POLITECNICO DI TORINO

Corso di Laurea Magistrale
In
Ingegneria del Cinema e dei Mezzi di Comunicazione

Tesi di Laurea Magistrale

“Cinematographic Shot Classification through Deep Learning”

Relatori
Professor Riccardo Antonio Silvio Antonino
Professoressa Tania Cerquitelli

Candidato
Bartolomeo Vacchetti

Anno Accademico 2018/2019

INDEX

DECLARATION OF INTENT

I started to work on this project about a year ago, when discussing with prof. Riccardo Antonino about some possible applications of artificial intelligence techniques in our field. Riccardo Antonino, besides being professor of the “Special Effects” course at the Polytechnic University of Turin, is also the coordinator of Robin Studio srls, a creative studio located in Turin. In particular, we were speculating on possible ways to save some time and speed up the production pipeline of a video. One of the most frustrating waste of time was the time spent looking at all the video files in order to find the next one to use in the video sequence. Back then, I was already aware of the power of deep learning applied to image processing, but I didn’t know how it worked. We started to take a deeper look at some applications with neural networks and we realized that they were used to classify objects inside images, but not images themselves. So we decided to start this experimental thesis.

This thesis is divided in 5 chapters. The first chapter starts with a quick introduction on machine learning, as a branch of weak AI. Then it digs deeper into the theory behind machine learning. After that there is an excursus on the different types of machine learning.

The second chapter focuses on neural networks, a particular class of machine learning algorithms which is able to perform deep learning. The first part of the chapter explains how a neural network works once its training phase is done. The second part of this chapter is about how actually a neural network learns.

The third chapter is about Convolutional Neural Networks, which is the type of neural network used in this thesis.

The fourth chapter focuses on the work done for this thesis. The goal was to create a convolutional neural network able to classify images into *film shots*[1]. The first part is a quick explanation on the different kinds of shots, while the second part focuses on the creation of the dataset used to train the neural network and on explaining why it was built the way it is. The third part of the chapter focuses on the convolutional neural network created for this thesis.

The fifth and final part of this thesis contains an explanation of a possible use of this application with the author’s conclusions.

---

[1] The shot word has different meanings depending on the context. In video production a shot is a sequence of frames that runs for an uninterrupted period of time, while in video editing for instance is a continuous footage. Here the term shot is intended as the composition of the image.

CHAPTER 1: INTRODUCTION

Artificial Intelligence

In the last few decades there have been so many technological innovations that many scholars started to refer to this historical period as the “Digital Revolution”. One of the most promising fields of the digital revolution is Artificial Intelligence, from now on AI. Although there are many interesting subfields of AI, usually when it is talked about, what is meant is machine learning, even though the person speaking may not know this[2].
Machine learning is a subfield of AI, and right now it has been applied to many different fields, from medicine to stock markets. The first studies were carried out back in the fifties, but back then it wasn’t much of a success, due to the lack of sufficient computational power of the first computers and also due to the skepticism regarding neural networks, the most powerful tool of machine learning.
Machine learning gained popularity once again in the nineties. The main factors responsible for the return of machine learning were the more powerful computers, the rediscovery of backpropagation, an algorithm that is still used to train artificial neural networks following a gradient based optimization algorithm, and the change in the goal of machine learning, which shifted from obtaining true AI to solving tasks of practical nature.
What allowed machine learning to flourish further were the even more powerful computers of the last decades and the huge amount of data available in these years thanks to the Internet.
Among the multitude of fields that exploits machine learning to do research there is the image processing field (a field from which machine learning borrowed a few techniques in the past). The most common use of machine learning algorithms applied to image processing is the identification and classification of objects inside an image, such as

[2] Actually when people talk or write about AI there is a lot of confusion, so it is necessary to clarify a little.
First of all it must be said that there are two kinds of AI, “weak” AI and “strong” AI.
Strong AI, also referred to as true AI, for now is just theoretical and it would imply machines capable of a kind of intelligence similar, if not superior, to human intelligence. This kind of intelligence would be typical of machines capable of formulating thoughts, having a conscience and being able to understand and process semantic concepts.
Weak AI, also referred to as narrow AI, is the actual AI, which is capable of solving specific tasks that are too difficult for humans because they require processing an enormous load of data. Although weak AI is capable of solving problems that are beyond the computational capacity of humans, it is not “intelligent” in a true sense, and that it is why it is called weak.

numbers, letters, animals, people and so on, using a particular kind of neural networks, convolutional neural networks.
The purpose of this thesis is to use a convolutional neural network, also known as CNN, in order to classify not objects inside images but images themselves, using a partition similar to the frame partition used in the movie industry (close up, extreme close up, long shot, medium shot etc...).

Machine Learning

The machine learning algorithms are algorithms capable of learning knowledge in order to solve a specific task without human intervention. Substantially machine learning exploits a series of techniques in order to improve the performance of an algorithm in performing a specific task. These techniques come from many different scientific fields, such as data mining, image processing, computational statistics, adaptive algorithms, pattern analysis and so on…
In order to perform well in its task, a machine learning algorithm needs a huge amount of data to be trained on, which, especially nowadays, is not too difficult to find. Once that the algorithm is ready and the dataset has been chosen, the algorithm analyzes the dataset in order to recognize and exploit the patterns hidden inside the analyzed data.
Thanks to recognized patterns machine learning algorithms are able to do predictions based on the models that they have learned through the data.
The machine learning's application fields are endless, infact its application are used not only in a scientific context, but also in a more mundane one. In Sebastian Raschka[3] own words:

"*Thanks to machine learning, we enjoy robust email spam filters, convenient text and voice recognition software, reliable Web search engines, challenging chess players, and, hopefully soon, safe and efficient self-driving cars.*"

A way to understand why machine learning algorithms are so powerful is to compare them with classic algorithms. When a classic algorithm is coded it is provided with some some rules, it receives data as input and as output it returns answers. On the other hand, a machine learning algorithm receives as input data and answers and it returns as output the rules.

---

[3] Sebastian Raschka is a machine learning researcher working as Assistant professor of Statistics at the University of Wisconsin-Madison. He is the author of "Python Machine Learning" a bestselling title at Packt and on Amazon.com, which received the ACM *Best of computing* Award in 2016.

Even if there are different types of machine learning (and also different ways to catalog them), the main idea beside the different types is the same and it is what allows machine learning algorithms to “understand” the rules from just data and answers. This idea can be interpreted as an application of Bayes’ Theorem.

Bayes theorem

Let’s take two random events A and B. The basic form of Bayes’ theorem is the following:

$$P(A|B)P(B) = P(B|A)P(A)$$

or alternately

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

which reads “the likelihood of the event A happening given that B is true is equal to the likelihood of the event B happening given that A is true multiplied by the probability that A is true divided by the probability that B is true”.
So before going ahead, let's take a look at some nomenclature. P(B) and P(A) are called prior or marginal probabilities because the events A and B don’t influence each other. On the other hand P(A|B) and P(B|A) are the posterior or conditional probabilities because in this case the events have a certain influence (see Appendix).
By using the hypothesis H and the data D instead of A and B respectively the resulting equation is

$$P(H|D) = \frac{P(D|H)P(H)}{P(D)}$$

which reads like “the likelihood, given that the data D is true, of a hypothesis H is equal to the likelihood of D, given that H is true, multiplied by the probability H to be true divided by the probability of D”.
Now suppose that there are more dataset D_1, D_2 … D_n then P(H|D) becomes

$$P(H|D_1, D_2, ...D_n)$$

And the whole equation becomes

$$P(H|D_1, D_2, ...D_n) = \frac{P(D_1,D_2,...D_n|H)P(H)}{P(D_1,D_2,...D_n)}$$

Since the different datasets are different one from another and since they are independent, it is possible to rewrite the equation as follows:

$$P(H|D_1, D_2, ...D_n) = \frac{P(D_1|H)P(D_2|H)...P(D_n|H)P(H)}{P(D_1,D_2,...D_n)}$$

The numerator on the right side of the equation can be rewritten as:

$$P(D_1|H)P(D_2|H)...P(D_n|H)P(H) = \prod_{i=1}^{n} P(D_i|H)$$

,

In order to continue it is necessary to find the argument that maximizes the likelihood of the hypothesis H over the data D. The maximization of the likelihood function is equivalent to the maximization of the logarithm, so the equation now is:

$$\arg\max_h P(H|D) = \arg\max_h \ \log(P(H)\prod_{i=1}^{n} P(D_i|H))$$

It is important to notice that $P(D_1,D_2...D_n)$ can now be omitted since what needs to be found is the argument that maximize the likelihood of H over the data D, hence $P(D_1,D_2...D_n)$ is irrelevant. The next step consists of exploiting a handy logarithm property.

$$log(a * b) = log(a) + log(b)$$

By applying such logarithm property the equation becomes:

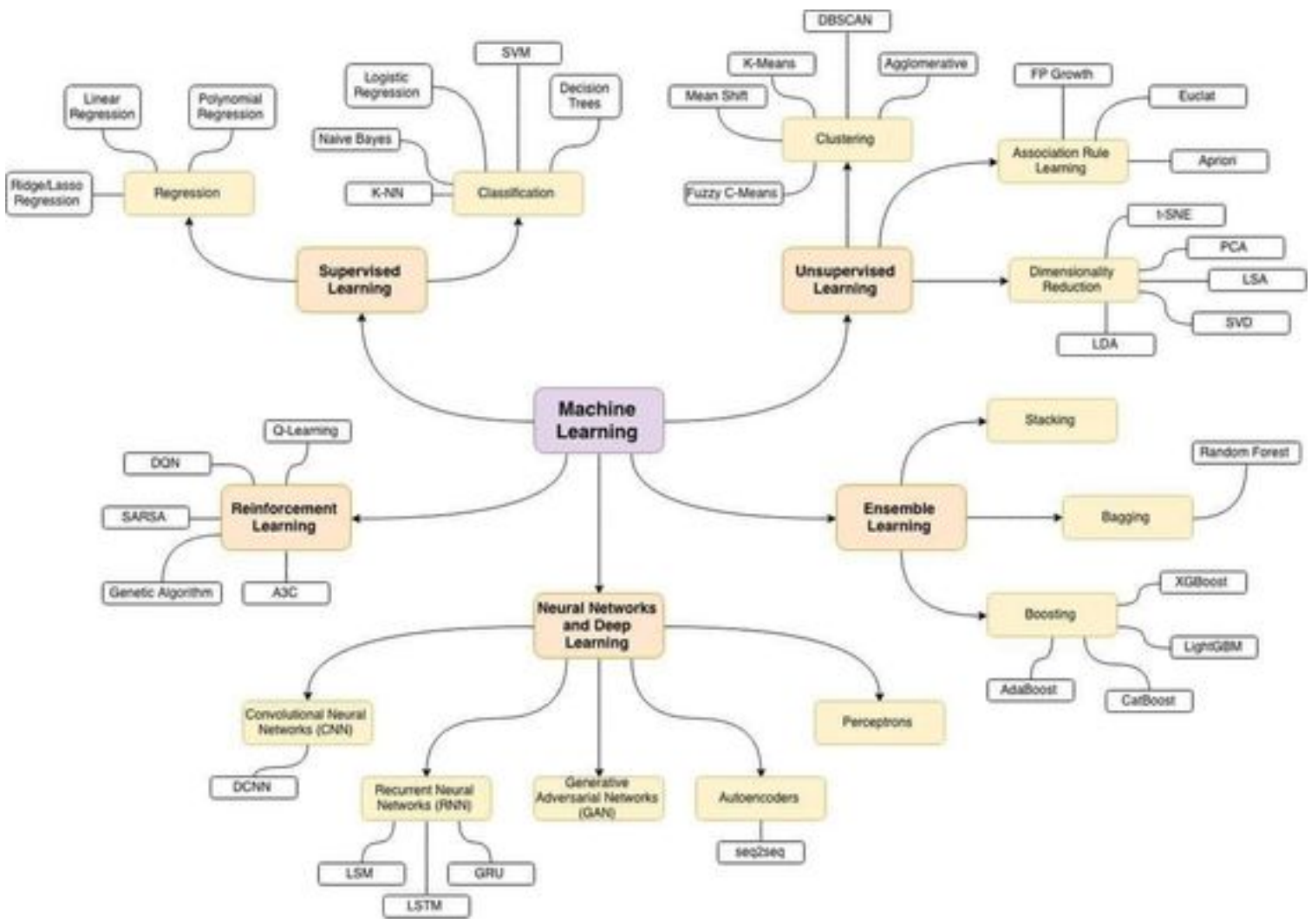
$$\arg\max_h P(H|D) = \arg\max_h \ \log(P(H) + \sum_{i=1}^{n} log(P(D_i|H)))$$

The higher the number n of different datasets considered the more log(P(H)) gets irrelevant and for a sufficiently large n it can be neglected.
All of this to say that, given a large enough dataset, it is possible to find a statistically reliable hypothesis without having to resort to a priori information on the data, which is quite often difficult to have.
In short it can be said that machine learning, from now on ML, is an algorithmic tool to learn approximate models from the data. The level of accuracy with which this can be

achieved depends critically on the quality and on the quantity of the data, and on the complexity of the underlying models one wants to infer. Given a space of possible models, the ML algorithm learns the parameters that identify the optimal model. Once this is done, the model itself can be used to make predictions on new data.

The Main Types of Machine Learning

ML algorithms, which have been designated to deal with different types of data, have three basic types: supervised learning, unsupervised learning and reinforcement learning. In addition to those three types there are ensemble learning and neural networks. These last two types combine various techniques from the three basic types. As a result, neural networks and ensemble learning algorithms can solve tasks much more complex than those solved by the basic types.



Supervised Learning

A supervised learning algorithm is called in this way because when it learns it already knows what it is going to learn. That is because the data that the algorithm receives during its training phase is labelled. This means that the desired output signals (labels) are already known, so what supervised learning algorithms do is to learn to map the inputs (data) to the outputs (labels). Once the algorithm has passed the training phase, it will have created a model based on which it will be able to make predictions. So when it receives new unlabelled data it will be able to recognize what it is, hopefully with a good level of accuracy.
Within the supervised learning field there are two subfields that are classification and regression. Classification is used to make predictions on discrete outputs while regression is used to make predictions on continuous outputs.
Classification, as the name suggests, consists of classifying the data into classes. A classic example of a classification algorithm are spam filters. Spam filters exploit the information recovered from the past spam in order to understand in its own way what is spam and what is not and then they are able to separate the spam from the regular emails. This kind of task is a classification task, more precisely a binary one, because the mail can either be spam or not-spam.
Regression[4], on the other hand, deals with predictions on continuous output. Suppose that there is one dependent variable and one or more independent variables, regression is useful in order to find a relationship between the independent variables and the dependent one. For instance, if someone wants to predict how much his sleeping hours affect his test scores, the sleeping hours are the independent variables and the test score is the dependent variable.
The algorithm developed for this thesis is based on a classification problem since its task is to understand which images belong to a certain shot class and which belong to other classes.
A fundamental quantity for machine learning algorithms is the accuracy, a percentage which describes the fraction of data that are correctly classified. Thanks to accuracy the algorithm is capable of understanding if its performance is good or bad.

Unsupervised Learning

The main difference between supervised and unsupervised machine learning algorithms is that the latter receives unlabeled data. This implies that, since there are no class

[4] The correct term is regression analysis, but regression is fine anyway.

labels, the unsupervised machine learning algorithms cannot rely on accuracy to improve their performance. So, during the training phase these algorithms receive unlabelled data and exploit the data structure in order to group the data into different classes.

The subfields of unsupervised learning are clustering, association and dimensionality reduction.

The clustering algorithms divide the data received based on similarities and differences, so similar objects will belong to the same class. In other words the task of clustering algorithms is to divide n samples in k partitions, where k is minor or equal to n(the latter is very rare). There are two ways to approach clustering: the top-down approach and the bottom up.

For instance, if someone has a dataset with height and weight of a certain amount of people, but it does not know the gender, he or she can use a clustering algorithm in order to separate the data. In this scenario there are going to be two groups, the first group will have people heavier and taller, while the second group will have shorter and lighter people. Although it is not a hundred percent correct, it can be inferred that the first group is mainly made of men, while the second is mainly composed of women.

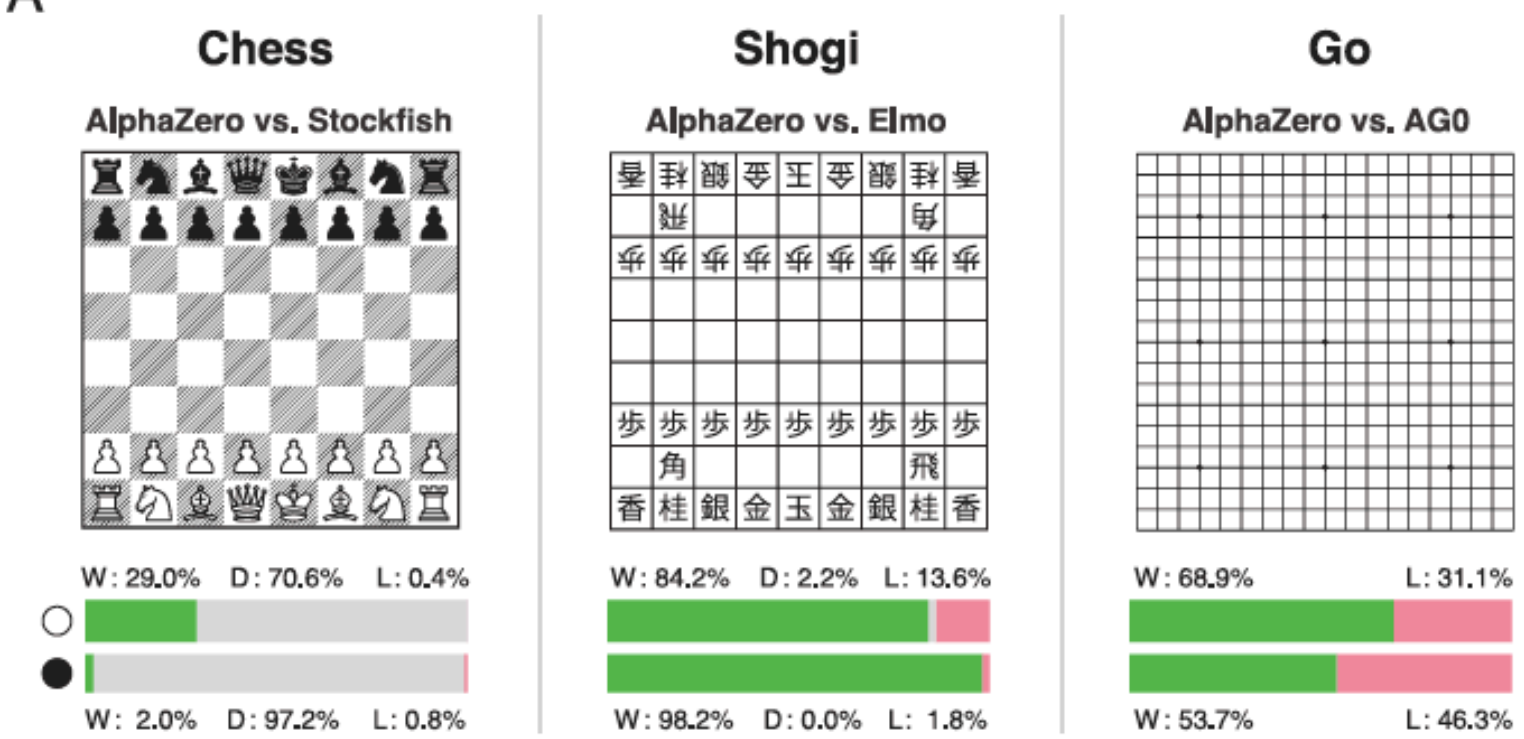
Another rather interesting subfield of unsupervised machine learning is dimensionality reduction. Dimensionality reduction comes in handy when the data to feed to an algorithm has a high dimensionality. Substantially it allows to find where there is the highest variance inside the data, which means also the in which dimensions there are the most useful information. It also allows to remove noise from the data that in some cases can be really annoying in order to make predictions. Usually dimensionality reduction is used to treat the data before feeding them to other machine learning algorithms, such as supervised machine learning algorithms or neural networks.

