POLITECNICO DI TORINO

Master of Science Degree in Mechatronics Engineering

Master's Thesis

Image processing algorithms for synthetic image resolution improvement

Supervisor:
Prof. Marcello Chiaberge

Candidate:
Francesco Salvetti

A.Y. 2018/2019

Abstract

The aim of this thesis is to develop a Machine Learning algorithm for Multi-image Super-resolution (MISR). Super-resolution is a well known image processing problem, whose aim is to process low resolution (LR) images in order to obtain their high resolution (HR) version. The super-resolution process tries to infer the possible values of missing pixels in order to generate high frequency information as coherently as possible with the original images. In general, we can distinguish between two different super-resolution approaches: Single-image Super-resolution (SISR) and Multi-image Super Resolution (MISR). The former tries to build the best LR-HR mapping analysing the features of a single LR image, while the latter takes as input multiple LR images exploiting the information derived from the small differences between the images such as changes in the point of view position and orientation, in the lightening condition and in the image exposition and contrast.
The thesis had as first objective to take part to the competition called "PROBA-V Super Resolution", organized by the Advanced Concept Team of the European Space Agency. The goal of the challenge was to obtain High-resolution images from low resolution ones from a dataset of pictures took from the PROBA-V satellite of the ESA. The work has been developed under the direction of the PIC4SeR (PoliTO Interdepartmental Centre for Service Robotics), which aims to integrate it into its agricultural related projects, for the satellite monitoring of the fields status.
The analysis of the state of the art for what concerns super-resolution reveals that Machine Learning approaches outperform the classical algorithms proposed for this problem. In particular, Neural Networks have been widely used in literature for Single-image Super-resolution, while this approach for Multi-image Super-resolution is relatively new. An original model to deal with competition problem has been studied,

trained and tested. The obtained results show that the multi-image approach can help in the improvement of existing algorithms for super-resolution. However, several issues can be further addressed to increase the model efficiency and performance, making this particular topic interesting for future work development.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The incessant world population growth of the last century is creating the need for a new way to think to agriculture. Food request is constantly increasing and with it the need of lowering costs and increase production efficiency. A lot of technologies are under development for the agriculture field and several of them involve robots and drones. Service robotics applied to smart agriculture represent the future for food production and will be involved in the whole process, from planting to crop monitoring and harvesting.

One of the most promising idea is to use already available satellite images to have a constant source of information for automatic monitoring of fields parameters. The development of this technology would save the costs for frequent drone flies to collect field shots, but has its principal limitation in the free satellite resolution. For most of the available imagery sources, an entire field is represented by few pixels, making the remote monitoring process impossible. The development of resolution enhancement algorithms can be vital for this research field and can open the way to new commercial solutions for smart agriculture.

This thesis studies the problem of super-resolution, that is the process of synthetically increase the resolution of an image, trying to recreate additional pixels as coherently as possible with the original information. The approach adopted to assess this problem is the development of a custom Machine Learning algorithm, able to generate a high-resolution version of multiple available low-resolution images. The proposed model is able to take several shots of the same scene, took in different times, and merge them

in order to obtain a single high-resolution image. The model is entirely based on CNN (convolutional neural networks) and has been trained from scratch on the available satellite images. The work has been developed under the direction of the PIC4SeR (PoliTO Interdepartmental Centre for Service Robotics), which aims to integrate it into its agricultural related projects, and was used to take part to the "PROBA-V Super Resolution" challenge.

Proba-V Challenge

The "PROBA-V Super Resolution", organized by the Advanced Concept Team of the European Space Agency, is a challenge, whose aim is to obtain high-resolution images from low-resolution ones from a dataset of pictures took by the PROBA-V satellite of the ESA. The dataset consists in 1160 training scenes of the Earth surface, divided into RED and NIR spectral bands, for which one high-resolution image and several low-resolution images are provided. The goal of the competition is to provide the high-resolution images for 290 testing scenes for which only the low-resolution images are available.

How to read this work

The thesis is organized as follows. Chapter 2 presents a general view of the theoretic concepts related to Machine Learning and, in particular, Neural Networks. Chapter 3 presents a brief review of already available super-resolution algorithms, especially the ones with an approach based on Machine Learning. Chapter 4 gives an introduction to the Proba-V challenge and analyses the available dataset and the scoring methods. Chapter 5 discusses the work platform from both software and hardware point of view. Finally, Chapter 6 presents the proposed model and the obtained results.

Chapter 2

Machine Learning

The primarily aim of Machine Learning is to program computers to learn from data. In general, it refers to a different approach to solve a problem in which we have a certain input X and we want to find a correspondent output y. The general relation can be written as

$$y = f(X) \quad , \tag{2.1}$$

where f is a generic function, that can be written in an analytic form or not, depending on the addressed problem. To solve the problem, we have to program computers to learn from available data the nearest approximation of f.
Classic approaches to this kind of issues focus on write down the best algorithm to approximate the true function f. In practice, they create a model with a fixed structure that consists in a list of explicit rules and steps, that is able to produce an output as close as possible to y, given the input X, for every possible (X, y) couples. This kind of approach works well for situations in which exact algorithms have been studied, but there are a lot of problems that are too complex to be directly addressed by an algorithm.

Machine Learning, instead of trying to write the best algorithm, changes radically the perspective and focuses on *data*. Now, instead of having a fixed model, based on specific hard-coded rules, we use a really general one for almost all kind of situations and we change the model parameters until the output data correspond to what we expected. To do this update, we use a feedback on how good the model is currently

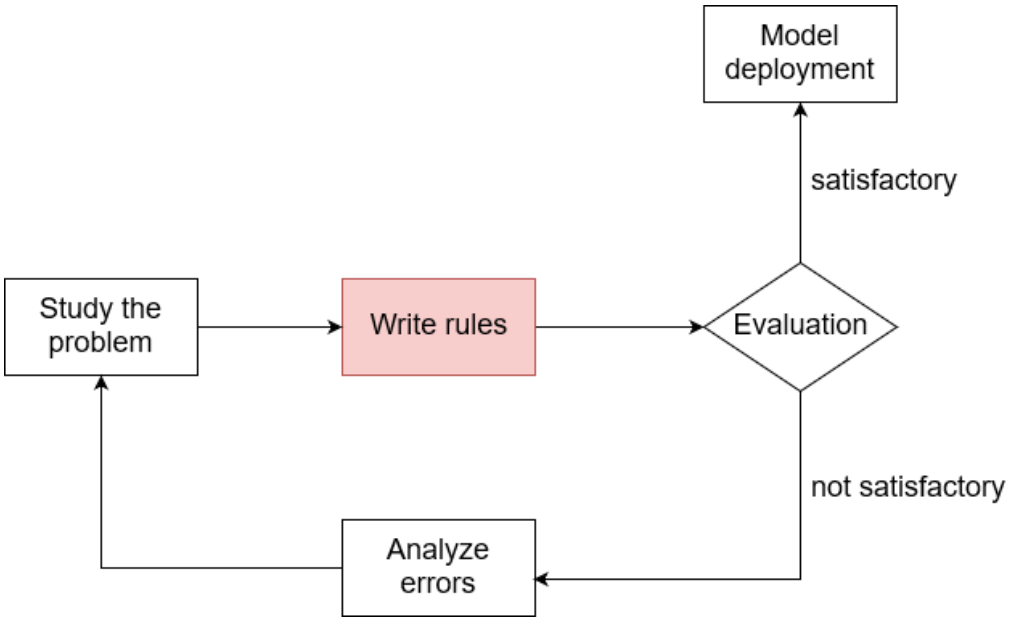
behaving, analysing the desired output y and the current output $\hat{y}$. If we are able to tune the parameters so that the model works fine for all the possible (X, y) couples, we can affirm that this system is a good approximation of the function f, even if we haven't directly write explicit rules down. The process of modifying those parameters is where the machine actually learns.

To provide a formal definition of what Machine Learning is, we can refer to the one presented by Tom Mitchell in 1997 [4]:

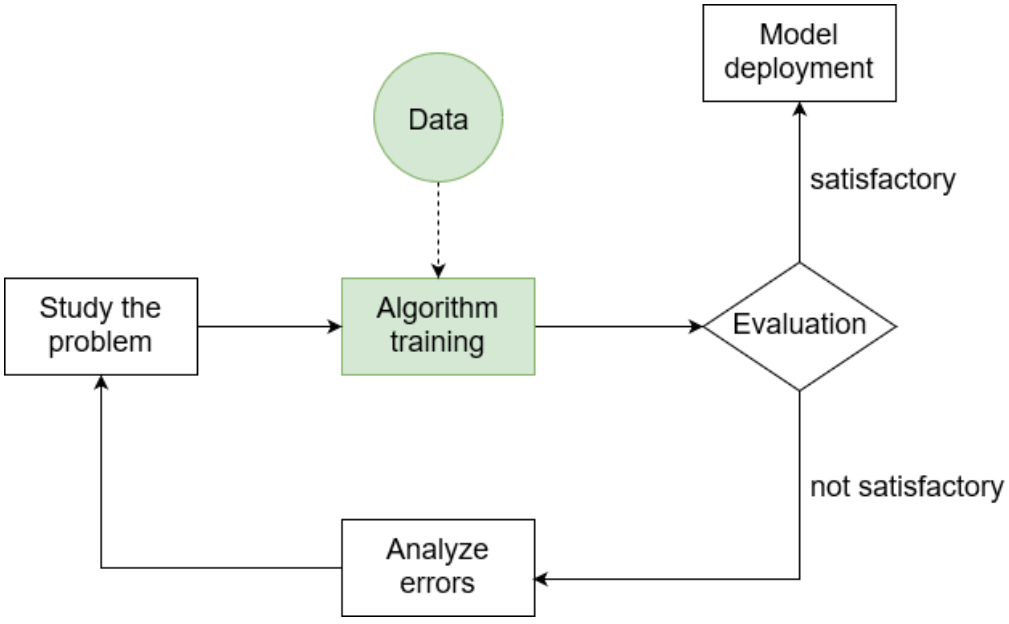
> "A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."

The task T referred by Mitchell is of course achieving the best algorithmic approximation of the function f; the performance P is a score of how well the model is behaving and is defined in various way, depending on the specific context; the experience E is the actual training process, where we tune the parameters on the basis of the available data.

In figure 2.1, a schematic representation of the classical and Machine Learning approaches is presented. The main difference between the two graphs is that in ML, instead of writing explicit rules, we modify the algorithm in the training process, in order to fit the available data.



(a)



(b)

Figure 2.1 – Traditional (a) vs Machine Learning (b) approaches

2.1 A brief history of Machine Learning

The year of birth of Machine Learning can be considered the 1943, when a neurophysiologist, Warren McCulloch and a mathematician, Walter Pitts, wrote a paper on the human brain activity, modelling a basic electrical circuit that emulated a simple neuron [8].

In the 1950s, several applications developed after that first contribution arose. In 1952, Arthur Samuel created a program that made an IBM computer get better in checkers the more it played it. In 1959, he also used for the first time the term *Machine Learning* [9], describing it as the "field of study that gives computers the ability to learn without being explicitly programmed". Other remarkable works of that period were the perceptron (1958) by Frank Rosenblatt [10] that was the first *feed-forward neural network* and the MADALINE – Multiple ADAptive LINear Elements (1959) by Bernard Widrow and Marcian Hoff of Stanford, that was the first neural network applied to a real problem (reducing echoes on phone lines).

Despite the big success of those works, during the the 1970s Machine Learning approach was almost abandoned, mainly due to the great success of the Von Neumann computer architecture and the technological limitations of those years, that made multilayer models almost impossible to be realized. The book *Perceptrons* (1969) from Minsky and Papert [11] shown that the perceptron of Rosenblatt was limited to only learn linearly separable patterns, marking the end of neural network research for the next decade.

Another fundamental step for Machine Learning development was in the 1980s, when for the first time the *backpropagation* algorithm was presented [12], that allowed to train multi-layered networks. After another silent period, at the end of the 1990s other progresses were made. In 1997, the Deep Blue IBM computer beat the world chess champion. In 1998, AT&T Bell Laboratories achieved good accuracy in recognizing handwritten digits.

In the 2000s, *Deep Learning* development gave Machine Learning another boost, allowing networks with a lot of layers to be trained and perform really complex tasks in different fields such as speech recognition or image processing. In the last ten years, a really huge number of Machine Learning-related researches has been published, with the development of a lot of ML-related projects such as GoogleBrain (2012) and DeepMind

(2014), as well as successful network architectures as AlexNet (2012) or ResNet (2015). This field of research is nowadays one of the most prolific, having gathered the attentions of big companies and industries and is definitely one the frontiers of Computer and Data Science.

2.2 Machine Learning algorithm categories

Machine Learning is a collective name that actually represents a lot of different algorithms, developed with different approaches. Common these algorithms are divided into the following categories:

- *supervised learning*: the computer is trained with labelled data, that means that a dataset of desired input/output couples is available
- *unsupervised learning*: takes a set of data that contains only inputs and try to find similarities and patterns in order to subdivided them in groups
- *semi-supervised learning*: falls in between the two previous group, having only a part of data labelled
- *reinforcement learning*: focuses on take actions in order to maximize the reward and minimize the risk and learns by observing effect of these decisions on the environment

Supervised learning methods include *classification* and *regression* algorithms. The first focus on study the pattern of input data in order to give it a label, identifying it as belonging to a particular class. The output is thus restricted to a specific set of values that represents all the possible classes taken in consideration. Regression algorithms can output any value and therefore are used to implement methods that want to modify the input, still conserving its nature and main characteristics. Super-resolution falls in the regression group, since focuses on manipulate some input images to get as output again an image, but with higher resolution.

2.3 Artificial Neural Networks

Artificial Neural Networks (ANN), usually simply called Neural Networks (NN), are maybe the most popular supervised learning Machine Learning algorithm and can be used for both classification and regression problems. They were conceived, as briefly explained in the section 2.1, from a simple mathematical model of the human brain functioning. They consist in a set of interconnected mathematical units, called *neurons*. Those neurons are the nodes af a directed graph structure whose edges are the links between them. ANN are organized in different *layers*, that consist in a number of neurons, all at the same depth. The tiniest network has two layers: the input one and the output one. The *hidden layers* are all those layers in between, that cannot be seen by the external. In a fully-connected NN, all the neurons of a layer are connected to all the neurons of the next layer and there are no connections between non-consecutive layers.

There exist two main types of Neural Networks: feed-forward NN and Recurrent NN. In the first case, no cycles are admitted in the connections between the different neurons, and the information flows only in a single direction, from the inputs to the outputs. The second type of networks allows cycles, that means that the outputs of a certain neuron can be brought back as input to a neuron of a previous layer. In this way, the network gains an internal state and can be used to implement a temporal dynamic behaviour, making the Recurrent NN particularly suited for tasks such as speech or handwriting recognition. Since the work I developed is based on feed-forward NN, from now on I will refer to them, only.

Recalling equation (2.1), the network represents f, that is the function that associates to a certain input X an output y. Figure 2.2 shows the basic scheme of a fully-connected NN with two hidden layers. In this example, the network takes as input a vector of two values and outputs a single one. If we assume that those values can be any real number, the model represents a $\mathbb{R}^2 \longrightarrow \mathbb{R}$ function.

To understand how the information changes as it flows from the input layer to the output one, we first have to analyse how an artificial neuron works.

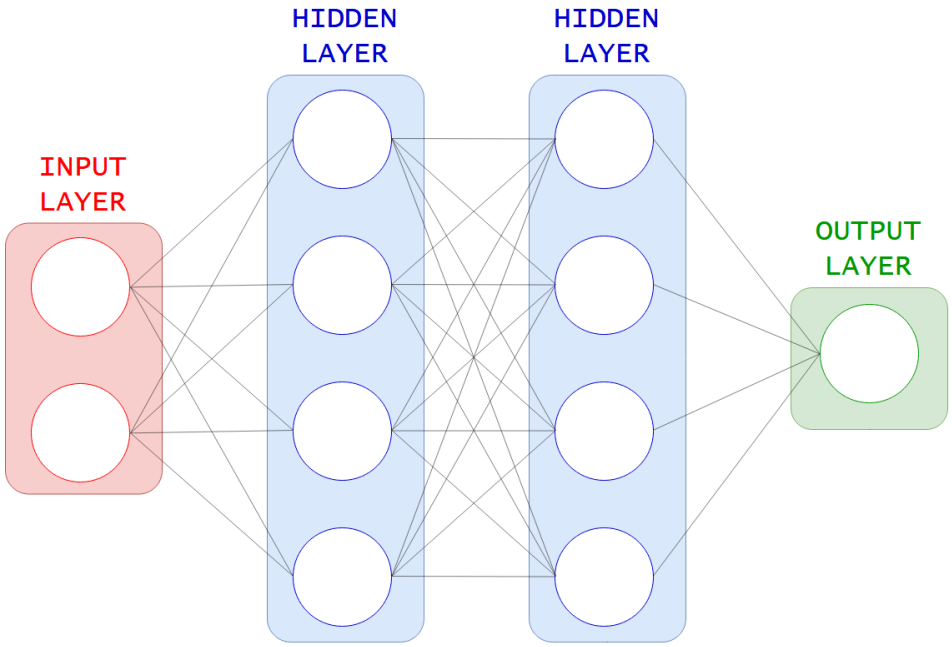


Figure 2.2 – A fully-connected NN with two hidden layers

2.3.1 Neuron model

A neuron is a cell, main component of the nervous tissue. It is composed by a cell body called *soma*, many branching extensions called *dendrites*, plus one very long extension called the *axon*. At the end of the axon, there are many terminals with at the end of them the *synapses*. A neuron can receive electric signals from other neurons through these synapses and the dendrites. If it receives a certain number of signals in a given time, the neuron transmit its own signal through the axon to other neurons. In figure 2.3 a schematic representation of a biological neuron is presented.

First generation of artificial neurons

The very first mathematical model of a neuron was the one presented by McCulloch and Pitts as said in section 2.1 and later called *artificial neuron*. [8] It's easy to demonstrate that we can build any logical preposition with this model. As an example, figure 2.4 shows the basic binaries operations, assuming that a neuron is activated when at least

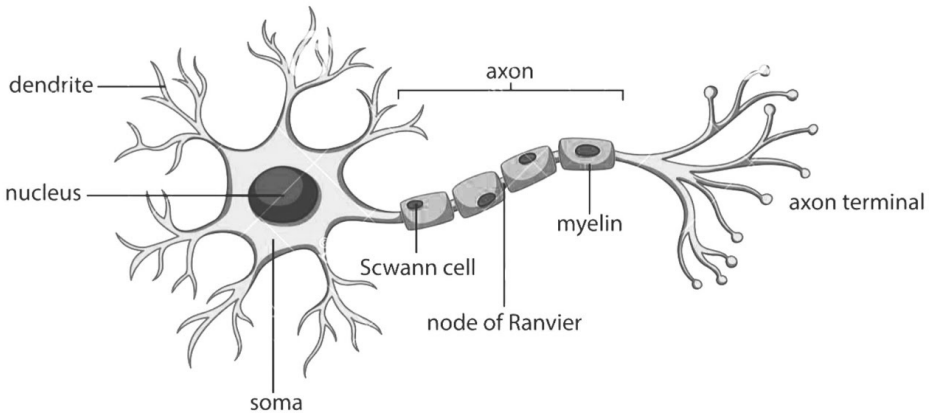


Figure 2.3 – A biological neuron

two inputs are active.

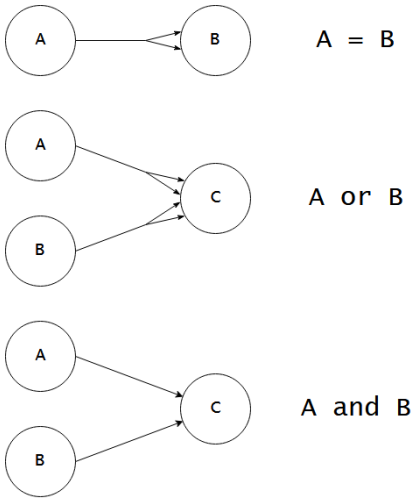


Figure 2.4 – Boolean operations with artificial neuron model

A generalization of the neuron model presented so far is the *LTU* (linear threshold unit). Instead of having only on/off values, we now have N numbers as inputs $(x_1, ..., x_N)$. Each connection (the synapses in the biological neuron) has an associated weight $(w_1, ..., w_N)$. The unit performs the weighted sum of all the inputs and applies a certain activation function φ to the result.

Therefore, the output can be written as:

$$y = \varphi\Big(\sum_{i=1}^{N} w_i\, x_i\Big) = \varphi(\mathbf{w}\cdot\mathbf{x}) \quad , \tag{2.2}$$

where $\mathbf{w}$ and $\mathbf{x}$ are respectively a row vector with the connections weights and a column vector with the input values. The applied activation function is usually the *Heaviside step*, the *sign* function or generically a *threshold gate*, that outputs 1 if the weighted sum of the inputs is above a certain value:

$$H(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

$$\operatorname{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ 1 & \text{if } z > 0 \end{cases} \tag{2.3}$$

$$\operatorname{thr}(z) = \begin{cases} 0 & \text{if } z < \theta \\ 1 & \text{if } z \geq \theta \end{cases}$$

This model of a neuron, with an added bias term obtained with an additional input x_0 with fixed value equal to 1, was used for the Perceptron of Frank Rosenblatt in 1958 (see section 2.1). That first type of Artificial Neural Network was used for a classification problem and simply used one layer of LTUs (the output layer). The network was fully-connected, meaning that each input was connected to every neuron. Figure 2.5 shows a LTU representation with bias input. The actual bias b is the weight w_0. The function implemented by the model is:

$$y = \varphi\Big(\sum_{i=1}^{N} w_i\, x_i \;+\; w_0\Big) = \varphi(\mathbf{w}\cdot\mathbf{x} + b) \tag{2.4}$$

The *Perceptron*, having simply one layer of neurons, has the intrinsic limit of being

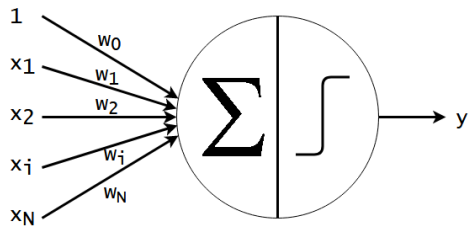
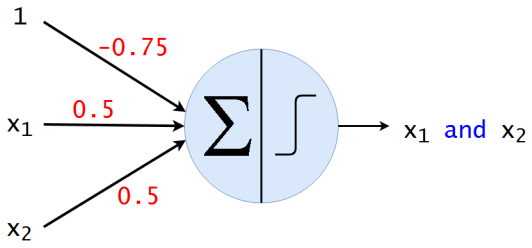
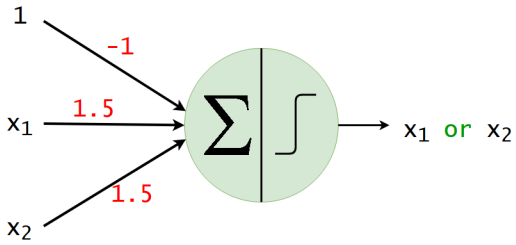


Figure 2.5 – Linear Threshold Unit with bias input

able to correctly classify only linearly separable features. As an example, using the Heaviside step as activation function, it's easy to perform simple boolean operations between two inputs as shown in figure 2.6. The two problems (*and*/*or* boolean functions) have linearly separable inputs, so they can be implemented by means of a single neuron. More complex functions, such as the *xor*, don't have linearly separable inputs and cannot be implemented with a single layer (see figure 2.7). To cope with these more complex situations, we have to add at least one hidden layer. This type of neuron model was the basic unit of the so-called **first generation** of neural networks, that were able to implement every binary (discrete) function with a single hidden layer.

