UNIVERSIDAD POLITÉCNICA DE MADRID
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS EN
TOPOGRAFÍA, GEODESIA Y CARTOGRAFÍA

GRADO EN INGENIERÍA DE LAS TECNOLOGÍAS DE LA
INFORMACIÓN GEOESPACIAL

TRABAJO FIN DE GRADO

EXPLORATION OF LEARNING TECHNIQUES TO IMPROVE THE SPATIAL RESOLUTION OF ORTHOIMAGES

Madrid, Septiembre del 2021

***Alumna*:**
Mónica Apellaniz Portos

***Tutores*:**
Miguel Ángel Manso Callejo
Calimanut-Ionut Cira

UNIVERSIDAD POLITÉCNICA DE MADRID

GRADO EN INGENIERÍA DE LAS TECNOLOGÍAS DE LA INFORMACIÓN GEOESPACIAL

2020 - 2021

TRABAJO FIN DE GRADO

Exploration of Learning Techniques To Improve the Spatial Resolution of Orthoimages

Mónica Apellaniz Portos

Tutores

Miguel Ángel Manso Callejo

Calimanut-Ionut Cira

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS EN TOPOGRAFÍA, GEODESIA Y CARTOGRAFÍA

Agradecimientos:

Gracias a mis tutores y profesores Miguel Ángel y Ionut. Por ayudarme con este proyecto. Por enseñarme a mejorar. Por el apoyo que me han dado durante toda esta etapa y el interés que han mostrado por mi futuro. Muchas gracias por todo.

Gracias a mis compañeros y amigos que han caminado junto a mí en esta aventura. Por el apoyo y comprensión. Por seguir alegrándonos con nuestros logros y sacando sonrisas siempre que las necesitemos.

Gracias a mis hermanas, por ese "amor de hermanas" que aunque no se entienda, siempre está. Gracias por nuestro apoyo tan leal e incondicional

Gracias una y mil veces más a mi madre. Por ser, estar y aguantar.
Te quiero.

Y gracias papá, porque has sido, eres y serás siempre mi fuente inagotable de inspiración.

Abstract:

The aerial images obtained by satellite and airborne sensors, as well as their orthoimages, there are representations of the Earth's surface with very varied uses and applications. Nowadays, this capturing technology has strongly improved, and its quality and informative value too. The resolution of digital orthoimages represents the amount of detail that can be seen in them. A higher resolution implies a greater number of pixels per unit of length, thus increasing the size of the image, which means, a greater use of resources for its storage and processing.

This project presents a study with a critical analysis of different upscaling methods, implemented in Python. The artificial neural networks applied for the upscaling procedure have won different computer vision competitions. However, in the project, the original architectures have been modified and adapted to our purpose, i.e., to improve the resolution of aerial images from PNOA (National Plan for Aerial Orthophotography) and other flights.

In the framework of the SROADEX project (PID2020-116448GB-I00), some methods such as the classic interpolation algorithms: bilinear, nearest neighbor, bicubic, and Lanczos; have been compared with AI algorithms: Antagonic Generative Networks and Feedback Networks.
The analysis and experiments were carried out using training, validation, and testing sets containing high resolution . These are made up of high-resolution orthoimages, with 10 and 15 cm/pixel. These images are representative of the Spanish geography, with coastal and urban areas, and with/without vegetation. For the evaluation and comparison of the mentioned upsampling methods, we are going to use two quantitative metrics used to measure the quality of the SR-generated images: the Maximum Signal to Noise Ratio (*PSNR*) and the Structural Similarity Index (*SSIM*). Depending on the obtained results, we propose some architecture modifications based on their hyperparameter influence and behavior.

Finally, trained networks have been applied to generate high-resolution images from low-resolution images, which implies a better use of available resources and a high increase in the informative value of the product. The networks significantly improved performance metrics and in the qualitative evaluation we observe high quality predictions.

Keywords:

Upscaling, Ortoimages, Python, Artificial Intelligence, Interpolation Methods, Bilinear, Bicubic, Lanczos, Generative Adversarial Networks, Feedback Networks, Super-resolution, PSNR, SSIM, PNOA, Spain.

Resumen:

Las imágenes aéreas obtenidas por satélite y sensores aerotransportados, así como las ortoimágenes que se generan a partir de ellas, son instantáneas de la superficie terrestre con usos y aplicaciones muy variados. Hoy en día la tecnología con la que se obtienen estas imágenes ha mejorado enormemente y, con ello, su calidad y valor informativo. Las ortofotografías digitales se caracterizan por su resolución, es decir, la cantidad de detalle que se puede ver en ellas. Una mayor resolución implica un mayor número de píxeles por unidad de longitud, incrementando así el tamaño de la imagen, lo que supone un mayor uso de recursos para su almacenamiento y procesamiento.

En este proyecto se presenta un estudio y análisis crítico de diferentes métodos de "upscaling" o mejora de la resolución de imágenes, implementados en Python. Estas arquitecturas de inteligencia artificial han ganado diferentes competiciones y, en este estudio, han sido modificadas y adaptadas para la mejora de la resolución en imágenes aéreas del PNOA (*Plan Nacional de Ortofotografía Aérea*), y otros vuelos.

En el marco del proyecto SROADEX (PID2020-116448GB-I00) se han comparado métodos como los algoritmos clásicos de interpolación: bilineal, vecino más cercano, bicúbico y Lanczos; con los obtenidos al entrenar los algoritmos de inteligencia artificial: Redes Generativas Antagónicas y las Redes de Retroalimentación.
El estudio y los experimentos se han realizado a partir de conjuntos de datos de entrenamiento, validación y evaluación, compuestos por ortoimágenes de alta resolución, 10 y 15 cm/píxel. Estas imágenes son representativas de la geografía de España con zonas costeras, urbanas y tanto con vegetación como sin ella.
Para la evaluación y comparación de los diferentes métodos presentados, se utilizan dos métricas cuantitativas de medida de la calidad de las imágenes generadas: la Proporción Máxima de Señal a Ruido (*Peak signal to noise ratio - PSNR*) y el Índice de Similitud Estructural (*Structural similarity index - SSIM*). En función de los resultados obtenidos, se justifican los cambios generados en las arquitecturas y la influencia de los hiperparámetros, así como el comportamiento de los mismos.

Finalmente, se han aplicado las redes entrenadas para generar imágenes de alta resolución a partir de imágenes de baja resolución, lo que implica un mayor aprovechamiento de los recursos disponibles y un aumento considerable del valor informativo del producto. Las redes mejoraron significativamente los valores de las métricas, y en la evaluación cualitativa observamos predicciones de alta calidad.

Keywords:

Upscaling, Ortoimágenes, Python, Inteligencia Artificial, Métodos de interpolación, Bilineal, Bicúbica, Lanczos, Redes Generativas Antagónicas, Redes de Retroalimentación, Super resolución, PSNR, SSIM, PNOA, España.

ACRONYMS

AI *Artificial Intelligence*

ML *Machine Learning*

DL *Deep Learning*

SVM *Support Vector Machines*

ANN *Artificial Neural network*

NN *Neural Network*

DNN *Deep Neural Network*

DN *Digital Number*

RGB *Red-Green-Blue*

NNI *Nearest Neighbor Interpolation*

GAN *Generative Adversarial Network*

SR-GAN *Super Resolution Generative Adversarial Network*

LR *Low Resolution*

HR *High Resolution*

SR *Super Resolution*

cm *Centimetre*

FC *Fully connected*

LRFB *Low Resolution Feature extraction Block*

FB *Feedback Block*

RB *Reconstruction Block*

PSNR *Peak Signal to Noise Ratio*

MSE *Mean Square Error*

SSIM *Structural Similarity Index*

HDTV *Structural High Definition Television*

DVR *Digital Video Recorder*

UAV *Unmanned Aerial Vehicle*

MRI *Magnetic Resonance Imaging*

PET *Positron Emission Tomography*

SISR *Single Image Super Resolution*

OS *Operating System*

GIS *Geographic Information System*

GUI *Graphical User Interface*

URL *Uniform Resource Locator*

PC *Personal Computer*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF CODE SNIPPETS

1. INTRODUCTION

1.1. Super-resolution

What is super-resolution? Is it useful? How can we get it? What is it used for? How is it applied and in which fields? When we are talking about super-resolution, are we talking about past, present, future…?

Image super-resolution refers to a set of algorithms whose final objective is to improve the spatial resolution of an image above the acquisition resolution (LR) improving the visual quality, i.e., recovering images from LR [94][90]. There are different methods and techniques related to the improvement of the resolution of the images:

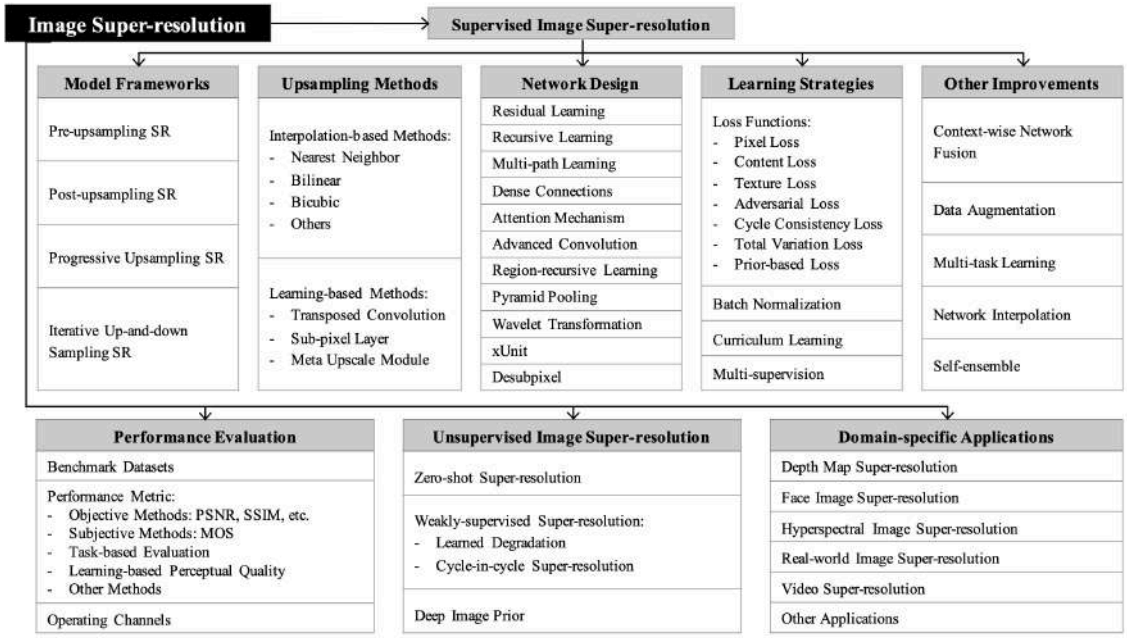

Fig. 1.1 Image Super-resolution survey [114].

We can distinguish two main methods: ***scaling methods***, which include the classic interpolation techniques, and ***the increased resolution methods by learning***, like the two networks that we are presenting in this project, SRGAN, and SRFBNs.

We can also differentiate both techniques by the number of LR images from the start, to finally obtain de HR image. If the SR image is obtained from a sequence of LR images taken from the same scene, extracting and fusing the common information, we are

talking about ***multi-imaging techniques*** or based on sequences, presented in papers like Irani and Peleg [50] and Farsiu et al. [30]; If the SR image is obtained from a unique LR image, we refer to single image SR [120], a very popular and used technique in the last couple of years, as we can see in papers from Glasner et al.[37], Dong et al. [26] and Ledig et al. [56]. As it follows, in this project we will explain SISR.

1.1.2.b Applications

It is an important class of image processing techniques in computer vision and image processing and enjoys a wide range of real-world applications, such as medical imaging, satellite imaging, surveillance and security, astronomical imaging, amongst others. In most digital imaging applications, HR images or videos are usually used for later image processing and analysis. As it is known, images are representations of the real world, and these are an enormous and very valuable source of information. The HR images have greater detail than LR because the higher the resolution, the more image details [Section 2.2]. For that, the contained information in HR or SR images is bigger, better, and even more useful. This information could be easily identified by humans but it is not always possible. Therefore, another application of these HR images is to help with representation for automatic machine perception. As we explained, this is a usual useful tool and it has several specific applications in our daily life [64]:

- ***Earth-observation remote sensing and Astronomical observation***: SR applied to satellite images [112][124][59][45][58].

- ***Medical Diagnosis***: as MRI, functional MRI, and PET. SR tries, as much as possible, to achieve true isotropic 3-D imaging [51][54].

- ***Surveillance and Security***: DVR for traffic [121] and security monitoring, as UAV surveillance video [127], and some biometric information identification [74].

- ***Regular video information enhancement***: 4K video games [65], HDTV [97], VR [21] etc.

1.2. Related Work

Deep learning has shown its superior performance in various computer vision tasks. With its rapid development in recent years, deep learning based on SR models has been actively explored. In general, the family of SR algorithms using deep learning techniques is very varied. These algorithms have different architectures, different types of loss functions or strategies, etc.

Below is shown in the [Table 1.2-I] some of the most popular proposed models for vision tasks. Today there are a variety of available datasets for SR images, depending of your interests you can choose it depending of the number of images, quality, resolution, pairs of LR-HR or only HR images, diversity, etc:

Table 1.2-I: List of public image datasets for super-resolution benchmarks [114].

Dataset	Amount	Avg. Resolution	Avg. Pixels	Format
BSDS300	300	(435; 367)	154; 401	JPG
BSDS500	500	(432; 370)	154; 401	JPG
DIV2K	1000	(1972; 1437)	2; 793; 250	PNG
General-100	100	(435; 381)	181; 108	BMP
L20	20	(3843; 2870)	11; 577; 492	PNG
Manga109	109	(826; 1169)	966; 011	PNG
OutdoorScene	10624	(553; 440)	249; 593	PNG
PIRM	200	(617; 482)	292; 021	PNG
Set5	5	(313; 336)	113; 491	PNG
Set14	14	(492; 446)	230; 203	PNG
T91	91	(264; 204)	58; 853	PNG
Urban100	100	(984; 797)	774; 314	PNG

In recent years, SR models based on deep learning have received more and more attention. In the [Table 1.2-II] below, we are going to describe most of the state-of-the-art SR models and some information about them, as upsampling methods applied, recursive and residual learning, dense connections, attention mechanism, etc.

Table 1.2-II: Super-resolution methodology employed by some representative models [114].

Method	Publication	Fw.	Up.	Rec.	Res.	Dense	Att.	L1	L2
SRCNN [27]	2014, ECCV	Pre.	Bicubic						√
DRCN [20]	2016, CVPR	Pre.	Bicubic	√	√				√
FSRCNN [28]	2016, ECCV	Post.	Deconv						√
ESPCN [10]	2017, CVPR	Pre.	Sub-Pixel						√
LapSRN [115]	2017, CVPR	Pro.	Bicubic		√			√	
DRRN [92]	2017, CVPR	Pre.	Bicubic	√	√				√
SRResNet [56]	2017, CVPR	Post.	Sub-Pixel		√				√
SRGAN [56]	2017, CVPR	Post.	Sub-Pixel		√				
EDSR [63]	2017, CVPRW	Post.	Sub-Pixel		√			√	
EnhanceNet [84]	2017, ICCV	Pre.	Bicubic		√				
MemNet [93]	2017, ICCV	Pre.	Bicubic	√	√	√			√
SRDenseNet [107]	2017, ICCV	Post.	Deconv		√	√			√
DBPN [66]	2018, CVPR	Iter.	Deconv		√	√			√
DSRN [42]	2018, CVPR	Pre.	Deconv	√	√				√
RDN [125]	2018, CVPR	Post.	Sub-Pixel		√	√		√	
CARN [2]	2018, ECCV	Post.	Sub-Pixel	√	√	√		√	
MSRN [57]	2018, ECCV	Post.	Sub-Pixel		√			√	
RCAN [126]	2018, ECCV	Post.	Sub-Pixel		√		√	√	
ESRGAN [113]	2018, ECCVW	Post.	Sub-Pixel		√	√		√	
RNAN [127]	2019, ICLR	Post.	Sub-Pixel		√		√	√	
Meta-RDN [46]	2019, CVPR	Post.	Meta Upscale		√	√		√	
SAN [18]	2019, CVPR	Post.	Sub-Pixel		√		√	√	
SRFBN [68]	2019, CVPR	Post.	Deconv	√	√	√		√	

2. THEORETICAL FRAMEWORK

2.1. Deep Learning

In recent years, deep learning has started to take a lot of importance inside the computational field and in many others where it has been applied too.

First, to understand what deep learning is and how it works, we need to define two essential concepts: artificial intelligence and machine learning.

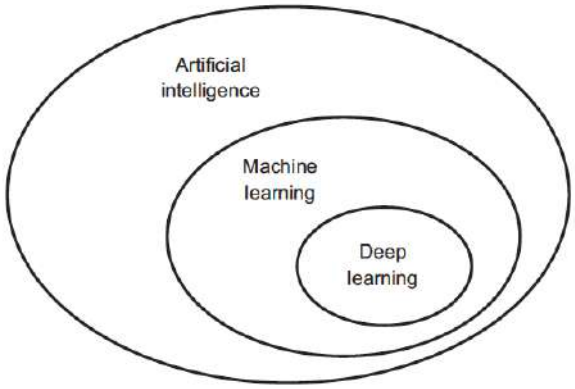

Fig. 2.1. Artificial Intelligence, machine learning, and deep learning [12].

2.1.1 Artificial Intelligence and Machine Learning

Artificial intelligence was born in the 1950s when the main unknown subject from the field of computer science was focused on the idea of machines self-thinking. A definition could be like François Chollet [12] wrote: "*the effort to automate intellectual tasks normally performed by humans.*" AI is a general field composed of machine learning and deep learning, among many others.

The first approach in this field is known as symbolic AI, very popular between the 1950s and 1980s, when programmers designed hardcoded rules and outcome answers, the perfect method to solve some logical problems like chess. This was the beginning of a new and more difficult line of research known as machine learning, a scientific discipline within the field of AI.

A definition of machine learning would be as follows: *a data analysis method that automates procedures of analytic model construction, based on the idea of machines self-thinking, with minimal human involvement during the process* [12]. The interest in this field was growing and with it, more concerns and questions emerged. They wanted something more than just a tool to help us do whatever we already know, but in a simpler way.

The main difference between symbolic AI and machine learning is that in the first one, we use machines like a tool to assist us, doing things that we already know but easier and faster. However, the purpose of machine learning is that computers can learn from their own experience, that is to say, by themselves. For that, the computer will receive the data and the answers expected from that data, and the outcome will be the rules; these rules can be applied to new data to produce original answers.

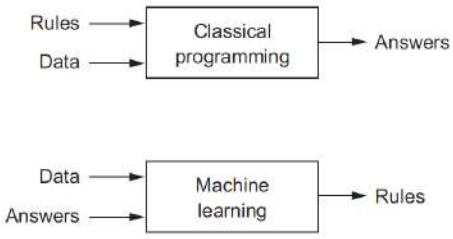

Fig. 2.1.1 Symbolic IA vs Machine Learning: the birth of a new era [12].

Inside machine learning, there are different types of learning techniques depending on its knowledge acquisition and task improvement [91][14]:

- ***Supervised learning***: is the most common form of machine learning. It starts from pairs of data/solution. It works with labeled data and tries to find a function or model to label the input data correctly. The purpose is to predict the output label from historic data. Some algorithms that use supervised learning are *Decision trees, Naïve Bayes, Linear and Logistic Regression, K-Nearest Neighbor, and SVM.*

- ***Unsupervised learning***: works with not labeled data, without the help of any targets. During the training, the algorithm adapts its own characteristics to understand and interpret the information with no human interaction. Some algorithms that use unsupervised learning are *Clustering methods, Anomaly detection, Neural Networks, and Approaches for learning latent variable models.*

Starting from this, we could define deep learning as a set of ML algorithms, an artificial learning method that tries to imitate human behaviour classifying and identifying patterns to improve and learn by itself.

The term deep learning emerges in the 1980s but till the 2010s it didn't reach its peak because of technology limitations. The current era, the era of the internet and big data, the perfect ally, create an ideal ecosystem for DL development and make it one of the most common tools used nowadays.

2.1.2 Artificial Neural Network

How we have explained before, deep learning is a subfield inside machine learning which tries to imitate and reproduce human brain behaviour. It is a set of techniques and algorithms whose final purpose is to obtain abstract models able to identify and extract data features by non-linear mathematical transformations.

2.1.2.a Perceptron

As in human brains, an ANN is based on a collection of connected units or nodes called artificial neurons, perceptrons [128]. ***Perceptron*** is the smallest and simplest network, it is known as 1-layer NN, and it was created by the American psychologist Frank Rosenblatt in 1957s [32].

This artificial neuron consists of some inputs n_1 and n_2, with each weight parameter w_n and the final output n_f. Inside the perceptron, there is an activation function that produces the outcome.

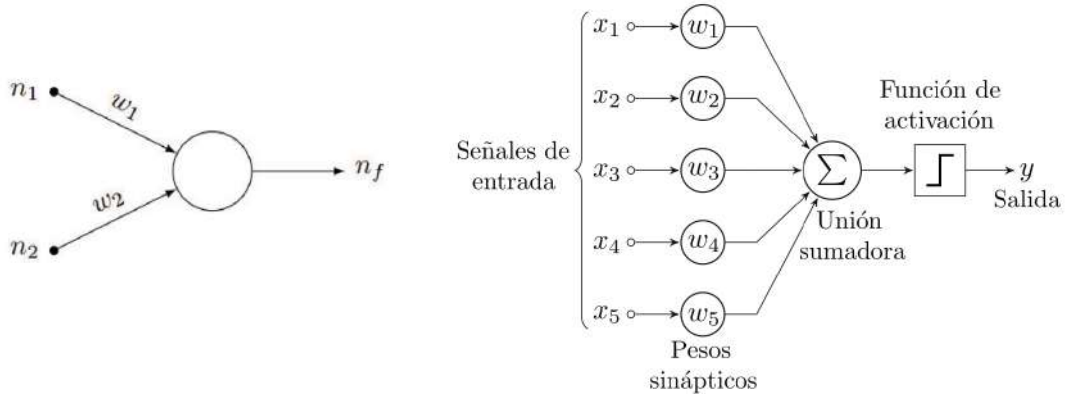

Fig. 2.1.2.a-I Perceptron behaviour [73].

Each input multiplies with each weight. Then, the input for the activation function is the sum of its inputs and depending on this value there will be a particular outcome. The choice of activation function in the output layer will define the type of predictions the model can make.

An ***activation function*** in a neural network defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network [89][88][14]. This choice has a large impact on the capability and performance of the neural network. Different activation functions may be used in different parts of the model.

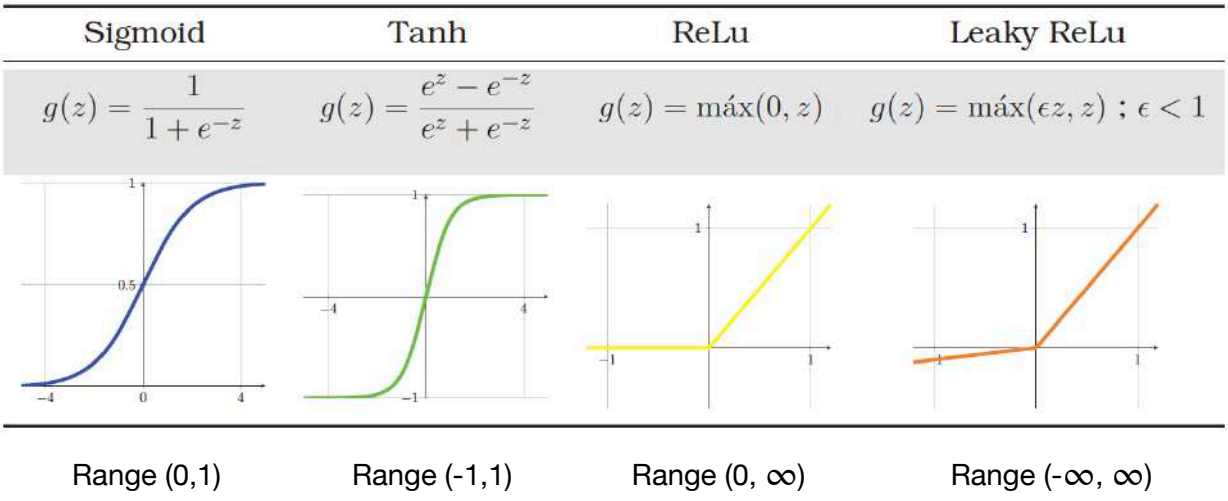

Fig. 2.1.2.a-II Activation functions: Sigmoid, Tanh, ReLu, and Leaky ReLu [33].

We could define the perceptron as two equations where x is the input, w is its weight, and $g(z)$ the activation function outcome:

$$z = x * w$$
$$y = g(z)$$

Perceptron can only solve separable linear functions but this limitation was overcome thanks to ANN. The connection between different layers of neurons allows representing more complex functions.

2.1.2.b Multilayer Perceptron

We can understand ANN as François Chollet [12]: "*like a multistage information-distillation operation, where information goes through successive filters and comes out increasingly purified*". Its final purpose is to do the input-to-target proceed correctly, with a deep sequence of simple data transformations that learn by showing examples. In these ***multilayer perceptrons*** [12][128], the outcome of the activation function will send to deeper levels.

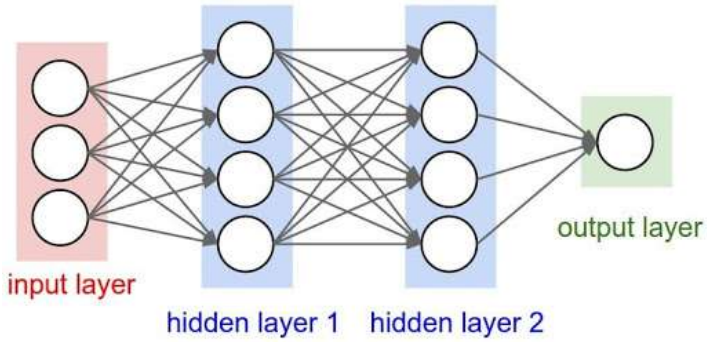

Fig. 2.1.2.b-I Example of the structure of an ANN [72].

First of all, we are going to define the most important components of the ANN structure [5][14]:

- ***Input layer***: the data is in this layer. Its function is to receive the input values to feed the network with them. The nodes in this step are passive, meaning they don't change the data. From the input layer, it duplicates each value and send them to all the hidden nodes.

- ***Hidden layer***: inside these layers, there are neurons that apply given transformations to the input values. These layers are connected with the output layer or another hidden layer, with weighted connections. The values entering a hidden node are multiplied by weights, then these values are added to produce a single number.

- ***Output layer***: this layer receives connections from hidden layers or directly from the input layer. After combining and changing the input, it returns an output value that corresponds to the prediction of the response variable.

As we explained, ANNs often involve tens or even hundreds of successive layers of representations [12][14]. In each layer, there are neurons connected between them, and each connection has a particular weight. ***Weights*** [116] are a bunch of numbers that stores the layer's specifications. The final purpose is to find the best values for these ***parameters*** [3] to obtain the best input-target relationship.

As we mentioned, a deep neural has a lot of layers and for each layer there are hundred of neurons and eventually, the network can contain tens of millions of parameters. Finding the value of the correct weight is a complex decision, since modifying the value of one parameter will affect the behaviour of all the others.

These parameters directly influence the final result, the output. We want to measure the quality of our network, i.e. how similar is our outcome to the real value. This is the job of the ***loss function*** [12] of the network. Taking the predictions of the network and the true target to calculate the difference between them. The network uses this score like feedback to cut this distance and update the value of the weights. This adjustment is the labor of the ***optimizer*** [77] thanks to the ***backpropagation*** [7][39] algorithm.

Initially, the weights of the network are randomly assigned, which implies that the loss score is very high at the beginning. During the training loopings, transformations are applied in the direction of a weights reduction, i.e. a loss score decrease, until the learning algorithm converges to an acceptable error approximation. Finally, the network achieves the final purpose, the similarity or closeness between the real targets and the outputs from the trained network.

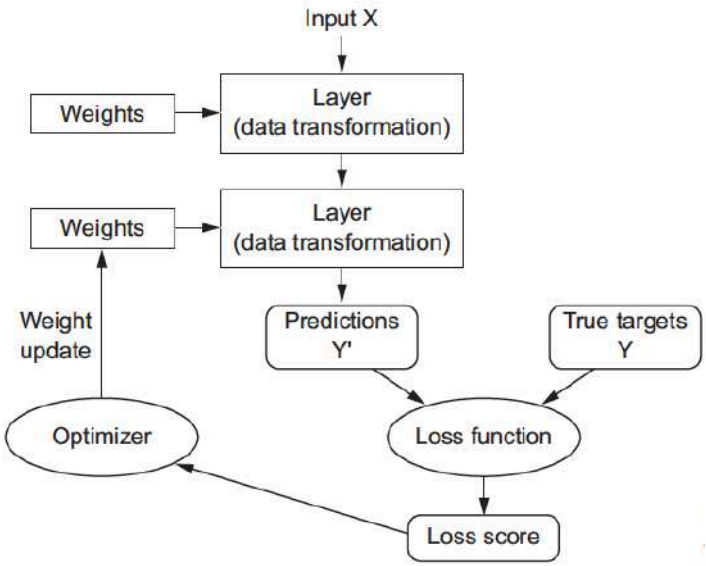

Fig. 2.1.2.b-II ANN: Weights, Loss function, Optimizer and Backpropagation [12].

2.1.3 Convolutional Neural Network

CNN is a type of multilayer perceptrons, there are very useful to artificial vision tasks. When we work with images, every pixel is the input so finally, there are a huge amount of inputs unmanageable to a simple network. Here is the beginning of a new DNNs alteration [83][14].

These networks consist of two parts, ***feature extraction***, and ***classification***. In the first part, the feature learning extracts the most important features or characteristics like edges, patrons, or shapes. This first part uses some filters in the search of characteristics, these filters are convolutions. Then, some methods are used to reduce the size or dimensions of these, but keeping these characteristics. This method of reduction, the pooling layer, is usually applied after a convolutional layer. In the second phase, classification is applied.

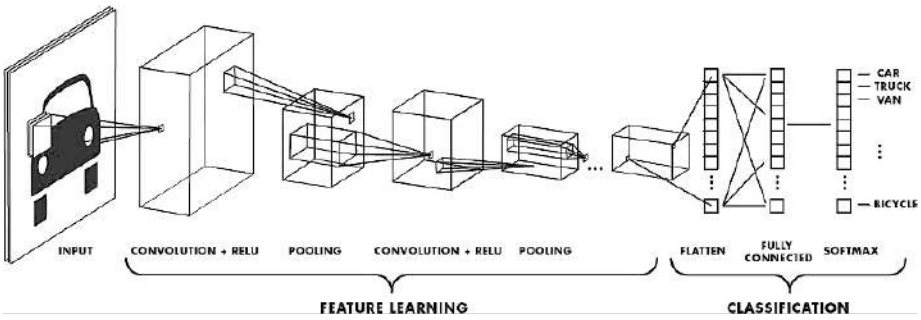

Fig. 2.1.3-I Convolutional Neural Networks [13].

To understand convolutional layers, we have to introduce a new term, the ***kernel***. These layers need two matrices, the image where we want to apply the convolution, and a smaller matrix called kernel. The result of the convolutional layer is another matrix with the input size, i.e. with the same dimensions as the original image. The kernel is applied for each element from the input matrix and then, the result is the sum of the products between both matrix elements. The kernel values start randomly and update with the backpropagation.

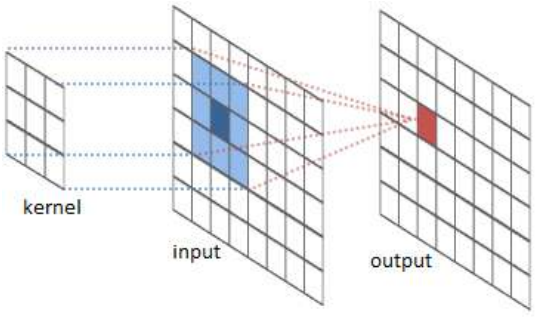

Fig. 2.1.3-b Kernel behaviour.

2.1.3.a Important Architectures

- ***VGGNet*** [14][110]: The runner-up in ILSVRC 2014 was the network from Karen Simonyan and Andrew Zisserman that became known as the VGGNet. Its main contribution was in showing that the depth of the network is a critical component for good performance. Their final best network contains 16 CONV/FC layers and, appealingly, features an extremely homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from the beginning to the end. It is more expensive to evaluate and uses a lot more memory and parameters. Most of these parameters are in the first FC layer, and it was found that these FC layers can be removed with no performance downgrade, significantly reducing the number of necessary parameters.

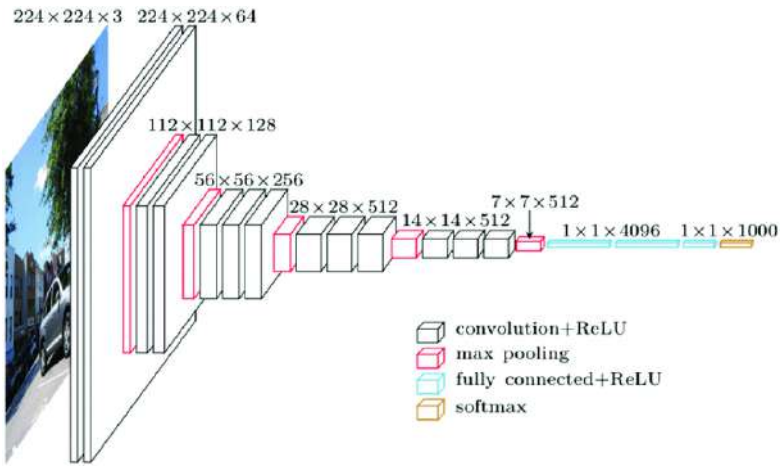

Fig. 2.1.3.a-I VGG16 Architecture.

- ***ResNet*** [14][1]: developed by Kaiming He et al., it was the winner of ILSVRC 2015. It features special skip connections and a heavy use of batch normalization. The architecture is also missing fully connected layers at the end of the network. ResNets are currently by far state of the art on Convolutional Neural Network models and are the default choice for using ConvNets in practice.

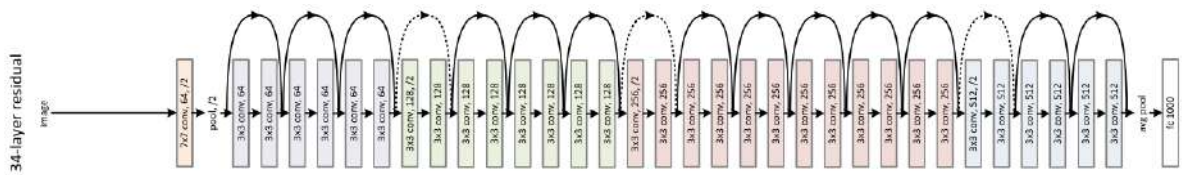

Fig. 2.4.3.a-II ResNet Architecture.

2.2. Digital Image

A digital image [48][70] is the 2D representation of a 3D object from the real world. It could be defined as a bidimensional function $f(x, y)$ where (x, y) are the spatial coordinates and the value of the function for each coordinate is the intensity of the image at that point.

A digital image is a bidimensional representation expressed in a numerical matrix form, usually binary [0,1]. Depending on if the image resolution is dynamic or static, it can be a vectorial graphic or a matrix image or what is the same, a map of bits or bitmap images. In this project, we will work with bitmap images. Below, we are going to introduce some of the most basic concepts about digital images [6][34].

2.2.1 Main concepts

- ***Brightness***: Brightness or intensity is the amount of radiation received by the object. It is recorded as a digital number, stores with a finite number of bits or binary digits.

- ***Contrast***: It is the difference between brightness and darkness.

- ***Dynamic range***: It shows the amount of brightness that it is able to represent. It is the difference between the maximum and minimum brightness value.

- ***Image resolution***: It is the detail an image holds. Resolution quantifies how close lines can be to each other and still be visible the distance or gap between both lines. Higher resolution means more image detail and smaller resolution size.

- ***Pixel***: In digital imaging, a pixel or picture element is the smallest item of information in an image. Pixels are arranged in a bidimensional grid, represented using squares. Typically, more pixels provide more accurate representations and heavier files. A pixel has an intensity value and a location address in the two dimensional image.

- ***Spatial resolution***: The spatial resolution refers to the size of the smallest object that can be resolved on the ground. In a digital image, the resolution is limited by the pixel size.

- ***Radiometric resolution***: The radiometric resolution of an imaging system describes its ability to discriminate very slight differences in energy. The finer the radiometric resolution of a sensor, the more sensitive it is to detecting small differences in reflected or emitted energy.

- ***Color or Bit depth***: is determined by the number of bits used to define each pixel. A color image is typically represented by a bit depth ranging from 8 to 24 or higher. With a 24-bit image, the bits are often divided into three groups, one for each channel: 8 for red, 8 for green, and 8 for blue. A 24-bit image offers 16.7 million or what is the same 2^{24} values and a range of [0, 255] to each channel.

- ***Size***: We can distinguish between ***pixel size***, which is the sum of all the pixels from the image; ***physical image size***, depends on the pixel size image and the resolution, it is given in centimeters; and ***weight of the image***, refers to the amount of memory used to store it, this is expressed with the number of bytes.

- ***Format and image storage***: The compression of images can imply a lost of information. Some of the most used formats are: *.jpg*, *.png*, *.tiff*, and *.gif*.

Table 2.2.1 Formats and Image Storage [23].

Image Format	Color Model	Transparency	Destination	Remarks
JPG	RGB	-	Web/Screen	Generational degradation
TIFF	RGB/CMYK	Yes	Printing	Layered images, image stacks
GIF	RGB	Yes	Web/Screen	Limited color, animated images
PNG	RGB	Yes	Web/Screen	Lossless comprenssion

File Format	Color Model	Transparency	Destination	Remarks
SVG	RGB	Yes	Web/Screen	Interactive, scriptable
EPS	RGB/CMYK	Yes	Printing	PostScript document
PDF	RGB/CMYK	Yes	Web/Screen + Printing	Includes PostScript, platform independent

2.2.2 Image quality metrics

Any processing applied to an image may cause an important loss of information or quality. In this project, we want to study and test different methods of re-sampling. For that, we are going to quantify them, since some of the obtained results are very similar to the human eye and this is a subjective procedure. In this case, we work with two of the most used methods, PSNR and SSIM [86][44].

2.2.2.a Peak signal to noise ratio - PSNR

PSNR is one of the most used quality measurement tools applied to image and signal processing. PSNR is the result of the logarithm of the mean square error of an image, where MSE traditionally uses the summation method as its main component. The PSNR value approaches infinity as the MSE approaches zero, i.e. a higher PSNR value provides a higher image quality.
In MSE grayscale images are calculated based on $M \times N$ dimensions, whereas in RGB images can be calculated based on $M \times N \times O$ dimensions:

$$MSE = \frac{1}{M \times N \times O} \sum_{x=1}^{M} \sum_{x=1}^{N} \sum_{x=1}^{O} \left[\left(I_{(x,y,z)} - I'_{(x,y,z)} \right)^2 \right]$$

Where M and N are image resolution, i.e. the height and the width; O is the number of image channels, in this case three because it is a RGB image; $I_{(x,y,z)}$ is the pixel value of the original image at the x, y coordinates and channel z; I' is the output image/processing result. MSE has a close relationship with PSNR because the MSE value is used to calculate the PSNR value:

$$PSNR = 10log10\left(\frac{\max^2}{MSE}\right)$$

The PSNR is composed of the error squared value as the main component and the highest scale value, the max. PSNR does not change the pixel value, it calculates the differences between two images, the PSNR will be smaller when this difference is bigger and vice versa.

2.2.2.b Structural similarity index - SSIM

The SSIM is a well-known quality metric used to measure the similarity between two images. It was developed by Wang et al. [114], and was built based on three main factors: luminance, contrast, and structure. These three factors replace the summation method, the MSE, used for calculating PSNR. The SSIM is defined as:

$$SSIM(i, i') = l(i, i')c(i, i')s(i, i')$$

$$l(i, i') = \frac{2\mu_i\mu_{i'} + C_1}{\mu_i^2 + \mu_{i'}^2 + C_1}$$

$$c(i, i') = \frac{2\sigma_i\sigma_{i'} + C_2}{\sigma_i^2 + \sigma_{i'}^2 + C_2}$$

$$s(i, i') = \frac{\sigma_{ii'} + C_3}{\sigma_i\sigma_{i'} + C_3}$$

The first term l is the luminance comparison function which measures the nearness between both mean luminance, μ_i and $\mu_{i'}$. The maximum value of this factor is 1 and it's obtained when $\mu_i = \mu_{i'}$.

The second term c is the contrast comparison function which compares the contrast of two images. For that, this function uses the standard deviation, σ_i and $\sigma_{i'}$. The maximum value of this factor is 1 and this happens only when $\sigma_i = \sigma_{i'}$.

The third term s is the structure comparison function which measures the correlation coefficient between the two images i_i and i'. Being $\sigma_{ii'}$ the covariance between i_i and i'. SSIM range is [0,1], 0 means no correlation and 1 means that $i = i'$.

C_1, C_2 and C_3 are positive constants that are used to avoid null denominators.

2.3. Interpolation Methods

The interpolation [34] is a mathematical procedure whose final purpose is to estimate new data from already known data. In this case, we are talking about interpolation methods applied to digital images. We want to know how to improve the image quality from the LR image by applying some efficient mathematical methods as interpolation.
Interpolation algorithms can be classified as: ***Adaptative***, detects local spatial features and makes effective choices depending on the algorithm; and ***Non-Adaptative***, perform interpolation in a fixed pattern for every pixel [67][71].

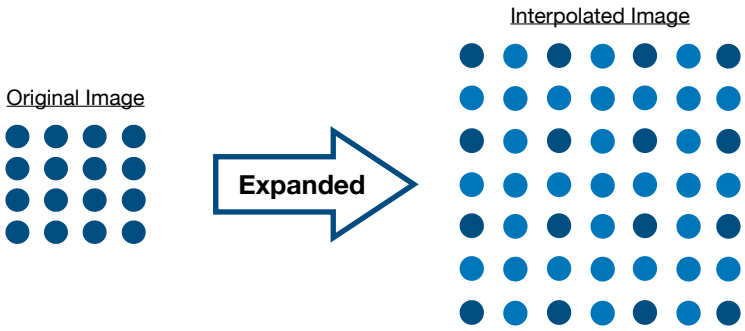

Fig. 2.3 Image Interpolation.

In this case, we'll discuss some of the most used Non-Adaptative interpolation algorithms, such as Nearest Neighbor, Bilinear, Bicubic. In addition, we will explain a fourth technique, Lanczos interpolation [29][22].

2.3.1 Nearest neighbor

This method is the simplest technique as it involves little calculation. The NNI resamples the pixel values which are present in the input vector or a matrix and round to the nearest integer. The result usually is a blurred or pixelated image, it gives poor quality image.

