the most used architecture in Machine Learning. From this analysis we will understand a lot of concepts helpful in the successive section regarding the Convolutional Neural Networks.

2.3 Feedforward Neural Network

A Feedforward Neural Network (FNN) is nothing more than a neural network with multiple layers, each one containing an high number of neurons and connected only in the flow direction (no loop connection). In our analysis we will consider only deep architecture, so structure with a consistent number of hidden layers. This kind of network is a very powerful machine learning tool, because during the years there have been proposed several techniques able to increase the quality of the training and so of the learning. We are going to present the most important ones, or better the main strategies that will be helpful also in the analysis of Convolutional Neural Networks. First of all, we need a notation which we will use during this theoretical dissertation.

- $\mathbf{w}^i_{jk}$: weight connecting the k^{th} element of the $(i-1)^{th}$ layer to the j^{th} element of the i^{th} element
- $\mathbf{b}^i_j$: bias of the j^{th} element in the i^{th} layer
- $\mathbf{x}_i$: generic input
- $\mathbf{y}_i$: generic output
- $\mathbf{y}_{D_i}$: generic desired output
- $\mathbf{a}^i_j$: activation function of the j^{th} element in the i^{th} layer.

Moreover, it is possible to find all this variable collected in vectors in order to compact complex notations. With N neurons in a layer:

- $\mathbf{W}^j_N$: weights in the j^{th} layer
- $\mathbf{B}^j_N$: biases in the j^{th} layer
- $\mathbf{a}^j$: vector of the activations in the j^{th} layer

Usually, the 1st layer takes the name of *Input Layer*, while the last takes the name of *Output Layer*.

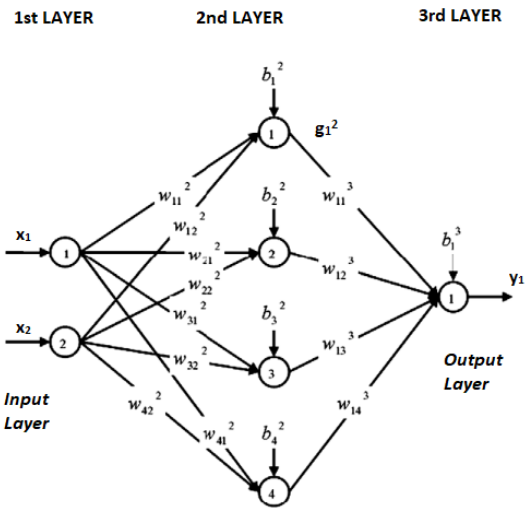


Figure 2.6: *Neural network with 3 layers with the variables labeled following the previous notation*

2.3.1 Activation functions

What we called *element* in the previous section, is actually an artificial neuron unit. In the first architectures, the most used unit was the Perceptron, which in a multilayer network could easily represent all the logic functions. One of the main problem of the perceptron is the activation function (the step), that can produce only binary response of the type Yes/No. So, in the modern Deep Learning application, the neural unit uses different activation functions, each one with some interesting properties that could be used in specific situations. In fact, a deep neural network requires a non linear activation if we want to operate with weights and biases that changes in a non linear fashion and so with a network able to learn also complex patterns. Moreover, we will see how non linear activation functions allow the use of the backpropagation algorithm.
In this chapter we will indicate the argument of the activations functions as:

$$z = wx + b$$

So, the generic output of a neuron unit will be:

$$a(z) = g(wx + b)$$

where g is a particular function here applicated on the neurons activity.

Sigmoid Function

$$a(z) = \frac{1}{1 + e^{-z}}$$

The first advantage with respect to the step function is the continuity of this non

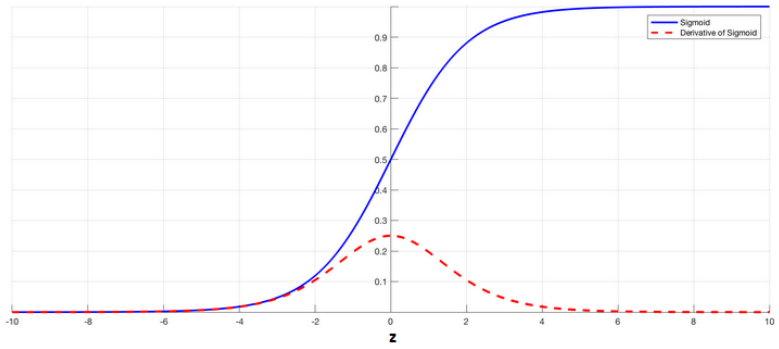


Figure 2.7: *Sigmoid Function and its gradient*

linear function, which permits to obtain non linear continuous response and not only a binary 0/1 output. Another interesting feature is the high magnitude shape of the gradient of the function (figure 2.7) in the ranges (−3,3) of z. This means that in this range small changes of z bring large variations of $g(z)$, so the output will be pushed towards the extremes of the shape; and this could be a cool feature in a double class classification problem. But there are also negative aspects: in the region far from (−3,3) the gradient is very small (bad aspects for the learning procedure, as we will see in the following sections) and the function is not symmetric around 0, so it produces only positive outputs. In order to solve this last problem, the *tanh* function is introduced.

Tanh Function

$$a(z) = \frac{1 - e^{-2x}}{1 + e^{2x}}$$

The tanh function has basically the same shape and characteristic of the sigmoid, but with some improvements. First of all, the function spans from (-1,+1), so it catches also megative outputs; on the second, the gradient has the same aspect, but with higher module on the (-3,3) range of z (this brings to higher response in a classification problem). However, it still preserves the problem of the small gradient in the range far from (-3,+3). In conclusion, tanh is an evolution of the sigmoid and for this reason is always preferred to it.

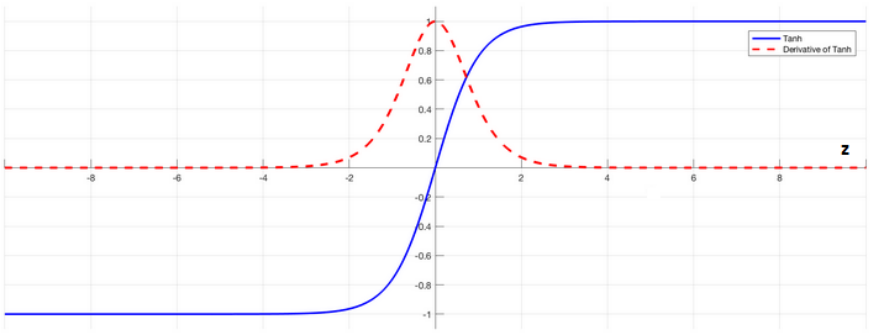


Figure 2.8: *Tanh function and its gradient*

ReLU and Softplus Function

Rectified Linear Unit function (ReLU) is practically always used in the unit of the hidden layers in modern deep neural networks.

$$a(z) = max(0, z)$$

Why ReLU is so much used? First of all it is non linear and we are starting to

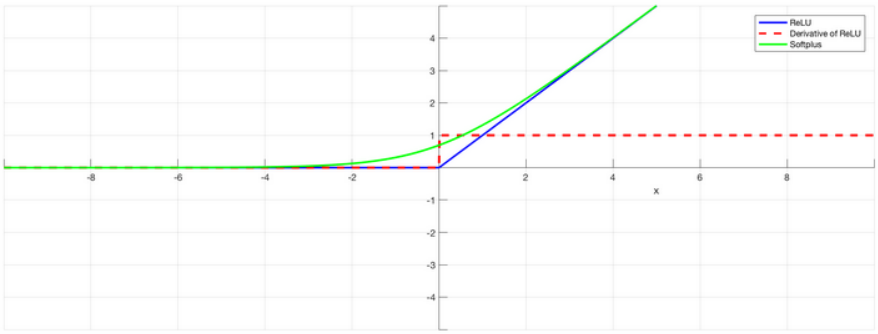


Figure 2.9: *ReLU function, its gradient and Softplus function*

appreciate how important is having non linear activations. But, mostly important, ReLU has the big feature to not activate always the neuron. In fact, if z is negative, than the response is 0 and the neuron will not be activated; this is something that we will appreciate as a good learning optimization technique for the network. But, as the previous examples, the problem regards the gradient, that here completely disappears in the negative range of z. An idea for solving the problem is to use the *Softplus* function.

$$g(z) = ln(1 + e^x)$$

As we can appreciate from figure 2.9, the Softplus as a shape practically equal to the ReLU, but improves the gradient magnitude in the early negative range of z.

But this is not such a big improvement and the expression of the function is very complex involving log and exp. So, in an hypothetical use, the computational cost will be higher than the effective improvement with respect to the simple ReLU function.

Leaky ReLU Function

$$a(z) = \begin{cases} az, & \text{x<0} \\ z, & \text{x>0} \end{cases}$$

This could be the best way to solve the problem of the gradient related to the

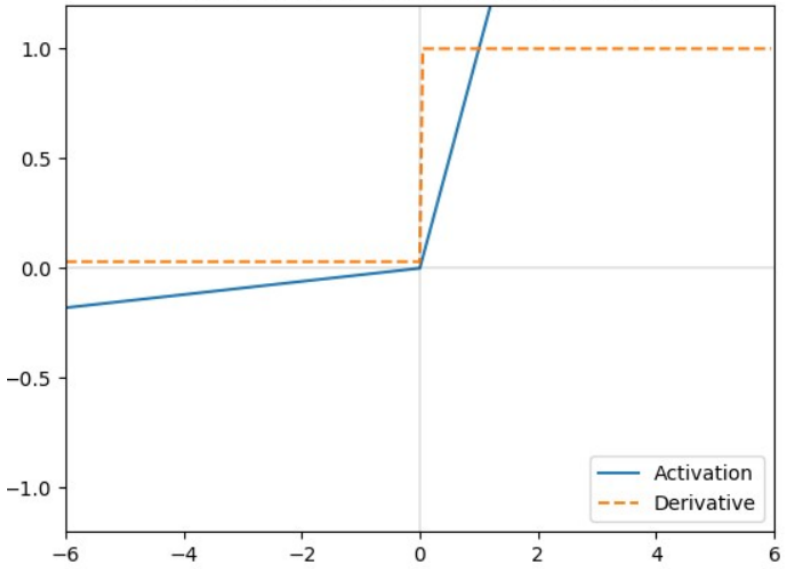


Figure 2.10: *Leaky ReLU function and its gradient*

ReLU function, by choosing a value of a similar to 0.001 or so. In fact, here the gradient in the negative range is not null, but at the same moment we lose the good feature of ReLU of non activating always all the neurons.

Softmax

This is probably one of the most important activation function we will see and in fact we will use it a lot in our Convolutional Neural Networks.

$$a(z)_j = \frac{e^{z_j}}{\sum_{i=1}^{k} e^{z_k}}$$

where: $\begin{cases} k = current\ \ layer \\ j = unit\ in\ the\ k^{th}\ layer \end{cases}$

In few words, this function assumes a probabilistic aspect, because it computes the probability of an output to be part of a certain class. For this reason, it is used as activation function in the last layer of a deep neural network used for classify an input inside predefined classes.

2.3.2 Learning procedures

The architectural structure of a Deep Neural Network can be thought as a multilayer perceptron with different activation functions. Now, it is the moment to explain what is the process that brings a network to *learn*. With 'learning' we indicate the capacity of a network to update autonomously the values of weights and biases in order to improve the final outputs and for example to be able to replicate the behavior of a function or to classify correctly a certain input. There are two main learning procedures in Machine Learning:

- **Supervised Learning:** With supervised is intended the procedure in which is available a complete dataset containing couples of input and output. In this way, the network can learn by training with this set and trying to replicate the exact output corresponding to the proper input. So, the objective of our network will be to make comparison between desired output and real output and set weights and variables in order to obtain results as much as closer to the desired one. This is the technique that we will use in our application.
- **Unsupervised Learning:** Here, the network does not have a prepared dataset from which it can learn. So, it will try to update the variables in order to extract as much information from the input as possible. It is clearly a more difficult approach.

Whatever the learning methodology chosen, it is required a mathematical approach for learning and in this case it necessary to introduce the *cost function*. A cost function can be considered as the reference of a learning procedure, whose objective will be to minimize it.

Cost Functions

There are several choices of cost functions presented in literature, but there are some requirements that must be followed in order to be used in technique such Gradient Descent which allows us to move on in the learning procedure.
First of all, the cost must be written in an *average* mode with respect to cost computed for a single training example

$$C = \frac{1}{n} \sum_{x} C_x \tag{2.1}$$

On the second, the gradient of the function will be essentially in all the algorithm we will use, so it must have a form which depends only on the activation function in the last layer, so the output activation function. In literature, there are presented a lot of cost functions that can be chosen, but in this thesis there are shown the most popular ones, nevertheless the ones that can help us in out dissertation.

- **Mean Square Error:** MSE, or also called *Maximum Likelihood*, has this form in a network with L layers and n inputs:

$$C(x, W_N^L, B_N^L, y_D) = \frac{1}{2}\sum_{j=1}^{n}(y_j - y_{D_j})^2 \tag{2.2}$$

  In a network with only 1 output, the gradient has this form:

$$\nabla C = (y - y_D) \tag{2.3}$$

- **Cross Entropy:** Cross Entropy cost function, also known as *Bernoulli negative log-likelihood*, has this form:

$$C(x, W_N^L, B_N^L, y_D) = -\sum_{j=1}^{n}[y_{D_j} ln y_j + (1 - y_{D_j}) ln(1 - y_j)] \tag{2.4}$$

  Again, in an network with only 1 output:

$$\nabla C = \frac{(y - y_D)}{y(1 - y)} \tag{2.5}$$

Gradient Descent

The cost function gives us a clear path for the learning procedure: we need to update progressively weights and biases in order to move towards outputs similar as much as possible to the desired ones. If we consider equation 2.2 as our cost function, it is easy to understand how we can catch this desired condition when $C(w, b, x, y_D) = 0$. So, the aim of our training is to **minimize the cost function**. How we can do that? It is very hard to collect all the aspects of a neural network inside an unique algorithm or procedure. So, firstly, we have to focus ourselves on the mathematical aspects and so the better way is to find a procedure which allows us to minimize a function, forgetting for a moment that this function is inside a complex neural architecture. One of the best technique that we can use is the *Gradient Descent* algorithm.
We start defining a cost function $C(v)$ depending on variables $v = v_1, v_2, ..., v_n$. For simplicity in the derivations, we can use a function depending only on two variables v_1, v_2, but at the end we are going to generalize the results for neural networks, in which the number of variables is surely much greater than 2. If we apply a little