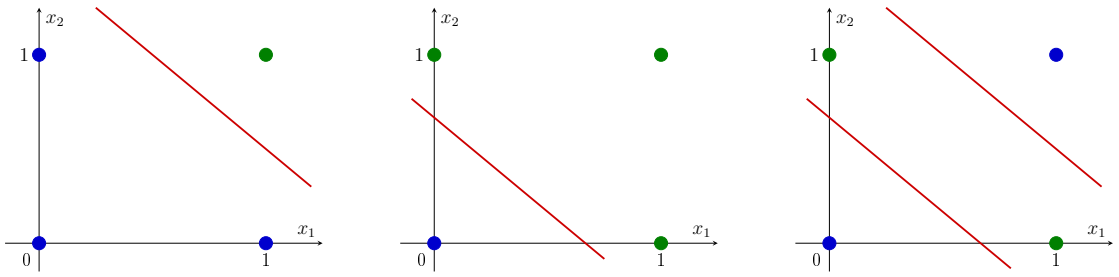


x_1	x_2	$z = w_0 + w_1 x_1 + w_2 x_2$	$\varphi(z)$
0	0	-0.75	0
0	1	-0.25	0
1	0	-0.25	0
1	1	0.25	1



x_1	x_2	$z = w_0 + w_1 x_1 + w_2 x_2$	$\varphi(z)$
0	0	-1	0
0	1	0.5	1
1	0	0.5	1
1	1	2	1

Figure 2.6 – *and*/*or* operations with LTU



(a) *AND* (b) *OR* (c) *XOR*

Figure 2.7 – *AND* and *OR* operations have linearly separable input data, while *XOR* needs at least two lines

Second generation of artificial neurons

To further generalize the model of the artificial neuron, we can change the activation function φ. Instead of using a binary function as the threshold gate (equation 2.3), that outputs either 0 or 1, it's possible to use a continuous function. In this way, we can build a network that can implement every continuous function with as single hidden layer.

One of the most used continuous activation is the *sigmoid*, also called *logistic function*:

$$\sigma(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + e^{-(\sum_{i=1}^{N} w_i\, x_i + b)}} \tag{2.5}$$

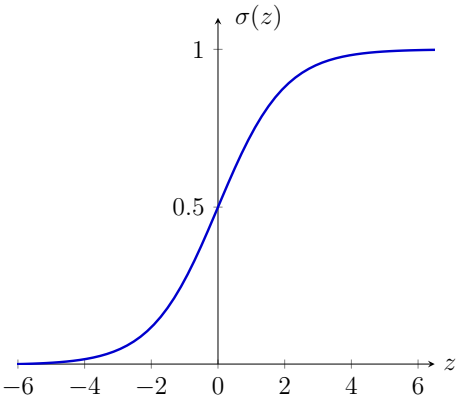
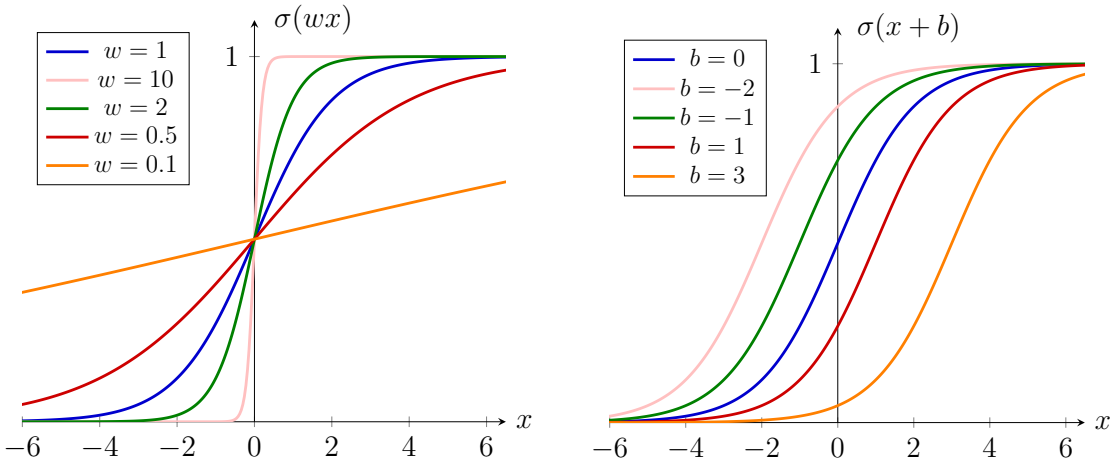


Figure 2.8 – Sigmoid activation function

Figure 2.8 shows the sigmoid output for the input z equal to the weighted sum of the neuron inputs. This function can be seen as a generalization of the step activation of the LTU. Effects of changes to weights and bias, considering a single input x, are shown in figure 2.9.



(a) Effect of changes to weight w

(b) Effect of changes to bias b

Figure 2.9 – Sigmoid activation function: effect of changes to weights and bias

The great advantage of using this type of activations is the fact that the artificial neuron can now handle analog value, since every value between 0 and 1 is possible. Furthermore, the function is now differentiable in each point of the domain, fact that helps the learning process, that is based on computing the gradient. The symbol of second generation neurons is shown in figure 2.10.

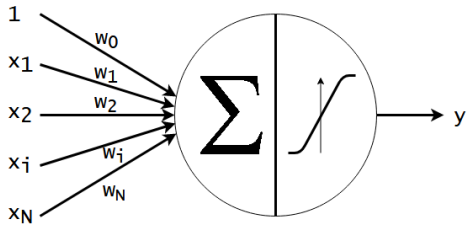


Figure 2.10 – Second generation artificial neuron symbol

Other activations

Linear The linear activation is simply a straight line with unitary slope. It's basically a situation when no activation at all is applied, and the output is exactly the weighted sum of the input plus the bias.

$$\varphi(z) = z = \sum_{i=1}^{N} w_i \, x_i + b \tag{2.6}$$

Tanh The *tanh* function is quite similar to the sigmoid, but with output ranging from -1 to 1. It has been proved that in certain conditions that activation behaves better than the sigmoid, mainly because it has higher gradient (ranging from 0 to 1) and is symmetric with respect to 0.

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2 \cdot \sigma(2z) - 1 \tag{2.7}$$

Softmax The *softmax* activation is usually used in classification problems as last layer. It basically applies a sigmoid-like activation to each neuron of the layer, normalizing all the outputs such that their sum is always 1. It's particularly useful for classification since the output vector can be used to represent the probability of that particular instance of belonging to the various classes. Considering a layer with F neurons, each component y_i of the output vector is equal to:

$$a_i = \frac{e^{z_i}}{\sum_k e^{z_k}} \quad k = 1, ..., F \tag{2.8}$$

From that, it holds:

$$\sum_{i=1}^{F} a_i = 1 \tag{2.9}$$

Thus, it is possible to consider the outputs as a discrete probability distribution among F different classes.

ReLU The *ReLU* (rectified linear unit) activation is a function defined as the positive part of its argument:

$$\text{ReLU}(z) = z^+ = \max(0, z) \tag{2.10}$$

When z is below 0, the unit is said to be inactive, since it outputs 0. When z is higher than 0, the unit behaves as a linear activation. The ReLU is widely used for deep neural networks, since it has been proved to be better for training deep structures with respect to classical activations as the sigmoid or the tanh.

Leaky ReLU The *Leaky ReLU* is a variation of the ReLU that allows a small gradient even when the unit is not active.

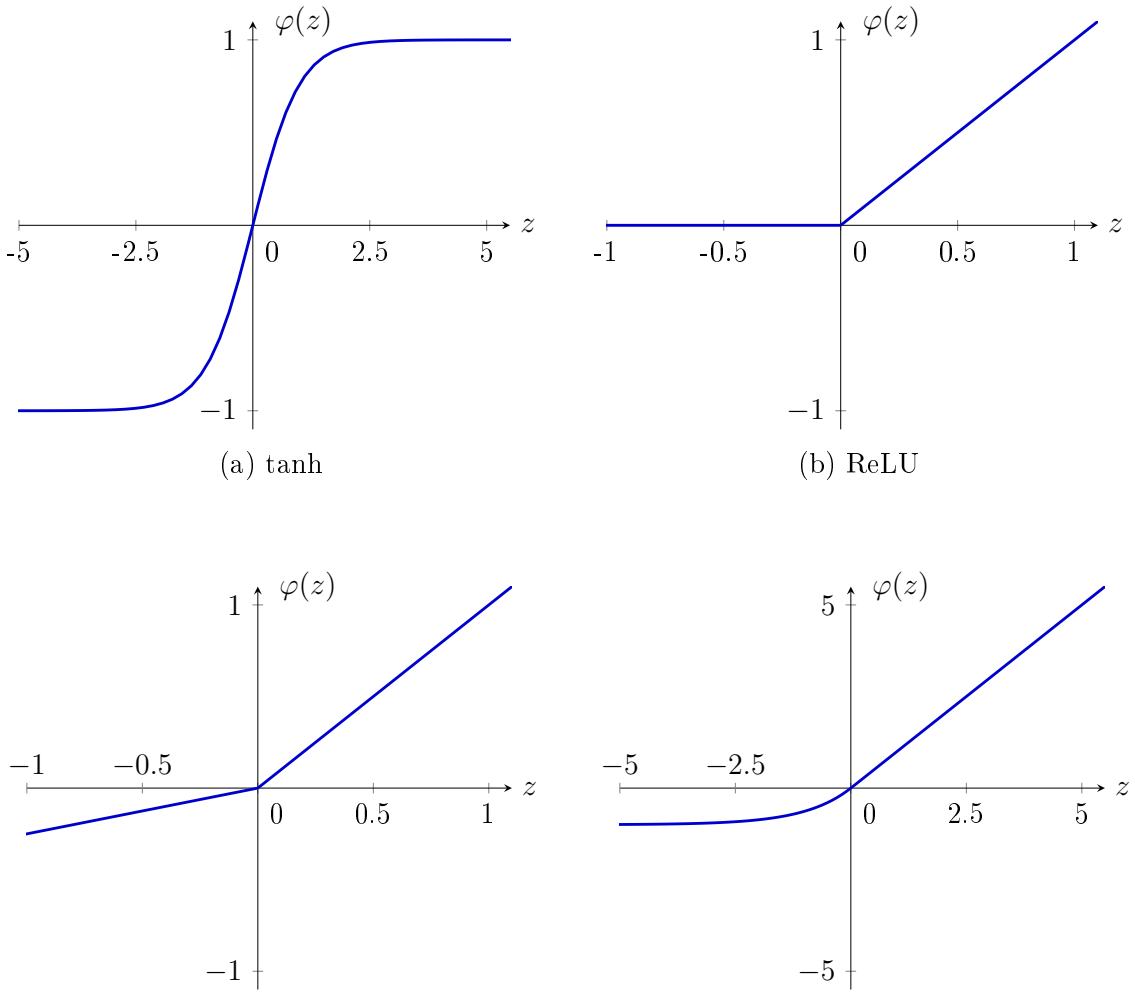
$$\mathrm{LReLU}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha\, z & \text{otherwise} \end{cases} \tag{2.11}$$

α is a parameter to be chosen by the network designer.

Parametric ReLU The *Parametric ReLU* is a Leaky ReLU in which α is not chosen *a priori* but can be optimized during the learning process.

ELU The *ELU* (exponential linear unit) is a particular ReLU in which the non-active part of the characteristic is neither constant nor linear but exponentially decreasing.

$$\mathrm{ELU}(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha\,(e^z - 1) & \text{otherwise} \end{cases} \tag{2.12}$$



(a) tanh (b) ReLU

(c) Leaky ReLU: $\alpha = 0.25$ (d) ELU: $\alpha = 1$

Figure 2.11 – Different activation functions

2.3.2 Training an Artificial Neural Network

After analysing the mathematical model of a neuron, we should focus on how to make a network learn. To do this, we should have a dataset composed of input-output (X, y) couples. During the training, we will use those known couples, to change the network parameters until it is capable of predict the desired outputs.

The training process takes place in two consecutive steps. First, a certain input X is propagated across the network until we get the predicted output $\hat{y}$. Then, we should

compute the error between the desired output y and the predicted $\hat{y}$ and update weights and biases in order to minimize it. Basically, the training process is a minimization problem, and the function to be minimized is called *loss* or *cost function*. It is a $\mathbb{R}^n \longrightarrow \mathbb{R}$ function, where n is the number of trainable parameters of the network, that are all the weights and biases of the neurons:

$$C(w_{11}, ..., w_{ij}, ..., w_{KN}, b_1, ..., b_j, ..., b_N) = C(\mathbf{p}) \quad , \tag{2.13}$$

where i is the index over the neuron inputs, j the index over the different neurons, K is the maximum number of inputs per neuron, N is the total number of neurons,and $\mathbf{p}$ is the set of all the networks parameters. The cost function associates to these parameter a real number, representing the error of the network predictions on a given set of (X, y) couples.
A typical loss function is the *MSE* (mean squared error), also called L2 loss:

$$\text{MSE}(y, \hat{y}) = \frac{\sum_{i=1}^{B} (\hat{y}_i - y_i)^2}{B} \tag{2.14}$$

The loss function takes as input the predicted $\hat{y}$ and the real y for a *batch* (vector) of inputs. The number B of different inputs in the batch is called *batch size*. The value of the error computed with the MSE is clearly dependent on the parameters values. Since the training phase is done over a certain fixed dataset, we can consider the MSE as a cost function that associate an error to the actual values of the networks parameters for the given batch. Figure 2.12 shows the MSE for a given batch in function of a generic single weight w. The cost function presents a minimum for a certain value w^* of the parameter: the aim of the training phase is to update w to make it assume the value w^*.

Gradient Descent

Since the actual cost function has a number of dimension equal to the number of parameters, it's clearly impossible to compute its value for each possible situation. An algorithm to update the parameters in order to move towards the minimum direction should be use. These algorithms are called *optimizers*. The simplest optimizer is the gradient descent (GD), and is based on the computation of the gradient of the loss

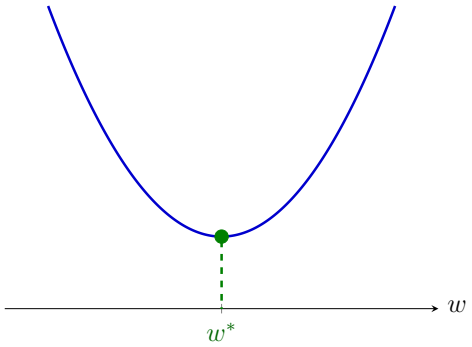


Figure 2.12 – Cost function for a single parameter w

function. Considering small variations of the parameters $\mathbf{p} = (p_1, ..., p_n)$, where n is the total number of parameters, we can approximate the correspondent change in the cost function:

$$\Delta C \approx \frac{\partial C}{\partial p_1}\Delta p_1 + \frac{\partial C}{\partial p_2}\Delta p_2 + ... + \frac{\partial C}{\partial p_n}\Delta p_n = \nabla C \cdot \Delta \mathbf{p} \quad , \tag{2.15}$$

where $\nabla C \equiv \left(\frac{\partial C}{\partial p_1}, \frac{\partial C}{\partial p_2}, ..., \frac{\partial C}{\partial p_n}\right)$ is the gradient of the cost function with respect to the parameters in the current point.
Given the relation 2.15, we can derive what should be the update to the parameter in order to *decrease* the cost function.

$$\nabla C \cdot \Delta \mathbf{p} < 0 \quad \Rightarrow \quad \Delta C < 0 \tag{2.16}$$

Thus, a possible good choice of $\Delta \mathbf{p}$ is:

$$\Delta \mathbf{p} = -\eta \nabla C \qquad \Rightarrow \quad \nabla C \cdot \Delta \mathbf{p} = -\eta \left\| \nabla C \right\|^2 < 0 \quad \text{if} \;\; \eta > 0 \tag{2.17}$$

The quantity η is called *learning rate* and is always positive. The choice of a good learning rate is vital, since a too small value causes a really slow convergence, while a too large value can cause an unstable behaviour. Usually the learning rate is decreased during the training process when the loss tends to stagnate, in order to allow a more fine search for local minima.
Figure 2.13 shows gradient descent in a unidimensional case: updating w as in equation

2.17 allows to slowly shift from the actual point towards the minimum w^*.

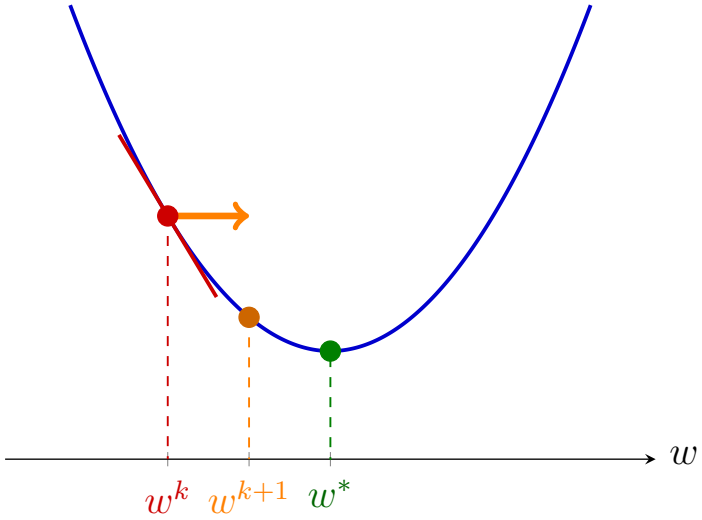


Figure 2.13 – Gradient descent: since in the current point the derivative is negative, the update will be positive

At this point, we have a rule to update the weights and the biases of the neurons at each iteration of the algorithm. Since the cost function is computed for a batch of inputs as said before, the single update to the parameters will usually derive only from a part of the training dataset. Once the training process has been repeated for the whole dataset (i.e for all the different batches), a training *epoch* is said to be completed. The process is then repeated for many epochs, until it gets convergence to a minimum loss point.

Backpropagation

Since we have a lot of neurons, each with a lot of parameters, to apply gradient descent we have to compute the partial derivative with respect to each of the n parameters. This operation would require a big computational effort if we have many layers with a high number of neurons, but the *backpropagation* algorithm comes in help to speed up the process. This procedure was applied to Neural Networks for the first time in

the 1980s and allows to reuse computations performed in the last layers to obtain the derivatives of the previous layers through an iterative algorithm.

The intuition of the algorithm can be understood referring to the simple situation represented in figure 2.14 with two layers and three neurons.

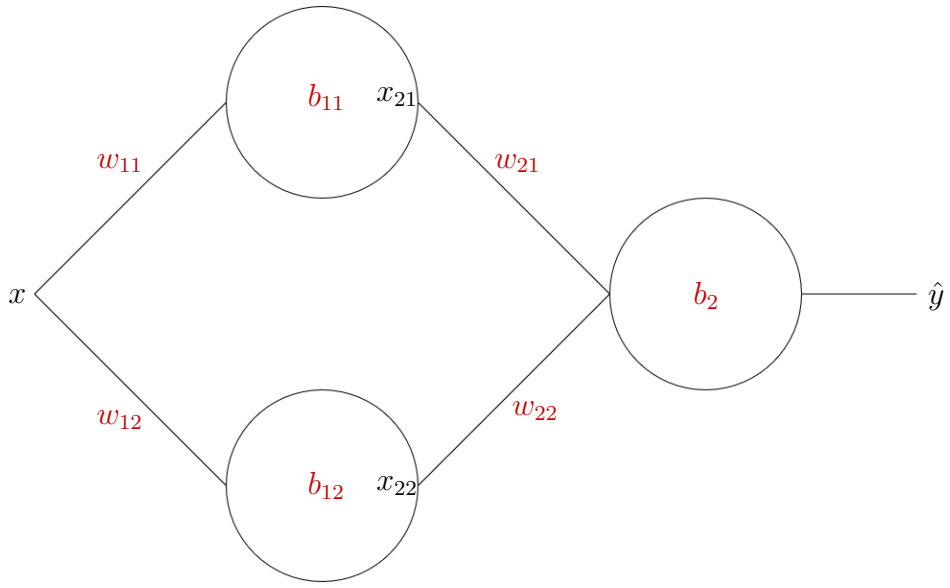


Figure 2.14 – Simple Neural Network to apply the backpropagation algorithm

To update a generic weight w_{ij} (where i is the layer and j is the neuron), we apply equation 2.17:

$$\Delta w_{ij} = -\eta \frac{\partial C}{\partial w_{ij}} \tag{2.18}$$

Starting from the last layer, we first recall equation 2.4 to compute the second layer output:

$$\hat{y} = \varphi_2\Big(\sum_{j=1}^{2} w_{2j}\, x_{2j} \;+\; b_2\Big) \tag{2.19}$$

We can compute the derivatives with respect the two weights w_{21} and w_{22} nad the bias b_2 applying the chain rule:

$$\frac{\partial C}{\partial w_{2j}} = \frac{\partial C}{\partial \varphi_2}\frac{\partial \varphi_2}{\partial w_{2j}} \qquad \frac{\partial C}{\partial b_2} = \frac{\partial C}{\partial \varphi_2}\frac{\partial \varphi_2}{\partial b_2} \tag{2.20}$$

The first term of the equations depends on the chosen loss and activation and we can call it δ_2, while the second terms can be rewritten as:

$$\frac{\partial \varphi_2}{\partial w_{2j}} = x_{2j} \qquad \frac{\partial \varphi_2}{\partial b_2} = 1 \tag{2.21}$$

Thus, the derivatives with respect to the last layer parameters are:

$$\frac{\partial C}{\partial w_{21}} = \delta_2 \, x_1 \qquad \frac{\partial C}{\partial w_{22}} = \delta_2 \, x_2 \qquad \frac{\partial C}{\partial b_2} = \delta_2 \tag{2.22}$$

Now we should obtain the derivatives for the first layer. The output of the first neuron is the input x_1 of the second layer:

$$x_1 = \varphi_1(w_{11} \, x \, + \, b_{11}) \tag{2.23}$$

With the same reasoning, we can compute the derivative:

$$\frac{\partial C}{\partial w_{11}} = \frac{\partial C}{\partial \varphi_1} \frac{\partial \varphi_1}{\partial w_{11}} = \frac{\partial C}{\partial \varphi_2} \frac{\partial \varphi_2}{\partial \varphi_1} \frac{\partial \varphi_1}{\partial w_{11}}$$

$$\delta_{11} = \frac{\partial C}{\partial \varphi_2} \frac{\partial \varphi_2}{\partial \varphi_1} = \delta_2 \frac{\partial \varphi_2}{\partial \varphi_1}$$

$$\frac{\partial C}{\partial w_{11}} = \delta_{11} \, x \tag{2.24}$$

Thus, we can use the computation of δ_2 performed for the second layer, also for the derivatives of the first layer. Iterating this reasoning for all the parameters, also for more complex networks, the complete backpropagation algorithm can be derived.

