E.T.S. de Ingeniería
Industrial, Informática y de
Telecomunicación

Super Resolution based on GAN networks applied to gaze estimation

Máster Universitario en Ingeniería en Telecomunicación

Trabajo Fin de Máster

Author:
Álvaro Baquedano Simón
Supervisor:
Arantxa Villanueva Larre
Gonzalo Garde Lecumberri

July 2020

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa
campus iberus

Super Resolution based on GAN networks applied to gaze estimation

A Deep Learning application project

Author
Álvaro Baquedano Simón

Supervisor
Arantxa Villanueva Larre
Dpto. de Ingeniería Eléctrica, Electrónica y de Comunicación
Gonzalo Garde Lecumberri
Dpto. de Ingeniería Eléctrica, Electrónica y de Comunicación

Máster Universitario en Ingeniería en Telecomunicación

E.T.S. de Ingeniería
Industrial, Informática y de
Telecomunicación

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

campus iberus

Pamplona, July 2020

Preface

"My supervisor during the development of this project, Arantxa, has been my computer vision and image processing teacher during my masters degree. Due to my passion, interest and experience in computer vision and artificial intelligence, I asked her if there was any available work related to these topics I could do for my final project. She then offered me this project and I couldn't refuse. One of the strands of work being developed within the Gaze Interaction for Everybody (GI4E) group is a gaze estimation system. The aim of this project is primarily to research for further improvements in the tracking process by increasing eye image resolution artificially."

Acknowledgments

Firstly, I would like to thank Arantxa, Gonzalo and Andoni for their patience, kindness and unconditional help throughout the development of my project. Without them it wouldn't have been possible to carry it out.

Secondly, I would like to acknowledge all the university professors that took part in my academic training along my degree.

Last but not least, I would like to thank my family, partner and close friends that helped me in my toughest times.

To my grandparents,
who encouraged me to study this career and made me be the kind of person I am.

If I have seen further
it is by standing
on the shoulders of Giants

Isaac Newton.

Contents

List of Figures

List of Tables

List of Code snippets

1 Introduction

What makes human being special and different from any other living being is its reasoning ability. Our brain is the one responsible of this, allowing us to process information coming from our senses, store memories, express emotions and much more, which makes this particular organ the most important one. Without it, we would be nothing more than organisms whose actions would rely primarily on our stimulus and reflexes.

For decades, human being has always dreamed of creating machines capable of thinking and performing tasks only humans could do. This is mainly due to his desire of easing heavy and difficult workloads, performing repetitive tasks more efficiently or developing new tasks the human being is not capable of. However, building these machines comes with a very complex computational challenge a brain can solve in a few fractions of a second. The science aimed to solve these problems and define the techniques, methodology and processes required to face them belongs to the Artificial Intelligence (AI) field of study. Briega (2017)

1.1 Artificial Intelligence

In the past years, AI has grown at dizzying speed, mainly due to the huge development of latest technologies. The range of applications where we can find AI is getting wider every day and, nowadays, it is difficult to find a sector in which at least a minimal AI isn't being applied. Some applications where AI is used are:

- **Healthcare:** Such as the use of novel tools based on AI allowing doctors to improve the accuracy of their diagnoses.
- **Cybersecurity:** This includes improving current Intrusion Detection System (IDS) and malware detection systems used by antivirus and detecting spam and fishing emails.
- **Autonomous vehicles:** Self-driving cars need to process all the information given by sensors in order to make decisions about the actions to take.
- **Virtual assistants:** All the current virtual assistants, like Google Assistant, Siri, Alexa or Cortana use a combination of AI techniques in order to perform their tasks.
- **Finance:** Banks commonly use fraud detection systems based on AI which analyzes patterns in expenses in order to identify suspicious activities before they may occur. They also use virtual brokers in order to improve the effectiveness of their investments.

Artificial Intelligence is relatively new, as it was born approximately in 1956. It takes several influences from other fields, such as Mathematics, Psychology, Biology and even Philosophy.

Its term is actually difficult to define. Even Alan Turing, considered as the father of computer science, avoided answering that question directly. Instead, he created the well-known Turing Test in order to define whether a system had an artificial intelligence or not. In a nutshell, it can be defined as a sub-field of computing science whose aim is to make machines perform tasks only humans could do. Alonso (2011-2012)

As AI is a very general term, it can be divided into several work areas, such as Machine Learning (ML), Deep Learning (DL), Natural Language Processing (NLP), Robotic Process Automation, Computer Vision (CV) and many more. The one that will be covered in this report is the ML area, and more precisely the DL area where Neural Networks belong to.

1.2 Machine Learning

ML is commonly described as an AI technique used to program a system or application aimed at making decisions, classifying events or generating new information in a completely autonomous manner based on some input elements. For example, it could be possible to program an application which identifies a plane in a given image of a plane. However, this will only be valid for that specific image.

ML applications, however, would be able to correctly identify a plane in any given image, as they are able to train themselves in order to learn how a plane looks out of a big dataset containing plane images. Moreno (2016)

The training algorithms commonly used in ML can be classified in four types: Fumo (2017)

- **Supervised:** In this method, the computer is fed with training data containing the input along with their corresponding labels. Human interaction is required, as he shall act as the teacher showing the correct answers to the model, which will learn comparing them with its predictions.
- **Unsupervised:** The computer is trained with no labeled data and there's no human interaction at all. Here, the model tries to find patterns in data. It is very useful in cases where the human doesn't know what to look for in the data.

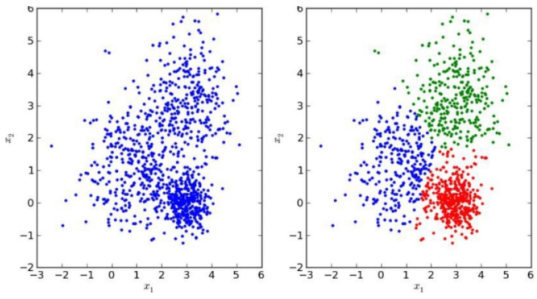


Figure 1.1: Unsupervised Learning Patterns. Source: deepai.org

- **Semi-supervised:** A mixture of labeled and unlabeled data is fed to the model. They

are mainly used in situations where the cost of labeling all the data is too high.

- **Reinforcement Learning:** The computer learns out of the observations gathered from the interaction with the environment, trying to minimize the fails and maximize the rewards. The algorithm (also called agent) tries to determine the ideal behavior within a specific context, in order to improve its performance. The agent learns its behavior by using simple reward feedback taken from the environment. These algorithms are commonly used in Deep Adversarial Networs (DAN), which where used as a basis for this work. They will be described with more detail later within this document.

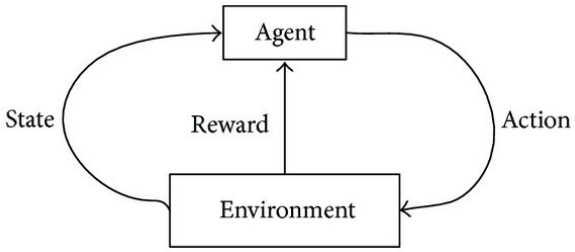


Figure 1.2: Reinforcement Learning block diagram. Fumo (2017)

Some of the most well-known models can be found within the supervised models field. They can be divided into two subcategories depending on their task (Regression or Classification) and type of output (Discrete or Continuous). The most relevant are: Fumo (2017)

- Regression.
- Linear regression.
- Decision tree.
- Random forest.
- Neural Networks.
- Logistic Regression.
- Support Vector Machine.

Some of them, such as Random Forest, Decision tree or Neural Network can have both discrete and continuous outputs. Fumo (2017)

The theme of this paper revolves around the application of discrete neural networks to image super-resolution generation based on DAN usage.

It is divided into several sections. Firstly, a comprehensive explanation of these concepts, their theoretical behavior and state of the art will be described. Secondly, the main objectives of this work will be presented, as well as the methodology used during its development. Then,

the steps followed in order to achieve the given objectives, along with its difficulties, improvements and successes will be detailed. Finally, the results obtained from our application will be shown along with our own conclusions about them.

2 Theoretical Framework

In this section, an extended description of the theoretical basis of this work will be presented. Firstly, a wide description of deep feed forward and convolutional neural networks fundamentals will be covered. Secondly, GAN fundamentals will be explained. Finally, the state of the art in neural network application in super resolution will be described, as well as the study on which this work is based.

2.1 Neural Networks

Although many details of how the brain processes information are still unknown, several models have been developed in an attempt to mimic its abilities. These models are called Artificial Neural Network (ANN) and they are one of the most used ML models in multiple AI fields.

A neural network is composed of several elements which are worth describing.

2.1.1 Neuron

The most basic element within an ANN is called *neuron*. Each neuron receives a set of x-values representing the features taken from one example of the training set. Besides, each value comes with its own parameters, called W(list of weights) and b(bias), which will change during the training process. The neuron will calculate a weighted sum of the input values by using the current values stored in W and adding the bias b to it. The result will be passed through a non-linear activation function. The schematic behavior of a neuron is shown in figure 2.1. The weighted average is shown in the formula 2.1. Skalski (2018)

$$z = w_1 x_1 + w_2 x_2 + ... + w_n x_n = w^\tau x \tag{2.1}$$

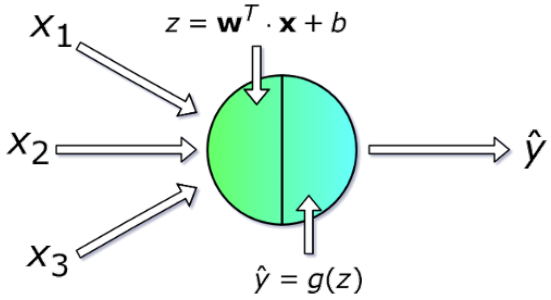


Figure 2.1: Single neuron schematic. Skalski (2018)

2.1.2 Layer

Just as our brain is made up of interconnected neurons, an ANN is made up of artificial neurons connected to each other and grouped at different levels that we call *layers*. These layers are named differently depending on their location within a network. The first layer, the one receiving the features as inputs, is called *input layer*, the last layer of the network which gives the final output of the network is called *output layer* and the ones laying between these two are called *hidden layers*, as their input and output values remain unknown.

The computations performed in each layer are nothing but the ones performed by each neuron in a parallel fashion. The formulas are shown in the equation 2.2. The schematic behavior of a single layer is shown in figure 2.2.

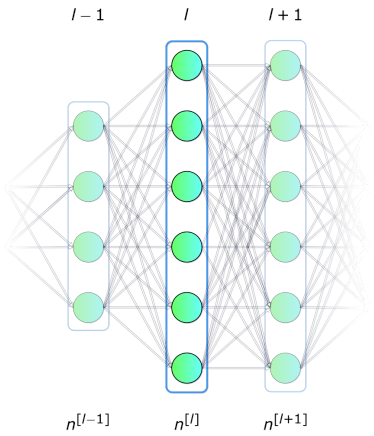


Figure 2.2: Single layer schematic. Skalski (2018)

$$z_i^{[l]} = w_i^{\tau} \cdot a^{[l-1]} + b_i \qquad a_i^{[l]} = g^{[l]}(z_i^{[l]}) \tag{2.2}$$

where: l → index of each layer.
i → index of neuron.
g → Activation function.
$a^{[l]}$ → activation for each layer l.

To sum up the above equations, the w, a, and z vectors of each neuron from a given layer will be grouped into matrices representing the values of an entire layer, named X, A and Z. The equation 2.2 will be rewritten into equation 2.3. Skalski (2018)

$$Z^{[l]} = W^{\tau} \cdot A^{[l-1]} + b^{[l]} \qquad A^{[l]} = g^{[l]}(Z^{[l]}) \tag{2.3}$$

2.1.3 Activation Functions

The activation function is a really important element of a neural network. Without it, its operations would remain a combination of linear functions, so it would be just a linear function itself. The usage of non-linear elements allows the network to get greater flexibility

and creation of complex functions during the learning process. The most common one is the Rectifier Linear Unit (ReLU) and all its variants, as it is widely used in hidden layers. For the output layer, the sigmoid function is the most popular, especially in binary classification, hence an output ranging between 0 and 1 is desired. The most common are shown in figure 2.3. Skalski (2018)

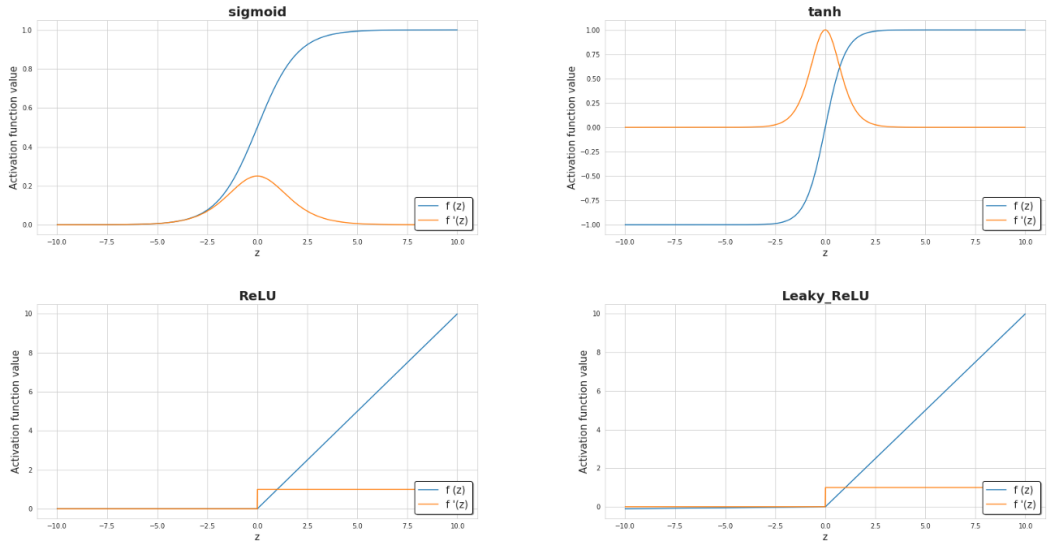


Figure 2.3: 4 common activation functions. Skalski (2018)

2.1.4 Loss Function

This function is used to help the network optimize its parameters during the training process. The main objective is to minimize the loss for a neural network by matching the target weights. This loss is computed by matching the actual value with the predicted value generated by the network. Generally speaking, the loss function is designed to show how far the network is from the actual solution. The most popular are: Skalski (2018)

- **MSE:** It is calculated by computing the mean of squared differences between the actual and the predicted value. It is commonly used when the network outputs a real value.
- **Binary Cross-entropy:** It is commonly used for binary classification tasks, where the network outputs a value ranged between 0 and 1 given by a sigmoid activation function.
- **Categorical Cross-entropy:** Used in multi-class classification tasks. The network outputs the same number of nodes as the classes. Each output will have a probabilistic format between 0 and 1 given by a softmax activation function.

2.1.5 Learning algorithm

As aforementioned, a neural network learning process is based on W and b parameters optimization with the aim of minimizing the loss function. This is achieved by executing a task

called *gradient descent method*, which enables the network to find a function minimum.

The math behind this process is based on calculating the values of the loss function partial derivatives with respect to each parameter of the neural network. As computing the partial derivative allows to obtain the slope of each function, it is possible to manipulate variables in order to move downhill in the loss graph. As it can be seen in the figure 2.4, the gradient is computed on each iteration, showing the network the direction in which it should move in order to minimize the loss. These computations seem fairly simple, however, in ANN, as we dispose of many parameters to change, it can be very complex. This is why it is commonly used an algorithm called *backpropagation*, which enables us to compute very complex gradients.