The last subfield of unsupervised learning is association rule learning. This subfield looks for significant relationships between values inside a database. Association rules were introduced at first in market analysis, and then later in data mining, bioinformatics, intrusion detection and other fields. If there is someone that wants to organize his shop he usually uses association rules to organize his shop, even if he might not know. For instance, if he sells beds it is likely that he will put bed sheets close to where he sells beds. He does that because he knows from his experience that if someone buys a new bed it is likely to buy also sheets. The association rule learning does the same looking at the record sales only on a much bigger database with millions of entries and can find relations between the products that a human would not notice.

Reinforcement Learning

Reinforcement learning works in a different way compared to supervised and unsupervised learning. Even if, like the other two, it has a training phase, its training phase is different from the others. Basically the purpose of reinforcement learning is to create an algorithm that improves its performance by interactions with the environment. In order to do that these kind of algorithms use a reward function which measures how well a certain action or a certain set of actions performs. Thanks to the interactions with the environment and the reward function the reinforcement learning algorithms are capable of elaborating complex and original strategies.
One of the most notorious companies that has developed a reinforcement learning algorithm is DeepMind, a company bought by Google in 2014. The company created AlphaGo, followed then by Alpha Zero, an algorithm capable of playing games and win against humans. AlphaGo is a machine learning algorithm capable of playing and defeating humans in the chinese game of Go, while AlphaZero is capable of playing not only Go but also Shogi(a Japanese game similar to chess) and normal chess.
These algorithms, AlphaZero especially, are capable not only of defeating human champions but they are also capable of defeating other algorithms developed in the past years, that defeated the human champions in the respective game.
During the training phase these algorithms have the sole purpose of winning, so the training phase starts with an untrained neural network that start to play against itself. At the beginning the algorithm moves are random, but after a while it learns from the wins and the losses and alters the neural network in order to choose better moves game after game. For AlphaZero the training phase lasted 9 hours for chess, 12 hours for Shogi and 13 days for Go, which is by far the most complicated game with 2,08x10^(170) legal board positions (and that is just the lower bound).
What is extraordinary is that the style adopted by the machines is very unpredictable and that brings the machine to use strategies that to human eyes seems reckless or just stupid (like use the king in an aggressive way in the game of chess). But even if these strategies seem reckless at first, in the end AlphaZero is the winner, as shown in the graphic below.

A



The black dot and the white dot represent respectively the color of the pawns used by AlphaZero, while W stands for wins, D stands for draws and L stands for losses.

Ensemble Learning

The basic idea behind ensemble learning is to use multiple learning algorithms at the same time to perform a task in order to obtain a better performance. So the same data are fed to the different learning algorithms. By doing so every different model will produce a slightly different prediction. These predictions are then combined into a final prediction that is more accurate than the single predictions.

For instance, if someone wants to create an ensemble learning algorithm in order to recognize numbers inside an image he can use a *linear regression*, a *support vector machine* and a *decision tree* all together. If all of these three algorithms have an accuracy of 60% after the training phase, which is not hard to obtain, the final accuracy is almost 65%, if the three algorithms have the same decision power (the demonstration is in the Appendix).

Ensemble learning is useful to reduce errors related to bias and variance and errors in general.

Ensemble learning algorithm can be used for both classification and regression tasks.

There are three subfields of ensemble learning: Bagging, also known as bootstrap aggregating, Boosting and Stacking.

The bagging subfields consist of training multiple models of the same learning method with different subsets taken by the main dataset. An example is the *random forest* model which is made of many decision tree algorithms. The important thing is that every algorithm has the same weight when a prediction is made.
Boosting is similar to bagging with one significant difference, that is in the final decision every model has a different weight. The weight of every algorithm is determined by its accuracy, the higher it is the more the weight of that algorithm grows. However, even if every model has a different weight, all the models are of the same kind.
Stacking differs from bagging and boosting for two reasons. The first one is that, while boosting and bagging use homogeneous models, stacking uses heterogeneous models. The second main difference is that stacking combines the predictions of the singular algorithms by using a meta-classifier, such as a neural network.

Neural Networks

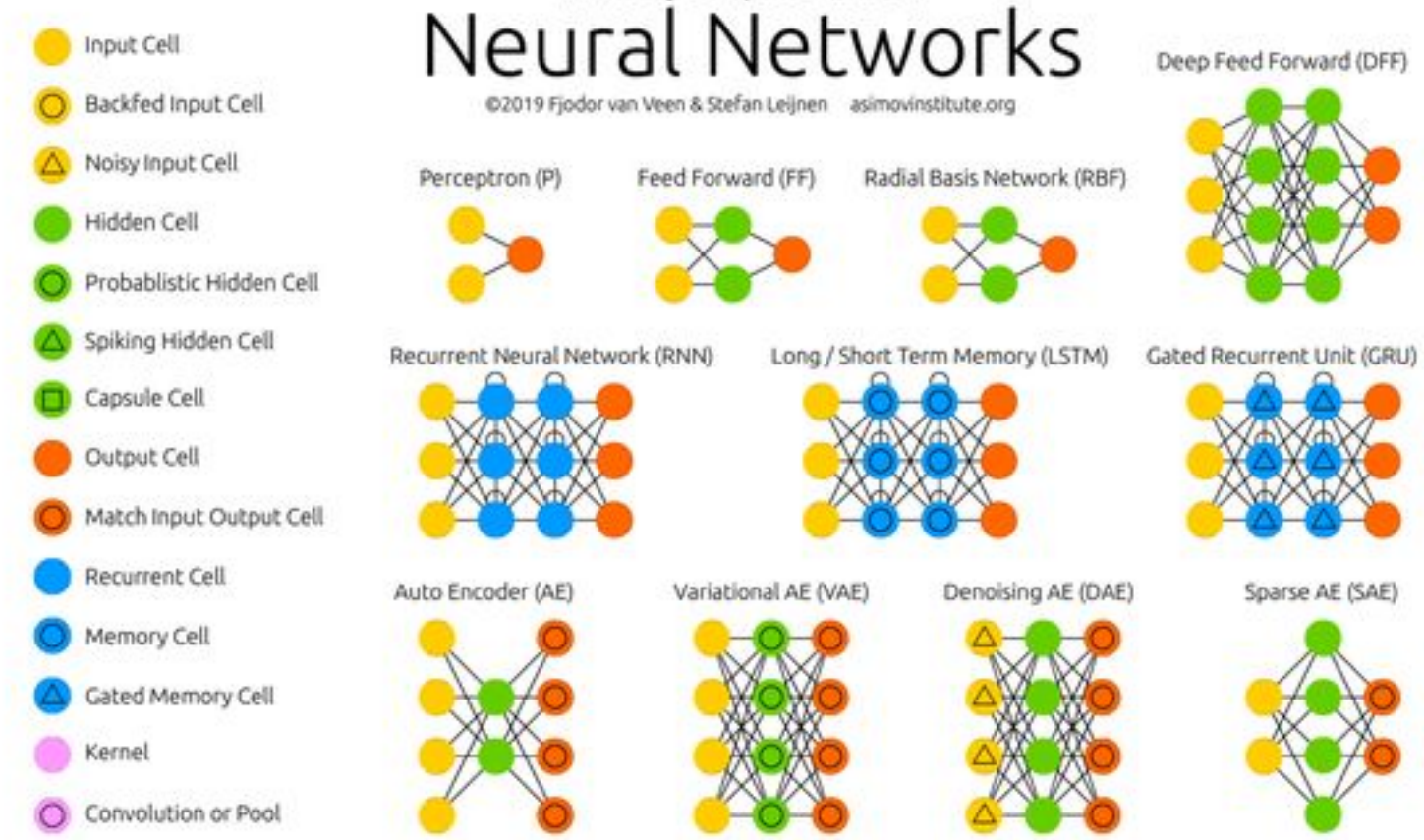
Neural networks are one of the most powerful tools of machine learning. What makes neural networks so powerful is their ability to perform *deep learning*. The term deep refers to the fact that neural networks have multiple layers of neurons. Each layer elaborates the information received from the previous layer and passes it as input to the following layer.
Neural networks are computational models built on the footprint of biological neural networks, also known as brains.
Like any brain, neural networks have their own neurons, even though their neurons are artificial and less sophisticated than actual neurons.
There are many different ways to connect the artificial neurons of a neural network, and each different configuration exploits different techniques and it is used for a different purpose.
The main subfields of neural network are the following: Perceptrons, Convolutional Neural Network (CNN), Recurrent Neural Networks (RNN), Generative Adversarial Networks (GAN) and Autoencoders.
Since Multilayer Perceptrons and Convolutional Neural Networks will be analyzed in a deeper way later for now there will be a quick excursus on the other three types.
Recurrent neural networks are able to save temporal information. This perk of theirs allows them to receive a series of input, where each input is related to the previous input and to the following input. So recurrent neural networks remember things learned from the training phase, like any other neural network, but they are also capable of understanding what kind of relationship there is between an input and the other inputs.

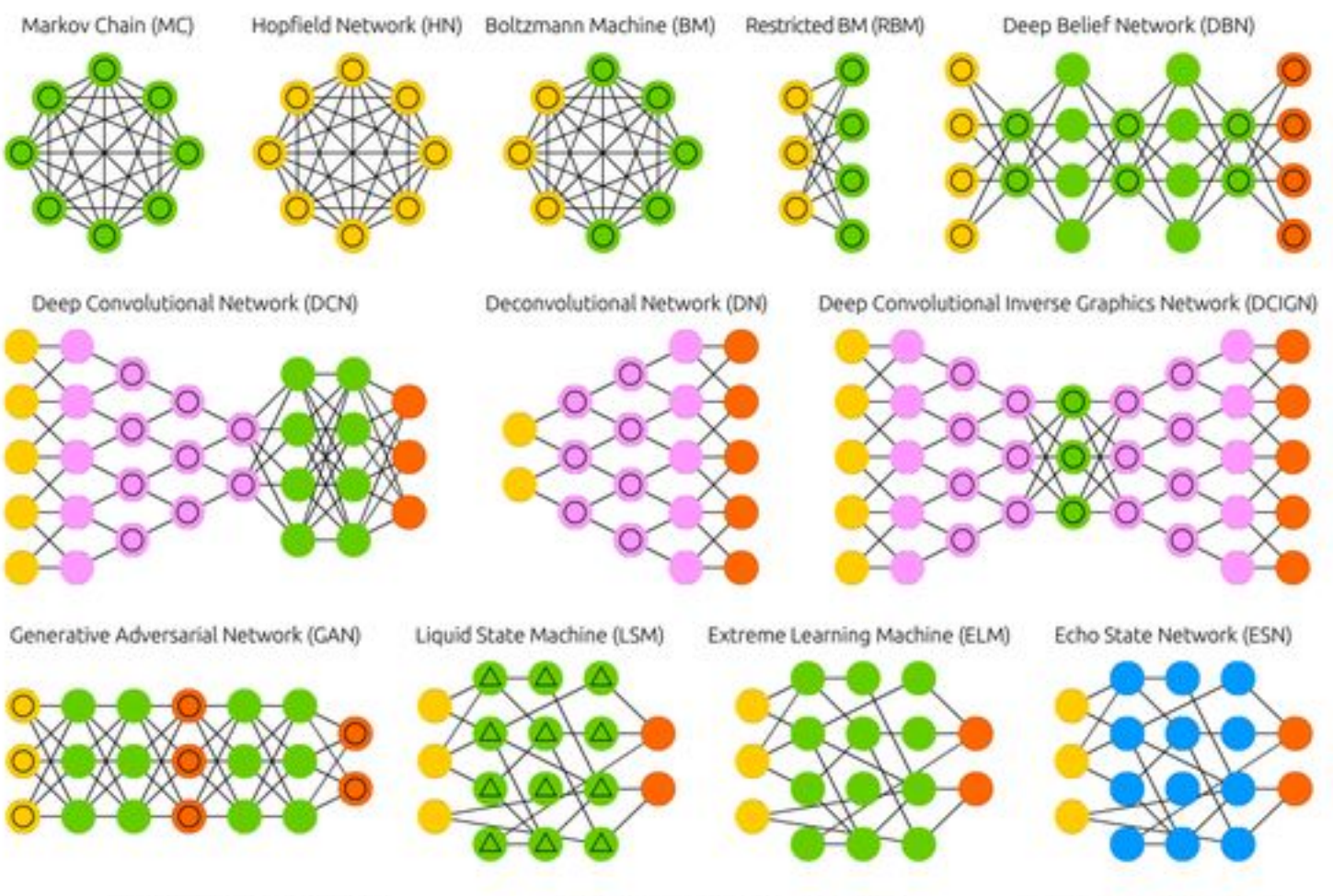
A mostly complete chart of

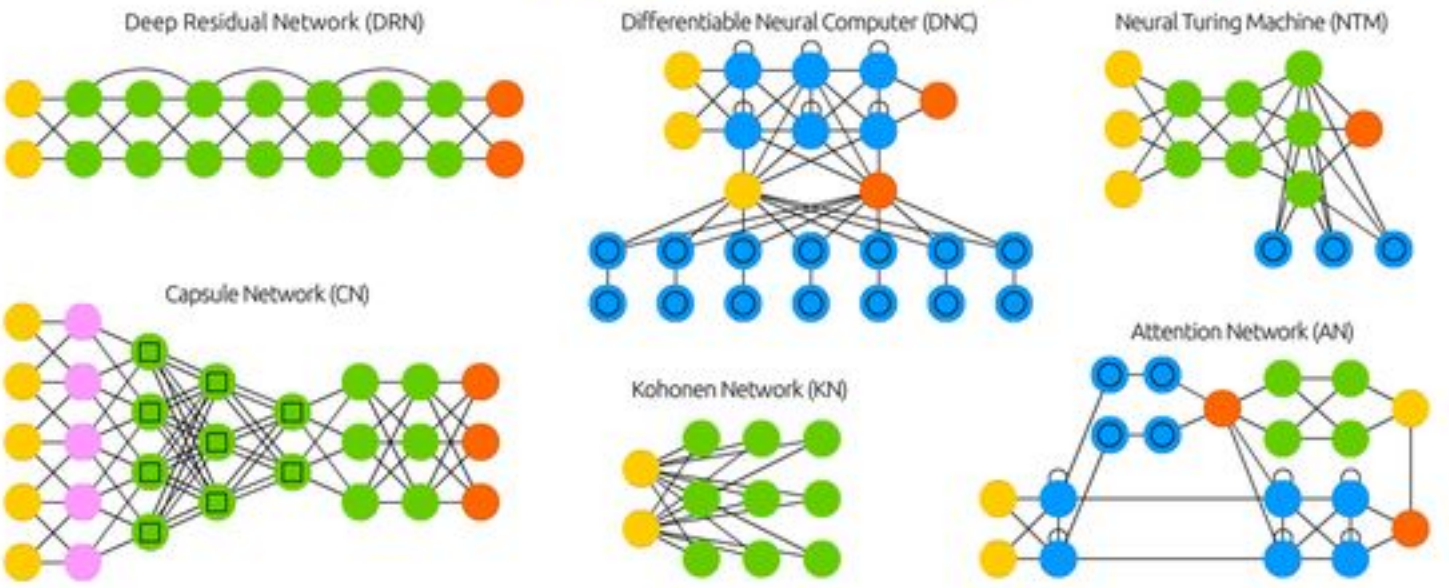
Neural Networks

 asimovinstitute.org

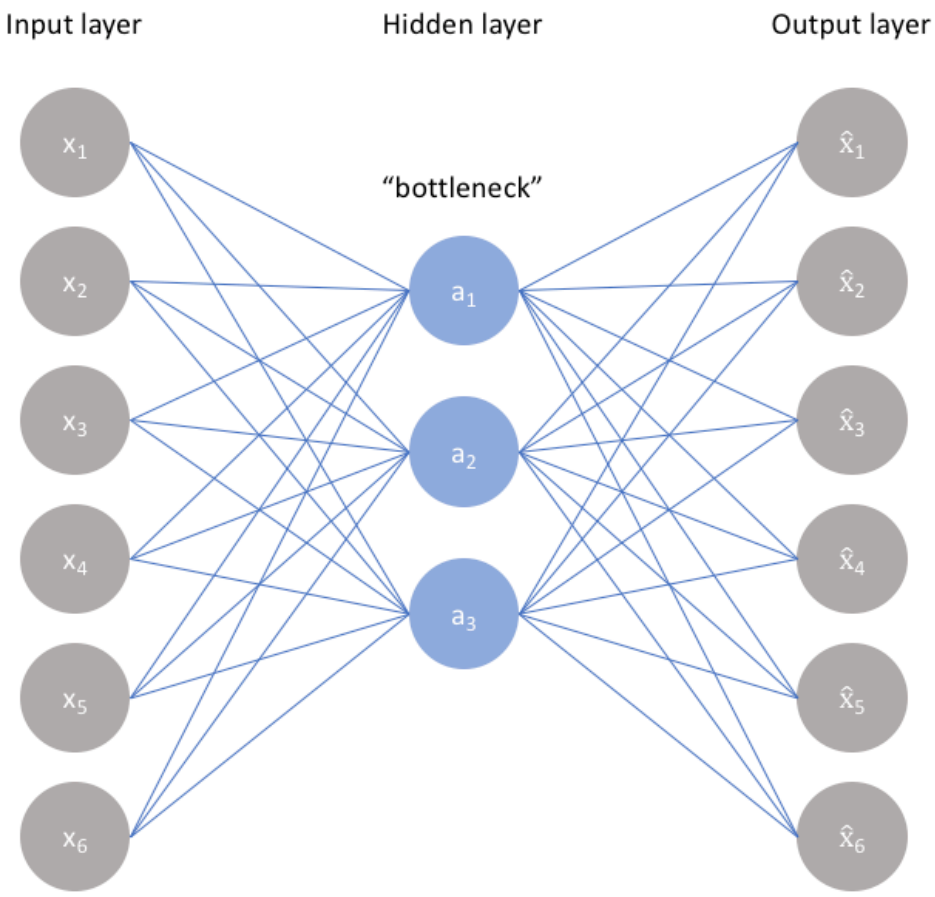
- Input Cell
- Backfed Input Cell
- Noisy Input Cell
- Hidden Cell
- Probablistic Hidden Cell
- Spiking Hidden Cell
- Capsule Cell
- Output Cell
- Match Input Output Cell
- Recurrent Cell
- Memory Cell
- Gated Memory Cell
- Kernel
- Convolution or Pool







For this reason recurrent neural networks are widely used in speech voice recognition, because they are the most suitable to capture the relations between vowels and consonants.
Autoencoders are usually used for dimensionality reduction. They are neural network that exploits unsupervised learning. Their structure can be summarized by the following picture.



Basically the output of an autoencoder is a replica of the input. During the training phase autoencoders learn to recreate the input as output using the information contained in the bottleneck layer. Since the net contains the original data it can check automatically its performance in recreating the original output. Autoencoders are peculiar because in their case the interesting part is not the output layer, but the bottleneck layer instead. When an autoencoder is fully trained the bottleneck layer contains all the information needed by the neural network to rebuild the original image.
Thanks to this ability to recreate the original data they are used for eliminating signal noise for instance.
Generative Adversarial Networks are maybe the neural networks with more potential. They have proven to be capable of unsupervised learning, supervised learning, reinforcement learning and even semi supervised learning[5]. The potential of generative

[5] Semi supervised learning is a class that falls between supervised learning and unsupervised learning. During the training phase the dataset is composed of a small minority of labeled data and a majority of unlabelled data.