Other loss functions

Together with the MSE (equation 2.14), other losses are commonly used when dealing with Artificial Neural Networks training.

Mean absolute error The MAE, also called L1 loss, is similar to the MSE, but is based on the simple difference between desired output y and predicted output $\hat{y}$. Considering B as the batch size, the loss is defined as:

$$\text{MAE}(y, \hat{y}) = \frac{\sum_{i=1}^{B}(\hat{y}_i - y_i)}{B} \tag{2.25}$$

Cross entropy The cross entropy loss is the most used loss for classification problems, since is particularly suitable to deal with probability distribution outputs. Considering C as the number of possible classes (and so the dimension of the output vector), the loss is defined as:

$$\text{CE} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \tag{2.26}$$

Categorical cross entropy It's a particular case of the cross entropy loss, used for multi-class classification problems, when the last layer has a softmax activation. Since the real output y is a vector with a single entry equal to 1, and all the others equal to 0, considering equation 2.8 and 2.26 together, we get:

$$\text{CE} = -\log(\hat{y}_{\text{real}}) = -\log\left(\frac{e^{z_{\text{real}}}}{\sum_i^C e^{z_i}}\right) \quad , \tag{2.27}$$

where $\hat{y}_{\text{real}}$ is the value of the predicted vector that corresponds to the real class (the 1 in the real vector y).

Binary cross entropy It's another particular case of the cross entropy loss, used for multi-label classification problems (when the element can belong to more than one class). The last layer has usually a sigmoid activation, since each output value represents a probability independent from the others (we can get more than a single 1). In this case the problem is usually subdivided in C binary problems, where the value $\hat{y}_i$ represent the network confidence of that element belonging to class i, while $1 - \hat{y}_i$ represents the confidence of non-belonging. Applying the cross entropy loss (equation 2.26) to each $(\hat{y}_i, 1 - \hat{y}_i)$ couple as if it was a dual class classification problem (with $\hat{y_{i1}} = \hat{y}_i$ and

$\hat{y_{i2}} = 1 - \hat{y_i}$), we get:

$$\mathrm{CE}_i = -\sum_{j=1}^{2} y_{ij} \log(\hat{y}_{ij}) = -y_i \log(\hat{y_i}) - (1 - y_i) \log(1 - \hat{y_i}) \tag{2.28}$$

Since the ground-truths y_i are either equal to 1 or equal to 0, the loss can be written as:

$$\mathrm{CE}_i = \begin{cases} -\log(\hat{y_i}) & \text{if } y_i = 1 \\ -\log(1 - \hat{y_i}) & \text{if } y_i = 0 \end{cases} \tag{2.29}$$

Then the different CE_i are simply summed up:

$$\mathrm{CE} = \sum_{i=1}^{C} \mathrm{CE}_i \tag{2.30}$$

Other optimizers

Also for the optimizer there exist a lot of different algorithms. Here we will briefly analyse the most used ones.

Batch Gradient Descent It's simply gradient descent applied to the whole training dataset at the same time, thus when the batch size is the size of the training dataset. Since a lot of data is evaluated in just one update, batch gradient descent is usually slow and can create memory problem.

Stochastic Gradient Descent I't gradient descent when the batch size is 1, thus each training point is propagated through the network singularly and the optimization is done on it alone. Contrary to batch gradient descent, it's usually fast, since it uses really few data at a time. The main problem related to stochastic gradient descent is that the objective function tends to heavily oscillate in consequent updates, since the optimization is local and not global.

Mini-batch Gradient Descent It's the intermediate condition between the previous: batch size is fixed to a certain value higher than 1 but lower than the dataset size. In this way it reduces oscillations typical of stochastic GD, but always keeping updating

speed and memory usage acceptable. Common values for batch size are between 32 and 256, but it can change depending on the specific problem.

Momentum It's a method that helps accelerate GD in the relevant direction, damping oscillations in the others. This method slightly change the update equation, keeping memory of the direction of the previous updates. The update vector of the current step m_t is built with the current gradient and with a fraction γ of the previous step update vector m_{t-1}.

$$\begin{aligned} m_t &= \gamma m_{t-1} + \eta \nabla C \\ \Delta \mathbf{p} &= -m_t \end{aligned} \tag{2.31}$$

The parameter γ, called momentum term, is always positive and lower than 1.

RMSprop This algorithm, together with others like *Adagrad* or *Adadelta*, adapts the learning rate to the different parameters performing updates proportional to the associated features frequency. The principle of the algorithm is to divide the learning rate by a quantity that is different for each parameter and is related on the previous updates. A running average E_t at time step t is defined recursively as:

$$E_t = \gamma E_{t-1} + (1-\gamma)(\nabla C)^2 \tag{2.32}$$

The term γ is similar to the momentum term and is always lower than 1. The running average accumulates a fraction of the previous squared value of the gradients. If a particular parameter p_i has been updated a lot in the past steps (i.e had high gradient), the correspondent running average value $E_{t,i}$ will be high. After some steps with small updates, $E_{t,i}$ will then start to decrease.
The update vector is computed dividing the learning rate by the square root of E_t plus a little value ϵ, to avoid dividing by 0. In this way, a frequently updated parameter will have a lower learning rate, while a rarer one will have a higher learning rate.

$$\Delta \mathbf{p} = -\frac{\eta}{\sqrt{E_t + \epsilon}} \nabla C \tag{2.33}$$

Adam Adaptive Moment Estimation algorithm, called Adam, computes adaptive learning rates like RMSprop, considering also a sort of momentum. In addition to a running average of the squared gradients E_t, Adam keeps also a running average of the gradients m_t, that can be seen as a sort of momentum with friction. The two averages at each time step t are defined as:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)(\nabla C) \\ E_t &= \beta_2 E_{t-1} + (1-\beta_2)(\nabla C)^2 \end{aligned} \tag{2.34}$$

Since the two vectors are initialized with zeros, they are biased towards 0, especially during the first steps and if the two parameters β_1 and β_2 are close to 1. To decrease the effects of these bias, the two values are corrected as:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{E}_t &= \frac{E_t}{1-\beta_2^t} \end{aligned} \tag{2.35}$$

In this way, when t is small, so in the first steps, if the values starts from 0, almost the entire value of te gradient is used to compute the two averages. When t increases the denominators tends to 0, making the bias correction irrelevant.
The parameter update is then computed as:

$$\Delta \mathbf{p} = -\frac{\eta}{\sqrt{\hat{E}_t} + \epsilon} \hat{m}_t \tag{2.36}$$

2.4 Convolutional Neural Networks

A particular type of feed-forward Artificial Neural Networks are the Convolutional Neural Networks (CNNs). This type of networks has emerged since the 1980s and has been developed starting from the study of particular neuron cells present in the visual cortex of animals. These neurons reacts to informations coming only from a specific region of the visual field and to particular geometric shapes like horizontal or vertical lines. On the following levels, there are, then, other neurons that have a bigger receptive field and can react to more complex patterns, starting from the informations of the previous level. Building a network with a similar architecture led to the discovery of CNNs and started the prolific field of Machine Learning applied to image processing.

2.4.1 2D convolutions

The basic element of a CNN is the *convolutional layer*. Since we're now referring to an image, the inputs are now organized as a 2D vector (i.e. a matrix) with shape of $W \times H$ pixels. The first big difference of CNNs with respect to fully-connected NNs is that neurons have now a *receptive field*, meaning that elaborates only the information coming from some part of the input matrix. Usually this receptive field is a squared area of $K \times K$ pixels. The operation implemented by a neuron is the same described in the previous sections, so it's a weighted sum of the inputs, then fed to an activation function φ (see equation 2.4). Since we have a receptive field of $K \times K$ pixels, each neuron will have a correspondent number of weights. That matrix of weights is called *convolutional kernel* and is applied to the entire input image. This means that in a convolutional layer different neurons are set up with *shared weights* in order to cover all the input pixels. The outputs of all the neurons are again organized in a 2D vector, and the result is a new image. The kernel is also called $filter$, since the operation is assimilable to the application of filters as in classical image processing. A single convolutional layer can have as output several images and therefore learn different kernels.
Figure 2.15 represents a convolutional layer that takes as input a 4×4 image and has 4 neurons with 3×3 kernel that together output the green image. The output matrix is reduced in dimensions since from a 4×4 square we can only consider four 3×3 sub-squares.

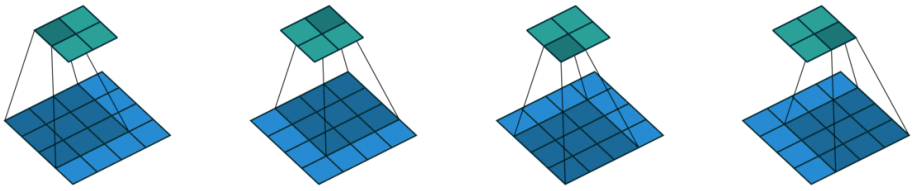


Figure 2.15 – Convolutional layer: $W = H = 4$, $K = 3$

Figure 2.16 represents a numerical example of a convolution with linear activation $\varphi(z) = z$ and with kernel:

$$\begin{pmatrix} 0 & 1 & 2 \\ 2 & 2 & 0 \\ 0 & 1 & 2 \end{pmatrix}$$

The filter is shifted along the input image (in blue) and the weighted sum between the pixel values and the kernel values is computed. The result is the value of the correspondent pixel in the output image (in green).

An important set of hyper-parameters [1] of a convolutional operation are the *strides*. These numbers represent the distances in pixels along each axis between two consecutive positions of the kernel. This means that when non-unitary strides are considered the kernel is shifted of more than one pixels and less neuron are implemented (i.e. the output image is smaller). If the same stride is used for all the axes , a single number S is used to represent that value.
Figure 2.17 represents a convolutional layer that takes as input a 5×5 image and uses a 3×3 kernel with strides 2 for both the axes. The output image is again a 4×4 image, even though the input image is bigger than the previous example: this is due to the fact that now the receptive fields of the different neurons are not shifted of a single pixel, but two.

Until now we saw always get an output matrix smaller with respect to the input and this come from the fact that a convolution aggregates the information coming from

[1] A hyper-parameter is a variable set by the network designer that is not changed during the optimization phase. A good choice of the hyper-parameters of a network is crucial during the design of a network and can vary sensibly the performance of a model.

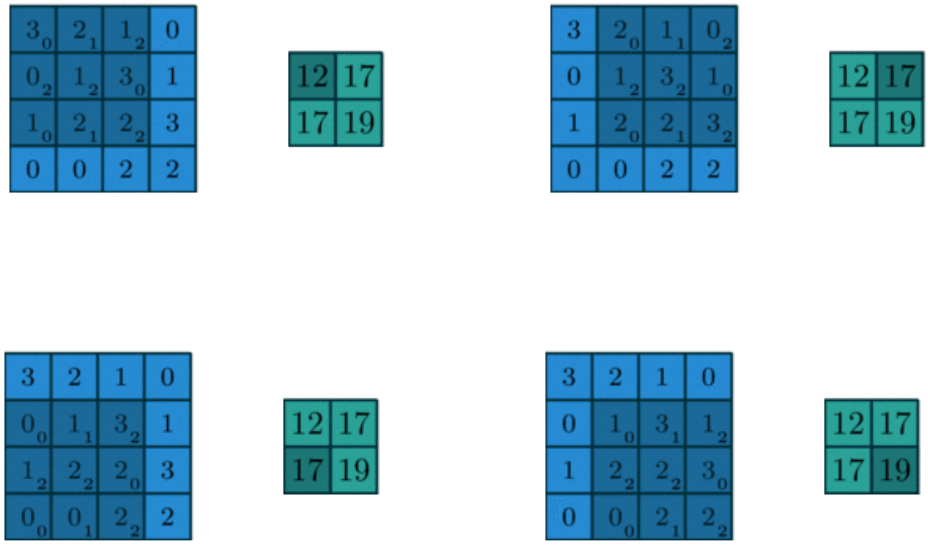


Figure 2.16 – Convolutional numerical layer: $W = H = 4$, $K = 3$

its receptive field into a single value, decreasing the complexity. However, especially when dealing with regression problems, in which we want to modify an input image remaining in the same information space, there can be situations in which we want to get as output matrices with the same shape of the input. To do so, the input image must be artificially increase with the so called *zero padding*: the image is filled with zeros at the beginning and at the end of the different axes. These added values are used when the kernel goes outside the image boundaries. The number of added values can differ depending on the axis. Non-zero padding is also possible, where different strategies are used to select the values to be added.
Figure 2.18 represents the corner operations of a convolutional layer that takes as input a 5×5 image with a zero padding of one pixel. The output image is now a 5×5 image and has indeed the same dimension of the input.

If we consider a square input image of dimension $D_i = H_i = W_i$ and symmetric padding P, strides S and kernel dimension K, we can compute the output dimension

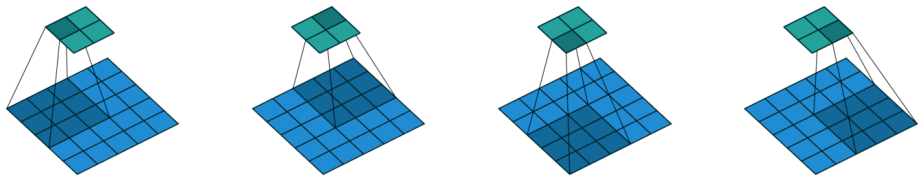


Figure 2.17 – Convolutional layer: $W = H = 5$, $K = 3$, $S = 2$

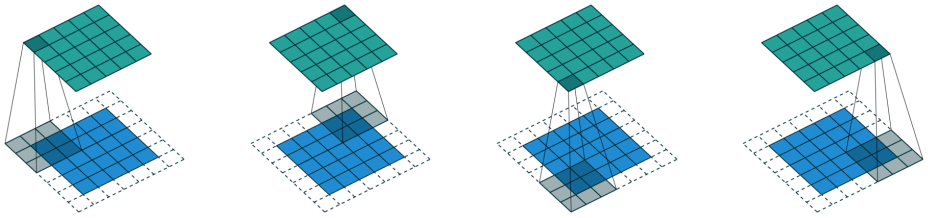


Figure 2.18 – Convolutional layer: $W = H = 5$, $K = 3$, with zero padding (corner operations)

D_o as:

$$D_o = \frac{D_i - K + 2P}{S} + 1 \tag{2.37}$$

2.4.2 CNN architecture

As said before, a convolutional layer can implement a lot of different kernels and outputs the correspondent pack of images, called *features maps*. Each kernel learned during the training phase represents indeed a feature and when it is shifted along the input image gives high outputs when it recognize that feature in the considered part of the image. The usual shape of the considered tensors is then $B \times H \times W \times F$, where B represents the batch dimension, H and W height and weight in pixels and F the feature maps. When another 2D convolutional layer is applied to a pack of F different feature maps, a number of kernels equal to F is considered and the results are then summed up in the output matrix. This means that having more channels in the input image means increasing the number of parameter of the 2D convolutional layer.

Typical CNN architectures has a lot of convolutional layers stacked one after the

other. In classification problems, the input space is usually quickly reduced by avoiding padding and adding average layers such as Max Pooling. After the convolutional part is then usually added some fully connected layers with a final softmax layer that interpret the features maps and classifies the image from them. In regression problems, the whole structure is usually entirely convolutional and padding is widely used to keep the same image dimensions.

Figure 2.19 represents the architecture of one of the first convolutional networks *LeNet-5* developed by Yann LeCun et al. in 1998 [16], that was used to automatically recognize handwritten digits on checks. The network presents two convolutional-pooling sequences, then followed by two fully connected layers and the output softmax layer with ten classes (one per possible digit).

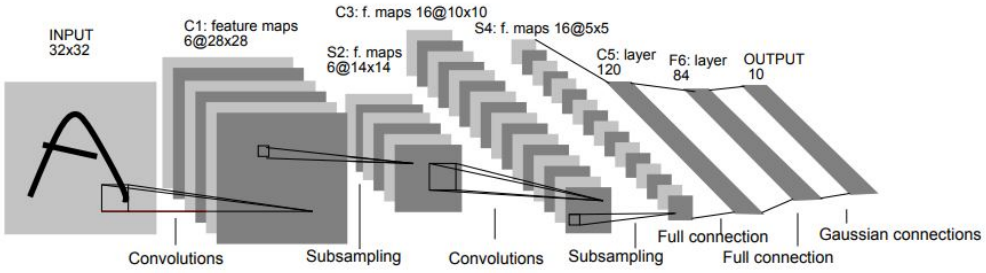


Figure 2.19 – LeNet-5 architecture as presented by Yann LeCun in 1998 [16].

2.4.3 3D convolutions

As said, when the input is a multidimensional image ($H \times W \times F$) a 2D convolutional operation considers one kernel per channel (last dimension) and then sums up all the results to get a 2D output matrix. The kernel dimension is then set to $K_1 \times K_2 \times F$ ($K = K_1 = K_2$ in case of a square kernel) and the convolution is done along all the channels at the same time. Another possible operation is the 3D convolution: in this case the designer set a three dimensional kernel with shape $K_1 \times K_2 \times K_3$ that is shifted along the 3D input tensor generating a 3D output. Usually $K_3 < F$, to generate a volume tensor; the case $K_3 = F$ coincides with the 2D convolution of a multi-channel input image and indeed outputs a 2D matrix. A 3D convolution can also

be used to manage a 3D image with more channels, so it can get as input 4D tensors ($H \times W \times F \times C$) similarly to 2D convolutions with 3D tensors.
Hyper-parameters like strides and padding are possible also for 3D convolutions, but this time they are expressed as lists of 3 numbers, specifying the desired value for each axis.

Chapter 3

SR State of the Art

Super-resolution (SR) are techniques that construct high resolution (HR) images from low resolution (LR) images. These techniques are popular in the field of image processing due to the importance of having good quality images for applications of different areas, such as medical imaging, satellite imaging and security imaging [17]. This chapter will present some basic concepts on *image resolution* and how it can be enhanced through SR; then some key Machine Learning approaches will be presented for both single-image super resolution (SISR) and multi-image super resolution (MISR).

3.1 General concepts

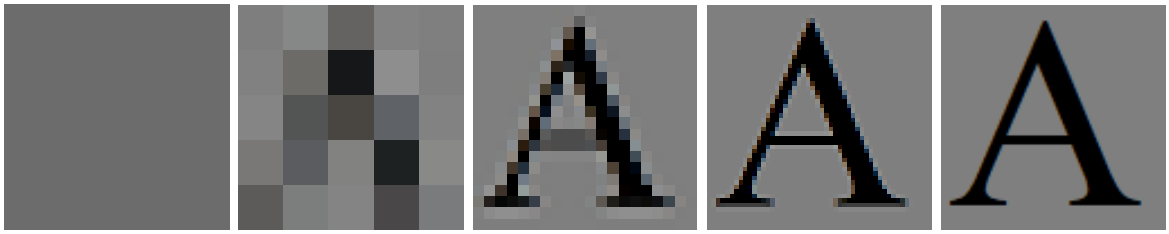
3.1.1 Image resolution

Resolution is a key concept for a digital image as it represents the quantity and the quality of information the image holds. There exist different definitions of image resolution, depending on the physic quantity we are focusing on, but each of them refers to the ability of the image to represent fine details.

Pixel resolution

It refers to the number of pixels used to cover the visual space captured by the image, usually expressed as column by row pixel dimensions $C \times R$. This is the common

meaning associated to the term *image resolution*, though it actually refers to the image size. *CIPA*[1] *DCG-001* guideline [19] explicitly states that "the term *resolution* shall not be used for the number of recorded pixel". Image 3.1 represents the same image in different pixel sizes. The number of available pixels limits the image quality: from that comes the misinterpretation of the term *resolution* as pixel count.



1×1 5×5 20×20 50×50 100×100

Figure 3.1 – Comparison between different pixel resolutions of the same image of the letter A

Spatial resolution

It refers to the details represented by the image with respect to space. The spatial resolution measured the smallest object that can be resolved by the sensor or the linear dimension on the ground represented by each pixel [20]. For satellite applications is often measured in *meters per pixel* and represents the amount of the original space contained in a single pixel.

It is important to underline that spatial resolution is limited by both the number of pixels used to represent the original space and the hardware ability to resolve different points. To account for the possible hardware limitation, the spatial resolution can be referred as the number of *independent* pixels used for unit length. That means that if 100 pixels are used to represent 100 meters, but they have all the same value, the actual spatial resolution is *100m per pixel*. The information is hold by a single pixel only, while the others are merely copies of it. Figure 3.2 represents an example of this kind. The first image has 256×256 pixels, while the second 1024×1024. Even though

[1] Camera & Imaging Products Association (CIPA) is an international industry association consisting of members engaged in the development, production or sale of imaging related devices including digital cameras.

(b) has a higher pixel resolution, its quality is worse since most of its pixels have the same value. We can then say that the spatial resolution of (a) is better.

(a) (b)

Figure 3.2 – Comparison between different spatial resolutions of the same image taken from the Set-5 dataset [21]. (b) has a higher pixel resolution but lower spatial resolution with respect to (a)

Spectral resolution

It describes the number and width of spectral bands in a sensor system [20]. It accounts in the frequency range of the light captured by the hardware system and it is fundamental to be considered when managing multi-bands images.

Temporal resolution

It refers to the details represented by a set of images with respect to time. It represents the number of scenes captured by a camera system in a given time period. For video signals, it is commonly measured in *frames per seconds* (fps).
For remote sensing, the temporal resolution is defined as a measure of the repeat cycle or frequency with which a sensor revisits the same part of the Earth's surface [20].

Radiometric resolution

It's also called *bit resolution*. It refers to the details represented by an image in terms of intensity value. It is determined by the number of bits used to represent the pixel values and therefore relates to the signal *quantization*. With N bits, the intensity level can range from 0 to $2^N - 1$. This range of possible values a pixel may assume is also referred as the *dynamic range* or *color depth*. Figure 3.3 represents an image in different radiometric resolutions. The ability of understanding an image content is strongly related to the number of bits used for it's representation.

1 bit 2 bits 4 bits 8 bits

Figure 3.3 – Comparison between different radiometric resolutions of the same image taken from the Set-5 dataset [21]. The image has 3 channels (RGB), thus the actual number of used bits is three times the radiometric resolution.