The parameters of the neural network are adjusted following the formula 2.4. α represents the learning rate, which is a parameter allowing to control how fast or slow the neural network will update its parameters. It will be explained later.

$$W^{[l]} = W^{[l]} - \alpha dW^{[l]} \tag{2.4}$$

$$b^{[l]} = b^{[l]} - \alpha db^{[l]} \tag{2.5}$$

These last parameters, dW and db, are computed following the chain rule with the partial derivatives of the loss function with respect to W and b, as shown in the equation 2.6. The *backpropagation* process is shown in figure 2.4. Skalski (2018)

$$dW^{[l]} = \frac{\partial L}{\partial W^{[l]}} \tag{2.6}$$

$$db^{[l]} = \frac{\partial L}{\partial b^{[l]}} \tag{2.7}$$

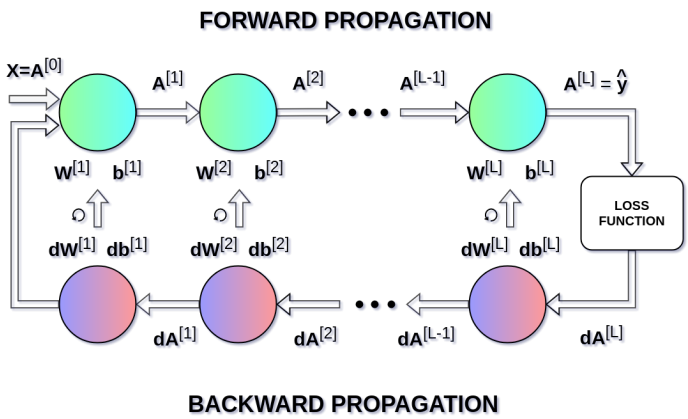


Figure 2.4: Forward and backward propagation. Skalski (2018)

2.1.6 Learning rate

In DFF neural networks trained using the stochastic gradient descent algorithm, the pace at which the weights are updated during training is referred as the step size or *learning rate*. This variable can be tuned as a configurable hyper-parameter used in the training process.

The learning rate controls how quickly a DFF model adapts to a certain problem. Smaller learning rates require more training epochs, as the weight changes between each update are smaller. On the other hand, larger learning rates result in fast changes and require less training epochs.

One of the challenges of training DFF neural networks involves the correct selection of the learning rate value, as a value that is too large can cause the model to converge too quickly, reaching a local minima and, therefore, a sub-optimal solution, whereas a learning rate that is too small can lead to a training process unable to converge. These effect are shown graphically in the figure 2.5. Brownlee (2019)

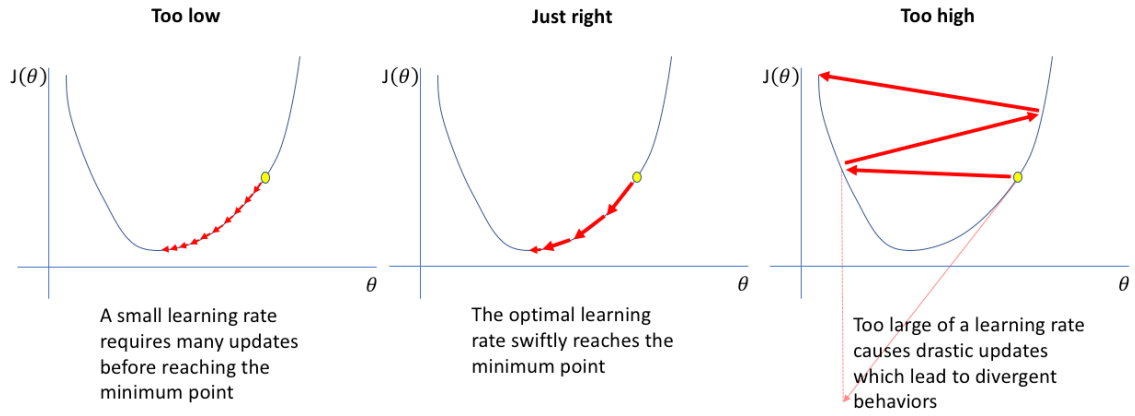


Figure 2.5: How the learning rate affects the model training convergence. Jordan (2018)

2.1.7 Types of Neural Networks

In ML field there exist a huge set of ANN architectures and models. They can be classified depending on multiple factors, such as their architecture, computations done by the neurons, type of layers, applications, etc. However, the most popular ones are: Tch (2017)

- **DFF:** These are nothing more than the model described above. Its neurons are commonly fully connected and their application is mainly for classification purposes based on an input. Even though they are quite impractical nowadays, they form the core of modern ML systems. As explained before, they compute linear operations.

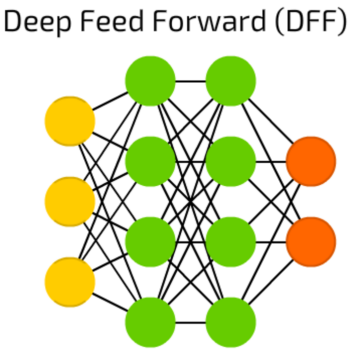


Figure 2.6: DFF architecture. Yellow dots mean input cells, green stand for hidden cells and orange ones represent output cells. Tch (2017)

- **RNN:** A new neuron concept is introduced: Recurrent cells. They allow a regular DFF to keep some of its recurrent cells in the same state for a fixed delay of iterations within the training process. This allows the network to train itself using parameters from previous iterations. This way, the past decisions have an influence on the new ones. They are commonly used in NLP, as a word can be analyzed only considering the context of previous words or sentences.

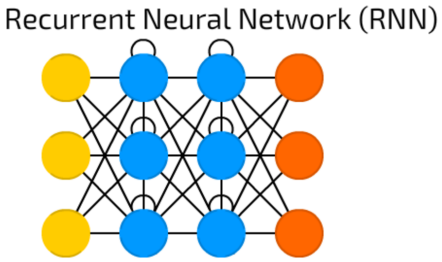


Figure 2.7: RNN architecture. Yellow dots mean input cells, blues stand for recurrent cells and orange ones represent output cells. Tch (2017)

- **CNN:** Nowadays, they have become the most valuable models of ANN. They are commonly used for image processing applications, such as image recognition. Instead of computing linear operations, their layers compute convolutions of the input images using kernels and filters. They will be explained with more detail later in this report.

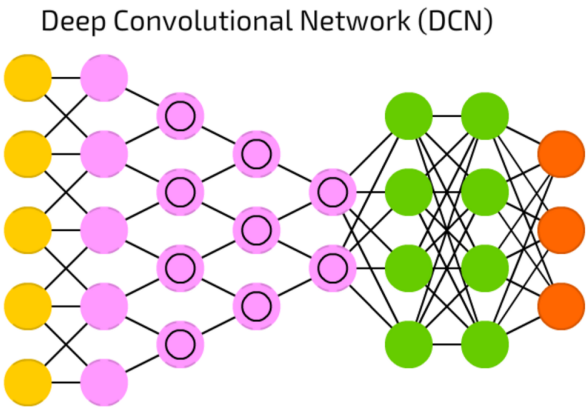


Figure 2.8: CNN architecture. Yellow dots mean input cells, pinks stand for convolution, pool or kernel cells and orange ones represent output cells. A DFF is attached to the end, as image recognition always includes one after the convolution stage. Tch (2017)

2.2 Convolutional Neural Networks

Computer vision means AI applied to image processing and recognition, that is, its main objective is to enable a machine to see its environment and react differently to different scenarios. In the last years, there has been a huge advancement in Deep Learning for Computer Vision applications, mainly thanks to great improvements in parallel computing using increasingly powerful GPUs, as the calculations done by neural networks can be arranged in matrix operations, as we learned from the previous section, and GPU are really good at performing these type of calculations. Image recognition and generation are the most widespread CV applications in which DL is used. Within this field, one of the most remarkable advancements has been, without no doubt, the CNN or ConvNets.

A ConvNet is a type of DL algorithm that, instead of computing linear operations from the input data as a regular DFF network does, it computes convolutions with different filters. That is, it is able to take an image as input and assign weights and biases to different aspects inside the image so that it is able to differentiate one from each other. This makes a CNN much more suitable for image processing than just turning a 4x4 matrix into a 16x1 vector and feeding it to a DFF neural network, as it is able to successfully capture the spatial and temporal dependencies in an image through the input image convolution with relevant filters.

Moreover, as images can be really heavy data containing a lot of information stored in multiple color spaces (RGB, HSV, Grayscale, etc.), a ConvNet can help in reducing them into a simpler form, making them easier to process by a machine and without losing critical features that differentiate them. Saha (2018)

2.2.1 Convolution Layer

As mentioned above, a ConvNet uses different filters, also called Kernels or K-matrices, which constitute the elements involved in carrying out the convolution operations. Each layer can

perform multiple convolutions as it can contain different filters of $NxNxC$ size, each of one seeking for different features inside the image. The C variable means the number of channels, as the Kernel has the same depth as of the input image.

A single convolution process is illustrated in the figures 2.9 and 2.10, where it can be seen the kernel movement along the image from left to right and from the top to the bottom, as well as the operations carried out by a kernel of $3x3x3$ size. Giving a clear example of a real-life scenario, a convolution layer holding 64 different filters of 3x3 size, when an input image of $5x5$ size is fed into that layer, the output will be of $3x3x64$, that is, 64 output matrices of $3x3$ size, each of one containing different features extracted from the image by each one of the 64 filters. Saha (2018)

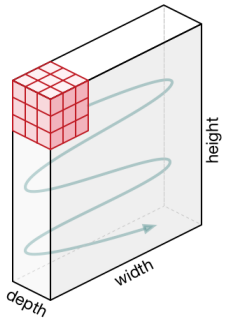


Figure 2.9: Movement trajectory of the Kernel. Saha (2018)

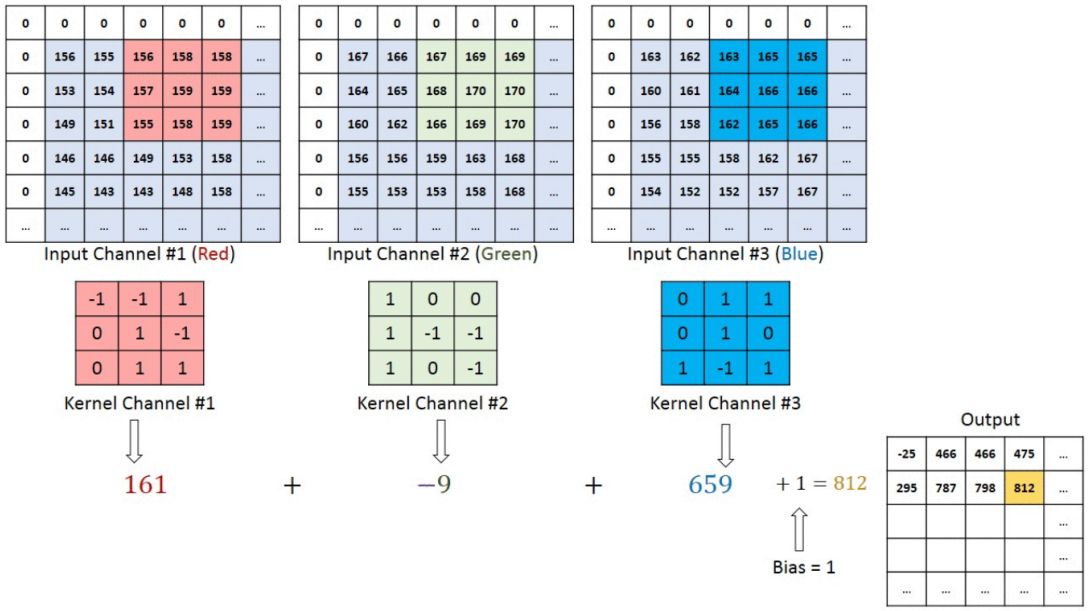


Figure 2.10: Convolution operation on a $MxNx3$ image matrix with a $3x3x3$ Kernel. Saha (2018)

In a nutshell, the main objective of the convolution operation is to extract high-level features from the input image. In order to increase its effectiveness, ConvNets need to be composed of multiple convolution layers, where the first one is responsible for extracting low-level features such as edges, colors, etc., and the next layers added to the model will extract increasingly higher-level features, giving as a result a complete feature map of every image fed into the network. Saha (2018)

2.2.2 Pooling Layer

The Pooling Layers perform a similar task as the Convolution Layers and are commonly located just after the latter with the aim of reducing their output dimensions. This process gives two main advantages. In one hand, the computational power required to process the data is effectively reduced and, in the other hand, it is very useful for extracting dominant features within an image, making the training process easier for the model. Saha (2018)

There exist two types of Pooling operations:

- **Max Pooling:** It returns the maximum value from an image portion covered by the pooling kernel. This is the most popular one, as it also performs as a noise suppressant ignoring the noisy activations.
- **Average Pooling:** It returns the average of all the values within the image portion.

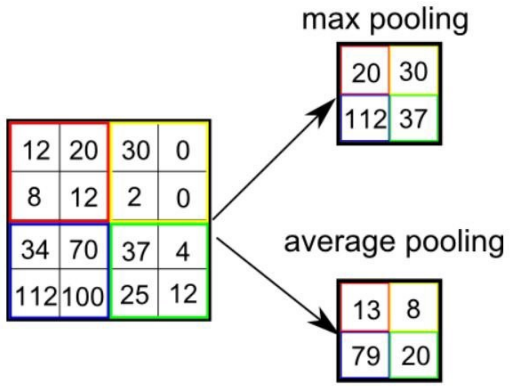


Figure 2.11: Types of Pooling. Saha (2018)

2.2.3 Classification Stage

Given an image recognition scenario, there is a need to reduce all the features obtained from the convolution layers into an output layer whose results match the possible outcomes' dimensions. In order to achieve that, it is very common to combine the output features from the last convolution layer with a DFF fully connected neural network by means of a Flatten Layer, allowing the CNN to introduce every single value from each feature matrix into every neuron of the input layer belonging to the DFF network. Saha (2018)

Finally, the output values of this last stage can be binary (only one output) or categorical (multiple outputs matching the number of possible classes), and its corresponding loss function will be used in order to train the whole network, as explained in the section 2.1.4. A full schematic of a combination between a CNN and DFF can be seen in the figure 2.12.

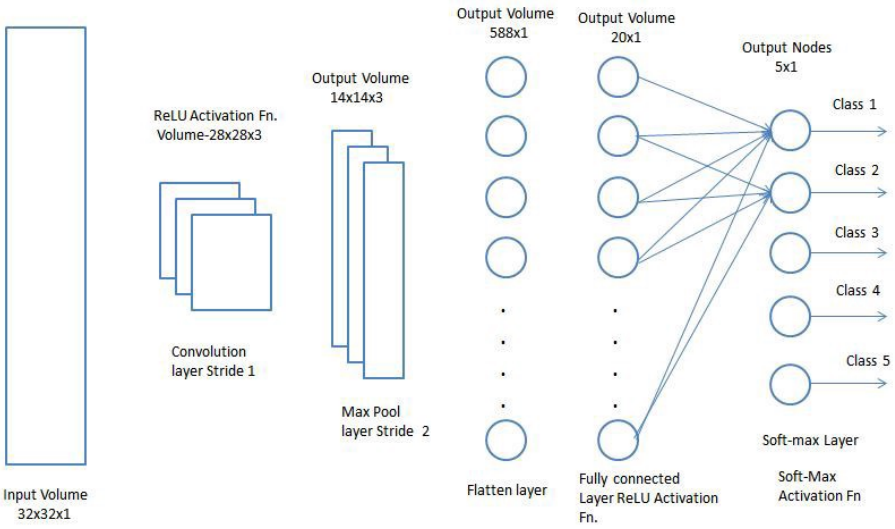


Figure 2.12: Combination of a CNN with a DFF. Saha (2018)

2.2.4 Types of CNN

Throughout the latest years, a huge amount of different CNN architectures have been implemented for multiple purposes, mostly for general image classification. These models have established the basis for any application using CNNs and nowadays are a reference for future research in the field of image classification and generation. Indeed, most of them have won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), an object recognition contest where dozens of institutions and research groups test their own CNN implementations. The most relevant are: Saha (2018)

- LeNet.
- AlexNet.
- VGGNet.
- GoogLeNet.
- ResNet.
- ZFNet.