The interpolation kernel is given as:

$$u(x) = \begin{cases} 0, & |x| > 0.5 \\ 1, & |x| < 0.5 \end{cases}$$

Where x is the distance between interpolating point and grid point.

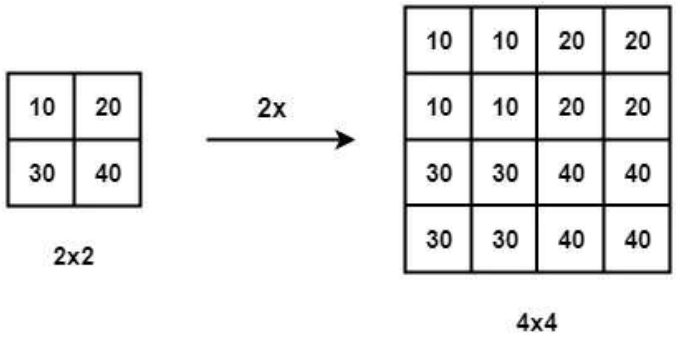

Fig. 2.3.1 Example of Nearest Neighbor Interpolation.

2.3.2 Bilinear

Linear interpolation estimates the value using linear polynomials, so the bi-linear method applies a linear interpolation in two directions, i.e. in both orthogonal directions of the image, x and y, or rows and columns. It uses the distance-weighted average of the four nearest pixel values to estimate a new pixel value.

Interpolation kernel of it for each direction is:

$$u(x) = \begin{cases} 0, & |x| > 1 \\ 1 - |x|, & |x| < 1 \end{cases}$$

Where x is the distance between interpolating point and grid point.

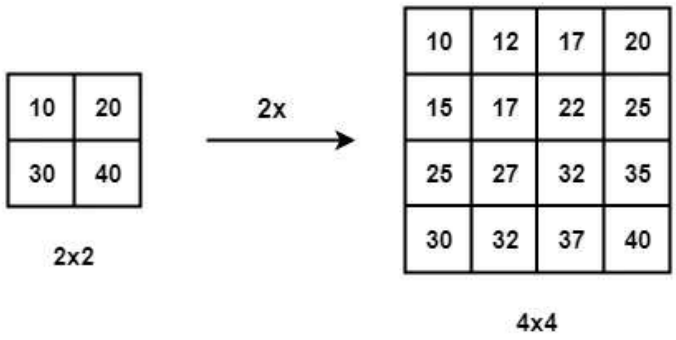

Fig. 2.3.2 Example of Bilinear Interpolation.

2.3.3 Bi-cubic

Bicubic interpolation is the most used non-adaptive technique. It uses a weighted average of sixteen pixels to calculate its final interpolated value. These pixels are at various distances from the unknown pixel. Bi-linear uses four nearest neighbors to determine the output, while bicubic uses sixteen, 4×4 neighbourhood.

Although it gives better results, sharper images but it takes more computational time than other two non-adaptive methods. The bicubic interpolation kernel is:

$$u(x) = \begin{cases} (a+2)|x|^3 - (a+3)|x|^2 + 1, & |x| \le 1 \\ a|x|^3 - 5a|x|^2 + 8a|x| - 4a, & 1 < |x| \le < 2 \\ 0, & otherwise \end{cases}$$

Where x is the distance between interpolating point and grid point and a is generally taken as *-0.5* to *-0.75*.

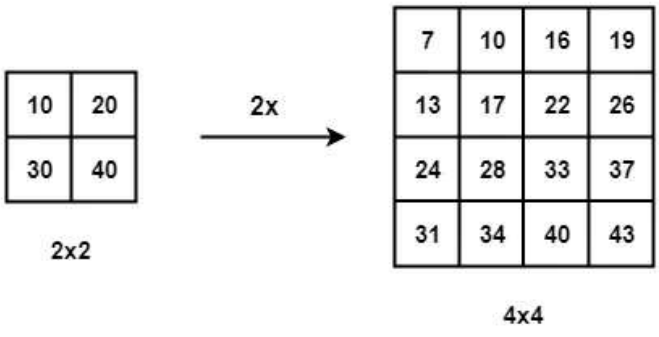

Fig. 2.3.3 Example of Bicubic Interpolation.

2.3.4 Lanczos

Lanczos interpolation is an application of a mathematical formula used like a low-pass filter, attenuating high frequencies and passing the lower than the threshold. This method is typically used to increase or change the sampling rate of a digital signal. Lanczos filter is used to interpolate the value of a digital signal between its samples. Lanczos kernel is the normalized sinc function, visioned by the Lanczos window, or sinc window:

$$u(x) = \begin{cases} Sinc(x)sinc(\frac{x}{a}), & -a < x < a \\ 0, & otherwise \end{cases}$$

Or which is the same,

$$u(x) = \begin{cases} 1, & x = 0 \\ \frac{asin(\pi x)sin(\frac{\pi x}{a})}{\pi 2 \times 2}, & 0 < |x| < a \\ 0, & otherwise \end{cases}$$

Where the parameter a is a positive integer, which value usually is *2* or *3*, and determines the size of the kernel. The kernel has $2a - 1$ lobes, a positive one at the centre and $a - 1$ alternating negative and positive lobes on each side.

The interpolation formula is obtained by the discrete convolution of some samples with the Lanczos kernel:

$$S(x) = \sum_{i=\lceil x \rceil - a + 1}^{\lceil x \rceil + a} s_i L(x - i)$$

Being s_i samples of one-dimensional signal, $\lceil x \rceil$ is a floor function and a is the filter size parameter.

Lanczos interpolation uses a neighborhood of the $2n \times 2n$ nearest mapped pixels, in this case we will use Lanczos-4, i.e. 8×8 pixel neighborhood.

2.4. Generative Adversarial Network

The SR-GAN was proposed by a group of investigators from Twitter in 2017 [56]. This is a model or architecture derived from the GANs.

GANs were introduced in 2014 by Ian J. Goodfellow and colleagues in the article Generative Adversarial Nets [38]. This model combines DL with the theory of games. GANs consist of two models simultaneously trained: the generator, trained to generate data; and the discriminator, which function is to distinguish between real and fake data, i.e. between real examples and data generated by the generator. As training progresses, both models improve. Generator tries to optimize the generated data to fool the discriminator.

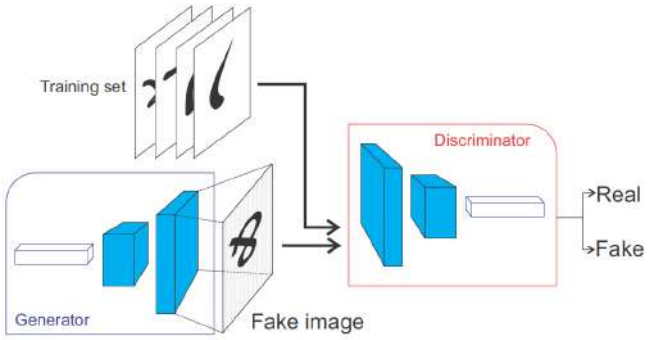

Fig. 2.4-I GAN Architecture.

GANs have many different applications and there are many papers about it [33][119] [41][11]. As the case follows, we are going to talk about the application of these models to improve imaging resolution. In SR-GANs the purpose is to generate HR images from LR. This process of resolution improving can be:

- ***Multi Image SR***: It uses multiple LR images to generate the final HR image.

- ***Single Image SR***: The HR image is generated from a unique LR image.

The SRGAN architecture is very complex, with different models and blocks that we will explain in the following points.

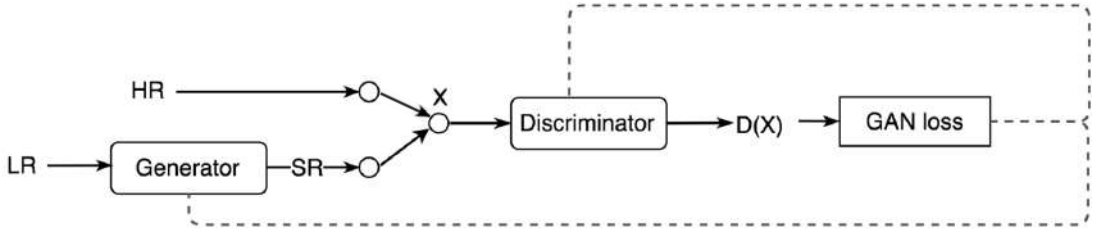

Fig. 2.4.b SR-GAN Architecture.

2.4.1 Generator

The generator [56][38] is a model inside the SR-GAN, and its goal is to create data, in this case, HR images from LR images.

The generator architecture contains residual networks instead of deep convolution networks. These are easier to train and allow them to be substantially deeper and to generate better results.

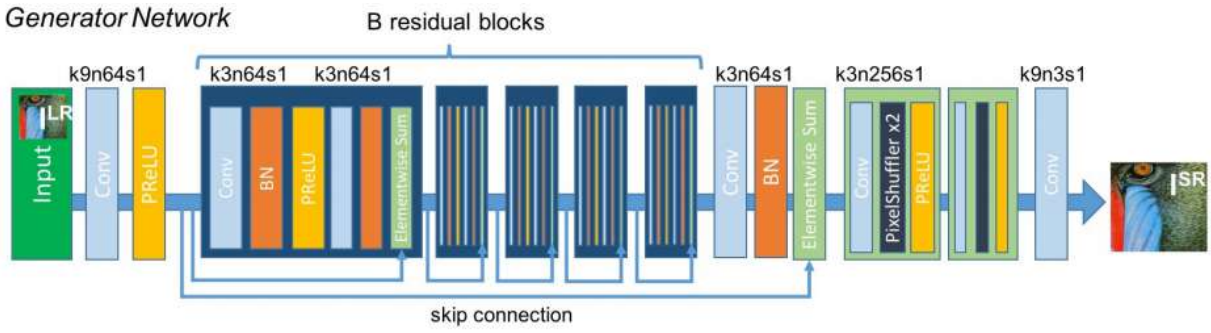

Fig. 2.4.1 Generator Architecture.

The input of the generator is the LR image. The architecture starts with a convolutional layer with a 9×9 kernel, 64 feature maps, and the ParametricReLU as the activation function. It uses PReLU instead of LeakyReLU, because it adaptively learns the parameters of rectifier and improves the accuracy without any extra computational cost.

There are *B* residual blocks, in this case, 16, originated by ResNet [1]. Within the residual block, uses two convolutional layers, with small 3×3 kernels, and 64 feature maps followed by batch-normalization layers and the PReLU activation function.

The residual network uses a type of connections called skip connections or shortcuts. The resolution of the input image is increased with two trained sub-pixel convolution layers.

2.4.2 Discriminator

The task of the discriminator [56][38] is to discriminate between real HR images and generated SR images. The network contains eight convolutional layers with kernel of 3×3, increasing by a factor of 2 from 64 to 512, i.e. 64, 128, 256 and 512 [110].

Strided convolutions are used to reduce the image resolution each time the number of

features is doubled. The resulting 512 feature maps are followed by two dense layers and a LeakyReLU. At the end, sigmoid activation function is used to obtain a probability for sample classification.

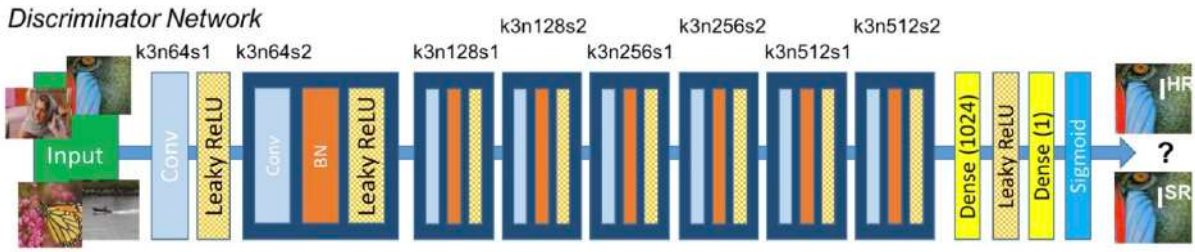

Fig. 2.4.2 Discriminator Architecture.

2.4.4 Loss Function

2.4.4.a Generator Loss

The perceptual loss function [56] is what characterizes and differentiates the SR-GAN model from other super-resolution models. This is the weighted sum of two loss components: content loss and adversarial loss.

$$l^{SR} = l_X^{SR} + 10^{-3} l_{Gen'}^{SR}$$

Being l^{SR} the total or final loss, l_X^{SR} the content loss, and $l_{Gen'}^{SR}$ the adversarial loss.

- ***Content Loss***: is defined as: the pixelwise MSE loss, differences between pixels; and the VGG loss, referred to differences between high characteristics from the image.

$$l_X^{SR} = l_{MSE}^{SR} + l_{VGG/i,j}^{SR}$$

Where l_{MSE}^{SR}, the pixelwise MSE loss is defined as:

$$l_{MSE}^{SR} = \frac{1}{r^2 WH} \sum_{x=1}^{rW} \sum_{y=1}^{rH} (I_{x,y}^{HR} - G_{\theta_G}(I^{LR})_{x,y})^2$$

Being r a scale factor, H and W the height and the weight of the image, respectively. G_{θ_G} is referred to the generator, I^{LR} and I^{HR} refer to low and high resolution image. So $G_{\theta_G}(I^{LR})$ is the result of applying the LR image like the input to the generator model, i.e. the super resolution image generated.

And $l^{SR}_{VGG/i,j}$, the VGG loss is defined as:

$$l^{SR}_{VGG/i,j} = \frac{1}{W_{i,j}H_{i,j}} \sum_{x=1}^{W_{i,j}} \sum_{y=1}^{H_{i,j}} (\phi_{i,j}(I^{HR})_{x,y} - \phi_{i,j}(G_{\theta_G}(I^{LR}))_{x,y})^2$$

Where $\phi_{i,j}$ is the feature map obtained by the j-th convolution, after the activation and before the layer i of max-pooling from the VGG network of the pre-training. The H and W are the height and the weight of the map feature of the VGG network. The loss $l^{SR}_{VGG/i,j}$ express the euclidean distance between the feature map of the image with the generated and the real image.

- ***Adversarial Loss***: is the loss function that forces the generator to generate more similar images to HR images. For that, it uses discriminator feedback.
As we have explained, the generator architecture tries to upsample the image from LR to SR. After that, the image passes into the discriminator that tries to distinguish between a SR and HR image. Finally, it generates the adversarial loss which is backpropagated into the generator architecture.

$$l^{SR}_{Gen} = \sum_{n=1}^{N} -logD_{\theta_D}(G_{\theta_G}(l^{LR}))$$

Where $G_{\theta_G}(l^{LR})$ is the image produced by the generator, and $D_{\theta_D}(G_{\theta_G}(l^{LR}))$ is the probability that the image is generated or real, according to the discriminator. Both models will improve, and the generator will be creating more realistic images.

2.4.4.b Discriminator Loss

To train the discriminator, the loss function uses the typical GAN discriminator loss as Goodfellow et al. defined [38]:

$$\min_{\theta_G} \max_{\theta_D} \mathbb{E}_{I^{HR}\sim p_{train}(I^{HR})}[logD_{\theta_D}(I^{HR})] + \mathbb{E}_{I^{LR}\sim p_G(I^{LR})}[log(1 - D_{\theta_D}(G_{\theta_G}(I^{LR})))]$$

2.5. Feedback Network

The SRFBN was proposed in BMVC2019 by Sichuan University, University of California, University of British Columbia, and Incheon National University [68].
SRFBN seeks a solution to the scarce information received and used by the low-level features, a possible limitation for the ability of the network. To solve this problem, in SRFBN the high-level information flows through feedback connections in a topdown manner to correct low-level features using more contextual information. In addition, the storage problem because of the number of parameters used during the training is considerably reduced with the recurrent layers.

2.5.1 SRFBN Structure

The SRFBN was proposed by Li et al. in 2019 [61], and it comes with a strong early reconstruction ability and can create the final high-resolution image step by step. The principle of the feedback scheme is that the information of a coarse SR image can facilitate an LR image to reconstruct a better SR image. This network has a strong influence of the paper written for CVPR17 by Zamir et al. [123].

The SRFBN structure has *T* iterations, in which each of it *t* is a sub-network ordered from 1 to T. The sub-network placed in each iteration contains three parts: an LR feature extraction block, ***LRFB***; a feedback block, ***FB***; and a reconstruction block, ***RB***. The weights of each block are shared across time.

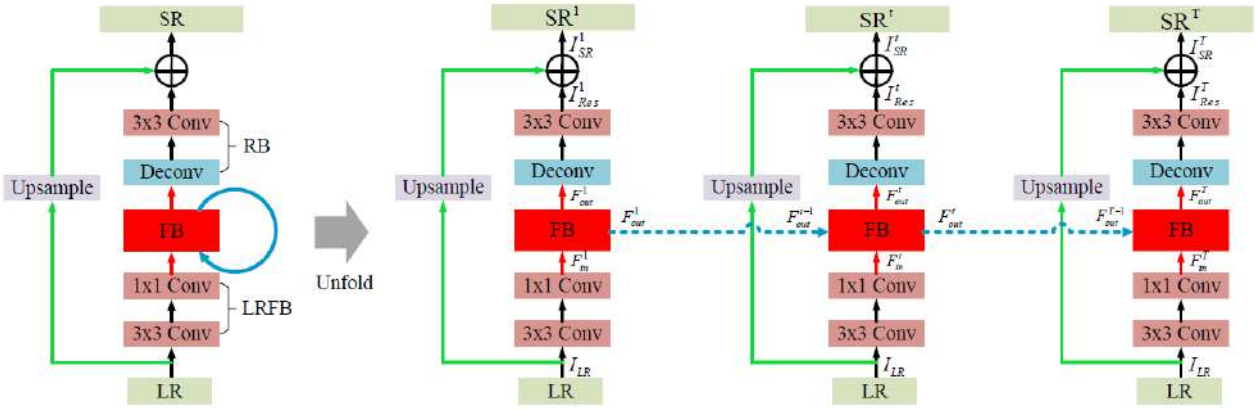

Fig. 2.5.1 SRFBN Architecture.

- ***The LR feature extraction block***: consists of *Conv(3,4m)* and *Conv(1, m)*, where *m* defines the base number of filters. The input image of LR is expressed as I_{LR} from which we obtain the shallow features F_{in}^{t}. The f_{LRFB} denotes the operations of the LR feature extraction block.

$$F_{in}^{t} = f_{LRFB}(I_{LR})$$

- ***The feedback block***: at the *t*-th iteration receives the hidden state from previous iteration F_{out}^{t-1} through a feedback connection and shallow features, F_{in}^{t}. F_{out}^{t} represents the output of the FB. The f_{FB} denotes the operations of the FB and actually represents the feedback process.

$$F_{in}^{t} = f_{FB}(F_{out}^{t-1}, F_{in}^{t})$$

- ***The reconstruction block***: uses *Deconv(k, m)* to upscale LR features F_{out}^{t} to HR, and *Conv(3, c_{out})* to generate a residual image I_{Res}^{t}. The c_{out} refers to the number of channels, in this case will be 3. Where f_{RB} denotes the operations of the reconstruction block.

$$I_{Res}^{t} = f_{RB}(F_{out}^{t})$$

The output image I_{SR}^{t} at the t-th iteration can be obtained by:

$$I_{SR}^{t} = I_{Res}^{t} + F_{UP}(I_{LR})$$

Where f_{UP} denotes the operation of an upsample kernel. The choice of the upsample kernel is arbitrary. We use a bilinear upsample kernel here.

After T iterations, we will get totally T SR images $(I_{SR}^{1}, I_{SR}^{2}, \ldots, I_{SR}^{T})$.

2.5.2 Curriculum learning strategy

Curriculum learning [61] describes a kind of learning similar to the strategy followed by humans. In this type of learning the beginning starts out with easy practices or examples increasing it gradually, to finally ending up with the most difficult task. The curriculum is based on the recovery difficulty.

In this case, we optimize the training with L1 Loss and in each iteration, each output has equal contribution as [123] do.

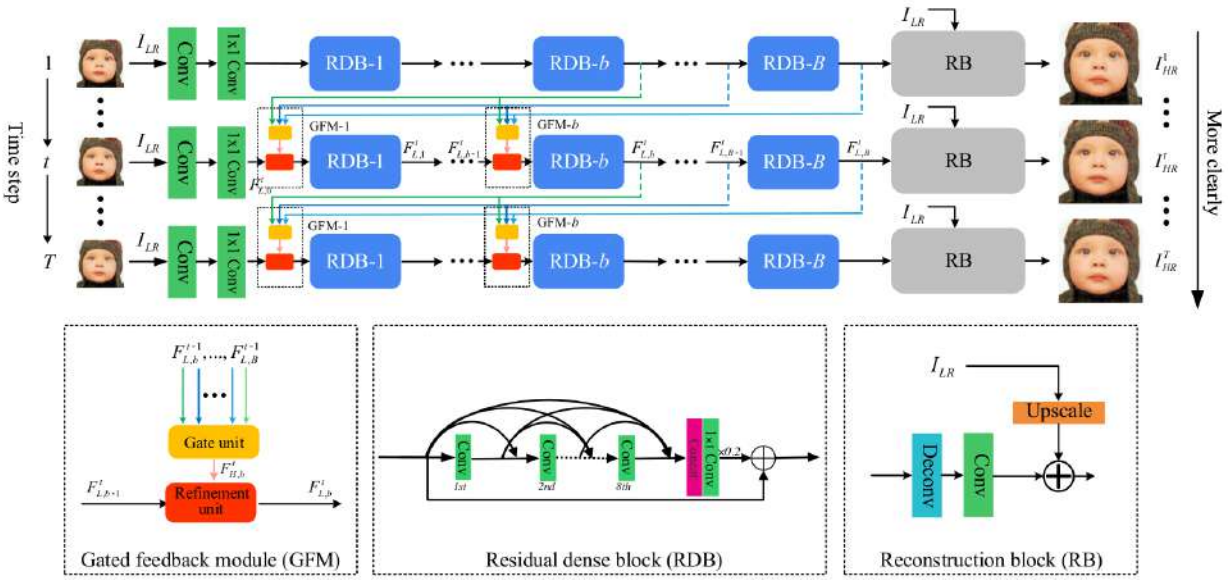

Fig. 2.5.2 Curriculum Learning in SRFBN.

3. TOOLS

3.1. Materiales

This chapter details the hardware and software tools used during the development of this project.

3.1.1 Hardware

In terms of hardware, we used two remote computers to carry out the experiments, and a third one to search for information and test the trained models.
The characteristics of the equipment used to train SR-GAN model defined in [Section 2.4]:

Table 3.1.1-I: Specifications of the PC used in SR-GAN training.

Specifications	Types
Central Processing Unit (CPU)	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
Motherboard (MB)	ASUSTeK COMPUTER INC. B85M-G
Random Access Memory (RAM)	4x8GB 1333MHz memories in DIMM (dual in-line memory module)
Graphics Processing Unit (GPU)	NVIDIA GeForce RTX 2080 Ti
Solid disk storage (SSD)	Crucial MX500 500GB

The characteristics of the equipment used to train SRFBN model defined in [Section 2.5]:

Table 3.1.1-II: Specifications of the PC used in SRFBN training.

Specifications	Types
Central Processing Unit (CPU)	Intel(R) Core(TM) i9-10850 CPU @ 3.60GHz
Motherboard (MB)	HP 8703
Random Access Memory (RAM)	64GB 3200MHz memories in DIMM (dual in-line memory module)
Graphics Processing Unit (GPU)	NVIDIA GeForce RTX 3080
Solid disk storage (SSD)	WDC WD BLACK SDBPNTY 1TB

The GPU is the key hardware component which makes possible performing all the computations within CNNs in a reasonable amount of time. Hence, its full specifications are listed in the next table:

Table 3.1.1-III: Specifications of the GPU used in SR-GAN training.

NVIDIA GeForce RTX 2080 Ti	
Driver Version	461.72 (Windows)
CUDA Cores	4352
Memory Throughput	14.0 Gbps
Memory interface bus	352-bits
Memory bandwidth	616.00 GB/s
Core base clock	1545 MHz
Dedicated memory	11264 MB GDDR6

Table 3.1.1-IV: Specifications of the GPU used in SRFBN training.

NVIDIA GeForce RTX 3080	
Driver Version	466.11 (Windows)
CUDA Cores	8704
Memory Throughput	19.0 Gbps
Memory interface bus	320-bits
Memory bandwidth	760.08 GB/s
Core base clock	1710 MHz
Dedicated memory	10240 MB GDDR6X

3.1.2 Software

In this section all the software tools used in this project will be described.

- ***OS*** : The OS that we have been used throughout the development of this project is ***Windows 10 Enterprise*** [118]**.** It is the most popular OS worldwide, it is simple, stable and really easy to use. To search for information, write de project and test the trained models, my own computer with ***macOS Big Sur 11.4*** [75] was used.

- ***Anaconda*** [4]: is a distribution of the Python and R programming languages for scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.), that aims to simplify package management and deployment. The distribution includes data-science packages suitable for Windows, Linux, and macOS. It is developed and maintained by Anaconda, Inc., which was founded by Peter Wang and Travis Oliphant in 2012. Package versions in Anaconda are managed by the package management system conda. This package manager was spun out as a separate open-source package as it ended up being useful on its own and for other things apart from Python. There is also a small, bootstrap version of Anaconda called Miniconda [69].
Anaconda distribution comes with over 250 packages automatically installed, and over 7,500 additional open-source packages can be installed from PyPI as well as the conda package and virtual environment manager. It also includes a GUI, Anaconda Navigator, as a graphical alternative to the command line interface.
For this project, we have been using two different versions of conda for each PC: ***conda 4.9.2***, for SRGAN, and ***conda 4.10.1*** for SRFBN.

- ***Python*** [117]: is currently very widespread due to the great advantages it presents. It is a high-level, interpreted, multi-platform, general-purpose programming language whose philosophy emphasizes the readability of the code. It is multi-paradigm, since it is object-oriented, and supports imperative and functional programming. It is managed by the Python Software Foundation and has an open-source license.

This language is one of the most popular programming languages in the scientific and research field, mostly due to its comprehensive standard library and ease of fast prototyping. In fact, it is the most popular language in the AI and DL fields.
In this project, most of the code was developed with ***Python 3.8.8***.

- ***TensorFlow*** [99]: is a free and an open-source software library for machine learning. It disposes of a comprehensive set of tools, libraries, and resources allowing to easily implement, test and deploy ML applications. It has been developed by Google Brain team, which included this library within the development of most of Google products and research projects. With the release of the 2.0.0 version in October 2019, TensorFlow has become simpler and easier to use.
The environment used to carry out SRGAN experiments has been ***version 2.3.0***. In SRFBN experiments, the original code does not use TensorFlow, but we have implemented some parts with it. For that, we used ***v.2.5.0***.

- ***Keras*** [52]: is an open-source library written in Python capable of running on top of several ML libraries such as TensorFlow, Microsoft Cognitive Toolkit, or PlaidML.
TensorFlow implements it on its core library since 2017 to make it even easier to design, develop and deploy ML applications by combining the easy graph capabilities from Keras, and the power and performance of TensorFlow.

- ***Pytorch*** [80]: is used for applications such as computer vision and natural language processing, primarily developed by Facebook's AI Research lab (FAIR). It is a free and an open-source software released under the Modified BSD license. It is a machine learning library for Python, mainly developed by the Facebook AI Research team and it is one of the widely used Machine learning libraries. PyTorch is built based on python and torch library which supports computations of tensors on Graphical Processing Units. Arguably PyTorch is TensorFlow’s biggest competitor to date, and it is currently a much-favored deep learning and artificial intelligence library in the research community.
Some of its advantages are dynamic computational graphs; different back-end support; imperative style, it is specially designed to be intuitive and easy to use; highly

extensible, PyTorch is deeply integrated with the C++ code, and it shares some C++ backend with the deep learning framework, Torch.
SRFBN experiments have been done with ***Pytorch v.1.8.1***. SRGAN experiments have not used it.

- ***CUDA*** [17]: GPUs can be used to compute-intensive mathematical operations much faster than a CPU would do. CUDA refers to a parallel computing platform including a compiler and a set of development tools created by Nvidia that enables software engineers to use a CUDA-compatible GPU for general purpose processing.
The version that has been used is: ***10.1*** during SRGAN experiments, which is compatible with our TensorFlow version; and ***11.1*** for SRFBN experiments.

- ***Matlab***: (Matrix Laboratory) [68] is a numerical computing environment providing a full Integrated Development Environment (IDE) with a proprietary programming language called M language. It is available for all the platforms currently available.
Its basic features include matrix manipulation, data and function representation, algorithm development, creation of Graphic User Interface (GUI) and communication with programs in other languages and with other hardware devices.
We have used Matlab to generate the training dataset for SRFBN experiments, applying data augmentation (crop,rotate. For that, we used ***MATLAB R2021a Update 3 (9.10.0)***.

- ***QGIS***: (until 2013 known as Quantum GIS) [25], it is a free and an open-source cross-platform desktop GIS application that supports viewing, editing, and analysis of geospatial data. It supports raster and vector layers; multiple formats of raster images are supported, and the software can georeference images. Also, it supports shapefiles, coverages, personal geodatabases, dxf, MapInfo, PostGIS, and other formats. Web services, including Web Map Service and Web Feature Service, are supported too.
Gary Sherman began development of Quantum GIS in early 2002, and it became an incubator project of the Open Source Geospatial Foundation in 2007. Version 1.0 was released in January 2009. Currently, QGIS is maintained by volunteer developers who regularly release updates and bug fixes.
To create the test datasets we have used ***QGIS Desktop 3.18.1***.

3.1.3 Data

Once the model and the loss function are determined, the dataset to be used in the training process must be selected. In the case of GAN training, it is especially important to dispose of a large image dataset, as the quality of the results obtained after the training session will be highly dependent on the amount and variety of cases the model learns during the process.

Our objective in this project is to find and compare the different upscaling methods of aerial images/photos. To achieve this goal, we are going to work with max-resolution orthoimages obtained from the IGN Download Centre [36] and other official open-data distributors from different Autonomous Communities in Spain. The dataset is composed of diverse high-resolution orthophotos, specifically of 0.15 m/pixel. We are looking for a variety of images to represent in the best way possible the different locations of the Earth's surface. For this, we search for images of the coastal surfaces or bodies of water; urban places, man-made artificial elements; valleys, plains with or without vegetation, and also different representations of the Spanish orography and hydrography. The aerial images or orthoimages are photos of the Earth's surface obtained via satellite or aero-transported sensors. As we have already explained, we are going to study the behaviour of our trained model over a variety of images from Earth's surfaces, and different resolutions and sensors. For this, we will evaluate them with diverse aerial images:

- ***PNOA Máxima Actualidad***: it has a resolution of 0,15 m/pixel. They are mosaics of the most recent orthophotos available in the *National Aerial Orthophotos Plan*. Each mosaic covers a page from MTN50.

- ***PNOA 10***: they are mosaics of orthophotos with a 0,10 m/pixel resolution, gathered from years of flights done by diverse institutions of the Public Administrations. Each orthophoto derives from cut pages of 1:2.000 (a subdivision of each MTN50 in 20 x 20 pages), and also cuts of the surface of 1x1 km of extension.

- ***Pleiades***: the Pleiades satellite [85] is composed of two optical satellites, Pleiades-1A and Pleiades-1B, launched on December 16, 2011. They work synchronized to take daily images of any place on the planet, increasing the chances of obtaining clear, cloud-free images. The resolution of the Pleiades system is 2 m/pixel.

4. METHODOLOGY

The methodology used for the development of this project comprehends four blocks: the first block corresponds to the previous planning and theoretical-practical training of the topic; once we have chosen the networks that we want to introduce, we create the datasets (training, validation, and testing); the third block refers to the training process and all that it carries: definition of hyperparameters, resources optimization, variations and modifications of the architecture, etc.; eventually, we evaluate the models from the obtained results, and we compare them.

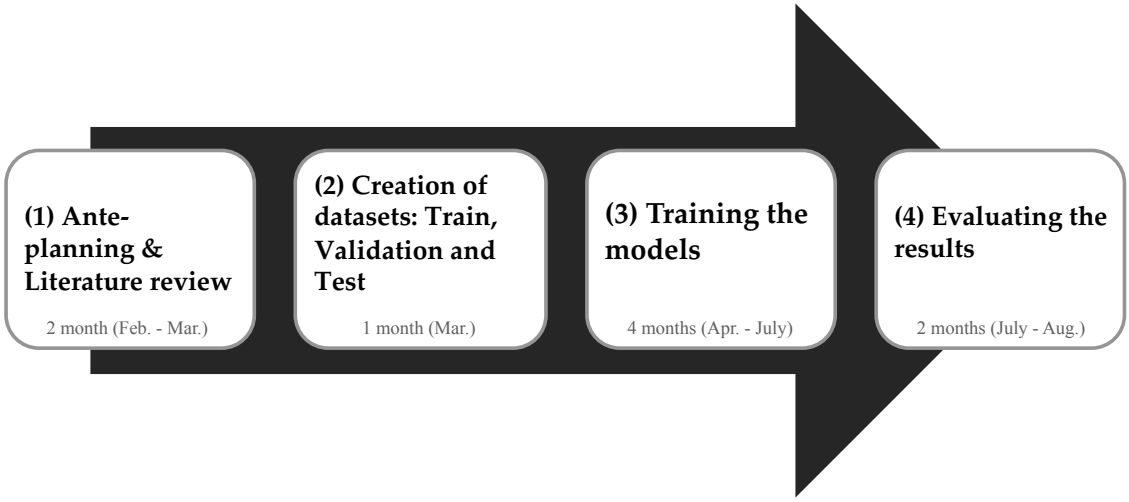

Fig. 4. Methodology followed in this project.

4.1. Data Pre-processing

Referring to the machine learning training processes, we can distinguish three different types of datasets depending on the use we will give them.

- ***Training Dataset***: is the sample of data used to fit the model. Usually, it is the biggest dataset. In this project, we have been working with an 80%.

- ***Validation Dataset***: is the sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters. The evaluation of the validation dataset is incorporated into the model configuration. In this project, we have been working with a 20%.

- ***Test Dataset***: is the sample of data used to provide an evaluation of a final trained model. The images from this dataset were not used during the training, i.e. the model has never seen this sample before. We will use this dataset to compare different fitted models.

As a summary, training and validation data include labels to monitor the performance metrics of the model. Instead of this, testing data is unlabeled and it is used to compare the built model with others.

In this project, we are going to work with aerial images that cover surfaces of several km2 in HR, which implies a bigger number of pixels per unit of length to be able to represent the total surface, this means that the files will be heavier. These files usually are about 6 and 9 GB, and they are composed of hundred thousands of pixels grouped in three channels, RGB. These files are extremely heavy to be processed easily, so we have used a tool generated by an investigation group called MERCATOR [76] for the creation and classification of tiles. This tool starts working with the HR orthophoto and generates a tile mesh of 1024x1024 that is used to divide the original image to obtain smaller representations, keeping the main resolution.

Graphical explanation of this process with an example of Cedeira, Galicia:

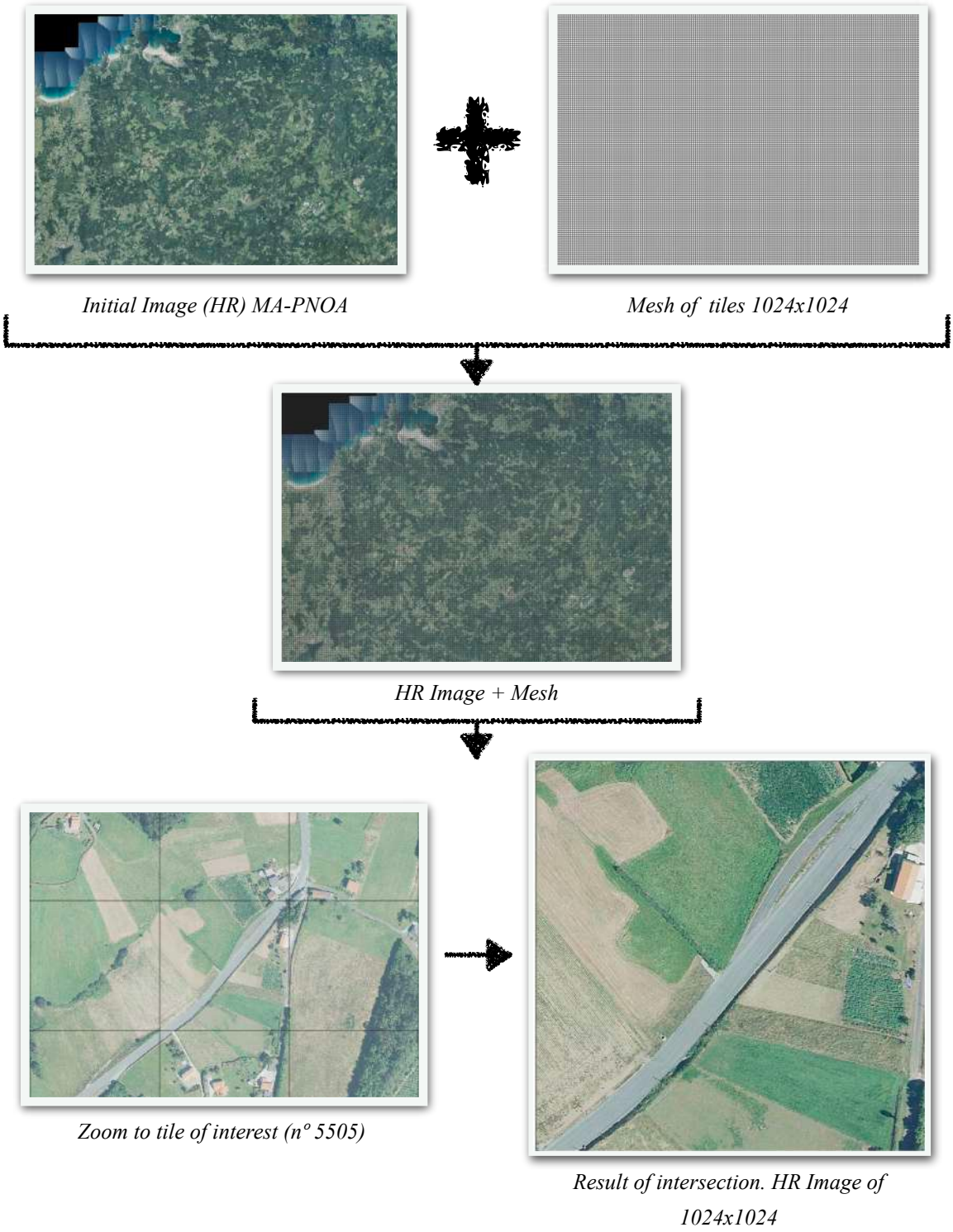

Fig. 4.1-I Example of tiles generation and HR images.

Once this tool is used, these HR images are put down into a process of downscaling, using the bicubic interpolation method of GDAL/OGR library [35].

Eventually, we obtain two folders, one with HR images and the second one with LR; each folder contains 39.371 images. This will be the total of images used for the training, and more specifically, we will obtain the validation and training dataset from this.

To divide all images into two different datasets, training and validation, we use the python library, *split-folders* [31]. This is executed with a command in the Anaconda prompt:

```
splitfolders  --output  D:\\Monica\\super-resolution-master\\viales
--ratio .8 .2 -- D:\\Monica\\tiles-viales-sr_256_x4
```

Defining variables: --*output*, is the path to the output folder; --*ratio*, determines the ratio to split, in this case, 80% to train and 20% to valid; and the last --, follows by the folder with the initial images.

Finally, we obtain the following datasets:

Table 4.1-I: Trainning and Validation Datasets.

Dataset	Number of images	Ratio
Train_HR	31.496	80 %
Valid_HR	7.875	20 %
Train_LR	31.496	80 %
Valid_LR	7.875	20 %

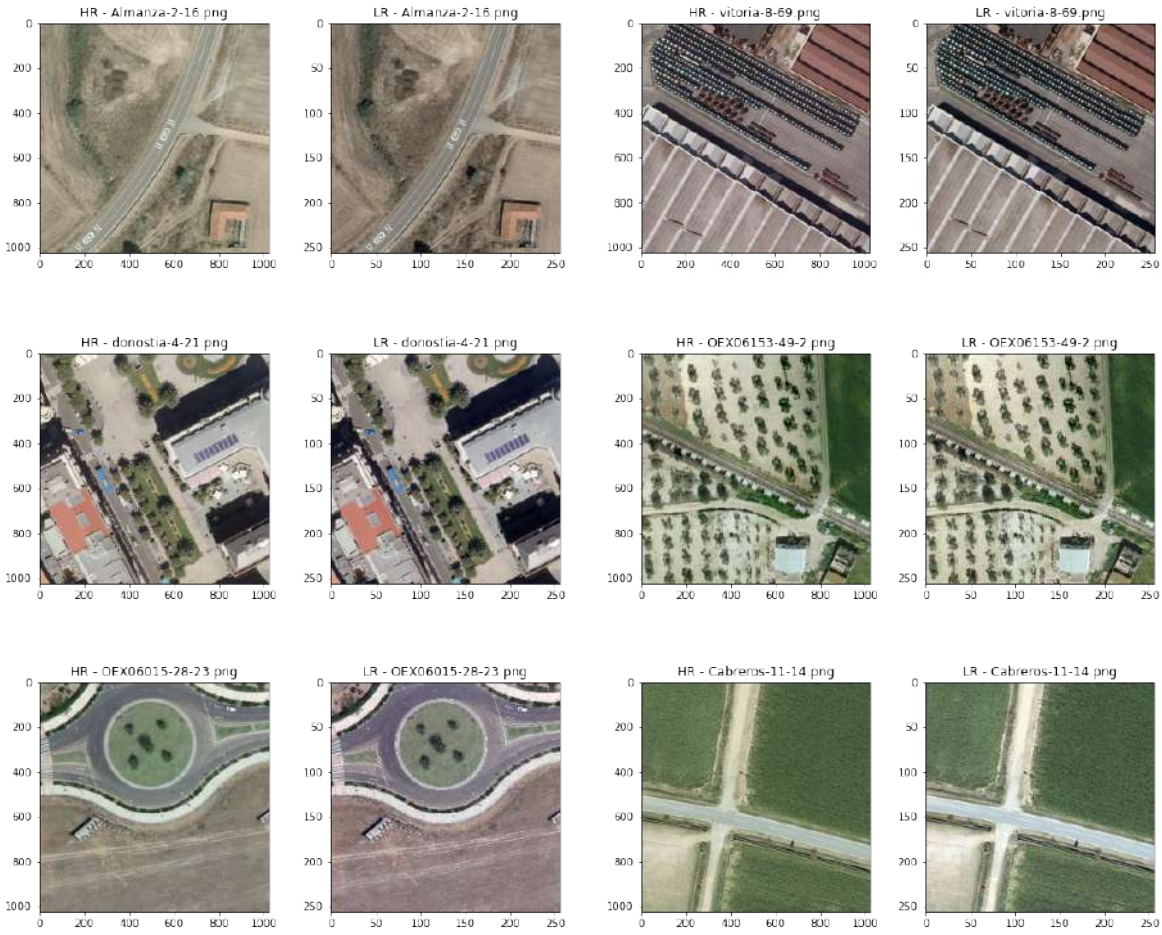

Fig. 4.1-II Some images from testing dataset (HR and LR)

We are interested in pairs of HR and LR test images to be able to apply the quantitative metrics/measurements, PSNR and SSIM, and after this, measure the quality of the generated images and eventually, evaluate the trained models.