3.1.2 Super-resolution concepts

As said, super-resolution techniques aims at restoring HR details starting from LR input images and thus focus on trying to improve *spatial resolution*. For our applications we will work with images that always have independent pixels, so an improvement of spatial resolution will cause a correspondent increase in pixel resolution. For this reason, from now on when we will talk about HR images, we will refer to images that represent the same scene of LR but with more pixels and thus higher spatial resolution. As an example, the LR scenes of the Proba-V dataset (see chapter 4) have 300m per pixel spatial resolution and are stored in 128×128 pixels PNG files, while HR have 100m per pixel resolution and thus are 3 times bigger (384×384 pixels).
SR algorithms can be classified into two main approaches:

- **SISR**: *single-image super-resolution* tries to reconstruct a HR image from a single LR image. It is a ill-posed[2] problem because a specific low-resolution (LR) input can correspond to a crop of possible high-resolution (HR) images [17].

- **MISR**: *multi-image super-resolution* uses several images of the same scene and is based on data fusion techniques. The basic idea behind MISR is to exploit the non-redundant information that comes from the subpixel shifts between the different LR images. These shifts make theoretically possible to extract high frequency information, making the SR problem more constrained and then less ill-posed.

One of the key aspect to be mentioned when dealing with multi-image condition, is for sure *image registration*. This is the process in which the set of available images is transformed in a common coordinate system, so that corresponding pixels represent the same physical point. This is a very important step for fusing together different frames, since it ensure that the data manipulation is done coherently. Due to its importance, image registration is the first step in a lot of algorithms for MISR that rely on interpolation between the pixel values.

3.1.3 Super-resolution history

Classical approaches to SR, developed mainly for MISR in the last decades of the 20^{th} century, were divided in different categories:

- *non-uniform interpolation*: after image registration, a non-uniformly spaced sampling grid is obtained; then these points are interpolated and resampled on the HR pixels grid; finally a restoration algorithm is used for blur and noise removal

- *frequency domain methods*: relate the LR aliased discrete Fourier transform coefficients to the sampled continuous Fourier transform of the unknown HR image

[2] A problem is well-posed in the sense of Hadamar sense [26] if:

1. a solution exist
2. the solution is unique
3. the solution's behaviour depends continuously on the initial data

Problems that are not well-posed are defined as ill-posed.

- *regularization*: the ill-posedness of the SR problem can be limited by imposing additional constraints as a smoothness constraint, Bayesian stochastic a-priori probability density function
- *projection onto convex sets*: iterative approach that starts from an estimation of the HR image from an arbitrary initialization and then projects it to a consistency set derived from a priori constraints
- *generative methods*: use strong class based a priori assumptions that force the reconstruction of certain features in the HR image

SISR algorithms were then developed in later years as well analysed by [27], mainly with these kind of approaches:

- *prediction models*: generate HR images from LR inputs through a predefined mathematical formula without training data; interpolation methods such as bicubic interpolation belong to this category
- *edge based methods*: algorithms focused on edge features analysis and learning
- *image statistical methods*: use different image properties to build priors, such as heavy-tailed gradient distribution or sparsity property of large gradients
- *patch based methods*: learn mapping functions from cropped patches of paired LR and HR training images; examples of training methods are weighted average, kernel regression, support vector regression, sparse dictionary representation, Markov Random Fields.

The latter category already exploited Machine Learning algorithms to learn LR-HR mapping from a dataset. These approaches lead to the first Convolutional Neural Network method for SISR in 2015 [28]. After that date, CNN and Deep Learning has been the most used approach to asses super-resolution problems.

3.2 Deep Learning for SISR

This section will present the bicubic interpolation algorithm, considered the baseline method for super-resolve an image and then the most relevant deep learning architectures for single-image super-resolution that have been analysed for the development of this work. For performance comparison of the different models, the PSNR metric has been used. For a detailed explanation of the mathematical properties of this criterion, see section 4.3.

3.2.1 Bicubic interpolation

Bicubic interpolation is one the most used algorithms to upscale an image, and is considered as baseline reference to compare other algorithms results in the literature. That method is an extension of cubic interpolation splines for values on a 2D equally spaced grid.
The method considers a 16 pixels square (4×4), with indexes ranging from -1 to 2. The interpolated values can be written as:

$$f(x,y) = \sum_{i=-1}^{2} \sum_{j=-1}^{2} a_{ij} x^i y^j \quad , \tag{3.1}$$

where a_{ij} are coefficients to be determined imposing constraints on the values of f and its derivatives f_x, f_y and f_{xy}. The interpolation is made on the unit square $[0,1] \times [0,1]$ to add values in between the pixels in the corners point $(0,0), (0,1), (1,0), (1,1)$.
The constraints are written such that:

- $f(x,y)$ match the pixels values $p(x,y)$ in the data points $\rightarrow$ 4 equations
- the derivatives f_x, f_y and f_{xy} match the pixels values derivatives p_x, p_y and p_{xy} in the data points $\rightarrow$ 12 equations

To approximate the derivatives the pixel values outside the unit square are considered and that is why the algorithm works on a 4×4 pixels grid. As an example, to compute $p_x(0,0)$ we consider the slope between the two adjacent points on the x axis $p(-1,0)$ and $p(1,0)$:

$$p_x(0,0) = \frac{p(-1,0) - p(1,0)}{2} \tag{3.2}$$

In this way, all the derivatives can be approximated, then the system can be solved to obtain all the 16 coefficients a_{ij}. Depending on the rescale factor, new pixel values in the unit square can be found with equation 3.1 choosing appropriate values for x and y in the range $(0, 1)$. Figure 3.4 represents the bicubic upscaling with scale 4 of an image.

(a) LR (64×64) (b) bicubic upscaling (256×256)

Figure 3.4 – Low resolution (a) vs bicubic upscaling with scale 4 (b) of an image taken from the Set-5 dataset [21]

3.2.2 SRCNN

The first proposed Convolutional Neural Network architecture for super-resolution is the SRCNN (super resolution convolutional neural network) developed by Dong et al. [28]. The model has a very tiny architecture, with only 3 convolutional layers. As input image, SRCNN uses an upscaled version of the images through bicubic interpolation. The images are then fed to the network that transforms the pixel values to improve them. The three convolutional layers are:

1. 64 feature maps with 9×9 kernel
2. 32 feature maps with 5×5 kernel
3. output layer with 5×5 kernel

The first two layers are followed by ReLU activation, while the last one has no activation (linear). All the layers use zero padding to ensure that the pixel dimension of the

image remain the same. Training has been done with the MSE as loss function and the PSNR as metric. The network outperforms all the previous methods for SR in terms of obtained PSNR for both single channel and RGB images.
With some experiments, the authors came to the conclusion that adding more layers the performance drops and thus that "the deeper the better" doesn't not work for SR. However, successive works proved that it was just due to training problems, and that going deep still can help to improve the performance.
Figure 3.5 shows a schematic representation of the SRCNN architecture.

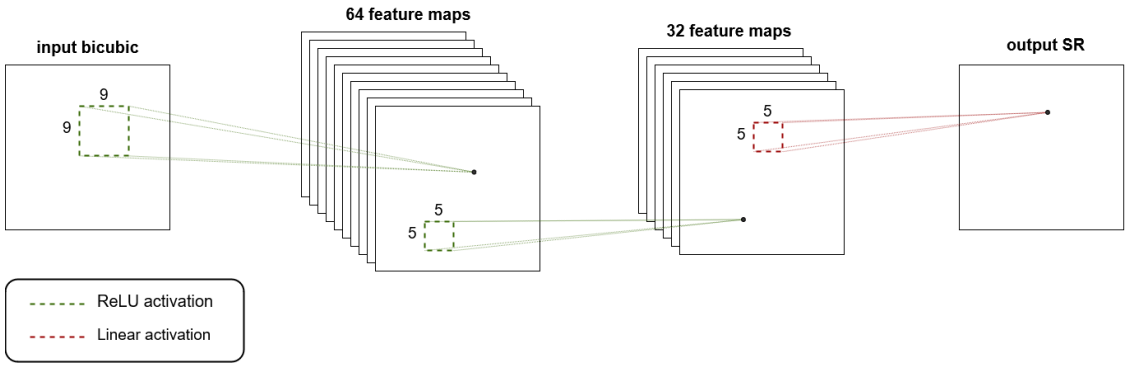


Figure 3.5 – Schematic representation of SRCNN architecture [28].

3.2.3 VDSR

VDSR (very deep super-resolution) [29] is the first very deep model used in SISR. It is a 20-layer network with 3 convolution kernels and with a *residual learning* architecture. With this type of architecture, the network doesn't learn the direct mapping from the bicubic input to the HR output, but it learns the residual, i.e. the difference, between the two. This means that the networks computes what should be added to the bicubic to obtain the HR image. The residual learning helps convergence and improves performance with respect to the direct mapping.

The authors trained VDSR with a MSE loss and mini-batch gradient descent with momentum as optimization algorithm. They used high learning rates to speed up the training procedure with gradient clipping to avoid diverging issues. The results obtained by the authors show that "the deeper the better" works for super-resolution, since they

outperform SRCNN. Figure 3.6 presents a schematic representation of VDSR. All the layers have ReLU activation and 64 filters with the exception of the last one before the residual adding, that has no activation and outputs a single image.

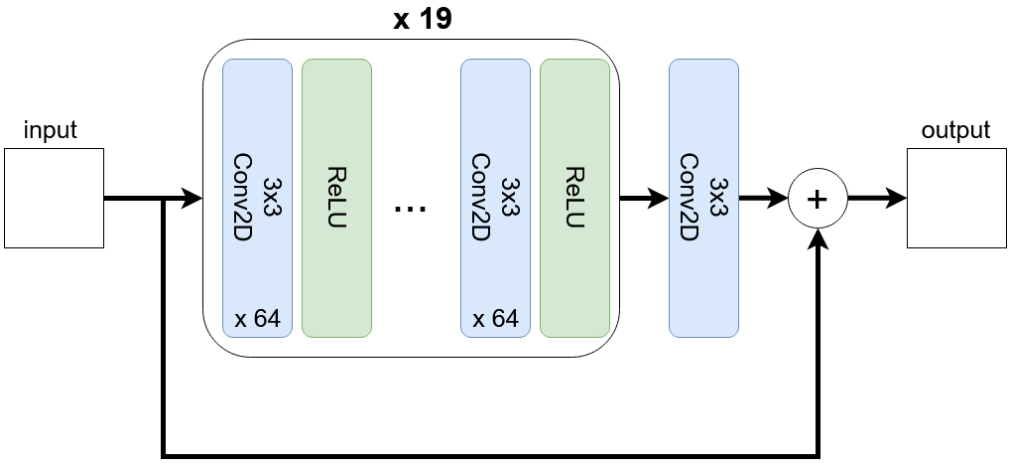


Figure 3.6 – Schematic representation of VDSR architecture [29].

3.2.4 DRRN

DRRN (deep recursive residual network) [30] instead of using a global residual branch, proposes multiple local residual blocks organized in a recursive fashion. Recursion for Deep Learning models stands for multiple blocks that shares the weights. In particular, the authors of DRRN proposes a network made of B recursive blocks, each made of a starting convolution and U residual units. A residual unit computes the local difference between the output of the starting convolution and the output of the previous residual unit passed through two convolutional layers made up of a batch normalization layer, a ReLU pre-activation and the actual convolution filter. Each filter in the different residual units shares the weights with the corresponding others, so that only two sets of weights are needed per recursive block. Figure 3.7 shows the DRRN architecture. The authors find out that the best values for the parameters are $B = 1$, $U = 25$, since it gets good results with fewer parameters, having only one recursive block with shared weights.

The training is done with mini-batch SGD and, as for VDSR, with a high initial learning rate and gradient clipping.

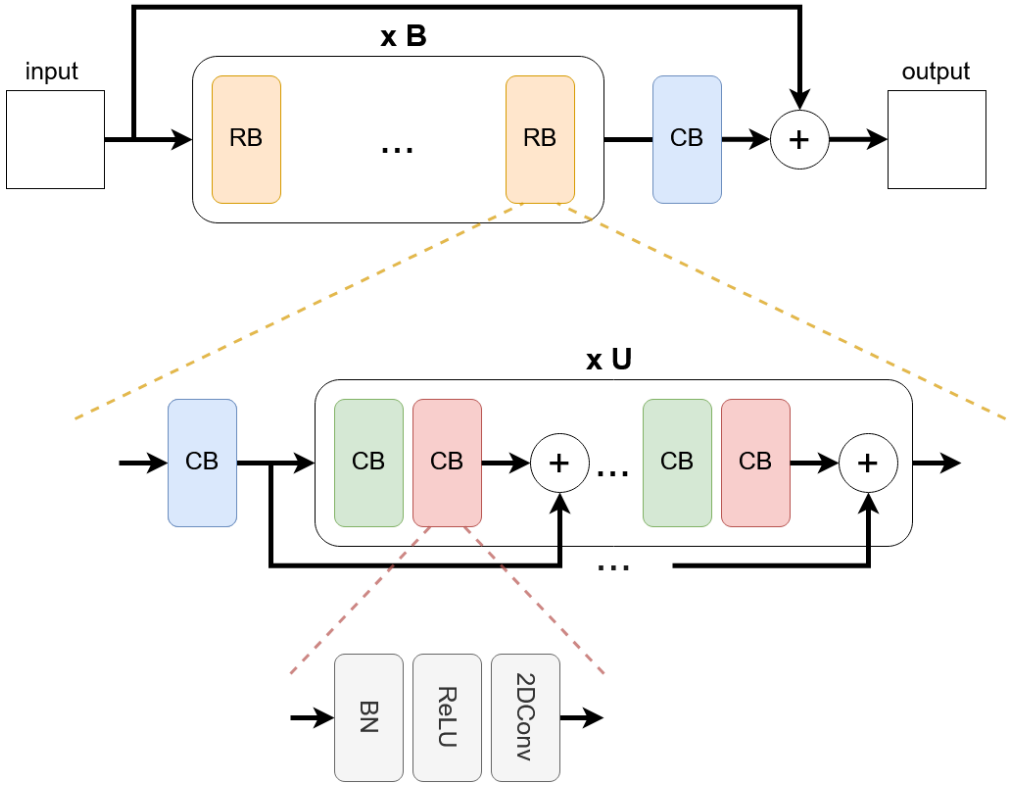


Figure 3.7 – Schematic representation of DRRN architecture [30]. Each Recursive Block (RB) is composed of U local residual units. Each Convolutional Block (CB) is composed of a Batch Normalization (BN) layer, a ReLU pre-activation and a 2D convolution. Weights are shared between green and red layers in the same recursive block.

3.2.5 EDSR

EDSR (enhanced deep residual network) [31] represents the current state of the art for SISR according to [17]. It follows the intuition of Ledig et al. with their SRResNet [32] to use an architecture similar to the ResNet [33], originally used for classification problems.

EDSR exploits both global and local residual learning. The local residual units are made of a lighter structure with respect to the original ResNet, with a simple chain of 2D convolution/ReLU/2D convolution and a scaling factor. The networks has 32 residual blocks and 256 feature maps per convolution. The multiplication factor is set to 0.1 and is used to stabilize the training procedure. The proposed architecture is really deep and uses a massive number of parameters, making it not suitable for low-end hardware.

A big innovation with respect to previously mentioned models, is that EDSR doesn't use the bicubic of the image as input, but uses directly the LR version, making all the feature extractions and manipulation at the lower resolution. The actual upsampling is only done in the last layers, after the adding of the residual to the original LR image. The upsampling is done using the subpixel convolutional layer proposed in [34]. This block consist in a convolutional layer that outputs s^2 channels, where s is the upsampling scaling factor; these channels are then reshaped in a single high resolution image, as shown in image 3.8. EDSR has a pre-upsample convolution that actually uses $s^2 \cdot 256$ filters. The output feature maps are then reshaped into 256 high resolution feature maps and then fed to a final convolutional layer that outputs the final SR image.

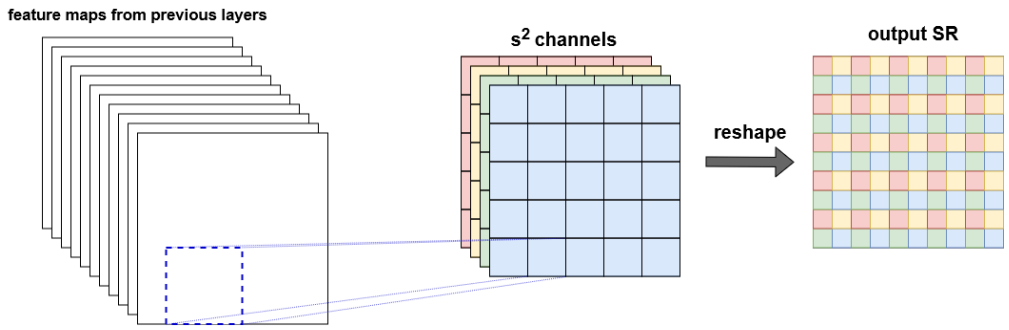


Figure 3.8 – The sub-pixel convolutional layer for upsampling [34]. In this case the scale factor is $s = 2$, thus 4 channels are used for the upsampling

Another feature of EDSR is the use of the geometric self-ensemble method proposed by [35]. This strategy is adopted after training, during the test phase. Each image is flipped and rotated in orer to generate 8 possible variations of it (LR_i), including the

original one.

$$\mathrm{LR}_i = T_i(LR) \quad , \quad i = 0, ..., 7 \tag{3.3}$$

All the obtained images are then passed through the network to obtain their super-resolved versions $\{\mathrm{SR}_0, ..., \mathrm{SR}_1\}$. The inverse transformations are then applied and the obtained images are averaged to get the final SR output.

$$\mathrm{SR} = \frac{1}{8} \sum_{i=0}^{7} T_i^{-1}(\mathrm{SR}_i) \tag{3.4}$$

The application of this method on average results in a boost of the network performance without adding parameters.

EDSR has been trained with Adam optimizer and mini-batches of 16 images. Instead of using MSE as loss, the authors chose the L1 loss or mean absolute error, stating that it provides better convergence. EDSR authors also proposed a multi-scale version of the network, that is able to provide several scaling factor at once, called MDSR.
Figure 3.9 shows the EDSR architecture.

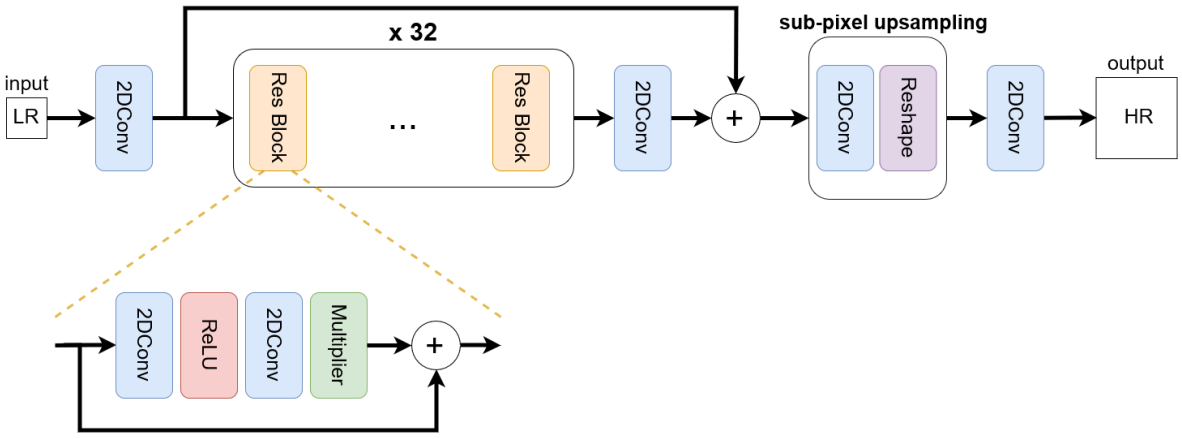


Figure 3.9 – Schematic representation of EDSR architecture [31].

3.2.6 SRDenseNet

Another popular network used for classification is the DenseNet [36]. In this architecture each layer is connected with all the preceding feature maps, allowing a strong feature exploration. Inspired from this architecture, another super-resolution model has been developed, called SRDenseNet [37]. This architecture uses several Dense Block (DB) and concatenates all the output features to finally reconstruct the output image. A DB has 8 convolution, each with 16 filters. All the feature maps are concatenated instead of being summed up, and this leads to $8 \cdot 16 = 128$ features maps per DB. The number 16 is called *growth rate* of the network.

The upsampling is done in the output with a *deconvolution layer*, also called *transposed convolution layer*. This layer can be seen as an inverse operation of a convolution that is able to have as output a bigger image starting from a smaller one. This type of operation is equivalent to a direct convolution with the same kernel dimension K and stride S, but $P = K - 1$ pad. The output dimension can be computed with equation 2.37:

$$D_o = \frac{D_i - K + 2P}{S} + 1$$

Figure 3.10 shows a transposed convolution that has a 2×2 input, $S = 1$, $K = 3$ and a zero-padding of the input equal to $P = 2$. Applying equation 2.37, we get an output dimension of 4.

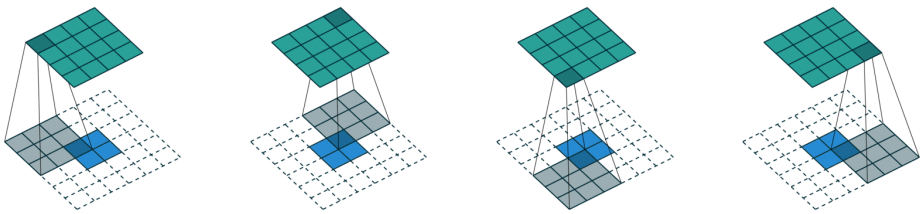


Figure 3.10 – Transposed convolutional layer: $D_i = 2$, $K = 3$, $S = 1$, with zero padding $P = 2$ (corner operations)

The deconvolution layer can be used in alternative to the sub-pixel convolution layer to upsample the image feature in the networks. However, since the deconvolution uses

a lot of padding zeros, that are artificially added to the image and are not coherent with the image content, it's preferable to use the sub-pixel method.

Figure 3.11 presents a schematic representation of the SRDenseNet architecture. All the output features of the different dense blocks are concatenated and fed to the final 1x1 convolution called *bottleneck layer* that decrease the complexity merging the information from the different features before the upsampling block composed of transposed convolution layers. All the convolutional layers are followed bu ReLU activation with the exception of the last reconstruction filter.