This work will use both a ResNet and a VGGNet. The first one for image generation and the second one for feature extraction. These architectures will be explained below.

2.2.4.1 VGGNet

VGGNet is one of the most used and best performing CNN architectures for image recognition. It was introduced by Simonyan and Zisserman in their work Simonyan & Zisserman (2015). This is a simple network only using 3x3 kernel-size convolution layers stacked in increasing depth, it uses max pooling for data reduction and two fully-connected of 4096 nodes at the end of the network, in the classification stage. A visualization of the architecture is presented in the figure 2.13.

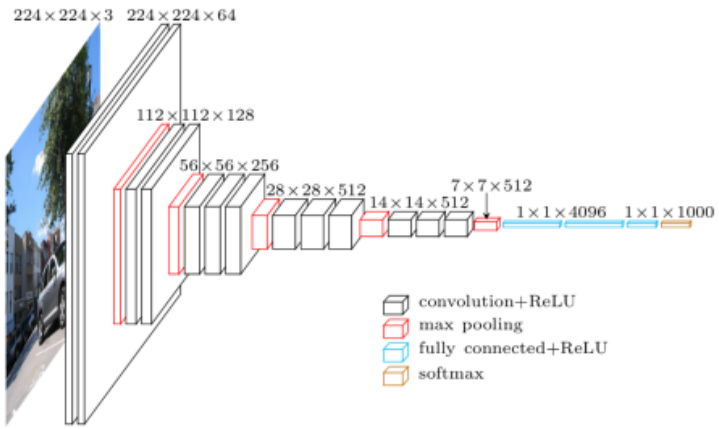


Figure 2.13: Schematic of the VGG architecture. Frossard (2016)

It is commonly called VGG16 or VGG19, where the number stands for the number of weight layers present in the network, as it can be seen in the table 2.14.

In spite of its performance and popularity, this architecture has two major drawbacks: It is difficult and slow to train and it is very heavy in terms of disk space. Mainly due to its fully connected nodes, it can be up to 574MB. Rosebrock (2017)

2.2.4.2 ResNet

This architecture is quite special, as it doesn't follow a traditional sequential architecture as the others do. Instead, it is made out of a collection of micro-architecture building blocks containing their own convolution, pooling and other layers which are then added to the main architecture. Its schematic can be seen in the figure 2.15.

It was introduced by He et al. in their work Deep Residual Learning for Image Recognition (Kaiming He (2015)) published in 2015. It has become one of the most important advancements in image-recognition using CNNs. The most popular architecture is the ResNet50, which is 50 layers deep, and it is an improvement over the VGGNet as it is easier to train thanks to its residual architecture. In spite it is much deeper than VGG19 model, it is actually lighter in terms of disk space as it uses global average pooling rather than fully-connected layers. Rosebrock (2017)

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 2.14: Configurations of VGGNets. Parameters are denoted as conv<kernel-size>-<number of filters>. Simonyan & Zisserman (2015)

2.3 Generative Adversarial Networks

So far, this work has explained how neural networks are applied to image recognition and classification purposes with supervised techniques, which is called *Discriminative Modeling*. In fact, these are the most popular applications of DL to real-life problem solving. However, a neural network can also be used to generate data, rather than identifying its contents. These networks are called *Generative Models*, which can be used to create a wide variety of data, such as text or images, and can be trained in an unsupervised manner. The main difference between these two lies in the problem they aim to solve. In one hand, discriminative models aim to solve the relationship between the features x and the labels y, that is, find the probability of y given x. On the other hand, generative models care about how to get x given y. Nicholson (2018)

The use of generative models to create artificial images has increased in the last few years. However, the methods used to train these kind of networks have remained in the direct way, that is, a pixel-wise difference between the generated and the ground truth image is performed. Although these methods have proven to be effective, generally there is a lack of detail in the generated images that clearly distances them from a real-life one.

However, everything changed when Ian Goodfellow and other researchers from the Uni-

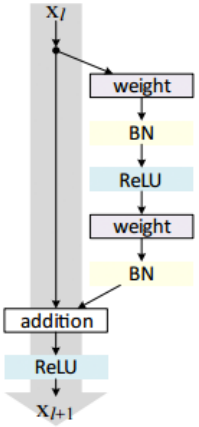


Figure 2.15: A residual module in the ResNet. Rosebrock (2017)

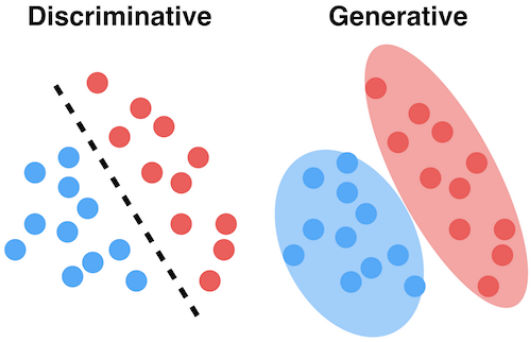


Figure 2.16: Graphical difference of how Discriminative and Generative networks work. Malhotra (2019)

versity of Montreal, in 2014, introduced the Generative Adversarial Networks (Goodfellow et al. (2014)), which where referred by Facebook's AI research director Yann LeCun as "the most interesting idea in the last 10 years in ML". These networks have a great ability to mimic any distribution of data in any domain, and it is the technology behind the well-known Deepfakes. Nicholson (2018) Ledig et al. (2017)

2.3.1 Fundamentals

A GAN network is composed by two neural networks, a generative and a discriminative model. These two compete against each other in a counterfeit identification contest in which the generator acts as the counterfeiter and the discriminator acts as the detective. The generator produces images with the aim of tricking the discriminator into believing that they are real. Meanwhile, the discriminator aims to detect the counterfeits.

As time passes by, the images produced by the generator become increasingly realistic

and difficult to identify as counterfeits and, at the same time, the discriminator improves its ability to identify them. This interaction between these two networks goes on until a point is reached where the discriminator cannot distinguish the counterfeits from the real ones. As a result, a great realistic image generator is obtained.

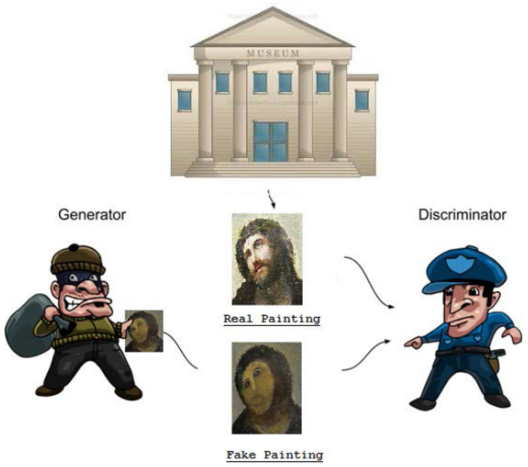


Figure 2.17: A common way of explaining GANs is by using the art counterfeiter - detective example.

2.3.2 Mathematical Model of a GAN

As described above, a GAN is composed of two neural network models denoted as Discriminator and Generator. The generator generates data out of a given input which commonly consists of random data or noise, though it can be anything else. On the other hand, the discriminator checks whether the data generated by the generator matches in a certain way the real data, returning a probability value between 0 and 1, where 0 means absolutely fake and 1 absolutely real. A common architecture of a GAN network can be seen in the figure 2.18.

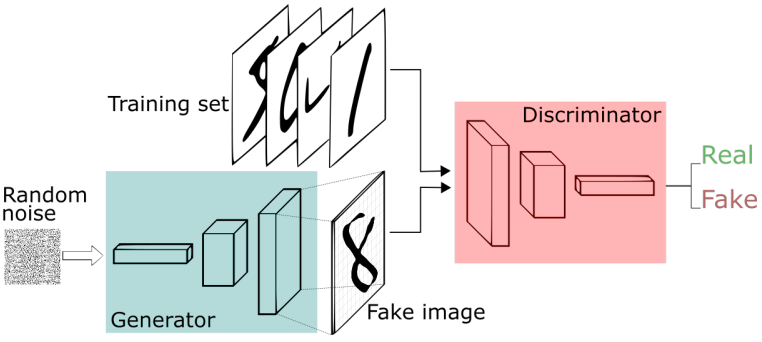


Figure 2.18: Schematic of a GAN architecture. Nicholson (2018)

Considering x as the real data input and z as the noise input, $G(z, \theta_1)$ and $D(x, \theta_2)$ represent the generator and discriminator models, where the first one's role is mapping input

noise variables z to the target ground truth x, and the second one outputs the probability that its input data actually was taken from the real dataset. θ_i represents the weights of both models. Mosquera (2018)

In one hand, the discriminator is trained to discern fake from real data, that is, its aim is to maximize the function $D(x)$ while minimizing $D(G(z))$. On the other hand, the generator is trained to fool the discriminator, hence it tries to maximize the probability that a fake data sample is classified as real, defined as $D(G(z))$. This competition to optimize opposite loss functions is called the *minimax game* with value function $V(G, D)$. It can be seen in the equation 2.8. Goodfellow et al. (2014)

$$\underset{G}{min}\underset{D}{max}V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{2.8}$$

where: $p_z(z)$ $\rightarrow$ Probability distribution of noise input data.
$p_{data}(x)$ $\rightarrow$ Probability distribution of real input data.

Finally, since both models are based on neural networks, it is possible to use a back-propagation gradient-based algorithm in order to train the GAN. This was previously explained in the section 2.1.5.

3 State of the art

SR isn't new. Reconstructing a high resolution photo-realistic image from its low resolution version has always been a classic problem in the CV field.

SR can be defined as a set of techniques aiming to estimate a high-resolution (HR) image out of its low-resolution (LR) counterpart. This can be a really difficult task to achieve, as in most practical applications all we have is a low resolution image as input, so that its high-resolution version will be predicted.

Since the beginning of SR, several techniques and algorithms have been developed in order to achieve a photo-realistic super-resolved image out of a low-resolution one. From the first to the last advancement in this field, researchers have always sought to achieve even greater results in terms of image quality and realism by improving high frequency detail preservation from an image Huang et al. (2017) Sajjadi et al. (2017).

Literature shows that the process does not only improve the perceptual quality of the image but it can contribute to recover information that is relevant for different computer vision tasks. In the last few years deep learning based SR models have gained momentum. A bunch of deep-learning models has been proposed to approach SR. The differences among the methods employed are related to factors such as the architecture, the learning methods or the loss function, among others Kim et al. (2016) Anwar et al. (2020) Wang et al. (2020).

The most remarkable technologies will be briefly described.

3.1 Interpolation

Interpolation is the earliest solution and also the most basic one. It consists in resizing a LR image by a certain factor, such as 2x or 4x using a well-known interpolation algorithms like bicubic, bilinear or nearest-neighbor. Sahu (2019)

3.2 SRCNN

With the great success of CNNs at that time, researchers came up with a SR implementation by using a Fully Convolutional Neural Network (FCNN), that is, a regular CNN without a fully-connected DFF neural network attached at the end of it, in order to predict an output map given an input image. In a nutshell, FCNN is an image-to-image mapping engine, and SRCNN is the first implementation of this technique. An example of this model is illustrated in figure 3.2. Sahu (2019)

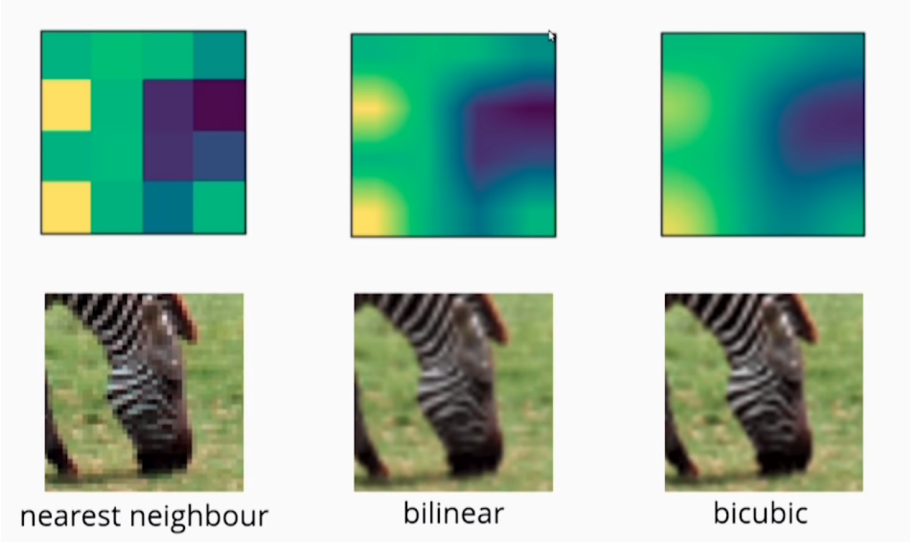


Figure 3.1: Results of the three different interpolation algorithms. Kańska (2019)

As it can be seen, the first step is to perform a regular interpolation, which will result in a HR version of the LR input. Then, the resulting image will be passed through a FCNN in order to further improve the quality of the resulting image.

3.3 SRResNet

This technique is based in the use of a ResNet model (which was already explained in the section 2.2.4.2), in order to perform the SR task. Here, the convolution layers are replaced by residual blocks, which can greatly improve model accuracy. Furthermore, the addition of sub-pixel convolution layers during the up-sampling phase rather than performing interpolation operations improve the model speed and accuracy, as these layers are learnable by the model. Sahu (2019)

3.4 Perceptual Loss

Using a SRResNet is an effective way to achieve a reasonably good-quality super-resolved image. In fact, this method will be used in this work as a warm-up for the generator model.

However, its performance is limited if a MSE loss function is used, which is very common in SR applications. This method is computed pixel-wise, that is, it computes the distance between two corresponding pixels in the generated and the ground truth. Minimizing this difference results in a lack of high-frequency texture details which leads to a overly-smoothed perceptually not convincing images.

On the other hand, using perceptual loss greatly improves SR image quality as it is based in the direct comparison between high-level feature representations of the generated and ground truth images. In order to extract those features, a certain inner layer from a pre-trained CNN

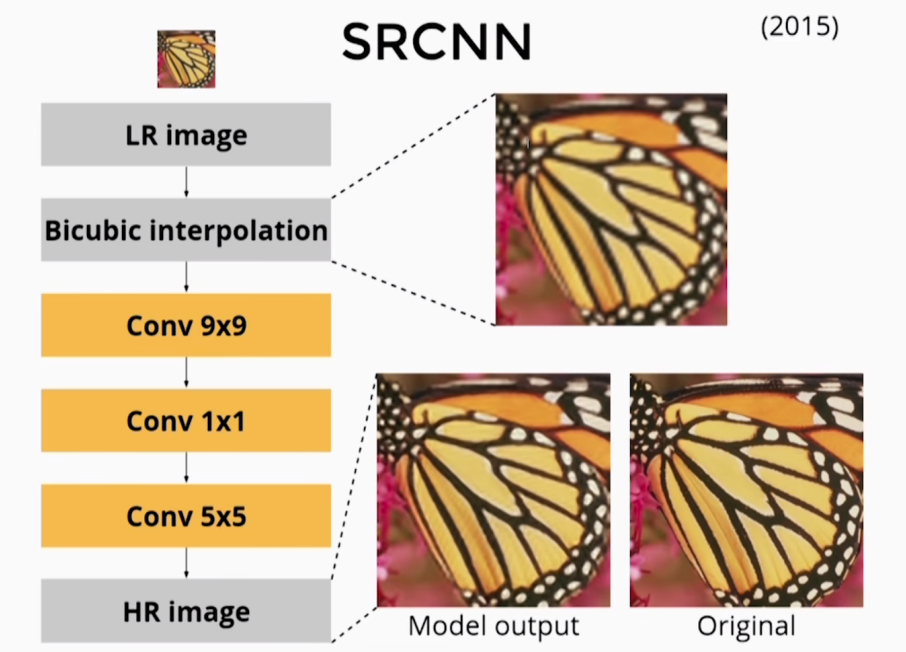


Figure 3.2: A SRCNN model example. Kańska (2019)

model is used as shown in figure 3.4. In summary, both images are passed through a pre-trained network, which is usually a ResNet or a VGGnet, and then the Euclidean distance between both feature maps is computed.