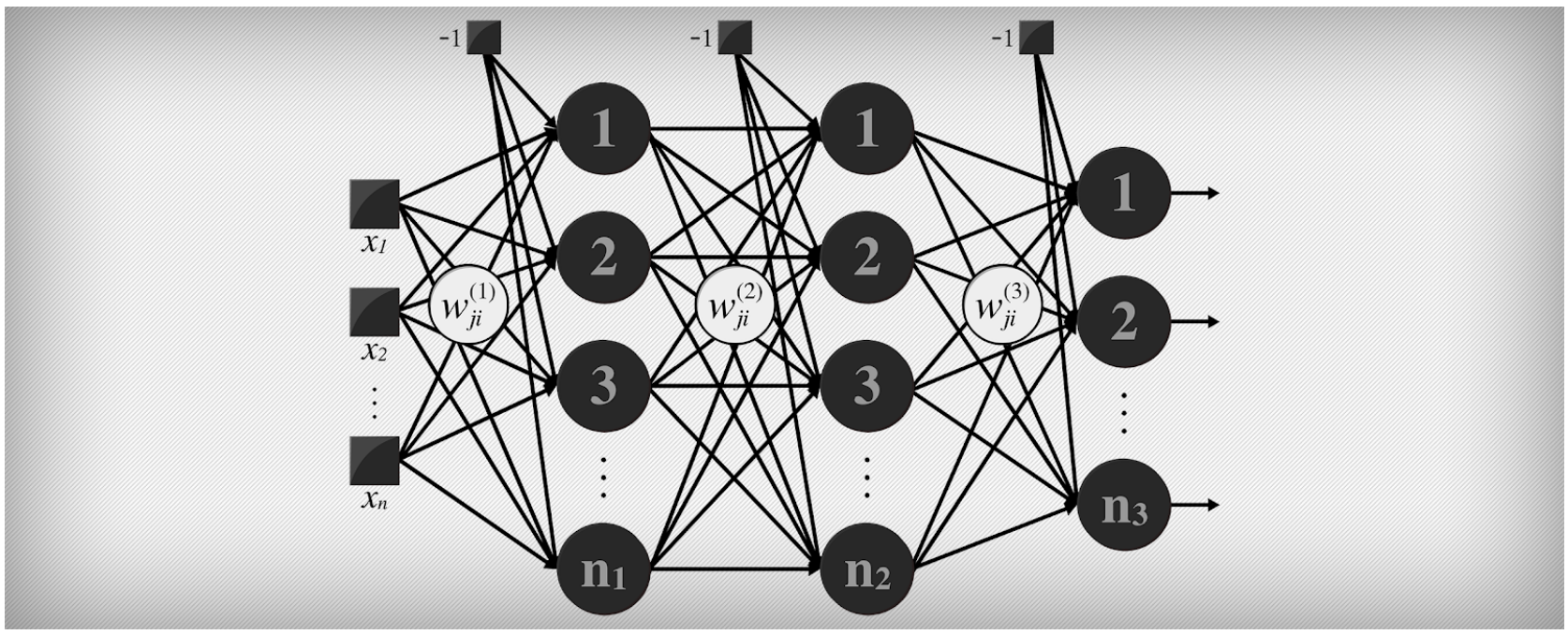
adversarial networks lies in their capacity to mimic any distribution of data, from images to music. The architecture of a gan is made of two neural networks competing one against the other in a “game”. The aim of both of the networks is to outperform the other one. In order to better understand how a gan works let’s look at a *srgan* (super resolution gan). This particular gan receives as input a low resolution image and returns as output the same image but with high resolution. The following is a synthetic explanation of how it works. One of the two neural networks is trained to give higher resolution to an image by modifying it, while the other neural network is trained to recognize whether an image is real or not. In this way both the neural networks, if built and trained properly, keep improving their performance to a point at which humans are not able to understand if the image is true or not.

This part concludes the first chapter. If the reader is interested in the Appendix there is a paragraph regarding some applications that exploit ML techniques in order to solve image processing tasks.

CHAPTER 2 : MULTILAYER PERCEPTRON

How a Neural Network Works

The *multilayer perceptron* is a *deep feedforward neural network*[6] and its neurons are the *perceptrons*[7]. The multilayer perceptron is a deep network used for classification, so this network uses supervised learning.



The image above shows a generic structure of a multilayer perceptron. The first layer (x_1,x_2,...x_n) is called input layer and it has as many nodes as the number of features of the data to be fed to the neural network. For instance, in this case study, since the images that the algorithm is working with have a resolution of 160x90 pixels, the first layer has 14'400 nodes, one for each pixel of the image.
The last layer is the output layer and it has as many nodes as the number of different classes in which the data is divided. For instance, if a neural network is working with the MNIST dataset, a database of handwritten digits, and has to recognize the numbers inside the images, the output layer would have 10 neurons, one for each digit from 0 to 9. In this case, as previously stated, since it was decided to classify the images into four film shot classes the output layer has 4 nodes.
The layers between the input layer and the output layer are called hidden layers and their number is not fixed beforehand. Also the number of nodes inside each hidden

[6] A feedforward neural network is a network in which the connections between the nodes of the different layers do not create a cycle.
[7] The perceptrons are binary classifiers.

layer is not fixed beforehand. In order to understand how many layers and how many nodes are needed, it is necessary to check which configuration gives the best results.
Every node[8] has a numerical value that is rescaled between 0 and 1 that is called activation. In the neural network used for this thesis the activation value of the nodes inside the input layer corresponds to a value on the gray scale.
Every activation influences the activation of the neurons of the following layer. The same thing happens inside of a brain, in fact when a neuron fires, other neurons fire as well.
So when an image is fed to a neural network, it activates some neurons that in turn activate other neurons and so on until the signal reaches the output layer. At the output layer every neuron will have a different activation value and the one with the highest numerical value represents the class that will be chosen by the neural network as the correct one.
In this kind of neural network every neuron of a layer is connected to every single neuron of the following layer[9] through the weights[10]. The value of the weights, which is also scaled between 0 and 1, is determined during the training phase.
So the activation value, from the first hidden layer onwards, is given by the following equation

$$a = (w_1 a_1 + w_2 a_2 + ... + w_n a_n)$$

In order to avoid the math to grow too much it is preferable that the resulting value is between 0 and 1, so all of this goes inside a sigmoid function or inside a relu (rectified linear unit). Let's see what happens with a sigmoid.

$$\sigma = \frac{1}{(1+e^{-x})}$$

So the equation becomes

$$a = \sigma(w_1 a_1 + w_2 a_2 + ... + w_n a_n)$$

---

[8] Here node, neuron or perceptron refer to the same object.
[9] These kind of layer is called *fully connected layer*.
[10] In the image above the weights are represented by the lines connecting the nodes.

The last unit to consider is the bias which is represented in the image above by the gray squares over the hidden layers and the output layer. The bias is a threshold that prevents the activation of a neuron below a certain value. So our equation for the activation value of a single neuron becomes

$$a = \sigma(w_1 a_1 + w_2 a_2 + ... + w_n a_n + b)$$

This formula provides the activation value for only a single neuron. When considering a whole layer the last formula becomes

$$a_0^1 = \sigma(w_{0,0} a_0^0 + w_{0,1} a_1^0 + ... + w_{0,n} a_n^0 + b_0)$$

This may look confusing, therefore before proceeding, let's see what all of these apexes and subscripts mean. On the left side of the equation there is a_0^1, which represents the first element of the second layer (the first layer is the input layer) the subscript 0 represents the number of the element considered, while the apex 1 represents the layer considered (the numbering starts from 0). On the right side of the equation there are $w_{0,0}a_0^0$ and the others. While a_0^0 follows the same rules of a_0^1 , $w_{0,0}$ represents the weight between the first element of the first layer and a_0^1 . The first subscript stands for the element being considered of the second layer and the second subscript stands for the element of the first/previous layer.
The equation for the activation values of every single element of a layer is

$$a^1 = \sigma(W a^0 + b)$$

In this scenario W is a matrix mxn of all the weights that connect the first layer to the second one, where m is the number of elements in the first layer and n is the number of elements of the second layer, while a and b are vectors with as many entries as the number of elements of the second layer.

How a Neural Network Learns

The neural network learns during the training phase. The aim of the training phase is to find the values of weights and biases that allow the neural network to classify the samples correctly. In supervised learning when the neural network receives a sample it also receives its label. If the neural network classifies the sample correctly, nothing happens, but if it classifies it incorrectly, the neural network changes the values of the weights and of the biases directly. For instance, if the algorithm is given a *close up* and it classifies it as a *close up*, nothing happens, but if it classifies it as a *half figure* the algorithm will know that it made a mistake. By changing the values of the weights, indirectly the neural network also changes the activation values of the neurons of the following layers in the network. The changes made to these values are made so as to allow the neural network to classify that sample correctly. After a certain amount of samples the neural network will have found the right values of the weights and it will be able to classify data in a satisfying way.

Once the training phase is over the neural network receives more samples with labels. These new samples are used to test the performance of the neural network on samples that it has never seen, but they don't change the values of the weights and biases anymore.

During the training phase the neural network goes through the dataset multiple times. Each time this happens, this is referred to as an *epoch*. By doing so the neural network updates the value of the weights multiple times and this allows it to find the right value of the weights.

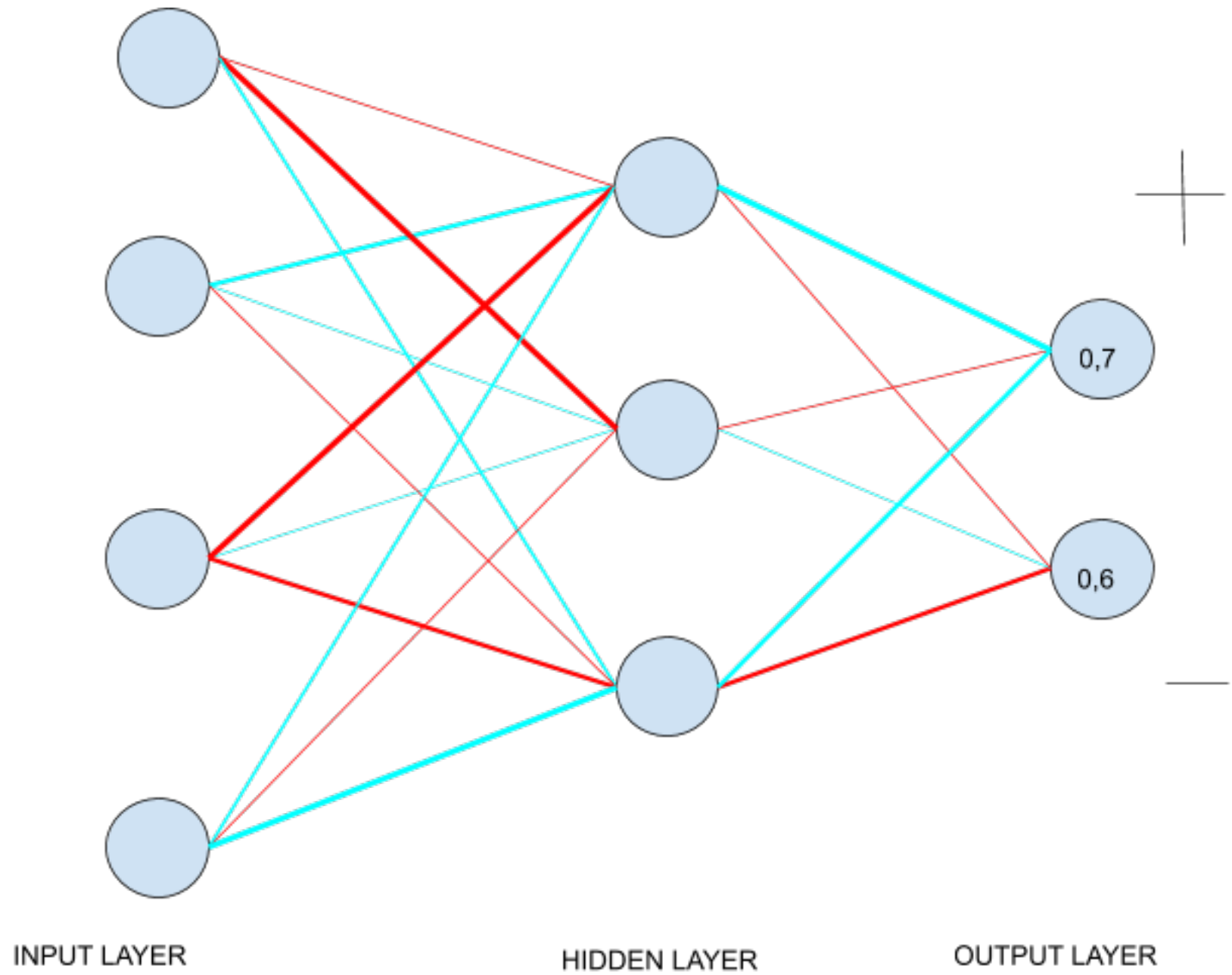
At first the accuracy of the algorithm will be pretty low, since it is basically classifying data randomly. However every time that it gets a wrong result, it starts to modify the value of the weights. The neural network is able to understand whether it is wrong or not thanks to label of the misclassified sample.

In order to evaluate the performance of the neural network, the algorithm used to create it provides it with a cost function, such as the *sum squared error* function. The cost function will provide a value at the end of every data evaluation. The higher the value of the cost function is, the worse the performance of the neural network. The cost function is essential, because it is what allows the neural network to understand how bad its classification is. The goal of the algorithm is to find the values of the weights in order to minimize the value of the cost function.

The cost function alone is necessary but not sufficient for the neural network to learn. There is another element necessary and that element is gradient descent. The cost function represents whether the neural network is doing a good classification or not,

while the gradient descent is what allows the neural network to understand how to change weights and biases in order to improve its performance.
The gradient descent is a technique that allows one to find the points of minimum and maximum of a function with multiple variables. By applying the gradient descent to the inputs of the cost function, or in other words to the weights and biases of the neural network, it is possible to see how much each weight is relevant for the prediction of the network and whether it needs to be increased or decreased.
The final piece of the puzzle regarding neural network learning is the backpropagation algorithm. Backpropagation is the technique that actually changes the values of the weights and biases of the neural network. So when the neural network classifies a sample in a wrong way it takes note of all the weights that connect the output layer with the previous layer. Then the neural network considers all the activation values of that layer and takes notes on the weights connecting that layer to the previous one and so on until it reaches the input layer. Once it reaches the input layer, the algorithm changes the weights connecting the input layer to the following layer. By doing so it changes the activation value of the neurons of the following layer and so on, until it reaches the output layer. Let's see a diagram and a simple example in order to better understand how it works; afterwards the formulas behind this concept will be shown.
Suppose that there is a neural network with the structure shown below, which is being trained in order to recognize if an image is a plus ("+") or a minus ("-"). The training phase is still at the beginning and the neural network misclassified the last sample as a plus instead of a minus. Thanks to the label of the sample the neural network recognizes its mistake, and therefore starts to take notes on which weight influenced its decision the most, and how it should change its value in order to classify the image properly. The thickness of the lines represents the magnitude of the weight, while the color indicates whether the value of the weight has to be increased or decreased. The circles are the neurons. Every neuron has its activation value, but in this case, since it is not real, the activation values have not been written. The only activation values written are those of the neurons in the output layer, simply because they indicate how sure the neural network[11] was of its decision. Once the neural network has taken note on every weight connecting the hidden layer to the output layer, it does the same thing between the hidden layer and the input layer. Since the activation values of the input layer cannot be changed, because it would imply changing the input, the algorithm changes the weights connecting it to the hidden layer. By doing this, also the activation values of the neurons of the hidden layer change by averaging the changes needed from each weight. The same thing happens between the hidden layer and the output layer.

[11] Such a neural network for image classification is very unlikely to exist, since the input layer consists of only four neurons.



This example was shown in order to better clarify what happens from a conceptual point of view when a neural network learns. Now it will be shown from the mathematical point of view.
Since the formulas for the different elements of an entire neural network are complex, at first a case will be shown consisting of a very simple neural network made of just 2 neurons and a weight connecting them, then as the neural network grows, so will the formulas.

In the case of just two neurons, the activation value of the second neuron is given by

$$a^L = \sigma(w^L a^{L-1} + b^L)$$

where a^L is the activation value of the neuron considered in the current layer, $a^{(L-1)}$ is the activation layer of the neuron of the previous layer, w^L and b^L are the weight and bias connecting the previous layer with the current layer and σ is the previously mentioned sigmoid function. The cost function for this simple neural network is given by

$$C_0 = (a^L - y)^2$$

Where y is the desired output. So the cost function is the sum squared error function. The lower the result of this function is, the closer the prediction of the neural network is to the real output. In order to understand how to minimize the cost function, it is necessary to understand how each parameter influences the cost function. It is possible to do that thanks to *the chain rule*[12]. From now on, instead of writing $w^L a^{(L-1)}+b^L$, z^L is going to be used for the sake of simplicity. The chain rule allows one to calculate the partial derivative of the cost function with respect to the weight.

$$\frac{\partial C_0}{\partial w^L} = \frac{\partial z^L}{\partial w^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial C_0}{\partial a^L}$$

Let's see what each of these terms is equal to.

$$\frac{\partial z^L}{\partial w^L} = a^{L-1}$$

$$\frac{\partial a^L}{\partial z^L} = \sigma'(z^L)$$

$$\frac{\partial C_0}{\partial a^L} = 2(a^L - y)$$

[12] The chain rule is the rule that allows the derivative of a composite function to be calculated. If the functions that constitute the composite function are derivable, then the composite function is itself derivable.

So in the end the gradient of the cost function for a single sample is equal to

$$\frac{\partial C_0}{\partial w^L} = (a^{L-1})(\sigma'(z^L))(2(a^L - y))$$

If all the samples are taken into account the previous formula becomes

$$\frac{\partial C}{\partial w^L} = \frac{1}{n}\sum_{k=0}^{n}\frac{\partial C_k}{\partial w^L}$$

And this is just for a neural network made of two neurons and one weight. For the gradient of the bias the formula is similar.

$$\frac{\partial C_0}{\partial b^L} = \frac{\partial z^L}{\partial b^L}\cdot\frac{\partial a^L}{\partial z^L}\cdot\frac{\partial C_0}{\partial a^L}$$

Furthermore, its derivative is simpler due to the fact that the derivative of $(\partial z^{(L)}/\partial b^{(L)})$ is equal to one, so the gradient of the cost function with respect to the bias is

$$\frac{\partial C_0}{\partial b^L} = (\sigma'(z^L))(2(a^L - y))$$

In order to find the sensitivity of the cost function with respect to the activation value of the previous layer the formula becomes

$$\frac{\partial C_0}{\partial a^{L-1}} = \frac{\partial z^L}{\partial a^{L-1}}\cdot\frac{\partial a^L}{\partial z^L}\cdot\frac{\partial C_0}{\partial a^L}$$

$$\frac{\partial C_0}{\partial a^{L-1}} = (w^L)(\sigma'(z^L))(2(a^L - y))$$

All of these formulas are valid for a neural network made of two neurons and one weight. However real neural networks have many more neurons, weights and biases compared to this simple example. Even if there are many more elements to consider, in reality the complexity doesn't change that much from a conceptual point of view[13]. The cost function for a singular sample becomes

$$C_0 = \sum_{j=0}^{n-1} (a_j^L - y_j)^2$$

Where j represents the jth neuron of the layer L. The z function, the one that takes into account weights, biases and activation value of the previous layer becomes

$$z_j^L = w_{j,0} a_0^{L-1} + w_{j,1} a_1^{L-1} + ... + w_{j,m} a_m^{L-1} + b_j$$

Where m indicates the number of neurons of the previous layer. So the partial derivative of the cost function becomes

$$\frac{\partial C_0}{\partial w_{jk}^L} = \frac{\partial z_j^L}{\partial w_{jk}^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial C_0}{\partial a_j^L}$$

It is interesting to see how the formula for the sensitivity of the cost function with respect to the activation values of the previous layer changes

$$\frac{\partial C_0}{\partial a_k^{L-1}} = \frac{\partial z_j^L}{\partial a_k^{L-1}} \cdot \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial C_0}{\partial a_j^L}$$

---

[13] From a computational point of view, it gets a lot heavier.

Where k indicates the kth neuron of the previous layer. This implies that the activation value of the current layer is affected by the activation values of all the neurons of the previous layer.
All of this was intended to lead to the actual formula that allows the neural network to minimize the cost function using its gradient, and this formula is

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial z_j^l}{\partial w_{jk}^l} \cdot \frac{\partial a_j^l}{\partial z_j^l} \cdot \frac{\partial C}{\partial a_j^l}$$

$$\frac{\partial C}{\partial w_{jk}^l} = (a_k^{l-1})(\sigma'(z^l))(\frac{\partial C}{\partial a_j^l})$$

Where *l* indicates the lth layer and $\partial C/\partial a_j^{(l)}$ is equal to

$$\frac{\partial C}{\partial a_j^l} = \sum_{j=0}^{n(l-1)+1} (w_{jk}^{l+1})(\sigma'(z^{l+1}))(\frac{\partial C}{\partial a_j^{l+1}})$$

A simpler way to write this formula is the following

$$\frac{\partial C}{\partial w_{jk}^l} = (a_k^{l-1})(\sigma'(z^l))(2(a_j^l - y_j))$$

All of this is necessary to determine how the weights of the neural network are changed in order to obtain a better performance out of the neural network itself. The update of the weights is described by the following formula

$$\vec{w} = \vec{w} + \Delta\vec{w}$$

Where Δŵ is equal to

$$\Delta\vec{w} = -\eta\nabla C(\vec{w})$$

Where ▽C(ŵ) is the gradient of the cost function and η is the learning rate. The learning rate is fixed a priori but it is a value that has to be chosen wisely. Since all of this is intended to find the minimum of the cost function, if the value of the learning rate is too big there is a risk of missing the actual minimum of the function. On the other hand, if the learning rate is too small, the algorithm will need too much time in order to find the value of the weights that minimize the cost function.