The final objective of this project is to apply these models to images obtained from sensors whose resolution is not that high, like the Pleiades satellite images, to reduce its resolution to a quarter, in this case. 0,5m/pixel. This is why we will not put down these images to the downscaling process, we will rather use PNOA images with a resolution of 0,5 m/pixel of the same location and captured in a certain near period or similar to the satellites shot, to use them as their correspondent HR image, as a reference of the specific quality that we want to obtain.

This way we will be able to apply the metrics, although the images aspect and the perceptible changes to the human eye will have a bigger influence. The high-resolution images were not obtained from the low-resolution ones and there are discrepancies between them, like for example the shadow positions which affect the obtained values when PSNR or SSIM are applied.

To create the test dataset from the PNOA images (MA and 10), we will generate in QGIS 1024x1024 pixel tiles meshes in QGIS, with the same tool "*Create grid*". The vertical and horizontal spacing parameters are obtained by multiplying the desired pixel size by the initial image resolution, in this case: *1024x0,15=153,6*. Eventually, we cut off the tiles we are interested in, with "*Cut raster by layer mask*", tool, as it is shown in the following picture:

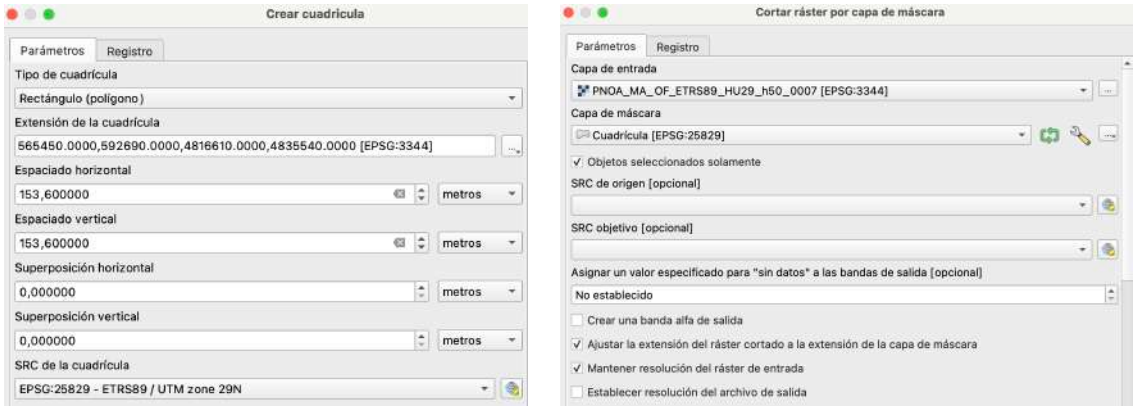

Fig. 4.1-III Mesh generation and tiles cutting with QGIS.

Due to the inaccuracy of the tool when working with figures that are not round, we decided to confirm the size of the images and reduce their resolution to obtain the pairs of images. For this we will define some functions in the python file "*test_tools.py*" and we will put them into practice in the jupyter notebook *"Test_Dataset.ipynb*".

First, we need to access the files, to eventually cut them according to the desired size, in this case

First we need to access to the files, to eventually cut them according to the desired size, in this case 1024x1024 pixels. Defined functions in "*test_tools.py*":

Code snippet 4.1: Definition of functions access to files and crop images.

```
import os
import cv2
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image


#TO OBTAIN FILE'S NAME

def files_name(dir,type_file):
    contenido = os.listdir(dir)
    names=[]
    for fichero in contenido:
        if os.path.isfile(os.path.join(dir, fichero)) and fichero.endswith(type_file):
            names.append(fichero)
    return names,dir


##############################################################################


#TO CROP EACH IMAGE

def crop_image(path,name,width,height):
    new_name = name[:-4] + "_" + str(width) + "x" + str(height) + ".png"
    img = cv2.imread(path+name)
    crop_img = img[0:width, 0:height]
    final_path = path + str(width) + "x" + str(height) + "/"
    if not os.path.exists(final_path):
        os.mkdir(final_path)
    return cv2.imwrite(final_path + new_name, crop_img)
```

Code snippet 4.1: Cropping images to 1024x1024.

```
from test_tools import files_name, crop_image, downscaling, load_image, plot_images


##############################################################################


#CROP IMAGES TO 1024x1024

#PNOA MA
[pnoa,pnoa_dir] = files_name("/Users/monica/Documents/TFG/SRGAN/Imgs_Test_QGIS/PNOA_MA/HR/",".png")
for file_name in pnoa:
    crop_image(pnoa_dir,file_name,1024,1024)

#PNOA 10
[pnoa10,pnoa10_dir] = files_name("/Users/monica/Documents/TFG/SRGAN/Imgs_Test_QGIS/PNOA10/HR/",".png")
for file_name in pnoa10:
    crop_image(pnoa10_dir,file_name,1024,1024)
```

Finally, we obtained the LR test images by downsampling the HR test images using a bicubic kernel with downsampling factor r = 4.

For the downscaling, we define a new function in which we have to define the HR images path that we want to reduce, the names of the files, and the scale that we want to use. As we have explained before, we are going to use the bicubic method of different python libraries, to finally use GDAL/OGR [35], the same library that has been used to generate the training and validation datasets:

Fig. 4.1-IV Downscaling with cv2, PIL, GDAL/OGR y Matlab libraries.

Code snippet 4.1: Definition of cropping function and its application.

```
#DOWNSCALING

def downscaling(path,name,scale):
    #img=load_image(path+name)
    #w=img.shape[1]
    #h=img.shape[0]
    img=Image.open(path+name)
    h,w = img.size
    a=int(w/scale)
    b=int(h/scale)
    size=(a,b)
    #Double cubic interpolation
    bicubic=img.resize(size, gdal.GRIORA_Cubic)
    new_path= path[:-1].replace("HR/"+str(w)+"x"+str(h),"LR")
    if not os.path.exists(new_path):
        os.mkdir(new_path)
    final_path=new_path+ "/" + str(int(w/scale))+"x"+str(int(h/scale)) + "/"
    if not os.path.exists(final_path):
        os.mkdir(final_path)
    new_name = name.replace(str(w)+"x"+str(h), str(int(w/scale))+"x"+str(int(h/scale)))
    #result1=cv2.imwrite(final_path+new_name, bicubic)
    result1=bicubic.save(fp=final_path+new_name)
```

Code snippet 4.1: Application of downscaling function.

```
#DOWNSCALING TO 256x256 AND PLOT

#PNOA MA
[pnoa,pnoa_dir] = files_name("/Users/monica/Documents/TFG/SRGAN/Imgs_Test_QGIS/PNOA_MA/HR/1024x1024/",".png")
for file_name in pnoa:
    downscaling(pnoa_dir,file_name,4)

#PNOA 10
[pnoa10,pnoa10_dir] = files_name("/Users/monica/Documents/TFG/SRGAN/Imgs_Test_QGIS/PNOA10/HR/1024x1024/",".png"
for file_name in pnoa10:
    downscaling(pnoa10_dir,file_name,4)
```

As a result, we have three different test datasets from different flights and images captured with diverse sensors. For each of these methods, two folders are created, one for HR images and the other for LR:

Table 4.1-II: Testing Dataset.

Flight	HR	LR	Nº of test images
PNOA Máxima Actualidad	0.15	0.6	14
PNOA 10	0.1	0,4	11
Satélite Pleiades	0.5 (PNOA)	2	12

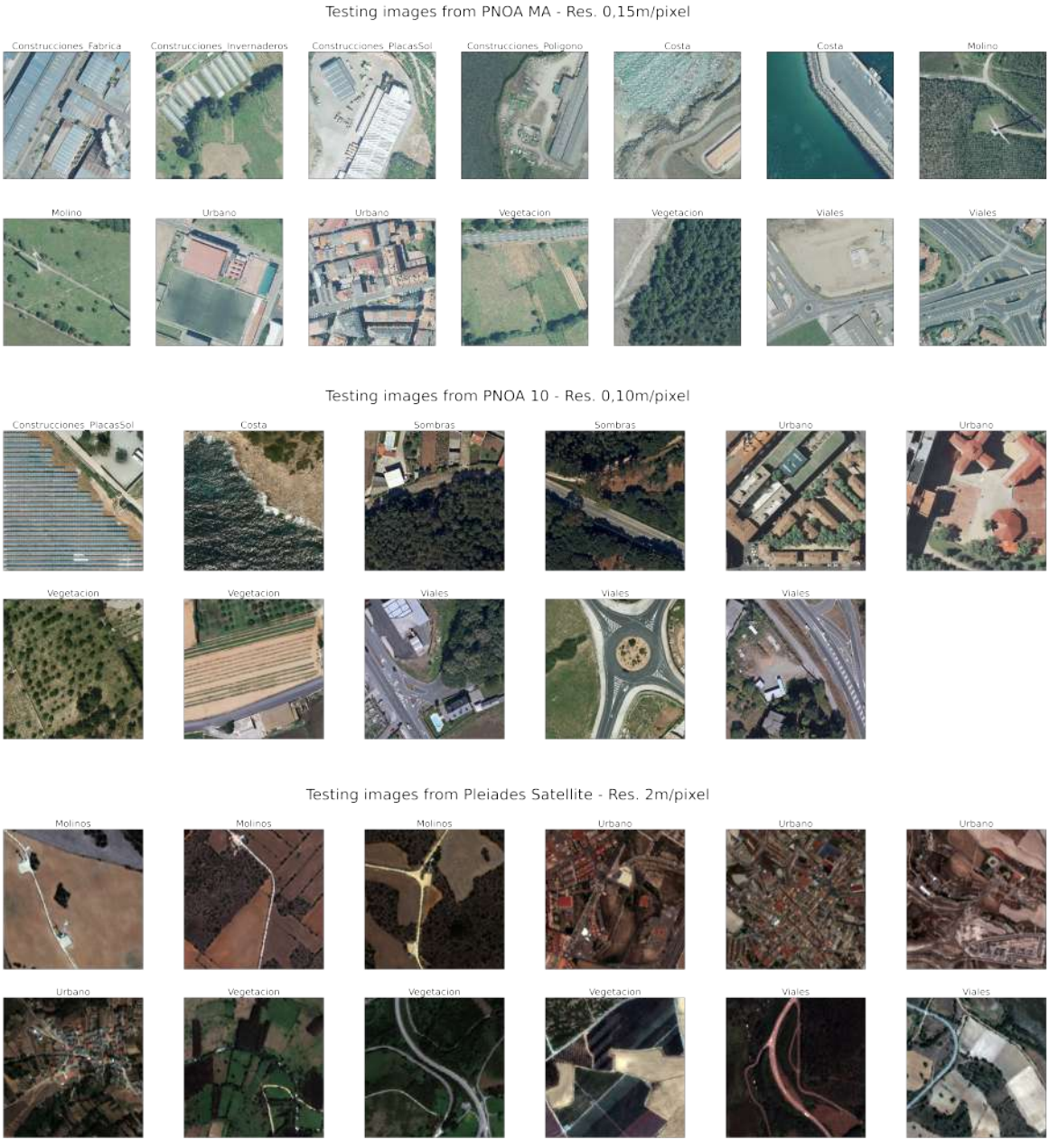

Fig. 4.1-V Images from testing dataset

4.1.1 Data Augmentation

How we previously mentioned, the dataset is the key to obtain good results. For that, the method known as data augmentation is a good choice to enrich our training. And what is data augmentation? It is a technique used to increase the diversity of the initial training dataset, applying different and random transformations such as rotation or cropping images [19][47].

Some of the most used transformations are padding, random rotating, re-scaling, vertical and horizontal flipping, translation, cropping, zooming, color modification, adding noise, etc. The operations used in this project are:

- ***Random Crop***: is the sample of data used to fit the model. It usually is the biggest dataset. In this project, we have been worked with an 80% [106].

- ***Random Flip:*** moves the pixels horizontally [100].

- ***Random Rotate:*** this transformation rotates images by 90 degrees (counter-clockwise) [101].

- ***Adding (Gaussian) Noise:*** adding noise to the input data helps to train our network, which means that it will generalize well even on noisy data. It applies additive zero-centered Gaussian noise and this is useful to mitigate overfitting [95] [9].

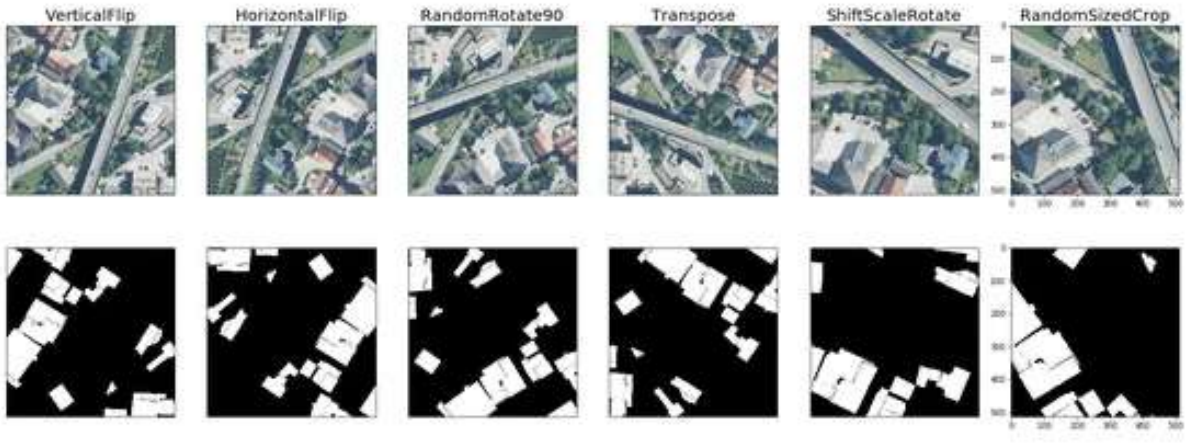

Fig. 4.1-V An example of data augmentation applied to satellite images [9].

4.2. First contact with the architecture

In this section, we will explain the origin of the models, as well as their implementation and code. We will submit the networks to a first test to verify their operation and make the necessary modifications to adapt them to our case study. Once the proper functioning of the model has been confirmed, we will do a hyperparameters recognition, those variables whose value and variation directly affect the network behaviour, and final results obtained.

4.2.1 SR-GAN

The architecture used, as we explained in [Section 2.4], was proposed in 2016 by a group of Twitter researchers. We have started from the implementation found in the GitHub repository [55], developed in Python with Tensorflow and Keras, and based on the work of Ledig et al. [56]. Due to the high costs to implement the architecture from the beginning, we have chosen to start from an existing one, modifying it to adapt it to our own case of study and use it in our project.

4.2.1.a Training First Experiment

We start applying the already trained model provided in the repository. This model has been trained with images from the DIV2K [24] whose nature and characteristics are far apart from the type of elements that we will be representing, that is, in the orthoimages. For this reason, it is to be expected that when applying this model, the results will be of poor quality. Below in Fig x.x, there are the SR images, obtained by applying the already trained model, which have obtained the highest PSNR and SSIM values. It is shown how the SR-generated images have much less sharpness compared to the HR images, a blurry grainy or noise effect is created that causes the straight elements, such as road signs, to have a pixelated effect and are not well defined.

PNOA MA:

Maximum PSNR: 29.2261
Maximum SSIM: 0.3891
Max. PSNR and SSIM are in different images: PNOA_MA_Viales2_SR and PNOA_MA_Viales_SR

PNOA 10:

Maximum PSNR: 30.4702
Maximum SSIM: 0.6901
Maximum PSNR and SSIM are in the same image: PNOA_10_Urbano_SR

Pleiades Satellite:

Maximum PSNR: 28.5056
Maximum SSIM: 0.5577
Maximum PSNR and SSIM are in the same image: Satelite_Molinos2_SR

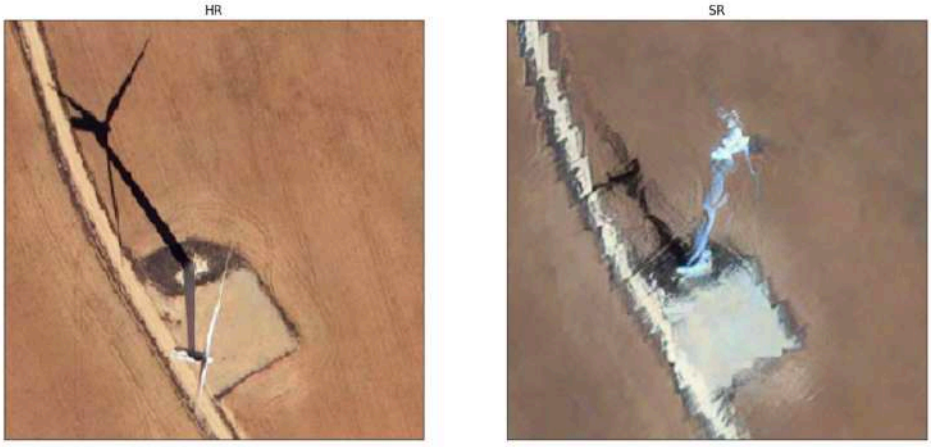

Fig. 4.2.1.a Results from the first experiment with default trained model.

Once the example model has been applied and the results are analyzed, we confirm the operation of the application and evaluation process, and we get ready to do the first training with our dataset.

As we discussed in [Section 3.1.2], this project has been developed with Tensorflow and Keras. We use Anaconda and the *jupyter notebook* library to access the project. We check and adapt the code, starting with the file "*Data.py*", to access the datasets. We define the train and valid folder paths (l.20 y l.21), so in this way, we can access the image files with the *listdir* method of *os* library:

Code snippet 4.2.1.a: Access to our Dataset.

```
1  import os
2  import tensorflow as tf
3  from os import listdir
4  from os.path import isfile, join
5
6  from tensorflow.python.data.experimental import AUTOTUNE
7
8
9  class DIV2K:
10     def __init__(self,
11                  scale=4,
12                  subset='train',
13                  downgrade='bicubic',
14                  images_dir='.div2k/images',
15                  caches_dir='.div2k/caches'):
16
17         self._ntire_2018 = True
18
19         _scales = [2, 3, 4, 8]
20         mypathHR_train = "G:\\Monica\\Anaconda_Notebook\\super-resolution-master\\.div2k\\images\\DIV2K_train_HR\\"
21         mypathHR_valid = "G:\\Monica\\Anaconda_Notebook\\super-resolution-master\\.div2k\\images\\DIV2K_valid_HR\\"
22
23         if scale in _scales:
24             self.scale = scale
25         else:
26             raise ValueError(f'scale must be in ${_scales}')
27
28         if subset == 'train':
29             self.image_ids = [f for f in listdir(mypathHR_train) if isfile(join(mypathHR_train, f))]
30         elif subset == 'valid':
31             self.image_ids = [f for f in listdir(mypathHR_valid) if isfile(join(mypathHR_valid, f))]
32         else:
33             raise ValueError("subset must be 'train' or 'valid'")
```

Once we have edited this first section, we review the Generator and Discriminator architectures defined in [Section 2.4.1] and [Section 2.4.2].

- ***Discriminator***: some of the most relevant parts of the Discriminator model implementation are shown below:

Code snippet 4.2.1.a: Discriminator Architecture.

```
53 def discriminator_block(x_in, num_filters, strides=1, batchnorm=True, momentum=0.8):
54     x = Conv2D(num_filters, kernel_size=3, strides=strides, padding='same')(x_in)
55     if batchnorm:
56         x = BatchNormalization(momentum=momentum)(x)
57     return LeakyReLU(alpha=0.2)(x)
58
59
60 def discriminator(num_filters=64):
61     x_in = Input(shape=(HR_SIZE, HR_SIZE, 3))
62     x = Lambda(normalize_m11)(x_in) #Normalizes RGB images to [0, 1]
63
64     x = discriminator_block(x, num_filters, batchnorm=False)
65     x = discriminator_block(x, num_filters, strides=2)
66
67     x = discriminator_block(x, num_filters * 2)
68     x = discriminator_block(x, num_filters * 2, strides=2)
69
70     x = discriminator_block(x, num_filters * 4)
71     x = discriminator_block(x, num_filters * 4, strides=2)
72
73     x = discriminator_block(x, num_filters * 8)
74     x = discriminator_block(x, num_filters * 8, strides=2)
75
76     x = Flatten()(x)
77
78     x = Dense(1024)(x)
79     x = LeakyReLU(alpha=0.2)(x)
80     x = Dense(1, activation='sigmoid')(x)
81
82     return Model(x_in, x)
```

"common.py"

```
43 def normalize_01(x):
44     """Normalizes RGB images to [0, 1]."""
45     return x / 255.0
```

- ***Generator***: the Generator architecture are described below:

Code snippet 4.2.1.a: Generator Architecture.

```
def res_block(x_in, num_filters, momentum=0.8):
    x = Conv2D(num_filters, kernel_size=3, padding='same')(x_in)
    x = BatchNormalization(momentum=momentum)(x)
    x = PReLU(shared_axes=[1, 2])(x)
    x = Conv2D(num_filters, kernel_size=3, padding='same')(x)
    x = BatchNormalization(momentum=momentum)(x)
    x = Add()([x_in, x])
    return x


def sr_resnet(num_filters=64, num_res_blocks=16):
    x_in = Input(shape=(None, None, 3))
    x = Lambda(normalize_01)(x_in)  #Normalizes RGB images to [-1, 1]

    x = Conv2D(num_filters, kernel_size=9, padding='same')(x)
    x = x_1 = PReLU(shared_axes=[1, 2])(x)

    for _ in range(num_res_blocks):
        x = res_block(x, num_filters)

    x = Conv2D(num_filters, kernel_size=3, padding='same')(x)
    x = BatchNormalization()(x)
    x = Add()([x_1, x])

    x = upsample(x, num_filters * 4)
    x = upsample(x, num_filters * 4)

    x = Conv2D(3, kernel_size=9, padding='same', activation='tanh')(x)
    x = Lambda(denormalize_m11)(x)

    return Model(x_in, x)


generator = sr_resnet
```

"common.py"

```
def normalize_m11(x):
    """Normalizes RGB images to [-1, 1]."""
    return x / 127.5 - 1
```

Once the architectures of both models have been defined, it is time to put them to work together and shape the complete framework of the SRGAN model. Our training method is based on the work of Ledig et al. [56], in which they identified two different stages within the training process: Pre-training the generator with MSE loss function and then performing the GAN training.

- ***Pre-Training***: this stage consists in performing an initial training of the generator using an MSE-based loss function before training the model with the GAN method. This pre-training process helps the generator reach a better result during the GAN training phase, as it avoids undesired local optima and assures the convergence of the model. It is defined in "*train.py*" and the most important parts of the process are illustrated with code below:

Code snippet 4.2.1.a: Pre-Training construction.

```
class SrganGeneratorTrainer(Trainer):
    def __init__(self,
                 model, #model=generator()
                 checkpoint_dir,
                 learning_rate=1e-4):
        super().__init__(model, loss=MeanSquaredError(), learning_rate=learning_rate, checkpoint_dir=checkpoint_dir)

    def train(self, train_dataset, valid_dataset, steps=1000000, evaluate_every=1000, save_best_only=True):
        super().train(train_dataset, valid_dataset, steps, evaluate_every, save_best_only)
```

Code snippet 4.2.1.a: Hyperparameters and Checkpoint definition.

```
class Trainer:
    def __init__(self,
                 model,
                 loss,
                 learning_rate,
                 checkpoint_dir='./ckpt/edsr'):

        self.now = None
        self.loss = loss #MSE
        self.checkpoint = tf.train.Checkpoint(step=tf.Variable(0),
                                              psnr=tf.Variable(-1.0),
                                              optimizer=Adam(learning_rate),
                                              model=model)
        self.checkpoint_manager = tf.train.CheckpointManager(checkpoint=self.checkpoint,
                                                             directory=checkpoint_dir,
                                                             max_to_keep=3)

        self.restore()
```

We run the first training, beginning with the pre-training. This first interaction with the network has been made with hyperparameters definitions and default architectures, which are explained in the following section [Section 4.2.1.b]. Finally, we store the loss values in a .csv file.

- ***SRGAN Training***: defined in "*train.py*" too:

Code snippet 4.2.1.a: SRGAN Training. Generator and Discriminator training.

```
class SrganTrainer:
    #
    # TODO: model and optimizer checkpoints
    #
    def __init__(self,
                 generator,
                 discriminator,
                 content_loss='VGG54',
                 learning_rate=PiecewiseConstantDecay(boundaries=[100000], values=[1e-4, 1e-5])):

        if content_loss == 'VGG22':
            self.vgg = srgan.vgg_22()
        elif content_loss == 'VGG54':
            self.vgg = srgan.vgg_54()
        else:
            raise ValueError("content_loss must be either 'VGG22' or 'VGG54'")

        self.content_loss = content_loss
        self.generator = generator
        self.discriminator = discriminator
        self.generator_optimizer = Adam(learning_rate=learning_rate)
        self.discriminator_optimizer = Adam(learning_rate=learning_rate)

        self.binary_cross_entropy = BinaryCrossentropy(from_logits=False)
        self.mean_squared_error = MeanSquaredError()

    def train(self, train_dataset, steps=200000):
        pls_metric = Mean()
        dls_metric = Mean()
        step = 0

        for lr, hr in train_dataset.take(steps):
            step += 1

            pl, dl = self.train_step(lr, hr)
            pls_metric(pl)
            dls_metric(dl)

            if step % 50 == 0:
                print(f'{step}/{steps}, perceptual loss = {pls_metric.result():.4f}, discriminator loss =
{dls_metric.result():.4f}')
                pls_metric.reset_states()
                dls_metric.reset_states()
```

In the complete network training the loss functions are defined, which are explained in [Section 2.4.4]. Generator Loss or what is the same, Adversarial Loss [Section 2.4.4.a] (*Perceptual Loss = Content* + 10^{-3}*Adversarial Loss)*; and Discriminator Loss [Section 2.4.4.b]:

Code snippet 4.2.1.a: SRGAN Training. Losses.

```
    @tf.function
    def train_step(self, lr, hr):
        with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
            lr = tf.cast(lr, tf.float32)
            hr = tf.cast(hr, tf.float32)

            sr = self.generator(lr, training=True)

            hr_output = self.discriminator(hr, training=True)
            sr_output = self.discriminator(sr, training=True)

            con_loss = self._content_loss(hr, sr)
            gen_loss = self._generator_loss(sr_output)
            perc_loss = con_loss + 0.001 * gen_loss
            disc_loss = self._discriminator_loss(hr_output, sr_output)

        gradients_of_generator = gen_tape.gradient(perc_loss, self.generator.trainable_variables)
        gradients_of_discriminator = disc_tape.gradient(disc_loss, self.discriminator.trainable_variables)

        self.generator_optimizer.apply_gradients(zip(gradients_of_generator, self.generator.trainable_variables))
        self.discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, self.discriminator.trainable_variables))

        return perc_loss, disc_loss

    @tf.function
    def _content_loss(self, hr, sr):
        sr = preprocess_input(sr)
        hr = preprocess_input(hr)
        sr_features = self.vgg(sr) / 12.75
        hr_features = self.vgg(hr) / 12.75
        return self.mean_squared_error(hr_features, sr_features)

    def _generator_loss(self, sr_out):
        return self.binary_cross_entropy(tf.ones_like(sr_out), sr_out)

    def _discriminator_loss(self, hr_out, sr_out):
        hr_loss = self.binary_cross_entropy(tf.ones_like(hr_out), hr_out)
        sr_loss = self.binary_cross_entropy(tf.zeros_like(sr_out), sr_out)
        return hr_loss + sr_loss
```

4.2.1.b Hyperparameters

Firstly, we need to know what are hyperparameters [3] and how we can differentiate them from ordinary parameters. We could define them as all the training variables set manually with a pre-determined value before starting the training [82]. Instead of these, the variables learned from the data during the training, the weights, are known as model parameters. These are excluded from the hyperparameter set.

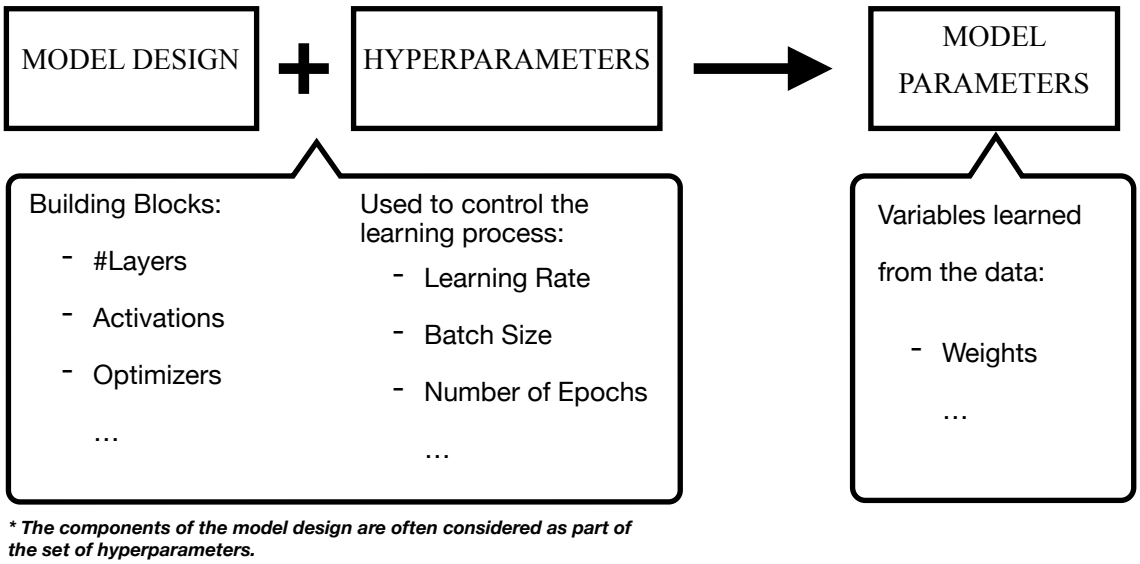

Fig. 4.2.1.b-I Scheme of the distribution of parameters in the model.

There are different methods of hyperparameter tuning [79], like *Grid Search* [49], *Random Search* [99], or *Bayesian Optimization* [111]. In this project, we have used the 100% manual method, *Trial and Error*. It consists of training and set different values for these hyperparameters to finally compare and analyze the results. Also, other factors, as processing time, must be considered. We have decided to use this method primarily because we have enough time to perform numerous experiments. Also, it is a good way to understand how the model works and how hyperparameters affect it.

Some of the most used hyperparameters:

- ***Kernel***: [Section 2.1.3] is a matrix that moves over the input data, performs the dot product with the sub-region of input data, and gets the output as the matrix of dot products.

- ***Stride***: defines the number of positions, rows or columns, the kernel will move.

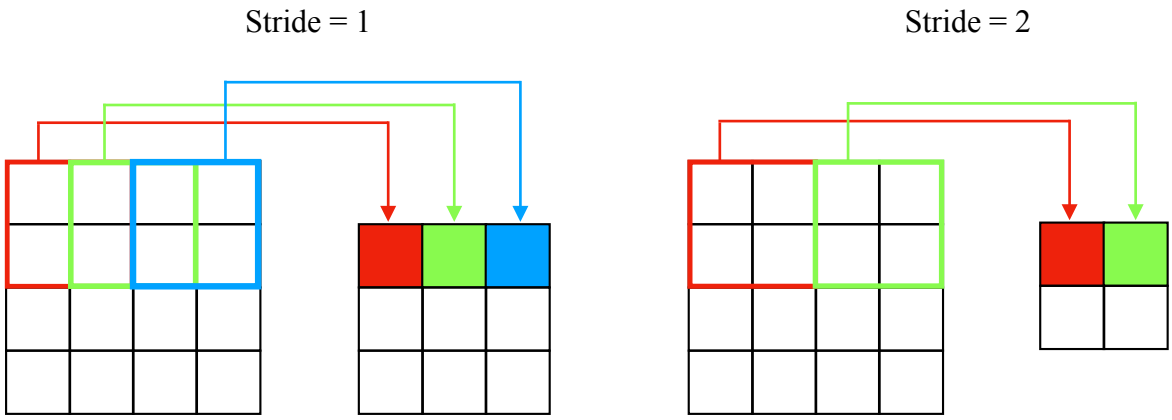

Fig. 4.2.1.b-II Scheme of stride behaviour.

- ***Learning rate***: defines how quickly a network updates its parameters. Choosing the learning rate is challenging, a low learning rate results in a long training process that could get stuck. Whereas a large learning rate speeds up the learning process, making it unstable, and may not converge. Usually a decaying Learning rate is preferred.

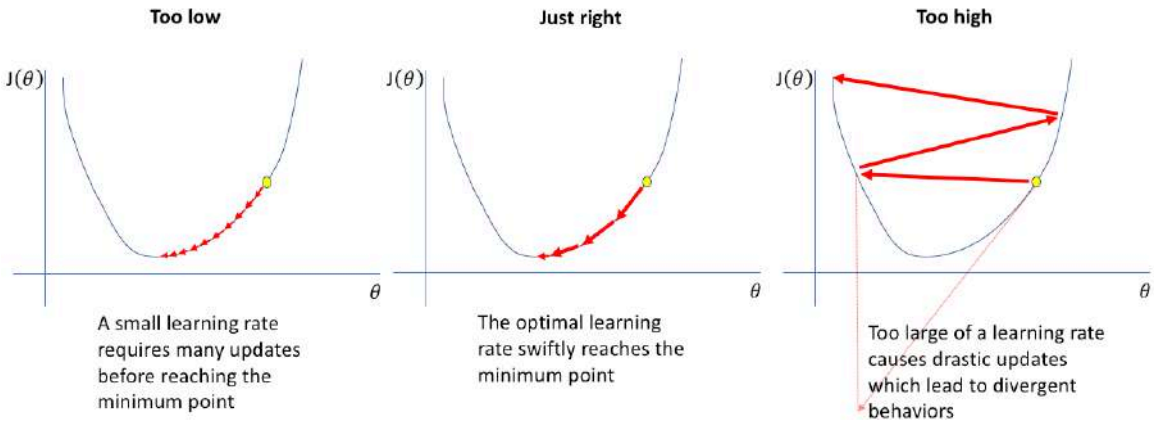

Fig. 4.2.1.b-III Influence of learning rate value on training [96].

- ***Batch size***: is the number of sub-samples given to the network during the training.

- ***Number of epochs***: is the number of times the model has total access to the whole training data during training. Its value should increase until the validation accuracy starts decreasing; when training accuracy is increasing, it is known as overfitting.

- ***Optimizer*****:** is an algorithm or method used to change attributes, such as weights or learning rate, to reduce losses. Optimizers are used to solve optimization problems by minimizing the function [77].

- ***Activation function***: As we explained in [Section 2.1.2.a], these functions are used to introduce nonlinearity to models.

- ***Loss function***: this was explained in [Section 2.1.2.b]. As a little reminder, the loss function is a method of evaluating how well specific algorithm models the given data.

- ***Momentum***: is a very popular technique that is used along with SGD. Instead of using only the gradient of the current step to guide the search, momentum also accumulates the gradient of the past steps to determine the direction to go. Momentum or SGD with momentum introduced by Qian [81], helps accelerate gradients vectors in the right directions, thus leading to faster converging.

Following the trial and error methodology, these hyperparameters have been modified for the different experiments and their results have been compared and analyzed. For this comparison, we applied these trained models to the test dataset. Results and a table with all the combinations used were recorded in Microsoft Excel, "*experiments.xlsx"*. Some of hyperparameter modifications tested:

- ***Batch Size***: 4, 6, 8, 16 and 32.

- ***Number of epochs (pre-training)***: 1.000, 75000, 100.000, 279000, 310000 and 1.000.000,

- ***Repeat count***: None, 4, 8 and 16.

- ***Content Loss - Pre-trained losses (VGG)***: we use a loss function based on the euclidean distance between feature maps extracted from the VGG19 [102]. VGG22 a loss defined on feature maps representing lower-level features; and VGG54 that is a loss defined on feature maps of higher level features from deeper network layers with more potential to focus on the content of the images.

Code snippet 4.2.1.b: Pre-trained losses VGG22 and VGG54 from VGG19.

```
85 def vgg_22():
86     return _vgg(5)
87
88
89 def vgg_54():
90     return _vgg(20)
91
92
93 def _vgg(output_layer):
94     vgg = VGG19(input_shape=(None, None, 3), include_top=False)
95     return Model(vgg.input, vgg.layers[output_layer].output)
```

- ***Learning rate***: All SRGAN variants were trained with 10^5 update iterations at a learning rate of 10^{-4} and another 10^5 iterations at a lower rate of 10^{-5}. For that, we use the library *PiecewiseConstant* from Tensorflow [98]:

PiecewiseConstantDecay(boundaries=[100000], values=[1e-4, 1e-5]).

- ***Optimizer***: Adam [53] with β1 = 0.9 [105].

- ***Activation Function***: Generator (ResNet), PReLU [*Code snippet 5.2.1.a: Generator Architecture - l.20*] [104]; and Discriminator (VGGNet), LeakyReLU [*Code snippet 4.2.1.a: Discriminator Architecture - l.57*] [103].

- ***Number of filters***: 64, 128 and 256.

- ***Number of residual blocks (Generator)*** [43]: 8, 16 and 32.

- ***Momentum***: 0.8 [56].

4.2.1.c Testing

After we have trained the different models, they must be evaluated. For this, we use the methods defined in "*Model/common.py*" where the LR image is introduced into the trained model. To do this, you must have previously loaded the weights to its corresponding model using "*load_weights*" method from Keras [96]. The weights are saved in *.h5* files [40]:

Code snippet 5.2.1.c: Loading weights to each model

```
pre_generator = generator()
gan_generator = generator()
gan_discriminator = discriminator()

pre_generator.load_weights(weights_file('/Users/monica/Downloads/srgan/batch32/pre_generatorx4_batch32.h5'
gan_generator.load_weights(weights_file('/Users/monica/Downloads/srgan/batch32/gan_generatorx4_batch32.h5'
gan_discriminator.load_weights(weights_file('/Users/monica/Downloads/srgan/batch32/gan_discriminatorx4_bat
```

Code snippet 5.2.1.c: Model/common.py

```
def resolve_single(model, lr):
    return resolve(model, tf.expand_dims(lr, axis=0))[0]


def resolve(model, lr_batch):
    lr_batch = tf.cast(lr_batch, tf.float32)
    sr_batch = model(lr_batch)
    sr_batch = tf.clip_by_value(sr_batch, 0, 255)
    sr_batch = tf.round(sr_batch)
    sr_batch = tf.cast(sr_batch, tf.uint8)
    return sr_batch
```

Code snippet 5.2.1.c: Resolve and Plot the resulting SR image

```
from model import resolve_single
from utils import load_image
from PIL import Image
import PIL
import tensorflow as tf


def resolve_and_plot(lr_image_path, new_path, name):
    if not os.path.exists(new_path):
        os.makedirs(new_path)
    lr = load_image(lr_image_path)

    pre_sr = resolve_single(pre_generator, lr)
    gan_sr = resolve_single(gan_generator, lr)
```

4.2.1.d Best Experiments

Once the hyperparameters and the method used have been introduced, in this section we are going to present the architectures with the best results, as well as the value of its hyperparameters and their influence on the final results.

Combinations are in the excel file with results. To evaluate each model, we have applied the network to the three test datasets (PNOA MA, PNOA 10, and Pleiades Satellite), calculating for each image the two metrics, SSIM and PSNR. Finally, we obtain the mean for each dataset that we will use to generally evaluate the architecture. As can be seen in the excel with the combinations, the best results are obtained when running a pre-training of a million epochs. As for the number of filters, it usually starts with a small number and increases. In the beginning, we can think that a higher number of filters or feature layers, better results, but in this case, we obtain very similar results. Regarding the residual blocks, the best results are obtained with networks composed of 8 and 16.

Finally, we see how batch size, i.e., the number of images that feed the network in each epoch, is a decisive parameter. Once again, we can think that the more images we introduce in each epoch, the better results will be, but we check that this is not always like this, and it depends on the combination of parameters used. The size of the batch size influences processing time and hardware requirements too.

Finally, the models with the highest metric values are:

Table 4.2.1.c: Hyperparameters values of the four best models - SRGAN.

Ranking	Batch Size	Nº Filters	Nº Residual Blocks
1	32	64	16
2	8	256	8
3	32	128	8
4	6	64	16

The number of parameters of each model are shown below, architectures are illustrated in Annexes. The architecture of the first and fourth combination is the same, so the number of parameters will be identical in both; only the batch size varies. The model with more parameters, how we could expect, coincides with the one with a larger architecture, i.e., with the architecture with a higher number of filters and residual blocks, in this case, it is the second-best model.

The generator architecture is the same for pre-training as for general training for the entire network, that is, the training for generator and discriminator together. The discriminator architecture is only used during the global GAN training. Architectures are added in Annexes.

Models 1 and 4.

NºFilters (64) + ResBlocks (16)

Generator

```
Total params: 1,554,883
Trainable params: 1,550,659
Non-trainable params: 4,224
```

Discriminator

```
Total params: 23,569,217
Trainable params: 23,565,505
Non-trainable params: 3,712
```

Model 2.

NºFilters (256) + ResBlocks (8)

Generator

```
Total params: 14,896,899
Trainable params: 14,888,195
Non-trainable params: 8,704
```

Discriminator

```
Total params: 150,451,457
Trainable params: 150,436,609
Non-trainable params: 14,848
```

Model 3.

NºFilters (128) + ResBlocks (8)

Generator

```
Total params: 3,762,051
Trainable params: 3,757,699
Non-trainable params: 4,352
```

Discriminator

```
Total params: 56,499,841
Trainable params: 56,492,417
Non-trainable params: 7,424
```

As we have already mentioned, the batch size does not influence the architecture or the number of its parameters, but it does directly affect the processing time. Depending on our hardware characteristics, this processing time will be longer or shorter. Therefore, in this case, it is important to have an SSD disk, whose capacity to read and transmit data is higher; as well as using GPUs instead of CPUs, thus avoiding problems such as "*bottleneck*" [109]. During the training process, it was important to check that the use of our resources was adequate, for this, we used the Windows tool "*Task Manager*" to visualize the resources consumption. To monitor the GPU performance, we had to observe the CUDA [16] consumption and the dedicated memory, it is recommended that these do not exceed 90-95% of its capacity. During intense training, i.e., a training with many parameters and a large number of input images; such as the training network where the generator and discriminator intervene, CUDAs usually work at 98% of their total capacity.

The average processing time that we have required to train each model has been around five days; using three of them for the pre-training of the generator and the remaining two days, for the training of the discriminator and generator together. These approximations are made from training with the hardware and software described in [Section 3].