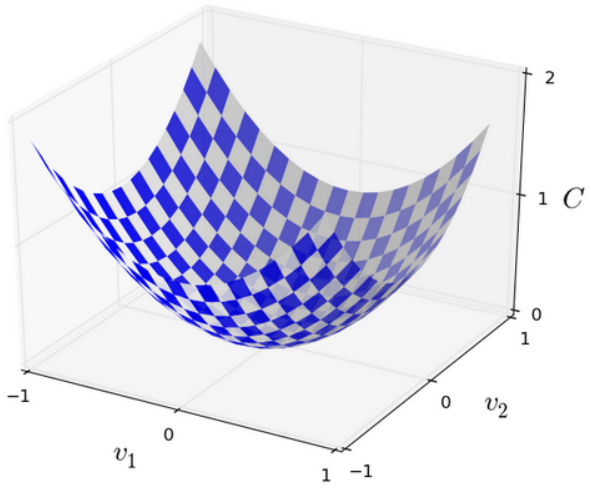


Figure 2.11: *Examples of a cost function C(v) depending from two variables* v_1 *and* v_2

change in v_1 and v_2, we can obtain also a change in C:

$$\Delta C = \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 \tag{2.6}$$

If we impose: $\begin{cases} \Delta v = (\Delta v_1, \Delta v_2)^{\mathrm{T}} \\ \nabla C = (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}) \end{cases}$

then:

$$\Delta C = \nabla C \Delta v \tag{2.7}$$

In this form, our objective becomes obtaining progressively a negative ΔC by modifying Δv: in this way, C will move towards the minimum.
By choosing:

$$\Delta v = -\eta \nabla C$$

Then:

$$\Delta C = -\eta ||\nabla C||^2 \tag{2.8}$$

So, due to the fact that $||\nabla C||^2$ is always a positive variable, we can be sure to obtain a ΔC negative. At the end of the algorithm, we know how to choose the next value v' starting from v.

$$v' = v - \eta \nabla C$$

In this approach we see for the first time η, known as *Learning Rate*, something that will become one of the most important hyperparameter inside the training procedure of a neural net. This parameter must be chosen properly: it must not be too high otherwise the approximation in 2.7 does not work, but at the same time

it must not be too low otherwise the change in Δv is very small and the training proceeds slowly.
With the obtained results, we can use the Gradient Descent also in the Neural Networks contest, by remembering that our cost function will depend on the values of weights and biases. The procedure has the intention to be iterative, so we need to train the network with different inputs and updating the variables each time a new output is produced. In the case of a network with K weights and L layers:

$$w_k \to w'_k = w_k - \eta \frac{\partial C}{\partial w_k} \quad \forall k = 1, ..., K$$

$$b_l \to b'_l = b_l - \eta \frac{\partial C}{\partial w_l} \quad \forall l = 1, ..., L \tag{2.9}$$

Stochastic Gradient Descent (SGD)

The Gradient Descent algorithm could be very powerful, but it has a main drawback. The cost function C is always an averaging term between all the costs obtained with the training inputs. But in neural networks the number of input is usually very high, so its computation (that has to be performed for each iteration) becomes hard time consuming. The easiest way to overcome this problem is to introduce the *Stochastic Gradient Descent*.
The idea is to estimate ∇C by computing ∇C_x for a small set of input: working with them can well approximate ∇C with a great gain in time performance.
We divide the set of input X in m smaller set $X_1, X_2, ..., X_m$ chosen randomly: we called them *Mini Batches*. The size of the mini batches will be one of the other fundamental hyperparameters because, regardless the gradient optimizer we could choose, this technique will be always used. So, we can expect something like this:

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C \tag{2.10}$$

And, rewriting 2.9

$$w_k \to w'_k = w_k - \frac{\eta}{m} \frac{\partial C_{X_j}}{\partial w_k} \quad \forall k = 1, ..., K \quad \forall j = 1, ..., m$$

$$b_l \to b'_l = b_l - \frac{\eta}{m} \frac{\partial C_{X_j}}{\partial w_l} \quad \forall l = 1, ..., L \quad \forall j = 1, ..., m \tag{2.11}$$

2.3.3 Backpropagation

We have already seen two different methods for providing an algorithm to the learning activity. Nevertheless, we still have to face the problem of the computation

of the gradient of the cost functions. For this reason, in 1986 was proposed and accepted the *Backpropagation* algorithm [4], which showed a lot of improvements in the training procedure. Essentially, it provides an expression for the gradient with respect to any weights or biases in the network, allowing us to overcome all the difficulties in the computation. Moreover from this algorithm, we can have a clear idea on how a change in the variables is reflected in the network behavior. Before explaining Backpropagation, we need some clarifications.
From the nomenclature previously defined, we can exploit the detailed matrix form of the activations in a generic layer i of the network, which will be used in the demonstration

$$a^i = a^i_1, a^i_2, ..., a^i_n, \quad with\, n:\ number\, of\, neurons\, in\, the\, layer$$

$$a^i_j = g(\sum_k w^i_{jk} a^{i-1}_k + b^i_j) \tag{2.12}$$

$$a^i = g(w^i a^{i-1} + b^i) \tag{2.13}$$

This expression shows us how each activation is strictly correlated to the previous one. It is also better to recall:

$$z^i = w^i a^{i-1} + b^i$$

$$a^i = g(z^i)$$

If we want to understand properly the backpropagation, we have to define a new quantity, the **error** δ^i_j of the j^{th} neuron in the i^{th} layer. When a neuron is excited by an input, it has to react by producing $a^i_j = g(z^i_j)$, but instead it produces an output $g(z^i_j + \Delta z^i_j)$. This variation propagates through the next layers, causing an overall variation in the cost equal to $\frac{\partial C}{\partial z^i_j}\Delta z^i_j$. In general, this variation can be very useful for us, because it allows to produce a negative variations in the cost, by simply choosing it with the opposite sign with respect to the derivative of the cost. This is now clear how it is important this procedure and so the error is defined as:

$$\delta^i_j = \frac{\partial C}{\partial z^i_j} \tag{2.14}$$

We can obviously recall the vector δ^i as the vector of the errors inside a layer i. So, backpropagation permits to compute δ^i_j and after that $\frac{\partial C}{\partial x^i_{jk}}$ and $\frac{\partial C}{\partial b^i_j}$, which is the final aim of all the learning procedures previously defined.
We are now ready to introduce the algorithm, but firstly we need to derive the *four fundamental equations of backpropagation.*

1. The first equation provides us the expression of the error in the last layer of the network L

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} \tag{2.15}$$

By applying the chain rule

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L} \quad where\, k : neurons\, in\, the\, layer \tag{2.16}$$

Taking into account that ∂a_k^L depends only on ∂z_j^L when $j = k$, we can rewrite:

$$\delta_j^L = \tfrac{\partial C}{\partial a_j^L}\tfrac{\partial a_j^L}{\partial z_j^L} \tag{2.17}$$

$$\delta_j^L = \tfrac{\partial C}{\partial a_j^L} g'(z_j^L) \tag{2.18}$$

This is the first equation, which exploits the component of the error in the last layer. It is an easy way to show how change the behavior of the network and it is also very easily computable. $\frac{\partial C}{\partial a_j^L}$ says how much changes the cost with respect to the activation of the j^{th} neuron and it is nothing more than a derivative of a function. $g'(z_j^L)$ shows how the activation function changes with respect to the weighted input.

2. The second equation gives us the expression of the error in a layer with respect to the error in the successive one

$$\delta_j^i = \tfrac{\partial C}{\partial z_j^i} \tag{2.19}$$

$$\delta_j^i = \sum_k \frac{\partial C}{\partial z_k^{i+1}}\frac{\partial z_k^{i+1}}{\partial z_j^i} \tag{2.20}$$

$$\delta_j^i = \sum_k \frac{\partial z_k^{i+1}}{\partial z_j^i}\delta_k^{i+1} \tag{2.21}$$

By notice that:

$$z_k^{i+1} = \sum_j w_{kj}^{i+1} a_j^i + b_k^{i+1} = \sum_j w_{kj}^{i+1} g(z_j^i) + b_k^{i+1} \tag{2.22}$$

$$\tfrac{\partial z_k^{i+1}}{\partial z_j^i} = w_{kj}^{i+1} g'(z_j^i) \tag{2.23}$$

Substituting in 2.21:

$$\delta_j^i = \sum_k w_{kj}^{i+1}\delta_k^{i+1} g'(z_j^i) \tag{2.24}$$

This is the final equation that we can rewrite in a matrix form:

$$\delta^i = ((w^{i+1})^{\mathrm{T}}\delta^{i+1}) \odot g'(z^i) \tag{2.25}$$

With these first two equations combined we can compute all the errors in a network *backward*, because we move from the last to the first layer.

3. The third equation relates the changes in the cost function with respect to the biases with the erro
$$\frac{\partial C}{\partial b_j^i} = \delta_j^i \tag{2.26}$$
They are exactly equal.

4. The fourth equation relates the changes of the cost with respect to the weights to the error
$$\frac{\partial C}{\partial w_{jk}^i} = a_k^{i-1}\delta_j^i \tag{2.27}$$
The demonstration for 2.26 and 2.27 can be easily derived with the same technique of the chain rule used in the first two equations.

Algorithm

We have now all the instruments for explaining the backpropagation algorithm.

- **Input:** Given an input x, derive the activation on the first layer a^1
- **Feedforward:** Starting from a^1 compute the activations for all the layers antil the last $a^L = g(z^L)$
- **Output error:** Now, compute the error in the last layer δ^L
- **Backpropagate the error:** Using the second equation of the backpropagation, derive the errors in each layer
- **Output:** Using the last two equations, derive the changes of the cost with respect to weights and biases

At the end, after backpropagation we are able to use easily the (stochastic) gradient descent algorithm to update the variables in the network.

2.3.4 Other gradient optimizers

We already know how to use properly the Stochastic Gradient Descent algorithm by taking advantage of the backpropagation. Actually, backpropagation can be used in many other gradient optimizer technique, which could bring much better results than the gradient descent, mostly in the speed of the learning procedure. We are going to see three other gradient optimizers between the most used one, but there are many other choices that can be exploited through the literature.

Momentum

Momentum based gradient descent is an optimized version of the simple gradient descent. Intuitively, it tries to conduct the descending motion of our cost function towards its minimum to the motion of a ball inside a valley. In this approach, we

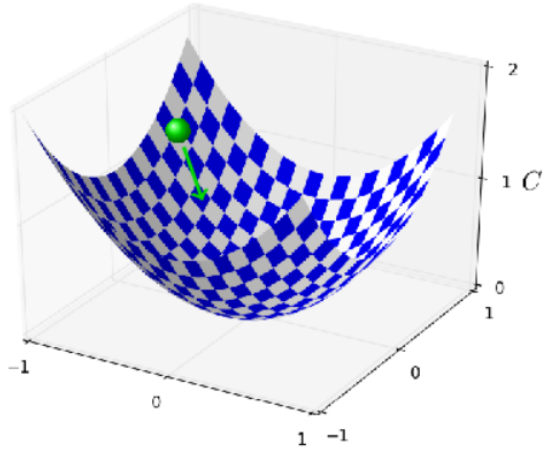


Figure 2.12: *The point represents our cost function in a particular moment of the procedure. The arrow suggests the direction to be taken in order to reach the minimum*

want to formulate an expression of the *velocity* of our changes in the variables (as always weights and biases), in order to control how fast our cost is varying. If we call w the variables and v the associated velocity:

$$v \to v' = \mu v - \eta \nabla C \tag{2.28}$$

$$w \to w' = w + v = w + \mu v - \eta \nabla C \tag{2.29}$$

μ is called *Momentum Coefficient* and can be considered another hyperparameter if the Momentum optimizer is used. Basically, setting this value between 0 and 1 allows us to set the value of the speed of the gradient variation,: so, Momentum is nothing else than a gradient descent much faster. However, this value must be chosen properly, otherwise we can speed up too much the procedure and we risk to overcome or completely miss the minimum of the cost.

Nesterov accelerated gradient

Nesterov accelerated gradient [5] is a slightly modification of the Momentum technique, which prevent the situation in which the optimizer is stacked in a local

minimum and can not overcome it to reach the global one.

$$v \to v' = \mu v - \eta \nabla C \tag{2.30}$$
$$v' \to v'' = \mu v' - \eta \nabla C \tag{2.31}$$
$$w \to w' = w + \mu v'' = w + \mu^2 v - (1+\mu)\eta \nabla C \tag{2.32}$$

Adam

Adam [6] introduces new variables in the construction of the algorithm for the optimization of the gradient.

- *Decaying average of the past squared gradients* **v**

$$v \to v' = \beta_2 v + (1-\beta_2)\nabla C^2$$

- *Decaying average of the past gradients* **m**

$$m \to m' = \beta_1 m + (1-\beta_1)\nabla C$$

These quantities take trace of the evolution of the gradient also through its squared value. We are interested in the bias-corrected value, because we do not want to be influenced by the value of β_1 and β_2 (usually very near to 1).

$$\hat{m}' = \frac{m'}{1-\beta_1}$$

$$\hat{v}' = \frac{v'}{1-\beta_2}$$

And finally:

$$w \to w' = w - \frac{\eta}{\sqrt{\hat{v}'} + \epsilon}\hat{m}' \tag{2.33}$$

ϵ is another coefficient usually very low ($\approx 10^{-8}$)

Weigth and biases initialization

As seen until now, all the gradient optimizers try to estimate the weights and biases in an iterative way and so it is necessary to find a good initialization for these variables.
The first approach could be initializing both of them as independent Gaussian random variables, normalized to have mean 0 and standard deviation 1. This is a very good solution, that in literature brought decent results. But there is another strategy a little bit precise. In fact, the previous idea is not so effective when we are in front of layers with a high number of neurons. In that case, the weighted

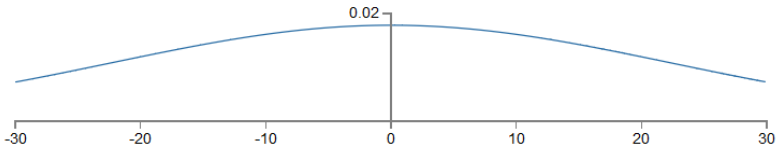


Figure 2.13: *Hypotetic shape of z with weights initializated with standard deviation 1*

input z will have also a gaussian shape, but very broad (Figure 2.13). This is not so good for us, because it is quite likely that $|z|$ will be very large ($z << 1$ or $z >> 1$). This will bring to an activation equal to 0 or 1 that in one of the first layers is synonymous of *neuron saturation*. Neuron saturation is one of the greatest problem in the training procedure: neurons producing very early results equal to 0 or 1 means that they are not learning at anymore, because they have already reached a final result. So, it is likely having instruments, but not using them.
An interesting idea would be initialize the weights with standard deviation equal to $\frac{1}{\sqrt{k}}$, with *k=number of neurons*. This could bring to a sharper shape of z, so much more expectations for the learning procedure.

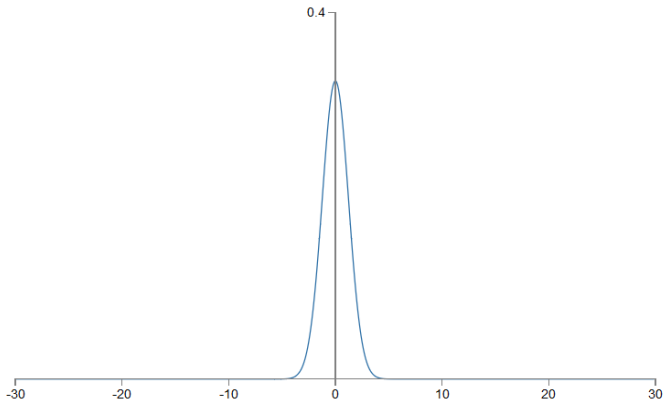


Figure 2.14: *Hypotetic shape of z with weights initializated with standard deviation modified*

2.3.5 Obstacles in learning

With the instruments already presented, we are able to construct a basic deep neural network, but there is already something that we need to explain. In particular, during the training of a deep network, could occur some problems that slows down the entire learning procedures. We will face two of them, the most known and dangerous ones. There will be presented also some strategies to solve them, but it is

important to explain here something crucial in the neural network world: in general, there are not fixed rules to obtain surely perfect results from a neural network. It is required a lot of experience and the ability to understand immediately where the problem is. Usually, the good setting of hyperparameters is the main starting point for a decent training procedure and a practical approach to this problem will be presented also in this thesis.