Eventually, this function sums all the squared errors and averages them, instead of summing the absolute error between pixels as MSE does. Sahu (2019)

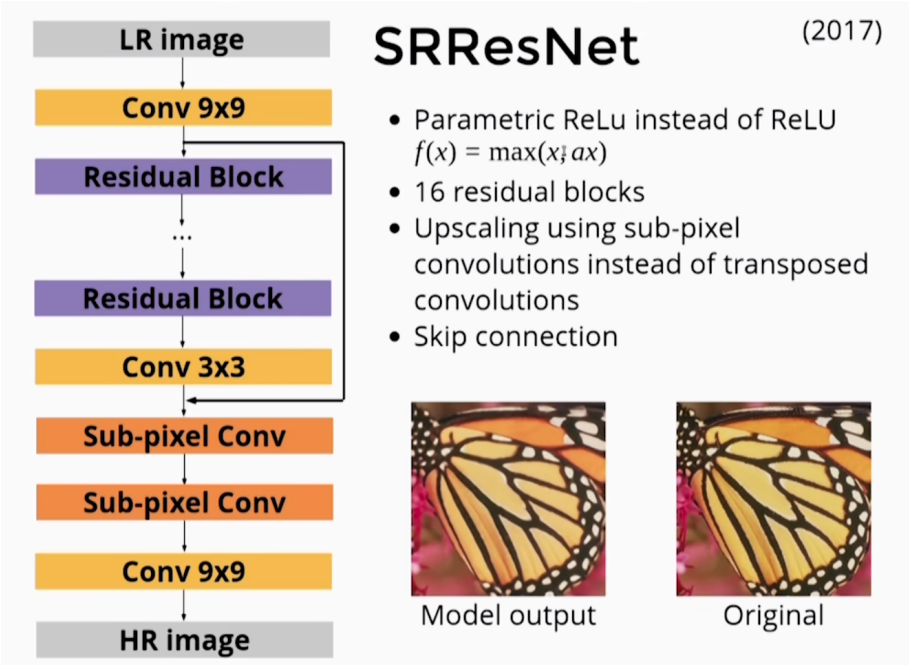


Figure 3.3: A SRResNet model example. Kańska (2019)

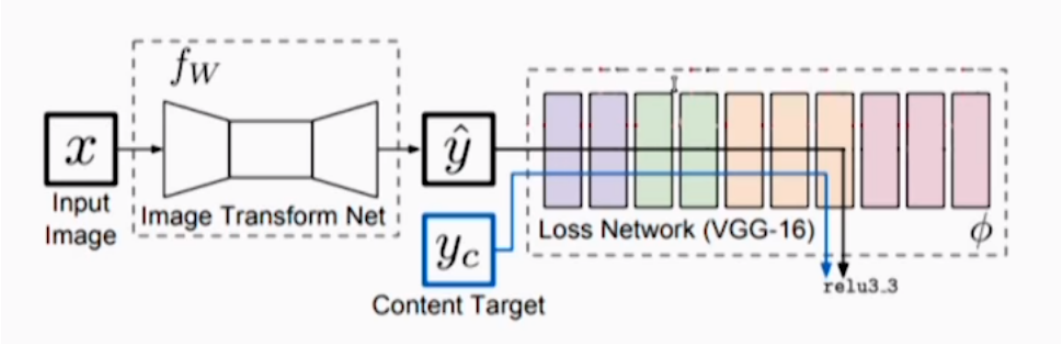


Figure 3.4: An inner pooling layer is extracted in order to get the features from the generated image.

4 Objectives

As mentioned in the preface, one of the lines of work of the group I worked with, GI4E, is the development of a gaze estimation system.

In the literature, there exist a great number of gaze estimation techniques as it is a technology in high demand nowadays. Many of the presented solutions are able to achieve an accuracy of around half a degree. However, these techniques tend to require very sophisticated and specific vision hardware, such as cameras constantly focusing the eye center, special lightning, ultra-high resolution IR cameras, and so on. This comes with a considerable cost with it, which in many cases can be inconvenient.

In contrast, as the name of the group itself stands, their technique aims to make gaze interaction more affordable by developing low resolution techniques in order to make them feasible to be used with images taken from a regular webcam.

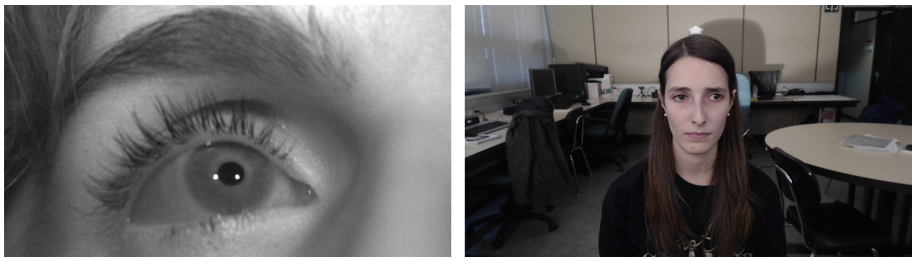


Figure 4.1: Comparison between high resolution images taken with an IR camera and low resolution images taken with a webcam.

With today's advances in AI and computational performance, it is now more possible than ever to achieve high-resolution-like results that would previously only be possible to get with a (commonly expensive) hardware upgrade, by means of software-based solutions.

This project is a clear example of this, as its aim is to improve the existing tracking precision using low resolution techniques by artificially increasing image resolution, which may enable the possibility to apply more reliable high resolution techniques, all of this without changing the image source.

5 Methodology

This chapter details the hardware and software tools used during the development of this project. Besides, the methodology followed throughout the project will also be presented.

5.1 Hardware Tools

The computer used for installing the software tools, developing the code, executing the training sessions, extracting the results and writing the report consists of a workstation holding the following specifications:

- **Central Processing Unit (CPU)**: Intel(R) Core(TM) i9-9900K @ 3.60GHz
- **Motherboard (MB)**: MSI Z370 PC PRO
- **Random Access Memory (RAM)**: 2x8GB 3200MHz memories in Dual-channel configuration.
- **Solid State Drive (SSD)**: Samsung EVO 970 1TB.
- **GPU**: NVIDIA GeForce RTX 2080.

As stated in section 2.2, 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 specs will be listed in the table 5.1.

NVIDIA GeForce RTX 2080 Specs	
Driver Version	445.87 (Windows) / 440.82 (Ubuntu)
CUDA Cores	2944
Tensor Cores	368
Memory Throughput	14.0 Gbps
Memory interface bus	256-bits
Memory bandwidth	448.00 GB/s
Core base clock	1815 MHz
Dedicated memory	8192 MB GDDR6

Table 5.1: Full specifications for the GPU used in this project.

5.2 Software Tools

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

5.2.1 OS

Two OSs have been used throughout the development of this project:

- **Windows 10 Pro 1909**: It is the most popular OS worldwide. It is simple, stable and really easy to use. It has been used as a starting point for our work.
- **Ubuntu 18.04.4 LTS**: Although all software tools are available for Windows systems, most of them have been developed with Linux systems in mind, so several tools which are only available on Linux. This is the reason why Ubuntu was also used.

5.2.2 Python

Python 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, along with Matlab, is one of the most popular programming languages in the scientific and research field, mostly due to its comprehensive standard library and ease to fast-prototyping. In fact, it is the most popular language in the AI and DL fields.

Throughout the project, several Python versions have been tested. However, the most used are the 3.6.9 version for Ubuntu OS and the 3.7 version for Windows.

5.2.3 TensorFlow

TensorFlow is a free and 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 2.0.0 version in October 2019, TensorFlow has become simpler and easier to use.

As with Python, for testing purposes several nightly versions have been used throughout the project. However, the most stable release used was 2.1.0 and 2.2.0 releases.

5.2.4 Keras

Keras 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 own core library since 2017 in order to make even easier to design, develop and deploy ML applications by combining the easy graph capabilities from Keras and the power and performance of TensorFlow.

5.2.5 TensorBoard

TensorBoard provides the visualization and tooling needed for machine learning experimentation:

- Tracking and visualizing metrics such as loss and accuracy
- Visualizing the model graph (ops and layers)
- Viewing histograms of weights, biases, or other tensors as they change over time
- Projecting embeddings to a lower dimensional spacey
- Displaying images, text, and audio data
- Profiling TensorFlow programs

5.2.6 CUDA

As mentioned above, GPUs can be used to compute intensive mathematical operations, such as matrix multiplication and convolution, 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.

TensorFlow makes use of this library for accelerating DL model training and inference. The version that has been used throughout the project is 10.1 version, which is compatible with both TensorFlow 2.1.0 and 2.2.0 versions.

5.2.7 MATLAB

MATLAB (Matrix Laboratory) 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.

5.2.8 PyCharm

PyCharm is an IDE specifically designed for the Python language. It is developed by the company JetBrains. It provides code analysis, debugger, and integration with version control systems such as Git.

PyCharm has been used for developing the project in Windows OS. In Ubuntu it has been discarded due to the need of getting a paid version in order to develop inside containers.

5.2.9 VS Code

Visual Studio Code is a source code editor developed by Microsoft for Windows, Linux and macOS. It includes support for debugging, integrated Git control, syntax highlighting, intelligent code completion, snippets, and so on.

Although it is relatively new, it has become very popular and very well accepted within the developer community due to its clean interface and comprehensive list of packages and plugins which highly improve the coding experience.

VS Code has been used in Ubuntu along with the Python and Remote Development plugins. The latter allows to develop inside a container with all the VS Code features, just as if it where the host machine. The container concept will be detailed in the Docker section.

5.2.10 Docker

Installing TensorFlow can be confusing, especially if you also need to install the CUDA libraries for GPU support.
Docker is the easiest way to enable TensorFlow GPU support on Linux since only the NVIDIA GPU driver is required on the host machine. That is, there is no need to install neither CUDA or TensorFlow inside the host.

Docker is an open source project that automates the deployment of applications within software containers. These containers can be seen as virtual machines isolated from the host OS that contain all the environment, configuration and dependencies needed to run a given application on any operating system, without the need for the host itself to have these required software installed. An explanatory diagram of how Docker works in our case can be seen in the figure 5.1.

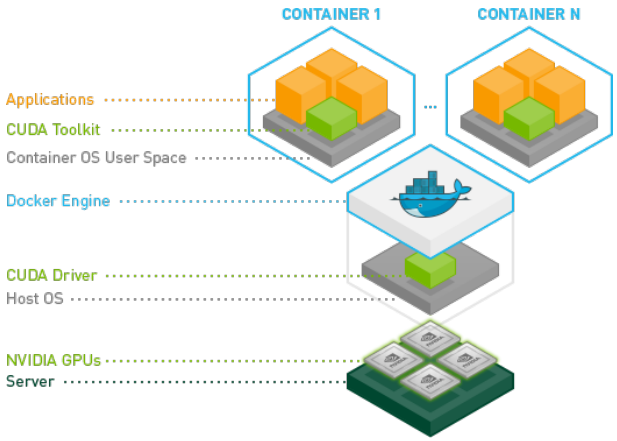


Figure 5.1: Schematic showing how Docker creates isolated OS instances with GPU capabilities. The orange blocks can be seen as TensorFlow installation, in our case. NVIDIA (2020)

The Docker image used for the development of the project has the name tag *tensorflow/tensorflow:nightly-*

gpu, which contains the latest experimental release from TensorFlow and its dependencies. However, the stable release version holding the tag *tensorflow/tensorflow:latest-gpu* should also work.

In order to allow Docker to get access to the GPU, NVIDIA offers a toolkit called nvidia-container-toolkit whose installation is mandatory. The following commands must be executed in order to get it installed:

Code snippet 5.1: NVIDIA Container Toolkit installation.

```
# Add the package repositories
distribution=$(. /etc/os-release;echo $ID$VERSION_ID)
curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | sudo apt-key add -
curl -s -L https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list | sudo tee /etc/apt/↩
    ↪ sources.list.d/nvidia-docker.list

sudo apt-get update && sudo apt-get install -y nvidia-container-toolkit
sudo systemctl restart docker
```

6 Super-resolution Applied to Eye Image using GANs

This chapter will dive into the work done in our project. Topics like the basis of the work it is based on, models used, loss functions, training methods, experiments done and so on will be explained in detail. In addition, each of these sections will be accompanied by its respective code in Python in order to be as practical and explanatory as possible.

6.1 Introduction

As we have seen in the chapter 3, in recent years there have been many breakthroughs in image super-resolution with the use of faster and deeper CNNs. However, all of them present a lack of detail in the results when using big up-scaling factors due to the way in which the error between the generated image and the ground truth image is calculated.

Therefore, this work is based on one of the latest and greatest advances in image super-resolution: GAN networks applied to image SR using a perceptual loss function, or SRGAN. All these concepts where explained in more detail in chapters 2.3 and 3. Ledig et al. (2017)

6.2 SRGAN model

As described in the section 2.3, a GAN consists of two main models, a generator and a discriminator. During the training, both networks are competing between each other with opposite objectives in a *min-max game*, the first one trying to fool the discriminator and the other one trying to discern fake images from real ones. The *minimax* equation for this specific case is shown in 6.1. Ledig et al. (2017)

$$\min_{\theta_G}\max_{\theta_D}\mathbb{E}_{I^{HR}\sim p_{train}(I^{HR})}[\log D_{\theta_D}(I^{HR})] + \mathbb{E}_{I^{LR}\sim p_G(I^{LR})}[\log(1 - D_{\theta_D}(G_{\theta_G}(I^{LR})))] \quad (6.1)$$

where: I^{HR} → High resolution image set.
I^{LR} → Low resolution image set.
$p_{train}(I^{HR})$ → Probability distribution of train HR input images.
$p_G(I^{LR})$ → Probability distribution of LR input images (can be considered as the noise input).

In this case, a LR image is obtained by applying a Gaussian filter to the ground truth image and subsequently down-sampling it by a factor k. That is, LR image's shape is $HxWxC$ and

the HR and SR images' shape is $kHxkWxC$. Thus, the generator and the discriminator will have the following roles:

- **Generator:** Given a LR image as input, it estimates its corresponding HR version by generating a SR image as output.
- **Discriminator:** It tries to discern real images from fake SR ones.

In this section each of these models will be explained and its architecture will be presented.