Another reason for choosing the learning rate carefully is the overfitting issue. When a model overfits, it means that it has a good performance when it deals with the samples from the training set, but when it deals with samples that are not from the training set, its performance is not as good, if not actually bad. This means that the neural network has found a local minimum of the cost function instead of the global minimum. With a small learning rate is a lot easier to find a local point of minima instead of the global point.

Also a learning rate which is too great can lead to overfitting, because there is a risk of surpassing the global minimum. There are many ways to handle overfitting.

One of the most simple ways is to increase the number of samples in the training set in such a way that samples from the same class are diversified. For instance, if a neural network has to classify images of cats and dogs and in its training set the images of dogs belonged only to large dogs, when it receives an image of a chihuahua, it is possible that the neural network classifies the dog as a cat due to the dog's moderate size (or even a rat maybe).

Another way to handle overfitting is to use a technique called data augmentation. Data augmentation, in the case of images, consists of applying transformation to the images of the dataset, such as rotation, zooming, cropping and so on. By doing so, the neural network has the chance to see new images belonging to the same class.

A different approach used to handle overfitting consists of using the mini-batch learning. The idea behind mini batch learning is to divide the training set into smaller sets, called batches or mini-batches. By doing so, the gradient descent will be calculated on every batch and every batch will have its weight update. Then in order to train the neural network every mini-batch gives its vote on what the best value for the weights are. In other words the gradient descent is calculated from the different gradient descent calculated from every batch. There is more than one way to decide the final values of the weights. It can be decided to average the values of the weights of every minibatch, it can be decided proportionally,where the values that receive more votes are the chosen

ones, or it can be decided that the batch that minimizes the cost function to the greatest degree is the correct one. The method chosen for choosing the values of the weights also depends on the kind of task that the neural network has to solve: it is not true that the global minimum of the cost function is always the best solution.
Before going on, here is a little summary on neural networks. Neural networks are made of an input layer, an output layer and a certain number of hidden layers, a number that depends on the kind of task that the neural network has to solve. Every layer is made of a certain number of neurons. In the case of a feed forward neural network, every neuron in a layer is connected to every neuron of the following layer. The number of neurons of the input layer corresponds to the number of features of the samples. In the case of image recognition the number of features usually coincides with the number of pixels of the images. The number of neurons of the output layer corresponds to the number of classes into which the samples will be divided. The number of neurons inside the hidden layers is determined by trial and error.
The aim of a neural network used for classification is to have a good performance. In order to evaluate its performance, the algorithm has at his disposal a cost function. When the algorithm misclassifies a sample, the cost function tells it how far the neural network is from the right result. The algorithm takes note and then updates the value of the weights thanks to the gradient descent. In order to update the weights the algorithm also needs a learning rate that can neither be too big nor too small. After a certain number of epochs and samples, the neural network is capable of reaching a good performance in classifying data.

CHAPTER 3: CONVOLUTIONAL NEURAL NETWORKS

Before seeing the application made for this thesis, it is necessary to introduce one last topic: Convolutional Neural Networks (CNN). This particular kind of neural network is particularly efficient in recognizing spatial patterns. Thanks to this ability they have been used mainly in image analysis, image recognition and so on. Basically they have a good performance whenever the data can be arranged in a visual disposition. For instance CNN can be used also for speech recognition, since the audio data can be represented in a graphical way. They have also been used for video games such as Space Invaders. The CNN recognizes the pattern of a certain disposition of the space ships and chooses the wisest move possible. If the data cannot be represented in a graphical way CNN's efficiency drops.

The ability of CNN to recognize spatial patterns exploits features detectors (which will be explained further along with the convolutional layers).

The main conceptual difference from a common neural network, such as the MLP, is the fact that CNN, instead of analyzing the whole image at once, analyses the patterns inside the image. The more complex is the CNN structure, the more complex patterns the CNN is able to recognize. In order to better understand how CNN works let's make an example. Suppose that there is a CNN used for facial recognition. The first convolutional layer is the one in charge of recognizing simple geometrical patterns, such as circles, squares, edges and angles. With these premises the second layer would be in charge of recognizing more complex objects, like noses and eyes for instance, while the third one could be in charge of recognizing faces. This example is actually an oversimplification on how a CNN works, but it helps to understand the main concept behind it.

From a structural point of view CNN are similar to MPL, the only difference is that the hidden layers are of a different kind. While the hidden layers of an MPL are all fully connected layers[14], the hidden layers of a CNN are also convolutional layers, relu layers, pooling layers and fully connected layers. Let's see what these new layers do starting from the most important one: the convolutional layers.

Convolutional Layers

The convolutional layers are the main core of a convolutional neural network. They are used to perform feature extraction. Like any other hidden layer the convolutional layers receive the input from the previous layer(that can be the input layer itself or some other hidden layer), "transform" the data received and then it gives the output to the following

[14] Which means that every node of one layer is connected to every node of the following layer

layer. The transformation applied by the convolutional layer is, like the name suggests, a convolution. Convolution is a linear operation where a filter is applied to the input matrix representing the image. After the convolution of a filter over an image a feature map is created. The feature map is the output of a convolution layer and usually serve as the input for the pooling layer.

In order to recognize pattern inside images the convolutional layer “applies” one or more filters, or feature detectors, to the image. These filters can be seen as little matrices nxn with, at first, random values. Each node of a single convolutional layer is a different filter applied to the image. While the values of the filters change over time in a similar way to the weights of a canonical neural network, so through backpropagation, the dimensions of the filters and its stride[15] are determined by the user. When the filters “slide” over the image they create a feature map. The creation of feature maps is a process called convolution. Let’s start with a simple example in order to better understand the main idea. Suppose that there is a CNN with only a convolutional layer with just a single node in it, and suppose that the matrix representing the filter is a 3x3 matrix. So the net receives an input and passes it to the convolution layer. The filter then convolves across each 3x3 block of pixels until it convolved the whole image. When the filter convolves a block of pixels what it actually does is the dot product between the values if the filter and the values of the pixel of the image. The result of the dot product gets stored and when the convolution of the filter over the whole image is done there is a representation of the initial input made of all of the stored results of the dot product. This new representation of the original input is passed to the next layer. Before passing on the other types of layers it is necessary to spend a few more words on the filters.

As previously stated the filters initially have random values that changes sample after sample, or batch after batch, through backpropagation. The filters can, obviously, be more than one. Every filter is in charge of recognizing a certain pattern. Let’s see another example, this time regarding the MNIST dataset. Suppose that the CNN has already completed the training phase, so its filters have non random values. The filters are shown in the next image. Suppose that where the values of the filters can be visually represented as follows: where there is a -1 the color associated is black, when the value is 0 the resulting color is gray and when the value is 1 the color is white.

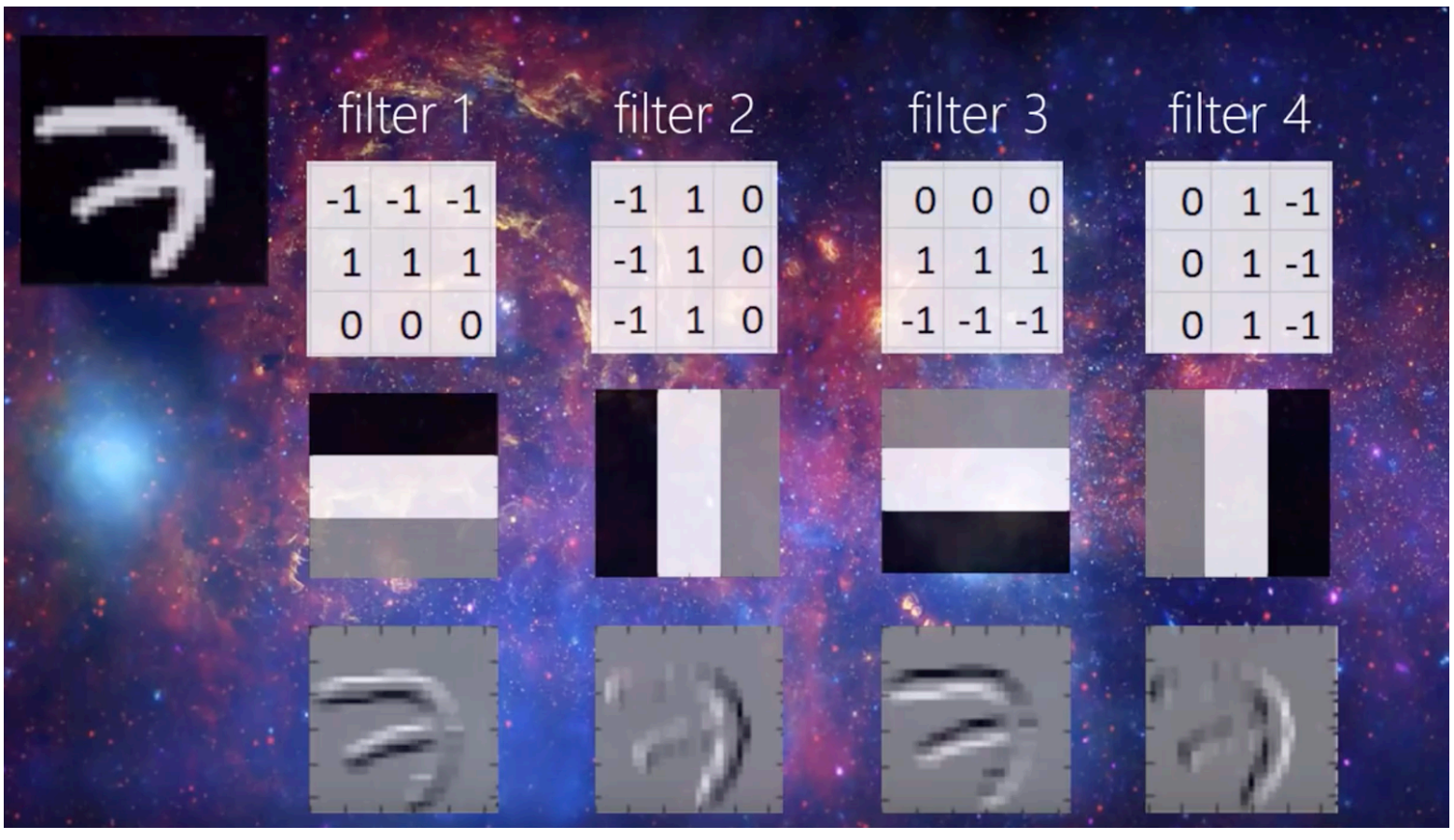
So if every filter convolves across the whole image it recognize a different piece of the image as shown in the image below.

The first row shows the filters, the second row shows the visual representation of the filters and the third and final rows shows the visual result of the convolution of one filter over the image. The white parts of the images represent the piece of pattern recognized by the filter. So in this scenario the first filter is able to recognize the horizontal superior

---

[15] The stride specifies the number of pixels that the filter moves over before performing another convolution.

edges, the second one is able to recognize the vertical left oriented edges, the third one is able to recognize the horizontal inferior edges and the fourth one recognizes the vertical right oriented edges.
These would be the filters at the beginning of the CNN, more complex filters are located in the deeper layers of the network. After the convolution layer sometimes there is a normalization layer.

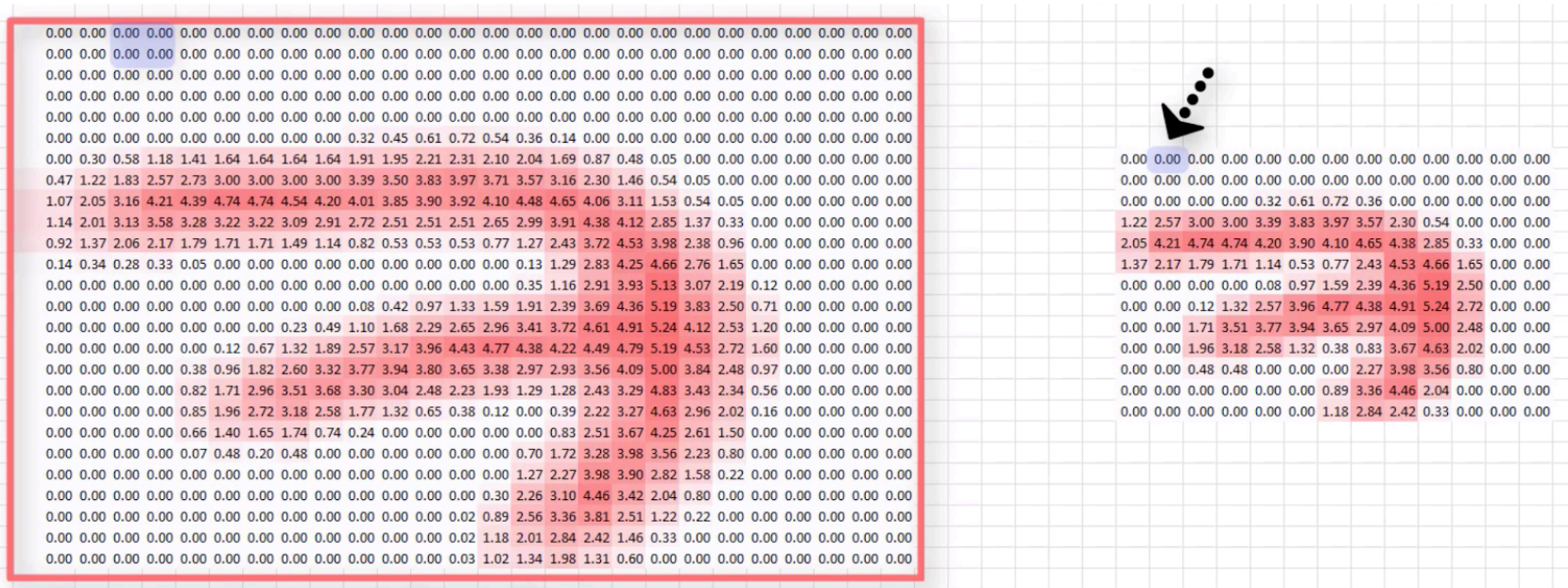


The normalization layer is used to cancel every negative value through the use of ReLU (Rectified Linear Unit), in order to avoid weird results. After this layer the data, goes into a pooling layer.

Pooling Layer

The pooling layer is another element that made CNN so popular. Usually there is a pooling layer after every convolutional layer. The pooling layer allows a dimensionality reduction on the number of features and pixels to consider. Like the convolutional layer the pooling layer applies a filter to the feature map that it receives from the convolutional layer. Also for the pooling filter is necessary to specify the dimension of the filter and the stride. There are many kinds of pooling that are used, but the most used are the Max Pooling and the Average Pooling. The max Pooling, which is by far the most popular,

takes the highest value of the pixels considered inside its window and stores it. The average pooling makes an average of all the values inside the window and then saves that value.



In the image above on the left there is a feature map, while on the right there is the new representation of the data after a Max Pooling operation. The dimension of the filter were 2x2 and the stride was 2. As a result of the max pooling operation the dimensions of the new data representation are 13x13 while before it was 26x26. So the new data representation is one fourth of the original one (26x26=676, 13x13=139, 139/676=1/4).
By applying a max pooling filter to the feature map, besides reducing the data dimension and the computational load, the most active pixels are selected.

CNN Summary

So how does the combination of convolutional layers and pooling layers allows to recognize patterns? When the CNN is trained the values of the convolutional filters are not random anymore, but actually represent patterns. After the convolution layer a feature map is passed to the pooling layer. The pooling layer takes into account only the highest values of the pixels, that are sufficient to determine the pattern. The other pixels are discarded. So on one hand there is a reduction of the data dimensions, and on the other hand the pixels that remain are the ones that are more active and using only them is still possible to recognize the pattern analysed.
As it was said at the beginning of this chapter with more convolutional layers (and pooling layers), the ability of the CNN to recognize patterns grows more and more.

CHAPTER 4: DATA TREATMENT AND CREATION OF A CNN

As previously stated, the purpose of this thesis is to create a machine learning algorithm capable of classifying images into film shot. First of all, it must be said that the whole set of film shot[16] was not taken into account, due to the fact that there is no database with images classified in such a way, so it was necessary to build the database from the start and in order to save time, the amount of film shot considered had to be reduced. The classes considered were *close up*, *half torso*, *half figure* and *full figure*. By doing so, the following types were excluded: *long shots*, *total interior*, *total exterior*, *extreme long shots*, and so on.

The neural network used is a Convolutional Neural Network (CNN). The structure of a CNN consists of a Multi Perceptron Layer, the neural network explained in the previous chapter, that has some additional layers such as the convolutional layers and the pooling layers.

The images used to train the algorithm are black and white and have a final resolution of 160x90 pixels. This resolution was chosen because it is a good compromise between the number of features that the neural network receives at its input layer. Since the dataset was built from the start every single image used had to be labeled manually. When the images in the dataset were labelled their resolution was 320x180. The resolution of 320x180 pixels was chosen because it is a resolution that still allows the human eye to recognize the type of image without too much effort. On the other hand such resolution was small enough for the compiler to show the images that were labelled. After labeling all the images, their resolution was lowered to 160x90. Such a choice was made in order to reduce the computational load that the computer had to process during the training phase.

Another reason for choosing the resolutions of 320x180 and later 160x90 concerns the aspect ratio of the images. Since 16:9 has been the main aspect ratio of most images[17] in the last decades, such resolution eased the creation of the dataset.

Last but not least, since the purpose of this thesis is to create a software that eases the editing of a video, and most videos have that aspect ratio, it was chosen also for this reason.

If a higher resolution was chosen that would have implied an even higher number of features. This leads to the reason why the images were completely desaturated.

---

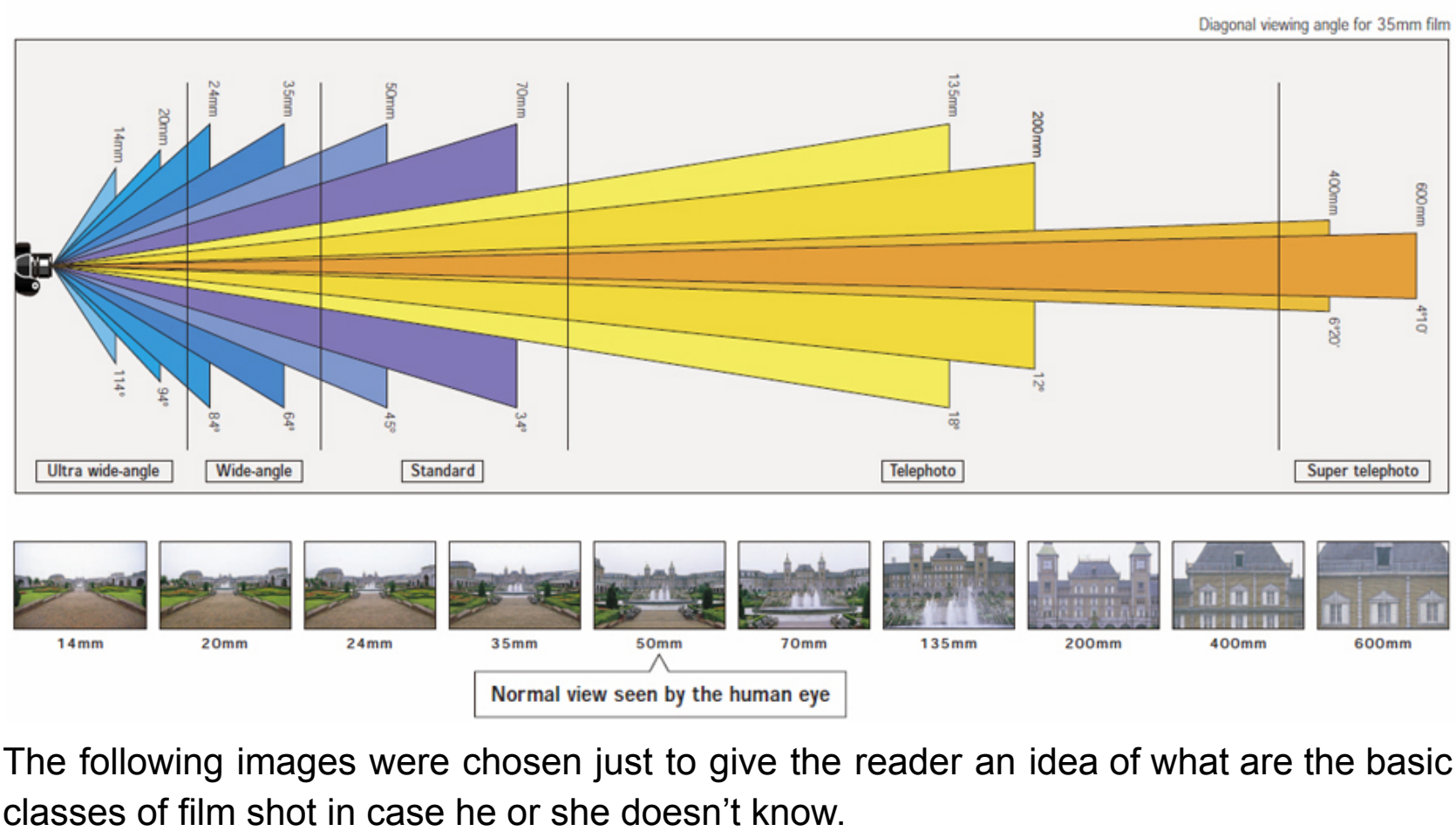
[16] The term "shot" is derived from the early days of film production when cameras were hand-cranked and operated similarly to the hand-cranked machine guns of the time. That is, a cameraman would "shoot" film the way someone would "shoot" bullets from a machine gun

[17] With images here are intended frames of videos and photographies. If the reader is interested in the story of such aspect ratio he will find a deepening in the appendix.