Vanishing Gradient

When we train deep neural networks, it is easy to exploit immediately one thing: as much as we proceed back to the input layer, as lower is the learning capability of a layer. This means that earliest layer will surely learn slowly than successive ones. This is the so called *Vanishing Gradient* problem, because the fall of the learning capability is due to the gradient of the cost that progressively decreases going deeper in the backpropagation. But the most interesting thing about this problem is that we can not avoid it: in a deep neural network the gradient is always **unstable** and it vanishes or **exploit** (gradient larger in the earlier layers). What we can do is understanding the problem.
For simplicity, we can think to a neural network with 4 layers, each one with only 1 neuron with sigmoid activation function.

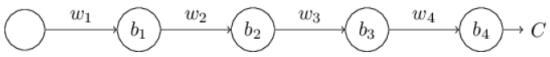


Figure 2.15: *Example of a 4 layer neural network*

We are interested in the expression of the gradient of the cost with respect to the bias in the first layer b_1:

$$\frac{\partial C}{\partial b_1} = g'(z_1)w_2g'(z_2)w_3g'(z_3)w_4g'(z_4)\frac{\partial C}{\partial a_4} \tag{2.34}$$

In this expression, we can see that, except for the final term in the last layer, depends on a series of elements of the type $w_jg'(z_j)$. If we want to understand how this term behaves, we have to recall the derivative of the sigmoid (g) The derivative reaches the maximum at $g'(0) = \frac{1}{4}$. If we start from the common initialization of weights as random variables with mean 0 and standard deviation 1, we can easily think that $|w_j| < 1$ and so $|w_jg'(z_j)| < \frac{1}{4}$. Notice tha, we can formulate the equation for the third layer:

$$\frac{\partial C}{\partial b_3} = w_3g'(z_3)w_4g'(z_4)\frac{\partial C}{\partial a_4} \tag{2.35}$$

The two expression are practically the same, but with the difference that 2.34 has two more $w_jg'(z_j)$ block. And now, it is almost clear how the vanishing gradient

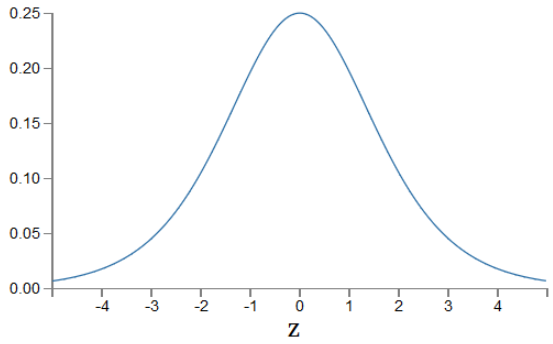


Figure 2.16: *Derivative of the sigmoid function*

occurs, because the earliest layers will also have an expression with terms progressively lower than the successive ones. And it is at the same time clear how this problem can not be solved, because it is something proper of the network itself.

Data overfitting

The *Overfitting problem* is very easy to explain: it occurs in the moment in which a network start to recognize specific features in the training input, without learning the general features that we want to highlight. So, it can not recognize these features in other input, even if during the training the results are very good. These characteristics of the overfitting makes it very easy to be recognized by simply taking a look to the accuracy of the network during train and test. Usually, high results in train not confirmed in test exploit the overfitting. The first ideas for reducing this problem are very immediate: increasing the number of input or reducing the depth of the network. At the same time is very clear how this technique are not good at all, because they reduce the strength of the network, so we will not gain anything in terms of performance. Fortunately, there is another technique for facing this problem, the so called *Regularization*.

2.3.6 Regularization

In general, regularization is nothing more than a modification in the cost function, which helps in avoiding the overfitting. The most used type of regularization is the L2 type.

L2

L2 regularization consists in adding in the expression of the cost function another term, called *regularization term*. For example, the form of the regularized cross

entropy is:

$$C = -\frac{1}{n}\sum_{x_j}[y_j ln a_j^L + (1-y_j) ln(1-a_j^L)] + \frac{\lambda}{2n}\sum_w w^2 \tag{2.36}$$

where n is the size of the input set and λ is the se called *regularization coefficient*, which in this case became an other hyperparameter.

Dropout

Probably the most successful regularization technique is the *Dropout* [7]. Actually, this is not a standard regularization procedure, because it does not change the cost function at all, but acts in a different way: during each cycle of the learning procedure, the neurons in a dropout regularized layer are not activated simultaneously. In this way, we can erase all the relation between close neurons and avoiding situation in which they learn only particular pattern between them, one of the principal cause of overfitting. In summary, we force the neurons to learn more complex pattern when they are not always linked to the same set of other neurons. So, when a dropout regularization is inserted ina layer, the hyperparameter *Dropout rate* must be specified.

2.4 Convolutional Neural Network

In this thesis, the deep learning network which will be used is the **Convolutional Neural Network**(CNN). CNNs were born (or better has been used for the first time) in 1998 [8] and their structure were based on the learning approach of the human brain, as we have already presented for the FNN. Nowadays, CNNs are strictly related to computer vision and AI and are used to process images and classify them autonomously or to detect specific pattern inside them.

2.4.1 CNNs elements

In order to understand properly this kind of network, we need to define some characteristics and create layers based on it. In fact, we will see how a convolutional neural network is composed by a cascade of layers pretty different one from each other.

Input/Output volumes

Essentially, each input image is seen as a matrix of pixels and so, it is necessary to define the size this inputs. An image size is commonly determined through its height and width, with the so called *resolution*; for example, a 4 bit image collocates 16 pixels in the range [0,15] and can be represented as a $4x4$ image. But usually the images are colored and so it must be added another dimension, the depth or also called the *channel size*. A generic colored image is represented through the RGB convention, so it must be introduced a depth of dimension equal to 3. For this reason we use the term *volume* when we talk about the dimension of an input or an output. For example, a generic $4x4$ colored image has a volume of $4x4x3$.

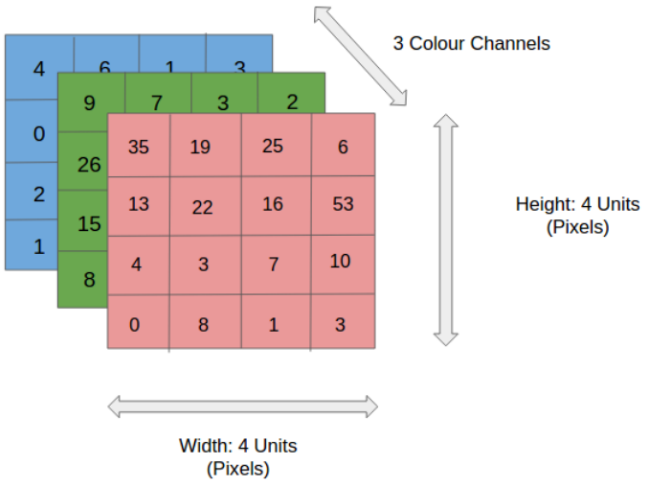


Figure 2.17: *Volume of a 4x4 colored image*

Feature

A feature is a particular element or combination of element we want to extract from the input image. In computer vision, learning starts from the recognition of patterns inside an image, for example the edges of an object or the eyes inside a human face image. Usually, a CNN is structured in order to recognize simple features in the earlier layer, while the successive ones try to locate something more complex.

Filters (or Kernels)

The *filter* is the first element of the layered architecture of CNNs. It tries to extract the features from an image. In particular, it is nothing more than a small matrix which has to be compared with the input, in the operation called *convolution* (equation 2.37) and this explains the name of the network.The size of the filters is strictly related to another key concept in CNN, the *Receptive Field*. We can think to the pixels of an image in a layer as neurons which must be connected to neurons in the successive layer; it is practically impossible creating a full connection between all of them, because the size of the images will surely lead to high computational effort. So, in CNNs, it is a set of of neurons in the input (receptive field) that communicates with the next layers and it is in this field where the filters operates. Essentially, the operation of convolution consists in multiple elementwise the filter matrix with the receptive field (the mathematical operation of *cross convolution*) and in projecting the result in the so called *activation map*. The activation map will become the input for the successive layer.

$$g(b + \sum_{l=0}^{K} \sum_{m=0}^{K} w_{l,m} a_{j+l,k+m}) \tag{2.37}$$

The equation 2.37 shows how, in a KxK receptive field, we have KxK weights and only 1 bias, so it is easy to deduct how a receptive field is considered as the unit of a CNN layer.
Obviously, the filter has to slide all over the input image moving through the different receptive fields. In fig 2.18 is shown an example of the filter operation.

2.4.2 CNNs Layers

With these notions, we can now introduce the different layers type used in a CNN architecture

Convolutional layer

Convolutional Layer is the key element of a CNN. Essentially, it performs the operations of the filters previously introduced. So, when a new Conv Layer is

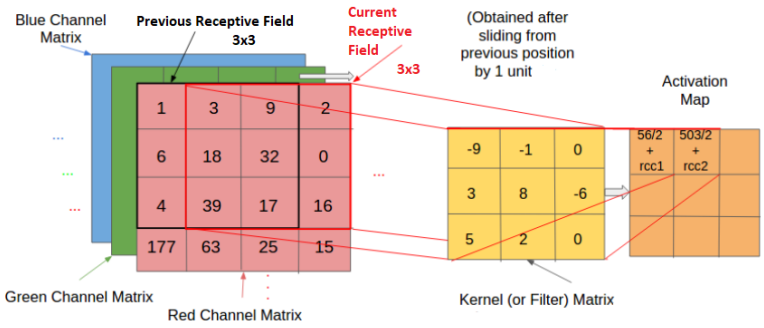


Figure 2.18: *Working principle of a filter*

introduced, it must be defined the size of the filters and the *stride* size, which is the number of pixels on which the filter moves during the sliding movement through another receptive field. It is important to underline an important features: a convolutional has always more than one filter, because in a single layer we want to extract more than feature from the input. So, a convolution will always produce an output with depth equal to the size of the so called *filter space*.
Usually, a convolutional layer is always combined with the *Zero Padding* procedure. If we think at the operation of convolution, effectively it produces an output with the dimension reduced with respect to the input(in width and height). This is not so good in an hypothetical deep network, because too many filters will surely bring to a strong reduction in the image size. The zero padding technique consists in virtually increasing the size of the input,by surrounding it with a series of pixels with value equal to 0, in order to obtain an output with the same size of the original input. This is a complete "safe" operation because the 0 pixels will not affect at all the convolution.

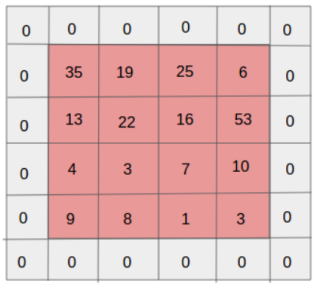


Figure 2.19: *Zero Padding example*

Pooling layer

In general, the reduction of the image size during the flow line of the network could be very useful, because at the end we will probably need a size of the output smaller than the input. The best way to do it in a controlled way is to insert in the network some *Pooling Layers*.
The idea of Pooling layers is to reduce the dimension of the image by deleting the pixels with "less" importance. So, the operation is practically the same of the filters: it slides a certain receptive field through the input and transform a set of pixels into a single one with a certain characteristic.
For example, the most used pooling layer is the **Max Pooling**. Here, a certain receptive field is compact into a single pixel with value equal to the maximum inside the field. So, also in the definition of a pooling layer must be declared the sizes of the "filters" and the stride.

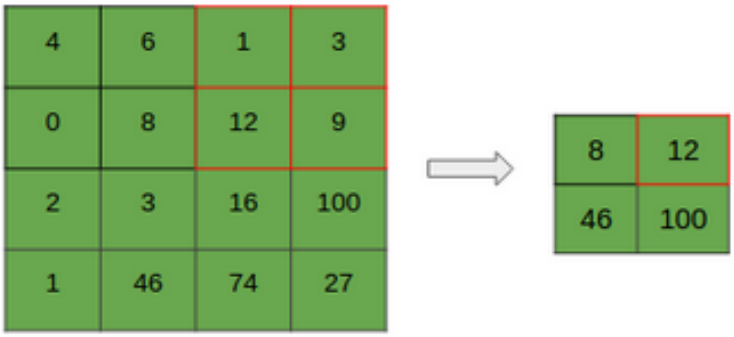


Figure 2.20: *Max Polling example with stride equal to 2*

Fully connected layer

In a CNN, it is common to find also fully connected layers, similar to the ones presented in the section dedicated to the FNN. These are fundamental in the moment in which we want to obtain an output from the network that is not an image. In that case, the output has a linear shape, totally different from the matrix approach of the convolutional layer, which explains the reason behind the presence of fully connected layers.
In our particular application, we will face a classification problem, so a configuration in which the network has to insert an image inside predefined output classes (two in our case). In the following image it is presented a generic structure of a very simple CNN for classification purpose. In this example we can see a lot of characteristics very common in all the CNN architecture present in literature: the convolutional layers have neurons with ReLU activation function; pooling is always

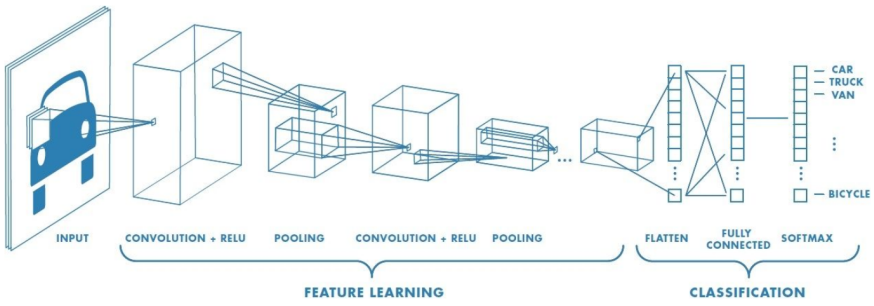


Figure 2.21: *Example of a CNN for vehicle classification*

interposed between two successive convolutions; there are 2 or 3 fully connected layer, all of them with dropout regularization except for the last one with a softmax activation function, that is the best choice for classification purposes.

2.5 Batch Normalization

Basically, CNNs follows all the other characteristics of a common FNN, with the differences of the convolutional layer previously defined. But we will continue to present the classical hyperparameters and the training procedure remains practically the same. In general, a CNN training is much slower due to the presence of images, but there is a technique (actually present in all the neural networks), which helps a lot in accelerate the training and in obtaining better performances.
In general, it is a good practice to normalize the input data (in our case the pixel value of a matrix) in a network. There are two main normalization technique.

- **Normalization:** It consists in shift all the input data in a range not so wide and usually very low. The most common choice is to normalize the input between [0,1]
- **Standardization:** This is a different type of normalization, which produces a result like:

$$x_{norm} = \frac{x - E(x)}{\sigma(x)}$$

  where $E(x)$ and $\sigma(x)$ are respectively the mean and the standard deviation of the input. It is like normalize the data with 0 mean and 1 standard deviation.