6.2.1 Generator architecture

The generator model used in this work is based on a fully-convolutional ResNet composed by 16 residual blocks. It is optimized to achieve an upscaling factor of 4x. Its architecture is presented in the figure 6.1, in which k corresponds to the kernel size (k x k), n to the number of channels and s corresponds to the stride (s x s). Ledig et al. (2017)

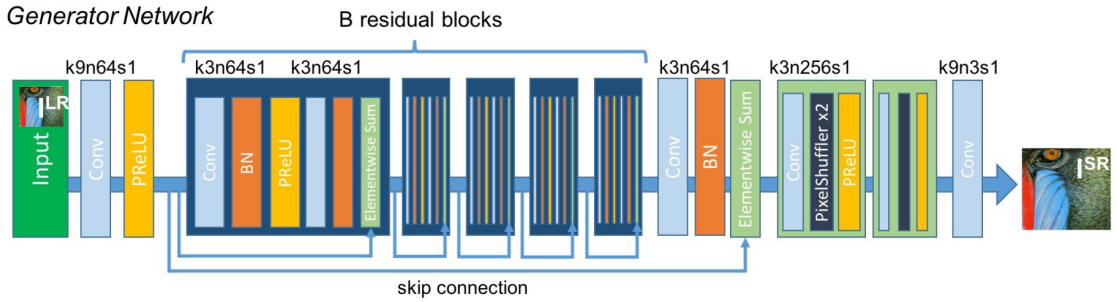


Figure 6.1: Generator architecture. Ledig et al. (2017)

6.2.2 Discriminator architecture

The discriminator model is based on a deep CNN connected to a DFF fully-connected layer at the end with a sigmoid activation function so that it returns a float value between 0 and 1, where 0 means completely fake and 1 means completely real. Its architecture is shown in the figure 6.2, having the same notation as the figure 6.1. Ledig et al. (2017)

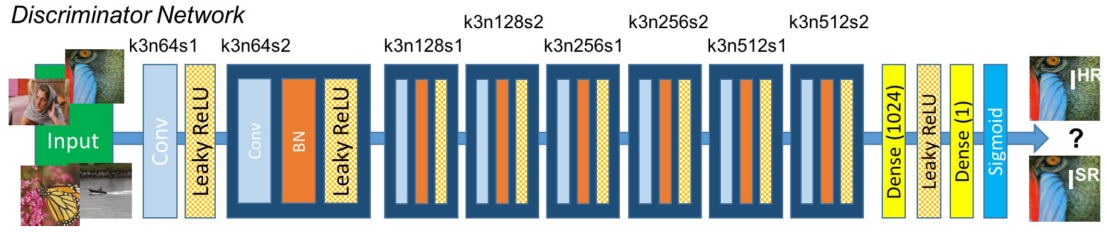


Figure 6.2: Discriminator architecture. Ledig et al. (2017)

6.2.3 Python implementation

The actual implementation of both models will be described below. As noted in the chapter 5, this project was developed using the TensorFlow library and Keras graph design notation.

Within the file *Network.py* both Generator and Discriminator architecture models can be found. Each one is implemented as a class object holding two methods: a hyper-parameter initialization method and a build method returning the constructed model based on its hyper-parameters.

6.2.3.1 Generator

Below can be found some of the most relevant parts of the Generator model implementation.

Code snippet 6.1: Generator's hyperparameter initialization method.

```
class Generator(object):
    def ___init___(self, data_format, axis, shared_axis, input_shape=None):
        self.shared_axis = shared_axis
        self.axis = axis
        self.B = 16
        self.data_format = data_format
        self.input_shape = input_shape
```

The Generator's build method is described below. The model is built layer by layer using the layer objects from Keras.

Code snippet 6.2: Generator model construction with the *build()* method.

```
    def build(self):
        # Input shape selection depending on if data_format is 'channels_last' or 'channels_first'.
        # If input_shape is None, the Generator won't have any defined input shape.
        if self.input_shape is None:
        input_generator = Input(shape=(None, None, 3) if self.data_format == 'channels_last' else (3, None
            ↪ , None))
        else:
        input_generator = Input(shape=self.input_shape)

        x = Conv2D(filters=64, kernel_size=(9, 9),
                   strides=(1, 1), padding='same',
                   activation=None)(input_generator)

        x_input_res_block = PReLU(alpha_initializer='zeros',
                                  alpha_regularizer=None,
                                  alpha_constraint=None,
                                  shared_axes=self.shared_axis)(x)

        x = x_input_res_block

        # Add B = 16 residual blocks.
        for _ in range(self.B):
        x = res_block(x, self.axis, self.shared_axis)

        x = Conv2D(64, kernel_size=(3, 3), strides=(1, 1), padding='same', activation=None, use_bias=False)
            ↪ (x)
        x = BatchNormalization(axis=self.axis)(x)

        # Adding skipped connections.
        x = add([x, x_input_res_block])

```

```
        # Upsampling blocks.
        x = up_block(x, self.shared_axis)
        x = up_block(x, self.shared_axis)

        # Output of the generator. Convolution layer with tanh activation ([−1, 1] image values).
        output_generator = Conv2D(3, kernel_size=(9, 9),
                                  strides=(1, 1), activation='tanh',
                                  use_bias=False, padding='same')(x)

        # Model creation.
        generator = Model(inputs=input_generator, outputs=output_generator, name="Generator")

        return generator
```

The *res_block()* and *up_block()* functions are implemented in the following code:

Code snippet 6.3: Definition of generator's *res_block()* and *up_block()* methods.

```
# Residual block.
def res_block(inputs, axis, shared_axis):
    x = Conv2D(64, kernel_size=(3, 3), strides=(1, 1), padding='same', activation=None,
               use_bias=False)(inputs)
    x = BatchNormalization(axis=axis)(x)
    x = PReLU(alpha_initializer='zeros', alpha_regularizer=None, alpha_constraint=None,
              shared_axes=shared_axis)(x)
    x = Conv2D(64, kernel_size=(3, 3), strides=(1, 1), padding='same', activation=None, use_bias=False)(x)
    x = BatchNormalization(axis=axis)(x)

    return add([x, inputs])


# Upsampling block.
def up_block(x, shared_axis):
    x = Conv2D(256, kernel_size=(3, 3), strides=(1, 1), padding='same', activation=None,
               use_bias=False)(x)
    x = UpSampling2D(size=(2, 2))(x)
    x = PReLU(alpha_initializer='zeros', alpha_regularizer=None, alpha_constraint=None,
              shared_axes=shared_axis)(x)
    return x
```

6.2.3.2 Discriminator

As for the Generator, the *___init___()* and *build()* methods of the Discriminator are described below.

Code snippet 6.4: Discriminator's hyperparameter initialization method.

```
class Discriminator(object):
    def ___init___(self, input_shape, data_format, axis):
        self.input_shape = input_shape
        self.data_format = data_format
        self.axis = axis
```

Code snippet 6.5: Discriminator model construction with the *build()* method.

```
def build(self):
    input_discriminator = Input(shape=self.input_shape)

    x = Conv2D(64, kernel_size=(3, 3), strides=(1, 1), use_bias=False,
               activation=None, padding='same')(input_discriminator)
    x = LeakyReLU(alpha=0.2)(x)

```

```
    x = conv_block(x, filters=64, kernel_size=(4, 4), strides=(2, 2), axis=self.axis)
    x = conv_block(x, filters=128, kernel_size=(3, 3), strides=(1, 1), axis=self.axis)
    x = conv_block(x, filters=128, kernel_size=(4, 4), strides=(2, 2), axis=self.axis)
    x = conv_block(x, filters=256, kernel_size=(3, 3), strides=(1, 1), axis=self.axis)
    x = conv_block(x, filters=256, kernel_size=(4, 4), strides=(2, 2), axis=self.axis)
    x = conv_block(x, filters=512, kernel_size=(3, 3), strides=(1, 1), axis=self.axis)
    x = conv_block(x, filters=512, kernel_size=(4, 4), strides=(2, 2), axis=self.axis)

    x = Flatten(data_format=self.data_format)(x)
    x = Dense(1024, activation=None)(x)
    x = LeakyReLU(alpha=0.2)(x)

    output_discriminator = Dense(1, activation='sigmoid')(x)

    discriminator_model = Model(inputs=input_discriminator, outputs=output_discriminator, name="↩
        ↪ Discriminator")

    return discriminator_model
```

The *conv_block()* is implemented in the following code:

Code snippet 6.6: Definition of discriminator's *conv_block()* method.

```
# Convolutional block.
def conv_block(x, filters, kernel_size, strides, axis):
    x = Conv2D(filters, kernel_size=kernel_size, strides=strides,
               activation=None, use_bias=False, padding='same')(x)
    x = BatchNormalization(axis=axis)(x)
    x = LeakyReLU(alpha=0.2)(x)
    return x
```

6.2.3.3 SRGAN

Once defined the architecture of both models, it is time to put the pieces together in order to create the full SRGAN model. This process is shown below.

Code snippet 6.7: SRGAN model construction.

```
def get_gan_model(discriminator_gan, generator_gan, input_shape):
    discriminator_gan.trainable = False

    input_gan = Input(shape=input_shape, name="SRGAN_Input")
    output_generator = generator_gan(input_gan)
    output_discriminator = discriminator_gan(output_generator)

    gan_model = Model(inputs=input_gan, outputs=[output_generator, output_discriminator], name="↩
        ↪ SRGAN")

    discriminator_gan.trainable = True

return gan_model
```

Here, the discriminator is set to not trainable while building the SRGAN model as it will be trained separately from the generator in an alternate manner during the loop, as it uses a different loss function to update its weights. This will be explained with more detail later.

6.3 Loss function

This is one of the key elements of this SR approach and is of critical importance for the generator performance.

With SRGAN, authors formulate the perceptual loss function as a weighted sum of a content loss (l_X^{SR}) and an adversarial loss component l_{Gen}^{SR}. This resulting loss function is defined in the equation 6.2. Ledig et al. (2017)

$$l^{SR} = \underbrace{\underbrace{l_X^{SR}}_{ContentLoss} + \underbrace{10^{-3} l_{Gen}^{SR}}_{AdversarialLoss}}_{PerceptualLoss} \tag{6.2}$$

The reasons of using a 10^{-3} factor for the adversarial loss are not explained by the authors. However, giving more relevance to the content loss rather than to the adversarial loss, but still keeping it, can allow the generator to focus in improving the content similarity between the generated and ground truth images whilst still forcing it to improve its results on each iteration. Ledig et al. (2017)

6.3.1 Content Loss

In the equation 6.2, l_X^{SR} holds an X as the authors present two possible content loss functions Ledig et al. (2017):

- **MSE loss (l_{MSE}^{SR}):** As noted in the chapter 3, the most of the state-of-the-art models focus in minimizing the MSE and maximizing the PSNR, both based on pixel-wise difference. However, even if the PSNR values obtained are high, the resulting images are not perceptually convincing.

$$I_{MSE}^{SR} = \frac{1}{r^2 WH} \sum_{x=1}^{rW} \sum_{y=1}^{rH} (I_{x,y}^{HR} - G_{\theta_G}(I^{LR})_{x,y})^2 \tag{6.3}$$

- **VGG loss (l_{VGG}^{SR}):** As explained in the chapter 3, one possible solution to the blurry generated images obtained when using pixel-wise MSE-based loss functions is the use of a perceptual loss function based on computing the Euclidean distance between the high-level feature representations of the generated and ground truth images. Those high-level features are obtained by passing both ground truth and generated images through a pre-trained CNN. In this case, a 19-layer VGG loss was used based on its ReLU activation layers, that is, taking the output features of a certain convolutional hidden layer. This is the best option as it gives the best results from both solutions and so it is the chosen one in this project.

$$I_{VGG/i,j}^{SR} = \frac{1}{W_{i,j} H_{i,j}} \sum_{x=1}^{W_{i,j}} \sum_{y=1}^{H_{i,j}} (\phi_{i,j}(I^{HR})_{x,y} - \phi_{i,j}(G_{\theta_G}(I^{LR}))_{x,y})^2 \tag{6.4}$$

6.3.2 Adversarial Loss

The remaining element of the perceptual function l^{SR}_{Gen} consists on the generative loss based on the probabilities of the discriminator over the samples. As described in section 2.3, the discriminator encourages the generator to improve on every iteration with the objective of fooling it. The loss function is defined by the equation 6.5, where $D_{\theta_D}(G_{\theta_G}(I^{LR}))$ is the probability that the generated SR image $G_{\theta_G}(I^{LR})$ is actually a real one. Ledig et al. (2017)

$$l^{SR}_{Gen} = \sum_{N=0}^{N} -\log D_{\theta_D}(G_{\theta_G}(I^{LR})) \tag{6.5}$$

6.4 Dataset description

Once the model and the loss functions are determined, the dataset to be used in the training process must be selected. In the case of GAN training, it is specially 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.

The dataset used in this work consists on a set of artificial eye images called U2Eyes Porta et al. (2019), which was developed and generated by the GI4E team. It includes 1000 users at a rate of 5875 images per user arranged in a three-level directory tree structure as shown in the figure 6.3. Due to space limitations as each user folder requires approximately 2.5 GB of space, 20 different users have been chosen in order to train our model, making a total of 117500 images. Porta et al. (2019)

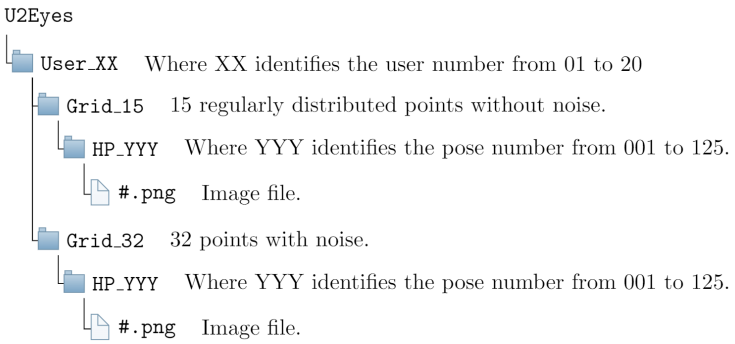


Figure 6.3: Folders and files tree structure.