Before going on with the reason why the images were desaturated a little excursus, that regards how a machine reads images, is necessary. When a machine reads an image what it actually sees is a sequence of values between 0 and 255. With black and white images these values are on the gray scale, so every value corresponds to a different shade of gray, with the 0 that corresponds to completely black and the 255 that corresponds to the absolute white. Every value corresponds to a pixel to be lit in a certain way. For instance, if all the values are zeros the image corresponding would be a completely black image. The fact that every value corresponds to a pixel is true only for black and white images. With colored images things change a little bit. For instance, RGB images have three scales: the red scale, the green scale and the blue scale. So, when a computer reads an RGB image, for every pixel it receives three values, one for the red scale, one for the green scale and one for the blue scale. When the values of all the three scales are the same, the color resulting corresponds to a shade of gray, otherwise the color resulting depends on the magnitude of those values and the proportion between those values.
Since the colors inside an image are not relevant in order to determine if that image is a close up or something else, it was decided to desaturate them in order to avoid to increase even more the number of features. In fact, with black and white images there are “only” 14’400 features to consider, one for each pixel (160x90=14’400), while if RGB images were used the number of features would have been three times higher (160x90x3=43’200), because instead of using only the gray scale, it would have been necessary to use the red scale, the green scale and the green scale.

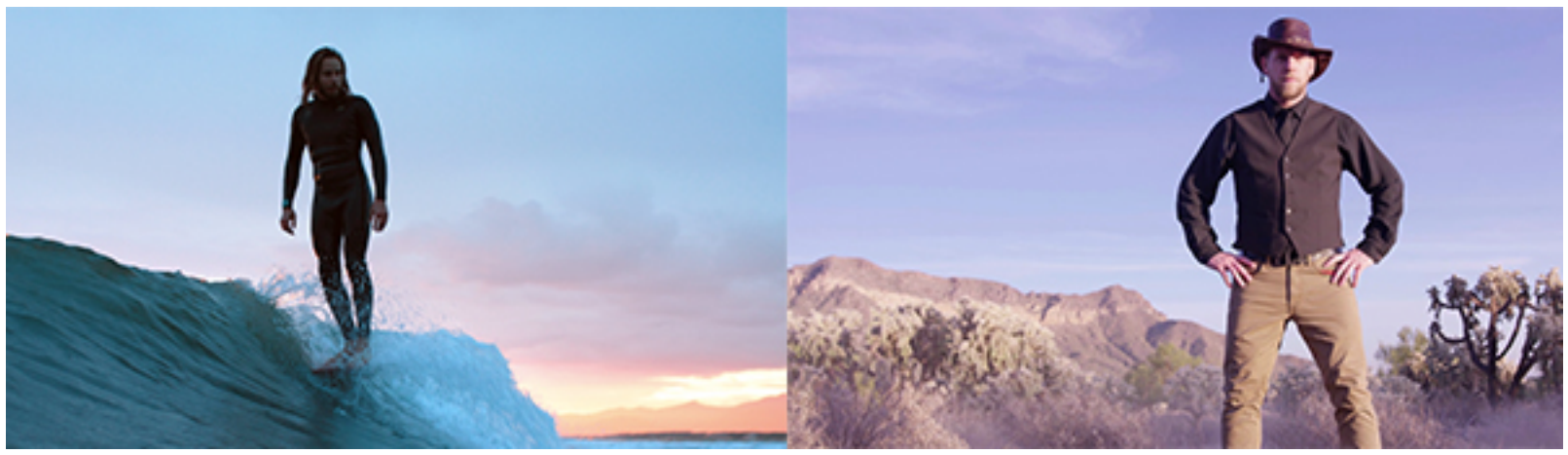
Cinematographic Shot Classes

As previously mentioned, the shot categories chosen for this work amount to four, but in the movie industry there are many more.
The most common way to classify the shots is on the basis of the field size, which is the method chosen for this study. Another way to classify the shots is based on the position of the camera.
The field size is determined by the portion of the subject and of the environment shown in the field of view of the camera. The field of view is determined by two factors. The first factor is the lens used. Every lens has a different angle of view: the shorter the lens, the wider the angle of view. The second factor is the distance of the subject from the camera.
There are many different types of shot but, in order to save time and space, below only the most basic types of film shots will be shown.



The following images were chosen just to give the reader an idea of what are the basic classes of film shot in case he or she doesn't know.

At the left there is an extreme long shot, while on the right there is a long shot. The extreme long shot is a shot with a wide open space, such as a mountain, a skyline and so on. The human figure is either absent or secondary compared to the environment. In the long shot the portion of space shown is smaller compared to the extreme long shot, however, the human figure is still secondary.



At the left there is a full shot, while on the right there is an american shot. The full figure shows the human form in its whole form, while the american shot shows three quarters of the human figure, from the knee to the head. It is said that the american shot was invented in western movies, in order to show also the gun hanging on the belt of the cowboys[18].

At the left there is a half figure, while on the right there is a half torso. These types of film shot have their explanation inside their own name.

[18] Actually the first precursor of the american shot, according to the italian wikipedia, was David Wark Griffith, the father of the classic cinematographic language.

At the left there is a close up, while on the right there is an extreme close up. The close up shows the subject from the shoulders up, while the extreme close up shows just a portion of the face and in some cases only the eyes of the subject.
As said in the introduction, the shot classification used in this study is a little bit different, in fact the types used to train the algorithm are just four. This choice was made in order to reduce the number of images that the algorithm had to process in order to be able to classify them. Having to deal with less classes implies having to find less images to label. The classes chosen for this thesis are the close up, the half torso, the half figure and the full figure. Some of the images fed to the algorithm were actually extreme close ups and american shots, but they were included in the close up and in the full figure classes respectively. This choice was made because when a neural network is trained it is necessary to train it with the same amount of samples from every class, otherwise there is the chance that the neural network cheats on the classification. In other words there is the risk that the neural network, after the training phase, misclassifies an image belonging to a class in which it knows that there are few samples. The neural network cheats because it is easier for it to misclassify those few examples instead of learning what they actually are.

Pipeline

The images in the database come from different sources. Some of them are from photographers, while others were downloaded from the internet. Before feeding them to the algorithm it was necessary to treat the images. The images needed to be treated in order to avoid errors and in order to convert them into a format that allows the neural network to work with them in a better and faster way. The first step consisted of resizing all the images with the same number of pixels. If the images have different sizes the neural network is not capable of processing them. The second step was about sorting

all images into the right class. The third and final step concerning the data treatment consisted of converting the images in the fastest and most readable format possible. After these steps the data were ready to be fed to the neural network. For every step an algorithm was created. Let's take a closer look to every step.

Resizing the images.

The first step was to convert all the images in jpeg with a pixel resolution of 320x180. In some cases some of them were already jpeg and it was necessary only a rescaling, while other images were png or ever raw files. Luckily there is a method in Photoshop that allows to apply a sequence of operations to multiple files at once using a batch operation. So a sequence of operations was created. The sequence of operations is pretty simple and it is divided in three steps. The first step consists in resizing the image dimensions, while the second one is a rescale operation on the canvas[19]. It is necessary to rescale both the image and the canvas in order to avoid useless images. If someone rescales only the image dimensions, the final file will have the right proportions, but if the image did not have the same proportion it will be deformed.
On the other hand, if someone rescale only the canvas size the final file will have the right proportions but it will show only a portion of the image, since it kept the original dimensions. Since not all of the images had the right proportion, by applying these first two steps in some cases the pixels on the image's edges were cropped. However the cropped pixels did not possess relevant information. The final step consists in saving all of the rescaled images as jpegs in a folder that will be used later on by a python code to label the images.

The first one, from left to right, is the original image, the second one is a file with only the canvas resized, while the third one is the file with only the resized image.

[19] The canvas is the background of the image.

The Image Sorting Algorithm

The second step was to catalog all the images with the right label, which was done by creating a simple code in python in order to speed up the process. This code reads the images one by one and let the user “label” them. Here label is between quotation marks because actually what this code really does is moving the images from a folder to other folders. The other algorithms will then label the images by using the source folder.

```
import os
import cv2
import matplotlib.pyplot as plt
from pathlib import Path
import shutil
DESTPP='/Users/bartolomeovacchetti/Desktop/ImgDataset/Primo piano'
DESTMB='/Users/bartolomeovacchetti/Desktop/ImgDataset/Mezzo busto'
DESTMF='/Users/bartolomeovacchetti/Desktop/ImgDataset/Mezza Figura'
DESTFI='/Users/bartolomeovacchetti/Desktop/ImgDataset/Figura intera'
```

This first part of the code import the libraries that will be used later and set the path for the destination folders. DESTPP and the others are the paths that leads to the different folders.

```
path=Path('/Users/bartolomeovacchetti/Desktop/ImgMix')
for img in os.listdir(path):
    try:
        img_path=os.path.join(path,img)
        img_array=cv2.imread(img_path, cv2.IMREAD_UNCHANGED)
        plt.imshow(img_array, cmap="gray")
        plt.show()
```

This part is actually the heart of the algorithm. The first line of code set the path for the source folder, then the code initializes a *for* cycle that has as many iterations as the files inside the source folder. The *try* command is used in order to prevent the algorithm to stop due to errors that sometimes are contained in the files. Then the algorithm creates a path for every single image that then is read and shown by the algorithm.

```
        v=int(input("inserire classe immagine 1 per pp, 2 per mb, 3 per mf, 4 per fi, 5 per
scartare: "))
```

```
if v==1:
    print("hai scelto pp")
    shutil.move(img_path, DESTPP)
elif v==2:
    print("hai scelto mb")
    shutil.move(img_path, DESTMB)

elif v==3:
    print("hai scelto mf")
    shutil.move(img_path, DESTMF)

elif v==4:
    print("hai scelto fi")
        shutil.move(img_path, DESTFI)

    else:
        print("scartato")
        pass

except Exception as e:
    print("errore")
    pass

except TypeError as te:
    print("errore type")
    pass
```

This last part is where the user “label” the images. The *except* commands are there to handle possible errors done by the algorithm while reading the images.

The Creation of the Dataset

The following step consists of creating the training dataset. The training dataset is split in two parts. The first part contains the images saved as CSV files, while the second part contains the labels corresponding to the images. CSV stands for Comma Separated Values and is a text file that uses a comma to separate its values. It is a good file format for programs that have to handle large quantities of data.

```
import numpy as np
import os
import cv2
import pickle
import random
from tensorflow.keras.utils import to_categorical


DATADIR='/Users/bartolomeovacchetti/Desktop/ImgDataset'
CATEGORIES=["figura intera", "mezza figura", "mezzo busto", "primo piano"]

training_data=[]
```

After importing the libraries needed the code assigns to the variable DATADIR the path to the image folder, while CATEGORIES is a list of the subfolders contained in the dataset. Training data is an empty list that will be filled later on.

```
for category in CATEGORIES:
    path=os.path.join(DATADIR, category)
    class_num=CATEGORIES.index(category)
    for img in os.listdir(path):
        try:
            img_array=cv2.imread(os.path.join(path,img), cv2.IMREAD_GRAYSCALE)
            img_array=img_array.ravel()
            training_data.append([img_array, class_num])

        except Exception as e:
            print("errore")
            pass
```

This piece of code creates a path to every image in every subfolder. It does that through a *for* cycle, so at first it will look at all the images contained in the first folder, then it will move to the second folder and so on. The label of the folder, and of every image contained inside of it, is the numerical value corresponding to the index of the folder. In this scenario the *figura intera* class will have the label 0, *mezza figura* the label 1 and so on. After that the code reads the image as a black and white image, converts the

corresponding matrix into a vector and then appends that vector together with the label of the image to the list *training_data*.

```
random.shuffle(training_data)

X=[]
y=[]


for features, labels in training_data:
    X.append(features)
    y.append(labels)

y=to_categorical(y)
```

These few lines of code are essential. First the elements of the training data are shuffled. This is done because when a neural network is trained it gets a better performance if the data are shuffled. Otherwise, if the neural network first sees all close up and then all half torso, there is a high chance that it will not learn anything. After being shuffled the couples data and its corresponding label are split into two lists using a for cycle with two indices.
The last line of this piece of code “one hot” encodes the labels. The one hot encoding is a process that converts categorical variables into a more ML friendly form. By doing so instead of having the classes represented as values their are presented in a different format. Let’s say that before the one hot encoding the classes are represented as follows: full figure=0, half figure=1, half torso=2, close up=3. After the one hot encoding the new representation of the classes is: full figure=(1,0,0,0), half figure=(0,1,0,0), half torso=(0,0,1,0) and close up=(0,0,0,1). This is very helpful in order to avoid that the classes with a higher value end up with a more relevance from a numerical point of view. Also, if someone is using the mean squared error function as the loss function, the one hot encoding prevents misclassification by the neural network.

```
X=np.array(X)
y=np.array(y)


pickle_out=open("Img.pickle", "wb")
pickle.dump(X, pickle_out)
```

```
pickle_out.close()

pickle_out=open("Lab.pickle", "wb")
pickle.dump(y, pickle_out)
pickle_out.close()
```

The two lists are then converted into numpy lists, then they are saved in pickles files. Those pickles files will then be opened by the algorithm used to create and train the CNN.
These two algorithms used to treat the data before feeding them to the neural network could have been merged together in a whole algorithm. However, in order to prevent that an error done during the execution of the image sorting algorithm could have repercussions on the creation of the dataset, it was decided to split the two processes. By doing so it is also easier to explain which algorithm does what.

The actual code

All of what has been seen so far are the steps required to prepare the data to be fed to the neural network.
Before looking at the code let's see the structure of the CNN. So there is the input level which consists of 14'400 nodes, one for each pixel of the image.
Then there is a convolutional layer followed by a pooling layer. Then there another convolution layer and another pooling layer. These two layers were added in order to increase the performance of the CNN in recognizing more sophisticated patterns. Since the convolutional layers are followed by the pooling layers the data dimensionality is reduced.
Then there is a *flatten* layer followed by three fully connected layers, the last of whom is the output layer. The *flatten* layer transforms the structure of the samples, so instead of matrices the neural network will now work with one dimensional arrays. This was done because fully connected layers works better with 1-d arrays. The output layer has for nodes, one for each class of film shot considered, which are: full figure, half figure, half torso and close up.
The code required to classifying images into film shot is pretty simple.The code is just 33 lines long, but what it does it is actually a little more complex than it looks. This code creates a neural network using the *keras*[20] and *tensorflow*[21] libraries.

---

[20] Keras is a library developed by the Tensorflow team. In order to Run it needs also the tensorflow library. It was developed with the purpose of speeding up the process of experimentation.

Let’s take a closer look to the code.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten, Conv2D, MaxPooling2D
import pickle
import numpy as np

X=pickle.load(open("ImgCNN.pickle", "rb"))
y=pickle.load(open("LabCNN.pickle", "rb"))
```

These first lines of code call the libraries needed for the creation of a neural network, together with the pickle library, which is necessary to use the dataset, since it was converted into a pickle file, and numpy, which is the good library in order handle matrices and vectors operations.
The last two lines of this first part of the code load the content of the two pickle files into two numpy arrays, X and y, one for the images and the other one for the labels.

```
X=X/255.0

model=Sequential()
model.add(Conv2D(128, (3,3), input_shape=X.shape[1:]))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2,2)))

model.add(Conv2D(64, (3,3)))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2,2)))
```

The first line of the code is there just to change the values in the vector X, which contains the values of every single pixel of the images, from the range 0-255 to the range 0-1, in order to ease the operations for the computer. Usually there are more

[21]Tensorflow is an open source software library developed for machine learning. It was created by the Google Brain team back in 2015 and it is one of the most used libraries for machine learning. It is used in a lot of different fields, such as voice recognition and image processing.

clean ways to normalize the data, but in this case it was sufficient to divide the values of the images by 255.
The *model=Sequential()* line of code specifies what kind of structure the neural network will have. By using the sequential structure a neural network created will be a feedforward type.
Then there are the convolutional layer followed by a normalization layer and a pooling layer. This line *model.add(Conv2D(128, (3,3), input_shape=X.shape[1:]))* is telling the compiler to add a convolutional layer with 128 nodes(=filters). It is also setting the dimensions (3x3) of the feature detectors, or filters. Finally *input_shape=X.shape[1:]* is just a way to tell the compiler to take in to account all the images in the dataset, no matter how many there are.
The line *model.add(Activation("relu"))* serve the purpose to eliminate eventual negative values. Then the *model.add(MaxPooling2D(pool_size=(2,2)))* just adds to the model a pooling layer that performs the max pooling operations with a filter of two by two pixels.
Then there are the fully connected layers.

```
model.add(Flatten())

model.add(Dense(64))
model.add(Activation('relu'))

model.add(Dense(32))
model.add(Activation('relu'))

model.add(Dense(4))
model.add(Activation("relu"))
```

The *model.add(Flatten())* is the previously mentioned layer that transform the matrices that represent the images in one dimensional vectors, in order to reduce the computational load. The following lines of code just add fully connected layers with their activation function. The last dense layer is the output layer and it has 4 nodes, one for each class.

```
model.compile(loss="mean_squared_error", optimizer="sgd", metrics=['accuracy'])
c=0
try:
    model.fit(X,y, batch_size=3, epochs=30, validation_split=0.1)
except ValueError as ve:
    c=c+1
```

```
    pass
print("numero errori: "+str(c))
model.summary()

model.save('CNN_Img_Clss_new.model')
```

These last lines of code are actually the most heavy for the computer from a computational point of view. The first one, *model.compile(etc..)* declare the loss function that will be used, so the *mean squared error,* which kind of gradient descent will be used as an optimizer, in this case *sgd* stand for *stochastic gradient descent.* Finally *metrics=['accuracy']* tells the compiler what parameters show after each epoch, in this case it showed the loss value and the accuracy value of both the training set and the validation set.
The *try* command is there just to prevent the arrest of the algorithm while training the network in case of errors. The *model.fit(etc...)* is the line of code that actually trains the CNN. X and y specifies which data and corresponding label to use in order to train the net, *batch_size* specifies how many images to pass together, epochs specifies how many times the net will have to train over the dataset and validation split tell to the algorithm which percentage of the dataset to use for the validation set, in this case 10%. The last line, *model.save('CNN_Img_Clss_new.model'),* saves the model created. In this way it can be used later to test if it actually works or not, without having to train the net from the beginning.
If the reader is interested in understanding better how this neural network actually works another code was written for this thesis. This code it is not as efficient as the code above, but it allows to understand better how a neural network works. In other words this other code is rough approximation of what is happening. It is a basic type of multilayer perceptron neural network, so there won't be the convolutional layers and the pooling layers. This code can be found in the appendix.

Results

The accuracy of the CNN developed for this thesis during the training phase is of 99%, while the accuracy reached with the validation set is around 60%.These results were obtained after 30 epochs.