The first normalizing procedure must be always done before the input layer of the network, so it become a part of the preprocessing phase. This procedure is very important because in a lot of case the input is in a wide range of values and the output of the network should be in a total different range. In a situation like this,

it is very probable that this strong change in the input values during the several layers could generate an unstable gradient and so fall into the already explained problem of vanishing or exploding gradient. Moreover, non normalized input data cause a slowdown in the training procedure. So, it is a good practice to normalize the input data in a not wide and small range.
But the problems can not be completely solved. In fact, even with normalized input, there could happen that during the optimization gradient procedure, some weights would be updated too much with respect to other ones. And having weights and biases too much different in different layer can cause again the problem of unstable gradient. So, it is here that we can introduce the **Batch Normalization** technique. In practice, Batch Norm is quite like using another layer whatever we want, so we can arbitrary choose which layer batch normalizes. It acts on the activation function and, before applying it, it performs a standardization:

$$z'_{BN} = \frac{z - E(z)}{\sigma(z)} \tag{2.38}$$

An then, it adds to coefficient:

$$z_{BN} = \beta z'_{BN} + \xi \tag{2.39}$$

In this way, by choosing properly β and ξ, we can produce in the different layers weights and biases always normalized, with controlled means and variances. This is a crucial technique, because in literature [9] it has been demonstrated how this technique drastically increase the speed of the learning procedure, producing much more stable results with respect to non batch normalized networks. This is the basic theory about CNN, which can give the generic notions in order to be able to work with them. In literature, a lot of novelties and variations on these architectures are present and we will use in our application one of them, the Inception architecture.

Chapter 3

Autonomous crack detector

As explained in the introduction of this thesis, autonomous crack detection system is something really discussed in literature in recent years. So, we want to present the current state of art, in order to motivate the decision made for the model used in our project. In fact, state of art helped us in thinking the correct setup for the images caption in terms of correct cameras, caption techniques, lights and so on and in appreciating the good results obtained through deep learning technique applied to similar problems.

3.1 State of art

Tunnel inspection is considered an interesting problem to be studied, because in a lot of situations it is not only difficult, but also dangerous for a human beings. So, creating a system that autonomously can detect problems inside a tunnel by simply looking given pictures can bring extreme positive effects, also on the time costs side.
The first proposed models in literature are based on computer vision technique, so not on Deep Learning. For example, in 2007 *Yu et al* [10] built a robotic system able to detect images inside a tunnel and then process them in order to find concrete cracks. This mobile robot was remotely controlled in order to maintain constant distance from the tunnels walls and it was equipped with CCD cameras, in order to get images in best condition as possible. Moreover, the capture system was also equipped with illuminators and encoders for computing speed and position of the tunnel. From the software point of view, the images are processed with computer vision technique in order to find images with concrete tunnels and measuring them: even if the measurements of cracks were very good, the system was very deficient in crack detection, with an error rate of around 80%. *Lee et al* [11] instead proposed a new method (*Image-mosaic technology*) for reconstructing 3D tunnels surfaces

from images taken by humans with cameras. Effectively, this is something completely different from our application, but the point in common is represented by the necessity to have very good starting images of the tunnel surfaces. Here, the authors used DLSR cameras easily recoverable in commerce with a 8MP resolution and obtained excellent results. Another interesting work was proposed by *Stent et al* [12] for automatic detection and classification of different changes on tunnel linings. Again, the authors proposed the usage of DLSR cameras for images capture, in particular they built a circular array of 5 DLSR in order to catch all the internal surface of the tunnel mounted on a rail and remotely controlled. Clearly, this is a very expensive structure and requires also a huge amount of preparation for being built.

The fist application of CNN in the problem of crack detection was due to *Makantakis et al* in 2015 [13], who proposed a vision based method using deep neural architecture for classification of images with or without a cracks. The proposed architecture was a 2 convolutional layers CNN, with a final softmax layer for the classification in crack or no crack detection. Here, the authors took more than 100000 images and used 80% of them for the training set. The objective of this project was to compare the Deep Learning approach with other computer vision algorithm and, by looking to the results, the authors asserted that this technique is widely better than all the most used image-processing approach. One of the best results in this new branch was reached by *Cha and Choi* in 2017, who worked in a project basically equal to the previous one. The images for the training of the network had been taken with a DLSR camera inside a tunnel in Manitoba and more than 40000 images were collected. In particular 80% of them are used as training set, while 10% each for testing and validation. The architecture of the CNN used is presented in figure 3.1, where C# stays for Convolutional layer, P# for pooling and BN for Batch Normalization. The training was performed with learning rate equal to 0.1 and logarithmic update during the epochs and with dropout rate equal to 0.5. The results obtained by running the algorithm on two GPUs were excellent, something around 97% accuracy in crack detection in the testing phase, but the authors did not specify exactly how they captured the images and how they built the model hardware. The most fascinating work present in literature regard-

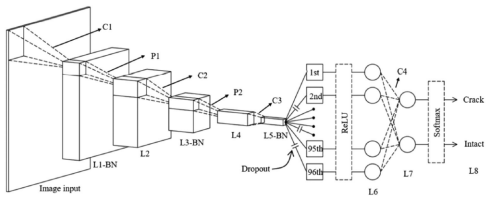


Figure 3.1: *Cha and Choi CNN architecture*

ing the tunnel inspection is the ROBO-SPECT European FP7 Project [14], whose objective is building a complete autonomous robot able to collect images without humans help and then process them inside a network to identify the presence of a crack. The robot is very complex itself and it is composed by a standard mobile vehicle with an extended crane with a robotic harm. On this robot is mounted the computer vision system, composed by two CCD cameras and two DLSRs and a powerful light system (not well specified). The robot has also a 3D laser scanner able to detect also the depth of the crack: when the images are passed to the CNN and a crack is detected, than a signal is passed to the robotic arm in order to be moved near the crack and activate the 3D laser. The architecture of the CNN is not described by the authors. This is surely a fascinating project, but there are several doubts about it: first of all, the capture system is very slow and the power system for the machine could be a problem. In general, with a system like this could be difficult inspecting a long tunnel.

3.2 The model

By looking to results and infrastructures used literature, this thesis aims to create a new system able to exploit the deep learning techniques for the crack detection, but at the same time to use a low cost equipment. Moreover, the system should find its main application in the urban tunnels near a city, so it is necessary to build it in such a way that it do not obstacle the normal traffic flow inside the tunnel.
The proposed model is composed by two main component: the **acquisition system** and the **deep neural network**. This last one will be discussed in the next chapter.

3.2.1 Acquisition System

The acquisition system of the model is the structure able to collect the images which has to be classified by the deep neural net.
The idea behind this acquisition system is to use low cost equipment, but at the same time it must be easy reusable and must not require a complex infrastructure for being used. Next to these main aspects, should be remembered what is the first aim of the system: creating a dataset for the network training. Hence, the pictures must be as good as possible in terms of resolution and, overall, an high number of images is required in order to obtain acceptable results. The first prototype of this acquisition system was basically characterized by a very cheap video caption, from which (thanks to a post-processing phase) a series of pictures have been extracted. Now, the acquisition procedure has completely changed, and it is based on a direct pictures caption through a DLSR camera. We chose it thanks to the good results that this kind of cameras produced in several works present in literature. The other problem is clearly the tunnel inspection method, because the expected result is a

system which has to work inside urban tunnels without disturbing in any manner the traffic flow. So that, the structure has been realized in order to be accommodated easily on a truck; in this way we are completely able to control the system inside the vehicle. The system is composed by four main components: DLSR camera, lighting system, Radio-frequency remote control and Camera Control.

1. **DLSR Camera:** *CANON EOS800D* model, with a 55mm lens. This model satisfies all the quality constraints required by our application. The 24MP resolution guarantees extremely good pictures, which can be taken at very high speed (till nominal 6fps) and in different mode. Moreover, the manual mode of the camera allows the user to set all the parameters of the camera manually, in order to adapt the camera to the difficult environmental conditions of tunnels. Another important feature of the camera is the capability of being controlled remotely, through its PC cable, by an external CPU, which is crucial for the system in order to have a camera able to shoot without the human interference.

Figure 3.2: *Canon EOS800D*

2. **Lighting System:** Inside a tunnel, we have always poor light conditions and so the camera must be helped by external light source in order to reproduce clear pictures. Two light sources have been selected, one fixed light source and a camera-flash. The fixed source is a 500-LED Panel (*Neewer 500LED Photo Studio Lighting Panel*), useful to create a strong and clear light also in a dark environment. Synchronously, we used also the *Neewer NW-561 LCD Display Speedlite Flash* to generate a strong light with the camera in order to improve the contrast on the taken picture.

Figure 3.3: *Neewer Flash (left) and Neweer 500 Led Panel (right)*

3. **Remote Control:** we setted the Radio frequent remote control in order to create a wireless connection between the camera and the flash mentioned above. In addition, it is essential to improve the scalability of the acquisition process: in a hypothetical expansion of the system with more cameras and flashes, they can be synchronized through this kind of control. In commerce, a reasonable solution in terms of costs and quality is offered by *PocketWizards*, which proposes an high number of instruments for the instruments synchronization. A couple of *PLUSIII* have been chosen in our system. One of them is connected to the camera through its own hotshoe, while the other is connected to the flash through a provided cable.

Figure 3.4: *PocketWizards PLUSIII*

4. **Camera Control:** the Camera Control is the system which has to activate the shooting functions of the camera. For this purpose, a *Raspberry PI3* has

been selected, basically for the reduced dimensions which allow us to easily fix it on the overall system and to connect it to the camera through a PC cable. The Raspberry is controlled through an external keyboard and a 7" LCD display. However, the main reason behind the choice of a Raspberry, is the capability to exploit through it the power of the Python library **gphoto2**. A simple Python code has been developed in order to activate the shutter release of the camera at its maximum speed: it is like having the shutter constantly hold down (the code is shown in Appendix A). Thanks to this technique, the camera produces pictures at the maximum speed allowed and the execution of the program can be easily stopped when the truck rides out of the tunnel.

Figure 3.5: *Raspberry PI3*

The overall system in presented in figure 3.6 and it respects all the constraints previously explained, whit a total cost below **2000$**. Everything is assembled on the truck through a home-made support, which is composed by an alluminium based structure that can be turned in order to catch images from different angle. The components have been attached to the structure through a wood panel.

Figure 3.6: *The system (up) mounted on the truck (down)*

Chapter 4

Inception

4.1 Introduction

Inception is the name of a neural network module introduced for the first time by the Google team *GoogLE Net* in the ILSVRC 2014 competition for computer vision [15]. Here, they proposed this new module in its first release (*Inception-v1*) inside a huge neural network called **GoogLE NET** and, thanks to the extraordinary results, they continued the working in the designing phase, producing 3 new versions in the successive years (*Inception-v2, Inception-v3, Inception-v4*).
The structure of Inception-v1 and GoogLE NET is well explained in [15], where the GoogleNet team members themselves describe deeply their ideas and their final implementation. In particular, they created the models by starting from the work of *Lin et al.* [16], which provides the idea of *Network in Network*, frequently used inside Inception. So, before starting talking about it, it is necessary to explain the concept of "Network in Network".

4.1.1 Network in Network

A generic CNN works practically as a Generalized Linear Model for the input Data Set, because effectively it applies a linear product followed by a non linear activation function, as well as a GLM does if we consider a non linear link function. One of the most important thing on GLM is that the components of the latent variables (what in CNN becomes the parameters of the network) are linearly separable; but this is not always true, specially in the Neural Networks, where the parameters are often correlated through non linear dependencies. So, Network In Network (NIN) module try to substitute the GLM structure of a convolutional layer with a *micro-network structure* able to approximate non linear functions. In particular, in the architecture proposed by Lin et Al., the chosen structure is a multilayer perceptron, that is universally recognized as a good non linear approximator and can be easily trained through the backpropagation algorithm.

This new module takes the name of MLPCONV layer.

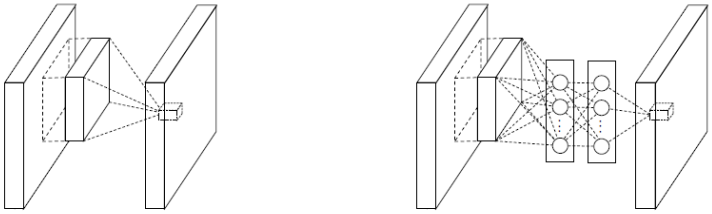


Figure 4.1: *Comparison between a standard convolutional layer (left) and a MLP-CONV layer (right)*

Moreover, they introduced another big novelty inside their network: instead of using the classical fully connected layer with dropout regularization after the last convolutional layer, they pass directly the information of the last MLPCONV to the softmax layer using a *global average pooling layer*. So, these are the two main structure introduced in this network.

- **MLPCONV:** As said before, the idea is to use the multilayer parceptron as micro-network non linear approximator. The following equations show how it works:

$$f^1_{i,j,k_1} = max\{w^{1\mathrm{T}}_{k_1} x_{ij} + b_{k_1}, 0\}$$
$$\vdots$$
$$f^n_{i,j,k_1} = max\{w^{n\mathrm{T}}_{k_n} x_{ij} + b_{k_n}, 0\}$$

  where:

$$\begin{cases} (i,j) = output\ pixel\ index \\ x_{ij} = input\ pixel\ square \\ k : feature\ map\ index \\ n : number\ of\ the\ layer\ in\ the\ perceptron \end{cases}$$

  The activation function f is always the rectifier linear unit.

- **Global Averare Pooling:** This layer is introduced to reduce the overfitting and to free the network from the dependency to the dropout regularization. The idea is to use one feature map for each class in the last MLPCONV and to feed the average of this feature directly inside the softmax layer.

The equations describing the MLPCONV layer are also very helpful in the view of the *Cross Channel Pooling Layer*. A cross channel pooling layer is a particular type of layer which operates in the filters space and modifies the structure of the features maps coming out from a convolutional layer. In few words is like computing

a convolution transversal with respect to the normal flow of the network. So, this NIN structure works as a cascade of cross channel pooling layers, each one with a rectifier linear unit at the end. This working principle is practically equal to the one computed by a convolution with 1x1 filters: it does not modify the significant pixels sides, but can modify the amount of features maps at the output of the layer. And this is exactly what GoogLE NET uses in order to implement the Inception modules.