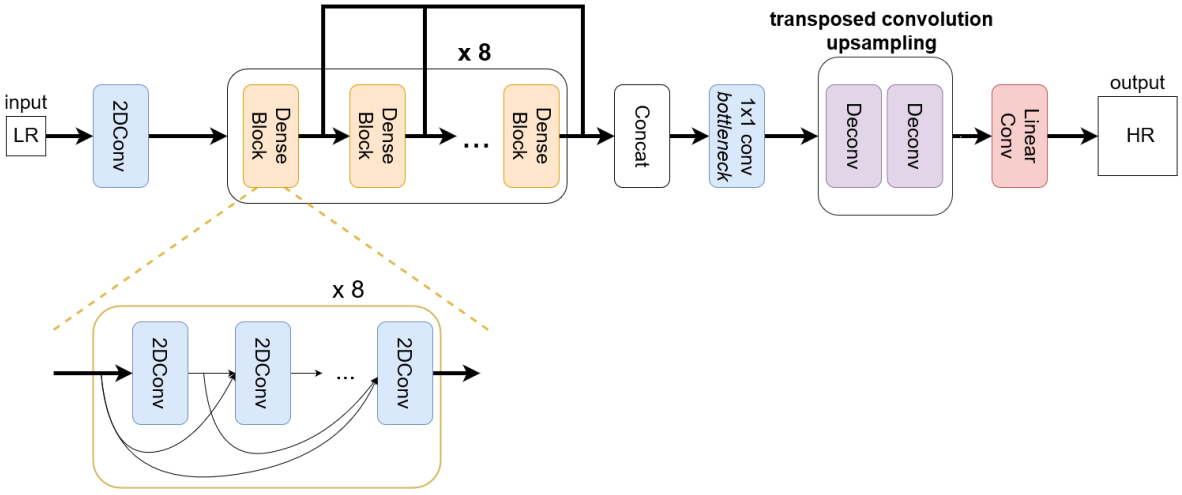


Figure 3.11 – Schematic representation of SRDenseNet architecture [37].

3.2.7 SISR models benchmark

Yang et al. [17] propose a review of the deep learning models for single-image super-resolution in which they analyse all the presented networks as well as other solutions to the SR problem. They found that there is a strong relation between model performances and the training dataset adopted. In particular, they identify two types of datasets:

- G200 [38] and Yang91 [39], characterized by, in total, 291 small images (on average 150×150)

- ImageNet [40] selections and DIV2K [41], characterized by much more numerous and large images

In general, models trained on the larger and higher-quality databases perform much better with respect to the others.
Another trend that Yang et al. find is that, generally, the performance improves as depth and the number of parameters grow. This definitely confirms that "the deeper the better" for deep learning based super-resolution, even if a strong limitation has to be underlined: the growth rate of performance decreases the more we increase the depth of the model. This means that adding parameters can help in the task, but also increase the model inefficiency, since the complexity increase is not proportional to the performance improvement. They also underline how a big challenge in the super-resolution future research will be the design of lighter and more efficient models with low performance degradation. This is particularly important since, although today adopted models can reach high accuracy levels, they are characterized by millions of parameters and are therefore difficult to be deployed and used for real world scenarios when the hardware and the operation timing have to be limited.

Table 3.1 shows PSNR results for x4 upscaling of different models as presented in Yang et al.'s review [17].

Model	**PSNR (x4)**	**Dataset**	**Parameters**
SRCNN [28]	30.49	ImageNet subset	57K
VDSR [29]	31.35	G200+Yang91	665K
DRRN [30]	31.68	G200+Yang91	297K
SRResNet [32]	32.05	ImageNet subset	1.5M
EDSR [31]	32.62	DIV2K	43M

Table 3.1 – Comparisons among different super-resolution models

How can be evinced by the table, EDSR currently represents the state of the art in term of PSNR results for single-image suer-resolution. However, the authors underline that MSE only has been used as optimization criterion for the benchmark evaluation, but it has been proved to be a poor criterion in some application, e.g. when the human

perception of the image quality is more important with respect to the pixels value adherence to the original.

Another big issue is the image degradation considered in the model. All the previous mentioned networks focus on bicubic degraded images. The authors show how EDSR performance drastically decreases when there's a degradation mismatch with respect to that of training.

3.3 Deep Learning for MISR

As said at the beginning of this chapter, multi-image super-resolution (MISR) uses several images of the same scene taken at different times and aims at building a high resolution version of the image exploiting the information coming from the intrinsic differences between the different frames, such as sub-pixel shifts. When the super-resolution problem was initially addressed, in the last decades of the 20^{th} century, algorithms were developed mainly for MISR, following different approaches, as mentioned in section 3.1.3. On the other hand, deep learning models, that have their focus on feature analysis of images, has been until now almost exclusively adopted in the context of SISR. The very few deep learning approaches to the problem will be briefly resumed in the following.

Restoration methods Both Li [42] and Wu [43] propose methods that use a deep learning network during a MISR process. However, analysing their algorithms, deep learning is actually used in a single-image fashion, since their convolutional networks are both preceded by other methods that first fuse the different images in a single one. The result of this first step is then fed to a CNN that takes care of further increase the quality of the output.

EvoNet EvoNet [44] is an algorithm that uses a CNN model inspired to SRResNet [32] to enhance multiple images. However, even in this case the actual super-resolution is done in a SISR way, since each image is independently enhanced by the network. The authors then use a fusion algorithm called EvoIM based on an optimization problem. Image registration is done as an intermediate step: the shifts are computed on the

original LR images, while the actual shift is done after the CNN. Later, the same authors proposed an evolution of the same architecture [45] that slightly changes the deep learning network, but always performing an independent process for each image.

Proba-V CNN The organizers of the Proba-V Challenge (see chapter 4) provided themselves a model [46] for the specific dataset after the competition end. Their paper has not been source of inspiration for the development of my model, since it was not yet available, but it has then been useful in understanding the obtained results and in writing this thesis. Their solution is strongly inspired by the SRCNN of Dong et al. [28] as well as by their revised version FSRCNN [47]. Their solution is actually a deep learning model for MISR, since it takes as input 5 LR images as different channels and perform a convolution on them together. The CNN has three convolutional layers followed by a transposed convolution that performs a x3 upscaling. The obtained HR feature maps are then averaged to obtain the output image. The CNN is trained with Adam optimizer, MSE loss and mini-batches of size 4. As metric they used a slightly modified version of the PSNR called cPSNR (see section 4.3). Their model performs on average better than the bicubic of the best available image and specifically they actually improve the condition of 251 out of 290 scenes (87%).

Chapter 4

Proba-V Challenge

Proba-V Super Resolution Challenge is a competition announced on the Kelvins website by the Advanced Concepts Team team of the ESA (European Space Agency) [48]. The purpose of the website is to promote space related competitions. Particular attention is devoted to Machine Learning in the resolutions of the problems.
The scope of the Proba-V challenge is to study strategies to super-resolve images of 78 Earth locations taken from the Proba-V satellite. The data was released in October 2018 and the competition ended on 1st June 2019. The official website presented the dataset and allowed submissions of results to be scored.

4.1 Proba-V satellite

The Proba-V is a satellite launched in May 2013 by the ESA. As stated on the official website [49], the 'V' stands for Vegetation, since that is the main focus of the satellite. It is designed to map land cover and vegetation growth across the entire globe every two day. It is equipped with instruments specifically thought for vegetation monitoring and produces images on different bands (blue, red, near-infrared and short wave infrared). The satellite is able to provide almost daily 300m per pixel resolution images, while roughly every 5 days 100m per pixel high resolution images. The goal of the challenge was to construct the HR images manipulating several LR images of the same scene, taken at different times. This process is called Multi-image Super-resolution (MISR)

and has already been applied before to satellite imagery.

As said, Proba-V provides images in different spectral bands. Among them, the two most important for vegetation monitoring are the RED and the NIR (near-infrared) bands due to their usage in computing the NDVI (normalized difference vegetation index). That is a widely used indicator for vegetation health that is computed as the normalized difference of the red and near-infrared reflectance measurements:

$$\mathrm{NDVI} = \frac{(\mathrm{NIR} - \mathrm{RED})}{(\mathrm{NIR} + \mathrm{RED})} \tag{4.1}$$

This formulation is justified by the fact that green plants absorbs solar light in the so-called photosynthetically active radiation (PAR) spectral region. The red band is part of the PAR region and is particularly photosynthetically active, thus live plants appear relatively dark in that band. On the contrary, leafs tend to reflect near-infrared radiation that is not useful for photosynthesis, appearing bright in that band. Thus, for healthy plants the $^{\mathrm{NIR}}/_{\mathrm{RED}}$ ratio is expected to be big. By contrast, clouds, snow and urban areas tend to be bright in the red radiation and dark in NIR, and thus have small $^{\mathrm{NIR}}/_{\mathrm{RED}}$ ratio. The NDVI is nothing more than a normalization of the $^{\mathrm{NIR}}/_{\mathrm{RED}}$ ratio, since it tends to 1 when the ratio is big and to -1 when the it is small.

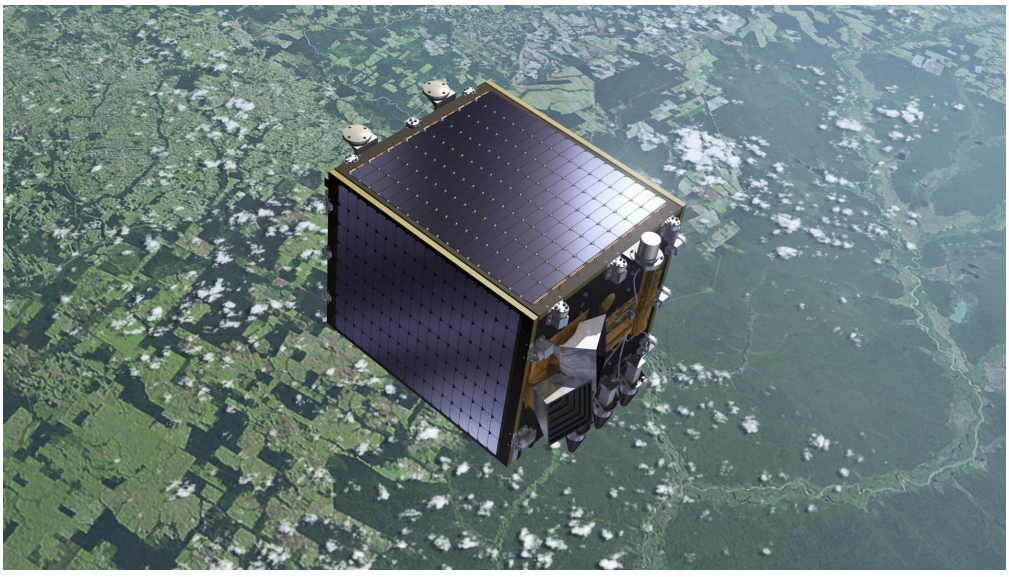
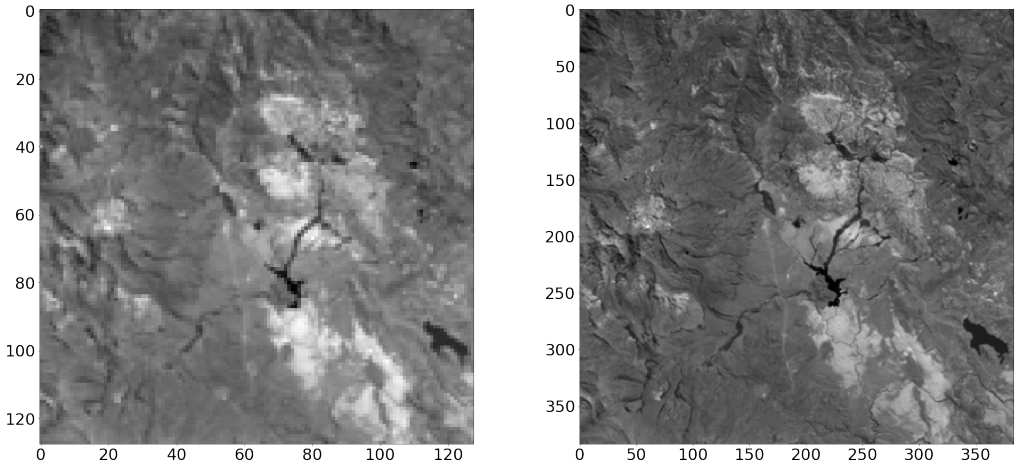


Figure 4.1 – Artist's view of the Proba-V satellite as presented on the ESA website [49]

4.2 Dataset

Due to the importance of the NVDI, the RED and NIR bands have been selected to be part of the challenge dataset. As said, the satellite provides frequent (almost daily) 300 m resolution images (defined from now on as low resolution LR) and roughly every 5 days 100 m resolution images (defined as high resolution HR). The dataset is split into two main parts: the train and the test parts. The first provides both LR and HR images, while the second provides LR only and the challengers are supposed to submit their super-resolved version of the HR. For both the parts, RED and NIR scenes are present.

The images are provided as 128x128 pixels greyscale patches for LR and 384x384 pixels for HR, thus giving a scaling factor 3. Given the satellite resolution, each patch represents a land surface of $(0.3\ {}^{\mathrm{km}}/_{\mathrm{pixel}} * 128\ \mathrm{pixels})^2 \simeq 1475\ \mathrm{km}^2$. Figure 4.2 presents two images of the same region in the two resolutions. It is important to underline that the a time window of 30 days has been used to select the images, so differences due to cloud coverage and small land changes are possible.



(a) LR (b) HR

Figure 4.2 – LR and HR images from the RED band

The creators of the dataset manually selected 74 regions of interest (ROIs) guided by analyses of annual global cloud coverage, in order to minimize the pixels that represent clouds instead of actual land. Figure 4.3 shows a global map with the 74 ROIs marked

in red. Each ROI, consist in a 16 patches size region (approximately 23600 km_2) in the two spectral bands (RED and NIR), leading to 2368 possible scenes.

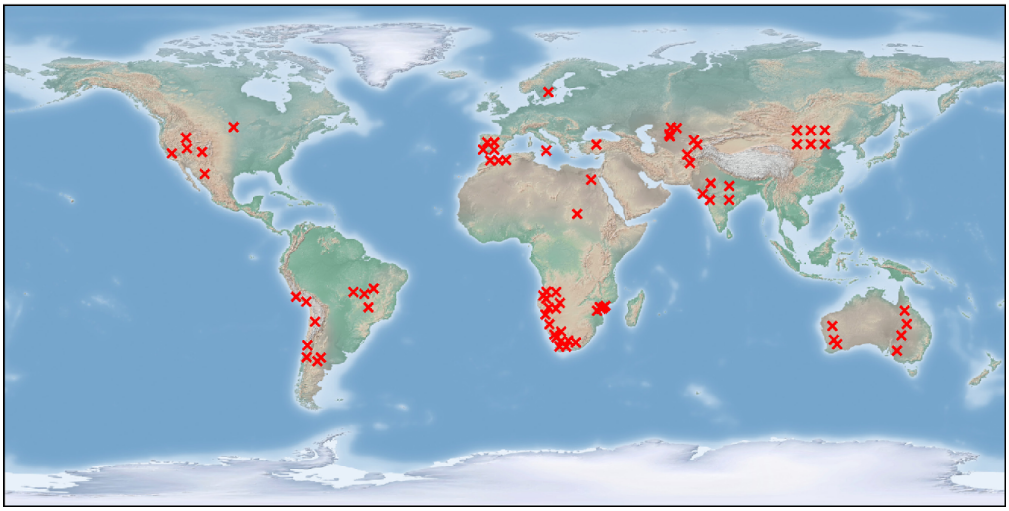
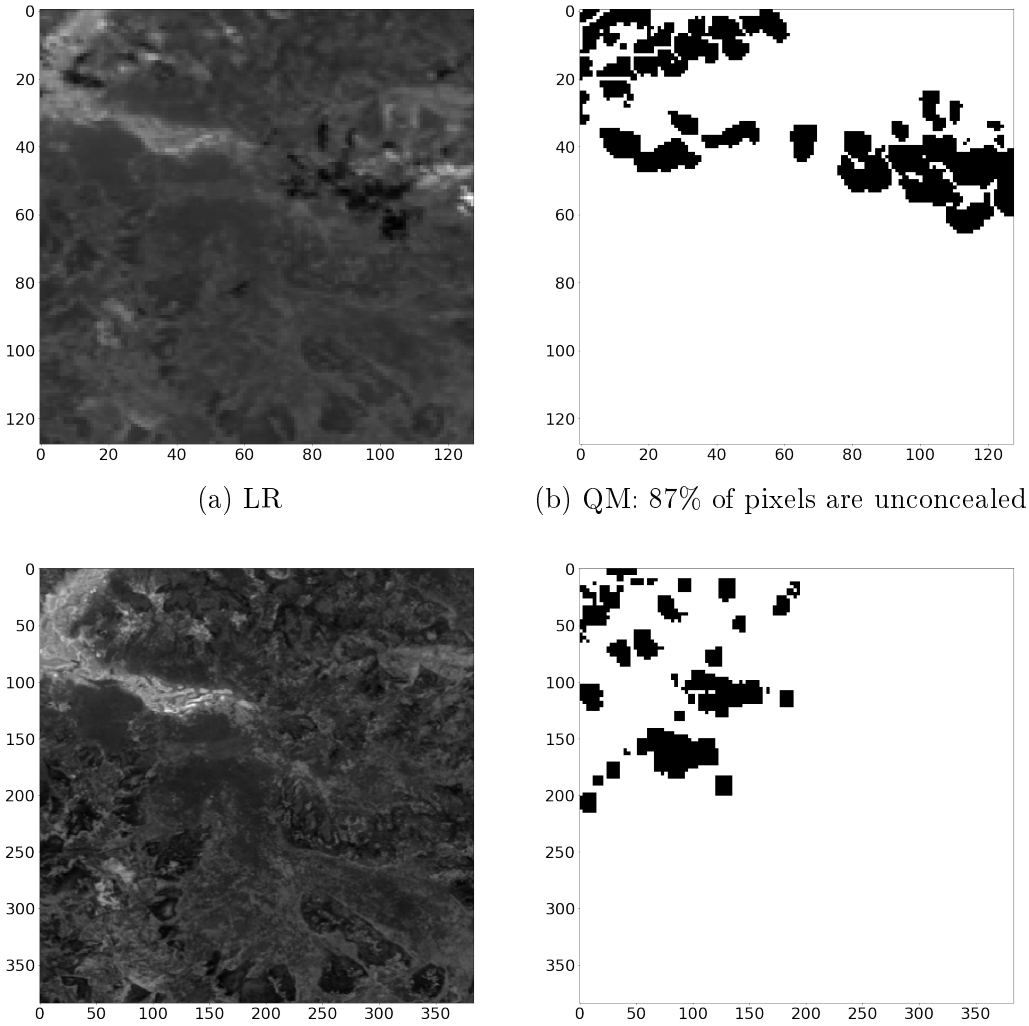


Figure 4.3 – Map of the dataset ROIs manually selected around the entire globe as presented in the challenge organizers' paper [46]

Out of the total 2368 scenes, quality selection based on cloud covered pixels has been made: LR images are accepted only if more than 60% of pixels are unconcealed, while HR images need at least 75%. The authors explain that this different threshold is justified by the fact that each scene would present multiple LR images, meaning that a particular bad pixel can be good in another image, while for HR only one image is provided. At the end of this selection, 1450 scenes were included in the dataset, about 61% of the total. Each scene includes at least 9 and up to 30 LR images. Binary maps are also provided for both LR and HR images, that mark with a 1 good pixels and with a 0 cloud covered pixels. These maps will be denoted from now on as quality maps (QM) for LR images and status maps (SM) for HR. Image 4.4 shows a LR image with its QM and the correspondent HR image with the SM. As can be seen, cloud coverage varies a lot between images of the same scene, since they refer to different times.

For what concerns bit-depth, all the images are generated with reflectance values ranging on 14 bits, but they are stored in a 16-bit png file. This makes them look relatively dark if directly opened with image viewers programs. Thus, only when visu-

alized, the white/black scale has been automatically rescaled with *matplotlib* package to match to the images minimum and maximum value, so that the contrast is maximized. Figure 4.5 shows the same image with and without rescaling.



(a) LR

(b) QM: 87% of pixels are unconcealed

(c) HR

(d) SM: 94% of pixels are unconcealed

Figure 4.4 – Quality and status maps of LR and HR images (NIR band)

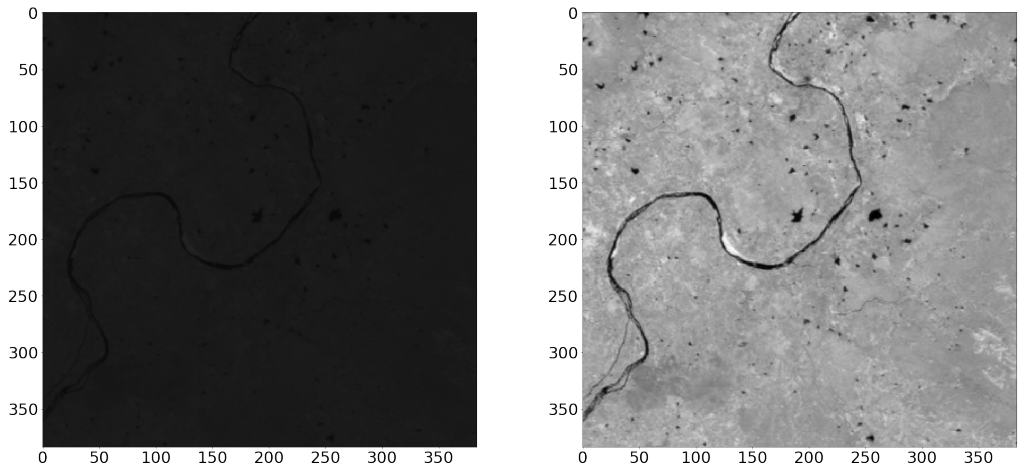


Figure 4.5 – The same image as visualized with the full 16-bit scale and with the scale reduced to its min/max values.

Out of the 1540 total scenes, 1160 (594 for RED and 566 for NIR) are for the train part, while the remaining 290 (146 for RED and 144 for NIR) are for the test part.

Each train scene consists in:

- 9 to 30 LR 16-bit images
- 9 to 30 LR binary quality maps (QM)
- 1 HR 16-bit image
- 1 HR binary status map (SM)

Each test scene consists in:

- 9 to 30 LR 16-bit images
- 9 to 30 LR binary quality maps (QM)
- 1 HR binary status map (SM)

The missing HR image has to be constructed and submitted as part of the challenge.

4.3 Scoring methods

4.3.1 PSNR

The most used metric for image quality is the *peak signal-to-noise ratio*, denoted as PSNR. It is defined as the ratio between the maximum power of a signal and the power of noise that affects it, expressed in decibel. Having a target image (the real HR image) and its constructed version (the super-resolved one), it's possible to compute the PSNR, starting from the MSE (mean squared error).
Starting from the definition of equation 2.14, we can compute the MSE between a target image HR and its super-resolved approximation SR, both of size $H \times W$ pixels, as:

$$\text{MSE} = \frac{1}{H \cdot W} \sum_{i=1}^{H} \sum_{j=1}^{W} (\text{HR}(i,j) - \text{SR}(i,j))^2 \tag{4.2}$$

The PSNR (in dB) is then computed as:

$$\text{PSNR} = 10 \cdot \log_{10} \left(\frac{I_{max}^2}{\text{MSE}} \right) \quad , \tag{4.3}$$

where I_{max} is the maximum value of intensity for a pixel. In the case of a digital image represented on N bits, I_{max} is usually equal to $2^N - 1$. The term I_{max} squared represents the maximum power of the signal, that is compared with the error power (the MSE). When the error is so large to be compared with the maximum signal power I_{max}, the PSNR tends to 0, since the ratio tends to 1; on the contrary, when the error tends to be null, the ratio and the PSNR tends to infinity. It's important to underline that the PSNR is a good index to measure the similarity between two images in their pixels intensity values and not for what concerns human perception. Other metrics perform better on measuring how similar two images appear to a person, but since the aim of the challenge is to try to get the *exact* HR image and not one that *looks like* that, the PSNR is the most suitable choice.