Each user was generated using a PCA model offered by UnityEyes containing 20 skin-textures and 5 eye-textures which were uniformly distributed across all users. Some examples can be seen in the figure 6.4. Wood et al. (2016)

The variability of the data not only resides on the eye and skin textures, but also on the position of the user head pose and number of look-at-points. Porta et al. (2019)

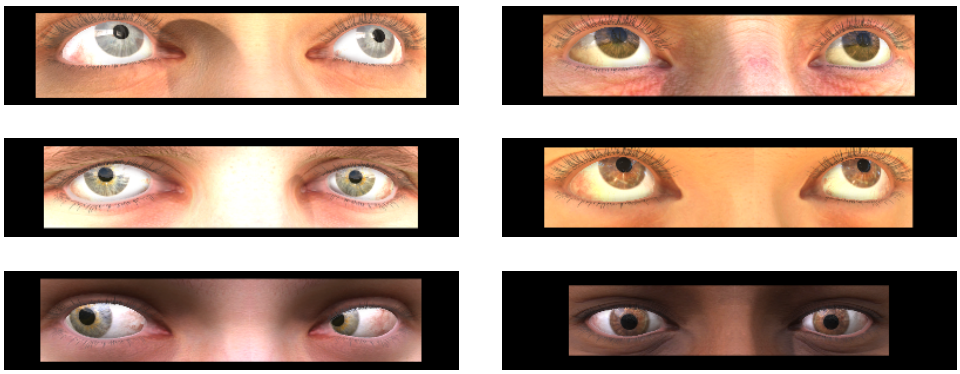


Figure 6.4: Images samples extracted from the dataset. Porta et al. (2019)

For each user, 125 head poses combining head center position and face orientation where modeled. These positions are distributed uniformly into five planes at different distances from the camera focus. Additionally, in order to avoid over-fitting, these points are randomized adding some uniform noise to them. This is shown in figure 6.5. Porta et al. (2019)

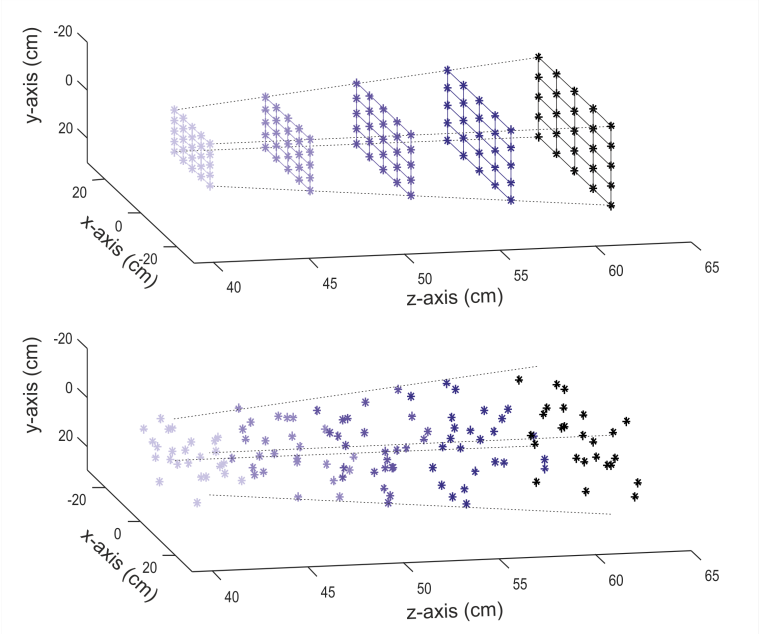


Figure 6.5: The 125 head positions following frustum distribution with five distances from the camera (up) and with added noise (down). Porta et al. (2019)

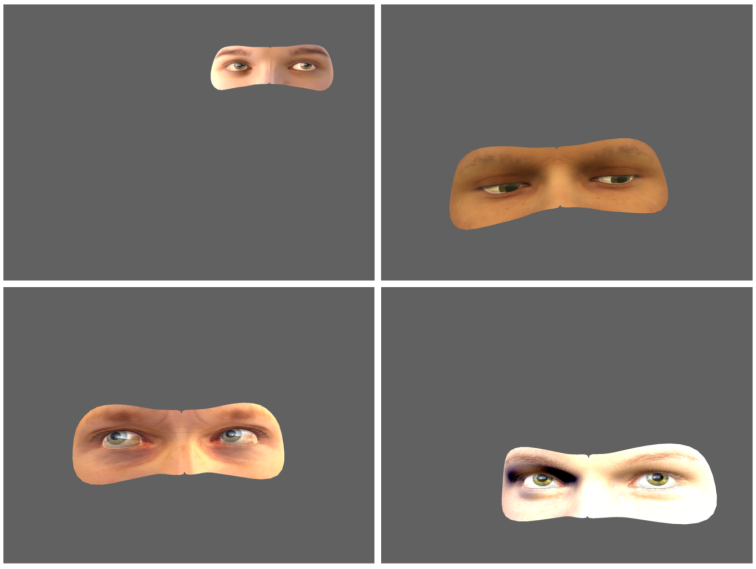
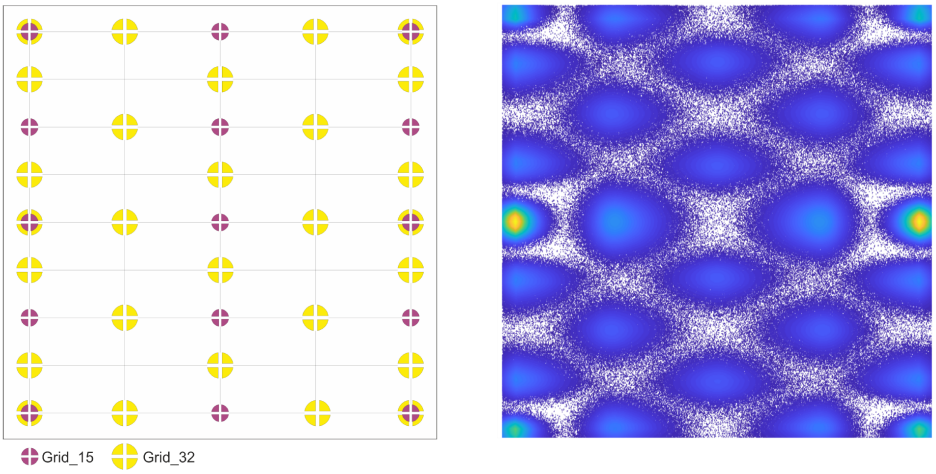


Figure 6.6: Image examples with different poses. Porta et al. (2019)

Moreover, for each head pose two gazing grids are modeled: the first one containing 15 points uniformly spread in a 220x220mm squared surface, and the second one containing 32 points not uniformly spread in the same surface. This is illustrated in the figure 6.7. Porta et al. (2019)



(a) Grid pattern of 15 and 32 points for each head pose.

(b) Point distribution for the 32 look-at-points grid.

Figure 6.7: 15 and 32 grid each user gazes for each pose. Porta et al. (2019)

6.5 SR metrics

The evaluation of SR techniques is not straightforward. Several efforts have been carried out in order to tackle the problem of assessing the performance of SR methods. Conclusions show that there is not a single criteria that can be generally applicable for every SR application domain. Among the existing general purpose metrics we can differentiate between numeric values such as PSNR Horé & Ziou (2010) or SSIM Wang et al. (2004) and more qualitative measures such as Mean Option Score (MOS) based on human raters opinions Streijl et al. (2016).

Alternatively, task-based evaluation permits to measure the performance of the reconstruction SR method by comparing the task results using original and reconstructed HR images. The impact produced by the SR generated images can be estimated according to the performance variation of our task Wang et al. (2020)

In this work, both, general purpose and task-based metrics have been used.

6.5.1 General purpose metrics

- **PSNR:** It is one of the most popular merit function. Given an image I of size k and its reconstruction named as $\hat{I}$, $PSNR$ is defined as:

$$PSNR = 10 \cdot \log_{10}\left(\frac{L^2}{\frac{1}{k}\sum_{i=1}^{k}(I(i) - \hat{I}(i))^2}\right) \tag{6.6}$$

  where L equals to 255 in general cases using 8-bit representations. Since the PSNR is only related to the pixel-level error and does not retrieve any performance value about visual perception, it often leads to poor performance in real scenes. However, despite of its low accuracy, due to the lack of appropriate perceptual metrics, it is the most widely employed method in order to compare SR works Horé & Ziou (2010).

- **SSIM:** It uses knowledge about human visual system in order to propose a quantifiable comparison tool. It is based on concepts such as luminance, contrast and structural information. Given an image I and its reconstruction named as $\hat{I}$, SSIM is defined as:

$$SSIM(I, \hat{I}) = \frac{(2\mu_I\mu_{\hat{I}} + c_1) \cdot (2\sigma_{I\hat{I}} + c_2)}{(\mu_I^2 + \mu_{\hat{I}}^2 + c_1) \cdot (\sigma_I^2 + \sigma_{\hat{I}}^2 + c_2)} \tag{6.7}$$

  where μ_X and σ_X are the average and the standard deviation of X respectively and σ_{XY} is the covariance between X and Y images. On the other hand, c_1 and c_2 are constants that depend on the dynamic range of the images that permit to account for the cases with weak denominator Wang et al. (2004).

  Due to the fact that $SSIM$ evaluates the reconstruction quality considering human visual system performance it retrieves an improved perceptual assessment.

6.5.2 Task-based evaluation: gaze estimation

In order to assess the performance of our model, reconstructed images will be used as input to a gaze estimation model that has previously been tested on the original images. The architecture of this network consists on a ResNet-18 He et al. (2016) as backbone, followed by a Global Average Pooling layer and three Fully-connected (FC) layers. The choice of the ResNet-18 as the core of our architecture is supported by its performance over Imagenet Deng et al. (2009) classification task while being simple enough to make retraining steps feasible in both, time and hardware requirements.

Results obtained over Imagenet ensure that the network is able to extract meaningful features from images. The three fully connected layers at the top of the network make use of these features to compute the X and Y coordinates of the look-at-points as it can be seen in figure 6.8. To avoid excessive over-fitting, L2 regularization with a regularization factor of 0.001, batch normalization (BN) after the fully connected layers, and a dropout layer with a dropout value of 0.5 were used. Additionally, Gaussian noise with a standard deviation of 0.2 was added to the input images to avoid over-fitting too.

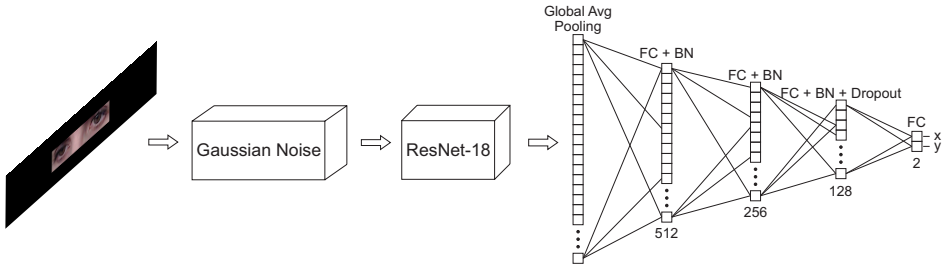


Figure 6.8: Architecture proposed. The backbone consists in a ResNet-18 to extract meaningful features from the image. Then, those features are fed into fully connected regression network to obtain the final gaze components. He et al. (2016)

The model is trained over the synthetic dataset U2Eyes using original images. The goal with this step is to bring closer the domain in which the model is trained to the final environment. The models are trained over 96 epochs, using a batch size of 128 images. The loss function employed is the Euclidean distance between the estimated and the real look-at-point, represented by the following equation:

$$Loss := \frac{1}{N}\sum_{i=1}^{N} \|p - \hat{p}\|_2, \tag{6.8}$$

where p is the real look-at-point $p(x, y)$, $\hat{p}$ is the estimated look-at-point, and N is the number of images per batch. Stochastic Gradient Descent (SGD) with momentum=0.95

using Nesterov momentum is employed to optimize the loss function. The learning rate schedule followed is based on the Cyclic learning rate schedule Smith (2017) in a triangular manner for the first 80 epochs, fluctuating among a max learning rate value of $3e^{-2}$ and a min learning rate of $3e^{-3}$, and a linear decrease for the remaining 16 epochs from $3e^{-3}$ to $3e^{-4}$. Garde et al. (2020).

6.5.3 Task-based evaluation: iris/pupil center and contour points estimation using an SDM

The GI4E team is also developing a semiautomatic tool for annotation of eye tracking images based on SDM Larumbe-Bergera et al. (2019). In order to introduce another testing to our SR model, some generated images extracted for each user from the U2Eye dataset where passed through this algorithm, and its results where stored and compared where compared with the ones obtained using original images using some Matlab scripts. In the figure 6.9 the algorithm is applied to an image from GI4E.

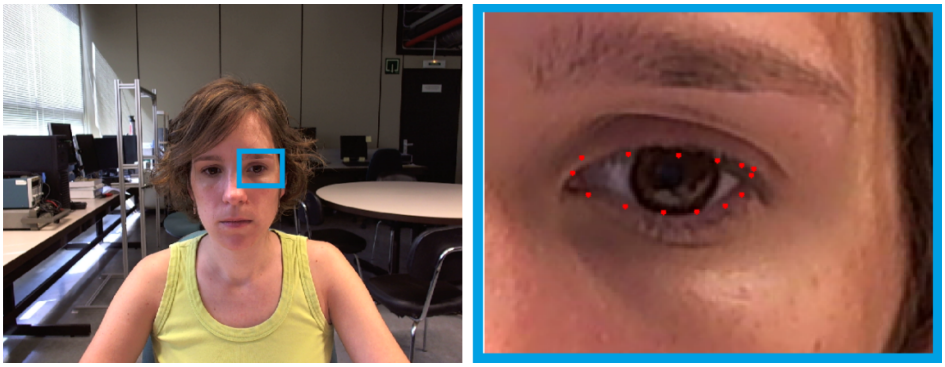


Figure 6.9: The cropped ROI is represented in which the points resulting from the segmentation are plotted in red. Porta et al. (2019)

6.6 Experiments

As mentioned in the dataset description section, the U2Eyes dataset contains 20 different head shapes and skin textures. However, only 5 eye textures are available. In order to evaluate our model, a Leave-One-Out strategy has been agreed considering the five textures separately. In other words, five experiments have been carried out in which all the users having the same texture have been excluded from the training for each of the training stages.

Regarding the gaze estimation model, the same strategy was used in order to assure the minimal overlapping between training and testing stages. The users were classified according to their eye texture following the same grouping carried out for SR model training. Thus, five independent gaze estimation models were trained for each one of the five testing textures.

6.6.1 Data pre-processing

The purpose of the SRGAN model is to scale an input image resolution by a factor of 4, which corresponds to a 16x increase in the number of pixels. The U2Eyes dataset used in our experiments has been processed in order to normalize the size of the images employed by cropping the eye region of interest and applying a black padding until the images reach a common shape resolution of 388x84 pixels.

However, the black frames used to pad the image to make it reach a normalized resolution of 388x84 pixels, as it can be seen in figure 6.4, made the generator produce some artifacts affecting ourROI. This effect is shown in figure 6.10.

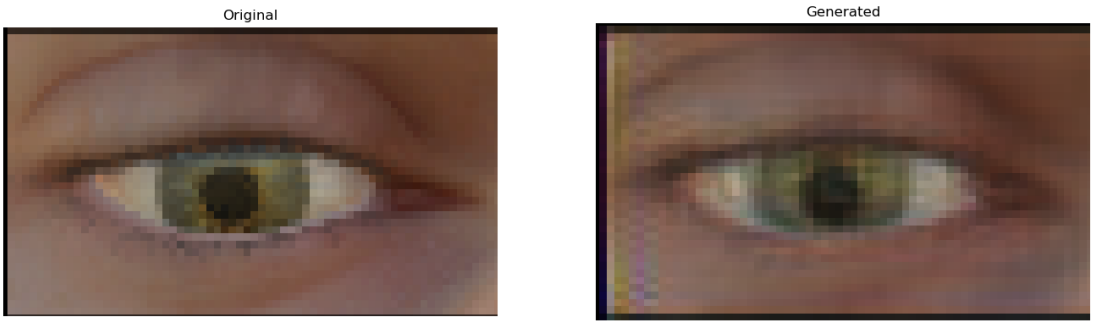


Figure 6.10: The generated image presents strange artifacts on the image's sides.

This is why we decided to fill those black frames by replicating the image borders until the image is fully covered. This can be seen in figure 6.11b.

(a) Gaze image with black frames. **(b)** Gaze image with filled frames.

Figure 6.11: Comparison between eye images with black frames and with filled frames.

6.6.2 Training the model

This section will detail the complete training process, starting with the basis of the process, parameters used and the pre-processing applied to the input data. As it was done in previous sections, every explanation will be accompanied by its respective Python code snippet.

Our training method is based on the work of Ledig et al. (2017), 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.

6.6.2.1 MSE-based pre-training

This stage consists in performing an initial training of the generator using a MSE-based loss function before training the model with the GAN method.

According to the authors, 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. In our case, this step is of high importance, as we were unable to make the GAN training converge to a good solution without pre-training the generator first. Ledig et al. (2017)

The most important parts of the process will be illustrated with code below. How the dataset is pre-processed and introduced into the model during the training process will be explained in a separate section.