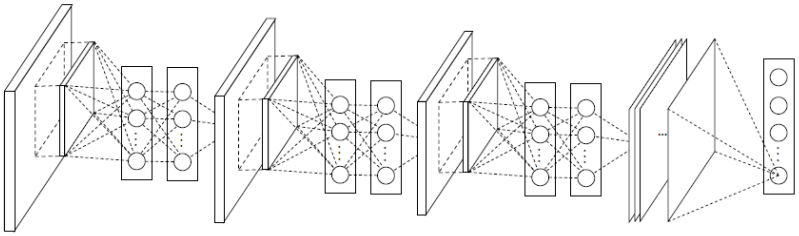


Figure 4.2: *Complete structure of a network with the NIN philosophy*

4.2 Inception-v1

"How can be increased the quality of a neural network?" Everything starts from this question. And the only admitted answer is to increase the size and the depth of the network. But this solution brings two big problems:

1. Large and deep networks use a huge number of parameters, so it is necessary a big Dataset, otherwise there will be surely the overfitting problem

2. Large and deep networks require a lot of computational effort.

Arora et Al. [17] demonstrate that the only way to solve both the problems is to use *sparse convolutional neural networks*, networks in which not all the neurons are used simultaneously. Even if they impose very strong mathematical conditions for the applications of their results, the empirical Hebbian principle *"Neurons that fire together, die together"* suggests that the sparse neural networks can be also used in the convolutional field. In particular, GoogLE NET applies the sparsity to the filter level: during the convolutional layers, not all the filters act with the same characteristics and with the same behavior. Moreover, Arora et Al. suggests an interesting way to construct properly an efficient network, in particular by building it layer by layer. The idea is to look at the output of the last layer and try to figure it out what are the outputs with high statistical correlations. The idea of the successive layer is to collect this output in clusters of correlations: in this way, the new output will be strongly dependent by the previous ones and will give to the

successive layer a clear overview on how the output are correlated. In the computer vision environment, this result can be translated in this way. In a convolutional layer we need to create filters which has the objective to extract from the input particular pixels with high statistical correlation, which means that can be seen as a particular region inside the image. In practice, we can create convolutions with the aim of finding simple patterns in the images, probably located in very strict local region of the images; so, we can use $1x1$ convolutions followed by rectifier linear unit. This 1x1 convolutions become fundamental, because risize the filter space and operate a normal convolutional layer thanks to the rectifer linear units. Than, $3x3$ and $5x5$ convolutions can exploit patterns spread in a larger local region. But all these structures can be found at any depth of the network, so this type of filters must be present from the beginning to the end of the network. Probably, it is better to use more $1x1$ convolutions in the lower layer in order to find immediately the easier features and then to increase the number of $3x3$ and $5x5$ filters by going up with the depth of the network, in order to extract much more complex features. (The idea of using filters with muximum size of $5x5$ is not in Inception-v1 something done for a particular mathematical reason, but only for comfort). So, this is how Inception works: a convolutional layer is the agglomerate of different convolutions with different filter size. Moreover, a max pooling is always inserted inside this structure, because it helps to produce outputs that stand out the pixels with higher priority.

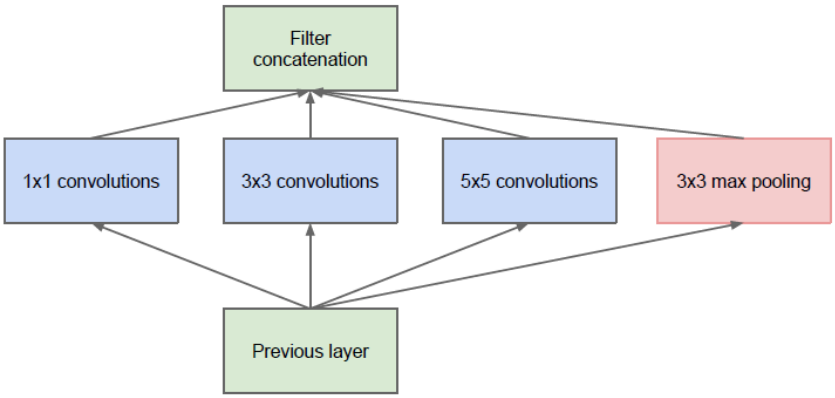


Figure 4.3: *Inception-v1 naive module*

But this model has some problems, first of all the high computational effort required by the big $5x5$ convolutions when the number of filters start increasing. But the solution can not be deleting them, because they are crucial in finding complex pattern in the images. So, the most immediate solution is to use again the 1x1 convolutions, this time only for resizing the filters space and reducing the computational effort for the higher convolutions.

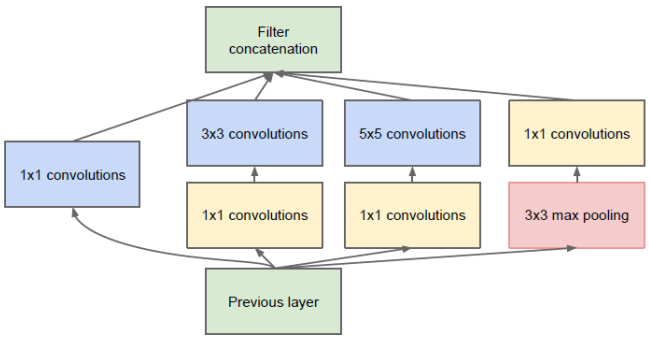


Figure 4.4: *Inception-v1 final module*

In summary, an Inception based network is composed by a series of this layers with some intermediate pooling layers that sporadically compact the size of the input for the successive layers. With this kind of structure we have two main advantages:

1. The computational effort is strongly reduced, thanks to the filter size reduction before high convolutions. In this way, we can perform deep analysis without loosing too much computational time.

2. The structured layer with different filters permits to analyze an image in different scales and, aggregating all the results gives to the next layer a complete view on how the image is composed.

As said before, the first network created with these modules is GoogLE NET

4.2.1 GoggLE NET

This is the schematic table of GoogLE NET

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Figure 4.5: *GoogLE NET compositition. "AxA reduce" stays for the number of filters 1x1 used for the reduction size before the AxA convolutions*

The network is 27 layers deep and there are more than 100 computational blocks. Some notable characteristics:

- The layers near to the input image are normal convolutions, because the experience suggested that they are better in the recognition of very simple patterns in the preliminary stages.

- The passage from convolutions to fully connected layers is mediated by an average pooling layer, which we know are more efficient than a simple intermediate fully connected layer.
- Fully connected layers are in the classical structures of a CNN for classification purposes: a layer with dropout regularization (40%) followed by a softmax layer. The final classification is on 1000 items because the net was firstly used for recognition of 1000 leaf-node categories inside the Imagenet Datatset hierarchy.

As we can see in the graph 4.6, there are 2 auxiliary networks which end up with a classification in the middle of the net. Why are they present? As we know, neural networks use the backpropagation algorithm for finding the value of the gradient during the training phase, in order to modify properly the parameters step by step and we also know how the depth of the structure can bring to the *vanishing gradient* problem, which brings to malfunctioning and bed results. So, Google Net team thinks to process classifications also in the middle, in order to not lose any information if the vanishing gradient arises. The final classification will be a linear combination oh the 3 performed, clearly giving higher priority to the final one. The structure of these auxiliary layers is the following:

- 1 Average Pooling Layer with $5x5$ filter size and #3 stride
- 128 filters $1x1$ convolutions for filter space reduction and rectifier linear unit activation
- Fully connected layer with 1024 units
- Dropout (*40%*) layer
- Softmax layer with 1000 classification units, the same as the main final classifier

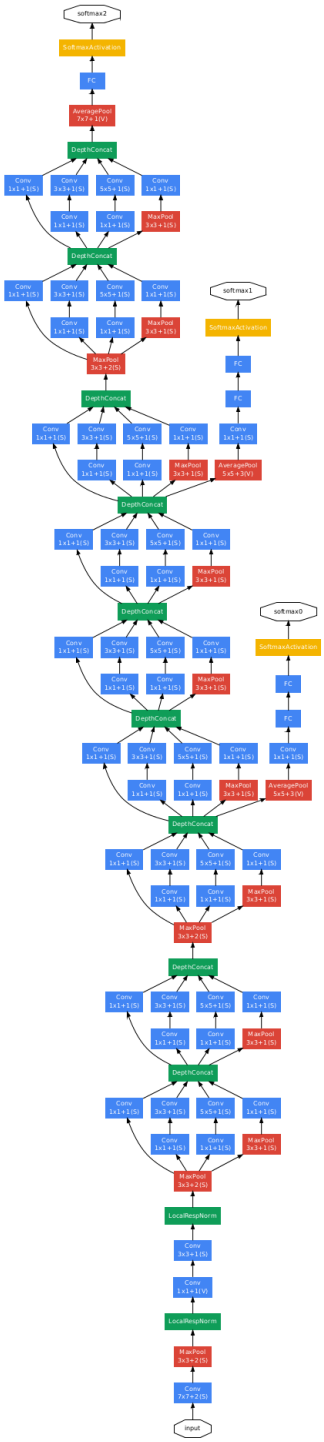


Figure 4.6: *Complete graph of GoogLE NET*

4.3 Successive versions

As explained before, GoogLE NET is a very powerful neural network with a lot of positive features with respect to networks previously built. But its complexity makes difficult operating changes of whatever type in the structure. Moreover, it seemed that the network was very efficient only in the particular case of the competiotion for which it was built. For example, by double up the filters space inside an Inception module, the complexity and the number of parameters incresed of $x4$ factor. So, *Szegedy et al.* in 2015 tries to increase the performances of the net by looking mostly on the improvement of the Inception modules and exploiting techniques and strategies that could be extended to all the deep neural networks. [18] As suggested already by [15], the filter size reduction is a fundamental key for reducing the complexity and the computational costs of the network. For example, a $5x5$ convolution with n filters is 2.78 times more expensive than a $3x3$ with the same number of filters. But, as previously said, larger convolutions are required in Inception-v1 for catching complex features in the images. An idea to overcome this problem, is to use again the concept of resizing a large convolution with a cascade of smaller convolution. So, a $5x5$ can be seen as a sequence of two $3x3$, with the second one operating as a fully connected layer.

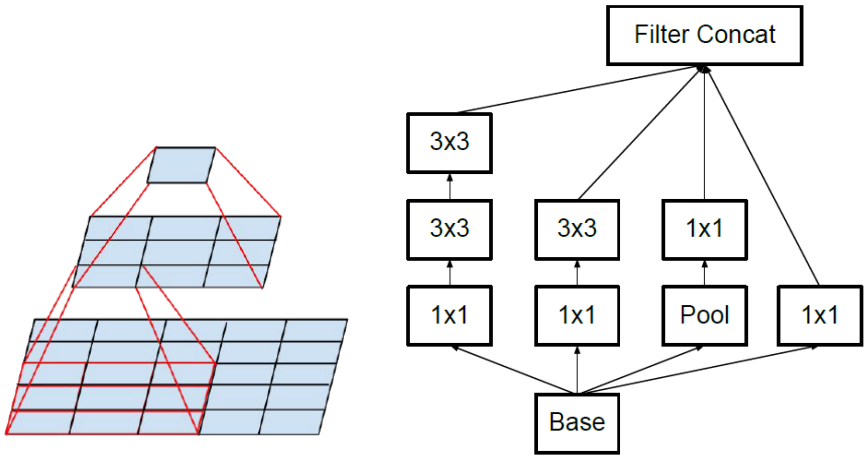


Figure 4.7: *On the left, the new structure replacing a 5x5 convolution. On the right, a type of Inception module with this new feature.*

These results show how applying convolutions greater than $3x3$ is not so efficient. So, it is possible to reduce again the size (i.e. $2x2$) in order to obtain better results? What it can be demonstrated is that a $3x3$ convolution can be better replaced by a sequence of a $3x1$ and a $1x3$, reducing by 33% the complexity of the system. Moreover, this technique seems to be efficient for any convolution size, even if the better results have been achieved only in the middle range of depth of the network, so when the dimension of the feature maps is already strongly reduced.

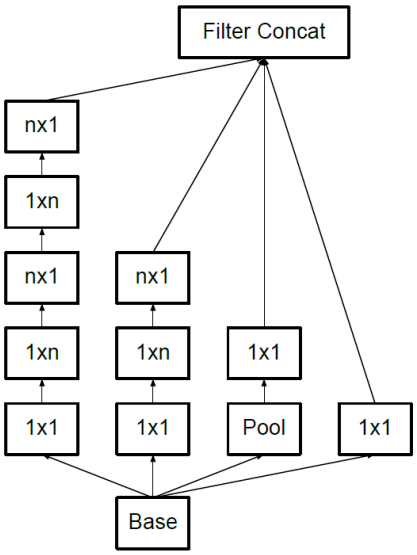


Figure 4.8: *Another Inception module with the 3x3 convolution factorization.*

The last evolution of the Incepetion modules was born from the idea of promoting high dimensional representation in order to exploit complicated features in the higher layers. This idea brings to the last module in figure 4.9

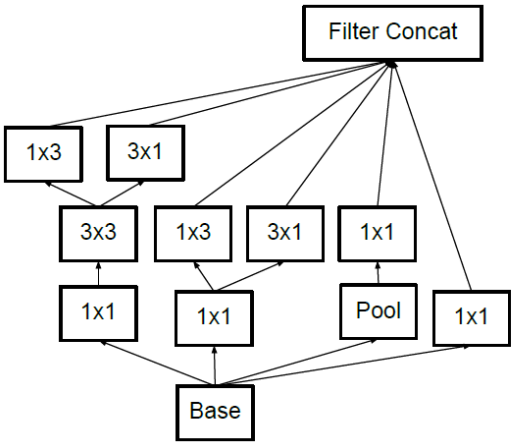


Figure 4.9: *Inception module for high dimensional representation.*

Another interesting argument inside GoogLE NET regards the auxiliary network. By looking the evolution of the train of a deep net with and without the auxiliary classifiers, the differences arise only in the final phase of the procedure,

when the net with auxiliary networks starts to obtain better accuracy. So, the role of this external branches must be redefined: they can not be considered as an help in the convergence of the gradients in the lower layers, but they must be seen as *regulizer*. In fact, the improvements became notable when these layers are batch-normalized or if they have dropout regularization inside.

At the end, another critical point is the feature map size reduction. In the Inception-v1 modules it is achieved by using $1x1$ convolutions followed by a pooling layer, which produces very good results, but increases the complexity of the system due to the $1x1$ convolutions. A structure like 4.11, instead, has shown a strongly reduction in computational costs, by producing the same results.

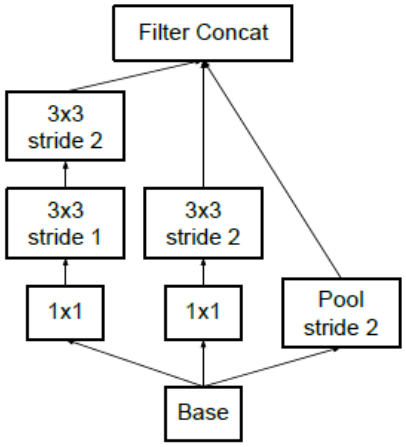


Figure 4.10: *Another Inception module for resizing the feature map size while increasing the filter space.*

4.3.1 Inception-v2 and Inception-v3

Inception-v2 is the evolution of GoogLE NET obtained by the connection of all the deductions previously explained.

TYPE	PATCH SIZE / STRIDE	INPUT SIZE
conv	$3x3/2$	$299x299x3$
conv	$3x3/1$	$149x149x32$
conv padded	$3x3/1$	$147x147x32$
pool	$3x3/2$	$147x147x64$
conv	$3x3/1$	$73x73x64$
conv	$3x3/2$	$71x71x80$
conv	$3x3/1$	$35x35x192$
3x Inception	Figure 4.7	$35x35x288$
5x Inception	Figure 4.8	$17x17x768$
2x Inception	Figure 4.9	$8x8x1280$
pool	$8x8$	$8x8x2048$
linear	logits	$1x1x2048$
softmax	classifier	$1x1x1000$

Table 4.1: Inception-v2 architecture

The traditional $7x7$ convolutions have been factorized in 3 consecutive $3x3$ convolutions. In the Inception part, everything start with 3 modules as the one in figure 4.7. Than, with a module equal to figure 4.11, the grids are resized to $17x17$ and then there is 5 modules like the one in figure 4.8. At the end, 2 other models like figure 4.9 end the Inception part. The convolutions involved in grid size reduction are marked with 0-padding and in all the Inception modules, while the other convolutions are not in 0-padding. In conclusion, all the layer uses batch-normalization.