```
Epoch 27/30
921/921 [==============================] - 134s 145ms/sample - loss: 0.0196 - acc: 0.9794 -
val_loss: 0.1490 - val_acc: 0.6505
Epoch 28/30
921/921 [==============================] - 135s 146ms/sample - loss: 0.0161 - acc: 0.9848 -
val_loss: 0.1541 - val_acc: 0.5728
Epoch 29/30
921/921 [==============================] - 135s 147ms/sample - loss: 0.0129 - acc: 0.9902 -
val_loss: 0.1518 - val_acc: 0.5922
Epoch 30/30
921/921 [==============================] - 136s 148ms/sample - loss: 0.0103 - acc: 0.9891 -
val_loss: 0.1545 - val_acc: 0.6019
```

If the epochs are increased to 40 the results slightly changes: while the accuracy on the training set still gravitates around 99%, the accuracy on the validation set rises to 64%.

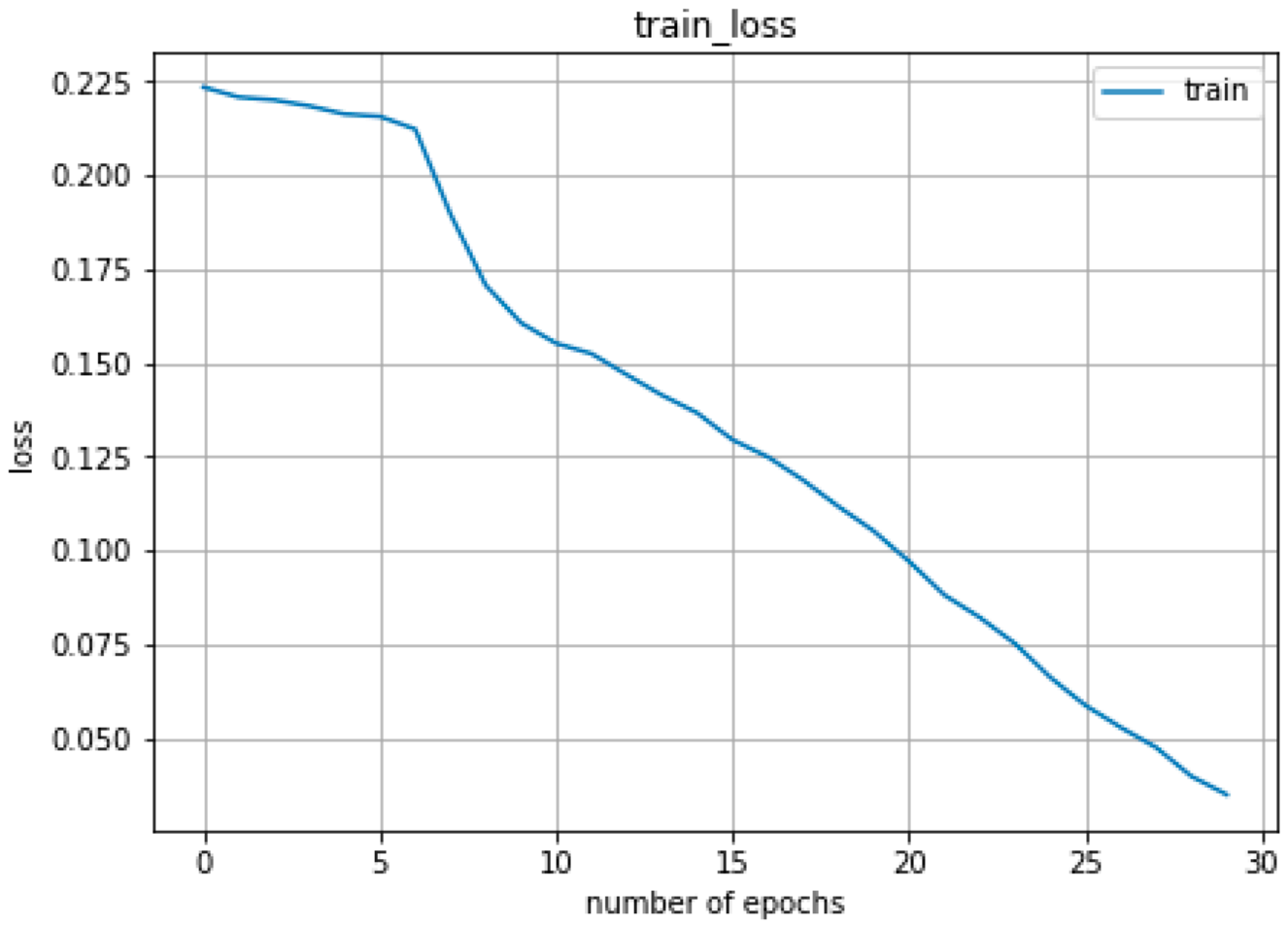
```
Epoch 37/40
921/921 [==============================] - 68s 74ms/sample - loss: 0.0072 - acc: 0.9891 -
val_loss: 0.1878 - val_acc: 0.5631
Epoch 38/40
921/921 [==============================] - 68s 74ms/sample - loss: 0.0080 - acc: 0.9891 -
val_loss: 0.1455 - val_acc: 0.6214
Epoch 39/40
921/921 [==============================] - 68s 74ms/sample - loss: 0.0064 - acc: 0.9891 -
val_loss: 0.1435 - val_acc: 0.6214
Epoch 40/40
921/921 [==============================] - 68s 74ms/sample - loss: 0.0062 - acc: 0.9924 -
val_loss: 0.1414 - val_acc: 0.6408
```

Increasing the number of epochs could be a strategy, however there are less brutal ways to reduce such difference between the training accuracy and the validation accuracy.

Such a difference between the training set accuracy and the validation set accuracy implies that the neural network overfits or underfits. In both scenarios the model has a good accuracy on the training set, but when it receives new images its accuracy drops. The first thing to do in order to understand if it is a problem of overfitting or underfitting is to plot the values of the loss function and see the slope.

As the reader can see the slope is always going down. This means that there is a problem of underfitting. The easiest way to solve an underfitting issue is to expand the dataset used for the training phase.

In case of an overfitting issue the slope would have increased after a certain epoch, so the values of the loss function would have increased instead of decreasing. In that scenario the minimum would have been exceeded. There are different ways to overcome an overfitting issue, such as data augmentation for instance, or , like in the case of the underfitting issue, expanding the dataset, possibly with variegate samples.



Instead of increasing the dataset other approaches can be tried. One of those approaches that can be tried in order to increase the accuracy of the CNN is to use the *cross-validation* technique.

The main idea behind cross-validation is to divide the dataset into k partitions. One of those partitions will be used for the test set, while the others will be used for the training set. So far is not so different from what was done by the code earlier. The difference lies in the fact that also the other partitions will be used for the validation set with different models. In order to be clearer let's see an example. Suppose that the k parameter mentioned earlier is set equal to three, so the dataset will be divided into three equal partitions. As well as the dataset also the model will be divided into three models. The first model will be trained with the first two partitions and will use the third partition as test set. The second model will be trained with the first and third partitions and will use the second partition as validation set. The third model will be trained with the second and third partitions and will use the first partition as validation set. Each of these models will reach a certain accuracy. Here the cross-validation technique is used in order to find out which data distribution trains better the CNN.

```
Epoch 29/30
682/682 [==============================] - 21s 31ms/sample - loss: 0.0497 - acc: 0.9120
Epoch 30/30
682/682 [==============================] - 21s 31ms/sample - loss: 0.0439 - acc: 0.9370
342/342 [==============================] - 3s 8ms/sample - loss: 0.1449 - acc: 0.5526
Model Evaluation [0.1448788220248027, 0.55263156]
```

```
Epoch 29/30
683/683 [==============================] - 20s 30ms/sample - loss: 0.0455 - acc: 0.9312
Epoch 30/30
683/683 [==============================] - 21s 30ms/sample - loss: 0.0404 - acc: 0.9370
341/341 [==============================] - 2s 7ms/sample - loss: 0.1841 - acc: 0.4868
Model Evaluation [0.18410846250148113, 0.48680353]
```

```
Epoch 29/30
683/683 [==============================] - 21s 30ms/sample - loss: 0.0357 - acc: 0.9649
Epoch 30/30
683/683 [==============================] - 21s 30ms/sample - loss: 0.0322 - acc: 0.9663
341/341 [==============================] - 2s 7ms/sample - loss: 0.1483 - acc: 0.5806
Model Evaluation [0.1483085656707937, 0.58064514]
```

The images above are screenshots that show the final accuracy on the training set and the test set. In this case the second model is the one that has reached the highest accuracy. However the performance of these models is lower compared to the model trained without the cross-validation. This is because by splitting the dataset in three partitions the training set is a lot smaller, infact now it has less than 700 samples, while before it had more than 900 samples. However if k, the parameter used to split the dataset, has a higher value things change. Let's set k equal to 10. In this way the validation set and the training set will have the same dimensions of the original training set, but every model will have a different training and validation sets. In this scenario the models trained will be 10. The downside of the cross-validation technique is that it takes a lot of time in order to be computed, since the number of models to be trained is higher.

```
Epoch 29/30
921/921 [==============================] - 27s 30ms/sample - loss: 0.0888 - acc: 0.7644
Epoch 30/30
921/921 [==============================] - 27s 30ms/sample - loss: 0.0828 - acc: 0.7818
103/103 [==============================] - 1s 9ms/sample - loss: 0.2692 - acc: 0.3301
Model Evaluation [0.26920055431648365, 0.33009708]
```

```
Epoch 29/30
921/921 [==============================] - 27s 30ms/sample - loss: 0.0992 - acc: 0.7220
Epoch 30/30
921/921 [==============================] - 27s 30ms/sample - loss: 0.0947 - acc: 0.7351
103/103 [==============================] - 1s 9ms/sample - loss: 0.1985 - acc: 0.4466
Model Evaluation [0.19854964084416918, 0.44660193]
```

```
Epoch 29/30
921/921 [==============================] - 27s 29ms/sample - loss: 0.0368 - acc: 0.9544
Epoch 30/30
921/921 [==============================] - 27s 29ms/sample - loss: 0.0318 - acc: 0.9631
103/103 [==============================] - 1s 8ms/sample - loss: 0.1439 - acc: 0.5922
Model Evaluation [0.14392266762488096, 0.592233]
```

```
Epoch 29/30
921/921 [==============================] - 27s 29ms/sample - loss: 0.0316 - acc: 0.9642
Epoch 30/30
921/921 [==============================] - 27s 29ms/sample - loss: 0.0267 - acc: 0.9729
103/103 [==============================] - 1s 9ms/sample - loss: 0.1854 - acc: 0.4660
Model Evaluation [0.18535456799187708, 0.46601942]
```

```
Epoch 29/30
922/922 [==============================] - 27s 29ms/sample - loss: 0.0382 - acc: 0.9501
Epoch 30/30
922/922 [==============================] - 27s 29ms/sample - loss: 0.0325 - acc: 0.9588
102/102 [==============================] - 1s 9ms/sample - loss: 0.2475 - acc: 0.3529
Model Evaluation [0.24753523340412215, 0.3529412]
```

```
Epoch 29/30
922/922 [==============================] - 27s 29ms/sample - loss: 0.0285 - acc: 0.9620
Epoch 30/30
922/922 [==============================] - 27s 29ms/sample - loss: 0.0248 - acc: 0.9729
102/102 [==============================] - 1s 9ms/sample - loss: 0.1800 - acc: 0.5000
Model Evaluation [0.17999009305939956, 0.5]
```

```
Epoch 29/30
922/922 [==============================] - 27s 29ms/sample - loss: 0.0406 - acc: 0.9469
Epoch 30/30
922/922 [==============================] - 27s 29ms/sample - loss: 0.0356 - acc: 0.9566
102/102 [==============================] - 1s 9ms/sample - loss: 0.1508 - acc: 0.5784
Model Evaluation [0.150806345483836, 0.57843137]
```

```
Epoch 29/30
922/922 [==============================] - 27s 29ms/sample - loss: 0.0232 - acc: 0.9772
Epoch 30/30
922/922 [==============================] - 27s 29ms/sample - loss: 0.0177 - acc: 0.9859
102/102 [==============================] - 1s 8ms/sample - loss: 0.1447 - acc: 0.5588
Model Evaluation [0.14474820593992868, 0.5588235]
```

```
Epoch 29/30
922/922 [==============================] - 27s 29ms/sample - loss: 0.0809 - acc: 0.7354
Epoch 30/30
922/922 [==============================] - 27s 29ms/sample - loss: 0.0788 - acc: 0.7364
102/102 [==============================] - 1s 8ms/sample - loss: 0.1603 - acc: 0.5000
Model Evaluation [0.16030222761864757, 0.5]
```

```
Epoch 29/30
922/922 [==============================] - 27s 29ms/sample - loss: 0.0303 - acc: 0.9577
Epoch 30/30
922/922 [==============================] - 27s 29ms/sample - loss: 0.0261 - acc: 0.9675
102/102 [==============================] - 1s 9ms/sample - loss: 0.1304 - acc: 0.6373
Model Evaluation [0.13044956852408016, 0.6372549]
```

As shown by the images above the accuracy on the test set changes in a significant way depending on the data used to train the different models. For instance the accuracy of the last model, which is around 63%, is almost twice the accuracy of the first model, which is 33%.

The cross-validation technique has proven useful since it was possible to find a data distribution that allows an accuracy of 63%. The algorithm that didn't used such technique allowed to create a CNN with an accuracy of 60%.

However the accuracy reached is the overall accuracy, which means that the accuracy on every single class can be different.

	precision	recall	f1-score	support
0	0.68	0.68	0.68	19
1	0.65	0.50	0.57	30
2	0.65	0.52	0.58	25
3	0.60	0.86	0.71	28
micro avg	0.64	0.64	0.64	102
macro avg	0.65	0.64	0.63	102
weighted avg	0.64	0.64	0.63	102

The numbers on the right indicate the classes of cinematographic shots, where 0 is the *full figure*, 1 is the *half figure*, 2 is the *half torso* and 3 is the *close up*.
So when it comes to classify full figure shots the model has a precision of 68% while when it classifies close up its precision drops to 60%. A precision of 68% in classifying full figure shots means that the model is able 68% of the time to not label a sample as a full figure shot when the sample is something else. The precision is given by the following ratio

$$precision = \frac{tff}{tff+fff}$$

where *tff* is the number of true full figure samples and *fff* is the number of false full figure samples.
Recall is the ability of the model to recognize all of the full figure samples inside the test set. It is defined as the following ratio

$$recall = \frac{tff}{tff+fse}$$

Where *fse* (false something else) is the number of samples that belong to the *full figure* class but are classified as something else.
The *f1-score* is the average between the *precision* and the *recall* parameters, while the support parameter indicates the number of samples inside that test set that belong to a certain class.

The *micro avg* is an average is calculated globally counting the total of true positives, false negatives and false positives, while the *macro avg* calculates metrics for each label finding the unweighted mean. On the other hand *weighted avg* takes into account labels imbalance[22] and compute a weighted mean. Since these two values are not so different it means that the dataset is quite balanced.
Another thing that could be done in order to see how CNNs can classify cinematographic shots is to reduce the number of classes considered. So instead of training the CNN on four classes it will be trained on two classes, the close up and the full figure. The reason to do such a thing is because some of the class considered for this study are very similar to each other. For instance the half torso is halfway between a close up and a half figure. This means that some samples belonging to the half figure class can be considered half torso and the other way around. Especially with a small dataset is normal for CNNs to not be able to classify in a proper way images that belongs to visually similar classes.
So another way to analyze the ability of a CNN to classify images into cinematographic shots is to consider just two classes very different one from another, so in this case the *full figure* and the *close up*.

```
Epoch 29/30
437/437 [==============================] - 13s 30ms/sample - loss: 0.0105 - acc: 0.9977
Epoch 30/30
437/437 [==============================] - 13s 30ms/sample - loss: 0.0087 - acc: 0.9977
49/49 [==============================] - 1s 10ms/sample - loss: 0.0747 - acc: 0.8776
Model Evaluation [0.0747026127516007, 0.877551]
```

```
Epoch 29/30
437/437 [==============================] - 13s 30ms/sample - loss: 0.0039 - acc: 1.0000
Epoch 30/30
437/437 [==============================] - 13s 30ms/sample - loss: 0.0029 - acc: 1.0000
49/49 [==============================] - 1s 11ms/sample - loss: 0.1675 - acc: 0.7755
Model Evaluation [0.16752366235061567, 0.7755102]
```

```
Epoch 29/30
437/437 [==============================] - 13s 29ms/sample - loss: 0.0103 - acc: 0.9977
Epoch 30/30
437/437 [==============================] - 13s 30ms/sample - loss: 0.0085 - acc: 0.9954
49/49 [==============================] - 1s 11ms/sample - loss: 0.1662 - acc: 0.7959
Model Evaluation [0.16622797293322428, 0.79591835]
```

```
Epoch 29/30
437/437 [==============================] - 13s 30ms/sample - loss: 0.0140 - acc: 0.9931
Epoch 30/30
437/437 [==============================] - 13s 30ms/sample - loss: 0.0111 - acc: 0.9954
49/49 [==============================] - 1s 11ms/sample - loss: 0.1842 - acc: 0.7347
Model Evaluation [0.18424536774353106, 0.7346939]
```

[22] The term label imbalance refers to a situation in which not every class has the same number of samples.

```
Epoch 29/30
438/438 [==============================] - 13s 30ms/sample - loss: 0.0204 - acc: 0.9886
Epoch 30/30
438/438 [==============================] - 13s 30ms/sample - loss: 0.0157 - acc: 0.9932
48/48 [==============================] - 1s 11ms/sample - loss: 0.1500 - acc: 0.8125
Model Evaluation [0.15000320474306741, 0.8125]
```

```
Epoch 29/30
438/438 [==============================] - 13s 30ms/sample - loss: 0.0079 - acc: 0.9977
Epoch 30/30
438/438 [==============================] - 13s 29ms/sample - loss: 0.0061 - acc: 0.9932
48/48 [==============================] - 1s 11ms/sample - loss: 0.1602 - acc: 0.7292
Model Evaluation [0.16019895672798157, 0.7291667]
```

```
Epoch 29/30
438/438 [==============================] - 13s 29ms/sample - loss: 0.0047 - acc: 1.0000
Epoch 30/30
438/438 [==============================] - 13s 29ms/sample - loss: 0.0038 - acc: 1.0000
48/48 [==============================] - 1s 12ms/sample - loss: 0.2412 - acc: 0.6875
Model Evaluation [0.24121553202470145, 0.6875]
```

```
Epoch 29/30
437/437 [==============================] - 13s 30ms/sample - loss: 0.0134 - acc: 0.9931
Epoch 30/30
437/437 [==============================] - 13s 30ms/sample - loss: 0.0101 - acc: 0.9977
49/49 [==============================] - 1s 11ms/sample - loss: 0.1511 - acc: 0.8367
Model Evaluation [0.15113750799578063, 0.8367347]
```

```
Epoch 29/30
438/438 [==============================] - 13s 29ms/sample - loss: 0.0117 - acc: 0.9954
Epoch 30/30
438/438 [==============================] - 13s 29ms/sample - loss: 0.0091 - acc: 0.9977
48/48 [==============================] - 1s 11ms/sample - loss: 0.1446 - acc: 0.8125
Model Evaluation [0.14459116260210672, 0.8125]
```

```
Epoch 29/30
437/437 [==============================] - 13s 29ms/sample - loss: 0.0082 - acc: 0.9977
Epoch 30/30
437/437 [==============================] - 13s 29ms/sample - loss: 0.0089 - acc: 0.9977
49/49 [==============================] - 1s 11ms/sample - loss: 0.0780 - acc: 0.9388
Model Evaluation [0.07804196426758961, 0.93877554]
```

With just two classes, even though the dataset is half of the original, the accuracy on the test set of the different models increases dramatically, floating from almost 69% to almost 94%.
Let’s take a closer look to the model with the almost 94% accuracy on the test set.