4.3.2 Bias corrected PSNR

One problem of the classic formulation of PSNR is that it doesn't take in consideration a possible bias in the pixels intensities. The problem can be easily shown with the following example. Suppose to have a perfect representation of the target HR with only an added constant value K to each of the pixels so that $\mathrm{SR}_1 = \mathrm{HR} + K$ and a second super-resolved image that is the superposition of the HR image and a Gaussian noise with 0 mean so that $\mathrm{SR}_2 = \mathrm{HR} + n$.
Gaussian noise has a normal distribution as probability density function p_n:

$$p_n(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(z-\mu)^2}{2\sigma^2}} \quad ,$$

where z is the noise intensity level, μ is the noise mean value and σ the standard deviation. Figure 4.6 shows the generation of SR_2 adding a noise with zero mean and 0.03 standard deviation.

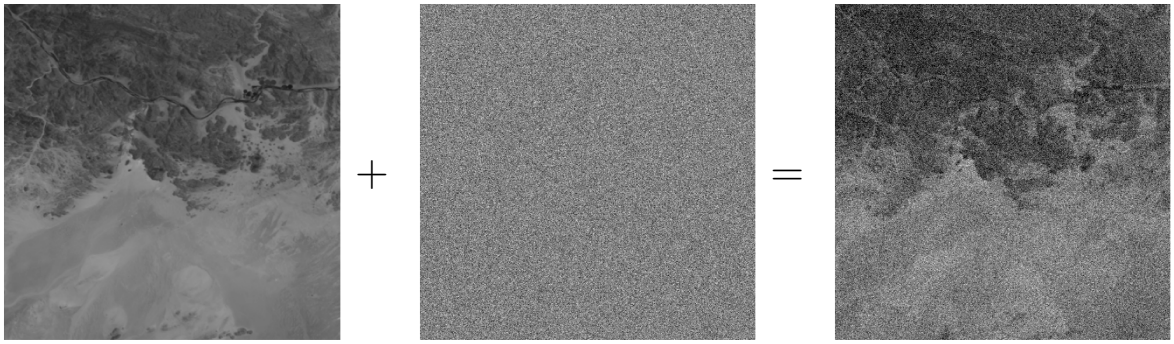
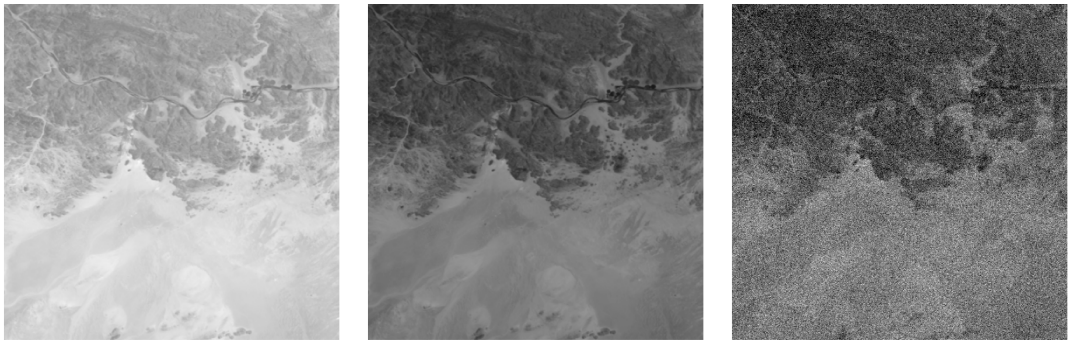


Figure 4.6 – Generation of a noisy image with added Gaussian noise ($\mu = 0$, $\sigma = 0.03$)

If we compare the two images with the original HR, as in figure 4.7 , SR_2 has a really degraded content, while SR_1 is simply more bright, but with all the details intact. However if we compare the PSNR computed as in equation 4.3, we get better results for SR_2 instead of SR_1. To take account of this problem, we can compute a bias component b between two images as follows:

$$b = \frac{1}{H \cdot W} \sum_{i=1}^{H} \sum_{j=1}^{W} (\mathrm{HR}(i,j) - \mathrm{SR}(i,j)) \tag{4.4}$$



(a) SR_1 (b) HR (c) SR_2

Figure 4.7 – Comparison between a biased image SR_1, the original HR and a noisy version SR_2. Parameters : $K = 0.1$, $\mu = 0$, $\sigma = 0.03$

This value, is nothing more than the mean absolute error between the pixel values of the two images. A slightly modified version of the MSE can be defined :

$$\text{bMSE} = \frac{1}{H \cdot W} \sum_{i=1}^{H} \sum_{j=1}^{W} (\text{HR}(i,j) - (\text{SR}(i,j) + b))^2 \tag{4.5}$$

Now we can compute the PSNR with bias correction as in equation 4.3, using the just computed bMSE instead of the classic mean squared error.
Table 4.1 shows the PSNR with and without bias correction on the images of the example. As can be seen, the correction is negligible for SR_2, since its pixel mean value is almost the same of the original (due to the zero mean of the Gaussian noise); on the contrary, the PSNR changes drastically for SR_1 and this demonstrates the importance of doing such a correction.

Image	**HR mean value**	**Image mean value**	**PSNR**	**corrected PSNR**
SR_1	0.1158	0.2158	19.99	322.72
SR_2	0.1158	0.1157	30.46	30.46

Table 4.1 – Effect of bias corrected PSNR

4.3.3 cPSNR

Due to specific characteristics of the available data, the competition rules present the bias corrected PSNR, but with a further modification. Since some HR pixels are concealed and not meaningful, it's useless to score the similarity of the correspondent SR pixels. Furthermore, instead of considering the entire 384x384 pixels HR image, a cropped part is used to take account of possible pixel shifts between the SR image and the target HR. Pixel shift is a big problem when dealing on satellite multiple shots of the same region: even the most accurate source can always have some sub-pixel shift between images took at different times or in different channels. The process of trying to realign multiple measurements, transforming them in a common coordinate system, is called *image registration*. There exist numbers of classical and machine learning inspired algorithms to deal with this problem, but the challenge authors [46] decided to use this simple criteria that takes into account discrete pixel shift in the HR domain and cloud coverage:

- crop the SR image of 3 pixels at each border, obtaining a 378x378 patch
- consider all the possible 378x378 patches from the HR target: for all $u, v \in 0, ..., 6$, $\mathrm{HR}_{u,v}$ is the subimage of HR with its upper left corner at coordinates (u, v) and its lower right corner at $(378 + u, 378 + v)$
- extract the correspondent subimages $\mathrm{SM}_{u,v}$ from the status map SM
- compute bias correction $b_{u,v}$ and $\mathrm{bMSE}_{u,v}$, considering the clear pixels only (those with a 1 in the SM)
- proceed in the computation of the bias corrected PSNR for each (u, v) couple

The final metric, called cPSNR, is then computed as the maximum between all the different $\mathrm{PSNR}_{u,v}$.
Denoting as $\mathrm{clear}(x)$ the set of clear pixels for image x, the following equations are used to compute the cPSNR:

$$b_{u,v} = \frac{1}{|\,\mathrm{clear}(\mathrm{HR}_{u,v})\,|} \sum_{\{i,j\} \in\, \mathrm{clear}(\mathrm{HR}_{u,v})} (\mathrm{HR}_{u,v}(i,j) - \mathrm{SR}(i,j)) \tag{4.6}$$

$$\mathrm{bMSE}_{u,v} = \frac{1}{|\,\mathrm{clear}(\mathrm{HR}_{u,v})\,|} \sum_{\{i,j\}\in\,\mathrm{clear}(\mathrm{HR}_{u,v})} \left(\mathrm{HR}_{u,v}(i,j) - (\mathrm{SR}(i,j)+b)\right)^2 \tag{4.7}$$

$$\mathrm{PSNR}_{u,v} = 10\cdot\log_{10}\left(\frac{I_{max}^2}{\mathrm{bMSE}_{u,v}}\right) \tag{4.8}$$

$$\mathrm{cPSNR} = \max_{u,v\in\{0,\ldots,6\}}\left(\mathrm{PSNR}_{u,v}\right) \tag{4.9}$$

4.3.4 Submission score

Given the different scenes present in the dataset, on average some output will have higher PSNR than others, even if generated with the same model. Since the challenge objective is to submit a super-resolved version of all the 290 test images, to normalization each scene contribution a file containing the cPSNR of the baseline solution is part of the download folder.
After having computed the cPSNR of all the images, the normalization values are used as reference to compute a score z_i for each scene:

$$z_i = \frac{N_i}{\mathrm{cPSNR}_i} \quad , \tag{4.10}$$

where N_i is the normalized value (cPSNR of the baseline solution) of the particular image. A certain image receives a score below 1 if the solution submitted has higher cPSNR than the baseline. The overall score of a submission is then the average of the individual scores.

$$Z(\mathrm{submission}) = \frac{1}{290}\sum_{i=1}^{290} z_i \tag{4.11}$$

A submission with score below 1 performs better on average than the baseline. Thus, the objective of the challenge is to get the lowest score possible.

4.3.5 Baseline solution

As said, the baseline solution is considered the starting point for super-resolution and indeed uses a very simple algorithm. It is based on the bicubic interpolation of the best images for each scene. As presented in section 3.2.1, the bicubic interpolation can be seen as the easiest algorithm for image resampling and thus for super-resolution.
The baseline algorithm is the following:

```
for each test image do
    compute LR clearances;
    create a subset with LR images with maximum clearance;
    for each image of the subset do
        compute the bicubic upscaling of LR;
    end
    SR = average of the upscaled images;
end
```

In this way the best LR images (those with highest percentage of clear pixels) are upscaled and averaged to get the SR approximation of the target image. The cPSNR of the solutions generated with this algorithm are stored for normalization in a dataset file called *norm.csv*.

Chapter 5

Work Platform

In this chapter the experimental platform is presented with a brief explanation of the chosen programming language and libraries as well as the hardware setup.

5.1 Software setup

A lot of programming languages are used for Machine Learning development. A non exhaustive list of possible choice is the following:

- *Python*: the most used one due to its flexibility and library richness
- *R*: used principally for statistical analysis and data manipulation
- *Matlab*: used with its Deep Learning Toolbox
- *C/C++*: widely used due to its execution efficiency

Python was chosen for the development of this work mainly due to previous experience with it and since it is the most used programming language for Machine Learning didactics. The development was made with the 3.5.2 release of the programming language and using the following libraries:

- NumPy [50]: numerical computation and data manipulation
- Matplotlib [51]: image and graph visualization

- Scikit-Learn [52]: data analysis
- OpenCV [53] and Scikit-Image [54]: image processing algorithms
- Jupyter Notebook [55]: development platform
- TensorFlow [56]: Machine Learning library
- Keras [57]: high level library for TensorFlow algorithms implementation

All the code was written in Python with the development platform Jupyter Notebook, a web-based interactive computational environment that allows to create documents with code that can be modularly executed and markdown comments for documentation. A notebook can be visualized and used simply with an internet browser and therefore is completely cross-platform and allows remote access to the work stored in the main machine. Jupyter Notebook supports different kernels (Python, R, C, C++ and others) to interpret the code and allows to work in virtual environments with different libraries installed.

5.1.1 TensorFlow and Keras

TensorFlow, developed by the Google Brain team, is one of the most used libraries for Machine Learning and provides APIs for Python, C/C++, Java and other programming languages. The library implements modules, classes and functions specifically designed to easily build, train and test machine learning models. The present work was developed using TensorFlow as Keras *backend*.
Keras is a model-level library that provides high-level building blocks for developing deep learning models and relies on low-level software, called backend, to perform the actual computations and tensor manipulations. Three backends are available for Keras: TensorFlow by Google, Theano by LISA Lab at Université de Montréal and CNTK by Microsoft. Keras and TensorFlow are perfectly integrated, allowing to use backend code for complex operations for which there is no a Keras implementation.

5.2 Hardware Setup

The hardware setup for the development of the thesis was provided by the PIC4SeR Centre at Politecnico di Torino. Since the specific problem was quite resource demanding, having to manipulate thousands of images, all the work was developed on a dedicated computer with Ubuntu 16.04 operating system. The machine mounted an i7-9700K Intel CPU, 32 GB of RAM memory and 32 GB of swap memory. The most important hardware components of the system were the available GPUs:

- one NVIDIA GeForce RTX 2080 Ti with 11 GB GDDR6 memory
- one NVIDIA GeForce RTX 2080 with 8 GB GDDR6 memory

Having a workstation with powerful GPUs available is crucial for Machine Learning projects, since computations are much more fast and efficient if done on specific high level chips as graphic cards are. In particular, Nvidia provides the *Compute Unified Device Architecture* (CUDA) to allow the so called GPGPU (General-Purpose computing on Graphics Processing Units) on its top-of-the-line products. The CUDA platform can be installed on a system providing direct access to the GPU virtual instruction set, allowing to treat the graphic card basically as a general purpose CPU. TensorFlow provides a specific library to support CUDA and GPU computations called *tensorflow-cpu*. If a system is provided with an updated version of CUDA and the TensorFlow library, Keras can be used on the GPUs as well, allowing very efficient computations.
Despite the very effective hardware provided, some limitations were experienced during the thesis development. The amount of RAM memory available caused some training problems:

- the number of channels (different time frames) per each image was limited to 9, even if some scene have up to 30 different channels
- the data augmentation (rotations and flip) couldn't be performed off-line, but it was executed during the training phase slowing down the execution
- all the CPU related operations were slightly slowed down due to the usage of the swap memory, that is stored on the hard drive, when RAM memory saturated

The available GPUs had good performance for computation but were limited in memory, too. The only 11 GB available on the best graphic card imposed the following restrictions:

- the number of layers for the model was limited to avoid memory overflow when instantiating all the needed tensors
- for the same reason, the number of feature maps generated in the convolutional layers were limited too
- the batch size used during training was strongly reduced due to memory overflow errors, leading to the extreme case of a stochastic condition (unitary batch size) for some models

Chapter 6

Proposed Model and Results

This chapter will briefly describe the main steps that led to the design of the model and will present the obtained results.

6.1 Pyhton framework

The first step in the model development has been the coding of a suitable python framework. The library, called `SRNet`, provides all the needed functions to easily manage the dataset and perform model evaluation and submission. `SRNet` is organized as follows:

- `download_dataset.py` allows to automatically download and extract the dataset from the ESA website
- `preprocessing.py` implements functions to select images from the dataset and open them as numpy arrays; the file also defines the `ProbaVDataSet` class, that automatically read the dataset folders to create an object that stores the images
- `submit.py` creates the submission file with the super-resolved test images that will be uploaded to the competition website

- `transformation.py` implements useful image processing functions to flip, rotate, upscale (with a bicubic interpolation) the images; it also allow to compute the pixel average between different images or create patches of a certain size
- `visualization.py` provides functions to easily show images from the dataset
- `scoring.py` implements functions to score the SR images accordingly to the scoring method exploited by the challenge organizers (see section 4.3)
- `training.py` provides implementation of the ESA loss, as well as functions for patch generation and the geometric self-ensemble method (see section 3.2.5)
- `models.py` is the file in which all the tested models have been implemented

6.2 Data preprocessing

In order to perform a model training, the available dataset has to be imported and preprocessed. In this section, all the preprocessing steps will be analyzed.

Data selection

As mentioned in section 4.2, the dataset is composed of 1160 training scenes and 290 test scenes. A single model has been designed for both NIR and RED bands, so all the images are treated in the same way, independently on their spectral property. Since the number of available LR images per scene is variable, we should choice the number of image to be used as channel width. To avoid selecting too poor quality images, a criterion based on the clear pixels is used. For each scene, the clear pixel of the high resolution image is computed from the sum of the pixel values of the SM (status map). A LR image is good if the number of its clear pixels (computed from the quality map QM) is as close as possible, in percentage, to the clearance of the HR image. Having a higher percentage of clear pixels with respect to the HR is good, but not required, since it will be useless to reconstruct pixels that are concealed in the target image. For

this reason, a threshold *thr* relative to the clearance of the HR image is used, leading the following criterion:

$$\begin{cases} \text{mean}(\text{QM}_i) < thr \cdot \text{mean}(\text{SM}) & \Rightarrow \text{LR}_i \text{ excluded} \\ \text{mean}(\text{QM}_i) \geq thr \cdot \text{mean}(\text{SM}) & \Rightarrow \text{LR}_i \text{ accepted} \end{cases} \tag{6.1}$$

Several values for *thr* have been tested and the results are proposed in table 6.1

thr	**Dataset section**	**Admitted**	**Excluded**	**Mean clearance**
0.8	train	20116	2178 (9.77%)	0.9565
	test	5062	440 (8.00%)	0.9561
0.9	train	18342	3952 (17.73%)	0.9705
	test	4628	874 (15.89%)	0.9703
0.95	train	16923	5371 (24.09%)	0.9773
	test	4225	1277 (23.21%)	0.9781
0.98	train	15384	6910 (30.99%)	0.9815
	test	3835	1667 (30.30%)	0.9825
1	train	12750	9544 (42.81%)	0.9830
	test	3285	2220 (40.35%)	0.9838

Table 6.1 – Data selection whit different *thr*

As definitive value, $thr = 0.98$ is selected, to avoid excluding a too large part of the dataset, and at the same time guarantee a mean clearance over 0.98. The mean number of images per scene, after the exclusion of the worst images, is about 13. Since the width should be constant for all the scenes, the value of 9 is selected, that is a good trade-off between model complexity and dataset richness. For all the scenes in which more than 9 LR images are available, the best 9 are selected; for those in which less than 9 are available, the best image is replicated to fill the empty spaces.

Data rescaling

As mentioned in section 4.2, the images are provided as 16-bit png files, so their pixel values range between 0 and $2^{16} - 1 = 65535$. In fact, the original images are represented over 14 bits, so the real intensity range is $[0, 2^{14} - 1]$. When loaded in memory, the images are automatically converted in a float numpy array and rescaled so that they

assume values between 0 and 1.

Data augmentation

Data augmentation is a well used technique used to both artificially enlarge the dataset and increase the model generalization. Data augmentation consists in applying some image transformations to the existing dataset, in order to generate new training instances that should be realistic and coherent with the real data. This technique, in addition to increase the number of data points, can really help the generalization of the model, since can avoid the condition in which the network simply memorizes the existing data points (overfitting) by slightly changing them.

Typical image transformations used for data augmentation are rotations, flip, resizing, crop, shifts. In the proposed model data augmentation has been performed with rotations of angles multiple of 90° and horizontal and vertical flips. The limitation to these two transformations is due to the specific characteristic of the super-resolution problem: the actual value of the pixel is fundamental to correctly learn the LR/HR relation, so false void pixel introduction should be avoided. Figure 6.1 shows how rotations of non 90° multiples can create areas in the image with missing pixels and thus should be avoided. The same applies to image shifts.

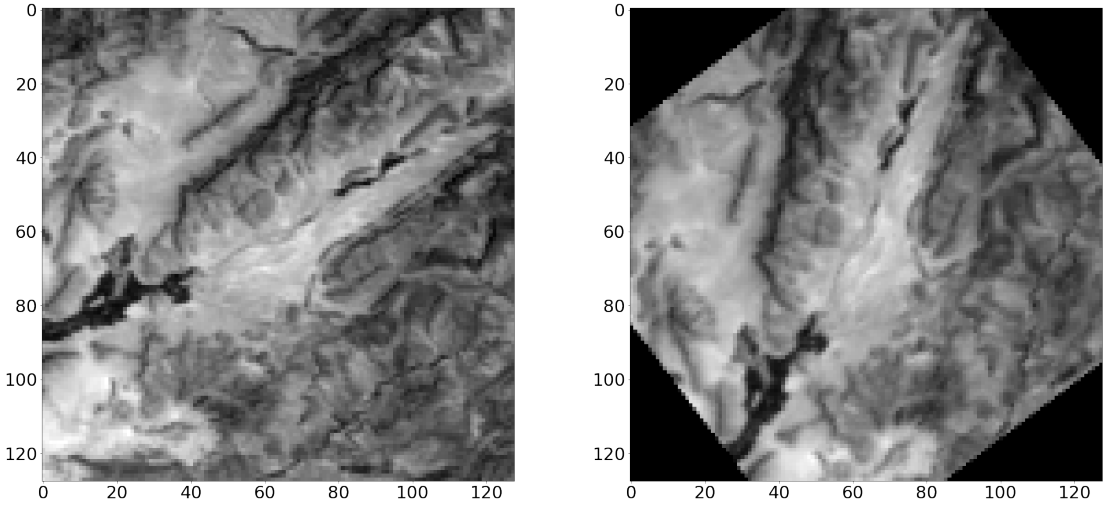
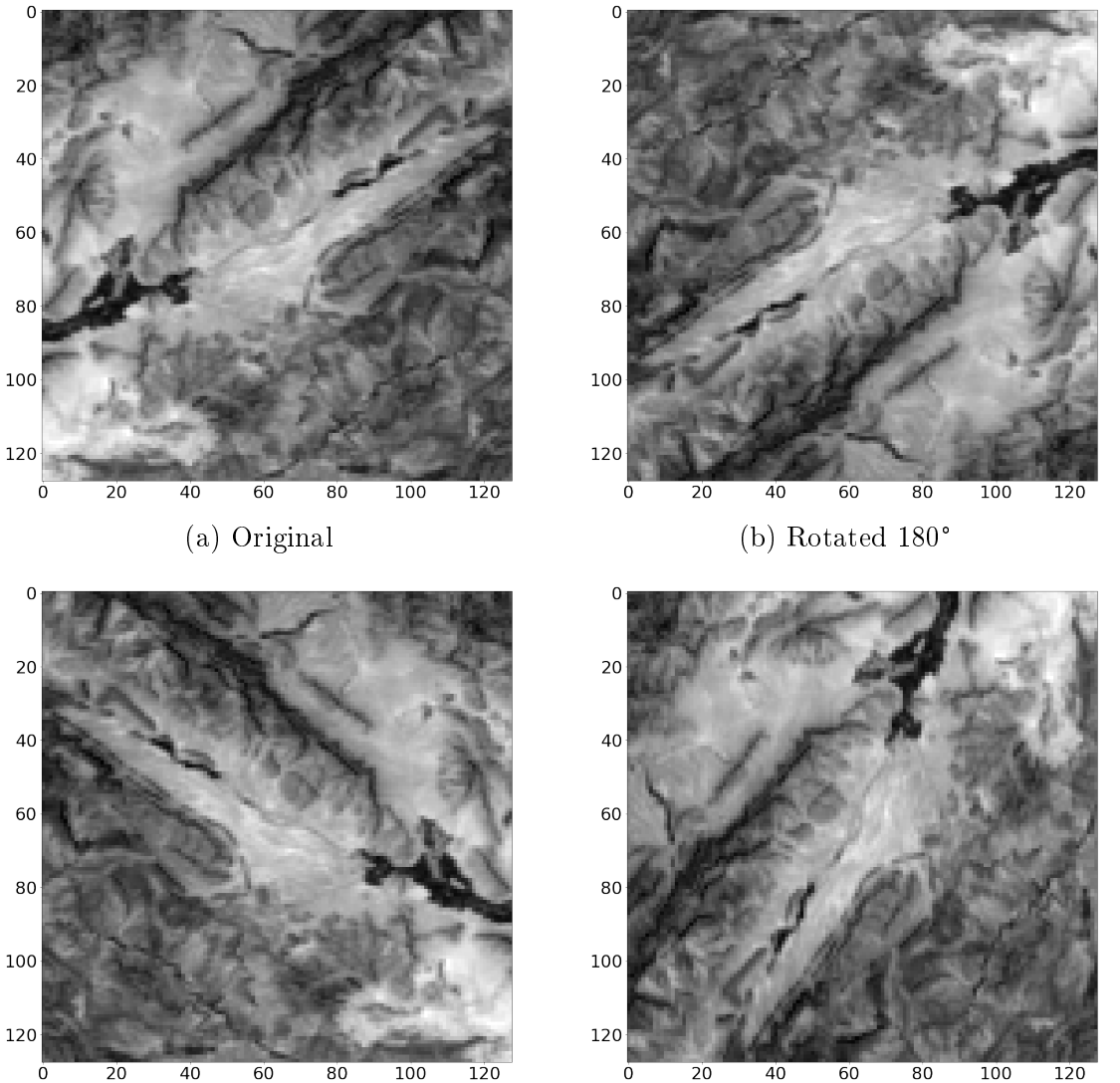


Figure 6.1 – Image rotation of 37° counter-clockwise. In the corners void pixels are obtained with the transformation.