Inception-v3 is a network totally equal to Inception-v2, with the only difference that the batch-normalization is also extended to the auxiliary networks

4.3.2 Inception-v4

Inception-v4 is the last version of the *pure* Inception modules proposed by Google [19]. No such great modifications in the idea occurred during its design, but only a straight decision to uniform all the Inception blocks and to re-elaborate a bit the structure of the part of the networks before the Inception modules. The grid reductions are still the same presented for Inception-v2 and so the location of the layers with batch-normalization technique.

TYPE	PATCH SIZE / STRIDE	OUTPUT SIZE
STEM	Figure 4.11	$35x35x384$
4x InceptionA	Figure 4.11a	$35x35x384$
Reduction	Figure 4.11	$17x17x1024$
7x InceptionB	Figure 4.12b	$17x17x1024$
Reduction	Figure 4.11	$8x8x1536$
2x InceptionC	Figure 4.12c	$8x8x1536$
Average Pool	$8x8$	$1x1x1536$
Dropout (*40%*)	Fully connected	$1x1x1536$
Softmax	Classifier	$1x1x1000$

Table 4.2: Inception-v4 architecture. The input of the Stem is a 299x299x3 image

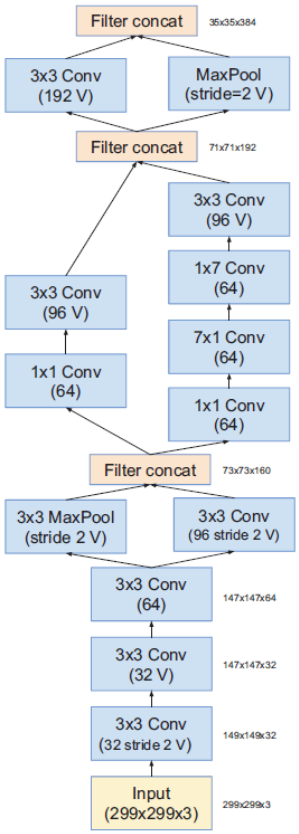
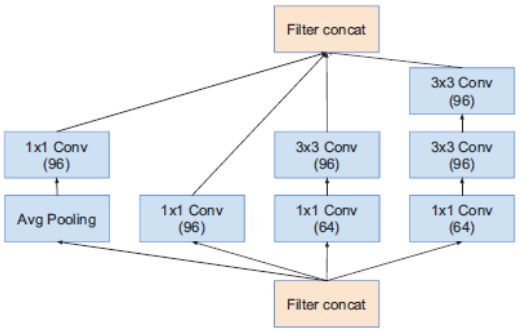
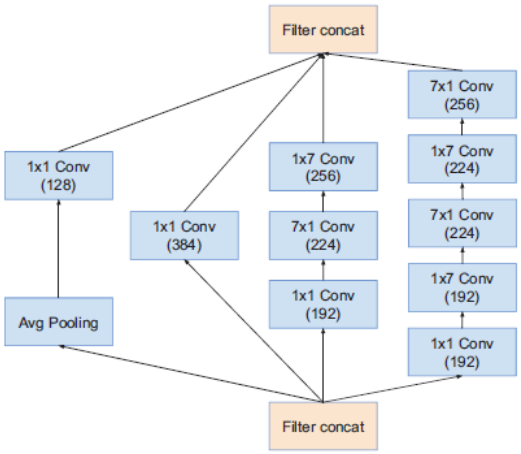


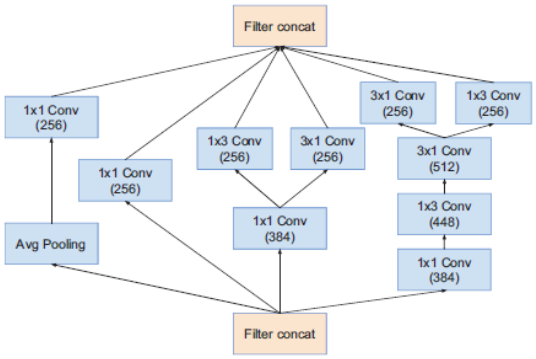
Figure 4.11: *Stem graph of Inception-v4*



(a) InceptionA



(b) InceptionB



(c) InceptionC

Figure 4.12: Inception-v4 modules

Chapter 5

Training: Transfer Learning

The idea is very simple: we want to use Inception-v4 as Deep Neural Network in our Prototype. Inception is an image classificator, which is able to classify an input image in a certain number of classes given by the user. In order to do that, we have to exploit the **Transfer Learning** technique. Transfer Learning is one of the most modern technique used in Machine Learning application, because it allows to apply knowledge previously learned in order to solve new tasks and problems [20]. It is very common in an intelligent world to work in such way and there are lots of examples which can easily explain how Transfer Learning works; for example, if we want to learn how to play the organ, than starting from the knowledge of the piano could accelerate the learning and make it much more powerful. With Deep NN the concept is exactly the same: if there are strong network previously trained for some applications, why do not use what they have already learned in our specific task? With Inception this work can be effectively done. V4 has been built in order to detect inside an image some general features (like for example shapes, colors , edges etc) with increasing difficulty moving towards the convolutional layers. In this way, we can use the results obtained by the training made by Google's huge computers and datasets and then adapt them to our specific task, by re-training only the last layer which is considered the decisional one. Inception-v4 graph is open source and can be found on the web and used by anyone through Transfer Learning. Obviously, we need to retrain the last network and in order to do that we need to create our dateset made by tunnel images that represent surfaces with cracks or without cracks and for this purpose the hardware explained in chapter 3 has been used. At this point, the training methodology used is the **Supervised Learning**: we provide a set of images already classified in two main categories, *Crack Detection* and *No Crack Detection* and the network, by confronting its results with the actual classification will modify its weights and biases through backpropagation.
In practice, the code first of all extracts the pretrained characteristics until the last layers from the given images (**Bottlenecks**) and then uses these Bottlenecks as input of the last layer. In this phase, we need to choose the ideal characteristics of

the training procedure, which will be presented in the next sections. All the code has been developed in Python 3.6.2, using the Tensorflow environment. This environment is quite simple in the syntax and it is also integrated with a useful tool, Tensorboard, which provides plots and graphs very useful for looking the progress of the learning procedure. The graph file containing Inception-v4 structure has been released by Google in late 2017 and it can be found on internet with a series of useful advises on how to use it in the proper way inside a suitable Python code.

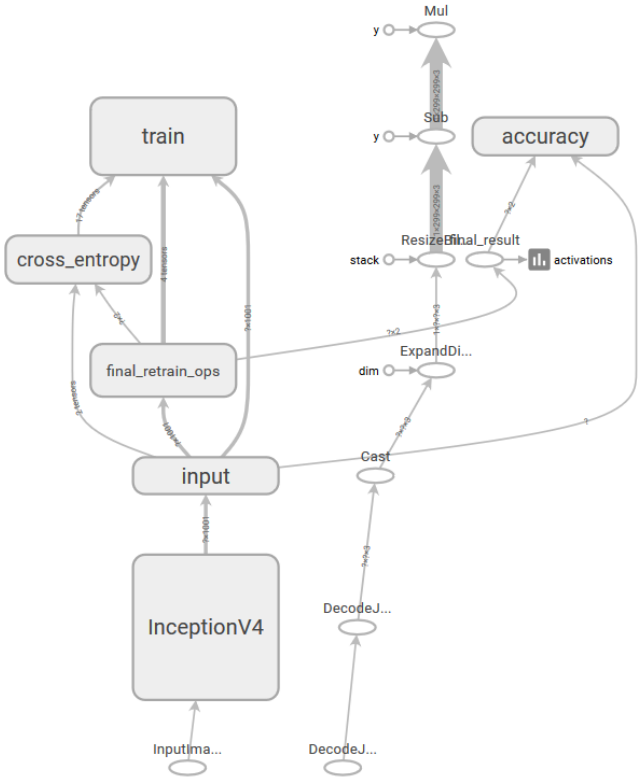


Figure 5.1: *Tensorboard graph of the retraining phase of Inception-v4*

5.1 Hyperparameters

As previously explained, the core of the retraining procedure is the choice of the characteristics of the process, which we call **Hyperparameters**. In all the theoretical part explained in the first chapter, we saw as some parameters could strongly afflict the learning performances of a deep neural network. In particular, in this work some of them have been taken as reference and several tests have been performed in order to find the best setting between them which guarantees the optimal

performance:

- Testing an Validation percentage
- Learning Rate
- Validation and Training Batches size
- Evaluation Batch

Regarding the sizes of Train, Validation and Test dataset, there is not a specific theory about and so, by looking different approaches in literature, we decide to use in all our tests the most common technique used, which is probably the one that guarantees the best results: 80% of the dataset in Training, 10% in Validation and 10% in Testing. This means that on the overall Dataset of **5120** images obtained with our prototype there are 4095 in the Training batch and 511 each in Validation and Test batches. Regarding this Dataset, an important feature must be underlined. Through our prototype we were able to collect around 3500 images inside two tunnels in Downtown Los Angeles (2^{nd} *Street and* 3^{rd} *Street Tunnel*) during a normal working day with a normal traffic flow, in order to test the working capability of the prototype in common situation. The captures went very good and no problems occurred with the traffic during our experiment. Given that, we decided to increase the number of images in the Dataset by using the Artifical Expansion of the data technique, which allowed us to increase the number of images by rotating the one we had of 90 degrees. Thanks to this, we reached an important number of images which performed good results during the retraining phase of Inception.

Figure 5.2: *Examples of images inside our Dataset, obtained through the prototype. On the left an examples with crack, on the right an exaple without crack*

5.2 Results and considerations

The training of the network requires a huge computational effort and obviously can be very time expensive. For this reason, the code should be executed on a GPU which can speed up the procedure. For this reason, we introduced in our prototype the **NVIDIA Jetson TX2 Development Kit**, a powerful GPU Linux-based system specifically built for AI applications, such that Deep Convolutional Neural Network. Preparing and flashing properly the Kit is quite complex and time expensive, but the results are impressive.

Figure 5.3: *NVIDIA Jetson TX2 Development Kit*

In fact, the network has been trained and tested both on Jetson and on a CPU and, even if the final results are less or more equal, the gain in terms of time was quite large. In the best test, the CPU system spends around 4 hours to complete the retraining, while Jetson (under the same conditions) spends only 2 hours. For this reason, the table below shows the results obtained on Jetson TX2, evaluated in terms of the value of the Cross Entropy function in the last training sample and, overall, in terms of accuracy of the final test.

Test	*Steps*	*Learning Rate*	*Train batch*	*Val Batch*	*Eval interval*	**Cross**	**Accuracy**
1	7000	0.01	200	100	100	0.0078	97.8%
2	7000	0.01	700	100	100	0.1252	95.1%
3	4000	0.01	500	100	100	0.1106	94.7%
4	1000	0.05	200	200	100	0.2313	90.0%
5	15000	0.001	200	200	100	0.0989	93.5%
6	1000	0.01	200	100	100	0.1263	93.7%
7	1500	0.02	200	200	100	0.1003	93.9%
8	10000	0.01	200	100	100	0.1111	94.9%

Table 5.1: Training results

The shape of the learning production can be seen thanks to *Tensorboard*, one of the main plug in of the Tensorflow environment. Tensorboard allows us to obtain

graphs showing the values of the main learning parameters at different steps and to see the overall schematics of the trained network. These graphs can help us to derive some considerations on the benefit of our training much more reliable than the one we can observe from the numerical data.
First of all, we can start from the best result achieved, which is the **Test 1**

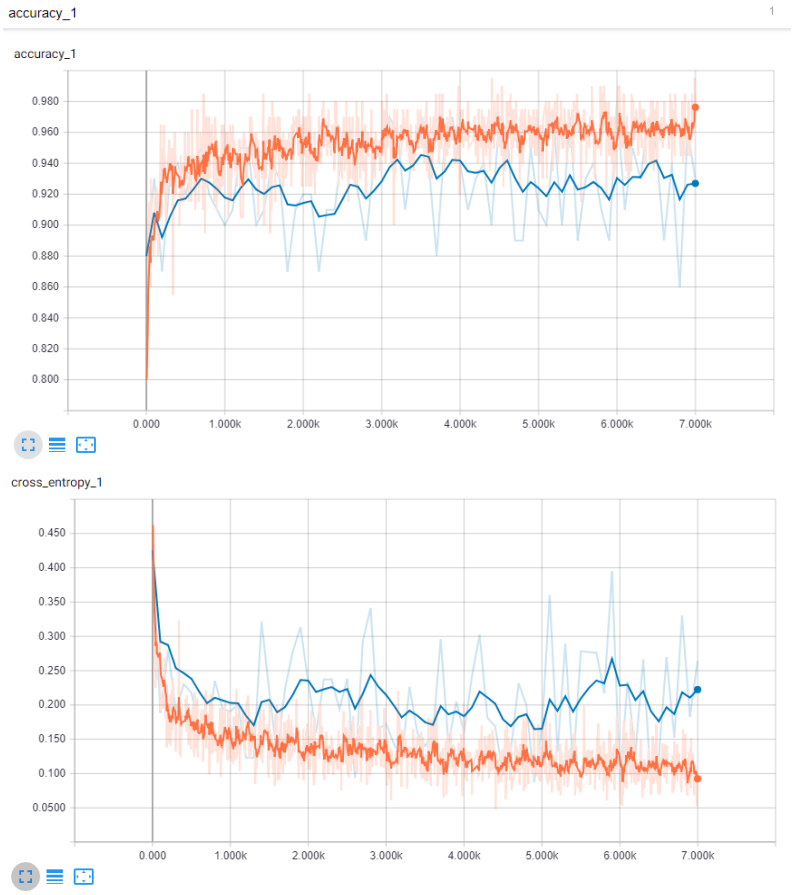
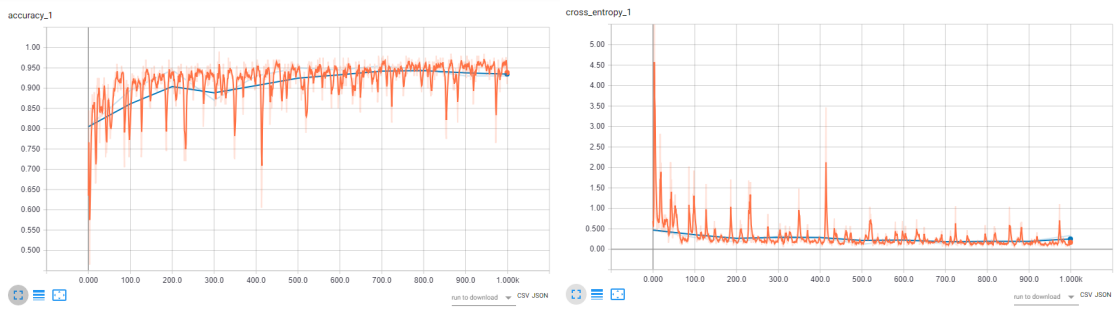


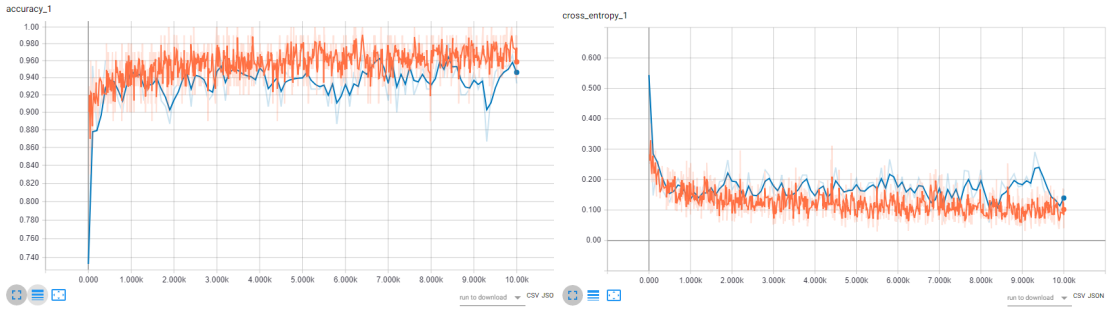
Figure 5.4: *Accuracy and Cross entropy values during the learning in Test 1. In orange the training batch, in blue the validation*

Here, we can appreciate the good results obtained with this configuration. Both in training and validation we have an increasing shape in the accuracy and a decreasing one in the cross entropy. This means that the overfitting is not so strong (but still present) and the dataset is quite good for this kind of application. Moreover, the final test accuracy is equal to 97.8%, which means that on 511 images only 11 have been misclassified. Moreover, on these 11 images, 10 corresponded to "false positive" where the network recognized the presence of a crack when no crack occurred. This means that only in one sample the network missed the recognition of the crack, which is the real and important task of the project. A false positive

is obviously a mistake in the recognition process, but it is less effective than a true positive in terms of physical meaning.
By looking other results, we can appreciate how the tuning of the hyperparameters strongly influences the learning procedure.