In first place, as in the paper, for optimization we used Adam with $\beta_1 = 0.9$ with a learning rate of 10^{-4}. The SRResNet was built with an MSE loss function. Ledig et al. (2017)

Code snippet 6.8: Optimizer and Generator construction.

```
common_optimizer = tf.keras.optimizers.Adam(lr=1e-4, beta_1=0.9)

generator = Network.Generator(data_format=data_format, axis=axis, shared_axis=shared_axis).build()
generator.compile(loss='mse', optimizer=common_optimizer)
```

In order to save the best model weights during training, some callbacks where added. Moreover, an EarlyStopping callback was also defined in order to stop the training in case the loss value do not fall by at least 0.001 in two consecutive epochs. Finally, a TensorBoard callback has been introduced in order to save the session events, allowing us to analyze the training session and identify when the loss function reaches a plateau state[1], and thus avoid over-fitting.

Code snippet 6.9: Callbacks construction.

```
checkpoint = ModelCheckpoint(
filepath='./outputs/checkpoints/SRResNet-MSE/weights.{''epoch:02d}-{'
'loss:.4f}.hdf5',
monitor='loss',
save_weights_only=True,
save_freq=2000,
verbose=2)

best_checkpoint = ModelCheckpoint(
filepath='./outputs/checkpoints/SRResNet-MSE/best_weights.hdf5',
monitor='loss',
save_weights_only=True,
save_best_only=True,
save_freq=2000,
verbose=2)

early_stop = EarlyStopping(monitor='loss', min_delta=0.001, patience=2, verbose=1, mode='min')

tensorboard_callback = TensorBoard(log_dir='logs/',
```

[1] A state in which the model is not improving its loss values, resulting in an horizontal line in the loss graphical representation.

```
20 histogram_freq=1,
21 update_freq=100,
22 profile_batch='200,250'
23 )
24
25 callbacks = [tensorboard_callback, checkpoint, best_checkpoint, early_stop]
```

Finally, the generator is trained for 3 epochs with a batch size of 6, as this value was the limit for our 8GB VRAM GPU. In each epoch, all the images of the dataset are covered, that is, the steps_per_epoch paramater is equal to the number of images in the dataset divided by the batch size.

Code snippet 6.10: Generator training method.

```
1 history = generator.fit(x=train_ds, epochs=epochs, steps_per_epoch=steps_per_epoch, callbacks=callbacks)
```

The epoch value is taken from the analysis of the loss value graph shown in the figure 6.12.

Figure 6.12: Loss value graph per batch processed (up). Loss value graph per epoch (down), where orange corresponds to training values and blue corresponds to validation values.

As it can be seen, from batch 20000 and 25000 (with batch size of 6) and epoch 1 to 2, the model reaches a plateau state. Therefore, there is no point in training more than 2 or

3 epochs using a batch size of 6 and a. This can be explained due to the great similarity between every image in the dataset.

When a plateau state is reached, it is very common in DFF neural networks to retrain the model using the saved weights from the previous session, but lowering the learning rate by a certain factor, with the aim of fine-tuning the model weights. Hence, the generator was trained again with a learning rate of 10^{-5} for another 3 epochs. However, probably due to the similarity between images, the loss value didn't improve as expected and there is no noticeable difference in the results.

6.6.2.2 SRGAN training

Once the pre-training phase is accomplished, the generator will be trained using the GAN method.

As mentioned in the Loss Function section, specifically in the equation 6.2, the loss function used in the SRGAN model is a weighted combination of two losses: a content loss, denoted as l_X^{SR}, and an adversarial loss (l_{Gen}^{SR}).

It has also been explained that the content loss can be a MSE-based loss ($l_M^{SR}SE$) or a perceptual loss based on a certain layer of a VGGnet CNN, noted as $l_V^{SR}GG$. The authors of the paper Ledig et al. (2017) performed some experiments by varying the activation layer to be set as the output of our features extractor. They identified two different losses:

- **SRGAN-VGG22 ($l_{VGG/2.2}^{SR}$):** A loss defined on lower-level features. The output layer holding the features corresponds to the 2^{nd} maxpooling layer of the 2^{nd} convolution layer (after activation). In other words, $\phi_{2,2}$ is used inside the formula 6.4.
- **SRGAN-VGG54 ($l_{VGG/5.4}^{SR}$):** A loss defined on higher-level features from deeper network layers which are more focused on the content of images. Following the same idea as in the first point, the used output layer corresponds to the 5^{th} maxpooling layer of the 4^{th} convolution layer, that is, $\phi_{5,4}$ is used inside the formula 6.4.

In their work, the authors tested the performance of the model using each of the presented content losses, including without the adversarial component. According to their results, the loss function combining an adversarial loss and a $l_{VGG/5.4}^{SR}$ function gave the best results. Hence, this is the configuration used for our work. Ledig et al. (2017)

As for the previous stage, the most important parts of the process will be illustrated with code below. As the discriminator needs to be trained apart from the generator, a custom loop was required in order to train the full SRGAN model.

Just as in the pre-training phase, for optimization we used Adam with $\beta_1 = 0.9$ with a learning rate of 10^{-4}. As it can be seen in the code snippet below, both generator and discriminator are built. The first one is compiled using a binary_crossentropy loss function, which is convenient for output activation float values between 0 and 1. Ledig et al. (2017)

Moreover, the Generator is loaded with the pre-trained weights obtained from the previous stage. Besides, it is not matched to a loss function, as it will be trained using the SRGAN loss function which will be built later on.

Code snippet 6.11: Discriminator, generator and feature extractor construction.

```
# Discriminator construction.
discriminator = Network.Discriminator(input_shape=target_shape, axis=axis, data_format=data_format).↩
    ↪ build()
discriminator.compile(loss='binary_crossentropy', optimizer=common_optimizer)

# Generator construction.
generator = Network.Generator(data_format=data_format,
axis=axis, shared_axis=shared_axis).build()
generator.load_weights('./saved_weights/SRResNet-MSE/best_weights.hdf5')

# Feature extractor construction.
features_extractor = build_vgg(target_shape)
```

In this code, the feature extractor is also built. Its function implementation is shown in code snippet below.

Code snippet 6.12: Feature extractor construction based on a pre-trained VGG19 model.

```
def build_vgg(target_shape_vgg):
vgg19 = VGG19(include_top=False, input_shape=target_shape_vgg, weights='imagenet')

vgg19.trainable = False
for layer in vgg19.layers:
layer.trainable = False

vgg_model = Model(inputs=vgg19.input, outputs=vgg19.layers[20].output, name="VGG")

return vgg_model
```

As explained before, it is a 19-layer VGGnet CNN pre-trained with the Imagenet dataset. The 20^{th} position of the layer array is used as the ouput of our feature extractor, as it corresponds to the 5^{th} maxpooling layer of the 4^{th} convolution. In case we want to get a VGG22, the output layer to be used corresponds to the 9^{th} position of the array. Ledig et al. (2017)

To continue, the complete SRGAN model is built from the parts we have already defined.

Code snippet 6.13: Building and compiling the SRGAN model.

```
# Building and compiling the SRGAN
gan = get_gan_model(discriminator_gan=discriminator, generator_gan=generator, input_shape=shape)

gan.compile(loss=[vgg_loss, 'binary_crossentropy'], loss_weights=[1, 1e-3],
optimizer=common_optimizer)
```

As it can be noted above, the SRGAN model is compiled with the loss function defined in the equation 6.2, where the *vgg_loss* corresponds to the content loss $l^{SR}_{VGG/5.4}$, and the *binary_crossentropy* corresponds to the adversarial loss function.

Note that the *compile()* method allows to set a weighted sum of loss functions by filling the *loss_weights* parameter with an array containing the corresponding scaling factors, which were extracted from the equation 6.2.

Code snippet 6.14: Building the content loss function using the feature extractor.

```
def vgg_loss(y_true, y_pred):
return 0.006 * K.mean(K.square(features_extractor(preprocess_vgg(y_pred)) -
features_extractor(preprocess_vgg(y_true))),
axis=-1)
```

In the code snippet above, the content loss function $l^{SR}_{VGG/5.4}$ is defined using the *feature_extractor* previously defined. As described in equation 6.4, the loss function is based on the Euclidean distance between the features extracted from the generated and the real images. The 0.006 factor applied to the formula, according to the authors, corresponds to a 1/12.75 scaling factor applied to the content loss function l^{SR}_{VGG} in order to obtain VGG loss values of a scale comparable to using a l^{SR}_{MSE} content loss. Ledig et al. (2017)

Finally, the full SRGAN is trained in a custom loop for another 3 epochs with a batch size of 6 images. Within this loop, the discriminator is trained separately from the generator. In each epoch, all the images of the dataset are covered, that is, the *steps_per_epoch* parameter is equal to the number of images in the dataset divided by the batch size.

In the first stage of this process, the discriminator is trained with fake and real images separately. A random noise is added to the labels in order to force the model to improve its decisions. This process can be seen in the code snippet below:

Code snippet 6.15: Discriminator training process.

```
# The discriminator's trainable state is set to true in order to be able to update its weights during this stage.
discriminator.trainable = True

# Every iteration a batch of 6 \gls{lr} and \gls{hr} is retrieved.
lr_images, hr_images = next(iterator)

# The actual state of the generator is used in order to generate a batch of images which will be fed to the ↩
    ↪ discriminator.
sr_images = generator.predict(lr_images)

# Generate batch of fake and real labels. They are randomized in order to force the discriminator to improve ↩
    ↪ its decision-making.
real_labels = np.random.uniform(0.7, 1.0, size=batch_size).astype(np.float32)
fake_labels = np.random.uniform(0.0, 0.3, size=batch_size).astype(np.float32)

# The resulting loss values are stored in two arrays.
d_loss_real = discriminator.train_on_batch(hr_images, real_labels)
d_losses_real.append(d_loss_real)

d_loss_fake = discriminator.train_on_batch(sr_images, fake_labels)
d_losses_fake.append(d_loss_fake)
```

Once the discriminator's weights are updated, in the second stage of the process the generator will be trained.

Code snippet 6.16: Generator training process with SRGAN method.

```
# The discriminator's weights are blocked again so that its weights cannot be updated while training the ↩
    ↪ generator.
discriminator.trainable = False

```

```
# A different batch of 6 \gls{lr} and \gls{hr} is retrieved.
lr_images, hr_images = next(iterator)

opposite_labels = np.ones((batch_size, 1)).astype(np.float32)

# Train the generator network by using the \gls{hr} images and the labels.
g_loss = gan.train_on_batch(lr_images, [hr_images, opposite_labels])
g_losses_mse_vgg.append(g_loss[0])
```

7 Results

In figure 7.1 random samples obtained from our model are shown. The fist row shows scaled versions of the LR images used as input to the SRGAN model (the size has been increased in order to make it comparable). In the second row original images are shown against which the rest of the images should be compared to. In the third and fourth rows SRGAN and SRResNet-MSE generated images are provided. SRGAN is the final result of our model and the one to be compared with the original one, however, SRResNet-MSE is included for the sake of completeness. Finally, in the last row the result of a bicubic standard interpolation applied to the LR image is provided.

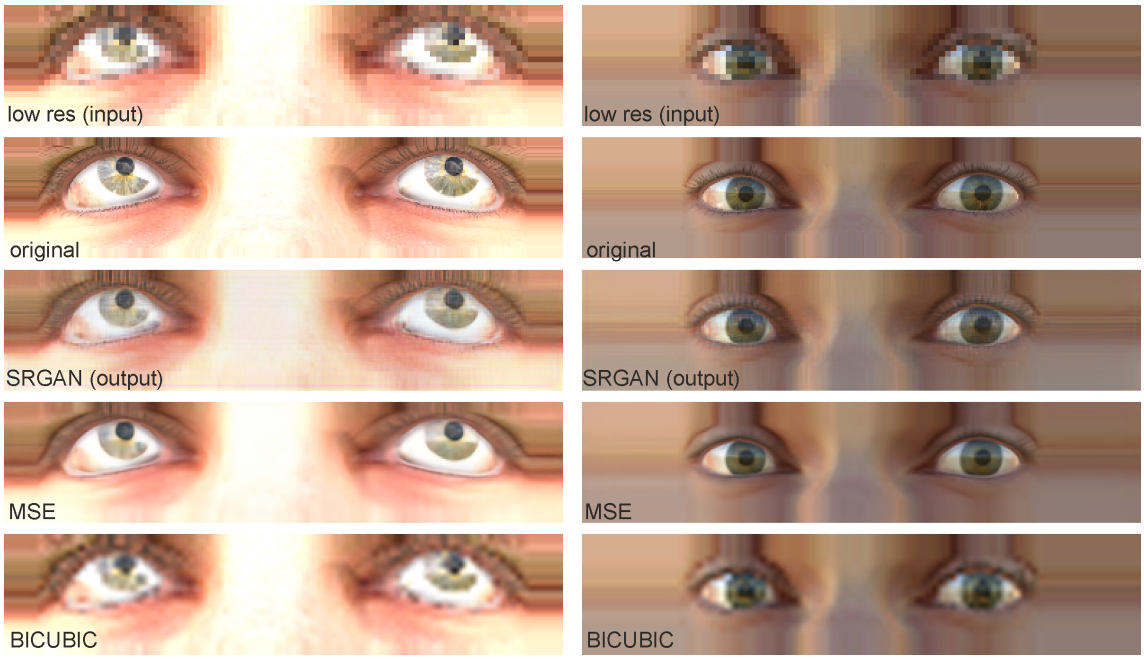


Figure 7.1: Examples of generated images.

From the figures it can be appreciated the ability of our SRGAN model to reconstruct the image and to recover details that are missing in the LR eye version. It can be visually appreciated that SRGAN images present the highest similarity when compared with the original ones.

As mentioned in section 6.5, two types of metrics will be used for comparison. Regarding PSNR and SSIM the following values are shown in table 7.1:

From this table can be noted that the PSNR for both SRResNet-MSE and SRGAN (pro-

	$PSNR$ (dB)	$SSIM$
MSE-image	32.93±2.52	0.86±0.05
SRGAN-image	27.67±1.75	0.83±0.05

Table 7.1: PSNR and SSIM merit figure values forSR images.

posed model) is comparable but the SR image obtained from MSE is not perceptually convincing. Hence, the following two conclusions can be obtained:

- The measurement indices PSNR and SSIM do not capture the perceptually relevant differences in images.
- The standard SR algorithms such as bicubic interpolation are unable to produce quality super resolved images with high frequency details.

The remarkable metric values obtained by the SRResNet-MSE are not surprising since the minimization of the MSE between the original and reconstructed images results in the maximization of the PSNR. This conclusion is not new and resembles to a large extent the findings provided by the literature Ledig et al. (2017). PSNR applied to SRResNet-MSE results, is not able to capture texture-rich features, consequently, high values of PSNR do not assure a perceptually better outcome. This probably justifies the emergence of MOS metrics in the field.

Users' images belonging to each one of the five textures were used as input to the texture-specific gaze estimation model. The gaze estimation accuracy obtained by the SRGAN and SRResNet-MSE models was compared with the one provided by the original images. The comparison is shown in figure 7.2.

As shown in the figure, the impact of using SRGAN images is negligible compared to the values obtained by the original images, which means that the information lost by the reconstruction procedure is not meaningful in terms of gaze estimation. Furthermore, according to the results the SRResNet-MSE images provide fully comparable results leading to the conclusion that high frequency details do not contribute to improve the model accuracy of the gaze estimation process.

Regarding iris/pupil center estimation, the SDM algorithm (Larumbe-Bergera et al. (2019)) has been applied to some images taken from the 20 users within the U2Eyes dataset. An accumulated error between the ground truth and the points obtained from original and generated images is shown in figure 7.3.