Due to memory limitations, data augmentation is performed during the training phase using a generator that computes the transformation when needed. The transformations are applied to both LR and HR to preserve coherence between the pixel mapping. A random rotation is applied between (0°, 90°, 180°, 270°) , while flip is randomly selected between no flip, horizontal and vertical. Figure 6.2 shows an image and three possible transformations applied to it.



(a) Original

(b) Rotated 180°

(c) Flipped horizontally

(d) Rotated 90° ccw, then flipped vertically

Figure 6.2 – Data augmentation on an image

6.3 Model architecture

To design the model architecture, inspiration was taken from the networks found in the literature. The main part of the network should extract features from the input images and merge them together to extract the output SR image. Several models proposed for SISR perform the whole computation in the HR space, having as input the bicubic interpolation of the input images; others do the upscaling process in the last part of the network, performing the feature extraction in the LR space.
The proposed architecture is divided in two parts:

1. the first block performs a SISR upscaling of the input images independently

2. the second block merges together the upscaled images to reconstruct a single HR output

The idea behind this choice is to use an input for the merging part of the network that is better than the bicubic, in order to improve the final result. Figure 6.3 shows a schematic representation of the proposed architecture.

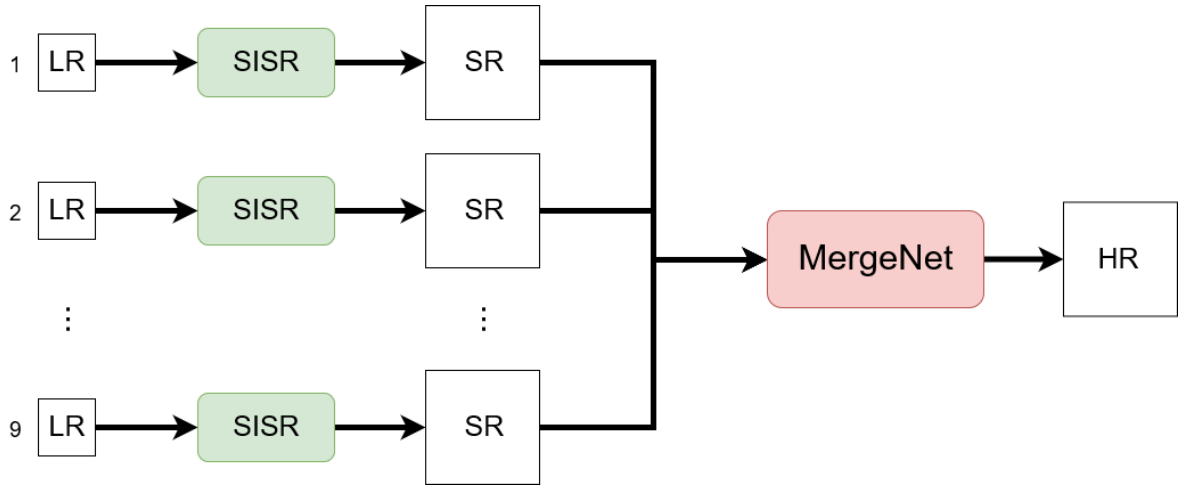


Figure 6.3 – Schematic representation of the proposed architecture

6.3.1 First block: Multi SISR

As mentioned, first part of the model performs independent image upscaling for the 9 LR images of a certain scene. Thus, that block implements a multi-image independent single-image super-resolution process. In fact, for efficiency reasons, the real implementation simply uses a single SISR network applied to the LR images organized in a batch, instead of actually implementing 9 times the same network.

For this first part of the model, EDSR [31] has been chosen as architecture, since it represents the SISR state of the art, according to [17]. It is composed of 32 local residual blocks, made of a chain of Conv2D, ReLU, Conv2D and a scaling factor equal to 0.1 for training stability reasons. All the convolutions have 3×3 kernel, 256 filters and have zero padding in order to keep the tensor dimensions. A global residual is performed and then the upsampling is made with the sub-pixel convolutional layer [34]. Since the scale for Proba-V Challenge is 3 (from 128×128 to 384×384), the convolution before the reshaping operation generates $256 \cdot 3^2$ feature maps, that are then reshaped into 256 HR feature maps. The last convolution outputs the super-resolved image.
Figure 6.4 shows the EDSR architecture as has been used for the proposed model. For a detailed description of the network and the up-sampling layer functionalities, refer to section 3.2.5.

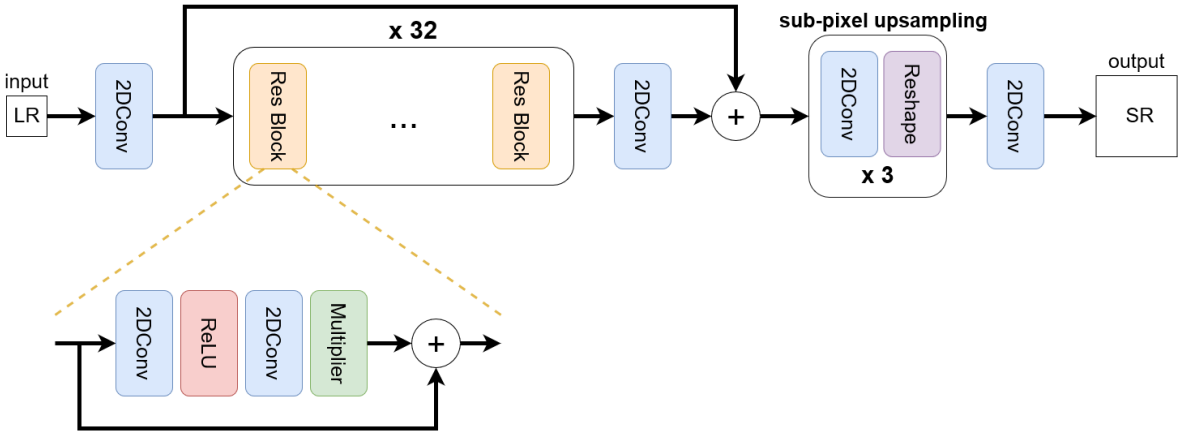


Figure 6.4 – Multi SISR architecture: EDSR

6.3.2 Second block: MergeNet

The second part of the model has the aim of merging the 9 SR images in order to further improve the super-resolution process. This block has basically to perform three main operations:

- extract cross-features between the various input images
- aggregate and reduce the extracted features
- fuse the results to get a single output image

Each of this operations has an associated sub-network part that will be analysed in the following sections.
The input of the second block is organized in a five dimensions tensor of shape $B \times W \times H \times C \times F$, where:

- B is the batch dimension
- W is the image width (384)
- H is the image height (384)
- C is the channel dimension (9)
- F is the feature dimension, that is 1 in the input and will contain the various feature maps extracted in the 3D convolutional layers

Thus, the 9 images coming from the SISR process are concatenated along the fourth axis before being fed to the MergeNet block.

Dense3D

Inspired by the DenseNet [36] and its super-resolution counterpart SRDenseNet [37], feature extraction from the SR images is performed by a densely connected network. This type of architecture concatenates all the features coming from previous convolutions increasing more and more the features entering in future layers. Since the new extracted maps are concatenated to the input, the filters number per convolutional

layer, called *growth rate*, is kept small, to avoid reaching an excessive number of features at the end of the Dense3D. Since we're not dealing with a 2D image, but with a set of 9 images, 3D convolutions are used instead of 2D, allowing the extraction of 3D cross-feature maps of shape $B \times 384 \times 384 \times 9 \times F$. Instead of a single convolution, a chain of ReLU, $1 \times 1 \times 1$ Conv3D (48 filters), ReLU, $3 \times 3 \times 3$ Conv3D (12 filters) is used as basic layer. This means that only the features after the second convolution are concatenated to the input: thus, the growth rate is actually 12, that is the number of the filters of the second convolution.
9 dense block are stacked one after the other, that leads to a total of $1 + 12 \cdot 9 = 109$ features in the output of this first sub-network. One thing to be underlined is that the set of 9 SR images, that composes the input to the MergeNet, is also present as output of the Dense3D, allowing to directly combine extracted features to the original images. Figure 6.5 shows the structure of the Dense3D.

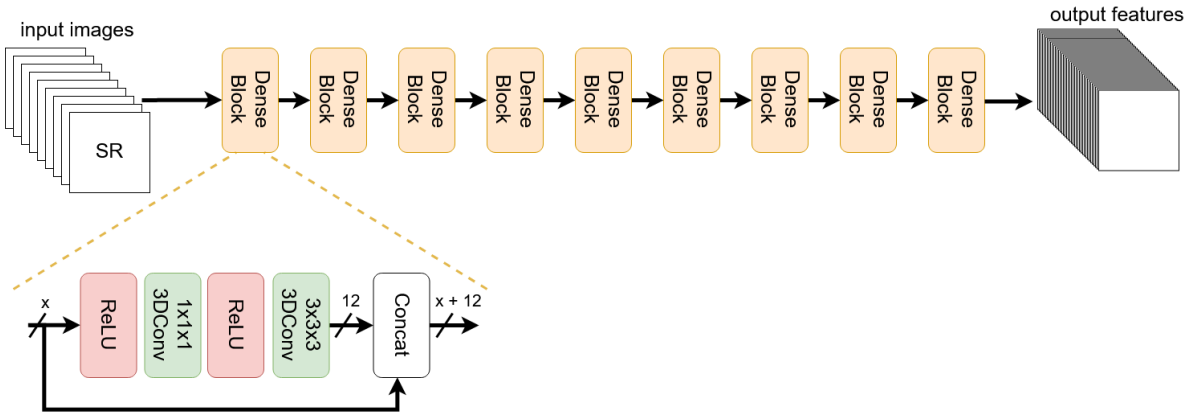


Figure 6.5 – MergeNet architecture: Dense3D

ReduceNet

After having extracted all the features with the Dense3D, the information should be aggregated and reduced to get a more compressed representation. To do so, a specific sub-network is used, characterized by 3D convolutions organized in a Inception-like fashion. The Inception block is an architecture proposed by [58] and is at the basis of the GoogleLeNet network. It applies 2D convolutions with different kernels to the same

input and then concatenates the output features, in order to simultaneously being able to explore different receptive field. The ReduceNet takes inspiration from that Inception block but applies with 3D Convolutions.
To all the features coming from the previous sub-networks are applied the following convolutions:

- a $1 \times 1 \times 1$ Conv3D with k filters
- a $1 \times 1 \times 1$ Conv3D with $3k$ filters followed by a $3 \times 3 \times 3$ Conv3D with k filters
- a $3 \times 3 \times 3$ Conv3D with k filters

All the 3D convolutions in this block have ReLU activation. 2 blocks of that kind are stacked one after the other respectively with $k = 16$, $k = 8$. After the last block two additional Conv3D are added, respectively with $1 \times 1 \times 1$ kernel and 24 filters and $3 \times 3 \times 3$ kernel and 1 filter. This means that the output of the ReduceNet is similar to the input of the MergeNet, that is a five-dimensional tensor with shape $B \times 384 \times 384 \times 9 \times 1$. This tensor represents a set of compacted extracted cross-features that should be fused together to get the effective output.
Figure 6.6 shows the architecture of the ReduceNet sub-network.

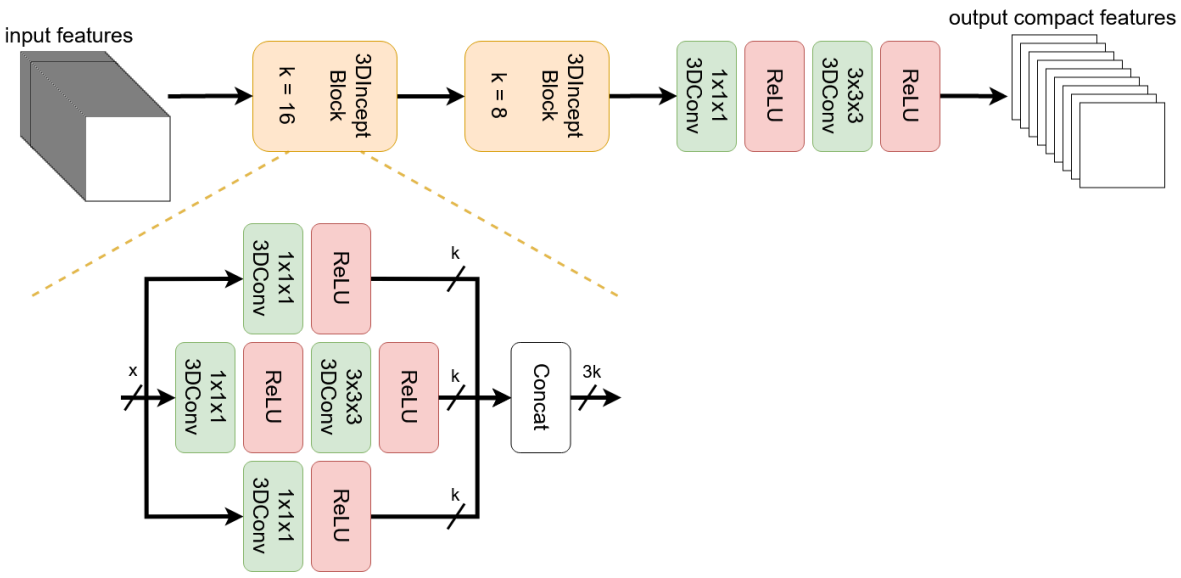


Figure 6.6 – MergeNet architecture: ReduceNet

FusionNet

The last part of the MergeNet should fuse the compacted cross-features together. First the last dimension of the tensor is deleted to get a four-dimensional data with shape $B \times 384 \times 384 \times 9$. These are considered as compact features of the target image, so 2D convolutions will be used from now on. This last sub-network has as input a special inception-inspired block that performs simultaneous Conv2D with 1×1, 3×3, 5×5 and 9×9 kernels, all with ReLU activation and 128 filters. Then a simple chain of three Conv2D with 3×3 kernel is used to compute the residual with respect to the average of the input features. The output is a $B \times 384 \times 384 \times 1$ tensor, which contains the SR images for the batch. FusionNet architecture is represented in figure 6.7.

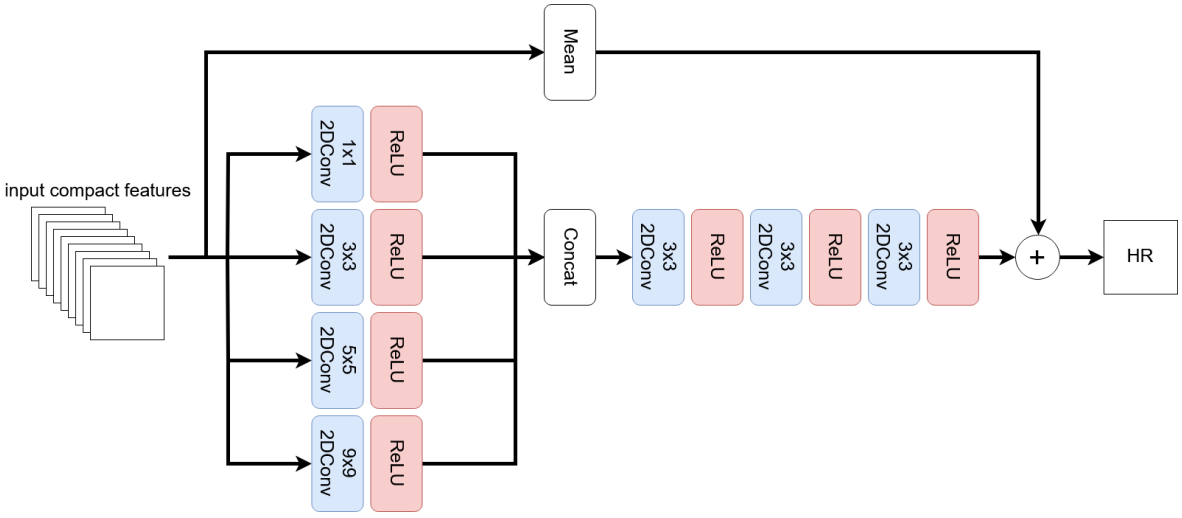


Figure 6.7 – MergeNet architecture: FusionNet

MergeNet overall architecture

Resuming, the MergeNet aggregates the SR images coming from the Multi SISR block and is composed of three sub-networks: Dense3D, ReduceNet and FusionNet. The first two are characterized by features extraction and manipulation of the 3D block of images, thus use 3D convolutions as main operation; the latter fuses the extracted features to get to the output SR image, thus uses 2D convolutions. The activation used for almost all the convolutional layers is the ReLU and zero padding is always used to

keep constant the tensor dimensions.
The total number of the network parameters is around 600K. It's important to underline that this value has to be kept low, since due to GPU memory limitations. Several experiments has been done with different values for the Dense3D grow rate k, different number of Dense Blocks or Incept Blocks, different quantity of filters per layer. Most of this set-ups ended in memory overflow error due to the huge amount of tensors and gradients to be instantiated during the training process.

One important aspect to be underlined is that in this model no image registration procedure is taken in account to avoid further complicating the network. This is done considering the fact that the Proba-V dataset contains subpixel shifts between the different LR images, that means a maximum shift of only 3 pixels in the HR domain. The error coming from the misalignment of the LR images is therefore negligible, also considering that the competition scoring method (see section 4.3) uses the best three-pixel SR/HR alignment for each submitted image.
An overall scheme of the architecture of the second block is represented in figure 6.8.

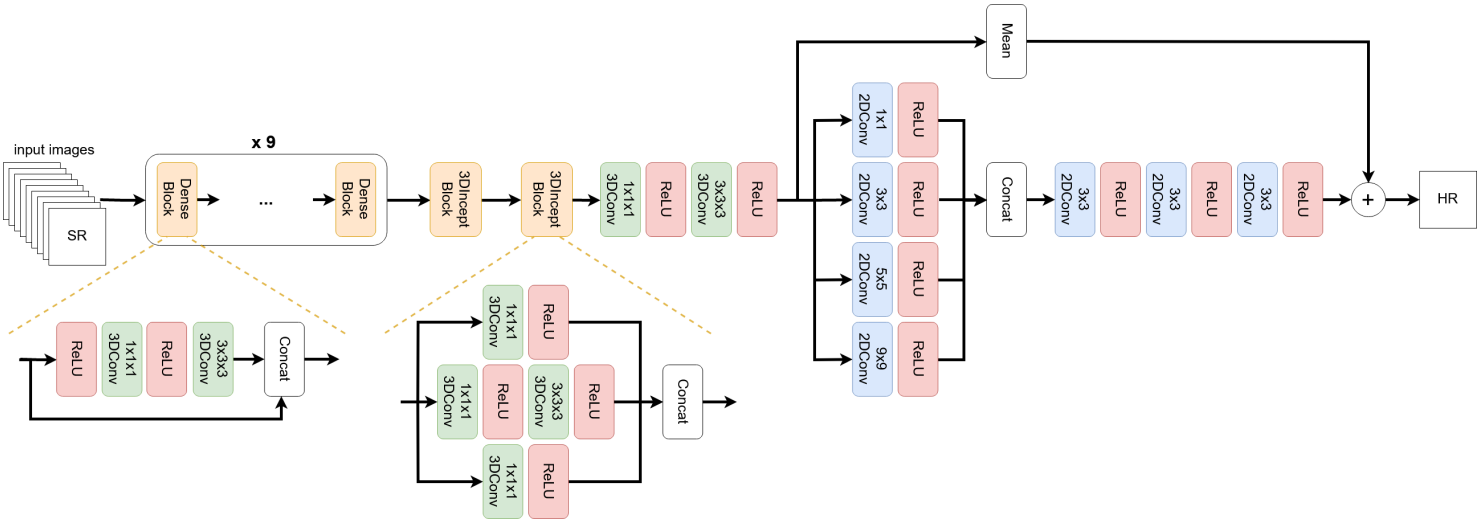


Figure 6.8 – Overall MergeNet architecture

6.4 Training

The training process of the model has been split in two sessions: firstly the Multi SISR network has been trained alone, then the MergeNet.

Multi SISR

Since this is a SISR problem in which we want to improve the resolution of each LR image without considering any possible relation with the other images, the training of that first block is done *without* using the LR images. Instead, a downscaled version of the HR image has been used as input, in order to not create problems with image registration or temporal differences such as different cloud coverage. Thus, the HR images (384×384 shape) are locally averaged at blocks of 3×3 pixels, resulting in images of dimension 128×128.