5.2.2 SRFBN

The architecture used and discussed in [Section 2.5.1] was proposed at BMVC2019 by Sichuan University, University of California, University of British Columbia, and Incheon National University [61]. We have started from the implementation found in the GitHub repository [60], developed in Pytorch, and based on the work of Li et al. [61]. As with SRGAN, the existing architecture has been modified and adapted to our project.

5.2.2.a Training First Experiment

As with the previous network, we run the trained network on our aerial images. This was trained with images from the DIV2K [24] and Flickr2K [62].

Once checked the performance of the network, we do the first training with our dataset. This project has been developed with Pytorch [Section 3.1.2] and runs entirely on the Anaconda command window.

We start adapting the code to our project. To do this, we review the file "*train_SRFBN_example.json*", where some of the most important characteristics of the architecture are defined.

Relevant data about the datasets:

Code snippet 5.2.2.a: JSON with datasets information.

```
  "datasets": {
    "train": {
      "mode": "LRHR", // Required during training
      "dataroot_HR": "/mnt/data/paper99/DIV2K/Augment/DIV2K_train_HR_aug/x4",
      "dataroot_LR": "/mnt/data/paper99/DIV2K/Augment/DIV2K_train_LR_aug/x4",
      "data_type": "npy", // Binary files, recommended during training
      "n_workers": 4, // Number of threads for data loading
      "batch_size": 16, // Input batch size
      "LR_size": 40, // Input (LR) patch size

      // Data augmentation
      "use_flip": true, // Whether use horizontal and vertical flips
      "use_rot": true, // Whether rotate 90 degrees
      "noise": "." // Noise type: "." (noise free) | "G" (add Gaussian noise) | "S" (add Poisson noise)
    },
    "val": {
      "mode": "LRHR",
      "dataroot_HR": "./results/HR/Set5/x4",
      "dataroot_LR": "./results/LR/LRBI/Set5/x4",
      "data_type": "img" // Image files
    }
  },
```

Some architecture hyperparameters:

Code snippet 5.2.2.a: JSON. Some hyperparameters from the architecture.

```
    "networks": {
        "which_model": "SRFBN",
        "num_features": 32, // Number of base feature maps
        "in_channels": 3, // Number of input channels (RGB == 3)
        "out_channels": 3, // Number of output channels (RGB == 3)
        "num_steps": 4, // Number of time steps (T)
        "num_groups": 3 // Number of projection groups (G)
    },
```

General network characteristics:

Code snippet 5.2.2.a: JSON. Network information.

```
{
    "mode": "sr",
    "use_cl": true, // Whether use multiple losses (required by our SRFBN)
    "gpu_ids": [0],

    "scale": 4, // Super resolution scale
    "is_train": true,
    "use_chop": true,
    "rgb_range": 255,
    "self_ensemble": false,
    "save_image": false, // Whether saving visual results during training

    "solver": {
        "type": "ADAM",
        "learning_rate": 0.0001,
        "weight_decay": 0,
        "lr_scheme": "MultiStepLR",
        "lr_steps": [200, 400, 600, 800],
        "lr_gamma": 0.5, // Gamma for learning rate scheduler
        "loss_type": "l1",
        "manual_seed": 0,
        "num_epochs": 1000, // Number of epochs to train
        "skip_threshold": 3, // Skipping batch that has large error for stable training
        "split_batch": 1, // Split the batch into smaller chunks for less GPU memory consumption during training
        "save_ckp_step": 50,
        "save_vis_step": 1,
        "pretrain": null, // Pre-train mode: null (from scratch) | "resume" (resume from specific checkpoint) | "finetune"
(finetune a new model based on a specific model)

        "pretrained_path": "./experiments/SRFBN_in3f32_x4/epochs/last_ckp.pth", // Path to *.pth (required by
"resume"/"finetune")
        "cl_weights": [1.0, 1.0, 1.0, 1.0] // weights for multiple losses (only works when option "use_cl" is true)
    }
}
```

These parameters, defined in the JSON file, will be used for training. A parser is used to access JSON values and use them; it is defined in "*options.py*" to access their values and thus be able to use them.

The "*train.py*" file access to JSON information is accessed. And there also check the existence of the datasets defined by their paths, and then, the training process begins. For this, the solvers are imported and run, because they manage the training:

Code snippet 5.2.2.a: __init__.py

```
from .SRSolver import SRSolver

def create_solver(opt):
    if opt['mode'] == 'sr':
        solver = SRSolver(opt)
    else:
        raise NotImplementedError

    return solver
```

The "*create_solver*" method calls"*SRsolver*" defined in "*SRsolver.py*" where first, it accesses the JSON. There, it is verified that the input values for the training parameters are correct. Otherwise, the process is interrupted and an error message is displayed on the screen:

Code snippet 5.2.2.a: Verifying the value of each training parameter defined in json file

```
32        if self.is_train:
33            self.model.train()
34
35            # set cl_loss --> Defined in json file
36            if self.use_cl:
37                self.cl_weights = self.opt['solver']['cl_weights']
38                assert self.cl_weights, "[Error] 'cl_weights' is not be declared when 'use_cl' is true"
39
40            # set loss --> Defined in json file
41            loss_type = self.train_opt['loss_type']
42            if loss_type == 'l1':
43                self.criterion_pix = nn.L1Loss()
44            elif loss_type == 'l2':
45                self.criterion_pix = nn.MSELoss()
46            else:
47                raise NotImplementedError('Loss type [%s] is not implemented!'%loss_type)
48
49            if self.use_gpu:
50                self.criterion_pix = self.criterion_pix.cuda()
51
52            # set optimizer --> Defined in json file
53            weight_decay = self.train_opt['weight_decay'] if self.train_opt['weight_decay'] else 0
54            optim_type = self.train_opt['type'].upper()
55            if optim_type == "ADAM":
56                self.optimizer = optim.Adam(self.model.parameters(),
57                                            lr=self.train_opt['learning_rate'], weight_decay=weight_decay)
58            else:
59                raise NotImplementedError('Loss type [%s] is not implemented!' % optim_type)
60
61            # set lr_scheduler --> Defined in json file
62            if self.train_opt['lr_scheme'].lower() == 'multisteplr':
63                self.scheduler = optim.lr_scheduler.MultiStepLR(self.optimizer,
64                                                                self.train_opt['lr_steps'],
65                                                                self.train_opt['lr_gamma'])
66            else:
67                raise NotImplementedError('Only MultiStepLR scheme is supported!')
68
69        self.load()
70        self.print_network()
71
72        print('===> Solver Initialized : [%s] || Use CL : [%s] || Use GPU : [%s]'%(self.__class__.__name__,
73                                                                                   self.use_cl, self.use_gpu))
74        if self.is_train:
75            print("optimizer: ", self.optimizer)
76            print("lr_scheduler milestones: %s   gamma: %f"%(self.scheduler.milestones, self.scheduler.gamma))
```

Once we have access to information and it is verified, the training process starts. This process follows some concrete steps: ***initialization*** of the process according to the epoch in which we are, ***model training*** for the corresponding epoch, ***model validation*** with metrics calculation, comparison of the obtained results, and ***storing and recording of the best work out***. These blocks are defined in "*train.py*" and these used solvers are defined in "*SRsolver.py*".

Initialization and training model:

Code snippet 5.2.2.a: Initialization and Training.

```
61        # Initialization
62        solver_log['epoch'] = epoch
63
64        # Train model
65        train_loss_list = []
66        with tqdm(total=len(train_loader), desc='Epoch: [%d/%d]'%(epoch, NUM_EPOCH), miniters=1) as t:
67            for iter, batch in enumerate(train_loader):
68                solver.feed_data(batch)
69                iter_loss = solver.train_step()
70                batch_size = batch['LR'].size(0)
71                train_loss_list.append(iter_loss*batch_size)
72                t.set_postfix_str("Batch Loss: %.4f" % iter_loss)
73                t.update()
74
75        solver_log['records']['train_loss'].append(sum(train_loss_list)/len(train_set))
76        solver_log['records']['lr'].append(solver.get_current_learning_rate())
77
78        print('\nEpoch: [%d/%d]   Avg Train Loss: %.6f' % (epoch,
79                                                          NUM_EPOCH,
80                                                          sum(train_loss_list)/len(train_set)))
```

"SRsolver.py"

```
83    def feed_data(self, batch, need_HR=True):
84        input = batch['LR']
85        self.LR.resize_(input.size()).copy_(input)
86
87        if need_HR:
88            target = batch['HR']
89            self.HR.resize_(target.size()).copy_(target)
```

Code snippet 5.2.2.a: Training.

```
92    def train_step(self):
93        self.model.train()
94        self.optimizer.zero_grad()
95
96        loss_batch = 0.0
97        sub_batch_size = int(self.LR.size(0) / self.split_batch)
98        for i in range(self.split_batch):
99            loss_sbatch = 0.0
100           split_LR = self.LR.narrow(0, i*sub_batch_size, sub_batch_size)
101           split_HR = self.HR.narrow(0, i*sub_batch_size, sub_batch_size)
102           if self.use_cl:
103               outputs = self.model(split_LR)
104               loss_steps = [self.criterion_pix(sr, split_HR) for sr in outputs]
105               for step in range(len(loss_steps)):
106                   loss_sbatch += self.cl_weights[step] * loss_steps[step]
107           else:
108               output = self.model(split_LR)
109               loss_sbatch = self.criterion_pix(output, split_HR)
110
111           loss_sbatch /= self.split_batch
112           loss_sbatch.backward()
113
114           loss_batch += (loss_sbatch.item())
115
116       # for stable training
117       if loss_batch < self.skip_threshold * self.last_epoch_loss:
118           self.optimizer.step()
119           self.last_epoch_loss = loss_batch
120       else:
121           print('[Warning] Skip this batch! (Loss: {})'.format(loss_batch))
122
123       self.model.eval()
124       return loss_batch
```

Validation, metric calculation, and record:

Code snippet 5.2.2.a: Validation and Metrics.

```
        print('===> Validating...',)

        psnr_list = []
        ssim_list = []
        val_loss_list = []

        for iter, batch in enumerate(val_loader):
            solver.feed_data(batch)
            iter_loss = solver.test()
            val_loss_list.append(iter_loss)

            # calculate evaluation metrics
            visuals = solver.get_current_visual()
            psnr, ssim = util.calc_metrics(visuals['SR'], visuals['HR'], crop_border=scale)
            psnr_list.append(psnr)
            ssim_list.append(ssim)

            if opt["save_image"]:
                solver.save_current_visual(epoch, iter)

        solver_log['records']['val_loss'].append(sum(val_loss_list)/len(val_loss_list))
        solver_log['records']['psnr'].append(sum(psnr_list)/len(psnr_list))
        solver_log['records']['ssim'].append(sum(ssim_list)/len(ssim_list))

        # record the best epoch
        epoch_is_best = False
        if solver_log['best_pred'] < (sum(psnr_list)/len(psnr_list)):
            solver_log['best_pred'] = (sum(psnr_list)/len(psnr_list))
            epoch_is_best = True
            solver_log['best_epoch'] = epoch

        print("[%s] PSNR: %.2f   SSIM: %.4f   Loss: %.6f   Best PSNR: %.2f in Epoch: [%d]" % (val_set.name(),
```

Code snippet 5.2.2.a: Testing.

```
    def test(self):
        self.model.eval()
        with torch.no_grad():
            forward_func = self._overlap_crop_forward if self.use_chop else self.model.forward
            if self.self_ensemble and not self.is_train:
                SR = self._forward_x8(self.LR, forward_func)
            else:
                SR = forward_func(self.LR)

            if isinstance(SR, list):
                self.SR = SR[-1]
            else:
                self.SR = SR

        self.model.train()
        if self.is_train:
            loss_pix = self.criterion_pix(self.SR, self.HR)
            return loss_pix.item()
```

For training we use the SRFBN architecture, described in [Section 2.5.1]:

Code snippet 5.2.2.a: Feature Extraction Block, Feedback Block and Reconstruction Block.

```
class SRFBN(nn.Module):
    def __init__(self, in_channels, out_channels, num_features, num_steps, num_groups, upscale_factor, act_type = 'prelu',
norm_type = None):
        super(SRFBN, self).__init__()

        if upscale_factor == 2:
            stride = 2
            padding = 2
            kernel_size = 6
        elif upscale_factor == 3:
            stride = 3
            padding = 2
            kernel_size = 7
        elif upscale_factor == 4:
            stride = 4
            padding = 2
            kernel_size = 8
        elif upscale_factor == 8:
            stride = 8
            padding = 2
            kernel_size = 12

        self.num_steps = num_steps
        self.num_features = num_features
        self.upscale_factor = upscale_factor

        rgb_mean = (0.4488, 0.4371, 0.4040)
        rgb_std = (1.0, 1.0, 1.0)
        self.sub_mean = MeanShift(rgb_mean, rgb_std)

        # LR feature extraction block
        self.conv_in = ConvBlock(in_channels, 4*num_features,
                                 kernel_size=3,
                                 act_type=act_type, norm_type=norm_type)
        self.feat_in = ConvBlock(4*num_features, num_features,
                                 kernel_size=1,
                                 act_type=act_type, norm_type=norm_type)

        # basic block
        self.block = FeedbackBlock(num_features, num_groups, upscale_factor, act_type, norm_type)

        # reconstruction block
        self.out = DeconvBlock(num_features, num_features,
                               kernel_size=kernel_size, stride=stride, padding=padding,
                               act_type='prelu', norm_type=norm_type)
        self.conv_out = ConvBlock(num_features, out_channels,
                                  kernel_size=3,
                                  act_type=None, norm_type=norm_type)

        self.add_mean = MeanShift(rgb_mean, rgb_std, 1)
```


Code snippet 5.2.2.a: SRFBN Architecture. Forward Method.

```
    def forward(self, x):
        self._reset_state()

        x = self.sub_mean(x)

        inter_res = nn.functional.interpolate(x, scale_factor=self.upscale_factor, mode='bilinear', align_corners=False)

        x = self.conv_in(x)
        x = self.feat_in(x)

        outs = []
        for _ in range(self.num_steps):
            h = self.block(x)

            h = torch.add(inter_res, self.conv_out(self.out(h)))
            h = self.add_mean(h)
            outs.append(h)

        return outs # return output of every timesteps
```

The characteristic feedback block of this network are illustrated below:

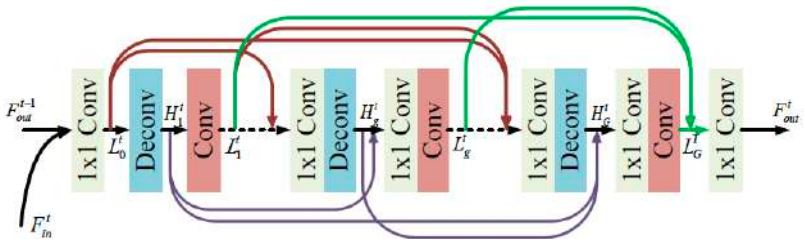

Fig. 5.2.2.a Feedback architecture.

Code snippet 5.2.2.a: SRFBN Architecture. Feedback Block.

```
class FeedbackBlock(nn.Module):
    def __init__(self, num_features, num_groups, upscale_factor, act_type, norm_type):
        super(FeedbackBlock, self).__init__()
        if upscale_factor == 2:
            stride = 2
            padding = 2
            kernel_size = 6
        elif upscale_factor == 3:
            stride = 3
            padding = 2
            kernel_size = 7
        elif upscale_factor == 4:
            stride = 4
            padding = 2
            kernel_size = 8
        elif upscale_factor == 8:
            stride = 8
            padding = 2
            kernel_size = 12

        self.num_groups = num_groups

        self.compress_in = ConvBlock(2*num_features, num_features,
                                     kernel_size=1,
                                     act_type=act_type, norm_type=norm_type)

        self.upBlocks = nn.ModuleList()
        self.downBlocks = nn.ModuleList()
        self.uptranBlocks = nn.ModuleList()
        self.downtranBlocks = nn.ModuleList()

        for idx in range(self.num_groups):
            self.upBlocks.append(DeconvBlock(num_features, num_features,
                                             kernel_size=kernel_size, stride=stride, padding=padding,
                                             act_type=act_type, norm_type=norm_type))
            self.downBlocks.append(ConvBlock(num_features, num_features,
                                             kernel_size=kernel_size, stride=stride, padding=padding,
                                             act_type=act_type, norm_type=norm_type, valid_padding=False))
            if idx > 0:
                self.uptranBlocks.append(ConvBlock(num_features*(idx+1), num_features,
                                                   kernel_size=1, stride=1,
                                                   act_type=act_type, norm_type=norm_type))
                self.downtranBlocks.append(ConvBlock(num_features*(idx+1), num_features,
                                                     kernel_size=1, stride=1,
                                                     act_type=act_type, norm_type=norm_type))

        self.compress_out = ConvBlock(num_groups*num_features, num_features,
                                      kernel_size=1,
                                      act_type=act_type, norm_type=norm_type)

        self.should_reset = True
        self.last_hidden = None
```

Code snippet 5.2.2.a: Modificación. Acceso a nuestro dataset.

```
    def forward(self, x):
        if self.should_reset:
            self.last_hidden = torch.zeros(x.size()).cuda()
            self.last_hidden.copy_(x)
            self.should_reset = False

        x = torch.cat((x, self.last_hidden), dim=1)
        x = self.compress_in(x)

        lr_features = []
        hr_features = []
        lr_features.append(x)

        for idx in range(self.num_groups):
            LD_L = torch.cat(tuple(lr_features), 1)    # when idx == 0, lr_features == [x]
            if idx > 0:
                LD_L = self.uptranBlocks[idx-1](LD_L)
            LD_H = self.upBlocks[idx](LD_L)

            hr_features.append(LD_H)

            LD_H = torch.cat(tuple(hr_features), 1)
            if idx > 0:
                LD_H = self.downtranBlocks[idx-1](LD_H)
            LD_L = self.downBlocks[idx](LD_H)

            lr_features.append(LD_L)

        del hr_features
        output = torch.cat(tuple(lr_features[1:]), 1)   # leave out input x, i.e. lr_features[0]
        output = self.compress_out(output)

        self.last_hidden = output

        return output
```

We run the first training with the default hyperparameters and architectures; these will be explained in the next [Section 4.2.2.b]. We store the loss values in a .csv file.

4.2.2.b Hyperparameters

In this case, the trial and error methodology is used too. These hyperparameters have been modified and applied, and their results have been compared and analyzed. Results and a table with all the combinations used were recorded in Microsoft Excel, *pruebas_resultados.xlsx*. Some of the hyperparameter modifications tested:

- ***Number of workers***: defines the number of threads for data loading. In this case, 4.

- ***Batch Size***: 16

- ***Number of epochs***: 100 and 1.000.000.

- ***Number of feature maps (nº of filters)***: 32 and 64.

- ***Activation Function***: PReLU [104].

- ***Number of time steps (T)*****:** 4.

- ***Number of projection groups in the FB (G)***: 3 and 6.

- ***Optimizer***: Adam [53] with β1 = 0.9 and weight decay = 0 [105].

- ***Learning rate***: 10^{-4}. This rate is multiplied by 0.5 ("*lr_gamma*") for every 200 epochs ("*lr_steps*").

- ***Loss type***: L1 and L2.

- ***Mode for upsampling - FB***: bilinear. Other methods of interpolation as bicubic are available too, but the bilinear method has the best results. For that, bilinear will be applied.

4.2.2.c Testing

To evaluate a model, we apply the generated and saved training weights, and then, this will be applied to images from the testing dataset. For that, we have to run in the *Anaconda shell* this command:

```
(base) C:\Users\ionut>conda activate SRFBN

(SRFBN) C:\Users\ionut>cd Desktop/Monica

(SRFBN) C:\Users\ionut\Desktop\Monica>cd SRFBN_CVPR19-master

(SRFBN)  C:\Users\ionut\Desktop\Monica\SRFBN_CVPR19-master>python  test.py
-opt options/test/test_SRFBN_example.json
```

First of all, you must adapt the file "*test.json*" with the information needed for the evaluation. Some information as the URL of LR images and its HR URL too, if these exist. Also, you need to specify the architecture dimensions of the model that you are going to evaluate and its path. It is very important to fill the option "*is_train*" as "*false*":

Code snippet 4.2.2.c: Test JSON.

```
{
    "mode": "sr",
    "use_cl": true,
    "gpu_ids": [0],

    "scale": 4,
    "degradation": "BI",
    "is_train": false,
    "use_chop": true,
    "rgb_range": 255,
    "self_ensemble": false,


    "datasets": {
        "PNOA_MA": {
            "mode": "LRHR",
            "dataroot_HR": "./results/HR/PNOA_MA/x4",
            "dataroot_LR": "./results/LR/LRBI/PNOA_MA/x4",
            "data_type": "img"
        },
        "PNOA10": {
            "mode": "LRHR",
            "dataroot_HR": "./results/HR/PNOA10/x4",
            "dataroot_LR": "./results/LR/LRBI/PNOA10/x4",
            "data_type": "img"

        "PLEIADES": {
            "mode": "LRHR",
            "dataroot_HR": "./results/HR/PLEIADES/x4",
            "dataroot_LR": "./results/LR/LRBI/PLEIADES/x4",
            "data_type": "img"
        }
    },

    "networks": {
        "which_model": "SRFBN",
        "num_features": 32,
        "in_channels": 3,
        "out_channels": 3,
        "num_steps": 4,
        "num_groups": 3
    },

    "solver": {
        "pretrained_path": "./models/SRFBN_in3f32_x4_prueba6/epochs/best_ckp.pth"
    },

    "name": "P6_SRFBN_in3f32_x4-best_ckp.pth"
}
```

The command shown above, calls the file"*test.py*", which we have already modified to generate individual folders with the SR results from each test dataset and write the metrics in a .csv file. To evaluate the model, the methods defined in "*solvers/ SRsolver.py*" are called:

Code snippet 4.2.2.c: Testing. Creation of .csv with metrics values.

```
    # initial configure
    scale = opt['scale']
    degrad = opt['degradation']
    network_opt = opt['networks']
    folder = opt['name']
    csv_name = opt['name']
    model_name = network_opt['which_model'].upper()
    if opt['self_ensemble']: model_name += 'plus'

    # create test dataloader
    bm_names =[]
    test_loaders = []
    for _, dataset_opt in sorted(opt['datasets'].items()):
        test_set = create_dataset(dataset_opt)
        test_loader = create_dataloader(test_set, dataset_opt)
        test_loaders.append(test_loader)
        print('===> Test Dataset: [%s]   Number of images: [%d]' % (test_set.name(), len(test_set)))
        bm_names.append(test_set.name())

    # create solver (and load model)
    solver = create_solver(opt)
    # Test phase
    print('===> Start Test')
    print("==================================================")
    print("Method: %s || Scale: %d || Degradation: %s"%(model_name, scale, degrad))

    for bm, test_loader in zip(bm_names, test_loaders):
        print("Test set : [%s]"%bm)

        sr_list = []
        path_list = []

        total_psnr = []
        total_ssim = []
        total_time = []
        for iter, batch in enumerate(test_loader):
            test_dataset = opt['test_dataset']
            solver.feed_data(batch, need_HR=need_HR)

            # calculate forward time
            t0 = time.time()
            solver.test()
            t1 = time.time()
            total_time.append((t1 - t0))

            visuals = solver.get_current_visual(need_HR=need_HR)
            sr_list.append(visuals['SR'])

            # calculate PSNR/SSIM metrics on Python
            if need_HR:
                psnr, ssim = util.calc_metrics(visuals['SR'], visuals['HR'], crop_border=scale)
                if not os.path.exists("./results/SR/docs/"+folder+"/"):
                    os.makedirs("./results/SR/docs/"+folder+"/")

                if not os.path.exists("./results/SR/docs/"+folder+"/"+csv_name+"_"+bm+".csv"):
                        d = open("./results/SR/docs/"+folder+"/"+csv_name+"_"+bm+".csv",'a+')
                        d.write("Name;PSNR;SSIM\n")
                        d.write("%s;%.4f;%.4f\n" % (os.path.basename(batch['LR_path'][0]),psnr,ssim))
                        d.close()
                else:
                    d = open("./results/SR/docs/"+folder+"/"+csv_name+"_"+bm+".csv",'a+')
                    d.write("%s;%.4f;%.4f\n" % (os.path.basename(batch['LR_path'][0]),psnr,ssim))
                    d.close()

                total_psnr.append(psnr)
                total_ssim.append(ssim)
                path_list.append(os.path.basename(batch['HR_path'][0]).replace('HR', model_name))
                print("[%d/%d] %s || PSNR(dB)/SSIM: %.2f/%.4f || Timer: %.4f sec ." % (iter+1, len(test_loader),
                                                                                       os.path.basename(batch['LR_path'][0]),
                                                                                       psnr, ssim,
                                                                                       (t1 - t0)))
            else:
                path_list.append(os.path.basename(batch['LR_path'][0]))
                print("[%d/%d] %s || Timer: %.4f sec ." % (iter + 1, len(test_loader),
                                                          os.path.basename(batch['LR_path'][0]),
                                                          (t1 - t0)))
```

Code snippet 4.2.2.c: Testing. Saving resultant SR images.

```
        # save SR results
        if need_HR:
            save_img_path = os.path.join('./results/SR/'+degrad, model_name, folder , "x%d"%scale, bm)
        else:
            save_img_path = os.path.join('./results/SR/NoMetrics/'+model_name, folder, "x%d"%scale,bm)

        print("===> Saving SR images of [%s]... Save Path: [%s]\n" % (bm, save_img_path))

        if not os.path.exists(save_img_path): os.makedirs(save_img_path)
        for img, name in zip(sr_list, path_list):
            imageio.imwrite(os.path.join(save_img_path, name), img)

    print("==================================================")
    print("===> Finished !")
```

4.2.2.d Best Experiments

Once the hyperparameters have been adapted, we make some possible combinations. In this section, we are going to show the four architectures with the best results.

As we have mentioned, these trainings are quite expensive, they require specific hardware as well as the availability of it for several days. For this network, the number of experiments and proposed combinations is considerably less than those carried out with SRGAN, due to mentioned requirements and some setbacks during the project. Even so, we can make conclusions, propose possible future combinations and the results that are expected from them. Combinations are in the excel file with results.

We evaluate the models with the three test datasets: PNOA MA, PNOA 10, and Pleiades Satellite; we will also calculate the quantitative metrics to draw more objective conclusions. These results are recorded in excel "*experiments.xlsx*", and as can be seen, the metrics values obtained by proposed and trained combinations are notably lower than SRGAN calculated metrics. But before explaining why, we are going to comment on the testing strategy we have followed.

This network has been trained with the same dataset that has been used to train the SRGAN. But in this network, we have varied the number of images with which we fed the network to speed up the training and see the network behaviour. Therefore, when we mention "*Checked dataset*" we refer to the same dataset, but reduced by 10%, i.e. 28.345 images, obtained after deleting some images; the same for the validation dataset, it is reduced by 10%. Another validation dataset is created, it is composed of 2,000 randomly selected images from the "*Original validation dataset*". In summary, we are going to work with the same training and validation datasets but with the following variations:

Table 4.2.2.d-I: Variaciones en los dataset de entrenamiento y validación (SRFBN).

Dataset	**Nº of Images**
Training - Original	31.496
Validation - Original	7.875
Training - "Checked"	28.345
Valid - "Checked"	7.087
Valid - "Reduced"	2.000

How we have explained, these variations were applied to speed up training experiments. We work with a lot of data and our hardware has limitations, so we wanted to release some memory and resources. Anyway, our goal is to analyze the network behaviour, so we prefer to save time and test different combinations to know how the network works and responds to any new change or variation and then, extrapolate these changes to a bigger training regardless of time limitations.

The principal change that we have tried is the number of epochs. The maximum of epochs that we have tried is a hundred, and we have seen how the network acts. So we can conclude, the number of epochs is an important hyperparameter. This has sense because how we explained in [*Section 2.5.2*], this network works following the curriculum learning strategy, it begins working with easier tasks and it is improving to end up with the most difficult. So if we train our network with a small number of epochs, it does not have enough time to improve. So, increasing the number of epochs would be a possible variation for future experiments. As in the other network, Adam optimizer and PreLu activation function are used in SRFBN. The interpolation method used to upsampling the images in each block is the bilinear method as in [61]; only nearest, linear, bilinear, bicubic, trilinear, and area are supported [108]. Related to the number of time steps (T) and projection groups (G), we use values proposed by [61], and we confirm that a larger number of G and T, better results. Weights are saved each 50 epochs and there were saved too if the network has the best results during training, till that moment, i.e., if the loss of the network is the lowest, or what is the same, if quantitative metrics are the highest. Finally, obtained results with this network are not as good as we expected, due to the limitations already mentioned. But there are enough to know how it works, so the experiments with the best results and that we will analyze are:

Table 4.2.2.d-II: Hyperparameters values of the four best models - SRFBN.

Ranking	Nº Filters	T	G	Nº Epochs	Dataset
1	64	4	6	1.000.000.000	Default
2	32	4	3	1.000.000.000	Default
3	32	4	3	100	Train: Checked Test: 2.000
4	32	4	3	100	Train: Checked Test: Checked

One more time, we can see how the dataset used to train our model is very important. The best results are obtained with a network trained with images that were not related to our topic, and we can see how training with our images and reducing the number of epochs, the most important hyperparameter to this network, from a million to a hundred, our results are less than one point of difference with the highest score. So we can suppose that if we trained our model with the same architecture and conditions that the model proposed in [60] but with our dataset, our results would be considerably increased, thus surpassing the results we have now. Hence, we propose to try these architectures in the future, increasing the number of epochs to 100.000 and 1.000.000, and then see the differences and their improvements.

The hardware used to train this network has to be powerful and with large memory. Therefore, to train these kinds of architectures, SSD disk and GPU are highly recommended.

The average processing time that we have required to execute these experiments has been a month, including several failed initial experiments. Normally, to train each model with the shown architectures and only a hundred epochs, about three days were needed.

Now, we are going to show the architecture used for each model. There are two different architectures, one for the first experiment and the other for the rest of the proposed combinations.

Model 2

NºFilters (64) + T (4) + G (6)

```
SRFBN(
  (sub_mean): MeanShift(3, 3, kernel_size=(1, 1), stride=(1, 1))
  (conv_in): Sequential(
    (0): Conv2d(3, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): PReLU(num_parameters=1)
  )
  (feat_in): Sequential(
    (0): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1))
    (1): PReLU(num_parameters=1)
  )
  (block): FeedbackBlock(
    (compress_in): Sequential(
      (0): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
      (1): PReLU(num_parameters=1)
    )
    (upBlocks): ModuleList(
      (0): Sequential(
        (0): ConvTranspose2d(64, 64, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
      (1): Sequential(
        (0): ConvTranspose2d(64, 64, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
      (2): Sequential(
        (0): ConvTranspose2d(64, 64, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
      (3): Sequential(
        (0): ConvTranspose2d(64, 64, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
      (4): Sequential(
        (0): ConvTranspose2d(64, 64, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
      (5): Sequential(
        (0): ConvTranspose2d(64, 64, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
    )
    (downBlocks): ModuleList(
      (0): Sequential(
        (0): Conv2d(64, 64, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
      (1): Sequential(
        (0): Conv2d(64, 64, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
      (2): Sequential(
        (0): Conv2d(64, 64, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
      (3): Sequential(
        (0): Conv2d(64, 64, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
      (4): Sequential(
        (0): Conv2d(64, 64, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
      (5): Sequential(
        (0): Conv2d(64, 64, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
    )
    (uptranBlocks): ModuleList(
      (0): Sequential(
        (0): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
        (1): PReLU(num_parameters=1)
      )
      (1): Sequential(
        (0): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1))
        (1): PReLU(num_parameters=1)
      )
      (2): Sequential(
        (0): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1))
        (1): PReLU(num_parameters=1)
      )
      (3): Sequential(
        (0): Conv2d(320, 64, kernel_size=(1, 1), stride=(1, 1))
        (1): PReLU(num_parameters=1)
      )
      (4): Sequential(
        (0): Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1))
        (1): PReLU(num_parameters=1)
      )
    )
    (downtranBlocks): ModuleList(
      (0): Sequential(
        (0): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
        (1): PReLU(num_parameters=1)
      )
      (1): Sequential(
        (0): Conv2d(192, 64, kernel_size=(1, 1), stride=(1, 1))
        (1): PReLU(num_parameters=1)
      )
      (2): Sequential(
        (0): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1))
        (1): PReLU(num_parameters=1)
      )
      (3): Sequential(
        (0): Conv2d(320, 64, kernel_size=(1, 1), stride=(1, 1))
        (1): PReLU(num_parameters=1)
      )
      (4): Sequential(
        (0): Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1))
        (1): PReLU(num_parameters=1)
      )
    )
    (compress_out): Sequential(
      (0): Conv2d(384, 64, kernel_size=(1, 1), stride=(1, 1))
      (1): PReLU(num_parameters=1)
    )
  )
  (out): Sequential(
    (0): ConvTranspose2d(64, 64, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
    (1): PReLU(num_parameters=1)
  )
  (conv_out): Sequential(
    (0): Conv2d(64, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (add_mean): MeanShift(3, 3, kernel_size=(1, 1), stride=(1, 1))
)
```

Models 1, 3 and 4.

NºFilters (32) + T (4) + G (3)

```
SRFBN(
  (sub_mean): MeanShift(3, 3, kernel_size=(1, 1), stride=(1, 1))
  (conv_in): Sequential(
    (0): Conv2d(3, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): PReLU(num_parameters=1)
  )
  (feat_in): Sequential(
    (0): Conv2d(128, 32, kernel_size=(1, 1), stride=(1, 1))
    (1): PReLU(num_parameters=1)
  )
  (block): FeedbackBlock(
    (compress_in): Sequential(
      (0): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1))
      (1): PReLU(num_parameters=1)
    )
    (upBlocks): ModuleList(
      (0): Sequential(
        (0): ConvTranspose2d(32, 32, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
      (1): Sequential(
        (0): ConvTranspose2d(32, 32, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
      (2): Sequential(
        (0): ConvTranspose2d(32, 32, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
    )
    (downBlocks): ModuleList(
      (0): Sequential(
        (0): Conv2d(32, 32, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
      (1): Sequential(
        (0): Conv2d(32, 32, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
      (2): Sequential(
        (0): Conv2d(32, 32, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
        (1): PReLU(num_parameters=1)
      )
    )
    (uptranBlocks): ModuleList(
      (0): Sequential(
        (0): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1))
        (1): PReLU(num_parameters=1)
      )
      (1): Sequential(
        (0): Conv2d(96, 32, kernel_size=(1, 1), stride=(1, 1))
        (1): PReLU(num_parameters=1)
      )
    )
    (downtranBlocks): ModuleList(
      (0): Sequential(
        (0): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1))
        (1): PReLU(num_parameters=1)
      )
      (1): Sequential(
        (0): Conv2d(96, 32, kernel_size=(1, 1), stride=(1, 1))
        (1): PReLU(num_parameters=1)
      )
    )
    (compress_out): Sequential(
      (0): Conv2d(96, 32, kernel_size=(1, 1), stride=(1, 1))
      (1): PReLU(num_parameters=1)
    )
  )
  (out): Sequential(
    (0): ConvTranspose2d(32, 32, kernel_size=(8, 8), stride=(4, 4), padding=(2, 2))
    (1): PReLU(num_parameters=1)
  )
  (conv_out): Sequential(
    (0): Conv2d(32, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (add_mean): MeanShift(3, 3, kernel_size=(1, 1), stride=(1, 1))
)
```

5. RESULTS AND DISCUSSION

Once the architectures with the best results have been presented, we will visually compare the generated SR images and their metrics.

For the calculation of the metrics, we define two methods in python that we will apply to all the models to evaluate it objectively:

Code snippet 5: Testing. Creation of .csv with metrics values.

```
#PSNR

from math import log10, sqrt
import cv2
import numpy as np
import tensorflow as tf

def PSNR(original, compressed):
    mse = np.mean((original - compressed) ** 2)
    if(mse == 0):  # MSE is zero means no noise is present in the signal .
                   # Therefore PSNR have no importance.
        return 100
    max_pixel = 255.0
    psnr = 20 * log10(max_pixel / sqrt(mse))
    return psnr

#SSIM

def SSIM(img1_path, img2_path):
    imageA = load_image(img1_path)
    imageB = load_image(img2_path)
    valor = tf.image.ssim(imageA,imageB,max_val=255)
    return valor
```

We write the results in a .csv and print on the screen the average value of PSNR and SSIM achieved with that model:

Code snippet 5: Testing. Creation of .csv with metrics values.

```
def psnr_ssim_txt(sr_path, hr_path, txt_name):
    if os.path.exists(sr_path+".DS_Store"):
        os.remove(sr_path+".DS_Store")
    if os.path.exists(hr_path+".DS_Store"):
        os.remove(hr_path+".DS_Store")
    name_sr=os.listdir(sr_path)
    name_sr.sort()
    name_hr=os.listdir(hr_path)
    name_hr.sort()
    psnr_mean=0
    ssim_mean=0
    for i in enumerate(name_sr):
        img_sr=sr_path + str(i[1])
        img_hr=hr_path + str(name_hr[i[0]])
        hr = load_image(img_hr)
        sr = load_image(img_sr)
        psnr=PSNR(hr,sr)
        psnr_mean = psnr_mean + psnr
        ssim=SSIM(img_hr,img_sr)
        ssim_mean = ssim_mean + ssim

        if i[0]==0:
            f = open(hr_path.replace("HR/1024x1024","SR/DOCS")+txt_name,'w')
            f.write('Name;PSNR;SSIM')
            print('Name;PSNR;SSIM')
            f.close()

        f = open(hr_path.replace("HR/1024x1024","SR/DOCS")+txt_name,'a')
        f.write('\n'+ str(i[1].replace(".png","")) + ';'+ str("{:.4f}".format(psnr))
        f.close()
        print('\n'+ str(i[1].replace(".png","")) + ';'+ str("{:.4f}".format(psnr)).r

    print('\n\n\nPSRN medio = ' + str("{:.4f}".format(psnr_mean/len(name_sr)).replac
```

On the one hand, the LR images from the test dataset upsampled by different interpolation methods seen in [*Sección 2.3*]. Lanczos interpolation [*Sección 2.3.4*][22] obtains the best results:

Table 5-I: Interpolation Upsampling. PSNR and SSIM mean for each test dataset and each method.

INTERPOLATION

	PNOA_MA		**PNOA10**		**PLEIADES**		**MEAN**	
	PSNR	**SSIM**	**PSNR**	**SSIM**	**PSNR**	**SSIM**	**PSNR**	**SSIM**
NN	30,8566	0,5772	**31,2151**	**0,6672**	27,8645	0,2579	29,9787	0,5008
Bilinear	30,9465	0,6078	**31,5584**	**0,7253**	27,8645	0,2681	30,1231	0,5337
Bicubic	30,7285	0,5691	**31,0511**	**0,6781**	27,8692	0,2701	29,8829	0,5058
Lanczos	30,9583	0,61	**31,5775**	**0,7285**	27,8647	0,2677	**30,1335**	**0,5354**

Table 5-II: SR-GAN Upsampling. PSNR and SSIM mean for each test dataset and each model. [*Section 4.2.1.d*]

SRGAN

	PNOA_MA		**PNOA10**		**PLEIADES**		**MEAN**	
	PSNR	**SSIM**	**PSNR**	**SSIM**	**PSNR**	**SSIM**	**PSNR**	**SSIM**
1	**29,8056**	**0,6254**	29,7606	0,7027	27,973	0,2617	**29,1797**	**0,5299**
2	29,2688	0,5503	**29,6371**	**0,6638**	27,9578	0,2602	28,9546	0,4914
3	29,1976	0,6003	**29,7314**	**0,7059**	27,925	0,2673	28,9513	0,5245
4	**29,647**	**0,5567**	29,1266	0,6438	28,0065	0,2573	28,9267	0,4859

Table 5-III: SRFBN Upsampling. PSNR and SSIM mean for each test dataset and each model. [*Section 4.2.2.d*]

SRFBN

	PNOA_MA		PNOA10		PLEIADES		MEAN	
	PSNR	**SSIM**	**PSNR**	**SSIM**	**PSNR**	**SSIM**	**PSNR**	**SSIM**
1	31,423	0,6944	**32,2505**	**0,7875**	27,8608	0,2606	**30,5114**	**0,5808**
2	31,3551	0,6847	**32,1988**	**0,7867**	27,8602	0,2613	30,4714	0,5776
3	**30,6503**	**0,6392**	30,5772	0,7600	27,9376	0,2718	**29,7217**	**0,5570**
4	**30,4918**	**0,6383**	30,4698	0,7523	27,9403	0,2734	29,6340	0,5547

The rest of the experiments and their respective results, are recorded in the results excel.

Based on the metric values, we can expect to generate better SR images with the Lanczos interpolation method than those generated with any of the SRGAN-trained models; and these will be similar to SRs generated by SRFBN models. In the next section [Section 5.1] we will see how this is not the case, but why?

To begin with, when calculating the metrics globally, ie, expressing it from the average of the values obtained from the three datasets; we are giving the same weight to the results obtained from the Pleiades image dataset as the rest, and this should not be like that. As we mentioned in [Section 3.1.3], we do not have Pleiades HR images as reference. We are generating SR images from LR but when calculating the metrics, we use images obtained from another flight as HR references. In this case, we are using images from the PNOA with the same resolution that we want to obtain after upsampling. These HR images have been captured by another sensor in a different flight too, but the most influencing change is that these images have been captured at different times. This time variation has a huge impact on quantitative metrics because HR and LR images are not equal. These images were taken on different dates at different hours. Surface changes, the light or weather conditions, cause changes between both images, and this affects the rigor of the metrics.