As it can be noted, approximately 80% of images tested present a normalized error of less or equal than 0.01 for both, original and SRGAN generated images. Furthermore, the curves are pretty close between each other, which proves once again the effectiveness of the SR model in giving original-like results. On the other hand, MSE generated and Bicubic images give worse results in terms of iris/pupil center estimation.

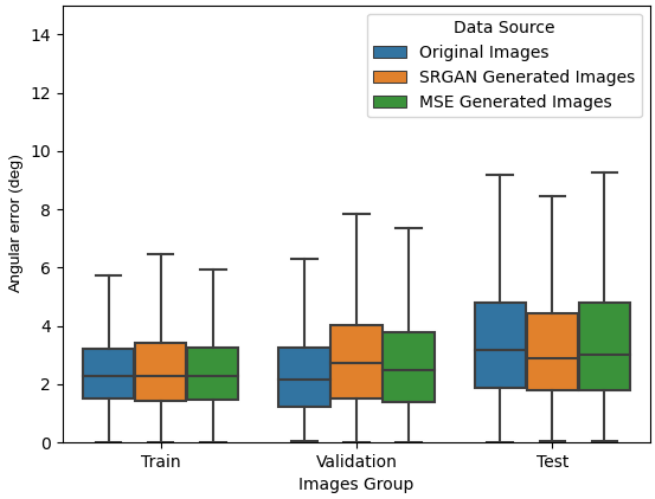
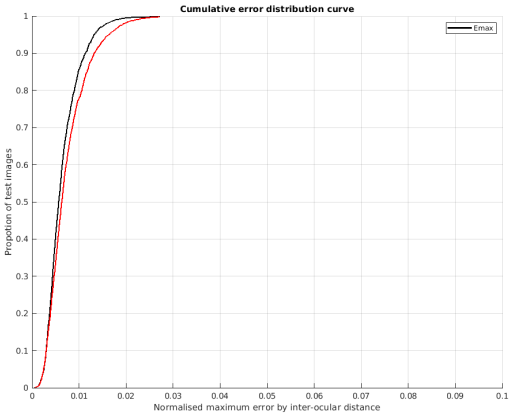


Figure 7.2: Gaze estimation errors provided by the model when using original, SRResNet-MSE and SRGAN images as input. Train, validation and test results are provided.

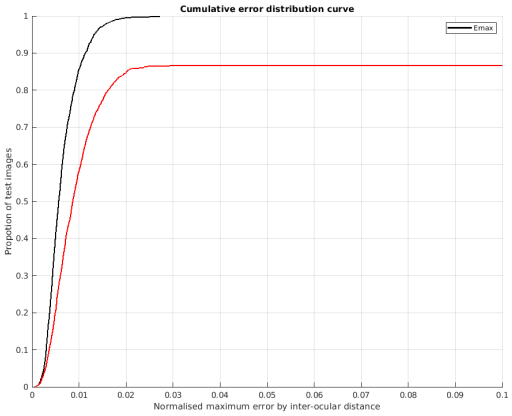
The SRGAN model is not useful only for reducing the computing performance requirements of the image acquisition process, as it can also be used to improve the image quality by increasing its resolution by a factor of 4. That is, when introducing an image with a size of 388x84 pixels to the generator, an enhanced image of size 1552x336 will be retrieved as output. Furthermore, the SRGAN can be employed to scale any image, however, the performance of the model varies according to the computer vision domain it is applied to.

The potential of having a domain-specific SR model permits to specialize the network in a specialized research field, i.e. eye tracking. In this respect, the model proposed in our work permits to reconstruct eye images which are not necessarily coming from the training dataset. As it can be noted in figure 7.4, the high quality results of applying SRGAN to real eye images are shown. A single eye from an enhanced 1552x336 size image is shown in order to highlight the improvement.

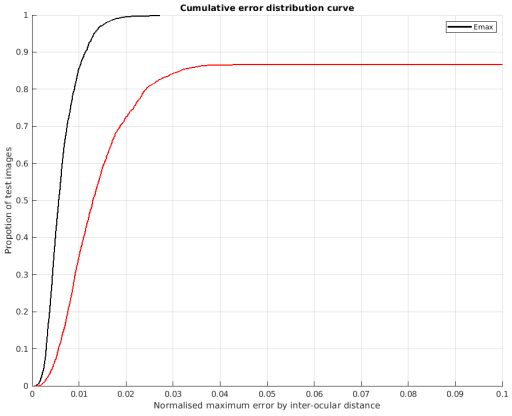
In contrast, the result of applying the model to other everyday images is shown in figure 7.5. The model achieves to increase the resolution of the image and to smooth the contours as for the eye images. However, the colors of the images are altered and the quality of the image is lower compared to the eyes from a perceptual point of view.



(a) SRGAN



(b) MSE



(c) Bicubic

Figure 7.3: Accumulated error comparison between original and generated images for the User 18. The error is normalized using the distance between both pupil centers points. Black line corresponds to results obtained using original images and the red line to the generated images results.

Figure 7.4: Binocular real eye 388x84 images are used as input to the SRGAN (upper row, cropped to single eye). The output of the network is an enhanced eye image of size 1552x336 (lower row, cropped to single eye). The quality can be visually evaluated.

Figure 7.5: In the upper row ROIs of 200x200 cropped from non-eye images are shown. In the lower row the generated 800x800 versions are provided.

8 Conclusions

In this report, a SR model has been presented to reconstruct eye tracking images. The model is implemented using an SRGAN training method based on two optimization steps, namely, SRResNet-MSE pre-training and final SRGAN training.

U2Eyes dataset has been used as training framework in order to feed our model. Our method achieves to reconstruct eye images with a perceptually noticeable quality. The numeric evaluation of the method delivers results confirming that standard metrics such as PSNR and SSIM are not able to match human perception. Furthermore, a gaze estimation model is used as a task-specific evaluation metric to measure the performance of our model. Comparable accuracy is obtained for original, SRResNet-MSE and SRGAN images. Therefore, computational resources could be saved during the acquisition of eye images since the missed information can be reconstructed in a post processing stage.

Interestingly, according to the results obtained, gaze estimation model apparently does not exploit high frequency details of the image in order to estimate gaze in the range of accuracies provided. Future research lines suggest to explore, among others, the importance of the domain in the performance of the SR model, to develop more specific eye tracking metrics for SR methods and to study the relevant image features for deep learning based gaze estimation.

Bibliography

Abass, A., Vinciguerra, R., Lopes, B. T., Bao, F., Vinciguerra, P., Jr, R. A., & Elsheikh, A. (2018). Positions of ocular geometrical and visual axes in brazilian, chinese and italian populations. *Current Eye Research*, *43*(11), 1404-1414. (PMID: 30009634) doi: 10.1080/02713683.2018.1500609

Alonso, J. B. (2011-2012). Introducción a la inteligencia artificial. Universitat Politècnica de Catalunya. Retrieved from `https://www.cs.upc.edu/~bejar/ia/transpas/teoria/1-IA-introduccion.pdf`

Anwar, S., Khan, S., & Barnes, N. (2020, May). A deep journey into super-resolution: A survey. *ACM Comput. Surv.*, *53*(3). doi: 10.1145/3390462

Briega, R. E. L. (2017). Introducción a la inteligencia artificial.. Retrieved from `https://es.slideshare.net/mentelibre/introduccion-a-las-redes-neuronales-artificiales`

Brownlee, J. (2019). Understand the impact of learning rate on neural network performance. machinelearningmastery.com. Retrieved from `https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/`

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *Cvpr09.*

Frossard, D. (2016). Vgg in tensorflow. cs.toronto.edu. Retrieved from `https://www.cs.toronto.edu/~frossard/post/vgg16/`

Fumo, D. (2017). Types of machine learning algorithms you should know. towardsdatascience.com. Retrieved from `https://towardsdatascience.com/types-of-machine-learning-algorithms-you-should-know-953a08248861`

Garde, G., Larumbe-Bergera, A., Bossavit, B., Cabeza, R., Porta, S., & Villanueva, A. (2020). Gaze estimation problem tackled through synthetic images. In *Acm symposium on eye tracking research and applications.* New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3379156.3391368

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., … Bengio, Y. (2014). Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems 27* (pp. 2672–2680). Curran Associates, Inc. Retrieved from `http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf`

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *2016 ieee conference on computer vision and pattern recognition (cvpr)* (p. 770-778).

Horé, A., & Ziou, D. (2010). Image quality metrics: Psnr vs. ssim. In *2010 20th international conference on pattern recognition* (p. 2366-2369).

Huang, Y., Shao, L., & Frangi, A. F. (2017). Simultaneous super-resolution and cross-modality synthesis of 3d medical images using weakly-supervised joint convolutional sparse coding. In *2017 IEEE conference on computer vision and pattern recognition, CVPR 2017, honolulu, hi, usa, july 21-26, 2017* (pp. 5787–5796). IEEE Computer Society. doi: 10.1109/CVPR.2017.613

Isaac, J. S., & Kulkarni, R. (2015). Super resolution techniques for medical image processing. In *2015 international conference on technologies for sustainable development (ictsd)* (p. 1-6).

Jordan, J. (2018). Setting the learning rate of your neural network. jeremyjordan.me. Retrieved from `https://www.jeremyjordan.me/nn-learning-rate/`

Kaiming He, S. R. J. S., Xiangyu Zhang. (2015). *Deep residual learning for image recognition.*

Kańska, K. (2019). Can you enhance that? single image super resolution. PyData. Retrieved from `https://www.youtube.com/watch?v=lmUxbRY7H2I`

Kim, J., Lee, J. K., & Lee, K. M. (2016). Accurate image super-resolution using very deep convolutional networks. In *2016 ieee conference on computer vision and pattern recognition (cvpr)* (p. 1646-1654).

Larumbe-Bergera, A., Porta, S., Cabeza, R., & Villanueva, A. (2019, 06). Seta: semiautomatic tool for annotation of eye tracking images. In (p. 1-5). doi: 10.1145/3314111.3319830

Ledig, C., Theis, L., Huszár, F., Caballero, J. A., Aitken, A., Tejani, A., … 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.

Malhotra, A. (2019). Generative classifiers v/s discriminative classifiers. medium.com. Retrieved from `https://medium.com/@akankshamalhotra24/generative-classifiers-v-s-discriminative-classifiers-1045f499d8cc`

Moreno, C. G. (2016). ¿qué es el deep learning y para qué sirve? Indra. Retrieved from `https://www.indracompany.com/es/blogneo/deep-learning-sirve`

Mosquera, D. G. (2018). Gans from scratch 1: A deep introduction. with code in pytorch and tensorflow. medium.com. Retrieved from `https://medium.com/ai-society/gans-from-scratch-1-a-deep-introduction-with-code-in-pytorch-and-tensorflow-cb03cdcdba0f`

Nicholson, C. (2018). A beginner's guide to generative adversarial networks (gans). pathmind.com. Retrieved from `https://pathmind.com/wiki/generative-adversarial-network-gan`

NVIDIA. (2020). Build and run docker containers leveraging nvidia gpus. Author. Retrieved from `https://github.com/NVIDIA/nvidia-docker`

Porta, S., Bossavit, B., Cabeza, R., Larumbe-Bergera, A., Garde, G., & Villanueva, A. (2019, Oct). U2eyes: A binocular dataset for eye tracking and gaze estimation. In *2019 ieee international conference on computer vision (iccv).*

Rosebrock, A. (2017). Imagenet: Vggnet, resnet, inception, and xception with keras. pyimagesearch.com. Retrieved from `https://www.pyimagesearch.com/2017/03/20/imagenet-vggnet-resnet-inception-xception-keras/`

Saha, S. (2018). A comprehensive guide to convolutional neural networks — the eli5 way. towardsdatascience.com. Retrieved from `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`

Sahu, B. (2019). An evolution in single image super resolution using deep learning. towardsdatascience.com. Retrieved from `https://towardsdatascience.com/an-evolution-in-single-image-super-resolution-using-deep-learning-66f0adfb2d6b`

Sajjadi, M. S. M., Schölkopf, B., & Hirsch, M. (2017). Enhancenet: Single image super-resolution through automated texture synthesis. In *2017 ieee international conference on computer vision (iccv)* (p. 4501-4510).

Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. *Conference paper at ICLR 2015.*

Skalski, P. (2018). Deep dive into math behind deep networks. towardsdatascience.com. Retrieved from `https://towardsdatascience.com/https-medium-com-piotr-skalski92-deep-dive-into-deep-networks-math-17660bc376ba`

Smith, L. N. (2017). *Cyclical learning rates for training neural networks.*

Streijl, R. C., Winkler, S., & Hands, D. S. (2016, March). Mean opinion score (mos) revisited: Methods and applications, limitations and alternatives. *Multimedia Syst.*, *22*(2), 213–227. doi: 10.1007/s00530-014-0446-1

Tch, A. (2017). The mostly complete chart of neural networks, explained. towardsdatascience.com. Retrieved from `https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464`

TensorFlow. (2019a). Better performance with the tf.data api. tensorflow.org. Retrieved from `https://www.tensorflow.org/guide/data_performance`

TensorFlow. (2019b). Tensorboard: Tensorflow's visualization toolkit. tensorflow.org. Retrieved from `https://www.tensorflow.org/tensorboard`

Wan, Q., Kaszowska, A., Panetta, K., & Taylor, H. (2019, 01). Enhanced head-mounted eye tracking data analysis using super-resolution. *Electronic Imaging*, *2019*, 647-1. doi: 10.2352/ISSN.2470-1173.2019.3.SDA-647

Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004, April). Image quality assessment: From error visibility to structural similarity. *Trans. Img. Proc.*, *13*(4), 600–612. doi: 10.1109/TIP.2003.819861

Wang, Z., Chen, J., & Hoi, S. C. H. (2020). Deep learning for image super-resolution: A survey. *IEEE transactions on pattern analysis and machine intelligence.*

Wood, E., Baltrušaitis, T., Morency, L.-P., Robinson, P., & Bulling, A. (2016). Learning an appearance-based gaze estimator from one million synthesised images. In *Proceedings of the ninth biennial acm symposium on eye tracking research and applications* (pp. 131–138).

Zhang, X., Chen, Q., Ng, R., & Koltun, V. (2019). Zoom to learn, learn to zoom. In *2019 ieee/cvf conference on computer vision and pattern recognition (cvpr)* (p. 3757-3765).

Acronyms

AI	Artificial Intelligence.
ANN	Artificial Neural Network.
CNN	Convolutional Neural Network.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
CV	Computer Vision.
DAN	Deep Adversarial Networs.
DFF	Deep Feed Forward.
DL	Deep Learning.
FC	Fully-connected.
FCNN	Fully Convolutional Neural Network.
GAN	Generative Adversarial Network.
GI4E	Gaze Interaction for Everybody.
GPU	Graphics Processing Unit.
GUI	Graphic User Interface.
HR	high-resolution.
IDE	Integrated Development Environment.
IDS	Intrusion Detection System.
ILSVRC	ImageNet Large Scale Visual Recognition Challenge.
IR	Infrared.
LR	low-resolution.
MB	Motherboard.
ML	Machine Learning.
MOS	Mean Option Score.
MSE	Mean Squared Error.
NLP	Natural Language Processing.
OS	Operative System.
PSNR	Peek Signal-to-Noise Ratio.
RAM	Random Access Memory.
ReLU	Rectifier Linear Unit.
ResNet	Residual Network.
RNN	Recurrent Neural Network.
ROI	Region of Interest.
SDM	Supervised-Descent-Method.
SGD	Stochastic Gradient Descent.
SR	Single image super-resolution.
SRCNN	Super-resolution Convolutional Neural Network.

SRGAN	Super Resolution Generative Adversarial Network.
SRResNet	Super-resolution Residual Network.
SSD	Solid State Drive.
SSIM	Structural Similarity.