```
              precision    recall  f1-score   support

           0       0.91      0.95      0.93        22
           1       0.96      0.93      0.94        27

   micro avg       0.94      0.94      0.94        49
   macro avg       0.94      0.94      0.94        49
weighted avg       0.94      0.94      0.94        49
```

All of this was shown in order to prove to the reader that if a more complete dataset was built it is actually possible to have an algorithm that uses a CNN in order to classify cinematographic shots.

CHAPTER 5: CONCLUSIONS

As it was stated in the introduction of this document, the purpose of this thesis is to use a Convolutional Neural Network, also known as CNN, in order to classify not objects inside images but images themselves, using a partition similar to the frame partition used in the movie industry (close up, extreme close up, long shot, medium shot etc...).
The idea behind this project was to see if it was possible to create a software that exploits machine learning techniques in order to classify the video files that come out of a camera into film shots. Such a software would allow to reduce the video editing time. The time saved would be the time that the editor has to spend looking for the next shot to put in the video sequence. When someone is editing a video all the video files don't have any indication regarding the type of film shot. If the video files are already divided in close up, half figure and so on the editor would not have to look at all the video files but only at a smaller set.
The CNN developed for this thesis is able to process images, but it cannot handle videos. However since videos are nothing more than a sequence of images this problem can easily be resolved. The easiest way to establish if a video sequence is a close up, a full figure or something else is to give to the CNN some frames of that video sequence and let the algorithm classify them. When the algorithm has finished to classify the frames that it has received the class with the most frames belonging to it is the right class. In this way even if the neural network misclassifies some frames if the majority of the frames is classified in the right way it is safe to assume that the whole sequence will be classified in the right way, provided that that algorithm has a high enough accuracy.
During the simulations the accuracy on the validation set reached floats around 60%[23], which is not ideal. However, even with such low accuracy, if the neural network receives more than one frame for video it should be able to classify the video sequence in the right way.
While the accuracy on the validation set is around 60%, the accuracy reached on the training set by the neural network was around 99%. As it was shown in the previous chapter such difference is due to the small dimension of the dataset used. Since there wasn't the chance to use a database with images already classified into cinematographic shots a database with such classification was created manually.
Since the purpose of this thesis is to see if it is possible to classify images into cinematographic shots, the number of classes considered is reduced to four instead of the whole set. Such a choice was made because if all the classes were considered the number of samples required would have grown exponentially. Like it was shown in the

[23] The result section is better described at page 47

results section, with only two classes and half of the dataset the accuracy, with some data distribution, exceeds 90%.
The dataset created for this thesis has 1024 samples. This means that every class has a little bit more than 250 samples, which is far from the ideal amount of samples for class. The minimum amount of samples for class would be around 1000 samples for class. Since the samples for class in this thesis are one fourth of the minimum is natural for the network to underfit.
With a proper dataset, consisting of also the other film shot classes, such as the *long shot* for instance, it would be possible to develop a plugin capable of reducing the time required to edit a video. The CNN used to develop such plugin would require at least 8000 samples.
Another use for such a software could be an implementation with a generative adversarial network for the film restoration.
In order to reduce even more the editing time of a video there are other ways to exploit machine learning techniques. One thing that could be done is to teach a neural network to recognize the actors inside each shot. This would cut the editing time even more. If someone is editing a video and has all the video files divided into classes with also the information regarding who is acting in that shot the editor should then know which scene the shot should belong to.
This of course is just a small portion of what neural networks could do. There are actually much more powerful tools that exploits deep learning, such as the video style transfer, an algorithm that exploits the *vgg*[24] in order to transfer the style of drawings to a whole video sequence without discrepancies.
The problem with these more powerful tools is that they require computers with a huge computational power. On the other hand multiclass classification mentioned earlier would not require too much computational power. It would just need proper datasets and time.
There other types of classification that could be implemented by CNN. The first one that comes to mind is the classification of the video based on the position of the camera with respect to the actors instead of the field of view of the camera. There are also the camera movements to take into account. If all of these different classifications were taken into account it could lead to a software that implements multiclass classification[25].
The main idea is to add as many metadata as possible in order to ease the editing process. After all, AI nowadays is intended to solve tasks of practical nature.

---

[24] It is a deep convolutional neural network developed by Oxford's Visual Geometry Group.
[25] It is a classification in which samples can have more than one label.

APPENDIX

In this section there are a few insights regarding different subjects that were only briefly mentioned in the main core of the thesis. Some of these insights concern some mathematical properties used, while others are about more notional concepts.

Bayes

In order to understand in a better way how bayes theorem works let's look at an example.
Suppose that there is a rare disease that hits one person every hundred thousand people. Now suppose that there is a test that can verify if someone is sick, that is correct 95% of the time.
So P(sick) represents the probability of a single person to have such disease and is equal to 0,00001, P(sick|test+) is the probability of actually being sick if the test results positive, which is unknown, P(test+|sick) is 95%, which is the probability of the test to result positive if a person is sick.
So P(sick|test +) is given by

$$P(sick|test+) = \frac{P(test+|sick)*P(sick)}{P(test+)}$$

With P(test+) equal to:

$$P(test+) = P(test+|sick) * P(sick) + P(test+|healthy) * P(healthy)$$

$$P(test+) = (0,95 * 0,00001) + (0,05 * 0,99999)$$

$$P(test+) \simeq 0,05$$

So P(sick| test positive) is equal to

$$P(sick|test+) \simeq \frac{0{,}95*0{,}00001}{0{,}05}$$

$$P(tsick|rest+) \simeq 0,00019$$

So even if the test results positive, the probability of a person of being actually sick is pretty low.

A Little Insight on Ensemble Learning

Suppose that there is an algorithm that performs ensemble learning made of three other learning algorithms. If the three learning algorithms have an accuracy of 60% and have the same decision power, the ensemble learning algorithm will have an accuracy higher than 60%. This statement will be proved with probability calculus.
Let's denote the event in which the ensemble learning algorithm gives the right answer with *P(right).* P(right) is equal to the sum of *P(3R)*, that stands for the probability that all of the three learning algorithms inside the ensemble learning algorithm give the right answer, and *P(2R)*, that stands for the probability that at least 2 out of 3 algorithms give the right answer. This is because the decision made by the ensemble learning algorithm will be the one that has received the majority of the votes from the learning algorithms.
Since the decision of a learning algorithm is not influenced by the decisions of the other learning algorithms, the event that one of them is right or wrong is independent from the results of the other. Let's also call the events in which the learning algorithms are right *A, B* and *C*, while the events in which they are not will be called *notA*, *notB* and *notC*.
So

$$P(right) = P(3R) + P(2R)$$

$$P(3R) = P(A) * P(B) * P(C)$$

$$P(3R) = 0,6 * 0,6 * 0,6$$

$$P(3R) = 0,216$$

$$P(2R) = P(A)P(B)P(notC) + P(A)P(notB)P(C) + P(notA)P(B)P(C)$$

$$P(2R) = 3(0,4 * 0,6 * 0,6)$$

$$P(2R) = 0,432$$

$$P(right) = 0,648$$

So the final accuracy of the ensemble learning algorithm is 64,8%. In order to verify that this is correct let's compute the probability that the ensemble learning algorithm gives the wrong answer, keeping in mind that the sum of the two probabilities must give 1 as result.

$$P(wrong) = P(3W) + P(2W)$$

$$P(3W) = P(notA) * P(notB) * P(notC)$$

$$P(3R) = 0,4 * 0,4 * 0,4$$

$$P(3W) = 0,064$$

$$P(2R) = P(A)P(notB)P(notC)+P(notA)P(notB)P(C)+P(notA)P(B)P(notC)$$

$$P(2W) = 3(0,4 * 0,4 * 0,6)$$

$$P(2W) = 0,288$$

$$P(wrong) = 0,352$$

So if P(tot) is the complete probability space it results that

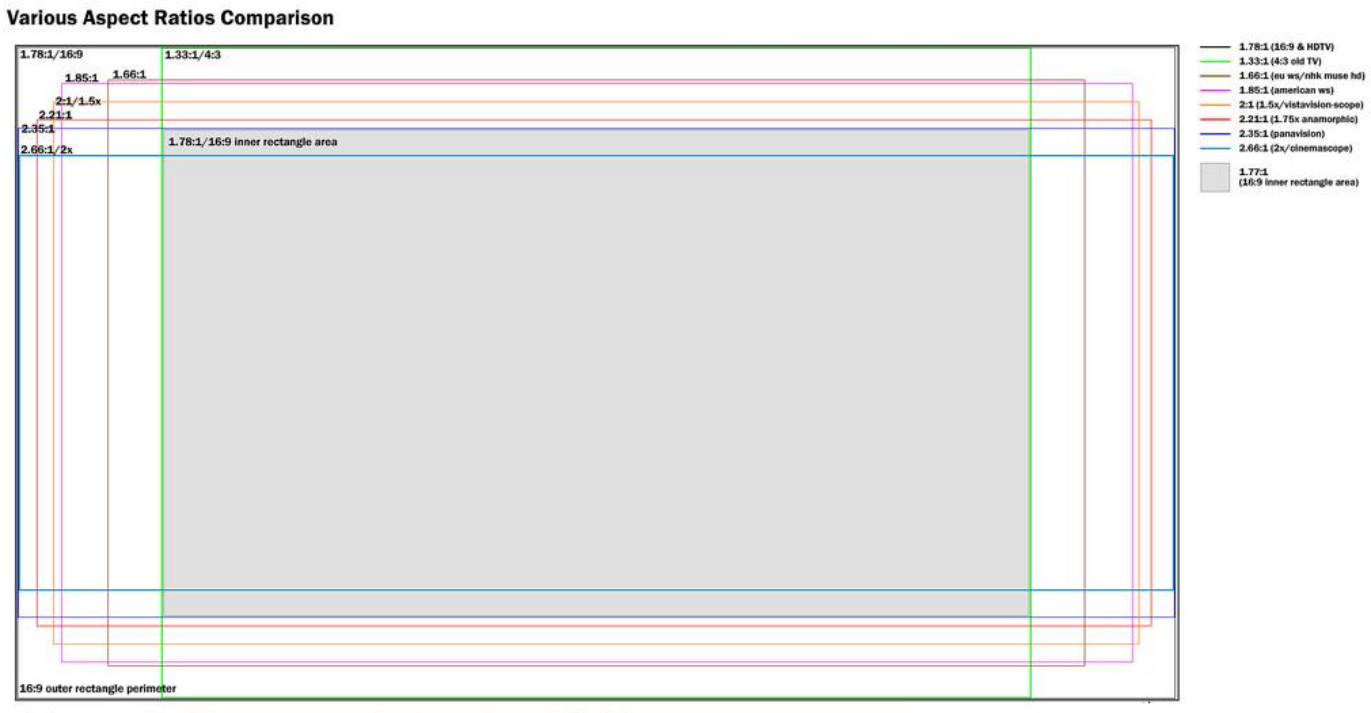
$$P(tot) = P(right) + P(wrong)$$

$$P(tot) = 0,648 + 0,352$$

$$P(tot) = 1$$

The 16:9 Aspect Ratio

This aspect ratio was developed by Dr. Kerns H. Powers, during the ‘80. It was at first a compromise between all the other aspect ratios, but then it became the most used one. Powers found this aspect ratio while comparing the different aspect ratios. He cut out rectangles shaped like the different aspect ratios, but all the rectangles had the same area.

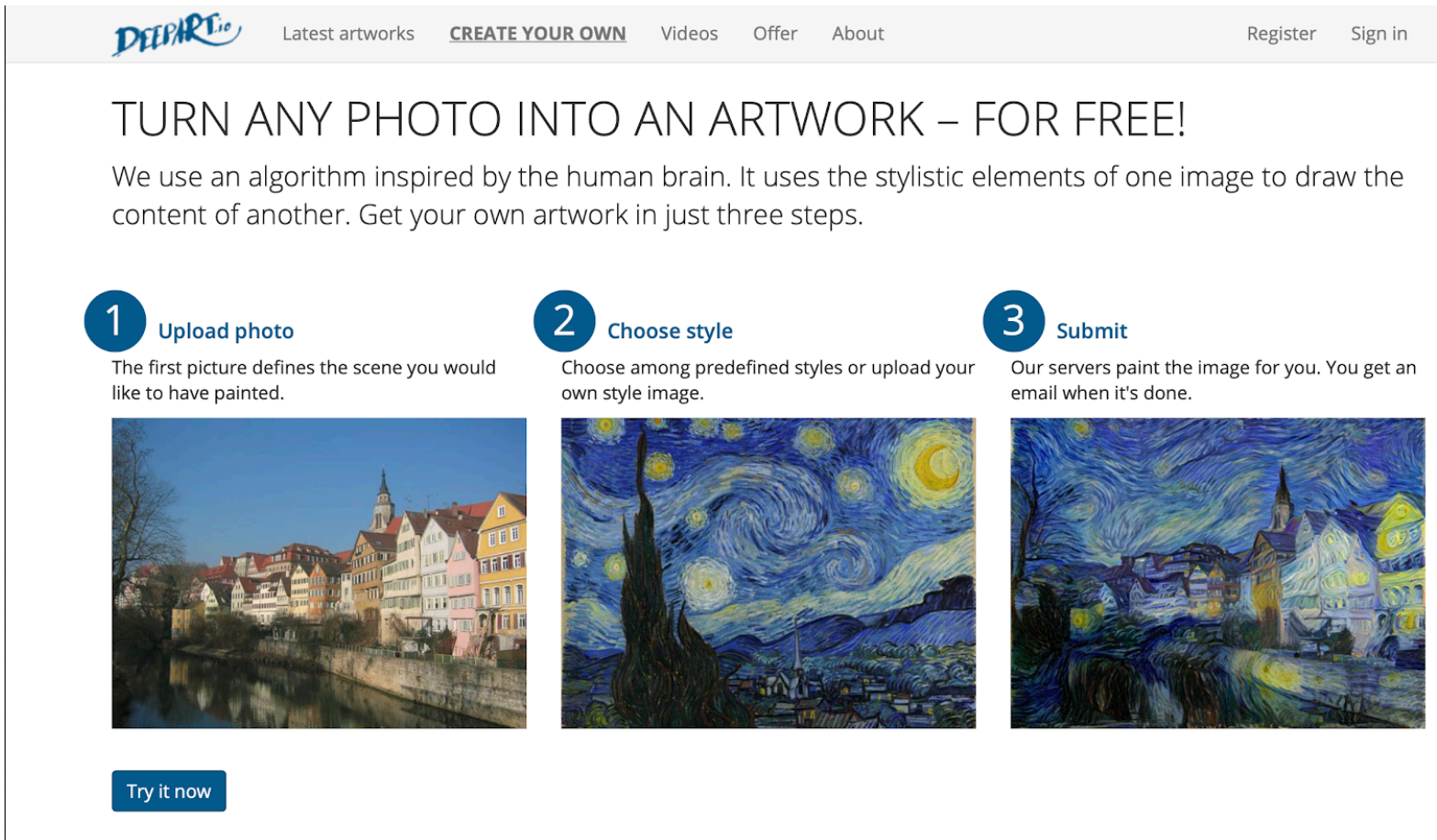


Kerns H. Powers' solution for 16:9

What Powers found out is that if the different aspect ratios are overlapped with their central points aligned two rectangles emerge, the outer rectangle, which contains all the other rectangles, and the inner rectangle, which is contained by all of the other rectangles. These two rectangles share the same proportion, that is the 16:9.
According to the italian wikipedia another aspect that benefitted the 16:9, especially when compared to the 4:3, is that the human psychovision is closer to the 16:9. The actual human field of view is around 4:3, but, due to evolutionary factors, such as the absence of flying predators,the human brain focuses its attention on the horizontal axis, rather than the vertical axis.

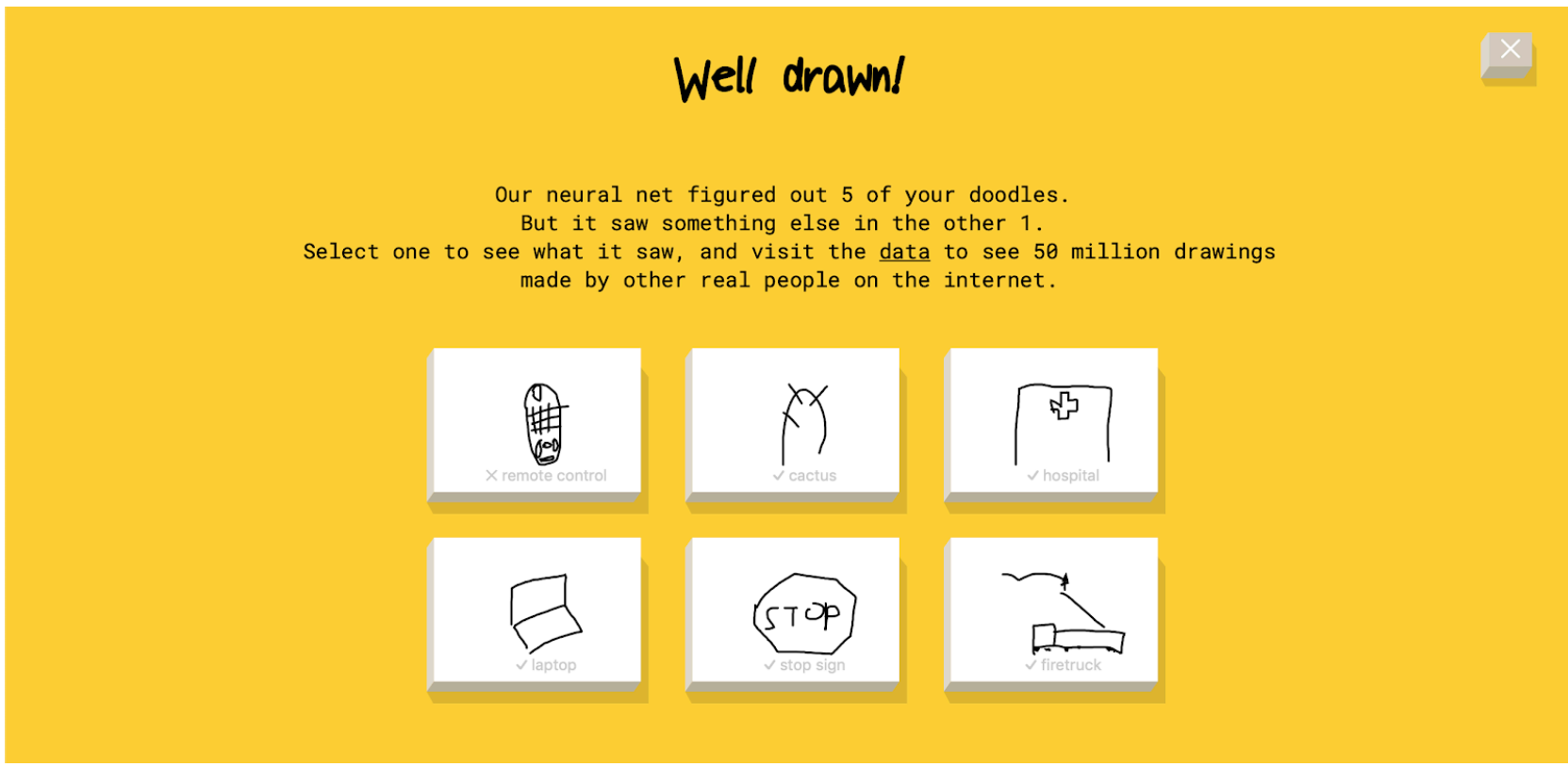
Three Application Based on Machine Learning

Among the different applications of machine learning there are few examples that are worthy of being mentioned for their relevance in image processing.
The first case is the neural network used by the Deep Art website.(Their algorithm is a variational autoencoder.) On this website users can upload images and choose which artistic style apply to them. For example you can take a selfie and then apply Van Gogh's style to it or even upload a different style from those that are already on the website.
The Deep Art Algorithm is able to replicate Van Gogh's style likely because its developers provided to it many Van Gogh's paintings. After a while the algorithm was able to recreate the artist's style accurately thanks to its neural network built on Van Gogh's paintings themselves(and also of other artists).



The second interesting example is the algorithm used by "Quick, Draw". "Quick Draw" is actually a game in which the player has to draw something like an object or an animal in twenty seconds and the algorithm tries to recognize what it is. After a few examples the game is finished and the player can see how many drawings were recognized by the algorithm. The player can also see what other players drew. The case of "Quick, Draw"

is particularly interesting because it is in constant development thanks to the game itself and a technique called “online learning”, which allows the algorithm to perform its task while continuing its training phase, while usually for machine learning algorithms once that the training phase is over there is no chance to modify the neural network again.



The third and final example is the Algorithmia algorithm, which is capable to color a black and white picture with realistic colors in few seconds, while for a human being using Photoshop a similar task would require hours. Even if in some cases its accuracy is not a hundred percent correct still in most cases its result are fine. Even in the worst case scenario still this algorithm provides a fine starting point in order to color in a proper manner a black and white picture.

Photo Colorization Before and After

Drag the purple line to reveal the before and after

The Code of a MLP

This code was created taking as a model the multilayer perceptron used by Sebastian Rashka in his book. While Rashka used his code to create a neural network able to recognize the handwritten digits contained in the MNIST dataset, this code was intended to recognize film shots using only a feedforward neural network, so there are not the convolutional and pooling layers. This code was developed without using the keras and tensorflow libraries, so the code is a lot longer compared to the CNN used for this study. Its efficiency is also lower, however it gives to the reader the chance, if he or she is interested, to look closer at the structure of a multilayer perceptron. Let's take a closer look.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import expit
import sys
import pickle
```