(a) Test 4



(b) Test 8

Figure 5.5: Accuracy and Cross Entropy values of training (orange) and validation (blue) of two different tests

In Test 4 and Test 8 the main differences in the hyperparameters with respect to the Test 1 are the Learning Rate and the number of steps respectively.
In Test 4, the worst one with only 90% in test accuracy, the Learning Rate has been increased a lot, (5 times more). As suggested by the theory, a bigger learning rate should accelerate the learning and in fact the final result has been reached in just 1000 steps. Obviously, we can observe how fast the network saturates and how clear is the presence of the overfitting.This justifies the theoretical discussion previously faced: the value of the learning rate must be properly balanced and a fast learning procedure is not always a good idea. At the same time, we can see in Table 5.1 also the counter part: Test 5 shows a situation in which a too much small learning rate does not produce a final good result 93.5%, even if it could reach better approximation as suggested by theory.
In Test 8, the learning has been chosen equal to the best setting, but the number of steps has been increased. This test helped us to exploit all the negative aspects of

overfitting. In a situation with all the other hyperparameters with the same value, an higher number of steps should guarantee a better learning. But the overfitting occurs and, instead of increasing the performance, reduces the power of the net. So, the network started learning "by heart" the images in the dataset and at the end we reached only 94.9% in test accuracy. Test 6, on the other hand, presents the same parameters of the best setting, but with a number of steps strongly decreased, only 1000 instead of 7000. The results does not change so much, because the final accuracy is much lower, only 93.7%. That demonstrates the fact that with a large number of images of high qualities, it is necessary to run the training procedure a lot of times but without going too far, otherwise the overfitting will cause the same problems in terms of final accuracy.
In general, all the tests in the table shows the technique which has to be followed during the tuning of the hyperparameters. There is not a clear theory about the correct value of the parameters to be used and so we had to try different combinations to find the right balance.

Chapter 6

Custom CNN

The huge potentiality of the Inception network provided by Google can be really understood when compared with an another network. For this reason, a quite simple CNN has been created in the Tensorflow environment, in order to exploit this consideration.

6.1 Architecture and hyperparameters

As said before, the CNN is basically very simple. It is a 5 layer deep convoutional neural neural network with the following characteristics:

- **Layer 1:** Convolutional layer with 3 filters and max pooling technique for the half time reduction of the pixels dimensions
- **Layer 2:** Convolutional layer with 32 filters and max pooling technique for the half time reduction of the pixels dimensions
- **Layer 3:** Convolutional layer with 64 filters and max pooling technique for the half time reduction of the pixels dimensions
- **Layer 4:** Fully connected layer with 1024 output neurons, ReLu activation function and Dropout regularization at 0.5
- **Layer 5:** Final Softmax decisional layer with a two-class classification

Even if the network is quite short, the training procedure in quite long, also in the Jetson TX2 environment: this is why the training has to be performed from the scratch and not only in the last layers like the Transfer Learning technique. So, some hyperparamters have been set and not changed along the different tests in order to gain as much time as possible. These choices have been done looking the results obtained with Inception and the results available in literature. For example:

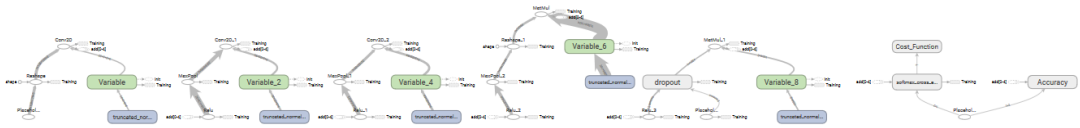


Figure 6.1: *Custom CNN shown in the Tensorboard environment*

- **Validation dataset:** 10% of the total dataset
- **Test dataset:** 10% of the total dataset
- **Train Batch size:** 200
- **Validation Batch Size:** 100
- **Evaluation Interval:** 100

In conclusion, several training tests have been performed by changing two main hyperparameters, *Learning Rate* and *Number of steps*

6.2 Results

The most relevant tests performed in the Jetson TX2 environment on the custom CNN are here exposed.

Test	*Steps*	*Learning Rate*	**Accuracy**
1	5000	0.00005	90.4%
2	5000	0.0001	92.8%
3	1000	0.0001	88.8%
4	5000	0.001	94.2%

Table 6.1: Training results

The following images show the accuracy results of the network obtained through Tensorboard.

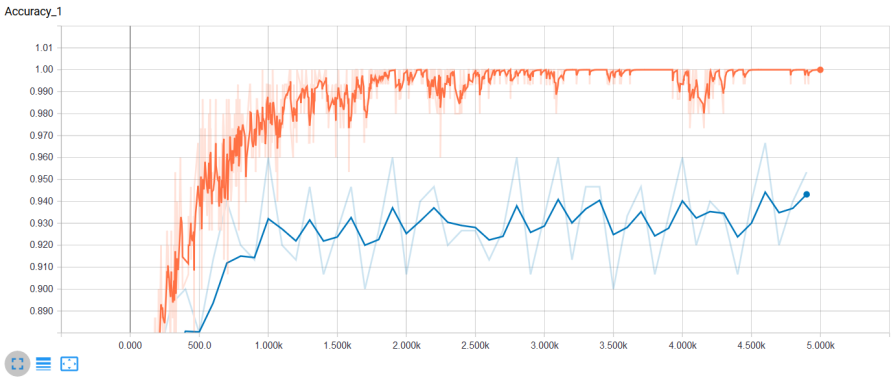


Figure 6.2: *Test 4 accuracy values showed in Tensorboard environment*

Test 4 in the custom CNN is the best one and reached 94.2% in terms of final test accuracy. This result is obviously much more lower than the best performance obtained with Inception and this graph shows us why it happened. Here, the overfitting is very strong and it is easily notable. After just 2000 steps the training accuracy has been reached 100% and the validation accuracy stops increasing. The reason behind this is obviously due to the short deepness of the network, because 5 layers going through saturation very fast and does not have the time to learn properly all the details of the presence of a crack in an image.

6.3 Code

As already explained, the CNN is developed in Python 3.6, exploiting the TensorFlow library. In the following section the code is presented, giving some explanations about the specific functions and procedures used. The entire code has been written in Object Oriented structure.

So, besides the declaration of the libraries used in the code, there is the definition of the first class. This class allows to extract the Data set used in our applications, also with the names and the classes in which they are classified.

Listing 6.1: Data extraction

```
import cv2
import os
import glob
from sklearn.utils import shuffle
import numpy as np
import tensorflow as tf
import time
from datetime import timedelta
import math
import random

class Data_loading():
    #Class in which the images are collected and prepared to be
       used
    def __init__(self):
        pass
    def load_images(self,user_path,size,classes):
        """
        This method will load the images from the corresponding
           path in the directory.
        NB: The images must be correctly prepared, with the
           folder for each class containing images belonging to
           that particular class

        """
        images=[]
        labels=[] #This vector contains the corresponding label
           or each image in the vector images
        names=[]
        cls=[] #This vector will contain all the classes in
           which we can classify the image
        print('Loading images..')
        for f in classes:
            ind=classes.index(f) #looking for the labels in
               which the images must be classified
            path=os.path.join(user_path,f,'*g') #identifying tha
                path given by the user
```

```
            files=glob.glob(path) #searching for the images in
                the corresponding path
            for i in files:
                image=cv2.imread(i)
                image=cv2.resize(image,(size,size),0,0,cv2.
                    INTER_LINEAR) #Extracting the image and
                    resizing it depending on the value given by
                    the user
                image=image.astype(np.float32)
                image=np.multiply(image,1.0/255.0) #Coverting
                    each image in a number
                images.append(image) #the images became an array
                label=np.zeros(len(classes))
                label[ind]=1.0
                labels.append(label) #Also the labels became an
                    array
                ibase=os.path.basename(i)
                names.append(ibase)
                cls.append(f)
        images=np.array(images)
        labels=np.array(labels)
        names=np.array(names)
        cls=np.array(cls)  #All the vectors are transformed into
            numpy vectors
        return images,labels,names,cls
```

The next class introduces the creation of a new batch for the learning procedures. The same function for the validation set has been created.

Listing 6.2: Batches

```
class Batch_preparation():
     #Class for preparing the dataset of the new batch to be
         considered
    def __init__(self):
        #definition of the constant variables inside the class
        self.epochs_completed_train=0
        self.epochs_index_train=0
        self.epochs_completed_val=0
        self.epochs_index_val=0

    def  next_batch_train(self,images,labels,names,
        train_batch_size):
        """
        This is a crucial method for the entire CNN.
        It creates a batch (starting from the size given by the
            user) by randomly choosing some images from the data
            set previously obtained
        """
```

```
        # In order to consider every input image, the starting
            image in the last one we have previously consider.
        # If the new batch is bigger than the total number pof
            the images, it means that we have completed an epoch.
        #So, the starting point is reset to 0 and the epoch
            completed index is updated
        s=self.epochs_index_train
        total_images=images.shape[0]
        self.epochs_index_train += batch_size

        if self.epochs_index_train > total_images:
            self.epochs_completed_train += 1
            s=0
            self.epochs_index_train=batch_size
            assert batch_size <= total_images
        end = self.epochs_index_train
        epoch=self.epochs_completed_train

        return images[s:end],labels[s:end],names[s:end],epoch#
            The new batch is ready

    def  next_batch_val(self,images,labels,names,
        validation_batch_size):
        """
        This is a crucial method for the entire CNN.
        It creates a batch (starting from the size given by the
            user) by randomly choosing some images from the data
            set previously obtained
        """
        # In order to consider every input image, the starting
            image in the last one we have previously consider.
        # If the new batch is bigger than the total number pof
            the images, it means that we have completed an epoch.
        # So, the starting point is reset to 0 and the epoch
            completed index is updated
        s=self.epochs_index_val
        total_images=images.shape[0]
        self.epochs_index_val += batch_size

        if self.epochs_index_val > total_images:
            self.epochs_completed_val += 1
            s=0
            self.epochs_index_val=batch_size
            assert batch_size <= total_images
        end = self.epochs_index_val
        epoch=self.epochs_completed_val

        return images[s:end],labels[s:end],names[s:end],epoch#
            The new batch is ready
```

After this, there is the class which divides the dataset in training, validation

and test data. This group of data will be passed to the *Batch_preparation* class for creating the batches used in the learning algorithm.

Listing 6.3: Data Preparation

```
class Data_set_preparation():
    #Subdivision of the set in Training,Validation and Test
    def __init__(self):
        pass
    def preparation(self,path,image_size,classes,val_size,
       test_size):
        """
        Classification of the images in Training, Validation and
            Test
        """
        L=Data_loading()
        images,labels,names,cls=L.load_images(path,image_size,
           classes) #Extracting the images
        images,labels,names,cls=shuffle(images,labels,names,cls)
            #Every time the program is run, the images change
           the position

        validation_size=int(val_size*images.shape[0])
        t_size=int(test_size*images.shape[0])

        # Definition of the parameters to be passed to
           Batch_preparation for creating data:

        #Validation
        validation_images=images[:validation_size]
        validation_labels=labels[:validation_size]
        validation_names=names[:validation_size]
        validation_cls=cls[:validation_size]

        #Test
        test_images=images[validation_size:validation_size+
           t_size]
        test_labels=labels[validation_size:validation_size+
           t_size]
        test_names=names[validation_size:validation_size+t_size]
        test_cls=cls[validation_size:validation_size+t_size]

        #Train
        train_images=images[validation_size+t_size:]
        train_labels=labels[validation_size+t_size:]
        train_names=names[validation_size+t_size:]
        train_cls=cls[validation_size+t_size:]
```

```
        return train_images,train_labels,train_names,train_cls,
            validation_images,validation_labels,validation_names,
            validation_cls,test_images,test_labels,test_names,
            test_cls
```

The *Functions* class collects all the instruments required for creating the layers in our CNN. So, it exploits the commands on the Tensorflow environment to generate all the convolutional layers required.

Listing 6.4: Network Functions

```
class Functions():
    """
    This class contains all the functions to build different
        layers that can be used for creating a CNN

    """

    def __init__(self):
        pass

    # Initial condition for weights
    def weights(self,shape):
        initial=tf.truncated_normal(shape=shape,stddev=0.1)
        # We create a series of weights in a normal distribution
            with standard deviation equal to 0.1.
        #This is a common structure that brings good results
        return tf.Variable(initial)

    # Initial condition for biases
    def biases(self,shape):
        initial=tf.constant(0.1, shape=shape)  # For the biases
            is not necessary a normal distribution, because a
            bias is the same for the entire neurons in the layer
        return tf.Variable(initial)

    # Convolutional Layer
    def convolutional(self, x,w_shape,b_shape):# Input and
        shapes required
        W = self.weights(w_shape)
        y = tf.nn.conv2d(x,W,[1,1,1,1],padding='SAME')
        # For the convolutional layer we choose a complete
            convolution (the filter strides pixel by pixel)
        # and the zero padding technique to obtain an output
            with the same dimensions of the input
        b=self.biases(b_shape)

        return tf.nn.relu(y+b) # In each convolutional operation
            , we add also the bias
```

```
    # Max Pooling layer: crucial for reducing the dimension
       after a convolutional layer
    def max_pooling(self,x):
        return tf.nn.max_pool(x,[1,2,2,1],[1,2,2,1],padding='
           SAME')  # This kind of pooling produces on the putput
            emages with height and length half reduced.

    # Fully connected layer
    def fully_connected(self,x,size):
        size2=int(x.get_shape()[1])
        W=self.weights([size2,size[0]])
        b=self.biases(size)
        return tf.matmul(x,W)+b    # Classical fully connected
           layer. The softmax unit will be added
```

Finally, we can introduce the *main* First of all, the basic declarations of the variables required in the code development, such as the **hyperparameters** or the size of each layer of the network.