5.1. SR Images.

We are going to differentiate each testing dataset and the diverse surfaces: vegetation, urban and coastal areas, constructions, roads, and windmills. Some SR images will be shown below, with a small detailed analysis. A general result discussion will be explained at the end of this chapter in [Section 5.1.d]:

5.1.a PNOA MA

- ***INTERPOLATION***

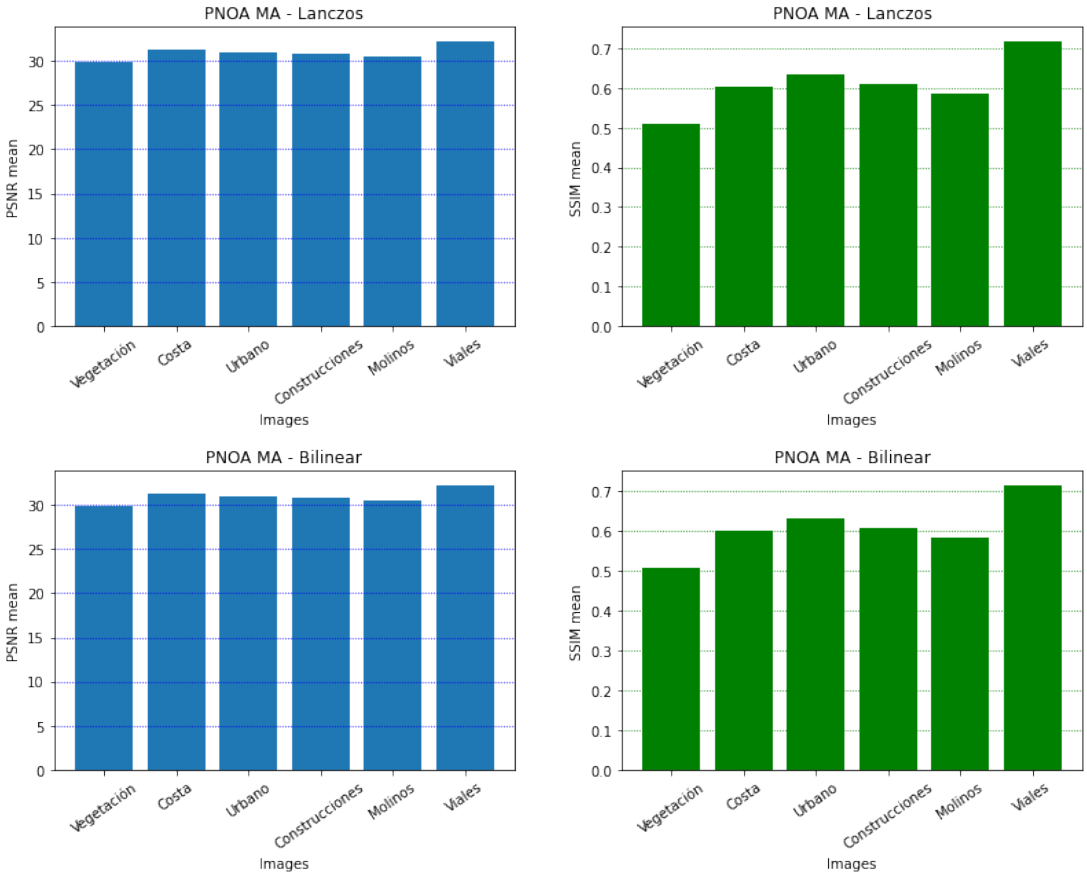

Fig. 5.1.a-I PNOA MA - Interpolation metrics.

- ***SRGAN***

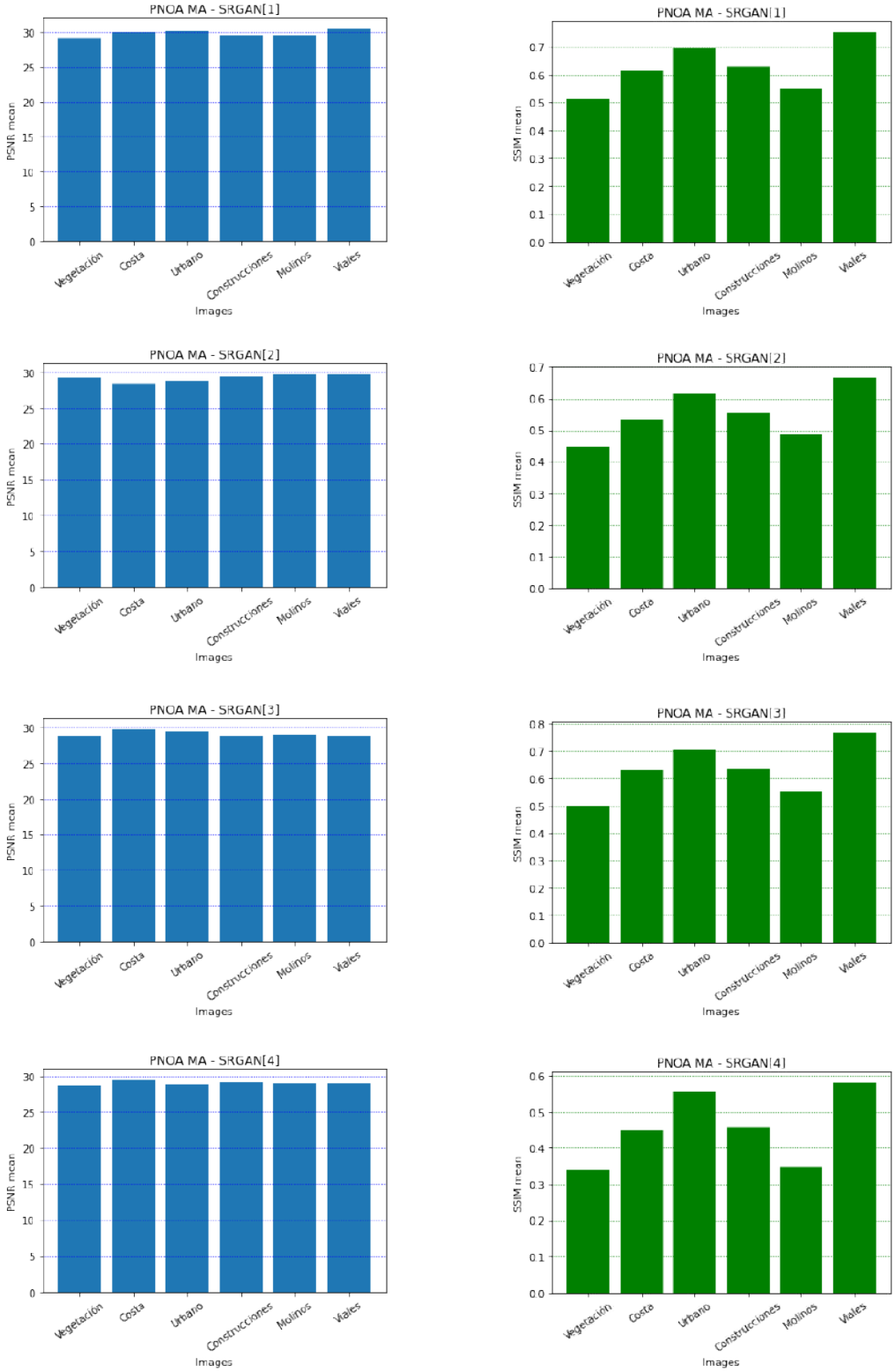

Fig. 5.1.a-II PNOA MA - SRGAN metrics.

- ***SRFBN***

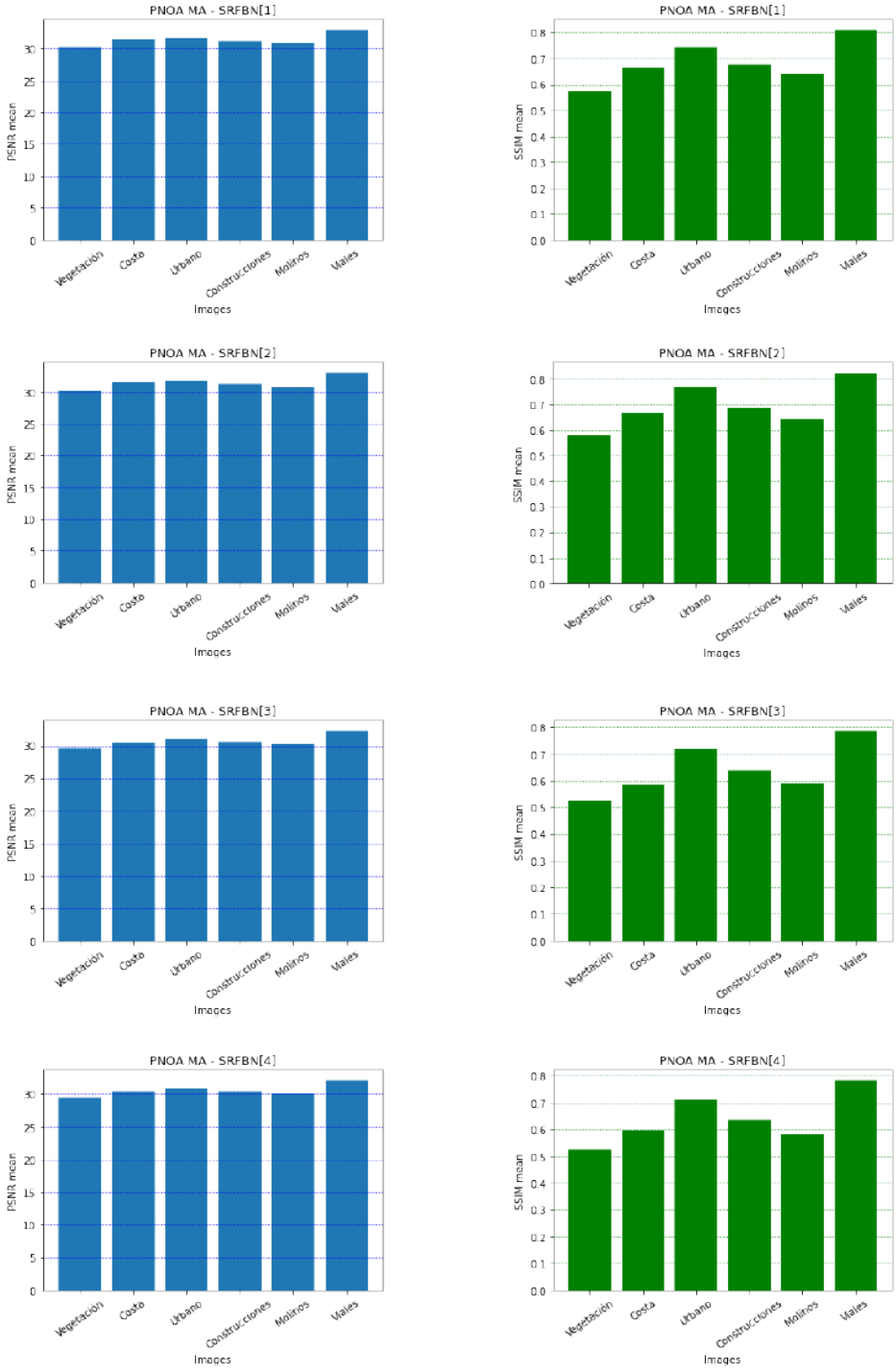

Fig. 5.1.a-III PNOA MA - SRFBN metrics.

The PSNR values are around the mean with small variations. The highest values are obtained in the images of roads, followed by the images of urban areas; the vegetation images have the lowest values. This pattern repeats for all three methods [Fig. 5.1.a-I], [Fig. 5.1.a-II], and [Fig. 5.1.a-III].

- ***VEGETATION***

Fig. 5.1.a-IV PNOA MA - SR images of vegetation.

In [Fig. 5.1.a-IV] are plotted two examples of vegetation images from the PNOA MA dataset after applying the proposed upsampling methods. In this example, we can see how the results of trained models are better than interpolation methods.

Visually, SR-generated images by interpolation methods, i.e., lanczos and bilinear, both are more blurred than other SR images. SRFBN results are visually better than interpolation but these are blurred too.

Fig. 5.1.a-V PNOA MA - SR images of vegetation.

In SRGAN images the generated results are so good, these have more texture and images are sharper but also, these are darker than the original image. This is specially appreciated in the third model, SRGAN-P3:

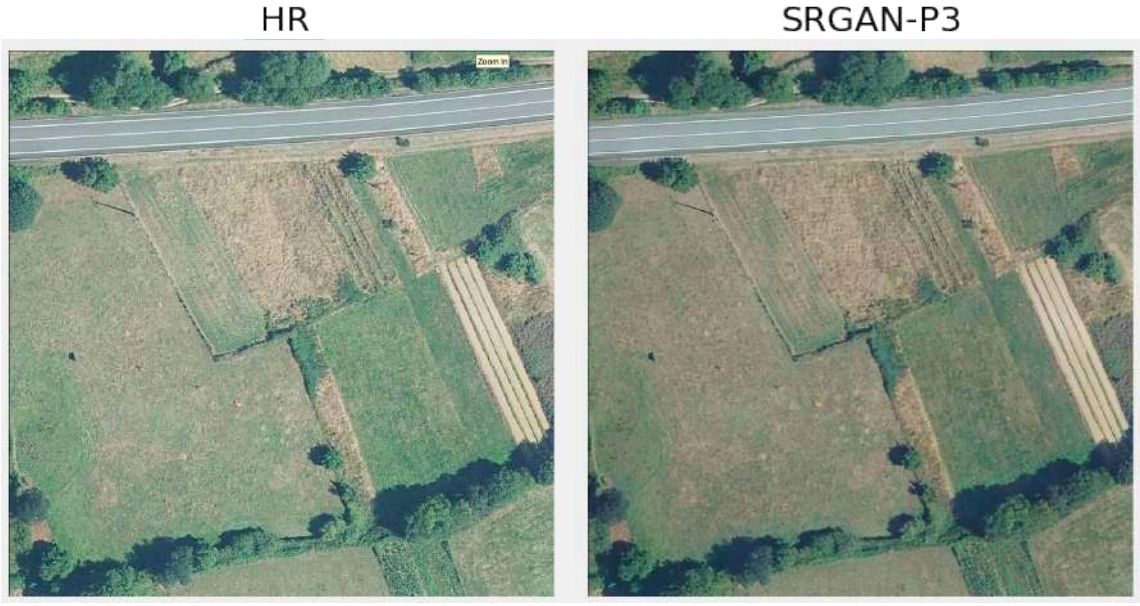

Fig. 5.1.a-VI PNOA MA - SR images of vegetation.

In conclusion, SR-generated images by SRGAN are the best results but they can be improved.

- ***WINDMILLS***

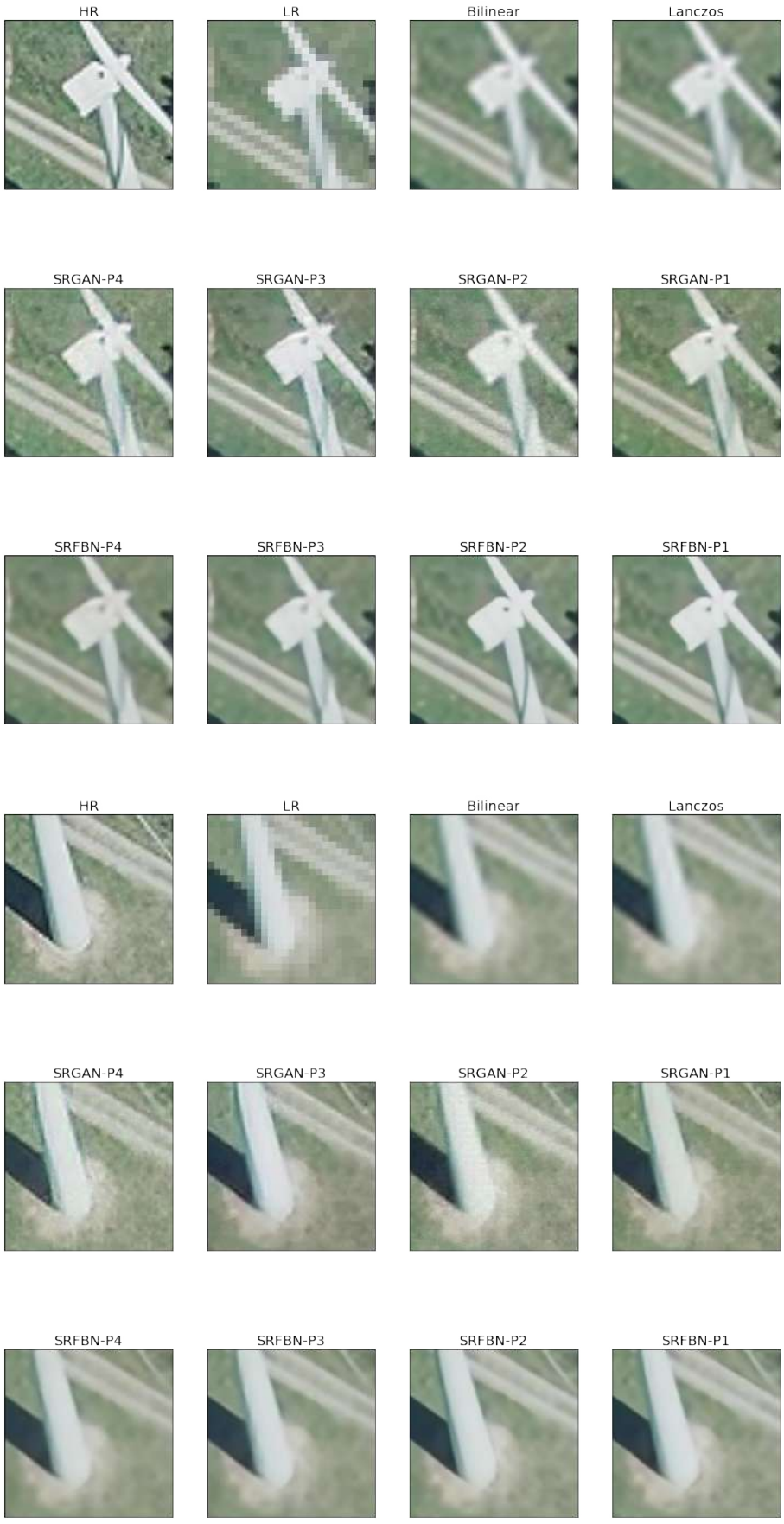

Fig. 5.1.a-VII PNOA MA - SR images of windmills (top and bottom).

- ***HUMAN CONSTRUCTIONS***

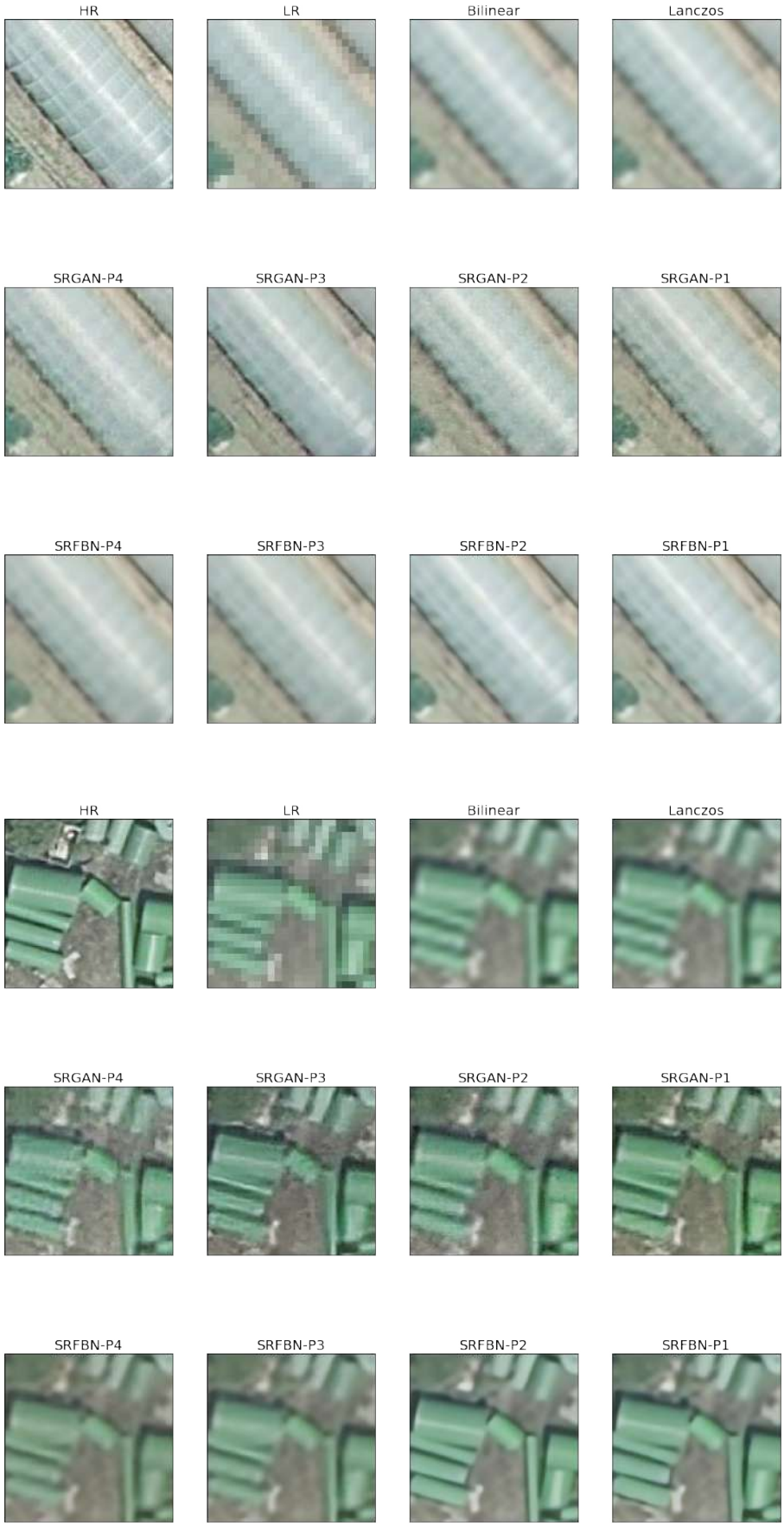

Fig. 5.1.a-VIII PNOA MA - SR images of human constructions (greenhouse and other objects).

In [Fig. 5.1.a-VII] are shown an image of a windmill, zoomed to see in detail the bottom of its structure, its base, and the top of the mill. In these examples, the improvement of upsampling models is remarkable. For human constructions images shown in [Fig. 5.1.a-VIII] results are satisfactory.

Again the generated images by interpolation method are the worst, there is an upgrade concerning the LR image, but objects keep being fuzzy although distinguishable.

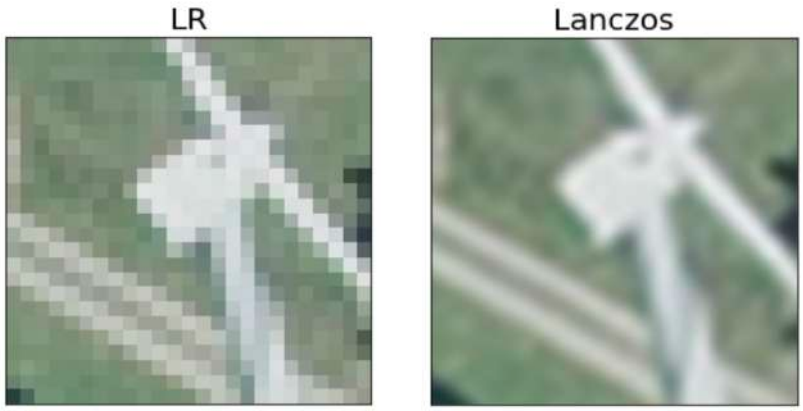

Fig. 5.1.a-IX Windmills image: LR vs. Lanczos.

The images generated by learning models are extraordinarily good. The objects silhouette are defined, totally distinguishable with the background, edges are sharped and marked. The SRGAN generates more grainy images than the smooth and clear SRFBN images, but both results have strongly improved. Some variations in brightness and contrast are present in the generated images.

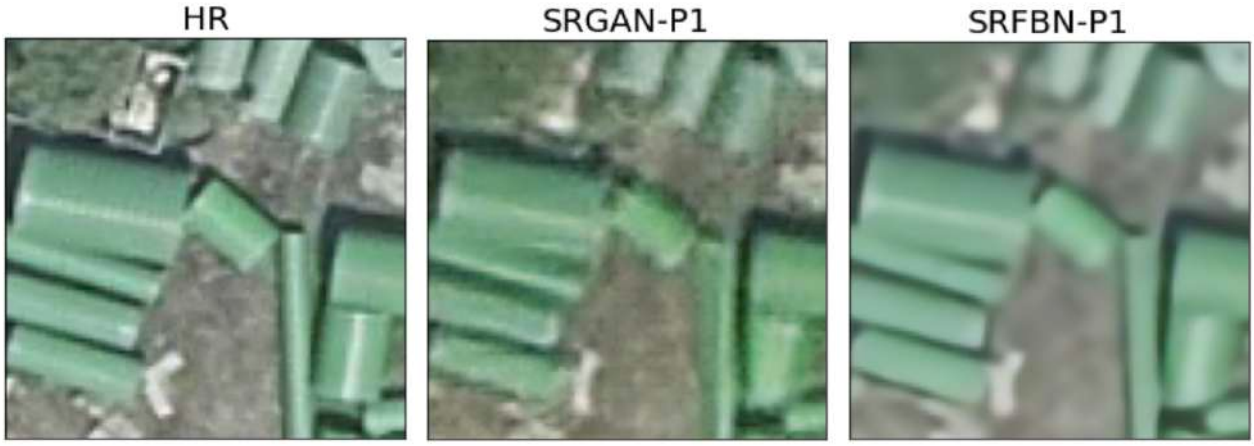

Fig. 5.1.a-X Amazing generated SR images by learning upsampling models.

- ***COASTAL AREAS***

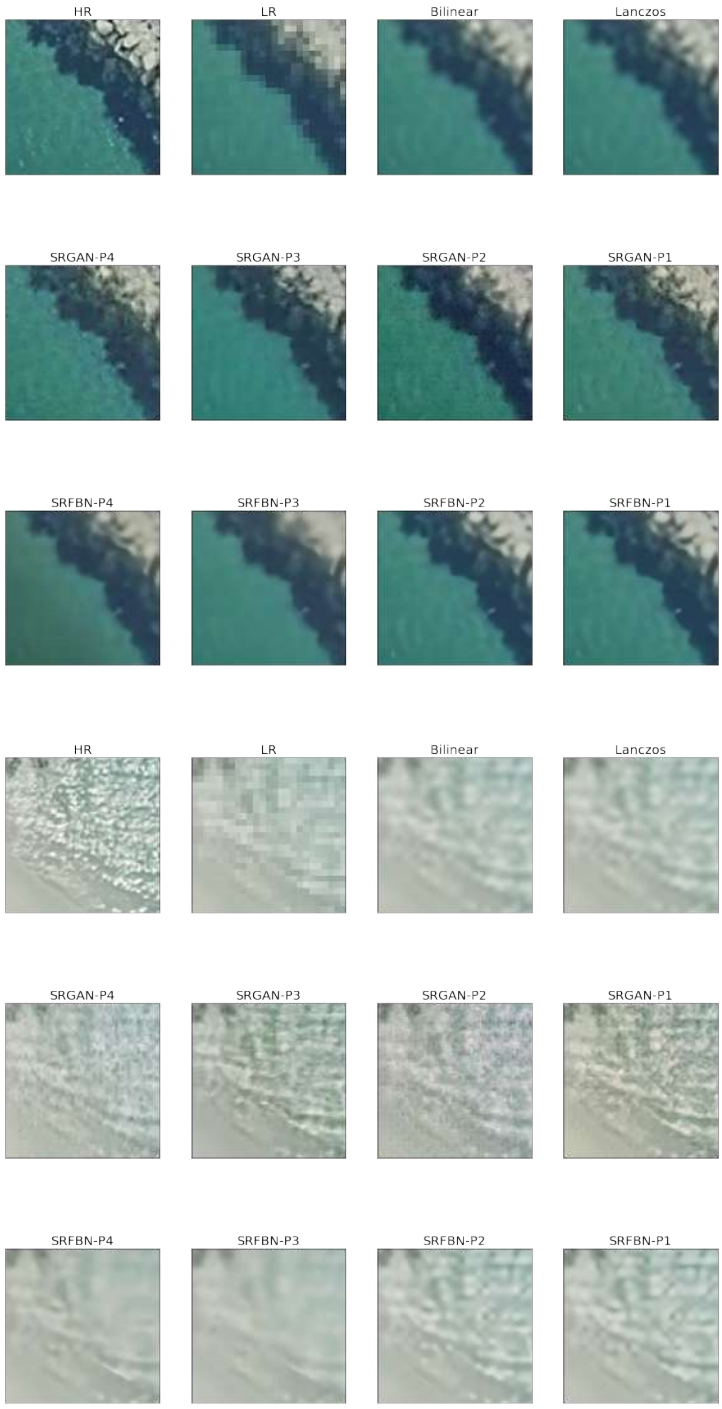

Fig. 5.1.a-XI PNOA MA - SR images of coastal areas.

We can see how water surfaces, in this case, coastal images, shown in [Fig. 5.1.a-XI] have improved, but these are not the best results. Water is not a homogenous surface, it is constantly moving and changing. In the figure above, we can see two different situations, the first image shows deep and calm water from a harbor; the second image is from a seashore with clear and shallow water.

- ***URBAN AREAS***

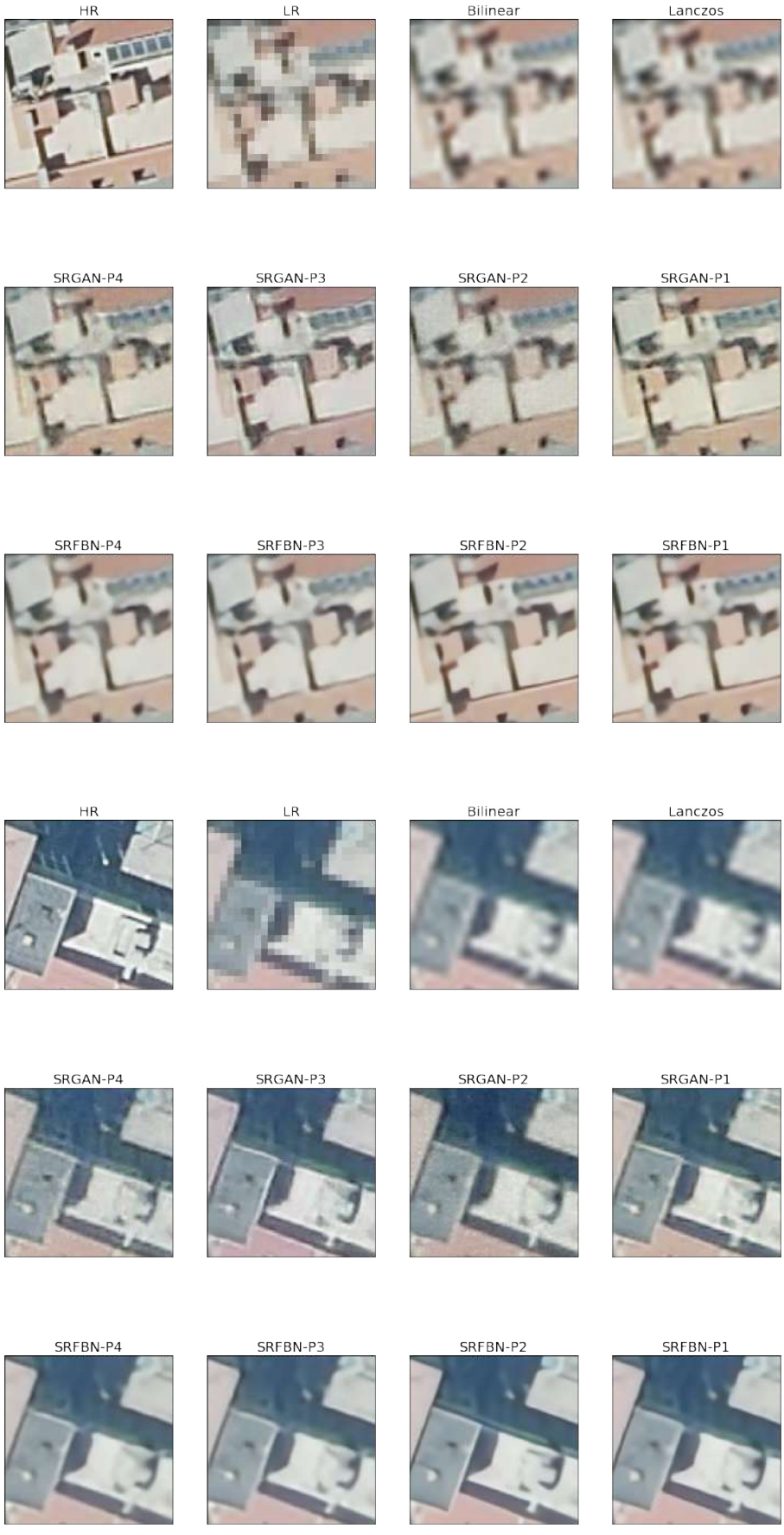

Fig. 5.1.a-XII PNOA MA - SR images of coastal areas.

- ***ROADS***

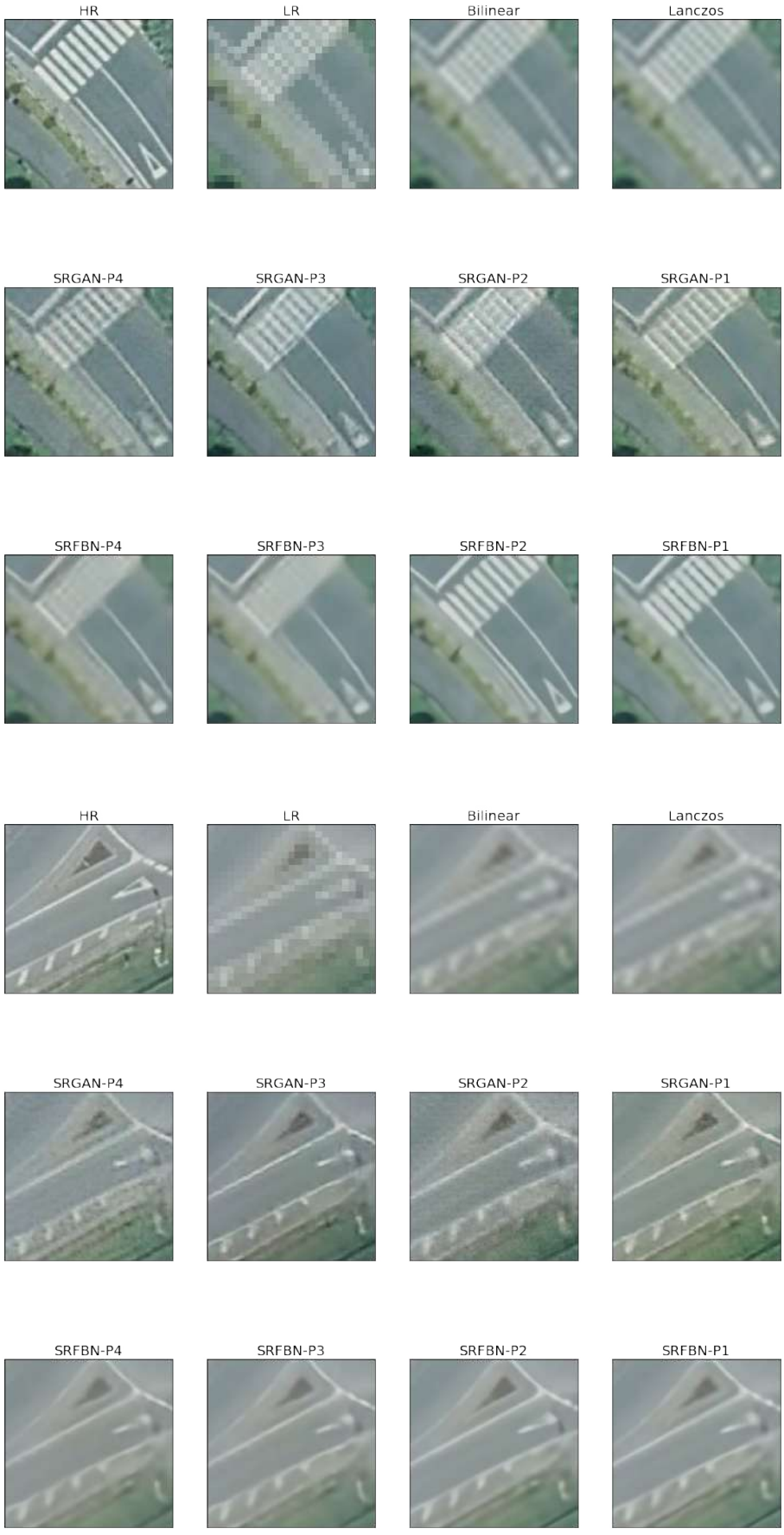

Fig. 5.1.a-XIII PNOA MA - SR images of roads.

As we saw in the charts from [Fig. 5.1.a-I] ,[Fig. 5.1.a-II], and [Fig. 5.1.a-III], metrics values are the highest for urban and roads images so as we could imagine, these results will be the best-generated images with the biggest improvement.

Urban images represent cities, villages, and urbanized areas full of buildings and squares, i.e., geometrical objects with straight edges and static bodies. These kinds of items are the easiest to recognize by deep learning algorithms. SR images from learning methods are better than interpolation ones, but SRGAN images suffer brightness variations, and SRFBN results have a smoothness effect although it generates the best results, as we can see in [Fig. 5.1.a-XII]:

Fig. 5.1.a-XIV Upsampled urban image by SRFBN vs. pixelated building.

Roads images have the highest metrics scores and generated images reflect that. Roads signs are very clear and as we can see in [Fig. 5.1.a-XIII], crosswalks is an ideal reference point to compare results and measure the image resolution. SRGAN SR images are affected by color differences and SRFBN is the best model, although the experiments generate very different results with widely varying qualities. The generated images from SRFBN architectures with 64 filters have better visual results than 32 filters models; in this last, stripes are blurred and it is not possible to distinguish them instead of 64 filters model where stripes are clearly separate by gaps:

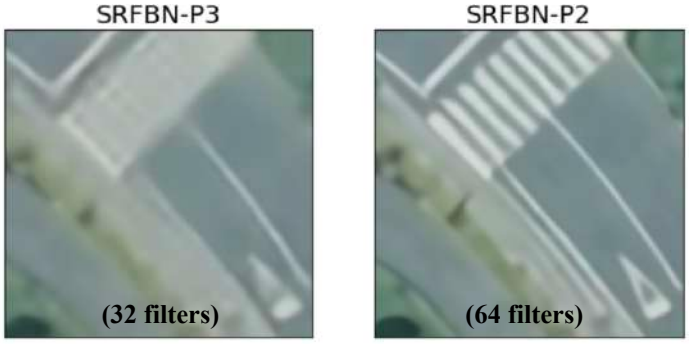

Fig. 5.1.a-XV The big difference between roads signs generated by SRFBN models with 64 and 32 filters.

5.1.b PNOA 10

- ***INTERPOLATION***

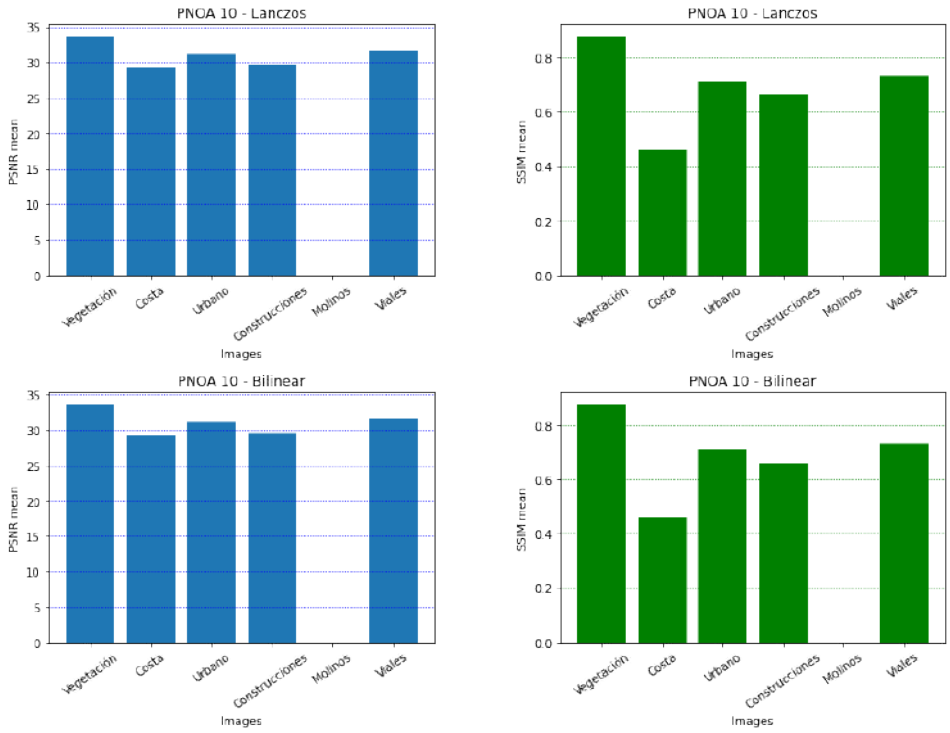

Fig. 5.1.b-I PNOA 10 - Interpolation metrics.

- ***SRGAN***

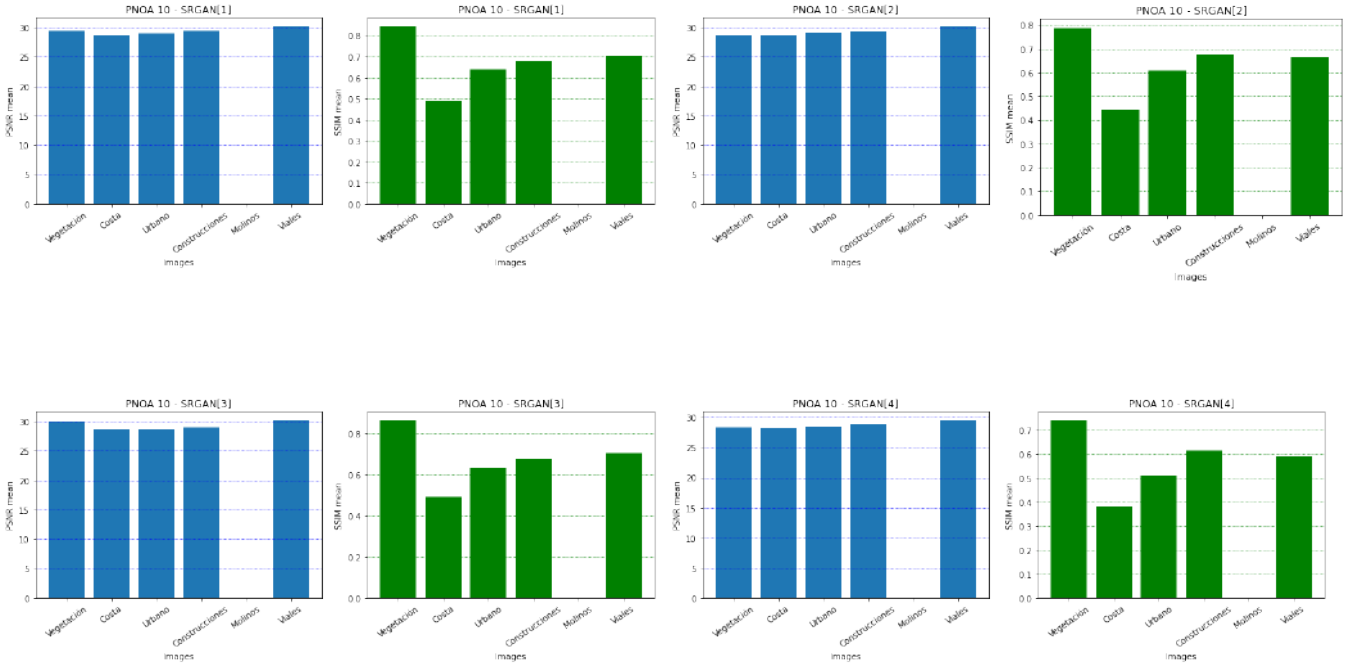

Fig. 5.1.b-II PNOA 10 - SRGAN metrics.