To better allow the learning of local features, the network has been trained with patches derived from the selection of part of the downscaled images, similarly to what found in literature for SISR . In particular, square sub-images of dimension $d = 48$ pixels with stride $s = 16$ has been generated. The stride is the number of pixels that separate two subsequent patches: if $s \neq d$, the patches are overlapping. Denoting the side dimension of the original image as d_o, the number of total patches edges that fit into an image side is:

$$n = \frac{d_o - d}{s} + 1 \tag{6.2}$$

If $d_o - d$ is divisible by s, n is integer and all the patches has exactly side dimension d. With the selected values, we get $n = 6$. Since the same reasoning can be done on both the image sides, the total number of patches generated by a single image is $n^2 = 36$. The HR image has to be coherently patched with scaled (3x) d and s to preserve the LR/HR matching.

Since the original LR images are not used for training the EDSR, there is no possible pixel shift or intensity bias between the input and the HR target, therefore, there is no need to use the challenge scoring method. L1 loss (mean absolute value) has been selected, instead of MSE, since the authors of EDSR suggest it to increase convergence. Adam has been chosen as optimizer, with learning rate set to 0.0001. A Keras callback is used to halve the learning rate if no loss improvements are achieved for more than 5

epochs until the minimum of 1×10^{-6} is reached. The network has been trained with mini-batch size of 16 patches for 90 epochs.

MergeNet

To generate the input images for the second block of the model, the dataset is selected with the threshold on clear pixels as described in section 6.2. The dataset LR images are then fed to the first block using the geometric self-ensemble described in the EDSR paper [31] (see section 3.2.5). The obtained SR images are then organized in a five-dimensional numpy array with shape $1160 \times 384 \times 384 \times 9 \times 1$. Unlike the first block, the MergeNet is trained with the entire images and not patches. This is due to the nature of the selected loss function, that is the criteria chosen by the challenge organizers (see section 4.3). This function performs a 3 pixels crop on each side of the SR image and then check for the HR sub-image that leads to the best cPSNR value. Due to this pixel crop, it is better to work on the effective image to avoid excluding pixels that are within the image and not at the borders.
Data augmentation is used as described in section 6.2, generating random rotations (90°, 180°, 270° or no rotation) and flip (horizontal, vertical or no flip) for each SR/HR couple. As in the previous training, the Adam optimizer is used, with initial learning rate equal to 0.0001 and the decay Keras callback that helps to boost the learning in case of a loss plateau. Among the several mini-batch sizes tested, the only that doesn't lead to GPU memory overflow error is 1, therefore stochastic descent is performed, with a single scene handled at a time.
Due to time limitations for the competition, the training is performed for a total number of 130 epochs, with an epoch mean time of 33.5 minutes, thus resulting in a total training time of almost three days. After the end of the challenge, the training has been resumed for a total of 227 epochs until convergence has been reached.

6.5 Results

Figure 6.9 shows the curve of the evolution of mean cPSNR for the whole dataset over the training epochs. The mean cPSNR obtained for epoch 130 (48.19 dB) is highlighted, since it is the value reached at the time of the final submission. The test images has been fed to the network and submitted and received the final score of 0.97219 (computed with the method described in section 4.3), that corresponds to the fifth place among the challenge participants.
A final mean cPSNR of 48.55 dB is reached at the end of the additional training epochs, performed after the end of the challenge. Is important to notice the two cPSNR spikes at epochs 169 and 213. These are due to the learning rate halving performed by the Keras decay callback. This function checks loss trend and, in case of 5 subsequent epochs with no improvements, halves the learning rate, to allow exploring smaller local minima. This process can help pushing further a training that is starts to converge. Since learning rate has been halved two times, its final value reaches 0.00025.

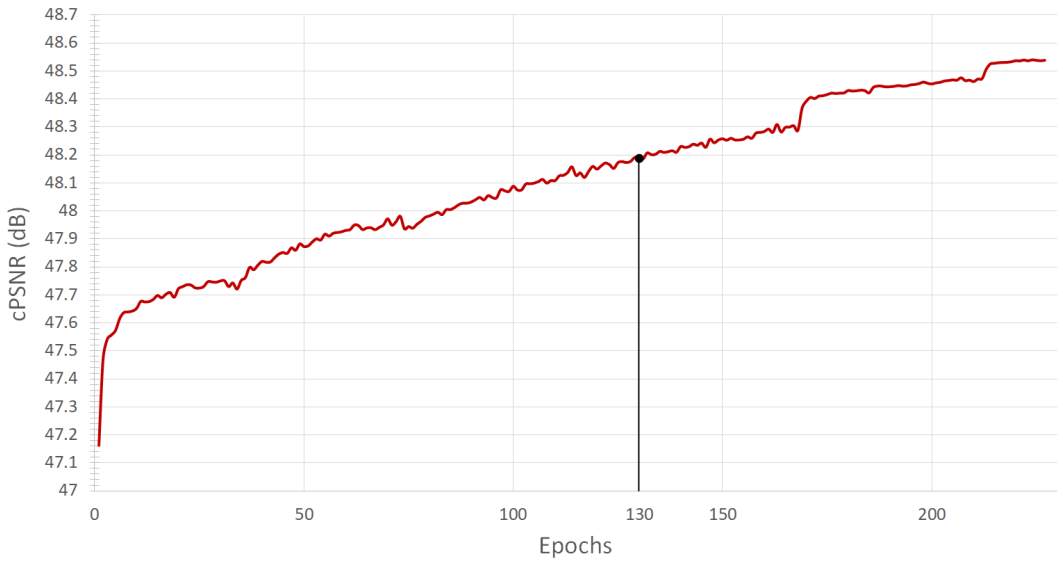
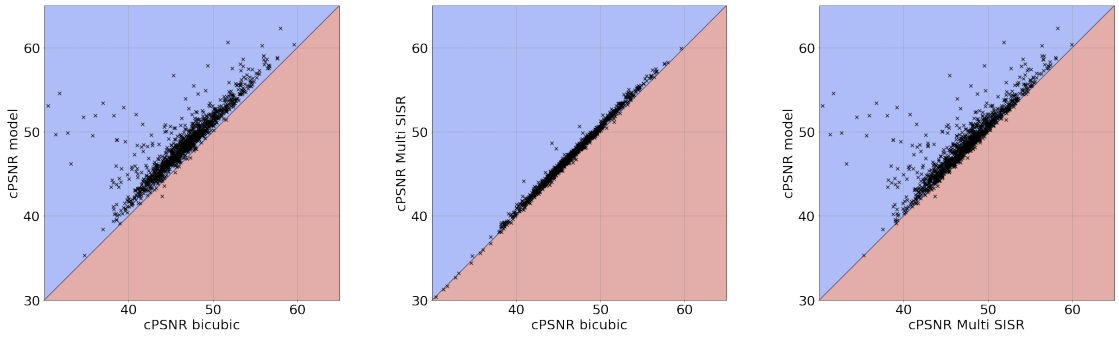


Figure 6.9 – Training curve of mean cPSNR vs epochs

A comparison between bicubic cPSNR and model cPSNR is performed for each image. Bicubic cPSNR is computed considering the average of the 9 LR images after being

upscaled with bicubic interpolation. The same process is performed with the images after the first block, considering the average of the 9 images coming from the Multi SISR block (EDSR). Results are shown in figure 6.10. Each cross represents a dataset image: being in the upper part of the graph means that cPSNR is higher for the left model. As can be seen, the proposed model achieves better results with respect to both bicubic and Multi SISR only. Table 6.2 presents some statistical information about cPSNR results on the train dataset with respect to the baseline solution presented in section 4.3.5. Figures 6.11 and 6.12 present some examples on training images.



(a) Proposed model vs mean bicubic

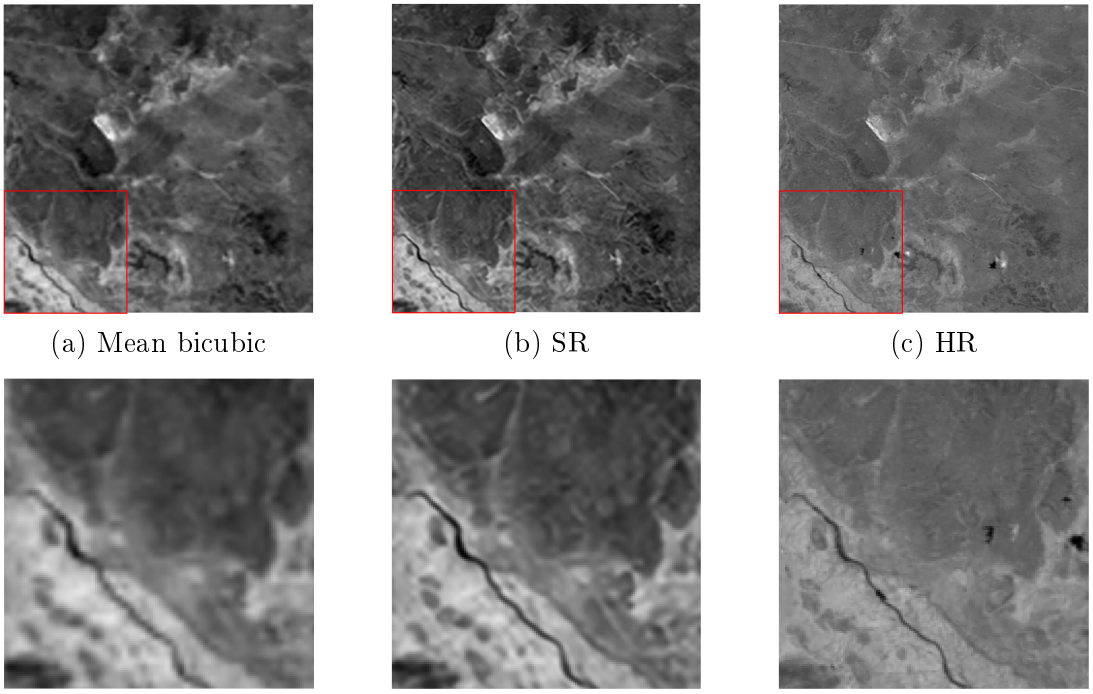
(b) Mean Multi SISR vs mean bicubic

(c) Proposed model vs Mean Multi SISR

Figure 6.10 – cPSNR comparison for different models. Multi SISR performs better than bicubic and the proposed model further improves almost all the images.

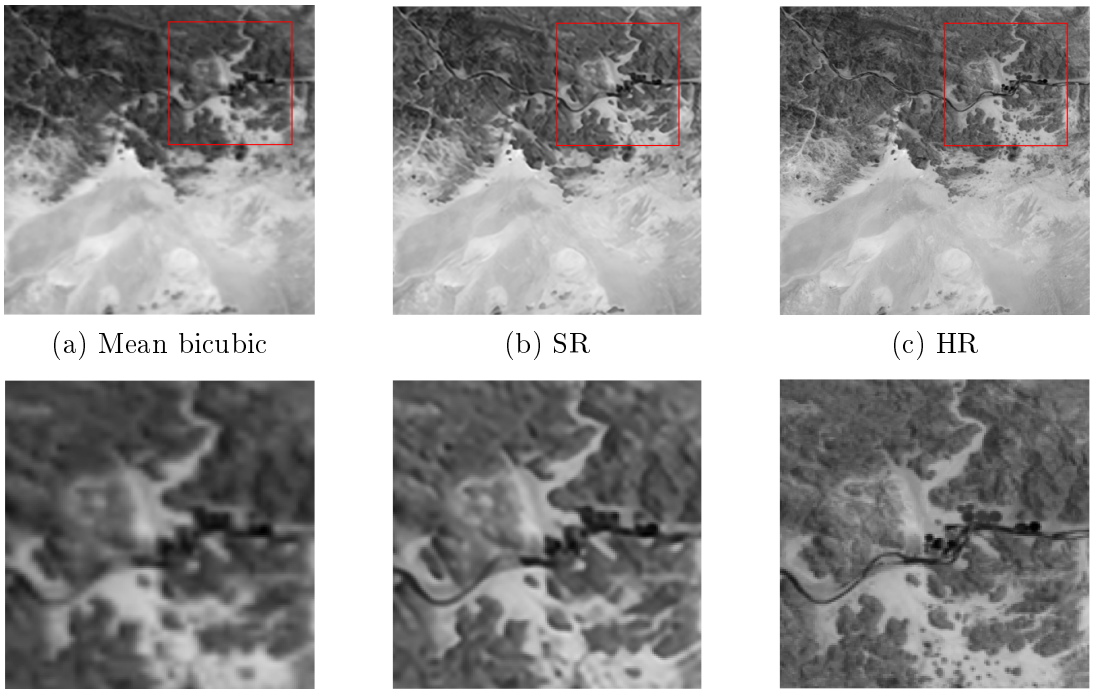
Index	**Model**	**Baseline**
Mean	48.55	46.49
Standard deviation	3.71	3.80
Minimum	35.36	31.45
Maximum	62.32	59.72
25^{th} percentile	46.07	44.11
50^{th} percentile	48.39	46.46
75^{th} percentile	50.76	48.76

Table 6.2 – cPSNR results statistics



(a) Mean bicubic (b) SR (c) HR

Figure 6.11 – cPSNR mean bicubic: 46.74 dB, cPSNR SR: 49.53 dB



(a) Mean bicubic (b) SR (c) HR

Figure 6.12 – cPSNR mean bicubic: 43.57 dB, cPSNR SR: 46.75 dB

6.6 Conclusions and Future Work

As demonstrated with the study of the state of the art, Deep Learning applied to super-resolution is a very prolific field of study. In particular, Deep Learning for Multi-image Super-resolution is a relatively new approach, with few proposed applications and that can be further analysed and developed.
The proposed model shows good results for the submitted dataset and has resulted in the fifth place in the competition. The results obtained, compared with the bicubic and SISR only predictions, show how using more images can help in reconstructing missing pixels and therefore can push super-resolution application beyond what has done until done, especially for satellite imagery, that is characterized by frequent reshots of the same scenes. That can be vital to improve performance of remote imaging applications for agriculture and environmental monitoring and land coverage study.

6.6.1 Future work

Further study can be performed to try to address the following problems emerged during the development of this project:

- reduce the model resource demand, in particular trying to simplify the network, still obtaining good results in term of PSNR;
- implement a registration deep learning method to try to further increase pixel coherence among the different images and thus improve performance;
- fuse the two model blocks in order to get a single compact network that can be trained end-to-end;
- try to remain in the LR domain for the most part of the feature extraction and manipulation, performing the upscaling process at the end of the network in order to reduce computational complexity and memory demand;
- perform separate trainings for the two available bands (NIR and RED) to verify it this improves results;

- try to reuse the same model in other contexts with respect to the one provided for the challenge.

All this aspects can be studied in order to develop a better model that can actually reach MISR state-of-the-art performance. One key aspect will be the study of the challenge winners solution [59], in order to understand similarities and differences with their approach.
One similar context to try continuing super-resolution research can be a satellite/drone image mapping, trying to refine freely available satellite images (such as those provided by Sentinel-2) using drone images of a certain area as HR reference. This would show that super-resolution can be successfully applied also for images obtained with very different technologies.

Bibliography

[1] A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 1st ed. O'Reilly Media, Inc., 2017.

[2] S. Shalev-Shwartz and S. Ben-David, *Understanding Machine Learning: From Theory to Algorithms*. New York, NY, USA: Cambridge University Press, 2014.

[3] G. Bonaccorso, *Machine Learning Algorithms: A Reference Guide to Popular Algorithms for Data Science and Machine Learning*. Packt Publishing, 2017.

[4] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[5] A history of machine learning. [Online]. Available: https://cloud.withgoogle.com/build/data-analytics/explore-history-machine-learning

[6] H. Mayo, H. Punchihewa, J. Emile, and J. Morrison. History of machine learning. [Online]. Available: https://www.doc.ic.ac.uk/~jce317/history-machine-learning.html

[7] E. Roberts. Neural networks: history. [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/index.html

[8] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, Dec 1943.

[9] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM J. Res. Dev.*, vol. 3, no. 3, pp. 210–229, Jul. 1959.

[10] F. Rosenblatt, *Principles of Neurodynamics*. Spartan Books, 1959.

[11] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.

[12] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-propagating Errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[13] D. Fumo. (2017) Types of machine learning algorithms you should know. [Online]. Available: https://towardsdatascience.com/types-of-machine-learning-algorithms-you-should-know-953a08248861

[14] R. Gómez. (2018) Understanding categorical cross-entropy loss, binary cross-entropy loss, softmax loss, logistic loss, focal loss and all those confusing names. [Online]. Available: https://gombru.github.io/2018/05/23/cross_entropy_loss/

[15] S. Ruder. (2016) An overview of gradient descent optimization algorithms. [Online]. Available: http://ruder.io/optimizing-gradient-descent/

[16] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, 1998, pp. 2278–2324.

[17] W. Yang, X. Zhang, Y. Tian, W. Wang, J.-H. Xue, and Q. Liao, "Deep learning for single image super-resolution: A brief review," *IEEE Transactions on Multimedia*, 2019.

[18] C. Solomon and T. Breckon, *Fundamentals of Digital Image Processing: A Practical Approach with Examples in Matlab*, 1st ed. Wiley Publishing, 2011.

[19] "Individual guidelines for noting digital camera specifications on number of pixels, image file and focal length of the lens," Camera & Imaging Products Association, Tokio, JP, Guideline, Jan. 2018.

[20] S. Liang, X. Li, and J. Wang, *Advanced remote sensing: terrestrial information extraction and applications*. Academic Press, 2012.

[21] M. Bevilacqua, A. Roumy, C. Guillemot, and M. L. Alberi-Morel, "Low-complexity single-image super-resolution based on nonnegative neighbor embedding," 2012.

[22] S. C. Park, M. K. Park, and M. G. Kang, "Super-resolution image reconstruction: a technical overview," *IEEE signal processing magazine*, vol. 20, no. 3, pp. 21–36, 2003.

[23] K. Srinivasan and J. Kanakaraj, "A study on super-resolution image reconstruction techniques," *Computer Engineering and Intelligent Systems*, vol. 2, no. 4, pp. 222–227, 2011.

[24] S. Borman and R. Stevenson, "Spatial resolution enhancement of low-resolution image sequences-a comprehensive review with directions for future research," *Lab. Image and Signal Analysis, University of Notre Dame, Tech. Rep*, 1998.

[25] S. Borman and R. L. Stevenson, "Super-resolution from image sequences-a review," in *1998 Midwest Symposium on Circuits and Systems (Cat. No. 98CB36268)*. IEEE, 1998, pp. 374–378.

[26] J. Hadamard, "Sur les problèmes aux dérivées partielles et leur signification physique," *Princeton university bulletin*, pp. 49–52, 1902.

[27] C.-Y. Yang, C. Ma, and M.-H. Yang, "Single-image super-resolution: A benchmark," in *European Conference on Computer Vision*. Springer, 2014, pp. 372–386.

[28] C. Dong, C. C. Loy, K. He, and X. Tang, "Image super-resolution using deep convolutional networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 2, pp. 295–307, Feb 2016.

[29] J. Kim, J. Kwon Lee, and K. Mu Lee, "Accurate image super-resolution using very deep convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 1646–1654.

[30] Y. Tai, J. Yang, and X. Liu, "Image super-resolution via deep recursive residual network," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 3147–3155.

[31] B. Lim, S. Son, H. Kim, S. Nah, and K. Mu Lee, "Enhanced deep residual networks for single image super-resolution," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2017, pp. 136–144.

[32] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang *et al.*, "Photo-realistic single image super-resolution using a generative adversarial network," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4681–4690.

[33] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[34] W. Shi, J. Caballero, F. Huszár, J. Totz, A. P. Aitken, R. Bishop, D. Rueckert, and Z. Wang, "Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 1874–1883.

[35] R. Timofte, R. Rothe, and L. Van Gool, "Seven ways to improve example-based single image super resolution," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 1865–1873.

[36] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[37] T. Tong, G. Li, X. Liu, and Q. Gao, "Image super-resolution using dense skip connections," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 4799–4807.

[38] D. Martin, C. Fowlkes, D. Tal, J. Malik *et al.*, "A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics." Iccv Vancouver:, 2001.

[39] J. Yang, J. Wright, T. S. Huang, and Y. Ma, "Image super-resolution via sparse representation," *IEEE transactions on image processing*, vol. 19, no. 11, pp. 2861–2873, 2010.

[40] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[41] E. Agustsson and R. Timofte, "Ntire 2017 challenge on single image super-resolution: Dataset and study," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017, pp. 126–135.

[42] Z. Li, S. Li, J. Wang, and H. Wang, "A novel multi-frame color images super-resolution framework based on deep convolutional neural network," in *2016 5th International Conference on Measurement, Instrumentation and Automation (ICMIA 2016)*. Atlantis Press, 2016.

[43] J. Wu, T. Yue, Q. Shen, X. Cao, and Z. Ma, "Multiple-image super resolution using both reconstruction optimization and deep neural network," in *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, 2017, pp. 1175–1179.

[44] M. Kawulok, P. Benecki, S. Piechaczek, K. Hrynczenko, D. Kostrzewa, and J. Nalepa, "Deep learning for multiple-image super-resolution," *arXiv preprint arXiv:1903.00440*, 2019.

[45] M. Kawulok, P. Benecki, K. Hrynczenko, D. Kostrzewa, S. Piechaczek, J. Nalepa, and B. Smolka, "Deep learning for fast super-resolution reconstruction from multiple images," in *Real-Time Image Processing and Deep Learning 2019*, vol. 10996. International Society for Optics and Photonics, 2019, p. 109960B.

[46] M. Märtens, D. Izzo, A. Krzic, and D. Cox, "Super-resolution of proba-v images using convolutional neural networks," *Astrodynamics*, pp. 1–16, 2019.

[47] C. Dong, C. C. Loy, and X. Tang, "Accelerating the super-resolution convolutional neural network," in *European conference on computer vision*. Springer, 2016, pp. 391–407.

[48] (2019) Proba-v super resolution. [Online]. Available: https://kelvins.esa.int/proba-v-super-resolution

[49] Proba-v. [Online]. Available: https://m.esa.int/Our_Activities/Observing_the_Earth/Proba-V/

[50] T. E. Oliphant, *A guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[51] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[52] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

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

[54] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors, "scikit-image: image processing in Python," *PeerJ*, vol. 2, p. e453, 6 2014. [Online]. Available: https://doi.org/10.7717/peerj.453

[55] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87 – 90.

[56] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[57] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[58] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Van-

houcke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[59] A. B. Molini, D. Valsesia, G. Fracastoro, and E. Magli, "Deepsum: Deep neural network for super-resolution of unregistered multitemporal images," *arXiv preprint arXiv:1907.06490*, 2019.

Acknowledgements

I would like to thank my thesis supervisor Prof. Marcello Chiaberge for giving me the opportunity to collaborate with the PIC4SeR and for his support throughout the entire development of this work. My gratitude goes also to all the people I met during these months, in particular the other thesis students that shared with me time and efforts needed to achieve our goals. I want to reserve a special mention to Vittorio Mazzia, for having guided me in addressing the technical aspects of the research and for his help during the last phases of the competition.
I must express all my gratitude to my family for being always present and to all my friends: you supported and encouraged me during these years.
Finally, thanks to Valeria for being always by my side and having shared with me most of the happy and difficult moments of these years.