Listing 6.5: Main

```
if __name__=='__main__':
    D=Data_set_preparation()
    B=Batch_preparation()

    log_path='Logs/CNN_1' # This is the path where I can save
       the summaries for visualization in TensorBoard
    # HYPERPARAMETERS
    batch_size=150
    classes=['crack','no_crack']
    num_classes=len(classes)
    val_size=0.1  #The sizes must be done in parts of 1. The
       code will translate it into percentage and finite number
       wrt the total number of images
    test_size=0.1
    img_size=64
    num_channels=3 #RGB images, so 3 channels
    filter1_size=[3,3,3]
    num_filters1=[32]
    filter2_size=[3,3]+num_filters1
    num_filters2=[32]
    filter3_size=[3,3]+num_filters2
    num_filters3=[64]
    fully_size=[1024]
    learning_rate = 0.00005
    steps=5000
    val_checking_step=100
    main_path='Dataset'
```

Then, the construction of the three-layer deep convolutional neural network, preceded by the preparation of the different data set.

Listing 6.6: Main

```
train_i,train_l,train_n,train_cls,val_i,val_l,val_n,val_cls,
    test_i,test_l,test_n,test_cls=D.preparation(main_path,
    img_size,classes,val_size,test_size)
#Preparation of the sets
#For confirmation, we can print the obtained values of the
    preparation
t=len(train_l)
v=len(val_l)
tt=len(test_l)
print('Number of images in the Training set:             {}'
    .format(t))
print('Number of images in the Validation set:           {}'
    .format(v))
print('Number of images in the Test set:                 {}'
    .format(tt))
print(len(train_i))
# Creation of the placeholders for tensorflow
input_shape=[img_size,img_size,num_channels]
x=tf.placeholder(tf.float32,shape=[None]+input_shape)
y=tf.placeholder(tf.float32,shape=[None,num_classes])
neurons_dropout=tf.placeholder(tf.float32)

# Now, we can define the various step of the layer
F=Functions()
x_in=tf.reshape(x,[-1,img_size,img_size,3]) # Necessary
    reshaping for having a number of pixels surely divisible
    for 2

# 1) CONVOLUTIONAL LAYER WITH 3x3 32 FILTERS AND MAX POOLING
    (Input Dimension: img_size x img_size x 1)
conv1=F.convolutional(x_in,filter1_size+num_filters1,
    num_filters1)
out1=F.max_pooling(conv1)

# 2) CONVOLUTIONAL LAYER WITH 3x3 32 FILTERS AND MAX POOLING
    (Input Dimension: img_size/2 x img_size/2 x 32)
conv2=F.convolutional(out1,filter2_size+num_filters2,
    num_filters2)
out2=F.max_pooling(conv2)

# 3) CONVOLUTIONAL LAYER WITH 64 3x3 FILTERS AND MAX POOLING
    (Input Dimension: img_size/4 x img_size)
conv3=F.convolutional(out2,filter3_size+num_filters3,
    num_filters3)
out3=F.max_pooling(conv3)
```

```
    # 4) DROPOUT FULLY CONNECTED LAYER
    shape_size=int((img_size/8)*(img_size/8)*int(num_filters3
        [0]))
    out3_flat=tf.reshape(out3,[-1,shape_size]) # First, flat the
         result of the convolutional layers
    full1=tf.nn.relu(F.fully_connected(out3_flat,fully_size)) #
        ReLu activation function
    drop1=tf.nn.dropout(full1,keep_prob=neurons_dropout)
    # Dropout regularization requires a percentage of neurons to
         be actived in each step.
    # It is fournished by the user, so it can be stored inside a
         placeholder

    # 5) SOFTMAX FINAL FULLY CONNECTED LAYER
    y_net=F.fully_connected(drop1,[num_classes])
    cross_entropy=tf.nn.softmax_cross_entropy_with_logits(labels
        =y,logits=y_net)
    # Taking into account that my cost function will take in
        consideration the cross entropy,
    # I need to use the softmax command with the logits, in
        order to reduce numerical problem and avoiding bugs
    print ('Network ready')
```

At the end, it is time to use the Tensorflow functions for preparing the environment in which the learning procedure will be executed.

Listing 6.7: Main

```
    sess=tf.InteractiveSession()
    # Evaluations and training functions
    # It is convenient to collect all the operations inside a
        scope section, in order to bge easily visible inside
        TensorBoard
    with tf.name_scope('Cost_Function'):
        cost_function=tf.reduce_mean(cross_entropy)  #Cost
            function
    with tf.name_scope('Training'):
        train_step=tf.train.AdamOptimizer(learning_rate).
            minimize(cost_function)
    with tf.name_scope('Accuracy'):
        prediction=tf.equal(tf.argmax(y_net,1),tf.argmax(y,1))
        accuracy=tf.reduce_mean(tf.cast(prediction,tf.float32))
            # I reduce the mean between my prediction and the
            real value, by showing the accuracy of the model
    print('Scopes created')

    # Now, I can define all the variables which I want to
        monitore inside TensorBoard
    tf.summary.scalar('Cost',cost_function)
```

```
    tf.summary.scalar('Accuracy',accuracy)
    merged_variables=tf.summary.merge_all()
    # Merging operation collects inside a unique variable all
        the summaries operation previously performed
    # and this is important to launch the TensorBoard operations
         only once
    print('Summaries created')

    # Now, we can launch the Tensorflow environment

    print('Starting training')
    sess.run(tf.global_variables_initializer()) #Inizialization
        of the variables previously defined (weights and biases)
    summary_writer=tf.summary.FileWriter(log_path,graph=sess.
        graph) # With this command we start writing the summaries
         for tensorboard
    for i in range(steps):
        # Cration of the batch
        x_input,y_true,names_batch,epoch=B.next_batch_train(
            train_i,train_l,train_n,batch_size)
        x_validation_batch,y_validation,names_val,epoch_v=B.
            next_batch_val(val_i,val_l,val_n,100)
        # Training procedure
        _,summary = sess.run([train_step,merged_variables],
            feed_dict={x:x_input,y:y_true,neurons_dropout:0.5})
        # I need to exploit the run session also for the
            summaries previously defined, in order to see the
            pregoresses on tensorBoard
        # I evaluate the accuracies in training and validation
            only every multiple of the validation checking step
        if i % val_checking_step == 0:
            acc_train,summary=sess.run([accuracy,
                merged_variables],feed_dict={x:x_input,y:y_true,
                neurons_dropout:0.5})
            acc_val,summary=sess.run([accuracy,merged_variables
                ],feed_dict={x:x_validation_batch,y:y_validation,
                neurons_dropout:1.0})
            # No dropout in validation and test performances: I
                want to use all the neurons to see how the
                training is proceeding

            print('Epoch {}, Step {}:'.format(epoch,i))
            print ('Training accuracy {} %'.format(acc_train
                *100))
            print ('Validation Accuracy {} %'.format(acc_val
                *100))

        summary_writer.add_summary(summary,i) # Updating the
            summaries each step
```

```
    # Final test check
    x_test=test_i
    y_test=test_l
    acc_test=sess.run(accuracy,feed_dict={x:x_test,y:y_test,
       neurons_dropout:1.0})
    print ('Final Test Accuracy: {} %'.format(acc_test*100))

    saver=tf.train.Saver()
    save_path=saver.save(sess , "Models/test1.ckpt")
    print("Model saved")
```

The proposed code will train the network using the Batch Normalization technique and plot a temporary result each certain number of steps. In practice, the result regard the accuracy of the train procedure, seen as the percentage of correct classification of the images. At the end of the training, a test is performed with images never seen by the network, in order to evaluate the final accuracy. The model is saved in a .ckpt file, which can be reused so that no more training is required.

Chapter 7

Conclusions

The aim of the entire training process made with the two different networks is to show the differences and the benefit of the different approaches.
It is quite clear, by simply looking the accuracies in the different tests, how Transfer Learning is much better than the custom CNN. As a matter of fact, we did not expect something different, because the huge differences in the deepness of the networks suggested better results even without testing them. But the point is clearly not this one, because our custom CNN could have been done even bigger than only 5 layer-deep and maybe could have reached accuracies more similar to Inception. But all this effort could not be justified when a technique like Transfer Learning is available. It is basically impossible to build a network even near to Inception and at the end it have required too much effort not only in the building phase, but moreover in the training process. This custom CNN took something like 8 hours to be trained from the scratch, while the retraining of Inception took just a couple of hours. So, Transfer Learning is surely the right choice for the network part of the prototype.
Some final considerations must be done also for the hardware. This prototype showed only benefit with respect to previous prototypes already tested during past years. In particular, the set of images are strongly improved in terms of stability and resolution and all the automatized system with Raspberry makes the life even more easier. Last year, an attempt with a *video* acquisition system has been made, with the dataset obtained by reprocessing the black and white video and extracting them in relation with the frame frequency. This technique did not produce a nice dataset, but the final results were quite similar. The main difference was the huge overfitting obtained with that dataset, because black and white pictures with low accuracies brought the system to a fast saturation. This new kind of dataset is much more reliable and, if more increased, could reach results even bigger than 97.8%. The benefit of this new dataset is also confirmed by the good results of the custom CNN which, even small, reached a good 94.2% with the proper parameters tuning.

In conclusion, this new prototype collected very good responses and could be ready to be introduced in the real time environment, as suggested by the following steps which will be completed in the next years.

7.1 Next steps

At the end of this work, we obtained a trained Neural Network, Inception which, thanks to the good dataset we built with our prototype is able to classify images for our purpose. Now, the last step is the effective testing of the network in real time.
Jetson TX2 can be used as the main element of the prototype and can be mounted in the truck and substituting completely Raspberry in the remote control of the prototype. At the same time it can run simple programs in order to exploit the resources of Inception which be already prepared. This means that, in the final phase of the project, the system can move inside a generic tunnel, collect images and immediately process them inside Inception to know in real time the result.
Another interesting problem regards the localization. Effectively we are going to collect images quite fast and the results can be shown after the inspected zone is already passed. It could be useful to associate each image to the proper localization inside the tunnel so that, at the end of the road, we can know exactly where the crack is and we can go back to eventually fix it. But the localization procedure in our environment can be quite difficult to implement. We are inside a tunnel and the GPS in not available and this means that the most immediate and efficient technique is not usable. During the development of this project, we tried to focus in parallel on some localization technique. The first idea was to use some sensors like the Hall Effect for detecting the number of rotations of the truck wheel during the motion and connecting it to the shooting frequency of the camera. The problem arose with the effective power of the sensor which was not able to detect the magnetic field of a magnet, even if was one of the strongest in commerce. We identify as the unique approach to use the bus of the truck for extract impulse on the gyroscope. This will be the other main implementation required by the system to be considered complete.

Bibliography

[1] W.Pitts W.S.McCulloch. Bull. math. biophys. 1943.

[2] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[3] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[4] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[5] Yurii Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[6] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[7] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[8] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[9] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[10] Seung-Nam Yu, Jae-Ho Jang, and Chang-Soo Han. Auto inspection system using a mobile robot for detecting concrete cracks in a tunnel. *Automation in Construction*, 16(3):255–261, 2007.

[11] Chia-Han Lee, Ya-Chu Chiu, Tai-Tien Wang, and Tsan-Hwei Huang. Application and validation of simple image-mosaic technology for interpreting cracks

on tunnel lining. *Tunnelling and Underground Space Technology*, 34:61–72, 2013.

[12] Simon Stent, Riccardo Gherardi, Björn Stenger, Kenichi Soga, and Roberto Cipolla. Visual change detection on tunnel linings. *Machine Vision and Applications*, 27(3):319–330, 2016.

[13] Konstantinos Makantasis, Eftychios Protopapadakis, Anastasios Doulamis, Nikolaos Doulamis, and Constantinos Loupos. Deep convolutional neural networks for efficient vision based tunnel inspection. In *Intelligent Computer Communication and Processing (ICCP), 2015 IEEE International Conference on*, pages 335–342. IEEE, 2015.

[14] Roberto Montero, Elisabeth Menendez, Juan G Victores, and Carlos Balaguer. Intelligent robotic system for autonomous crack detection and caracterization in concrete tunnels. In *Autonomous Robot Systems and Competitions (ICARSC), 2017 IEEE International Conference on*, pages 316–321. IEEE, 2017.

[15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. Going deeper with convolutions. Cvpr, 2015.

[16] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[17] Sanjeev Arora, Aditya Bhaskara, Rong Ge, and Tengyu Ma. Provable bounds for learning some deep representations. In *International Conference on Machine Learning*, pages 584–592, 2014.

[18] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[19] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, page 12, 2017.

[20] Sinno Jialin Pan, Qiang Yang, et al. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.

Chapter 8

Appendix A

Python code for camera control

Listing 8.1: Main

```
from sh import gphoto2 as gp
import signal, os, subprocess

# Everytime the camera is attached to the PI, a gphoto process
    starts automatically and does not allow any other actions on
    the camera.
# We need to close it

def killGphoto2Process():
    p = subprocess.Popen(['ps', '-A'], stdout=subprocess.PIPE)
    out, err = p.communicate()

    # Search for the process we want to kill
    for line in out.splitlines():
        if b'gvfsd-gphoto2' in line:
            # Kill that process!
            pid = int(line.split(None,1)[0])
            os.kill(pid, signal.SIGKILL)

# Lines in the terminal which allow to control the shutter of
    the camera
triggerCommand = ["--set-config=eosremoterelease=5"]

def captureImages():
    gp(triggerCommand) # Hold down the shutter
def stopImages():
    gp(endCommand) # Release the shutter

# Main
```

```
while(True):
    killGphoto2Process()
    captureImages()
```

The code is essentially very simple, because it has only to exploit the *gphoto2* library and its trigger command on the shutter of the camera. This kind of code allows to excite the shutter at its maximum speed and, when desired, the execution can be stopped through the command line. It is very interesting the first part of the code, the so called *killGphoto2Process* function. In the moment in which the camera is connected to the Raspberry, the library automatically generates the software connection for the opening of the internal memory of the camera and so creates a OS process. So, if we want to have access to the camera again, it is necessary to block this process: this function does that and in fact it is recalled every time the code starts, so that whatever previous process is in act, it will be blocked and our program can be executed without problem.

Chapter 9

Appendix B

Matlab code for Artificial expansion of the data

```
clear
close all
clc

path='source/';
format='.jpg';

tot_img=1744;

for ii=1189:tot_img
    ext='';
    if (ii<10)
        ext='000';
    elseif (ii>9 && ii<100)
        ext='00';
    elseif (ii>99 && ii<1000)
        ext='0';
    end

    name=[path ext num2str(ii) format],

    x = imread(name);


    step=90;
    tot_rot=3;
    for rr=1:tot_rot

        angle=step*rr;

        y=imrotate(x,angle);
```

```
            imwrite(y,['dest' num2str(angle) '/' num2str(ii) '-'
                num2str(angle) format]);
        end

    end
```

This is the simple Matlab code used for rotating images and enlarging the dataset. Actually, this code is able to rotate the images of 90,180 and 270 degrees, but we used only the 90 degrees rotation.