- ***SRFBN***

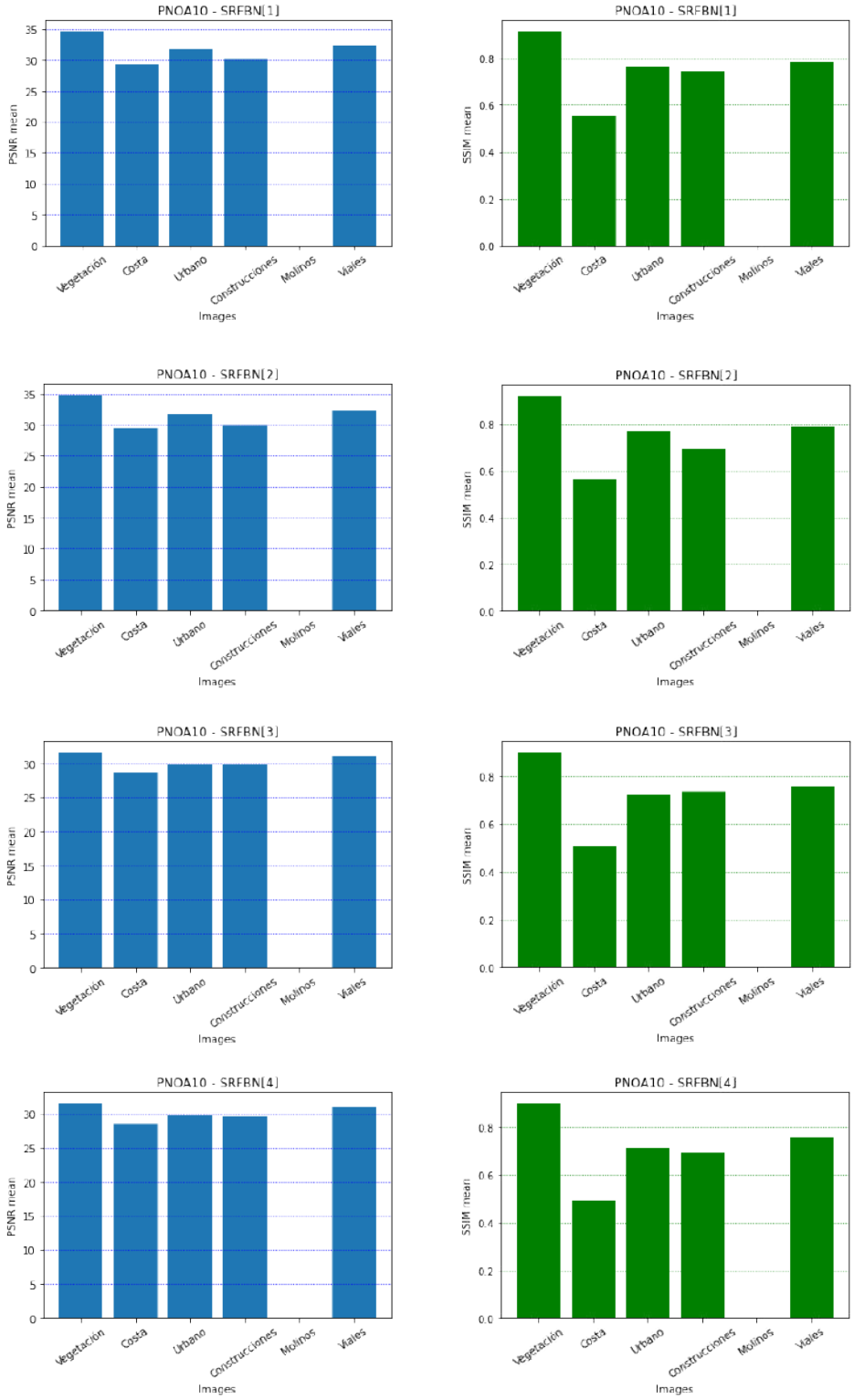

Fig. 5.1.b-III PNOA 10 - SRFBN metrics.

This dataset does not have any image of windmills. This time we see a correlation between PSNR and SSIM in [Fig. 5.1.b-I], [Fig. 5.1.b-II], and [Fig. 5.1.b-III]. The images of vegetation have the highest score, and coastal images have the lowest. In SSIM values, we can see more abrupt differences.

- ***VEGETATION***

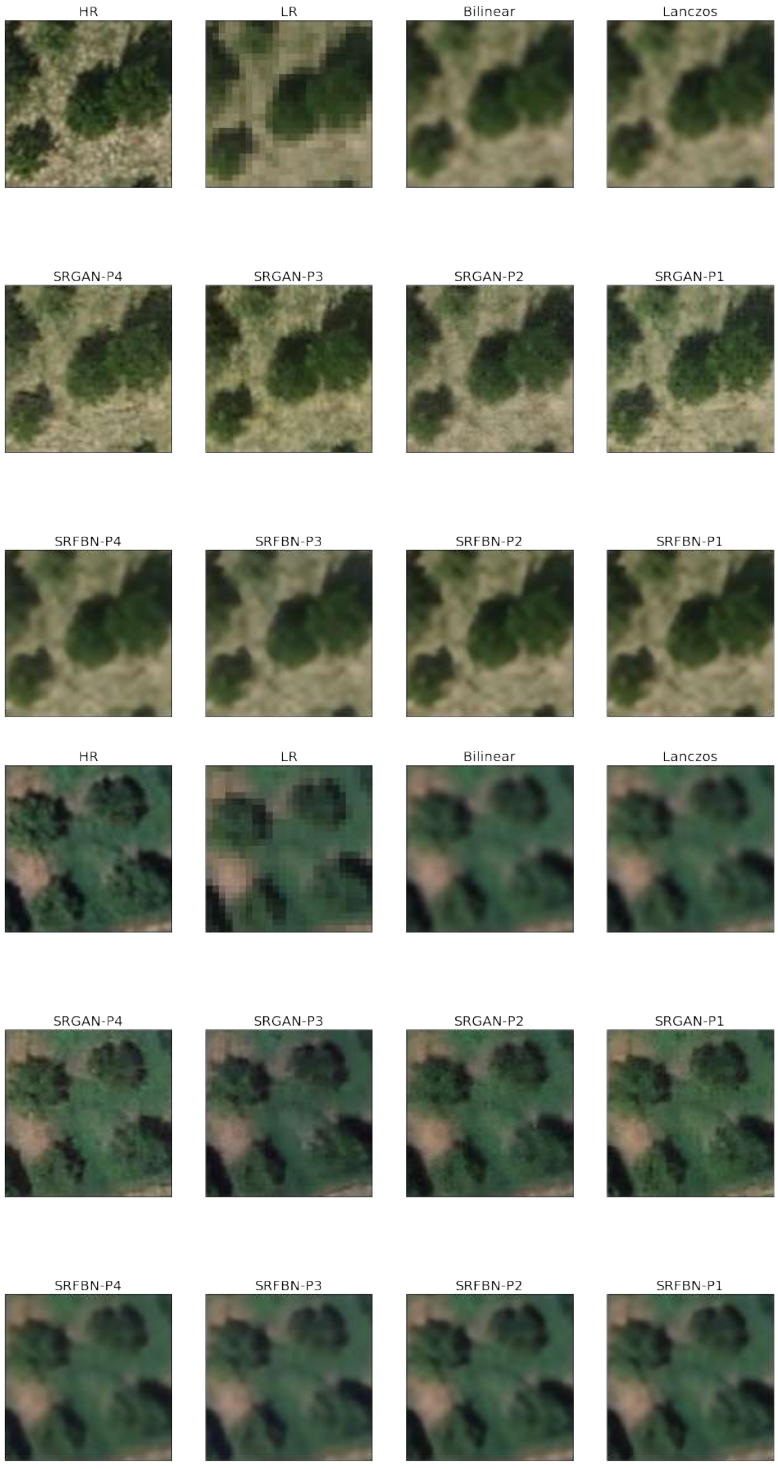

Fig. 5.1.b-IV PNOA 10 - SR images of vegetation.

In [Fig. 5.1.b-IV] are plotted two examples of SR-generated vegetation images from the PNOA 10 dataset. These results are similar to SR from PNOA MA [Fig. 5.1.a-IV], so its comments can be extrapolated to this case too. Interpolation results are the worst, despite its high metrics values, PSNR above thirty, and SSIM between the 60% and 85% of similarity with HR image.

- ***HUMAN CONSTRUCTIONS***

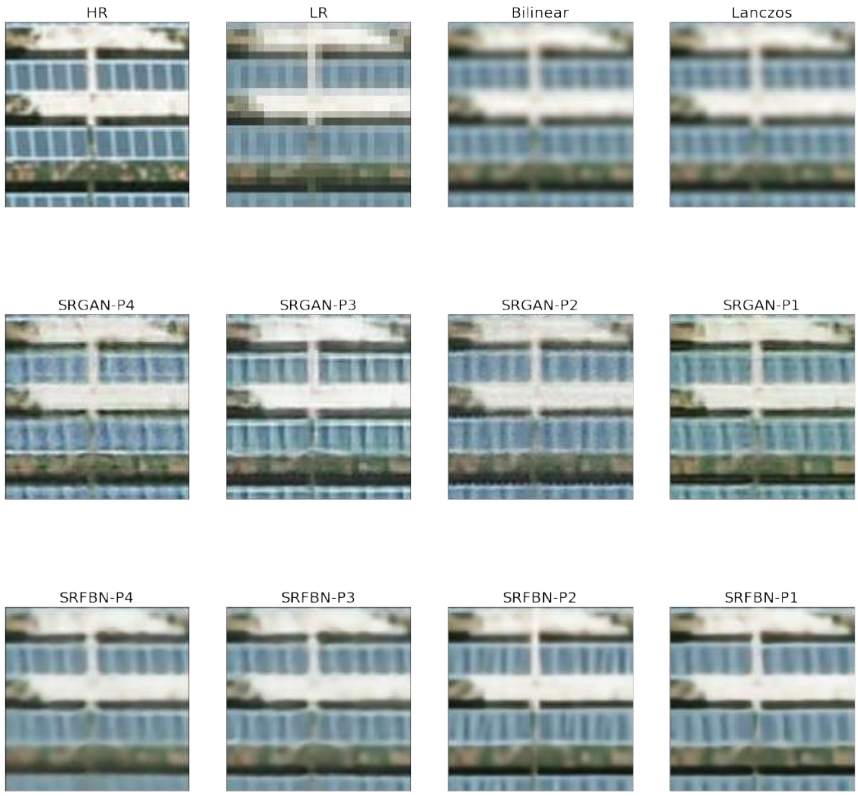

Fig. 5.1.b-V PNOA10 - SR images of human constructions (Solar panels).

- ***COASTAL AREAS***

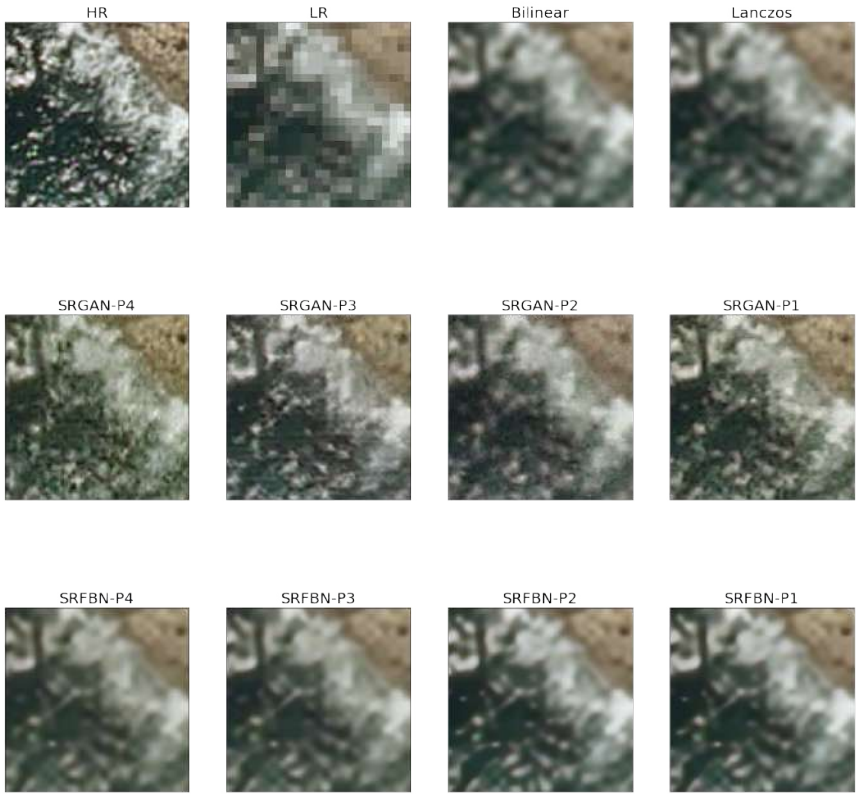

Fig. 5.1.b-VI PNOA 10 - SR images of coastal areas.

The SR images with constructions and from coastal areas are the lowest metric values. These results are illustrated in [Fig. 5.1.b-V] and [Fig. 5.1.b-VI], where we can see how most of the results are blurred and fuzzy, details are not correctly represented.

For example, SR images with solar panels have a white line that limits each panel. This line is sharped in HR images, whereas in generated images this separation is diffuse. The best metrics results match our personal visual perception, and the best score is obtained by SRFBN. The second place is not clear because metric values and visual quality do not match, this is because SRGAN has brightness and color variations that directly affect metrics values. Interpolation obtained the most blurred images. The results have improved since the LR image, but they are still scoped to improve:

Fig. 5.1.b-VII Improved but still room for improvement, SR-generated images from solar panels.

Coastal images obtained the worst metric values, less than thirty as PSNR score, and around 50% of SSIM. As we have explained in coastal images of the PNOA MA, this is not motionless, it is constantly moving and changing so the initial quality is not the desired one. However, the object-background distinction is very clear, we can visually separate the water surface of the ground.

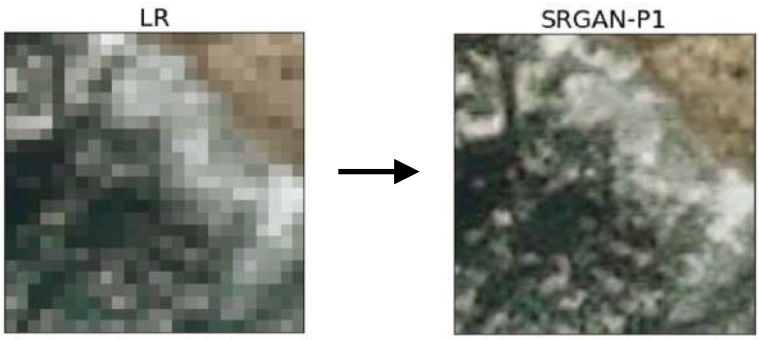

Fig. 5.1.b-VIII From pixelated LR image to a SR-generated coastal image.

- ***URBAN AREAS***

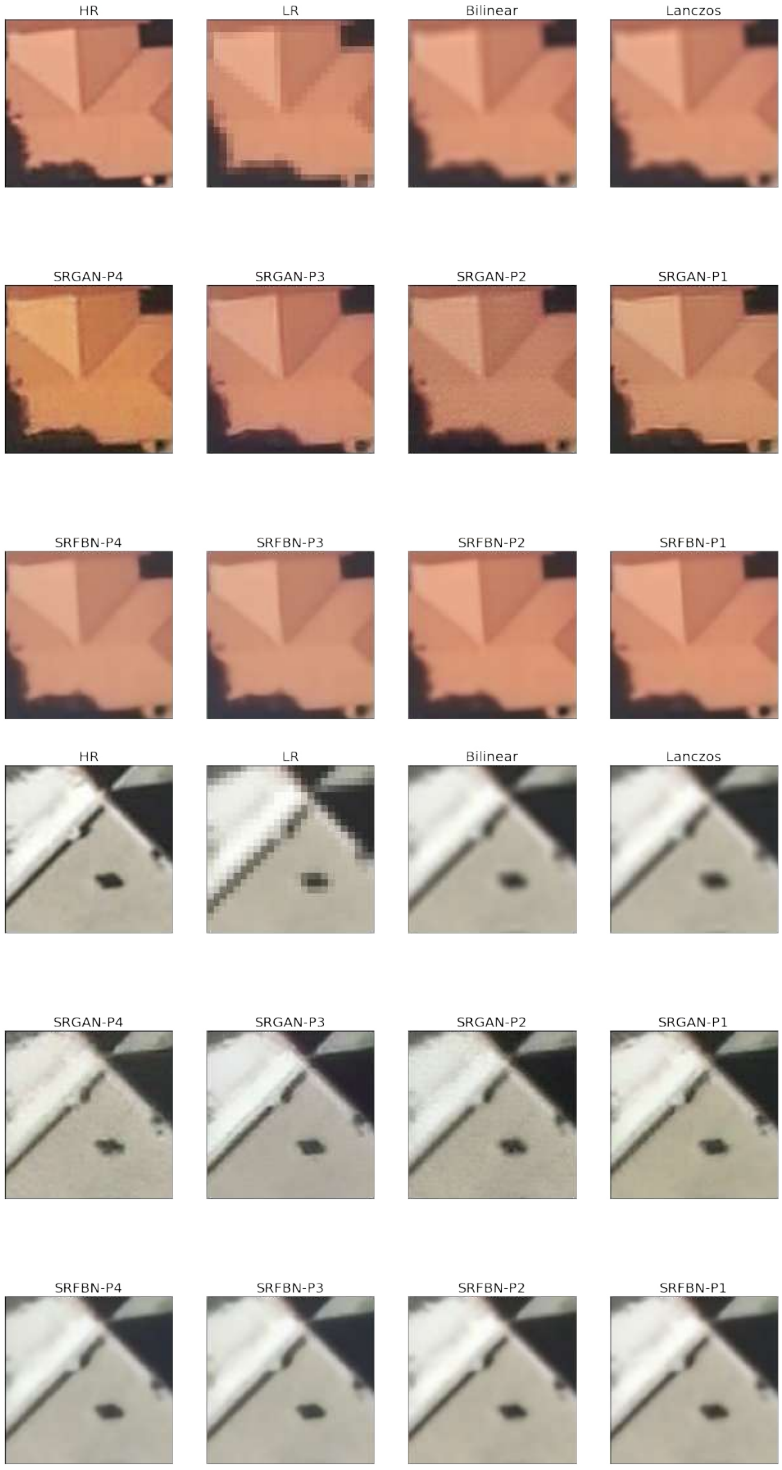

Fig. 5.1.b-IX PNOA 10 - SR images of coastal areas (buildings).

The SR images from urban areas obtain high scores and, we perceive in them a big improvement, as we can see in the images above [Fig. 5.1.b-IX]. The straight elements represented in this category help to the generated images by learning networks, where the pixelated edges and corners become to clear-cut elements. This category have good results.

- ***ROADS***

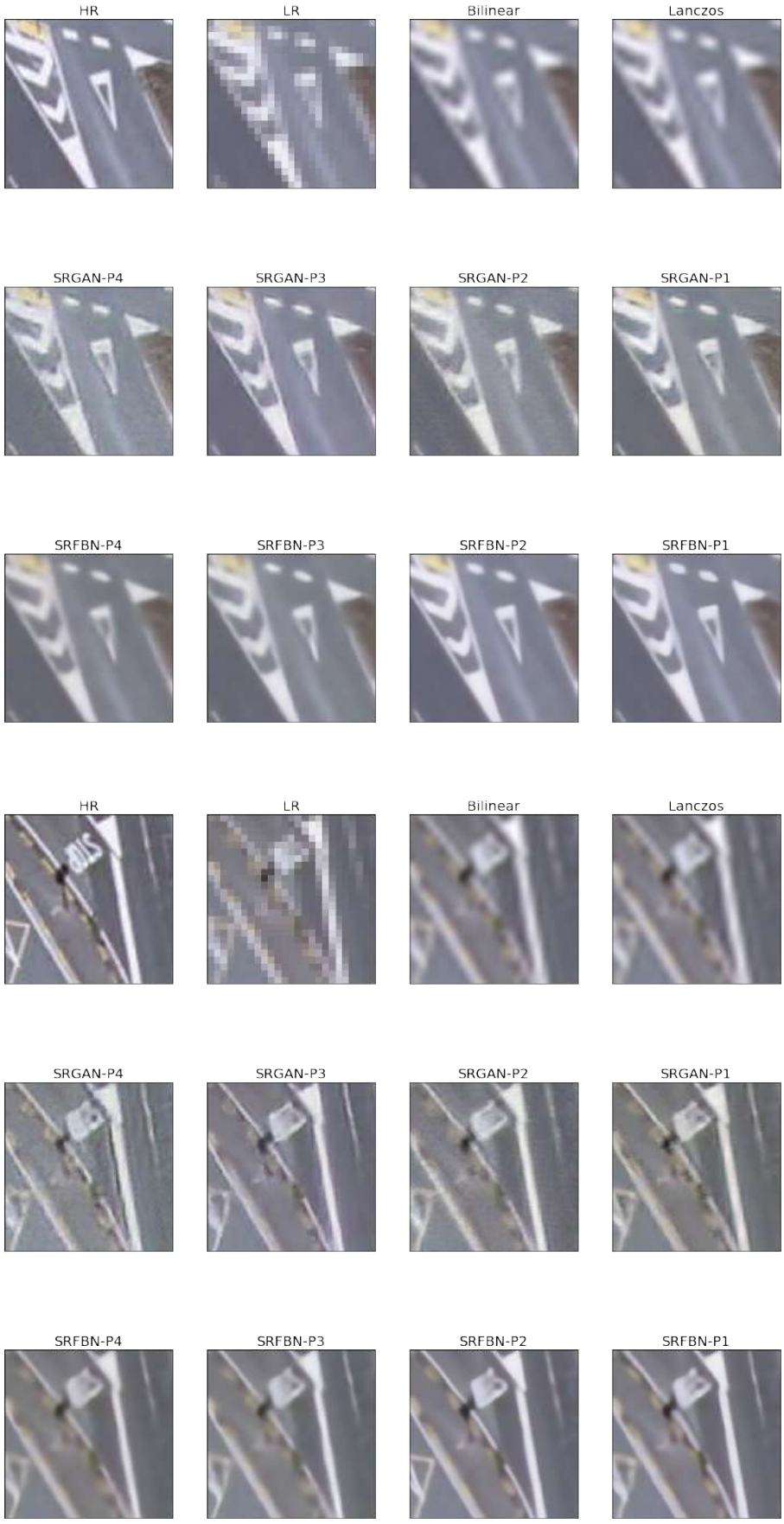

Fig. 5.1.b-X PNOA 10 - SR images of roads.

The generated road images are illustrated in [Fig. 5.1.b-X] and these are good examples of the huge capability of upsampling methods based on learning. The road signs have strongly improved becoming sharper and more discernible. The written worldly signs as “STOP” or “BUS”, still being fuzzy, and these are not possible to read but there are about interpretable thanks to their background. For example, if there is a continuous white line after the written sign, we should suppose this is a STOP. To improve the interpretability skills of networks, we should adapt our training dataset, and train the models with roads signs. In this case, our networks would be able to find this kind of patrons, resulting in easier and understandable written signs. The generated traffic lines have an amazing quality:

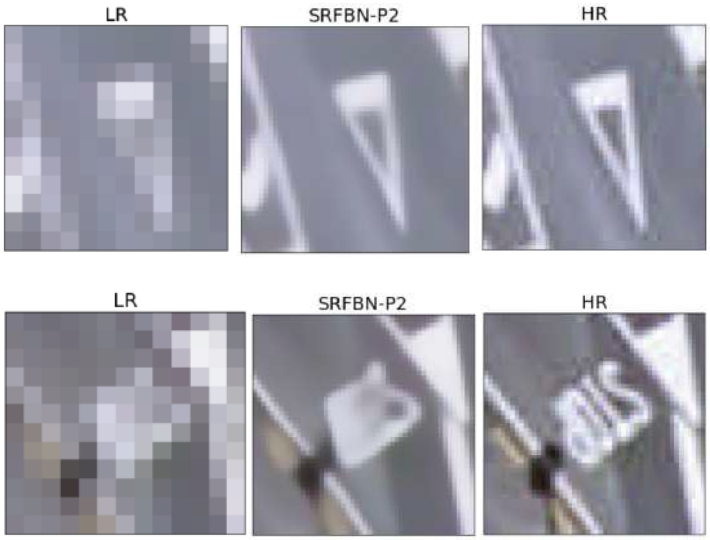

Fig. 5.1.b-XI The evolution of traffic signs after applying upsampling methods.

In this dataset, there are two images with shadows areas, to see how is the upsampling when pixels values suffer abrupt changes. As we can see in the figure below. In HR images, these elements are not well represented. Nevertheless, SRFBN achieves to generate it subtly, contrary to interpolations methods:

Fig. 5.1.b-XII The presence of shadows and its effect on SR-generated images by upsampling.

5.1.c PLEIADES

- ***INTERPOLATION***

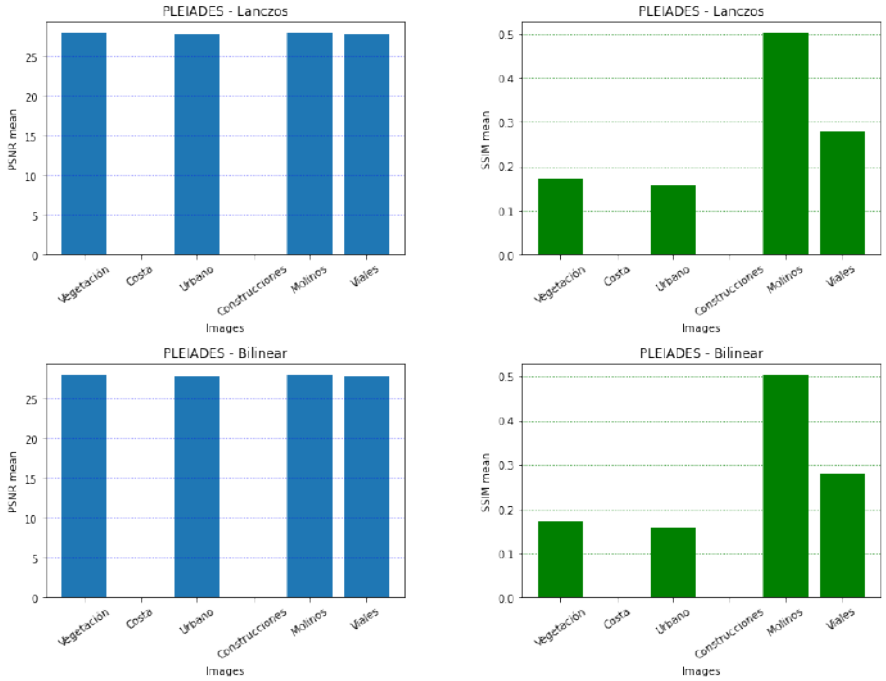

Fig. 5.1.c-I Pleiades Satellite - Interpolation metrics.

- ***SRGAN***

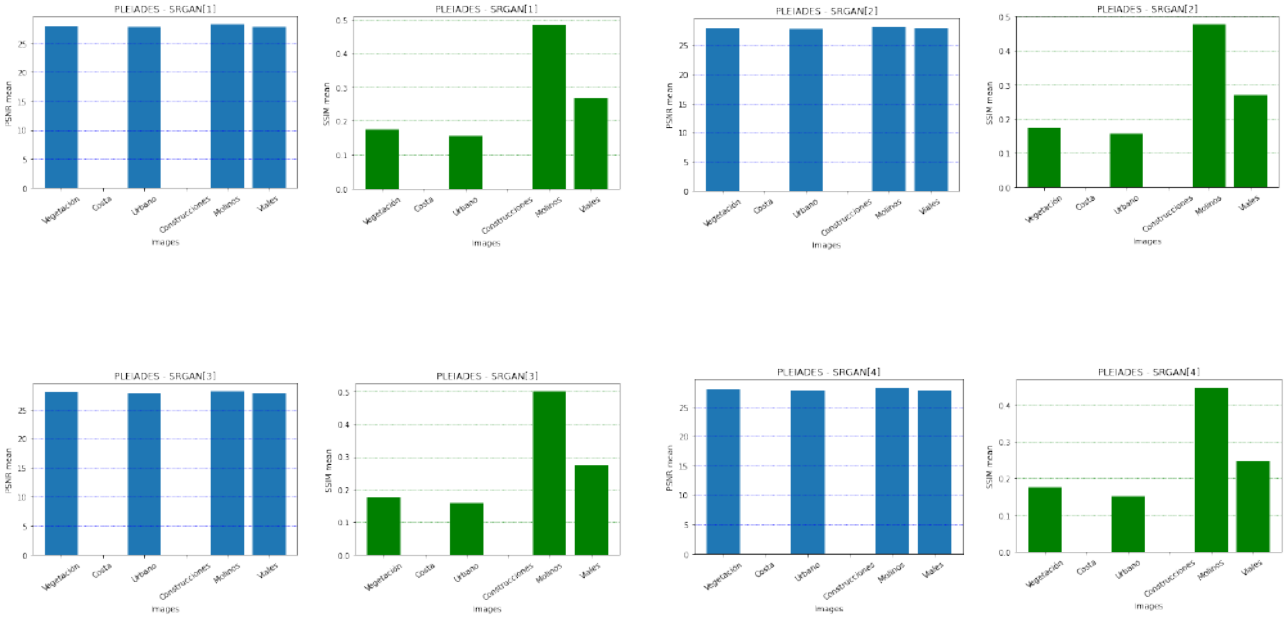

Fig. 5.1.c-II Pleiades Satellite - SRGAN metrics.

- ***SRFBN***

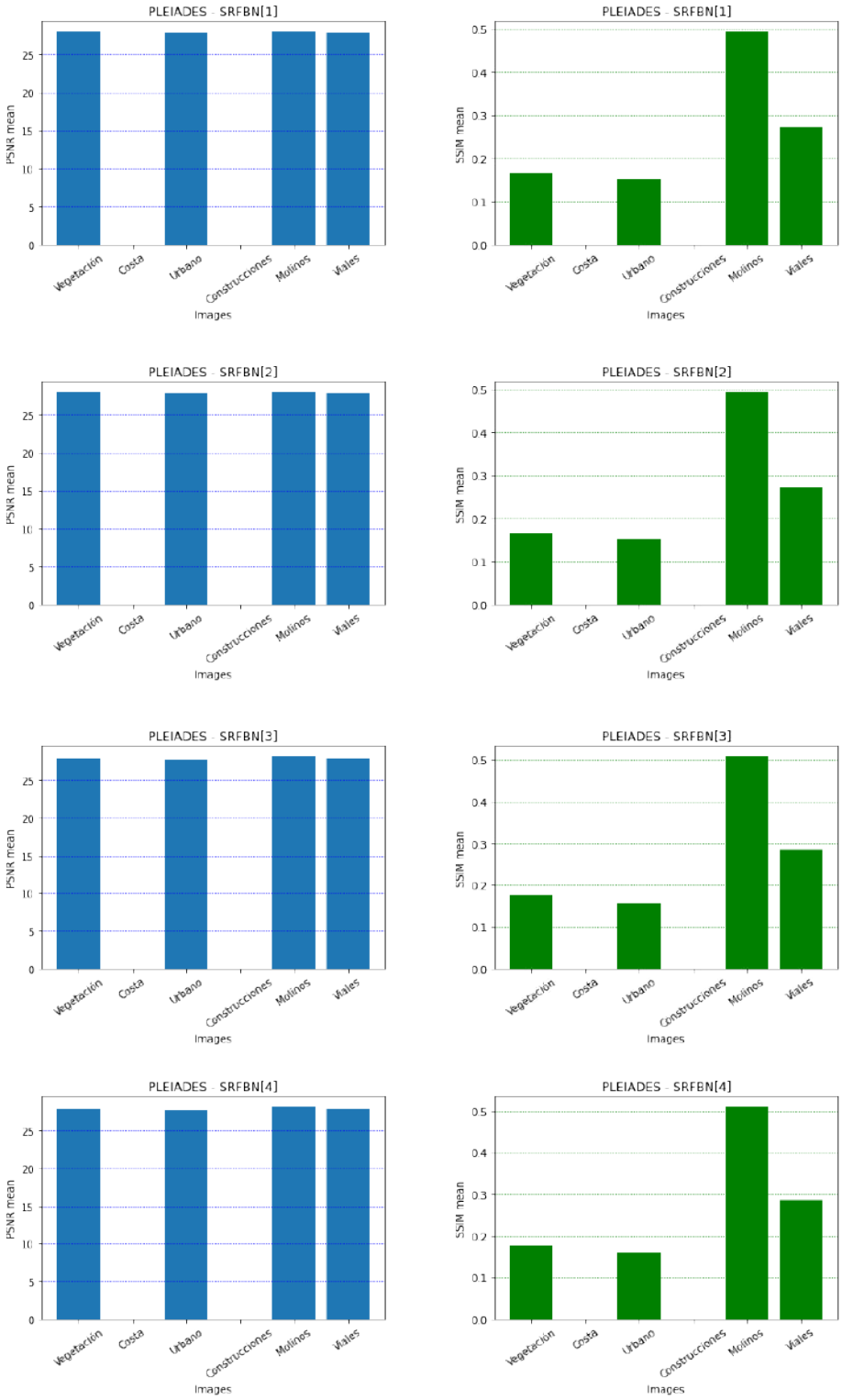

Fig. 5.1.c-III Pleiades Satellite - SRFBN metrics.

This dataset only has images of vegetation, urban areas, windmills, and roads. The images of windmills have the highest score, and urban areas have the lowest. PSNR scores are so similar, in SSIM values exist big differences between image scores [Fig. 5.1.c-I], [Fig. 5.1.c-II], and [Fig. 5.1.c-III].

- ***VEGETATION***

Fig. 5.1.c-IV Pleiades Satellite - SR images of vegetation.

- ***URBAN AREAS***

Fig. 5.1.c-V Pleiades Satellite - SR images of urban area.

As we have mentioned in [Section 3.1.3], the Pleiades satellite dataset is the only one where its LR images are the same as captured by the sensor, i.e., the LR images are not the result of applying downscaling on HR images. While for PNOA MA and PNOA 10, we are working with LR images obtained by applying downscaling on HR images, LR images with resolutions of less than a meter, of a few cm. Satellite images have two meters of resolution and we want to reduce it to half of a meter. For this, the generated images are not as detailed as SR-generated images from the other two datasets, but we are not looking for the same quality nor working with the same resolution, so our results are how we expected, although these are able to improve.

Relative to vegetation and urban areas, the generated images from these categories do not look good. The SR images do not improve as much from LR. The objects and surfaces represented in these categories are very similar to their background, and details are not defined as in the urban images from other datasets. Results are not pixelated but there look like a painting with “brush strokes”, and objects are not sharp and clear, it is difficult to identify them. Even so, generated images by our networks have better results than interpolation, and there are less pixelated than LR:

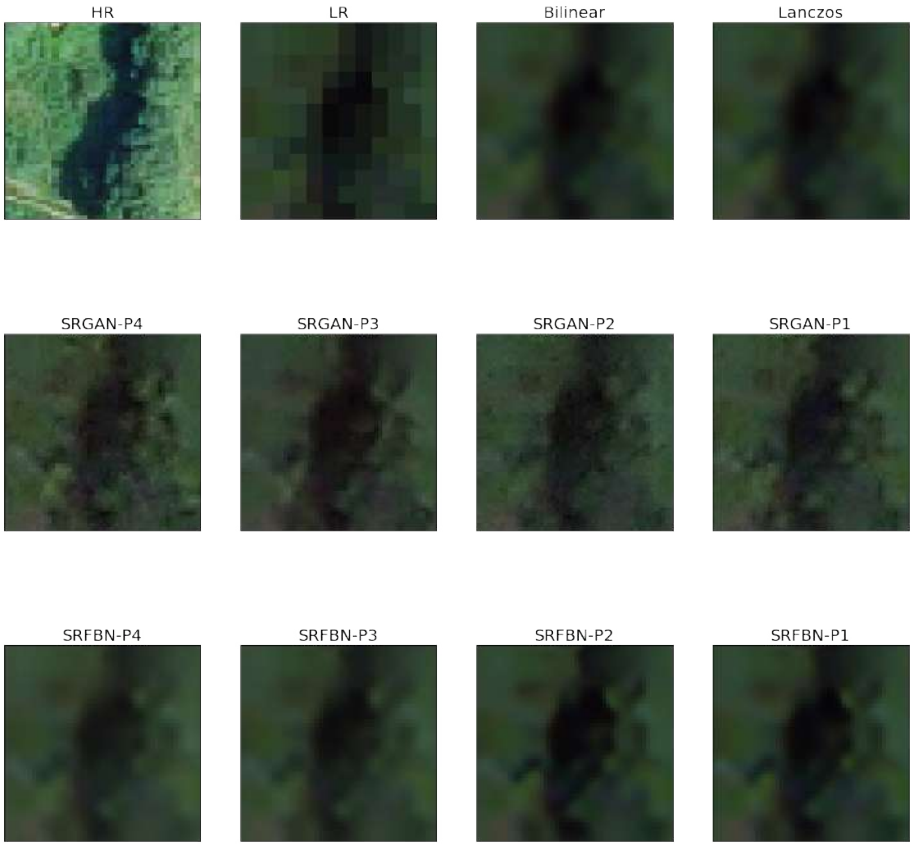

Fig. 5.1.c-VI A vegetation image from the Pleiades and its improvement over its pixelated origin.

- ***WINDMILLS***

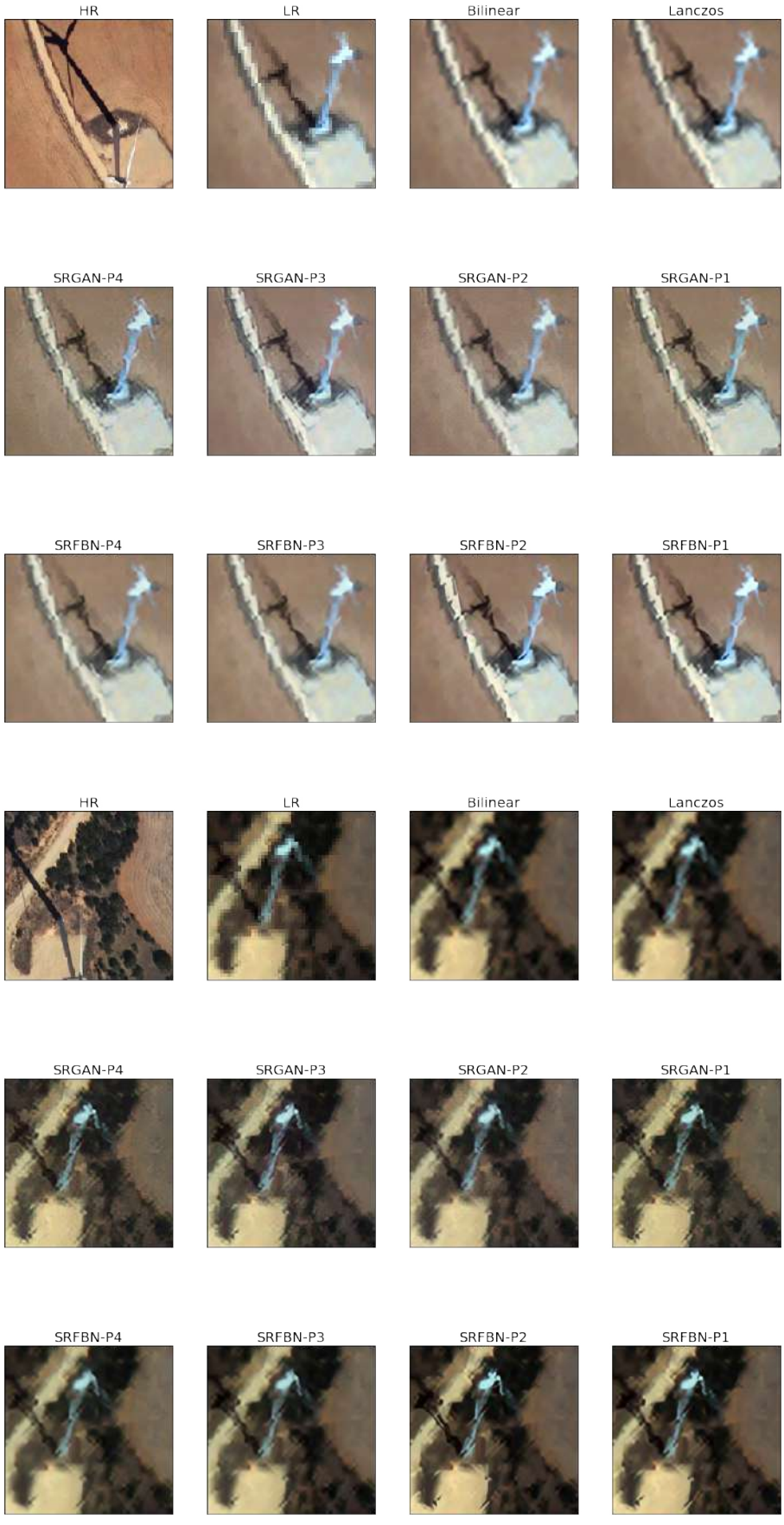

Fig. 5.1.c-VII Pleiades Satellite - SR images of windmills.

The [Fig. 5.1.c-VII] illustrates two different images from the Pleiades with windmills after applying our three upsampling methods. In this category, we see the highest improvement. The LR images are pixelated and this characteristic complicates the upsampling process. Interpolations have the worst results, they are blurred but they keep being better than LR images. The real improvement is achieved with the upsampling methods by learning, i.e., with SRGAN and SRFBN:

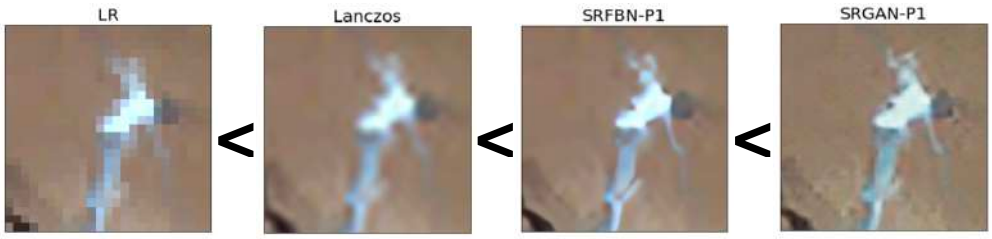

Fig. 5.1.c-VIII An example of upsampled mill captured by Pleiades.

The highest scores for this category is obtained with SRGAN, the object is sharper than in SRFBN where it is smoother. The improvement is clear, how from a totally pixelated image we can distinguish the mill silhouette. In SRGAN images continue generating images when color and brightness variations, in the [Fig. 5.1.c-VIII] this variation is easy to identify in the background color.