The first lines of the code, as usual, call some libraries in order to exploit their functions. Numpy is the library used to handle images in a byte format, matplotlib.pyplot is the library that allows the compiler to draw graphs, sys is the library that is capable of interaction with the operating system used by the computer, pickle is the library used the save the images as byte files and from scipy.special the logistic function[26] is imported, also known as the expit function.

```
class NeuralNetMPL(object):
    def __init__(self, n_output, n_features, n_hidden=3, l1=0.0, l2=0.0, epochs=5,
eta=0.001,
                 alpha=0.0, decrease_const=0.0, shuffle=True, minibatches=1,
random_state=None):
        np.random.seed(random_state)
        self.n_output=n_output
        self.n_features = n_features
        self.n_hidden = n_hidden
        self.w1, self.w2 = self._initialize_weights()
        self.l1 = l1
        self.l2 = l2
```

[26] The logistic function is defined as follows: expit(x)=1/(1+exp(-x))

```
        self.epochs = epochs
        self.eta = eta
        self.alpha = alpha
        self.decrease_const = decrease_const
        self.shuffle = shuffle
        self.minibatches = minibatches
```

The next step consists of creating a class for the neural networks that receives as parameters the number of nodes in the output layer, the number of nodes at the input layer, which corresponds to the number of features of the data, the number of nodes in the hidden layer, the parameters for regularization *l1* and *l2,* which are going to be explained later, the number of *epochs*, the learning rate *eta*, the *alpha* parameter, which is a parameter used for the weighs update, the decrease constant, which can be used for an adaptive learning in which the learning rate decreases over time in order to achieve a better convergence, shuffle is a boolean variable that, when its value is true shuffles the data before every epoch in order to avoid the algorithm to get stuck in cycles, the number of *minibatches* and the *random_state*, which is going to be explained later. Some of these variables have some values, that are going to be overwritten when a new neural network is declared unless the new neural network doesn't have the values of those variables declared. There are other few parameters that are initialised when a new neural network is declared which are the weights, that are initialised through the function _initialize_weights, and the np.random.seed, which is used to initialize a pseudo-random number generator which is going to be used later.

```
    def _initialize_weights(self):
        w1=np.random.uniform(-1.0, 1.0, size=self.n_hidden*(self.n_features+1))
        w1=w1.reshape(self.n_hidden, self.n_features+1)
        w2=np.random.uniform(-1.0, 1.0, size=self.n_output*(self.n_hidden+1))
        w2=w2.reshape(self.n_output, self.n_hidden+1)
        return w1, w2
```

This is the method used to initialize the weights. This neural network consists of one input layer, one hidden layer and one output layer, so there are two sets of weights, one that connects the input layer to the hidden layer and one that connects the hidden layer to the output layer. Their values are set between 1 and -1 using the pseudo-number generator. The size of the vectors containing the values of the weights is determined by the number of nodes in each layer connected though the weights. For instance if the output layers has 4 nodes and the hidden layer has 6 nodes the size of the vector

containing the weights is 24. The reshape method allows to place the weights in their place.
In order to better understand how a neural networks works from now on the explanation will follow the temporal sequence instead of the written sequence(RIMETTERE MEGLIO).

```
lab=open("Lab.pickle", "rb")
lab_train=pickle.load(lab)
lab.close()

img=open("Img.pickle", "rb")
img_train=pickle.load(img)
img.close()
nin=NeuralNetMPL(n_output=4,n_features=img_train.shape[1],n_hidden=64,l2=0.01,l1=
0.01,epochs=30,eta=0.001,alpha=0.001,decrease_const=0.0001,shuffle=True,
minibatches=30,random_state=1)

nin.fit(img_train, lab_train, print_progress=True)
```

The first two chunks of code open the files containing the images and their labels. The following chunk declares a new neural network called *nin* with all of its parameters.
The fit method is used to actually train the neural network. The only parameters that it receives are the images, which are stored as numerical values inside the numpy array img_train, and their labels, which are stored inside lab_train. The first thing that the fit method does is to create an empty array called *cost_.* Then, after creating copies of X and y (the arrays containing the images and the labels), it calls the method that encode the labels.

```
 def fit(self, X, y, print_progress=False):
        self.cost_=[]
        X_data, y_data=X.copy(), y.copy()
        y_enc=self.encode_labels(y, self.n_output)
```

The first thing that the fit method does is to create an empty array called *cost_.* Then, after creating copies of X and y (the arrays containing the images and the labels), it calls the method that encodes the labels.

```
def encode_labels(self, y,k):
```

```
        onehot=np.zeros((k, y.shape[0]))
        for idx, val in enumerate(y):
            onehot[val, idx]=1.0
        return onehot
```

The encode_labels method applies the one hot encoding to the labels. The one hot encoding is a process that converts categorical variables into a more ML friendly form. By doing so instead of having the classes represented as values their are presented in a different format. Let's say that before the one hot encoding the classes are represented as follows: full figure=0, half figure=1, half torso=2, close up=3. After the one hot encoding the new representation of the classes is: full figure=(1,0,0,0), half figure=(0,1,0,0), half torso=(0,0,1,0) and close up=(0,0,0,1). This is very helpful in order to avoid that the classes with a higher value end up with a more relevance from a numerical point of view. Also, if someone is using the mean squared error function as the loss function, the one hot encoding prevents misclassification by the neural network. The next thing that the fit method does is the creation of the $\Delta\hat{w}$, the weight update.

```
        delta_w1_prev = np.zeros(self.w1.shape)
        delta_w2_prev = np.zeros(self.w2.shape)
```

These arrays have the same shape of the arrays containing the actual weights and their values are equal to zero, for now at least.

```
        for i in range(self.epochs):
            self.eta/=(1 + self.decrease_const*i)
```

With this for cycle it begins the core of the fit method. The number of iterations depends on the number of epochs. At the beginning of every iteration the is update of the learning rate. The new value of the learning rate is given by the sum of the former value of the learning rate and the result of the multiplication between the decrease constant, which was declared when the neural network was initialised, and the number of the actual iteration.

```
            if print_progress:
                sys.stderr.write('\rEpoch: %d/%d' % (i+1, self.epochs))
                sys.stderr.flush()
```

This chunk of code tells the compiler to write on the screen at which epochs the simulation is.

```
            if self.shuffle:
                idx = np.random.permutation(y_data.shape[0])
                X_data, y_data = X_data[idx], y_data[idx]

            mini = np.array_split(range(y_data.shape[0]), self.minibatches)
```

The first chunk of code shuffles the data before the computation of the gradient,and the following weight update. The second chunk splits the data into minibatches. The mini variable is then used in the following for cycle. This implies that for every minibatch goes through the following operations.

```
            for idx in mini:
                # feedforward
                a1, z2, a2, z3, a3 = self.feedforward(X[idx], self.w1, self.w2)
                cost = self._get_cost(y_enc=y_enc[:, idx],output=a3,w1=self.w1,w2=self.w2)
                self.cost_.append(cost)
```

This chunk of code uses two methods in order to compute the cost value. In order to determine the cost is necessary to set the values for *a1, z2, a2, z3, a3.* Such thing is done through the feedforward method.

```
def feedforward(self, X, w1, w2):
        a1=self._add_bias_unit(X, how='column')
        z2=w1.dot(a1.T)
        a2=self.sigmoid(z2)
        a2=self._add_bias_unit(a2, how='row')
        z3=w2.dot(a2)
        a3=self.sigmoid(z3)

        return a1, z2, a2, z3, a3
```

This method receives as input the X array, which contains the data regarding the images and the values of the weights. Before seeing how every element that the feedforward method gives as output is computed, let's see what *a1, z2, a2, z3* and *a3* stand for. The a1, a2 and a3 arrays are the arrays containing the activation values of the nodes of the input layer, the hidden layer and the output layer. The parameter z was defined in chapter 2 of this document, and it was defined as follows: $z^L=w^L a^{(L-1)}+b^L$. The values of a1 are the values stored inside the X array, the one that contains the images

saved as a string of numbers, so it necessary to add only the bias unit through the *_add_bias_unit* method.

```
def _add_bias_unit(self, X, how='column'):
        if how=='column':
            X_new = np.ones((X.shape[0], X.shape[1]+1))
            X_new[:, 1:] = X
        elif how=='row':
            X_new=np.ones((X.shape[0]+1, X.shape[1]))
            X_new[1:, :] = X
        else:
            raise AttributeError('how must be row or column')
        return X_new
```

The *_add_bias_unit* method keeps the original values received as inputs and just adds a row or a column to the original data. Once a1 has its bias unit z2 is defined as the dot product between the weight array w1, which contains the values of the weights that are between the input layer and the input layer, and a1 transposed. The following lines of code set the values for a2 through the *sigmoid* method and then apply the *_add_bias_unit* method also to a2.

```
def sigmoid(self, z):
            # expit is equivalent to 1.0/(1.0 + np.exp(-z))
        return expit(z)
```

The sigmoid method just put what it receives as an input inside the sigmoid, or expit, function. The remaining elements are computed in the same way as the ones seen so far. Let's now go back to the fit method.

```
                # compute gradient via backpropagation
            grad1, grad2 = self._get_gradient(a1=a1, a2=a2,a3=a3,
z2=z2,y_enc=y_enc[:, idx], w1=self.w1,w2=self.w2)
```

The next step of the fit method is to compute the gradient through the *_get_gradient* method.

```
    def _get_gradient(self, a1, a2, a3, z2, y_enc, w1, w2):
            # backpropagation
        sigma3 = a3 - y_enc
        z2 = self._add_bias_unit(z2, how='row')
```

```
        sigma2 = w2.T.dot(sigma3) * self.sigmoid_gradient(z2)
        sigma2 = sigma2[1:, :]
        grad1 = sigma2.dot(a1)
        grad2 = sigma3.dot(a2.T)

            # regularize
        grad1[:, 1:] += (w1[:, 1:] * (self.l1 + self.l2))
        grad2[:, 1:] += (w2[:, 1:] * (self.l1 + self.l2))

        return grad1, grad2
```

The _get_gradient method receives as input the arrays containing the activation values of the different layers (a1, a2 and a3), the z2 element, the labels encoded with the one hot encoding (y_enc), and the arrays containing the values of the weights (w1and w2). Then it defines the values of the array sigma3 as the difference between the values stored in the array a2, so the activation values of the hidden layer, and the values stored in the encoded array that contains the values of the classes. In these lines of codes is where the backpropagation is applied (if the reader doesn't remember how the backpropagation works, inside chapter 2 there is the full explanation).
Before giving the gradients as output is necessary to regularize their values through the l1 and l2 parameters, that were defined when the new neural network was declared. Before going on, let's take a quick look at the sigmoid_gradient method that is called inside the _get_gradient method.

```
def sigmoid_gradient(self, z):
        sg=self.sigmoid(z)
        return sg*(1-sg)
```

In order to compute the sigmoid gradient the sigmoid_gradient method recalls the sigmoid method in order to have the sigmoid function. After storing the sigmoid function inside a variable, it multiplies that variable, that is called sg, for(1-sg), so 1 minus the variable itself, and gives the result as output.
The last lines of the fit method compute the weights update.

```
                # update weights
            delta_w1, delta_w2 = self.eta * grad1,self.eta * grad2
            self.w1 -= (delta_w1 + (self.alpha * delta_w1_prev))
            self.w2 -= (delta_w2 + (self.alpha * delta_w2_prev))
            delta_w1_prev, delta_w2_prev = delta_w1, delta_w2
```

```
        return self
```

The parameter alpha was declared when the new neural network was created. With these last lines of code the training phase of the neural network is over. The following step is to check how the neural networks performs with images that it has never seen before. These images are stored in what is called the validation set. Before taking a look at the *predict* method it necessary to spend a few words on the validation set. The validation set must contain images that are saved in the same way as the training set, so they must have the same dimensions and they need to be stored as strings of numbers otherwise the neural network won't be able to process them.
The first thing to do with the validation set is to feed it to the neural network and see how it performs.

```
y_train_pred = nin.predict(val_img_train)
```

The predict method is built as follows.

```
 def predict(self, X):
        a1, z2, a2, z3, a3=self.feedforward(X, self.w1, self.w2)
        y_pred=np.argmax(z3, axis=0)
        return y_pred
```

The predict method calls once again the feedforward method in order to compute a1,a2,a3,z2 and z3. Then it uses the argmax function with the values stored inside z3 in order to make its predictions regarding the new unlabelled data received.
When the neural network has finished to classify the images inside the validation set it compares its result with the actual labels of the images.

```
acc = np.sum(lab_train == y_train_pred, axis=0) / img_train.shape[0]
print('Training accuracy: %.2f%%' % (acc * 100))
```

This concludes this part of the Appendix.

How to Use the CNN after the Training Phase

Once the CNN is trained, and after that its model has been saved, is time to see how it performs with new images[27]. This code loads the model trained and all of its parameters and uses it to classify images. Its performance will be the same one reached with the validation set.

```
import cv2
import tensorflow as tf
import os
CATEGORIES=["figura intera", "mezza figura","mezzo busto", "primo piano"]
```

This first chunk of code imports the libraries necessary to process images and to interact with the operating system. *CATEGORIES* is just a list with the class names. It will be used later.

```
path='/Users/bartolomeovacchetti/Desktop/Img No Label'

def prepare(filepath):

    new_img=cv2.imread(filepath, cv2.IMREAD_GRAYSCALE)
    new_img=cv2.resize(new_img, (160,90))

    return new_img.reshape(-1, 160, 90,1)

model=tf.keras.models.load_model('CNN_Img_Clss_new.model')
```

The first line of this chunk just declares a variable containing the path to the folder in which there are the images that the CNN has to classify.

The function *prepare* receives as input the path to every single image and then desaturate and rescale every single image. Without this function it wouldn't be possible for the CNN to analyse the new images because they would have a different number of features compared to the ones on which the CNN has been trained on.

---

[27] These images are not part of the validation set, because after classification the network has no way to know if its classification is right or not.

The last line of code loads the model of the CNN saved previously.

```
for img in os.listdir(path):
    try:
        link=os.path.join(path, img)
        img_array=cv2.imread(os.path.join(path,img), cv2.IMREAD_GRAYSCALE)
        img_array=cv2.resize(img_array, (160, 90))
        prediction=model.predict_classes([prepare(link)])
        print(prediction)
        print(CATEGORIES[int(prediction)])
    except Exception as e:
        print("errore")
        pass
```

This last chunk combines the path of the folder with the counter *img.* By doing so the code creates a link for every single image. Such link is passed as input to the function *prepare* that returns as output the image reshaped in order to fit into the CNN model through *predict_classes,* which predicts the class of the image by using the CNN. The value that *predict_classes* gives as output is saved inside the variable *prediction*.
The two *print* functions show the predicted class of the image. The difference between the two is that the first one print just the number of the output corresponding to the predicted class, while the second one uses such number as an index inside the *CATEGORIES* list and print the corresponding name of the class based on the value of the *prediction* variable.

SITOGRAPHY

http://www.intelligenzaartificiale.it/intelligenza-artificiale-forte-e-debole/

https://it.wikipedia.org/wiki/Apprendimento_automatico

https://deepmind.com/

https://it.wikipedia.org/wiki/Go_(gioco)

https://medium.com/@jonathan_hui/gan-super-resolution-gan-srgan-b471da7270ec

https://it.wikipedia.org/wiki/Analisi_della_regressione

https://www.youtube.com/watch?v=m-S9Hojj1as

https://www.youtube.com/watch?v=IEfrr0Yr684

https://towardsdatascience.com/ensemble-methods-bagging-boosting-and-stacking-c9214a10a205

https://towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce

https://en.wikipedia.org/wiki/Semi-supervised_learning

https://en.wikipedia.org/wiki/Generative_adversarial_network

https://skymind.ai/wiki/generative-adversarial-network-gan

https://it.wikipedia.org/wiki/TensorFlow

https://keras.io

https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy

https://www.youtube.com/watch?v=tIeHLnjs5U8

https://www.youtube.com/watch?v=vMh0zPT0tLI

https://machinelearningmastery.com/build-multi-layer-perceptron-neural-network-models-keras/

https://film-grab.com

https://www.youtube.com/watch?v=YRhxdVk_sIs

https://www.youtube.com/watch?v=FmpDIaiMIeA

https://www.youtube.com/watch?v=ZjM_XQa5s6s

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html

BIBLIOGRAPHY

Raschka S., *Python Machine Learning. Unlock deeper insights into machine learning with this vital guide to cutting-edge predictive analytics*., Birmingham-Mumbai, Packt Publishing, 2015.

Alonge G., *Il Cinema. Tecnica e Linguaggio. Un'introduzione.*, Kaplan, Edizioni Kaplan, 2011.

Chollet F., *Deep Learning with Python.* Manning Publications, 2017.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, Demis Hassabis**.,** *A general reinforcement learning algorithm that masters chess, shogi and Go through self-play,* Science, 2018.

SPECIAL THANKS

The first thanks goes to all of my friends and my family, relatives by blood and not. That said, there are some people that deserve a special mention, not only for the support that they gave me during the drafting of this thesis, but for their support and help through my university career. I hope I don't forget any of them. The first[28] thanks goes to my mother and my father for maintaining me during the studies and for their constant support and guidance. Another thanks goes to Riccardo Zecchina for making me passionate about ML and for teaching me the basics of it. I want to thank my siblings, Francesca, Sara and Arturo for lightning me up whenever I was down during these years.

Next on the list there are my grandparents: Elda, Rita and Arturo. They made these years unique with their stories and debates. I also want to thank them for the endless stream of food that they provided to me during these years during lunches and dinners and for being able to always create an oasis of peace from the frenzy of our society.

The next thanks goes to the Professor Riccardo Antonino, who mentored me for the last few years, from an academic point of view to the working point of view. I also want to thank Professor Tania Cerquitelli for the help received and her insights on ML.

I want to thank all of my cousins, especially Lorenzo, for his capacity to take action and interest in almost everything, and Filippo, which I believe is the most patient and kind person I know, two increasingly rare qualities.

Another thanks goes to Riccardo Livermore, Davide Giacosa, Chiara Attanasio and Monica Schettino, my old highschool friends with whom I shared more than one laugh. I also want to thank again Monica, since a part of the dataset used consists of her photographies.

Gustavo, my ex-acquired cousin, and his girlfriend Chiara are next on the list. They have always been there for me, whether it was a question of having a beer and a chat or more serious issues.

I want also to thank Familia Samir, which consists of Lello, Samiro, la Shoppina and Pliz. These years wouldn't have been as fun as they were if it weren't for you. I want specifically to thank Lello for always working with me and for being a constant reference from a practical point of view to a more philosophical one.

Another thanks goes to Pietro Lanzillotti, a.k.a. The Dude, for his open mindedness, his honesty and his, sometimes severe, but proper spurs.

Last, but definitely not least, I want to thank Sarah Roati. She, more than anyone, supported me in these last years and she is one of the most altruistic people that I know, capable of putting others interests before her own.

---

[28] The order of the special thanks is not related to the contributions made for this thesis but follows the following order, close family, professors and friends.