Also, the metrics obtained with this dataset are not correct due to the HR images used and their origin, as we previously mentioned. A clear example of this in [Fig. 5.1.c-IX] where the mill position in the satellite images is not the same in HR images from PNOA:

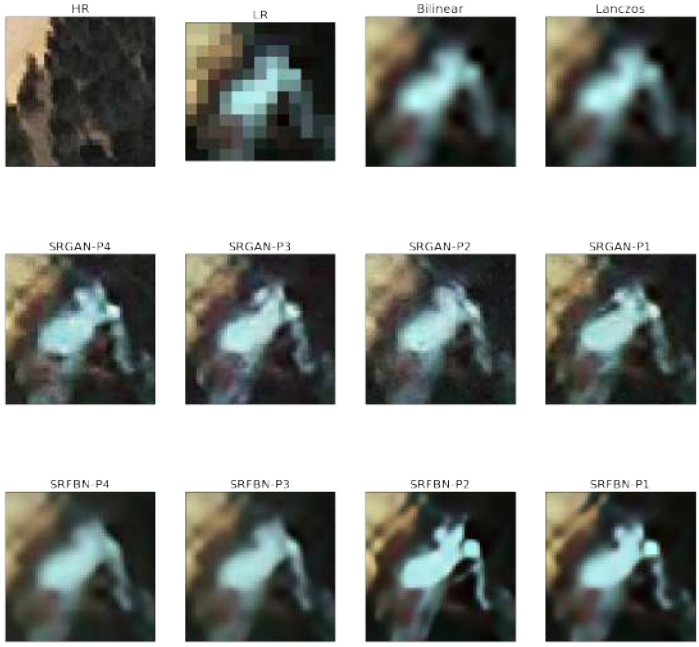

Fig. 5.1.c-IX The big differences between HR and LR, the principal cause of low metrics values.

- ***ROADS***

Fig. 5.1.c-X Pleiades Satellite - SR images of roads.

For this category, the results are so similar to [Fig. 5.1.c-V]. Roads are identifiable and clear items, but generated images have the already mentioned "drawing effect" (*pg. 103*). The generated images are not pixelated although, the objects of interest are smooth with irregular edges. In this case, traffic and road signs are impossible to distinguish.

5.1.d Discussion

After analyzing the results and charts, we have seen how PSNR and SSIM are related, as we have explained in [Section 2.2.2]; high PSNR values mean high SSIM. In general, the variance of PSNR scores is lower than SSIM variance, as we can see graphically in the charts as it is illustrated in [Fig. 5.1.a-I], [Fig. 5.1.a-II], and [Fig. 5.1.a-III]; where we can see how PSNR values are more similar for all the categories, there are no abrupt changes in their scores, these are around thirty, and it is the same for the PNOA 10 dataset. The average of metrics values for the satellite dataset is upper twenty-five and lower than thirty, but PSNR scores keep varying close to the average value, not as SSIM. SSIM values have more extremely variations as we can see in [Fig. 5.1.c-I], [Fig. 5.1.c-II], or in [Fig. 5.1.c-III], the minimum SSIM value is around the 10%, and the highest one surpasses the 50% of similarity.

As a summary, the [Table 5.1.d] shows the metrics values of SR-generated images:

Table 5.1.d: Summary of score metrics obtained by each model.

Dataset	Model	PSNR				SSIM			
		Max.	**Category**	**Min.**	**Category**	**Max.**	**Category**	**Min.**	**Category**
PNOA MA	Bilinear	33,6850	Roads	29,4542	Coast	0,7954	Roads	0,4374	Veg.
	Lanczos	33,7080	Roads	29,4611	Coast	0,7971	Roads	0,4390	Veg.
	SRGAN-P4	30,2227	Roads	28,5192	Veg.	0,5946	Roads	0,3068	Veg.
	SRGAN-P3	31,2423	Coast	28,0531	Const.	0,8124	Roads	0,4496	Veg.
	SRGAN-P2	30,4559	Const.	28,3616	Coast	0,6785	Roads	0,3969	Veg.
	SRGAN-P1	31,1710	Coast	28,8939	Coast	0,7908	Roads	0,4566	Veg.
	SRFBN-P4	33,4026	Roads	29,1700	Veg.	0,8302	Roads	0,4509	Coast
	SRFBN-P3	33,5766	Roads	29,3002	Veg.	0,8330	Roads	0,4288	Coast
	SRFBN-P2	**34,4728**	**Roads**	**29,6583**	**Coast**	**0,8541**	**Roads**	**0,5120**	**Veg.**
	SRFBN-P1	34,3638	Roads	29,6429	Coast	0,8470	Roads	0,5067	Veg.
	Bilinear	34,6529	Urban	29,1521	Coast	0,9009	Urban	0,4604	Coast
	Lanczos	34,6794	Urban	29,1512	Coast	0,9030	Urban	0,4629	Coast

Dataset	Model	PSNR				SSIM			
		Max.	Category	Min.	Category	Max.	Category	Min.	Category
PNOA 10	SRGAN-P4	30,2327	Veg.	28,0351	Urban	0,7446	Urban	0,3838	Coast
	SRGAN-P3	30,9761	Roads	28,3509	Veg.	0,8884	Urban	0,4902	Coast
	SRGAN-P2	31,2808	Veg.	28,5278	Urban	0,7996	Urban	0,4425	Coast
	SRGAN-P1	31,6464	Veg.	28,5330	Veg.	0,8608	Urban	0,4879	Coast
	SRFBN-P4	32,0739	Veg.	28,5036	Coast	0,9209	Urban	0,4911	Coast
	SRFBN-P3	32,1382	Urban	28,6733	Coast	0,9224	Urban	0,5083	Coast
	SRFBN-P2	**36,0783**	**Urban**	**29,3920**	**Coast**	**0,9399**	**Urban**	**0,5639**	**Coast**
	SRFBN-P1	35,8884	Urban	29,3799	Coast	0,9352	Urban	0,5521	Coast
PLEIADES	Bilinear	28,6948	Mills	27,6097	Mills	0,6061	Mills	0,1137	Urban
	Lanczos	28,6947	Mills	27,6084	Mills	0,6058	Mills	0,1135	Urban
	SRGAN-P4	28,5784	Mills	27,6903	Roads	0,5035	Mills	0,1087	Urban
	SRGAN-P3	28,4608	Mills	27,6436	Veg.	0,5928	Mills	0,1134	Urban
	SRGAN-P2	28,5941	Mills	27,6738	Veg.	0,5484	Mills	0,1131	Urban
	SRGAN-P1	**28,8869**	**Mills**	**27,6652**	**Veg.**	**0,5741**	**Mills**	**0,1121**	**Urban**
	SRFBN-P4	28,7085	Mills	27,6353	Veg.	0,6051	Mills	0,1159	Urban
	SRFBN-P3	28,8380	Mills	27,6379	Veg.	0,6057	Mills	0,1143	Urban
	SRFBN-P2	28,6892	Mills	27,5994	Mills	0,5970	Mills	0,1109	Urban
	SRFBN-P1	28,6877	Mills	27,5976	Mills	0,5985	Mills	0,1113	Urban

For each dataset and each category, these metric values are different, for example in the Pleiades dataset, vegetation and urban areas images have the worst scores, instead of PNOA 10 where these categories have the best scores. But we see a general patron, images with roads have good test results for every test dataset. This could be strongly related to our training dataset. As we have mentioned above, the dataset has a direct influence on our results, if our dataset is mostly composed of images with roads, the generated roads images will have better results than others like coastal or vegetation images. So we can deduce that roads are present in a big number of images in our training dataset.

As for SR-generated images, we can see how results from PNOA MA and PNOA10 are considerably better than Pleiades satellite SR images. Also, we have to consider that the original resolution from these datasets is different, being the Pleiades dataset, the one with the lowest resolution. Furthermore, these images from satellites are more affected by the atmosphere, and these could suffer distortions and other accuracy issues to consider. As we have mentioned on numerous occasions, for this testing dataset of Pleiades images, quantitative metrics are not a rigorous value. For that, we are going to evaluate the influence of our trained models on these test images based on graphic results, i.e., the generated SR images. Below is a general analysis per each category:

- ***Vegetation:*** it is well represented in SR images and it is easy to identify. Visually, the best results are from the **PNOA MA dataset**, and even interpolation methods obtained better PSNR and SSIM results, the most realistic SR images are generated by **SRGAN**. The vegetation in SR images from interpolation looks very blurred and with difficulties to identify and distinguish objects from the background. The situation of SR images from SRFBN is similar to this, but results are better than interpolation.

- ***Urban areas and Human constructions:*** these images are plenty of regular and clear objects, as roofs with straight lines, good defined squares and edges, etc. Again, interpolation results should have a similar quality to SRFBN results, but there are completely worst. **Best results are generated by trained models** from both networks. Having a little better results with **SRFBN**, because we can appreciate some noise in SR images from SRGAN, as a granulated effect.

- ***Coastal images:*** these kinds of surfaces are not usually well represented in aerial images, because water is not motionless and it is very reflecting. But for this category, interpolation generates blurred images, and **both networks have better visual results**. **SRGAN** results are sharper.

- ***Windmills:*** we have decided to make a single category for these objects because these are easy to identify. Also, these SR images could be useful for future applications related to windmills, wind energy, etc. One more time, generated images from Pleiades dataset have a lower quality than the other one. But in this dataset too, generated images by networks are better than interpolation blurred results. In the **PNOA MA dataset**, we can

appreciate a huge improvement and a big difference between interpolation results and generated images from training by learning. SR images from both networks have high quality and sharp objects, it is easy to identify edges and small details. In the SRGAN results, we see some noise and generated images are a little bit darker than the original one, obtaining **SRFBN** the best results.

- ***Roads:*** this category is very useful for current daily applications and future projects. Roads are a lot of different surfaces or areas, so it is understandable the big presence of these elements (roads, paths, etc.) in our training and validation datasets. These elements also have a similar appearance; they are made by the same or similar materials, traffic signals are standardized, etc. These SR images have good results: in pleiades dataset, the quality of generated images is not as good as in the other two datasets, but these elements are easy to identify and distinguish with the background; in PNOA MA and PNOA 10, SR images are very good, being one more time, the images generated by networks better than upsampling images by interpolation. Traffic lines are very clear in both datasets, and some traffic signs as written words, like “STOP” or “BUS”, are blurred and they are not legible. SR images from PNOA 10 are better than PNOA MA, but this was expected because their initial resolutions are different, being the **PNOA 10** resolution higher than PNOA MA [Section 3.1.3]. Again, SRGAN images are noisy and darker than sharp, clear, and smooth images generated by **SRFBN**.

5.2. Demo

The demo has been developed with libraries compatible with different software. Its purpose is to plot two images to their comparison. It is thought to compare SR-generated images from different models and architectures; we can scroll, zoom in, and zoom out of both images at the same time, i.e., like a mirror, seeing the same cells or pixels of each image.

5.2.a Matlab

Code snippet 5.2.a: DEMO - Matlab

demo.m

```
% DEMO

img1 = imread('/Users/monica/Documents/TFG/IMAGENES/PNOAMA/INTERPOLATION/NN/PNOA_MA_Molino_
img2 = imread('/Users/monica/Documents/TFG/IMAGENES/PNOAMA/MEJORES RESULTADOS/SRGAN/1_Batch

fig = figure();
ax(1) = axes('Units','normalized','Position', [ .1 .1 .4 .8]);
ax(2) = axes('Units','normalized','Position', [ .525 .1 .4 .8]);

imshow(img1, 'Parent', ax(1))
imshow(img2, 'Parent', ax(2))
linkaxes(ax)
```

Fig. 5.2.a Emerging window or pop-up with an example (DEMO - Matlab).

5.2.b Python (Jupyter Notebook)

Code snippet 5.2.b: DEMO - Python

```
%matplotlib widget

import matplotlib.pyplot as plt
import pandas as pd
from test_tools import load_image
import imshowpair

image1 = load_image('/Users/monica/Documents/TFG/SRGAN/Imgs_Test_QGIS/PNOA_MA/LR/256x256/PNOA_MA_Construc
image2 = load_image('/Users/monica/Documents/TFG/SRGAN/Imgs_Test_QGIS/PNOA_MA/LR/cv2/PNOA_MA_Construccione


imshowpair.imshowpair(image1,image2)
```

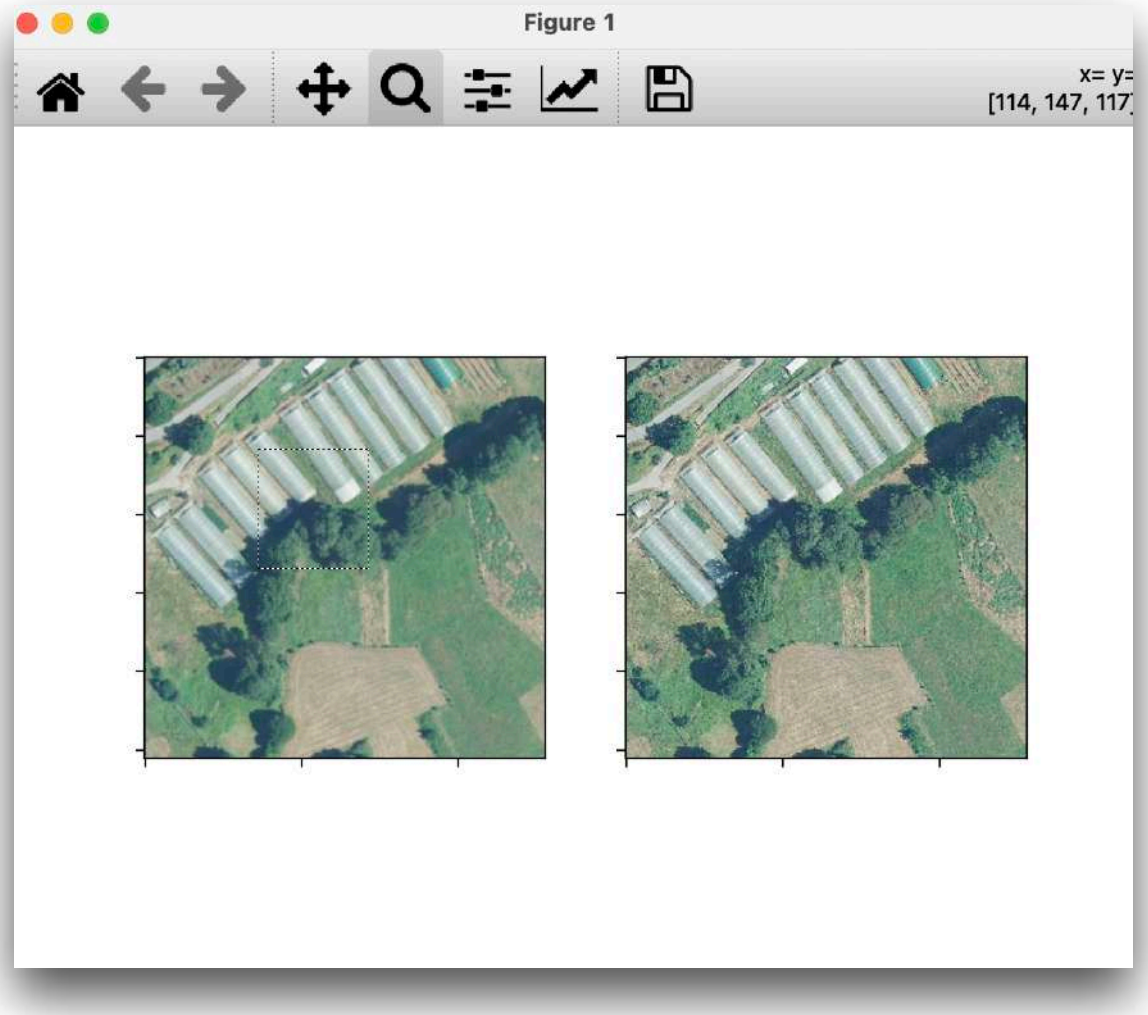

Fig. 5.2.b Emerge window with an example (DEMO - Python).

6. SOCIOECONOMIC ENVIROMENT

In this chapter, we present an economic estimation of the cost of this project. These have been divided into three specific costs: hardware costs, software costs, and labor costs. At the end of the document, the final cost of the project is expressed as two different final budgets, the material execution and the execution by contract.

6.1. Itemized Budget

6.1.a Hardware cost

The characteristics of the hardware used for this project were described in [Section 3.1.1]. We have used our PC for looking for information and testing the trained models; the other two PC with GPUs were used to carry out the experiments. To calculate the total cost, we have considered the working life of the equipment and the amortization factor. To our PC and both SSD, we consider ten years as their useful life [78][15]; for the PC used for experiments is two years. The usage time depends on each material, for example, SRGAN experiments were running for four months, instead of SRFBN experiments which duration was one month. These costs have been described in [Table 6.1.a]:

Table 6.1.a: Hardware costs

Material		Gross Cost	Unit	Usage Time (months)	Useful Life (months)	Amortization Factor	Total Cost
Personal PC		1.300,00 €	1	7	120	1/17	75,83 €
PC for experiments + GPU	Practicas + NVIDIA GeForce RTX 2080 Ti	1.850,00 €	1	4	36	1/9	205,56 €
	Omen + NVIDIA GeForce RTX 3080	2.150,00 €	1	1	36	1/36	59,72 €
SSD	Crucial MX500 500GB	60,00 €	1	4	120	1/30	2,00 €
	WDC WD BLACK SDBPNTY 1TB	120,00 €	1	1	120	1/120	1,00 €
TOTAL							**344,11 €**

6.1.b Software cost

The software characteristics were described in [Section 3.1.2]. To calculate the amortization factor, we use 4 years as Windows 10 Enterprise useful life; for the other licenses, i.e., for Microsoft Office and MATLAB, they have one year of useful life. This cost is calculated below, in [Table 6.1.b]:

Table 6.1.b: Software costs

Material		Gross Cost	Unit	Usage Time (months)	Useful Life (months)	Amortization Factor	Total Cost
Microsoft Office		69,00 €	1	4	12	1/3	23,00 €
Anaconda		Free	1	5	-	-	0,00 €
Python		Free	1	5	-	-	0,00 €
TensorFlow		Free	1	5	-	-	0,00 €
Keras		Free	1	5	-	-	0,00 €
Pytorch		Free	1	5	-	-	0,00 €
CUDA		Free	1	4	-	-	0,00 €
MATLAB R2021a Update 3 (9.10.0)		250,00 €	1	1	12	1/12	20,83 €
os	macOS Big Sur 11.4	Free	1	7	-	-	0,00 €
	Windows 10 Enterprise	259,00 €	1	4	48	1/12	21,58 €
TOTAL							**65,42 €**

6.1.c Labor cost

This project has been carried out by a geoinformatics engineer. To define this total cost, we had used the guide of the Adecco Group 2021 to define the salary, and time equivalent ion for ECTs (1 ECTs = 30h). This cost is calculated below:

Table 6.1.c: Labor costs

Position	Final Project (ECTs)	ECTs (h)	Annual Salary (€/year)	Unit Price (€/h)	Total Cost
Geoinformatics Engineer	12	360	26.084,88 €	14,56 €	5.240,27 €
TOTAL					**5.240,27 €**

6.2. Final Budget

6.2.a Material Execution Budget

The material execution budget has been calculated by adding the partial costs of each work unit previously exposed. The results are shown in [Table 6.2.a-I]:

Table 6.2.a-I: Material Execution Budget

Individual Cost	Total Cost
Labor	5.240,27 €
Hardware	344,11 €
Software	65,42 €
Total	**5.649,79 €**

The material execution budget of the project before adding taxes is **five thousand six hundred forty-nine euros and seventy-nine cents**.

6.2.b Execution Budget by Contract

The execution budget by contract has been calculated from the material execution budget calculated before, adding a 15% of benefits and the Spanish percentage tax of 21%. This is shown in the next [Table 6.2.a-II]:

Table 6.2.a-II: Material Execution Budget

Components	Total Cost
Material Execution Budget	5.649,79 €
Benefits (15%)	847,47 €
General Expenses (13%)	734,47 €
Total Before Taxes	**6.497,26 €**
VAT Tax (21%)	1.364,43 €
Execution Budget by Contract	**8.750,40 €**

The final execution budget by contract of this project is **eight thousand seven hundred fifty euros and forty cents**.

7. CONCLUSIONS AND FUTURE WORK

The purpose of this project was to investigate and compare some of the most common and newest super-resolution methods applied to aerial images. Also, we want to give an overview of SR and deep learning. For that, we have delved into the most common deep learning elements and their performance from a more theoretical point of view. And we have concluded with an empirical application of these concepts and their influence on final results. Below, we are going to highlight some of the most important ideas or conclusions finally obtained:

- **The great importance of using adequate training and validation dataset**: as we mentioned in the document, one of the most important things to consider before starting the training is to build a correct and complete dataset. These images will feed the network, and it will use it to learn about these. When we trained a network, the final purpose is to apply this network to similar images and obtain good results. For that, it is important **to compose the training and validation dataset with a group of images that can represent most of the general characteristics of the images of our interest**. Applying **data augmentation** [Section 4.1.1] also increases the quality of our input; it shows images with different changes or variations to our network, and this will improve its skills. Related to our dataset too, it is important to have a **homogenous dataset**, e.g., if we are interested in improving the resolution of images with roads and windmills, our dataset should have a similar number of images for both categories to make a more objective evaluation of results; if we do not have roads images in our dataset and we train our network only with windmills images, we will not have good results if we apply the trained network to images with roads.

  Another good example of this point is the results that we have obtained with the Pleiades satellite test dataset. To improve our SR images [Section 5.1.a], we should train our models with satellite images or images with the same resolution as Pleiades. Another recommendation or possible future experiment would be to train the architectures of SRFBN-P1 and P2 models with our dataset, i.e., with aerial images; this change will improve the metrics values and SR images.

- **A good definition of hyperparameters and their values**: we have followed a completely manual procedure to see the influence of these hyperparameters in our network, as we have explained in [Section 4.2.1.b] and [Section 4.2.2.b]. But if you want **to apply a faster and more efficient technique** you can try some **automatic techniques** as [49], [8], or [100]. A good definition of these values is super important to train well and efficiently. A good example of obtained results without adequate hyperparameter values is the SRFBN experiments, because how we explain in [Chapter 5.2.2.d], we have defined a small number of epochs for training and this directly affect other hyperparameters as learning rate, which is multiplied by 0,5 for every 200 epochs how we detailed in [Chapter 4.2.2.b], so this was not applied in most of the experiments because these were training with only 100 epochs. To improve our SR images generated by SRFBN-P3 and P4, we could increase the number of epochs to 100.000 and a million and for sure, we will obtain better results; and maybe, we could see other possible hyperparameters variations.

- **Hardware requirements**: these kinds of experiments used **big amounts of data** so specific hardware requirements are needed. Hundreds of images are introduced in training, and each pixel is used as input in each epoch. Also, training have several layers that generate more outputs that will be used as input for the next layers, and this will be repeated as many times as layers we have. Finally, we have to process a huge number of parameters. These **millions of parameters** have to be processed numerous times, for that is important to use adequate hardware. Before training starts running, we should think about its requirements. For these experiments, the use of **GPU** is highly recommended, and **parallel training** would be a good option too. The usage of **SSD** instead of a conventional disk is recommended to avoid possible problems as "*the bottleneck*" [109].

To conclude, we summarize the most significant visual and qualitative characteristics of the SR-generated images according to the upsampling method applied in [Table 7-I], and [Table 7-II]. In [Table 7-III] we are going to summarize the quantitative values.

Table 7-I: Summary of advantages and disadvantages of each upsampling method.

Model	Advantages	Disadvantages
NN Bilinear Bicubic Lanczos	Low complexity High efficiency Not special hardware requirements	Bad results
SRGAN	Better results Easy access to open-source Lot of available and accessible documentation	Huge and specific dataset Hyperparameter Optimization Lots of processing time The use of GPU and SSD is recommended
SRFBN		

Table 7-II: Summary of visual effects in SR-generated images for each upsampling model.

Model	Visual effects in SR images
Interpolation Methods	Blurred images. Difficult details detection. Bad object/background distinction.
SRGAN	Dark images. Noisy images (as a granulated filter). Good object/background distinction.
SRFBN	Good object/background distinction. Sharp images. Clear lines and edges. Good color adaptation.

Concerning qualitative characteristics, the best methods are SRGAN and SRFBN, regardless of time processing and hardware requirements. The SR-generated considerably improves, being the SRFBN the best network, in concrete, the first experiment ("*1_SRFBN-S_x4_BI*"), i.e., the trained model offered by [60], but these results are still scoped to improve. To obtain better results, we should use the same architecture, i.e.:

Table 7-III: Architecture with the best general qualitative (visual) results.

Model	Batch size	Workers	Filters	Epochs	T	G
1_SRFBN-S_x4_BI	16	4	32	1.000.000	4	3

The architecture shown in [Table 7-III] obtains the best general visual results but if we only work with images with stripes as roads; or single and concrete items as windmills, it is recommended to use the ("2_*SRFBN_x4_BI*") with the architecture shown below:

Table 7-IV: Architecture with the best qualitative (visual) results of specific and delimited items.

Model	Batch size	Workers	Filters	Epochs	T	G
2_*SRFBN_x4_BI*	16	4	64	1.000.000	4	6

The fact that this last architecture described in [Table 7-IV] had better visual results of edges, stripes, and elements well differentiate respect the background; it is because the architecture is deeper, with the double of extract feature maps, i.e., more filters are applied to our images, starting with easy and simple characteristics extraction, as edges, and growing the difficulty level. This process is reinforced by the curriculum learning strategy followed by this network and explained in [Section 2.5.2].

In conclusion, **SRFBN is the network with more possibilities to obtain the most optimal results**. Finally, **the best model is ("*1_SRFBN-S_x4_BI*")** and it is described in [Table 7-III]. To improve its results it would be possible, for that, it is recommended to train the same architecture with a specific training and validation dataset of our interest, but this is not strictly necessary because **results are very visually superior to LR**.

Also, to improve SRGAN visual results, it is recommended to modified the average of RGB values of the validating and training datasets. Then the results will be improved and the SR-generated images will not be as darker as results already obtained. This is an important detail to check, for future experiments with new datasets.

Regarding quantitative analysis, as we have analyzed in [Section 4.2.1.d] and [Section 4.2.2.d] all results are recorded in individual .csv, and in an excel where mean values have been calculated for each testing dataset and each experiment. As we can see in

[Table 5-I], [Table 5-II], and [Table 5-III], **SRFBN has the best metrics scores** followed by interpolation methods, and SRGAN with the lowest values. How we have demonstrated in [Section 5.1.d], the metrics scores help us, but not always these values are reliable and rigorous, as in the Pleiades dataset. Hence, **a visual analysis is indispensable to support quantitative results**.

How we have explained, it is very important to define correctly the average RGB values of each dataset, if these values are not correct, our generated images could suffer color and brightness variations as we had in SRGAN [Fig. 5.1.a-VI]. These variations affect metrics scores too, so maybe the network is able to generate images items perfectly but each pixel value is corrupted by small variation, obtaining lower metric values.

In conclusion, **the method with the highest PSNR and SSIM values is SRFBN,** model number one, **"*1_SRFBN-S_x4_BI*"** [Table 7-III]. Obtaining these maximum and minimum values:

Table 7-V: The best quantitative results (PSNR and SSIM) of SRFBN-P1

Model	Testing Dataset	PSNR max	SSIM max	PSNR min	SSIM min
1_SRFBN-S_x4_BI	PNOA MA	34,3638	0,847	29,6429	0,5067
	PNOA 10	**35,8884**	**0,9352**	29,3799	0,5521
	PLEIADES	28,6877	0,5985	27,5976	0,1113

Finally, the best results have been obtained with SRFBN, with the model 1_SRFBN-S_x4_BI, but these scores could improve applying the last recommendation.

How we have mentioned in [Section 1.1], SR methodology has several applications in different fields but this project has been realized with the purpose of studying and understanding how deep learning works and its performance in the SR world. Applying these technics and results to our field, we have some possible future applications as:

- Improving and updating the national cartography.
- Evaluating and analyzing of the effects of a natural disaster.
- Autonomous driving.
- Occupation and land-use.
- Cadastre etc.

A. ANNEXES

A.1. Generator Architecture - Models 1 and 4

Model: "model_6"

Layer (type)	Output Shape	Param #	Connected to
input_7 (InputLayer)	[(None, None, None,	0	
lambda_18 (Lambda)	(None, None, None, 3	0	input_7[0][0]
conv2d_164 (Conv2D)	(None, None, None, 6	15616	lambda_18[0][0]
p_re_lu_76 (PReLU)	(None, None, None, 6	64	conv2d_164[0][0]
conv2d_165 (Conv2D)	(None, None, None, 6	36928	p_re_lu_76[0][0]
batch_normalization_146 (BatchN	(None, None, None, 6	256	conv2d_165[0][0]
p_re_lu_77 (PReLU)	(None, None, None, 6	64	batch_normalization_146[0][0]
conv2d_166 (Conv2D)	(None, None, None, 6	36928	p_re_lu_77[0][0]
batch_normalization_147 (BatchN	(None, None, None, 6	256	conv2d_166[0][0]
add_68 (Add)	(None, None, None, 6	0	p_re_lu_76[0][0] batch_normalization_147[0][0]
conv2d_167 (Conv2D)	(None, None, None, 6	36928	add_68[0][0]
batch_normalization_148 (BatchN	(None, None, None, 6	256	conv2d_167[0][0]
p_re_lu_78 (PReLU)	(None, None, None, 6	64	batch_normalization_148[0][0]
conv2d_168 (Conv2D)	(None, None, None, 6	36928	p_re_lu_78[0][0]
batch_normalization_149 (BatchN	(None, None, None, 6	256	conv2d_168[0][0]
add_69 (Add)	(None, None, None, 6	0	add_68[0][0] batch_normalization_149[0][0]
conv2d_169 (Conv2D)	(None, None, None, 6	36928	add_69[0][0]
batch_normalization_150 (BatchN	(None, None, None, 6	256	conv2d_169[0][0]
p_re_lu_79 (PReLU)	(None, None, None, 6	64	batch_normalization_150[0][0]
conv2d_170 (Conv2D)	(None, None, None, 6	36928	p_re_lu_79[0][0]
batch_normalization_151 (BatchN	(None, None, None, 6	256	conv2d_170[0][0]
add_70 (Add)	(None, None, None, 6	0	add_69[0][0] batch_normalization_151[0][0]
conv2d_171 (Conv2D)	(None, None, None, 6	36928	add_70[0][0]
batch_normalization_152 (BatchN	(None, None, None, 6	256	conv2d_171[0][0]
p_re_lu_80 (PReLU)	(None, None, None, 6	64	batch_normalization_152[0][0]
conv2d_172 (Conv2D)	(None, None, None, 6	36928	p_re_lu_80[0][0]
batch_normalization_153 (BatchN	(None, None, None, 6	256	conv2d_172[0][0]
add_71 (Add)	(None, None, None, 6	0	add_70[0][0] batch_normalization_153[0][0]
conv2d_173 (Conv2D)	(None, None, None, 6	36928	add_71[0][0]
batch_normalization_154 (BatchN	(None, None, None, 6	256	conv2d_173[0][0]
p_re_lu_81 (PReLU)	(None, None, None, 6	64	batch_normalization_154[0][0]
conv2d_174 (Conv2D)	(None, None, None, 6	36928	p_re_lu_81[0][0]
batch_normalization_155 (BatchN	(None, None, None, 6	256	conv2d_174[0][0]
add_72 (Add)	(None, None, None, 6	0	add_71[0][0] batch_normalization_155[0][0]
conv2d_175 (Conv2D)	(None, None, None, 6	36928	add_72[0][0]
batch_normalization_156 (BatchN	(None, None, None, 6	256	conv2d_175[0][0]
p_re_lu_82 (PReLU)	(None, None, None, 6	64	batch_normalization_156[0][0]
conv2d_176 (Conv2D)	(None, None, None, 6	36928	p_re_lu_82[0][0]
batch_normalization_157 (BatchN	(None, None, None, 6	256	conv2d_176[0][0]
add_73 (Add)	(None, None, None, 6	0	add_72[0][0] batch_normalization_157[0][0]
conv2d_177 (Conv2D)	(None, None, None, 6	36928	add_73[0][0]
batch_normalization_158 (BatchN	(None, None, None, 6	256	conv2d_177[0][0]
p_re_lu_83 (PReLU)	(None, None, None, 6	64	batch_normalization_158[0][0]
conv2d_178 (Conv2D)	(None, None, None, 6	36928	p_re_lu_83[0][0]
batch_normalization_159 (BatchN	(None, None, None, 6	256	conv2d_178[0][0]
add_74 (Add)	(None, None, None, 6	0	add_73[0][0] batch_normalization_159[0][0]
conv2d_179 (Conv2D)	(None, None, None, 6	36928	add_74[0][0]
batch_normalization_160 (BatchN	(None, None, None, 6	256	conv2d_179[0][0]
p_re_lu_84 (PReLU)	(None, None, None, 6	64	batch_normalization_160[0][0]
conv2d_180 (Conv2D)	(None, None, None, 6	36928	p_re_lu_84[0][0]
batch_normalization_161 (BatchN	(None, None, None, 6	256	conv2d_180[0][0]
add_75 (Add)	(None, None, None, 6	0	add_74[0][0] batch_normalization_161[0][0]
conv2d_181 (Conv2D)	(None, None, None, 6	36928	add_75[0][0]
batch_normalization_162 (BatchN	(None, None, None, 6	256	conv2d_181[0][0]
p_re_lu_85 (PReLU)	(None, None, None, 6	64	batch_normalization_162[0][0]
conv2d_182 (Conv2D)	(None, None, None, 6	36928	p_re_lu_85[0][0]
batch_normalization_163 (BatchN	(None, None, None, 6	256	conv2d_182[0][0]
add_76 (Add)	(None, None, None, 6	0	add_75[0][0] batch_normalization_163[0][0]
conv2d_183 (Conv2D)	(None, None, None, 6	36928	add_76[0][0]

batch_normalization_164 (BatchN	(None, None, None, 6 256	conv2d_183[0][0]
p_re_lu_86 (PReLU)	(None, None, None, 6 64	batch_normalization_164[0][0]
conv2d_184 (Conv2D)	(None, None, None, 6 36928	p_re_lu_86[0][0]
batch_normalization_165 (BatchN	(None, None, None, 6 256	conv2d_184[0][0]
add_77 (Add)	(None, None, None, 6 0	add_76[0][0] batch_normalization_165[0][0]
conv2d_185 (Conv2D)	(None, None, None, 6 36928	add_77[0][0]
batch_normalization_166 (BatchN	(None, None, None, 6 256	conv2d_185[0][0]
p_re_lu_87 (PReLU)	(None, None, None, 6 64	batch_normalization_166[0][0]
conv2d_186 (Conv2D)	(None, None, None, 6 36928	p_re_lu_87[0][0]
batch_normalization_167 (BatchN	(None, None, None, 6 256	conv2d_186[0][0]
add_78 (Add)	(None, None, None, 6 0	add_77[0][0] batch_normalization_167[0][0]
conv2d_187 (Conv2D)	(None, None, None, 6 36928	add_78[0][0]
batch_normalization_168 (BatchN	(None, None, None, 6 256	conv2d_187[0][0]
p_re_lu_88 (PReLU)	(None, None, None, 6 64	batch_normalization_168[0][0]
conv2d_188 (Conv2D)	(None, None, None, 6 36928	p_re_lu_88[0][0]
batch_normalization_169 (BatchN	(None, None, None, 6 256	conv2d_188[0][0]
add_79 (Add)	(None, None, None, 6 0	add_78[0][0] batch_normalization_169[0][0]
conv2d_189 (Conv2D)	(None, None, None, 6 36928	add_79[0][0]
batch_normalization_170 (BatchN	(None, None, None, 6 256	conv2d_189[0][0]
p_re_lu_89 (PReLU)	(None, None, None, 6 64	batch_normalization_170[0][0]
conv2d_190 (Conv2D)	(None, None, None, 6 36928	p_re_lu_89[0][0]
batch_normalization_171 (BatchN	(None, None, None, 6 256	conv2d_190[0][0]
add_80 (Add)	(None, None, None, 6 0	add_79[0][0] batch_normalization_171[0][0]
conv2d_191 (Conv2D)	(None, None, None, 6 36928	add_80[0][0]
batch_normalization_172 (BatchN	(None, None, None, 6 256	conv2d_191[0][0]
p_re_lu_90 (PReLU)	(None, None, None, 6 64	batch_normalization_172[0][0]
conv2d_192 (Conv2D)	(None, None, None, 6 36928	p_re_lu_90[0][0]
batch_normalization_173 (BatchN	(None, None, None, 6 256	conv2d_192[0][0]
add_81 (Add)	(None, None, None, 6 0	add_80[0][0] batch_normalization_173[0][0]
conv2d_193 (Conv2D)	(None, None, None, 6 36928	add_81[0][0]
batch_normalization_174 (BatchN	(None, None, None, 6 256	conv2d_193[0][0]
p_re_lu_91 (PReLU)	(None, None, None, 6 64	batch_normalization_174[0][0]
conv2d_194 (Conv2D)	(None, None, None, 6 36928	p_re_lu_91[0][0]
batch_normalization_175 (BatchN	(None, None, None, 6 256	conv2d_194[0][0]
add_82 (Add)	(None, None, None, 6 0	add_81[0][0] batch_normalization_175[0][0]
conv2d_195 (Conv2D)	(None, None, None, 6 36928	add_82[0][0]
batch_normalization_176 (BatchN	(None, None, None, 6 256	conv2d_195[0][0]
p_re_lu_92 (PReLU)	(None, None, None, 6 64	batch_normalization_176[0][0]
conv2d_196 (Conv2D)	(None, None, None, 6 36928	p_re_lu_92[0][0]
batch_normalization_177 (BatchN	(None, None, None, 6 256	conv2d_196[0][0]
add_83 (Add)	(None, None, None, 6 0	add_82[0][0] batch_normalization_177[0][0]
conv2d_197 (Conv2D)	(None, None, None, 6 36928	add_83[0][0]
batch_normalization_178 (BatchN	(None, None, None, 6 256	conv2d_197[0][0]
add_84 (Add)	(None, None, None, 6 0	p_re_lu_76[0][0] batch_normalization_178[0][0]
conv2d_198 (Conv2D)	(None, None, None, 2 147712	add_84[0][0]
lambda_19 (Lambda)	(None, None, None, 6 0	conv2d_198[0][0]
p_re_lu_93 (PReLU)	(None, None, None, 6 64	lambda_19[0][0]
conv2d_199 (Conv2D)	(None, None, None, 2 147712	p_re_lu_93[0][0]
lambda_20 (Lambda)	(None, None, None, 6 0	conv2d_199[0][0]
p_re_lu_94 (PReLU)	(None, None, None, 6 64	lambda_20[0][0]
conv2d_200 (Conv2D)	(None, None, None, 3 15555	p_re_lu_94[0][0]
lambda_21 (Lambda)	(None, None, None, 3 0	conv2d_200[0][0]

A.2. Discriminator Architecture - Models 1 and 4

Model: "model_8"

Layer (type)	Output Shape	Param #
input_9 (InputLayer)	[(None, 96, 96, 3)]	0
lambda_26 (Lambda)	(None, 96, 96, 3)	0
conv2d_238 (Conv2D)	(None, 96, 96, 64)	1792
leaky_re_lu_18 (LeakyReLU)	(None, 96, 96, 64)	0
conv2d_239 (Conv2D)	(None, 48, 48, 64)	36928
batch_normalization_212 (Bat	(None, 48, 48, 64)	256
leaky_re_lu_19 (LeakyReLU)	(None, 48, 48, 64)	0
conv2d_240 (Conv2D)	(None, 48, 48, 128)	73856
batch_normalization_213 (Bat	(None, 48, 48, 128)	512
leaky_re_lu_20 (LeakyReLU)	(None, 48, 48, 128)	0
conv2d_241 (Conv2D)	(None, 24, 24, 128)	147584
batch_normalization_214 (Bat	(None, 24, 24, 128)	512
leaky_re_lu_21 (LeakyReLU)	(None, 24, 24, 128)	0
conv2d_242 (Conv2D)	(None, 24, 24, 256)	295168
batch_normalization_215 (Bat	(None, 24, 24, 256)	1024
leaky_re_lu_22 (LeakyReLU)	(None, 24, 24, 256)	0
conv2d_243 (Conv2D)	(None, 12, 12, 256)	590080
batch_normalization_216 (Bat	(None, 12, 12, 256)	1024
leaky_re_lu_23 (LeakyReLU)	(None, 12, 12, 256)	0
conv2d_244 (Conv2D)	(None, 12, 12, 512)	1180160
batch_normalization_217 (Bat	(None, 12, 12, 512)	2048
leaky_re_lu_24 (LeakyReLU)	(None, 12, 12, 512)	0
conv2d_245 (Conv2D)	(None, 6, 6, 512)	2359808
batch_normalization_218 (Bat	(None, 6, 6, 512)	2048
leaky_re_lu_25 (LeakyReLU)	(None, 6, 6, 512)	0
flatten_2 (Flatten)	(None, 18432)	0
dense_4 (Dense)	(None, 1024)	18875392
leaky_re_lu_26 (LeakyReLU)	(None, 1024)	0
dense_5 (Dense)	(None, 1)	1025

A.3. Generator Architecture - Models 2

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, None, None,	0	
lambda (Lambda)	(None, None, None, 3	0	input_1[0][0]
conv2d (Conv2D)	(None, None, None, 2	62464	lambda[0][0]
p_re_lu (PReLU)	(None, None, None, 2	256	conv2d[0][0]
conv2d_1 (Conv2D)	(None, None, None, 2	590080	p_re_lu[0][0]
batch_normalization (BatchNorma	(None, None, None, 2	1024	conv2d_1[0][0]
p_re_lu_1 (PReLU)	(None, None, None, 2	256	batch_normalization[0][0]
conv2d_2 (Conv2D)	(None, None, None, 2	590080	p_re_lu_1[0][0]
batch_normalization_1 (BatchNor	(None, None, None, 2	1024	conv2d_2[0][0]
add (Add)	(None, None, None, 2	0	p_re_lu[0][0] batch_normalization_1[0][0]
conv2d_3 (Conv2D)	(None, None, None, 2	590080	add[0][0]
batch_normalization_2 (BatchNor	(None, None, None, 2	1024	conv2d_3[0][0]
p_re_lu_2 (PReLU)	(None, None, None, 2	256	batch_normalization_2[0][0]
conv2d_4 (Conv2D)	(None, None, None, 2	590080	p_re_lu_2[0][0]
batch_normalization_3 (BatchNor	(None, None, None, 2	1024	conv2d_4[0][0]
add_1 (Add)	(None, None, None, 2	0	add[0][0] batch_normalization_3[0][0]
conv2d_5 (Conv2D)	(None, None, None, 2	590080	add_1[0][0]
batch_normalization_4 (BatchNor	(None, None, None, 2	1024	conv2d_5[0][0]
p_re_lu_3 (PReLU)	(None, None, None, 2	256	batch_normalization_4[0][0]
conv2d_6 (Conv2D)	(None, None, None, 2	590080	p_re_lu_3[0][0]
batch_normalization_5 (BatchNor	(None, None, None, 2	1024	conv2d_6[0][0]
add_2 (Add)	(None, None, None, 2	0	add_1[0][0] batch_normalization_5[0][0]
conv2d_7 (Conv2D)	(None, None, None, 2	590080	add_2[0][0]
batch_normalization_6 (BatchNor	(None, None, None, 2	1024	conv2d_7[0][0]
p_re_lu_4 (PReLU)	(None, None, None, 2	256	batch_normalization_6[0][0]
conv2d_8 (Conv2D)	(None, None, None, 2	590080	p_re_lu_4[0][0]
batch_normalization_7 (BatchNor	(None, None, None, 2	1024	conv2d_8[0][0]
add_3 (Add)	(None, None, None, 2	0	add_2[0][0] batch_normalization_7[0][0]
conv2d_9 (Conv2D)	(None, None, None, 2	590080	add_3[0][0]
batch_normalization_8 (BatchNor	(None, None, None, 2	1024	conv2d_9[0][0]
p_re_lu_5 (PReLU)	(None, None, None, 2	256	batch_normalization_8[0][0]
conv2d_10 (Conv2D)	(None, None, None, 2	590080	p_re_lu_5[0][0]
batch_normalization_9 (BatchNor	(None, None, None, 2	1024	conv2d_10[0][0]
add_4 (Add)	(None, None, None, 2	0	add_3[0][0] batch_normalization_9[0][0]
conv2d_11 (Conv2D)	(None, None, None, 2	590080	add_4[0][0]
batch_normalization_10 (BatchNo	(None, None, None, 2	1024	conv2d_11[0][0]
p_re_lu_6 (PReLU)	(None, None, None, 2	256	batch_normalization_10[0][0]
conv2d_12 (Conv2D)	(None, None, None, 2	590080	p_re_lu_6[0][0]
batch_normalization_11 (BatchNo	(None, None, None, 2	1024	conv2d_12[0][0]
add_5 (Add)	(None, None, None, 2	0	add_4[0][0] batch_normalization_11[0][0]
conv2d_13 (Conv2D)	(None, None, None, 2	590080	add_5[0][0]
batch_normalization_12 (BatchNo	(None, None, None, 2	1024	conv2d_13[0][0]
p_re_lu_7 (PReLU)	(None, None, None, 2	256	batch_normalization_12[0][0]
conv2d_14 (Conv2D)	(None, None, None, 2	590080	p_re_lu_7[0][0]
batch_normalization_13 (BatchNo	(None, None, None, 2	1024	conv2d_14[0][0]
add_6 (Add)	(None, None, None, 2	0	add_5[0][0] batch_normalization_13[0][0]
conv2d_15 (Conv2D)	(None, None, None, 2	590080	add_6[0][0]
batch_normalization_14 (BatchNo	(None, None, None, 2	1024	conv2d_15[0][0]
p_re_lu_8 (PReLU)	(None, None, None, 2	256	batch_normalization_14[0][0]
conv2d_16 (Conv2D)	(None, None, None, 2	590080	p_re_lu_8[0][0]
batch_normalization_15 (BatchNo	(None, None, None, 2	1024	conv2d_16[0][0]
add_7 (Add)	(None, None, None, 2	0	add_6[0][0] batch_normalization_15[0][0]
conv2d_17 (Conv2D)	(None, None, None, 2	590080	add_7[0][0]
batch_normalization_16 (BatchNo	(None, None, None, 2	1024	conv2d_17[0][0]
add_8 (Add)	(None, None, None, 2	0	p_re_lu[0][0] batch_normalization_16[0][0]
conv2d_18 (Conv2D)	(None, None, None, 1	2360320	add_8[0][0]
lambda_1 (Lambda)	(None, None, None, 2	0	conv2d_18[0][0]
p_re_lu_9 (PReLU)	(None, None, None, 2	256	lambda_1[0][0]
conv2d_19 (Conv2D)	(None, None, None, 1	2360320	p_re_lu_9[0][0]
lambda_2 (Lambda)	(None, None, None, 2	0	conv2d_19[0][0]
p_re_lu_10 (PReLU)	(None, None, None, 2	256	lambda_2[0][0]
conv2d_20 (Conv2D)	(None, None, None, 3	62211	p_re_lu_10[0][0]
lambda_3 (Lambda)	(None, None, None, 3	0	conv2d_20[0][0]

A.4. Discriminator Architecture - Models 2

Model: "model_2"

Layer (type)	Output Shape	Param #
input_3 (InputLayer)	[(None, 96, 96, 3)]	0
lambda_8 (Lambda)	(None, 96, 96, 3)	0
conv2d_42 (Conv2D)	(None, 96, 96, 256)	7168
leaky_re_lu (LeakyReLU)	(None, 96, 96, 256)	0
conv2d_43 (Conv2D)	(None, 48, 48, 256)	590080
batch_normalization_34 (Batc	(None, 48, 48, 256)	1024
leaky_re_lu_1 (LeakyReLU)	(None, 48, 48, 256)	0
conv2d_44 (Conv2D)	(None, 48, 48, 512)	1180160
batch_normalization_35 (Batc	(None, 48, 48, 512)	2048
leaky_re_lu_2 (LeakyReLU)	(None, 48, 48, 512)	0
conv2d_45 (Conv2D)	(None, 24, 24, 512)	2359808
batch_normalization_36 (Batc	(None, 24, 24, 512)	2048
leaky_re_lu_3 (LeakyReLU)	(None, 24, 24, 512)	0
conv2d_46 (Conv2D)	(None, 24, 24, 1024)	4719616
batch_normalization_37 (Batc	(None, 24, 24, 1024)	4096
leaky_re_lu_4 (LeakyReLU)	(None, 24, 24, 1024)	0
conv2d_47 (Conv2D)	(None, 12, 12, 1024)	9438208
batch_normalization_38 (Batc	(None, 12, 12, 1024)	4096
leaky_re_lu_5 (LeakyReLU)	(None, 12, 12, 1024)	0
conv2d_48 (Conv2D)	(None, 12, 12, 2048)	18876416
batch_normalization_39 (Batc	(None, 12, 12, 2048)	8192
leaky_re_lu_6 (LeakyReLU)	(None, 12, 12, 2048)	0
conv2d_49 (Conv2D)	(None, 6, 6, 2048)	37750784
batch_normalization_40 (Batc	(None, 6, 6, 2048)	8192
leaky_re_lu_7 (LeakyReLU)	(None, 6, 6, 2048)	0
flatten (Flatten)	(None, 73728)	0
dense (Dense)	(None, 1024)	75498496
leaky_re_lu_8 (LeakyReLU)	(None, 1024)	0
dense_1 (Dense)	(None, 1)	1025

A.5. Generator Architecture - Models 3

Model: "model_3"

Layer (type)	Output Shape	Param #	Connected to
input_4 (InputLayer)	[(None, None, None,	0	
lambda_9 (Lambda)	(None, None, None, 3	0	input_4[0][0]
conv2d_50 (Conv2D)	(None, None, None, 1	31232	lambda_9[0][0]
p_re_lu_22 (PReLU)	(None, None, None, 1	128	conv2d_50[0][0]
conv2d_51 (Conv2D)	(None, None, None, 1	147584	p_re_lu_22[0][0]
batch_normalization_41 (BatchNo	(None, None, None, 1	512	conv2d_51[0][0]
p_re_lu_23 (PReLU)	(None, None, None, 1	128	batch_normalization_41[0][0]
conv2d_52 (Conv2D)	(None, None, None, 1	147584	p_re_lu_23[0][0]
batch_normalization_42 (BatchNo	(None, None, None, 1	512	conv2d_52[0][0]
add_18 (Add)	(None, None, None, 1	0	p_re_lu_22[0][0] batch_normalization_42[0][0]
conv2d_53 (Conv2D)	(None, None, None, 1	147584	add_18[0][0]
batch_normalization_43 (BatchNo	(None, None, None, 1	512	conv2d_53[0][0]
p_re_lu_24 (PReLU)	(None, None, None, 1	128	batch_normalization_43[0][0]
conv2d_54 (Conv2D)	(None, None, None, 1	147584	p_re_lu_24[0][0]
batch_normalization_44 (BatchNo	(None, None, None, 1	512	conv2d_54[0][0]
add_19 (Add)	(None, None, None, 1	0	add_18[0][0] batch_normalization_44[0][0]
conv2d_55 (Conv2D)	(None, None, None, 1	147584	add_19[0][0]
batch_normalization_45 (BatchNo	(None, None, None, 1	512	conv2d_55[0][0]
p_re_lu_25 (PReLU)	(None, None, None, 1	128	batch_normalization_45[0][0]
conv2d_56 (Conv2D)	(None, None, None, 1	147584	p_re_lu_25[0][0]
batch_normalization_46 (BatchNo	(None, None, None, 1	512	conv2d_56[0][0]
add_20 (Add)	(None, None, None, 1	0	add_19[0][0] batch_normalization_46[0][0]
conv2d_57 (Conv2D)	(None, None, None, 1	147584	add_20[0][0]
batch_normalization_47 (BatchNo	(None, None, None, 1	512	conv2d_57[0][0]
p_re_lu_26 (PReLU)	(None, None, None, 1	128	batch_normalization_47[0][0]
conv2d_58 (Conv2D)	(None, None, None, 1	147584	p_re_lu_26[0][0]
batch_normalization_48 (BatchNo	(None, None, None, 1	512	conv2d_58[0][0]
add_21 (Add)	(None, None, None, 1	0	add_20[0][0] batch_normalization_48[0][0]
conv2d_59 (Conv2D)	(None, None, None, 1	147584	add_21[0][0]
batch_normalization_49 (BatchNo	(None, None, None, 1	512	conv2d_59[0][0]
p_re_lu_27 (PReLU)	(None, None, None, 1	128	batch_normalization_49[0][0]
conv2d_60 (Conv2D)	(None, None, None, 1	147584	p_re_lu_27[0][0]
batch_normalization_50 (BatchNo	(None, None, None, 1	512	conv2d_60[0][0]
add_22 (Add)	(None, None, None, 1	0	add_21[0][0] batch_normalization_50[0][0]
conv2d_61 (Conv2D)	(None, None, None, 1	147584	add_22[0][0]
batch_normalization_51 (BatchNo	(None, None, None, 1	512	conv2d_61[0][0]
p_re_lu_28 (PReLU)	(None, None, None, 1	128	batch_normalization_51[0][0]
conv2d_62 (Conv2D)	(None, None, None, 1	147584	p_re_lu_28[0][0]
batch_normalization_52 (BatchNo	(None, None, None, 1	512	conv2d_62[0][0]
add_23 (Add)	(None, None, None, 1	0	add_22[0][0] batch_normalization_52[0][0]
conv2d_63 (Conv2D)	(None, None, None, 1	147584	add_23[0][0]
batch_normalization_53 (BatchNo	(None, None, None, 1	512	conv2d_63[0][0]
p_re_lu_29 (PReLU)	(None, None, None, 1	128	batch_normalization_53[0][0]
conv2d_64 (Conv2D)	(None, None, None, 1	147584	p_re_lu_29[0][0]
batch_normalization_54 (BatchNo	(None, None, None, 1	512	conv2d_64[0][0]
add_24 (Add)	(None, None, None, 1	0	add_23[0][0] batch_normalization_54[0][0]

conv2d_65 (Conv2D)	(None, None, None, 1	147584	add_24[0][0]
batch_normalization_55 (BatchNo	(None, None, None, 1	512	conv2d_65[0][0]
p_re_lu_30 (PReLU)	(None, None, None, 1	128	batch_normalization_55[0][0]
conv2d_66 (Conv2D)	(None, None, None, 1	147584	p_re_lu_30[0][0]
batch_normalization_56 (BatchNo	(None, None, None, 1	512	conv2d_66[0][0]
add_25 (Add)	(None, None, None, 1	0	add_24[0][0] batch_normalization_56[0][0]
conv2d_67 (Conv2D)	(None, None, None, 1	147584	add_25[0][0]
batch_normalization_57 (BatchNo	(None, None, None, 1	512	conv2d_67[0][0]
add_26 (Add)	(None, None, None, 1	0	p_re_lu_22[0][0] batch_normalization_57[0][0]
conv2d_68 (Conv2D)	(None, None, None, 5	590336	add_26[0][0]
lambda_10 (Lambda)	(None, None, None, 1	0	conv2d_68[0][0]
p_re_lu_31 (PReLU)	(None, None, None, 1	128	lambda_10[0][0]
conv2d_69 (Conv2D)	(None, None, None, 5	590336	p_re_lu_31[0][0]
lambda_11 (Lambda)	(None, None, None, 1	0	conv2d_69[0][0]
p_re_lu_32 (PReLU)	(None, None, None, 1	128	lambda_11[0][0]
conv2d_70 (Conv2D)	(None, None, None, 3	31107	p_re_lu_32[0][0]
lambda_12 (Lambda)	(None, None, None, 3	0	conv2d_70[0][0]

A.6. Discriminator Architecture - Models 3

```
Model: "model_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_6 (InputLayer)         [(None, 96, 96, 3)]       0
_________________________________________________________________
lambda_17 (Lambda)           (None, 96, 96, 3)         0
_________________________________________________________________
conv2d_92 (Conv2D)           (None, 96, 96, 128)       3584
_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)    (None, 96, 96, 128)       0
_________________________________________________________________
conv2d_93 (Conv2D)           (None, 48, 48, 128)       147584
_________________________________________________________________
batch_normalization_75 (Batc (None, 48, 48, 128)       512
_________________________________________________________________
leaky_re_lu_10 (LeakyReLU)   (None, 48, 48, 128)       0
_________________________________________________________________
conv2d_94 (Conv2D)           (None, 48, 48, 256)       295168
_________________________________________________________________
batch_normalization_76 (Batc (None, 48, 48, 256)       1024
_________________________________________________________________
leaky_re_lu_11 (LeakyReLU)   (None, 48, 48, 256)       0
_________________________________________________________________
conv2d_95 (Conv2D)           (None, 24, 24, 256)       590080
_________________________________________________________________
batch_normalization_77 (Batc (None, 24, 24, 256)       1024
_________________________________________________________________
leaky_re_lu_12 (LeakyReLU)   (None, 24, 24, 256)       0
_________________________________________________________________
conv2d_96 (Conv2D)           (None, 24, 24, 512)       1180160
_________________________________________________________________
batch_normalization_78 (Batc (None, 24, 24, 512)       2048
_________________________________________________________________
leaky_re_lu_13 (LeakyReLU)   (None, 24, 24, 512)       0
_________________________________________________________________
conv2d_97 (Conv2D)           (None, 12, 12, 512)       2359808
_________________________________________________________________
batch_normalization_79 (Batc (None, 12, 12, 512)       2048
_________________________________________________________________
leaky_re_lu_14 (LeakyReLU)   (None, 12, 12, 512)       0
_________________________________________________________________
conv2d_98 (Conv2D)           (None, 12, 12, 1024)      4719616
_________________________________________________________________
batch_normalization_80 (Batc (None, 12, 12, 1024)      4096
_________________________________________________________________
leaky_re_lu_15 (LeakyReLU)   (None, 12, 12, 1024)      0
_________________________________________________________________
conv2d_99 (Conv2D)           (None, 6, 6, 1024)        9438208
_________________________________________________________________
batch_normalization_81 (Batc (None, 6, 6, 1024)        4096
_________________________________________________________________
leaky_re_lu_16 (LeakyReLU)   (None, 6, 6, 1024)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 36864)             0
_________________________________________________________________
dense_2 (Dense)              (None, 1024)              37749760
_________________________________________________________________
leaky_re_lu_17 (LeakyReLU)   (None, 1024)              0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 1025
=================================================================
```

A.7. Github

https://github.com/moniap24/TFG_2021

REFERENCES

[1] *7.6. Residual Networks (ResNet)—Dive into Deep Learning 0.17.0 documentation*. (n.d.). Retrieved 29 August 2021, from https://d2l.ai/chapter_convolutional-modern/resnet.html

[2] Ahn, N., Kang, B., & Sohn, K.-A. (2018). Fast, Accurate, and Lightweight Super-Resolution with Cascading Residual Network. *ArXiv:1803.08664 [Cs]*. http://arxiv.org/abs/1803.08664

[3] Alto, V. (2019, July 6). *Neural Networks: Parameters, hyperparameters and optimization strategies*. Medium. https://towardsdatascience.com/neural-networks-parameters-hyperparameters-and-optimization-strategies-3f0842fac0a5.Kumar, S. (2020, June 9).

[4] *Anaconda Documentation—Anaconda documentation*. (n.d.). Retrieved 5 July 2021, from https://docs.anaconda.com/

[5] *Artificial Neural Network for Machine Learning– Structure & Layers | by Rinu Gour | Javarevisited | Medium*. (n.d.). Retrieved 13 August 2021, from https://medium.com/javarevisited/artificial-neural-network-for-machine-learning-structure-layers-a031fcb279d7.

[6] *Basic digital image concepts—Knowledge Kitchen*. (n.d.). Retrieved 30 August 2021, from https://knowledge.kitchen/Basic_digital_image_concepts

[7] Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: A survey. *ArXiv:1502.05767 [Cs, Stat]*. http://arxiv.org/abs/1502.05767.

[8] Bergstra, J., & Bengio, Y. (n.d.). Random Search for Hyper-Parameter Optimization. 25.

[9] Buslaev, A., Iglovikov, V., Khvedchenya, E., Parinov, A., Druzhinin, M., & Kalinin, A. (2020). Albumentations: Fast and Flexible Image Augmentations. *Information*, *11*, 125. https://doi.org/10.3390/info11020125

[10] Caballero, J., Ledig, C., Aitken, A., Acosta, A., Totz, J., Wang, Z., & Shi, W. (2017). *Real-Time Video Super-Resolution with Spatio-Temporal Networks and Motion Compensation*. *ArXiv:1611.05250 [Cs]*. http://arxiv.org/abs/1611.05250

[11] Calcagni, L. L. R. (n.d.). Redes Generativas Antagónicas y sus aplicaciones. 72.

[12] Chollet, F. (2018). *Deep learning with Python*. Manning Publications Co.

[13] *Convolutional Neural Networks For All | Part II | by Jan Zawadzki | Machine Learning World | Medium*. (n.d.). Retrieved 31 August 2021, from https://medium.com/machine-learning-world/convolutional-neural-networks-for-all-part-ii-b4cb41d424fd

[14] *CS231n Convolutional Neural Networks for Visual Recognition*. (n.d.). Retrieved 29 August 2021, from https://cs231n.github.io/

[15] *¿Cuánto tiempo duran las Mac antes de que sea necesario reemplazarlas? - SpanishNewsNow*. (n.d.). Retrieved 4 September 2021, from https://www.spanishnewsnow.com/cuanto-tiempo-duran-las-mac-antes-de-que-sea-necesario-reemplazarlas/

[16] *CUDA CORES de NVIDIA ¿ Qué son?* (2017, August 3). HardwarEsfera. https://hardwaresfera.com/articulos/hablamos-profundidad-los-nvidia-cuda-core/

[17] *CUDA Toolkit Documentation*. (n.d.). Retrieved 5 July 2021, from https://docs.nvidia.com/cuda/

[18] Dai, T., Cai, J., Zhang, Y., Xia, S.-T., & Zhang, L. (2019). Second-Order Attention Network for Single Image Super-Resolution. *2019 IEEE/CVF Conference on Computer Vision*

and Pattern Recognition (CVPR), 11057–11066. https://doi.org/10.1109/CVPR.2019.01132

[19] *Data augmentation | TensorFlow Core*. (n.d.). TensorFlow. Retrieved 14 August 2021, from https://www.tensorflow.org/tutorials/images/data_augmentation

[20] *Deeply-Recursive Convolutional Network for Image Super-Resolution | IEEE Conference* Publication | IEEE Xplore. (n.d.). Retrieved 3 September 2021, from https://ieeexplore.ieee.org/document/7780550

[21] Dejan Grabovičkić, Pablo Benitez, Juan C. Miñano, Pablo Zamora, Marina Buljan, Bharathwaj Narasimhan, Milena I. Nikolic, Jesus Lopez, Jorge Gorospe, Eduardo Sanchez, Carmen Lastres, & Ruben Mohedano. (2017). *Super-resolution optics for virtual reality*. *10335*. https://doi.org/10.1117/12.2270178

[22] Dianyuan Han. (2013). Comparison of Commonly Used Image Interpolation Methods. Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013), 1556–1559. https://doi.org/10.2991/iccsee.2013.391

[23] Digital image file formats | IlluScientia: Scientific illustration & animation - graphisme & illustration scientifique. (n.d.). Retrieved 5 April 2021, from http://www.illuscientia.com/resources/digital-image-file-formats/

[24] *DIV2K Dataset*. (n.d.). Retrieved 2 August 2021, from https://data.vision.ee.ethz.ch/cvl/DIV2K/

[25] *Documentation*. (n.d.). Retrieved 5 July 2021, from https://www.qgis.org/en/docs/index.html

[26] Dong, C., Loy, C. C., He, K., & Tang, X. (2015). Image Super-Resolution Using Deep Convolutional Networks. *ArXiv:1501.00092 [Cs]*. http://arxiv.org/abs/1501.00092

[27] Dong, C., Loy, C. C., He, K., & Tang, X. (2014). *Learning a Deep Convolutional Network for Image Super-Resolution*. In D. Fleet, T. Pajdla, B. Schiele, & T. Tuytelaars (Eds.), *Computer Vision – ECCV 2014* (pp. 184–199). Springer International Publishing. https://doi.org/10.1007/978-3-319-10593-2_13

[28] Dong, C., Loy, C. C., & Tang, X. (2016). *Accelerating the Super-Resolution Convolutional Neural Network*. *ArXiv:1608.00367 [Cs]*. http://arxiv.org/abs/1608.00367

[29] Fadnavis, S. (2014). Image Interpolation Techniques in Digital Image Processing: An Overview. *International Journal Of Engineering Research and Application*, *4*, 2248–962270.

[30] Farsiu, S., Robinson, M. D., Elad, M., & Milanfar, P. (2004). Fast and Robust Multiframe Super Resolution. *IEEE Transactions on Image Processing*, *13*(10), 1327–1344. https://doi.org/10.1109/TIP.2004.834669

[31] Filter, J. (n.d.). split-folders: Split folders with files (e.g. images) into training, validation and test (dataset) folders. (0.4.3) [Python]. Retrieved 30 July 2021, from https://github.com/jfilter/split-folders

[32] Frank Rosenblatt. (2021). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Frank_Rosenblatt&oldid=1032776043.

[33] García, J. B., & Pedrero, A. M. G. (n.d.). Generación de Imágenes Médicas Anotadas para el Entrenamiento de Modelos Automáticos de Análisis de Anomalías. 67.

[34] Garcıa, R., & Montolıo, M. (n.d.). La interpolación aplicada al procesamiento de imágenes digitales. 74.

[35] *GDAL/OGR Python API*. (n.d.). Retrieved 24 July 2021, from https://gdal.org/python/

[36] Geográfica, O. A. C. N. de I. (n.d.). *Centro de Descargas del CNIG (IGN)*. Centro de Descargas del CNIG. Retrieved 30 July 2021, from http://centrodedescargas.cnig.es

[37] Glasner, D., Bagon, S., & Irani, M. (2009). Super-resolution from a single image. *2009 IEEE 12th International Conference on Computer Vision*, 349–356. https://doi.org/10.1109/ICCV.2009.5459271

[38] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative Adversarial Networks. *ArXiv:1406.2661 [Cs, Stat]*. http://arxiv.org/abs/1406.2661

[39] *Gradient descent*. (2017, July 14). Jeremy Jordan. https://www.jeremyjordan.me/gradient-descent/

[40] *Guardando y Serializando Modelos con TensorFlow Keras*. (n.d.). Retrieved 29 August 2021, from https://www.tensorflow.org/guide/keras/save_and_serialize?hl=es-419

[41] Gupta, R., Sharma, A., & Kumar, A. (2020). Super-Resolution using GANs for Medical Imaging. *Procedia Computer Science*, *173*, 28–35. https://doi.org/10.1016/j.procs.2020.06.005

[42] Han, W., Chang, S., Liu, D., Yu, M., Witbrock, M., & Huang, T. S. (2018). Image Super-Resolution via Dual-State Recurrent Networks. *ArXiv:1805.02704 [Cs]*. http://arxiv.org/abs/1805.02704

[43] He, K., Zhang, X., Ren, S., & Sun, J. (n.d.). *Identity Mappings in Deep Residual Networks*. 15.

[44] Horé, A., & Ziou, D. (2010). *Image quality metrics: PSNR vs. SSIM*. 2366–2369. https://doi.org/10.1109/ICPR.2010.579 Hughes, C. G., & Ramsey, M. S. (2010). Super-resolution of THEMIS thermal infrared data: Compositional relationships of surface units below the 100 meter scale on Mars. *Icarus*, *208*(2), 704–720. https://doi.org/10.1016/j.icarus.2010.02.023

[45] Hughes, C. G., & Ramsey, M. S. (2010). Super-resolution of THEMIS thermal infrared data: Compositional relationships of surface units below the 100 meter scale on Mars. *Icarus*, *208*(2), 704–720. https://doi.org/10.1016/j.icarus.2010.02.023

[46] Hu, X., Mu, H., Zhang, X., Wang, Z., Tan, T., & Sun, J. (2019). Meta-SR: A Magnification-Arbitrary Network for Super-Resolution. *ArXiv:1903.00875 [Cs]*. http://arxiv.org/abs/1903.00875

[47] Image Augmentation | Pytorch Image Augmentation. (2019, December 5). *Analytics Vidhya*. https://www.analyticsvidhya.com/blog/2019/12/image-augmentation-deep-learning-pytorch/

[48] Imagen digital. (2021). In *Wikipedia, la enciclopedia libre*. https://es.wikipedia.org/w/index.php?title=Imagen_digital&oldid=133463295

[49] *Intro to Model Tuning: Grid and Random Search*. (n.d.). Retrieved 24 July 2021, from https://kaggle.com/willkoehrsen/intro-to-model-tuning-grid-and-random-search

[50] Irani, M., & Peleg, S. (1991). Improving resolution by image registration. *CVGIP: Graphical Models and Image Processing*, *53*(3), 231–239. https://doi.org/10.1016/1049-9652(91)90045-L

[51] J. S. Isaac and R. Kulkarni, "Super resolution techniques for medical image processing," *2015 International Conference on Technologies for Sustainable Development (ICTSD)*, 2015, pp. 1-6, doi: 10.1109/ICTSD.2015.7095900.

[52] *Keras: The Python deep learning API*. (n.d.). Retrieved 5 July 2021, from https://keras.io/

[53] Kingma, D. P., & Ba, J. (2017). Adam: A Method for Stochastic Optimization. *ArXiv:1412.6980 [Cs]*. http://arxiv.org/abs/1412.6980

[54] Kouame, D., & Ploquin, M. (2009). Super-resolution in medical imaging: An illustrative approach through ultrasound. *2009 IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, 249–252. https://doi.org/10.1109/ISBI.2009.5193030

[55] Krasser, M. (2021). *Single Image Super-Resolution with EDSR, WDSR and SRGAN* [Jupyter Notebook]. https://github.com/krasserm/super-resolution (Original work published 2018)

[56] Ledig, C., Theis, L., Huszar, F., Caballero, J., Cunningham, A., Acosta, A., Aitken, A., Tejani, A., Totz, J., Wang, Z., & Shi, W. (2017). Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 105–114. https://doi.org/10.1109/CVPR.2017.19

[57] Li, J., Fang, F., Mei, K., & Zhang, G. (2018). *Multi-scale Residual Network for Image Super-Resolution*. 517–532. https://openaccess.thecvf.com/content_ECCV_2018/html/Juncheng_Li_Multi-scale_Residual_Network_ECCV_2018_paper.html

[58] Li, L., Yu, Q., Yuan, Y., Shang, Y., Lu, H., & Sun, X. (2009). Super-resolution reconstruction and higher-degree function deformation model based matching for Chang'E-1 lunar images. *Science in China Series E: Technological Sciences*, *52*(12), 3468. https://doi.org/10.1007/s11431-009-0334-7

[59] Li, Z., Peng, Q., Bhanu, B., Zhang, Q., & He, H. (2018). Super resolution for astronomical observations. *Astrophysics and Space Science*, *363*. https://doi.org/10.1007/s10509-018-3315-0

[60] Li, Z. (2021). *Feedback Network for Image Super-Resolution [arXiv] [CVF] [Poster]* [Python]. https://github.com/Paper99/SRFBN_CVPR19 (Original work published 2019)

[61] Li, Z., Yang, J., Liu, Z., Yang, X., Jeon, G., & Wu, W. (2019). Feedback Network for Image Super-Resolution. *ArXiv:1903.09814 [Cs]*. http://arxiv.org/abs/1903.09814

[62] Lim, B. (2021). *NTIRE2017 Super-resolution Challenge: SNU_CVLab* [Lua]. https://github.com/limbee/NTIRE2017 (Original work published 2017) Chollet, F. (2018). *Deep learning with Python*. Manning Publications Co.

[63] Lim, B., Son, S., Kim, H., Nah, S., & Lee, K. M. (2017). Enhanced Deep Residual Networks for Single Image Super-Resolution. *ArXiv:1707.02921 [Cs]*. http://arxiv.org/abs/1707.02921

[64] Linwei Yue, Huanfeng Shen, Jie Li, Qiangqiang Yuan, Hongyan Zhang, Liangpei Zhang (2016), Image super-resolution: The techniques, applications, and future. *Signal Processing* (128), 389-408, ISSN 0165-1684, https://doi.org/10.1016/j.sigpro.2016.05.002

[65] List of games that support 4K resolution—PCGamingWiki PCGW - bugs, fixes, crashes, mods, guides and improvements for every PC game. (n.d.). Retrieved 27 August 2021, from https://www.pcgamingwiki.com/wiki/List_of_games_that_support_4K_resolution

[66] M. Haris, G. Shakhnarovich, and N. Ukita, (2018). Deep backp-rojection networks for super-resolution. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[67] Mahajan, S. H., & Harpale, V. K. (2015). Adaptive and Non-adaptive Image Interpolation Techniques. *2015 International Conference on Computing Communication Control and Automation*, 772–775. https://doi.org/10.1109/ICCUBEA.2015.154

[68] *MATLAB Documentation*. (n.d.). Retrieved 5 July 2021, from https://www.mathworks.com/help/matlab/

[69] *Miniconda—Conda documentation*. (n.d.). Retrieved 5 July 2021, from https://docs.conda.io/en/latest/miniconda.html

[70] Morales, L. G. (2021). Tratamiento Digital de la Imagen. BOD GmbH DE.

[71] Moses, C. J., Selvathi, D., & Sophia, V. M. A. (2014). VLSI Architectures for Image Interpolation: A Survey. *VLSI Design*, *2014*, e872501. https://doi.org/10.1155/2014/872501

[72] *Multi-layer Perceptron using Keras on MNIST dataset for Digit Classification | by Rana singh | Analytics Vidhya | Medium*. (n.d.). Retrieved 31 August 2021, from https://medium.com/analytics-vidhya/multi-layer-perceptron-using-keras-on-mnist-dataset-for-digit-classification-problem-relu-a276cbf05e97

[73] Munt, A. M. (n.d.). Introducción a los modelos de redes neuronales artificiales El Perceptrón simple y multicapa. 47.

[74] Nguyen, K., Fookes, C., Sridharan, S., Tistarelli, M., & Nixon, M. (2018). Super-Resolution for Biometrics: A Comprehensive Survey. *Pattern Recognition*, *78*, 23–42. https://doi.org/10.1016/j.patcog.2018.01.002

[75] *Novedades de las actualizaciones para macOS Big Sur*. (n.d.). Apple Support. Retrieved 5 July 2021, from https://support.apple.com/es-es/HT211896

[76] *Observatorio de I+D+i UPM*. (n.d.). Retrieved 3 September 2021, from http://www.upm.es/observatorio/vi/index.jsp?pageac=grupo.jsp&idGrupo=337

[77] *Overview of various Optimizers in Neural Networks*. Medium. https://towardsdatascience.com/overview-of-various-optimizers-in-neural-networks-17c1be2df6d5.

[78] Pastor, J. (2020, January 27). *Mitos y realidades de la fiabilidad de los SSD: Su vida útil es (probablemente) más larga de lo que esperabas*. Xataka. https://www.xataka.com/componentes/mitos-realidades-fiabilidad-ssd-su-vida-util-probablemente-larga-que-esperabas

[79] *Practical Guide to Hyperparameters Optimization for Deep Learning Models*. (2018, September 5). FloydHub Blog. https://blog.floydhub.com/guide-to-hyperparameters-search-for-deep-learning-models/

[80] *PyTorch*. (n.d.). Retrieved 5 July 2021, from https://www.pytorch.org

[81] Qian, N. (1999). On the Momentum Term in Gradient Descent Learning Algorithms.

[82] Radhakrishnan, P. (2017, October 18). *What are Hyperparameters? And How to tune the Hyperparameters in a Deep Neural Network?* Medium. https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a

[83] Saha, S. (2018, December 17). *A Comprehensive Guide to Convolutional Neural Networks—The ELI5 way*. Medium. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[84] Sajjadi, M. S. M., Schölkopf, B., & Hirsch, M. (2017). EnhanceNet: Single Image Super-Resolution Through Automated Texture Synthesis. *ArXiv:1612.07919 [Cs, Stat]*. http://arxiv.org/abs/1612.07919

[85] Satélite de imágenes PLEIADES. (n.d.). *Geocento Spain proveedor online de imágenes de satélites*. Retrieved 30 July 2021, from http://geocento.es/galeria-de-satelites-para-buscar-y-adquirir-imagenes/satelite-imagenes-pleiades/

[86] Setiadi, D. R. I. M. (2021). PSNR vs SSIM: Imperceptibility quality assessment for image steganography. *Multimedia Tools and Applications*, *80*(6), 8423–8444. https://doi.org/10.1007/s11042-020-10035-z

[87] *Setting the learning rate of your neural network*. (n.d.). Retrieved 24 July 2021, from https://www.jeremyjordan.me/nn-learning-rate/

[88] SHARMA, S. (2021, July 4). *Activation Functions in Neural Networks*. Medium. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

[89] Sharma, S., Sharma, S., Scholar, U., & Athaiya, A. (2020). *ACTIVATION FUNCTIONS IN NEURAL NETWORKS*. *International Journal of Engineering Applied Sciences and Technology*. *04*. *310-316*. *10.33564/IJEAST.2020.v04i12.054*.

[90] Sung Cheol Park, Min Kyu Park, & Moon Gi Kang. (2003). Super-resolution image reconstruction: A technical overview. *IEEE Signal Processing Magazine*, *20*(3), 21–36. https://doi.org/10.1109/MSP.2003.1203207

[91] *Supervised vs. Unsupervised Learning: What's the Difference?* (2021, March 12). https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning.

[92] Tai, Y., Yang, J., & Liu, X. (2017). Image Super-Resolution via Deep Recursive Residual Network. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2790–2798. https://doi.org/10.1109/CVPR.2017.298

[93] Tai, Y., Yang, J., Liu, X., & Xu, C. (2017). MemNet: A Persistent Memory Network for Image Restoration. *ArXiv:1708.02209 [Cs]*. http://arxiv.org/abs/1708.02209

[94] Tanco, M. M. (n.d.). Super-Resolución en Imágenes. 79.

[95] Team, K. (n.d.). *Keras documentation: GaussianNoise layer*. Retrieved 14 August 2021, from https://keras.io/api/layers/regularization_layers/gaussian_noise/

[96] Team, K. (n.d.). *Keras documentation: Model saving & serialization APIs*. Retrieved 29 August 2021, from https://keras.io/api/models/model_saving_apis/#load_weights-method

[97] *Televisión digital—Televisión en Alta Definición*. (n.d.). Retrieved 27 August 2021, from https://televisiondigital.mineco.gob.es/ayuda-ciudadano/sala-prensa/Paginas/que-es-hdtv.aspx

[98] TensorFlow. Retrieved 24 July 2021, from https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/PiecewiseConstantDecay?hl=es-419

[99] *Welcome to Python.org*. (n.d.). Python.Org. Retrieved 5 July 2021, from https://www.python.org/

[100] *Tf.image.flip_left_right | TensorFlow Core v2.5.0*. (n.d.). TensorFlow. Retrieved 14 August 2021, from https://www.tensorflow.org/api_docs/python/tf/image/flip_left_right?hl=es-419

[101] *Tf.image.rot90 | TensorFlow Core v2.5.0*. (n.d.). TensorFlow. Retrieved 14 August 2021, from https://www.tensorflow.org/api_docs/python/tf/image/rot90?hl=es-419

[102] *Tf.keras.applications.vgg19.VGG19 | TensorFlow Core v2.5.0*. (n.d.). Retrieved 24 July 2021, from https://www.tensorflow.org/api_docs/python/tf/keras/applications/vgg19/VGG19*Tf.keras.optimizers.schedules.PiecewiseConstantDecay*. (n.d.). TensorFlow. Retrieved 24 July 2021, from https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/PiecewiseConstantDecay?hl=es-419

[103] *Tf.keras.layers.LeakyReLU | TensorFlow Core v2.5.0*. (n.d.). Retrieved 2 August 2021, from https://www.tensorflow.org/api_docs/python/tf/keras/layers/LeakyReLU

[104] *Tf.keras.layers.PReLU | TensorFlow Core v2.5.0*. (n.d.). Retrieved 2 August 2021, from https://www.tensorflow.org/api_docs/python/tf/keras/layers/PReLU

[105] *Tf.keras.optimizers.Adam | TensorFlow Core v2.5.0*. (n.d.). TensorFlow. Retrieved 24 July 2021, from https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam?hl=es-419

[106] *Tf.random.uniform | TensorFlow Core v2.5.0*. (n.d.). TensorFlow. Retrieved 14 August 2021, from https://www.tensorflow.org/api_docs/python/tf/random/uniform?hl=es-419

[107] Tong, T., Li, G., Liu, X., & Gao, Q. (2017). *Image Super-Resolution Using Dense Skip Connections*. 4799–4807. https://openaccess.thecvf.com/content_iccv_2017/html/Tong_Image_Super-Resolution_Using_ICCV_2017_paper.html

[108] *torch.nn.functional.interpolate—PyTorch 1.9.0 documentation*. (n.d.). Retrieved 31 August 2021, from https://pytorch.org/docs/

stable/generated/torch.nn.functional.interpolate.html

[109] Usuario, S. garantizas que tu hardware esté adecuadamente equilibrado puedes evitar un cuello de botella y lograr una mejor experiencia de. (n.d.). *¿Qué es lo que provoca cuellos de botella en mi PC?* Intel. Retrieved 25 August 2021, from https://www.intel.com/content/www/es/es/gaming/resources/what-is-bottlenecking-my-pc.html

[110] *VGGNet Architecture Explained. VGGNet is a Convolutional Neural… | by Prabin Nepal | Analytics Vidhya | Medium*. (n.d.). Retrieved 29 August 2021, from https://medium.com/analytics-vidhya/vggnet-architecture-explained-e5c7318aa5b6

[111] Victoria, H., & Maragatham, G. (2021). Automatic tuning of hyperparameters using Bayesian optimization. *Evolving Systems*, *12*. https://doi.org/10.1007/s12530-020-09345-2

[112] Wang, H., & Wen, D. (2014). *The progress of sub-pixel imaging methods. 9142*, 91420K. https://doi.org/10.1117/12.2054205

[113] Wang, X., Yu, K., Wu, S., Gu, J., Liu, Y., Dong, C., Loy, C. C., Qiao, Y., & Tang, X. (2018). ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks. *ArXiv:1809.00219 [Cs]*. http://arxiv.org/abs/1809.00219

[114] Wang, Z., Chen, J., & Hoi, S. C. H. (2020). Deep Learning for Image Super-resolution: A Survey. *ArXiv:1902.06068 [Cs]*. http://arxiv.org/abs/1902.06068

[115] W.-S. Lai, J.-B. Huang, N. Ahuja, and M.-H. Yang, (2017) Deep laplacian pyramid networks for fast and accurate superresolution. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[116] *Weight (Artificial Neural Network)*. (2019, May 17). DeepAI. https://deepai.org/machine-learning-glossary-and-terms/weight-artificial-neural-network.

[117] *Welcome to Python.org*. (n.d.). Python.Org. Retrieved 5 July 2021, from https://www.python.org/

[118] *Windows 10 Enterprise OS - Microsoft 365*. (n.d.). Retrieved 5 July 2021, from https://www.microsoft.com/es-es/microsoft-365/windows/windows-10-enterprise

[119] Xiong, Y., Guo, S., Chen, J., Deng, X., Sun, L., Zheng, X., & Xu, W. (2020). Improved SRGAN for Remote Sensing Image Super-Resolution Across Locations and Sensors. *Remote Sensing*, *12*(8). https://doi.org/10.3390/rs12081263

[120] Yang, W., Zhang, X., Tian, Y., Wang, W., & Xue, J.-H. (2019). Deep Learning for Single Image Super-Resolution: A Brief Review. *IEEE Transactions on Multimedia*, *21*(12), 3106–3121. https://doi.org/10.1109/TMM.2019.2919431

[121] Yaoyuan Liang. 2021. *Unsupervised Super Resolution Reconstruction of Traffic Surveillance Vehicle Images*. (2021) 13th International Conference on Machine Learning and Computing. Association for Computing Machinery, New York, NY, USA, 336–341. DOI: https://doi.org/10.1145/3457682.3457734

[122] Yi Wang, R. Fevig and R. R. Schultz, "Super-resolution mosaicking of UAV surveillance video," *2008 15th IEEE International Conference on Image Processing*, 2008, pp. 345-348, doi: 10.1109/ICIP.2008.4711762.H.

[123] Zamir, A. R., Wu, T.-L., Sun, L., Shen, W. B., Shi, B. E., Malik, J., & Savarese, S. (n.d.). *Feedback Networks*. 10.

[124] Zhang, H., Yang, Z., Zhang, L., & Shen, H. (2014). Super-Resolution Reconstruction for Multi-Angle Remote Sensing Images Considering Resolution Differences. *Remote Sensing*, *6*(1). https://doi.org/10.3390/rs6010637

[125] Zhang, Y., Tian, Y., Kong, Y., Zhong, B., & Fu, Y. (2018). Residual Dense Network for

Image Super-Resolution. *ArXiv:1802.08797 [Cs]*. http://arxiv.org/abs/1802.08797

[126] Zhang, Y., Li, K., Li, K., Wang, L., Zhong, B., & Fu, Y. (2018). Image Super-Resolution Using Very Deep Residual Channel Attention Networks. In V. Ferrari, M. Hebert, C. Sminchisescu, & Y. Weiss (Eds.), *Computer Vision – ECCV 2018* (Vol. 11211, pp. 294–310). Springer International Publishing. https://doi.org/10.1007/978-3-030-01234-2_18

[127] Zhang, Y., Li, K., Li, K., Zhong, B., & Fu, Y. (2019). Residual Non-local Attention Networks for Image Restoration. *ArXiv:1903.10082 [Cs]*. http://arxiv.org/abs/1903.10082

[128] Zhang, Z. (2018). Artificial Neural Network. In Z. Zhang, *Multivariate Time Series Analysis in Climate and Environmental Research* (pp. 1–35). Springer International Publishing. https://doi.org/10.1007/978-3-319-67340-0_1.